ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA - SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA INFORMATICA E DELLE TELECOMUNICAZIONI

VIRTUALIZZAZIONE DI FUNZIONI DI RETE SU PIATTAFORME PER CLOUD COMPUTING

Elaborato in

Applicazioni e Tecniche di Telecomunicazioni

Relatore
Prof. Ing. WALTER CERRONI

Presentato da
LUIGI PONTI

Correlatore
Ing. CHIARA CONTOLI

SESSIONE II

ANNO ACCADEMICO 2014/2015

Alla mia famiglia
per il costante supporto ricevuto

Indice

Sommario

Questo documento affronta le novità ed i vantaggi introdotti nel mondo delle reti di telecomunicazioni dai paradigmi di Software Defined Networking e Network Functions Virtualization, affrontandone prima gli aspetti teorici, per poi applicarne i concetti nella pratica, tramite casi di studio gradualmente più complessi.

Tali innovazioni rappresentano un'evoluzione dell'architettura delle reti predisposte alla presenza di più utenti connessi alle risorse da esse offerte, trovando quindi applicazione soprattutto nell'emergente ambiente di Cloud Computing e realizzando in questo modo reti altamente dinamiche e programmabili, tramite la virtualizzazione dei servizi di rete richiesti per l'ottimizzazione dell'utilizzo di risorse.

Motivo di tale lavoro è la ricerca di soluzioni ai problemi di staticità e dipendenza, dai fornitori dei nodi intermedi, della rete Internet, i maggiori ostacoli per lo sviluppo delle architetture Cloud.

L'obiettivo principale dello studio presentato in questo documento è quello di valutare l'effettiva convenienza dell'applicazione di tali paradigmi nella creazione di reti, controllando in questo modo che le promesse di aumento di autonomia e dinamismo vengano rispettate.

Tale scopo viene perseguito attraverso l'implementazione di entrambi i paradigmi SDN e NFV nelle sperimentazioni effettuate sulle reti di livello L2 ed L3 del modello OSI.

Il risultato ottenuto da tali casi di studio è infine un'interessante conferma dei vantaggi presentati durante lo studio teorico delle innovazioni in analisi, rendendo esse una possibile soluzione futura alle problematiche attuali delle reti.

Capitolo 1

Introduzione

Il mondo delle telecomunicazioni, negli ultimi cinquant'anni, è stato rivoluzionato dalla nascita e sviluppo di un paradigma totalmente innovativo, il *Cloud Computing*. Alla base di tale concetto vi è l'erogazione di risorse informatiche, come l'archiviazione, l'elaborazione o la trasmissione di dati, caratterizzato dall'utilizzo, attraverso Internet e perciò principalmente a distanza, di risorse preesistenti e configurabili che sono assegnate, rapidamente e convenientemente, grazie a procedure automatizzate, a partire da un insieme di risorse, e quindi reti, condivise con altri utenti. Quando l'utente rilascia la risorsa, essa viene similmente riconfigurata nello stato iniziale e rimessa a disposizione nel pool condiviso delle risorse, con altrettanta velocità ed economia per il fornitore. [1]

Grazie alla sua praticità, questa innovazione ha riscosso sempre più successo diventando ormai parte della quotidianità di tutti gli utenti della rete, ma provocando di conseguenza un aumento esponenziale dei dispositivi presenti sulla rete, della richiesta di servizi e perciò del traffico da essi generato. Tale crescita ha dato luce alla necessità di un'evoluzione del sistema circostante, quindi all'introduzione di nuove tecnologie ed alla nascita di concetti che rivoluzionassero la struttura tradizionale della rete, ormai statica e dipendente dai fornitori dei nodi intermedi della rete stessa.

In questi nodi, chiamati *middle-boxes*, vengono implementate alcune funzioni quali firewall e NAT, ma il problema che li affligge è la loro natura di dispositivi proprietari, che priva dell'opportunità di fornire nuovi servizi di rete senza costi indifferenti e senza difficoltà nell'integrazione con altri componenti già presenti sulla rete stessa, rendendo così lento e totalmente dipendente dal

venditore il processo di rinnovo. Questa situazione, che ha rallentato notevolmente lo sviluppo ed il progresso della rete, ha quindi richiesto un nuovo approccio, per poter superare tali ostacoli.

Le due maggiori innovazioni che contribuiscono a questo radicale cambiamento sono il Software-Defined Networking ed il Network Functions Virtualization.

Il Software-Defined Networking (SDN) è un approccio al computer networking che permette agli amministratori di rete di separare il piano di controllo del traffico dal piano di reindirizzamento dello stesso, dando perciò la possibilità di avere una gestione esterna, ma soprattutto dinamica della rete, questo tramite il protocollo OpenFlow, illustrato nel Capitolo 2.

Oltre alla separazione appena illustrata, per acquisire l'indipendenza dal fornitore delle middle-boxes, permettendo quindi lo sblocco della staticità dei dispositivi stessi, è necessario mantenere la presenza dei servizi di rete già disponibili fino ad ora, oltre ad eventuali nuove funzioni aggiuntive.

Risulta quindi fondamentale la presenza del Network Functions Virtualization (NFV), concetto di architettura di rete in cui è previsto che le funzionalità di rete siano realizzate virtualmente tramite applicazioni software chiamate *VNF*, istanziabili direttamente sui server, creando più *Virtual Machine* sulla stessa macchina fisica. In questo modo si è raggiunta l'indipendenza del software dall'hardware sottostante, le cui specificità vengono mascherate dal sistema di virtualizzazione. [2]

Questo documento affronta le tematiche della gestione della rete e degli utenti in essa presenti, puntando ad ottenere autonomia e dinamismo nel controllo dei flussi di dati, col massimo rendimento delle risorse disponibili. L'obiettivo di questo studio è il progetto di un orchestratore capace di affrontare ogni condizione di traffico nella rete considerata, distinguendo anche due tipologie di utenti, con differenti priorità, e quindi fornendo dinamicamente un servizio che rispetti le qualità previste dai contratti sottoscritti dagli utenti stessi.

Si è perciò analizzata una situazione di volta in volta più simile al caso reale, nelle quali due utenti con diverse priorità usufruiscono di un accesso alla rete Internet, applicandovi le novità ed i vantaggi caratteristici dei concetti SDN e NFV precedentemente introdotti.

Il Capitolo 2 tratta in modo più specifico ed approfondito i paradigmi SDN e NFV, illustrandone le applicazioni esistenti, ovvero il protocollo OpenFlow

ed il software OpenStack, e fornendo le basi teoriche per meglio comprendere lo studio effettuato in questo documento.

Nel Capitolo 3 viene introdotto ed analizzato il controller SDN Ryu, che è il software scelto per realizzare l'orchestratore della rete, obiettivo principale di questo studio. In tale capitolo, oltre agli accenni teorici sul controller, sono affrontati alcuni esempi di applicazioni, grazie alle quali è possibile avere il primo approccio pratico col software.

Di seguito, nel Capitolo 4 viene presentato il concetto di Dynamic Service Chaining, illustrandone le basi e, soprattutto, gli aspetti operativi, introducendo quindi i casi di studio L2 ed L3 successivamente affrontati in modo più pratico nel Capitolo 5.

Infine, il capitolo 6 tratta l'implementazione di un controller Ryu generalizzato, prendendo spunto dai casi di studio del Capitolo 5 e applicando strutture e concetti che permettano di ottenere ancora maggiore autonomia e dinamismo, rendendo tale applicazione non più dedicata ad una specifica rete singola, ma indipendente dalla quantità di utenti ad essa connessi, senza dover procedere ad alcuna modifica del codice.

Al termine del documento, sono inoltre presenti, nell'Appendice A, tutti i codici utilizzati negli esempi e nei casi pratici affrontati nel corso di questo studio.

Capitolo 2

Introduzione a SDN e NFV

Come accennato nel Capitolo 1, la nascita di queste due nuove modalità di approccio dell'architettura di rete hanno dato la possibilità di procedere verso un risveglio dalla situazione di staticità attuale ed un progresso potenziale che possa andare incontro ed unirsi al paradigma del *Cloud Computing*.

È perciò di facile intuizione l'importanza di tali innovazioni e l'attenzione che la loro nascita ha suscitato nell'attuale mondo delle telecomunicazioni.

Vengono quindi analizzati in questo capitolo tali concetti per semplificarne la comprensione ed acquisire le basi per affrontare al meglio lo studio svolto in questo documento.

2.1 Software-Defined Networking (SDN)

Il Software Defined Networking (SDN) è una proposta di organizzazione dell'architettura di rete che permette la gestione dei servizi della rete stessa, attraverso l'astrazione di funzionalità a basso livello. Questo grazie al disaccoppiamento del sistema che prende le decisioni relative all'invio del traffico, lo strato di controllo o *control plane*, dai sistemi, ad esso sottostanti, che inoltrano il traffico alla destinazione selezionata, lo strato di reindirizzamento o *data plane*. [3] L'architettura presentata dal SDN è espressa dalla Figura 2.1.

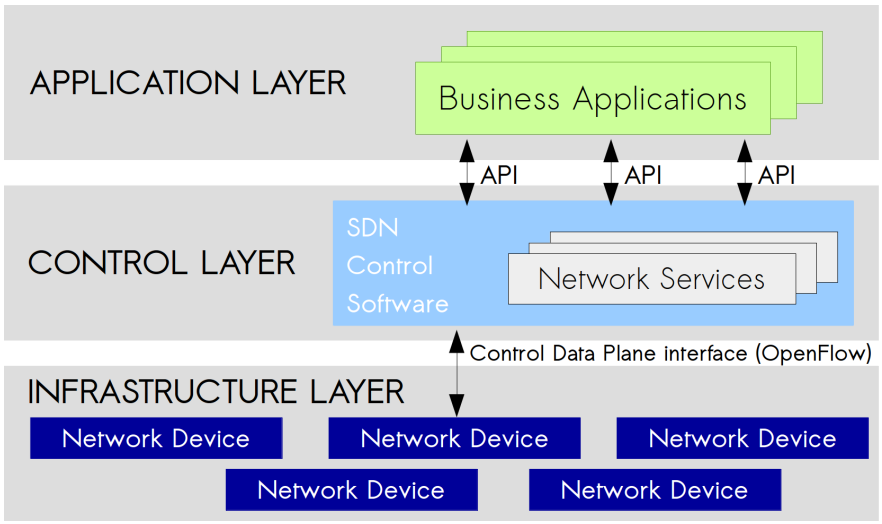

Figura 2.1: Architettura SDN.

Viene in questo modo centralizzata localmente l'intelligence e la gestione dello stato della rete, separando quindi l'infrastruttura della rete stessa dalle applicazioni. I principali benefici offerti sono:

- Controllo centralizzato dei dispositivi di rete;
- Automazione e gestione delle reti migliorata;
- Maggiore possibilità di implementazione di nuovi servizi;
- Programmabilità semplificata per gli operatori;
- Maggiore affidabilità;
- Esperienza lato utente migliorata ed indipendenza dal produttore.

Il risultato di quanto introdotto è quindi un guadagno in programmabilità, automazione e controllo di rete, permettendo la creazione di reti altamente modulari e flessibili che possano adattarsi velocemente e, soprattutto, dinamicamente alle necessità dell'utente.

In tal modo, le funzionalità di controllo, finora strettamente legate a dispositivi di rete implementati secondo un approccio "monolitico, migrano a tendere verso uno strato di controller SDN o "sistema operativo di rete" (Network OS) separato. [2]

Tale approccio richiede però la necessità di far comunicare il control plane col data plane. Ciò è reso possibile dal protocollo OpenFlow, che è la prima interfaccia standard realizzata specificatamente per il sistema SDN, meglio descritta nella sezione seguente.

2.1.1 OpenFlow

Gestito dalla Open Networking Foundation (ONF), OpenFlow è un protocollo di comunicazione che prevede la presenza di uno switch o un router e di un controller, esterno ad esso, che abbia la possibilità di accedere al forwarding plane del dispositivo per modificare regole e determinare il percorso dei pacchetti attraverso una rete di switch in modo dinamico.

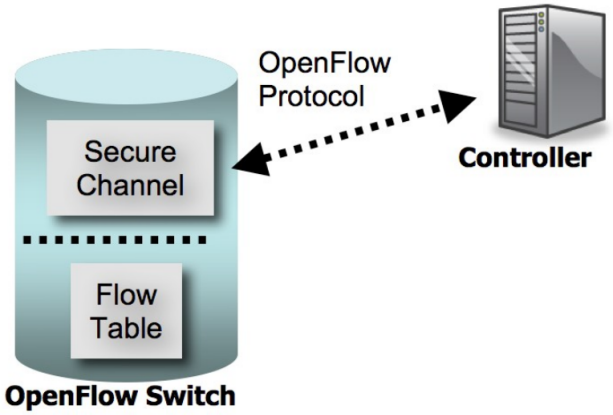

Figura 2.2: Protocollo OpenFlow. [4]

Uno switch OpenFlow consiste in una *flow table*, che realizza il lookup, l'inoltro ed eventualmente la modifica dei pacchetti, e un'interfaccia chiamata *secure channel* attraverso cui il controller configura e dirige lo switch, ricevendo da esso eventi e trasmettendo, tramite esso, pacchetti nella rete. La flow table contiene una lista di *flow entries*, ovvero di regole, alle quali corrispondono determinate azioni da svolgere sui pacchetti, che vengono consultate ogni volta

che lo switch processa un pacchetto. Nel caso un pacchetto non trovi alcun riscontro nella tabella (*Table-miss*), esso viene inoltrato, attraverso il secure channel, al controller che prenderà decisioni su come gestire tali pacchetti senza una valida flow entry, modificando la flow table dello switch, quindi aggiungendo o rimuovendo regole. [4]

Una flow entry è composta da:

- **header**, intestazione da confrontare col pacchetto, ad esempio numeri di porta dello switch o della sorgente/destinazione, indirizzi MAC o IP, protocolli (ARP, ICMP, TCP, UDP, IPv4, IPv6, ...), VLAN ID, ... ;
- **counter**, contatori di attività;
- **actions**, azioni da applicare ai pacchetti che corrispondono alla flow entry, come l'inoltro, il drop (sia verso porte fisiche, che virtuali) e/o la modifica dell'header dei pacchetti.

Attraverso tali regole è possibile avere il pieno controllo dello switch, e quindi del traffico nella rete, anche in modo dinamico, programmando adeguatamente il controller, il cui compito è quindi la gestione dello switch e dei pacchetti senza una valida flow entry, agendo sulle regole presenti nella flow table. La scelta del controller è totalmente libera, i principali sono: POX (scritto in Python), OpenDaylight (Java), Ryu (Python), Floodlight (Java) e Beacon (Java).

2.2 Network Functions Virtualization (NFV)

Introdotto nell'ottobre del 2012 al "SDN and OpenFlow World Congress", il Network Functions Virtualization (NFV) è un paradigma che punta a trasformare il modo in cui sono architetturate le reti, sfruttando la tecnologia di virtualizzazione, ora in piena evoluzione, per concentrare diverse tipologie di dispositivi in hardware generico, come server, nodi di rete ed in prossimità dell'utente finale, come illustrato dalla Figura 2.3. Ciò richiede quindi l'implementazione di funzioni di rete tramite software che possa essere utilizzato su un vasto range di server fisici standard e che possa essere mosso, o istanziato, in differenti locazioni nella rete secondo le necessità, senza l'installazione di nuovi dispositivi.

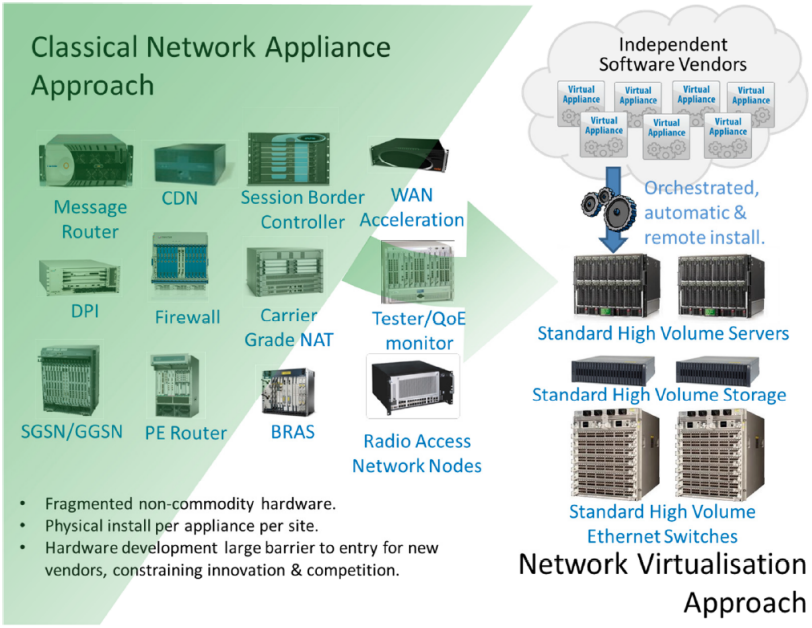

Figura 2.3: Network Functions Virtualisation. [5]

Con questo approccio si riuscirebbero a superare i problemi relativi alle middle-boxes, ovvero i dispositivi hardware proprietari. La loro presenza nelle reti infatti, rende difficile lanciare nuovi servizi e funzionalità di rete a causa dei notevoli costi da affrontare per l'aggiornamento e l'integrazione dei dispositivi già presenti. Oltre a tali problematiche, la natura di sistema basato sull'hardware, per poter restare al passo col progresso tecnologico attuale, è caratterizzato dalla continua necessità di rinnovo, che ne riducono, o eliminano, ogni profitto.

Il paradigma NFV propone quindi di sostituire tali middle-boxes proprietarie, causa principale dell'ossificazione del sistema, con dispositivi hardware general-purpose, meno costosi, con la possibilità, grazie alla virtualizzazione, di realizzare middle-boxes riprogrammabili dinamicamente. Questa evoluzione del sistema, comporterebbe i seguenti benefici:

- Riduzione dei costi e del consumo di energia, raggruppando più dispositivi in uno unico, grazie alla virtualizzazione

- Aumento della velocità del *Time To Market*, ovvero diminuzione del tempo che intercorre dall'ideazione alla commercializzazione di funzionalità e servizi, minimizzando il ciclo di innovazione della rete, in quanto è sufficiente agire sul software per effettuare aggiornamenti e migliorie;

- Possibilità di svolgere test direttamente sulla stessa infrastruttura fornisce risultati più efficienti ed una maggiore integrazione, riducendo i costi di sviluppo ed il *Time To Market'*;

- Possibilità di agire da remoto sui dispositivi della rete, aumentando la velocità dell'assistenza;

- Maggiore apertura del sistema, dovuta alla libertà del software, utilizzabile da qualsiasi utente, incoraggiando quindi una maggiore innovazione che porti nuovi servizi con un rischio minore;

- Migliore configurazione e topologia delle reti in tempo reale, basata sullo stato attuale della rete, riguardo al traffico ed ai servizi forniti, agendo in modo dinamico;

- Supporto al *multi-tenancy*, ovvero la possibilità di fornire, in contemporanea a più utenti, connettività, servizi ed applicazioni configurate su misura sullo stesso hardware, ma separandone i domini per mantenerne la sicurezza;

- Riduzione del consumo di energia sfruttando funzionalità di *power management* e risparmio energetico nei server standard, attuando un controllo più efficiente del carico di lavoro;

- Aumento dell'efficienza dei servizi, grazie alla possibilità di agire con facilità sui sistemi già presenti, eliminando la necessità di hardware specifico, riparando eventuali malfunzionamenti in modo automatico, via software, spostando anche temporaneamente il carico di lavoro nella rete e favorendo il rinnovo della rete, essendo sufficiente effettuare aggiornamenti del software.

2.2.1 OpenStack

Originariamente promosso da Rackspace e NASA e con oltre 200 società che si sono unite al progetto tra cui AT&T, AMD, Cisco, Dell, EMC, Ericsson, F5, HP, IBM, Intel, NEC, NetApp, Red Hat, SUSE, VMware, Oracle e Yahoo [2], OpenStack è un software di cloud computing gratuito ed open-source che permette di gestire piattaforme cloud, cioè cluster di macchine fisiche che ospitano alcuni server, che sono offerti all'utente come un servizio (*IaaS: Infrastructure as a Service*). L'utente può quindi utilizzare tali server per creare un'infrastruttura virtuale, composta da server e applicazioni di rete, come Firewall, NAT, eccetera, implementando il tutto tramite Virtual Machines.

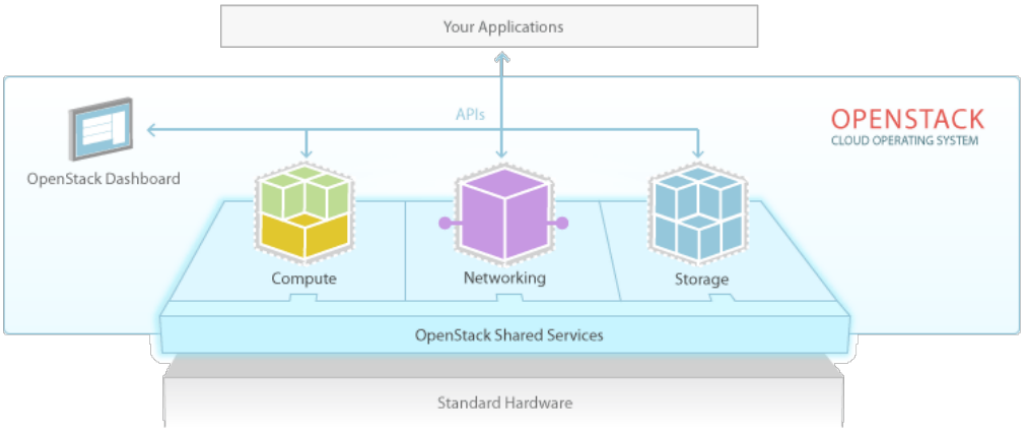

Figura 2.4: Struttura di OpenStack. [6]

Tale gestione è effettuabile tramite una dashboard grafica, in maniera più trasparente e semplificata, oppure mediante terminale, con maggiori possibilità di controllo e perciò per gli utenti più esperti, grazie alle API provviste.

Componenti di OpenStack

I componenti di OpenStack sono:

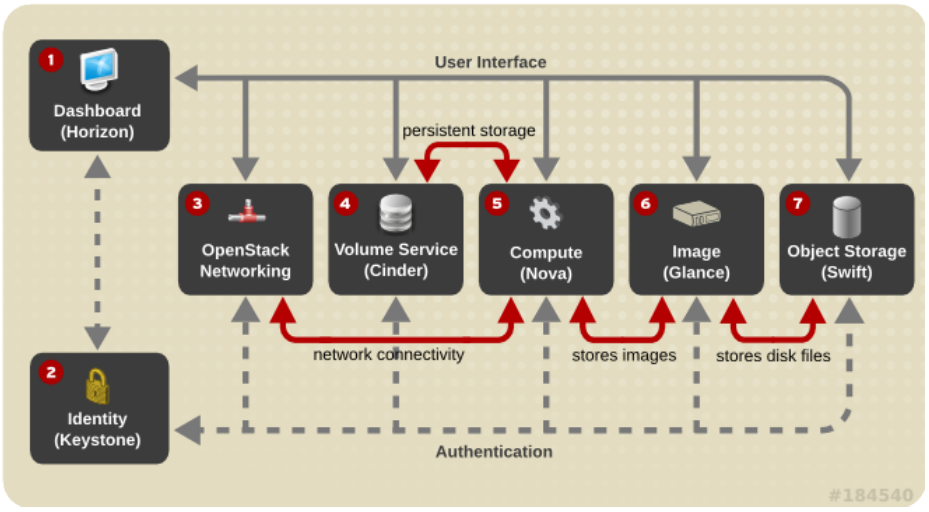

Figura 2.5: Componenti di OpenStack (da Havana in poi). [9]

- **Nova**, controller per il cloud computing, scritto in Python e progettato per gestire e automatizzare il pool di risorse del computer e può funzionare con tecnologie di virtualizzazione ampiamente utilizzate, come pure in configurazioni bare-metal e high-performance;

- **Neutron**, (in precedenza *Quantum*) è un sistema per la gestione delle reti e degli indirizzi IP. Neutron assicura che la rete non sarà il collo di bottiglia o fattore limitante in un Cloud e offre agli utenti una reale gestione self-service anche delle loro configurazioni di rete. Neutron inoltre fornisce differenti modelli di rete per le diverse applicazioni o i gruppi utente. I modelli standard includono flat networks o VLAN per la separazione del traffico. Gestisce quindi gli indirizzi IP, consentendo l'assegnazione di indirizzi IP statici dedicati oppure tramite DHCP;

- **Keystone**, gestore delle autenticazioni, fornisce un elenco degli utenti indicando i servizi di OpenStack ai quali essi possono accedere. Si comporta come un comune sistema di autenticazione all'interno del sistema

Cloud, è di facile integrazione e supporta svariate forme di autenticazione incluse le credenziali standard username e password. Utenti e strumenti esterni possono determinare a quali risorse possono accedere;

- **Glance**, gestore delle immagini, si occupa dei servizi di scoperta, registrazione e consegna per le immagini dei server. Le immagini immagazzinate possono essere anche utilizzate come modello. Glance può inoltre conservare e catalogare un numero illimitato di backup;

- **Cinder**, storage persistente a livello di dispositivi a blocchi per il loro utilizzo da parte delle istanze di Nova. Il sistema di storage a blocchi gestisce la creazione, il collegamento e lo scollegamento dei dispositivi a blocchi ai server;

- **Swift**, sistema di storage scalabile e ridondante. Gli oggetti e i files sono memorizzati su diversi dischi distribuiti su diversi server nel centro di calcolo. Swift si fa carico del compito di assicurare la replicazione e l'integrità dei dati all'interno del cluster;

- **Horizon**, è la Dashboard e fornisce una interfaccia grafica, sia per gli amministratori che per gli utenti, per l'accesso e la gestione delle risorse fornita dal Cloud. [8]

Cluster OpenStack

Un cluster OpenStack è composto da:

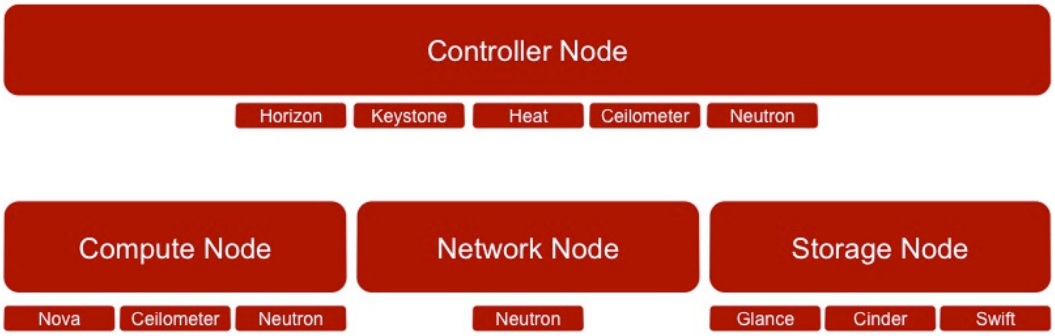

Figura 2.6: Struttura di un cluster OpenStack. [10]

- un *Controller node*, che gestisce la piattaforma cloud;
- un *Network node*, che ospita i servizi di rete cloud;
- un numero di *Compute nodes*, che eseguono le Virtual Machines;
- un numero di *Storage nodes*, per le immagini delle VM e i dati. [7]

Tali nodi sono connessi tra loro mediante due reti, una di Management ed una di Data. Il Controller node ed il Network node sono connessi alla rete esterna. La rete dati è utilizzata per la comunicazione tra VM, mentre quella di Management permette all'amministratore di accedere ai nodi del cluster ed è sfruttata per le comunicazioni di servizio. La rete esterna, infine, permette alle VM di accedere ad internet, così come agli utenti di accedere alle macchine stesse.

Multi-tenancy

Il supporto al *multi-tenancy*, fornito da OpenStack, permette la possibilità di avere contemporaneamente più utenti, denominati *tenant*, nello stesso cluster. Le sessioni relative ad utenti differenti devono essere senza dubbio isolate tra loro ed a tal proposito sono utilizzate VLAN e *network namespaces*. I network

namespaces sono contenitori isolati in cui è possibile configurare reti non visibili dall'esterno del namespace. Un network namespace può essere utilizzato per contenere una specifica funzionalità di rete o per fornire un servizio di rete isolato, aiutando ad organizzare una complessa installazione di rete che permetta perciò la presenza di più utenti, senza rinunciare a sicurezza e privacy degli stessi. I network namespaces permettono infatti di separare e isolare più domini di rete all'interno di un singolo host semplicemente replicando lo stack software della rete. Un processo eseguito all'interno di un namespace vede solo specifiche interfacce, tabelle e regole. I namespaces garantiscono l'isolamento L3: in questo modo le interfacce relative a tenant diversi possono avere indirizzi IP sovrapposti.

Per proteggere invece le macchine virtuali da attacchi esterni o accessi non autorizzati si ricorre invece ai *security groups* Un security group è un insieme di regole che implementano un firewall configurabile dall'utente: durante la creazione delle Virtual Machines si pos- sono associare a queste ultime determinati security groups, implementati mediante regole iptables sul compute node. [9]

Capitolo 3

Controller SDN: RYU

In questo capitolo viene descritto il controller Ryu, utilizzato nei casi di studio del Capitolo 5, sottolineandone gli aspetti operativi e mostrando infine qualche esempio di utilizzo.

3.1 Introduzione al controller

Figura 3.1: Logo del controller Ryu. [11]

Ryu, che in giapponese significa sia drago, che flusso, è un software SDN open-source, implementato interamente in Python, utilizzato come controller nei sistemi SDN. Come gli altri controller SDN, Ryu fornisce componenti software con API ben definite che permettono agli sviluppatori di creare nuove applicazioni di gestione e controllo della rete. Uno dei punti di forza di Ryu è la

sua capacità di supportare svariati protocolli di southbound per la gestione dei dispositivi, ad esempio Openflow, Netconf, OF-config, eccetera. [12]

3.2 Architettura

Come ogni controller SDN, Ryu può creare ed inviare dei messaggi OpenFlow, rilevare eventi asincroni, tra i quali la rimozione di un flusso, e gestire ed analizzare pacchetti in arrivo al controller.

La Figura 3.2 sottostante rappresenta l'architettura del framework del controller Ryu:

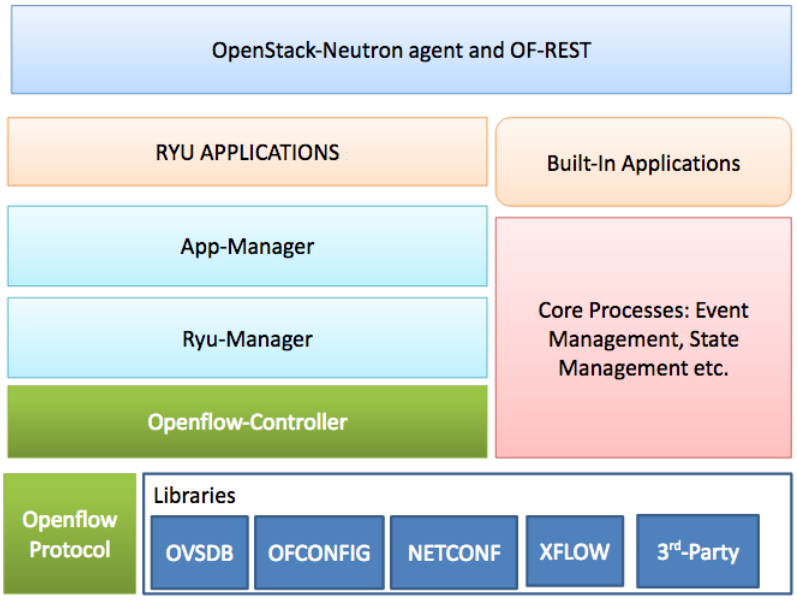

Figura 3.2: Architettura di Ryu. [12]

3.2.1 Librerie

Ryu dispone di un'impressionante quantità di librerie, spaziando dal supporto verso più protocolli di southbound, a numerose operazioni per processare i pacchetti di rete.

Per quanto riguarda i protocolli southbound, Ryu supporta OF-Config, Open vSwitch Database Management Protocol (OVSDB), NETCONF, XFlow (Netflow e Sflow) ed altri protocolli di terze parti. La libreria di Ryu riferita ai pacchetti aiuta nell'analisi e nella realizzazione di vari pacchetti dei protocolli, come VLAN, MPLS, GRE, eccetera.

3.2.2 Protocollo e controller OpenFlow

Ryu include una libreria di codifica e decodifica del protocollo OpenFlow, che supporta fino alla versione 1.4 .

Uno dei componenti chiave dell'architettura di Ryu è il controller OpenFlow, che è il responsabile della gestione degli switch OpenFlow utilizzati per configurare i flussi, gestire gli eventi, eccetera. Il controller OpenFlow è una delle sorgenti di eventi interne nell'architettura di Ryu.

La tabella seguente, in Figura 3.3, riassume i messaggi, la struttura e le corrispondenti API del protocollo OpenFlow di Ryu.

Controller to Switch Messages	Asynchronous Messages	Symmetric Messages	Structures
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	Flow-match
send_msg API and packet builder APIs	set_ev_cls API and packet parser APIs	Both Send and Event APIs	

Figura 3.3: Messaggi, struttura e API di OpenFlow in Ryu. [12]

3.2.3 Ryu Manager

Il *Ryu manager* è l'eseguibile principale. Quando viene avviato, ascolta tramite un indirizzo IP specificato (ad esempio 0.0.0.0) ed una porta specificata (6633 di default). Osservando ciò, ogni switch OpenFlow (hardware o Open vSwitch oppure OVS) può connettersi al Ryu manager. Il manager è il componente fondamentale di tutte le applicazioni di Ryu, che ne ereditano sempre la classe *app_manager*. [12]

3.2.4 Ryu Northbound

Al livello API, Ryu include un plug-in di Neutron, per quanto riguarda OpenStack, che supporta le configurazioni VLAN e GRE-based. Ryu è inoltre predisposto di una interfaccia REST per le relative operazioni di OpenFlow. [12]

3.2.5 Applicazioni di Ryu

Ryu è distribuito con svariate applicazioni come semplici switch, router, firewall, GRE tunnel, VLAN, eccetera. Le applicazioni sono entità *single-threaded*, ovvero eseguite singolarmente una alla volta, che implementano varie funzionalità. Per comunicare tra loro, tali applicazioni si inviano vicendevolmente eventi asincroni.

L'architettura funzionale di un'applicazione Ryu è mostrata in Figura 3.4

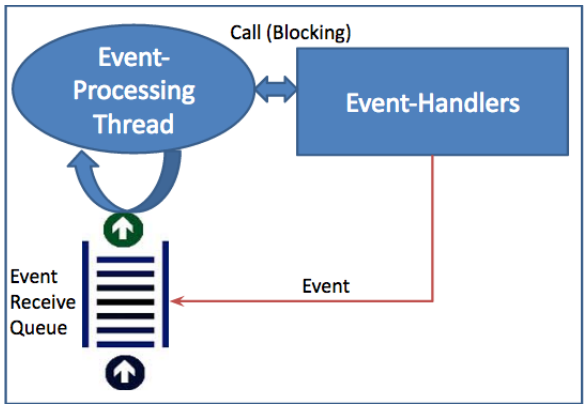

Figura 3.4: Architettura funzionale di un'applicazione di Ryu. [12]

Ogni applicazione è predisposta di una coda per gli eventi ricevuti, principalmente gestita con logica FIFO (ovvero *First In First Out*, dove i primi eventi ricevuti sono i primi ad essere letti) per preservare l'ordine di tali eventi. A seconda della tipologia di evento, l'applicazione poi richiamerà un relativo componente responsabile per la tipologia riscontrata che anterrà il controllo del sistema fino al termine della sua funzione, senza processare altri eventi nel frattempo.

Le applicazioni di Ryu possono essere eseguite fornendo un file di configurazione al Ryu manager:

```
ryu-manager [--flagfile <path to configuration file>]
    [generic/application specific options...]
```

3.3 Installazione

L'installazione è semplice e veloce [13], è sufficiente utilizzare il comando *pip* in questo modo:

```
% pip install ryu
```

Altrimenti, se si preferisce effettuare l'installazione dal codice sorgente:

```
% git clone git://github.com/osrg/ryu.git
% cd ryu; python ./setup.py install
```

3.4 Scrivere applicazioni in Ryu

In questa sezione, sarà introdotta una semplice applicazione di Ryu che faccia lavorare gli switch OpenFlow come uno switch di livello L2. [12]

Innanzitutto è di fondamentale importanza consultare la documentazione di riferimento delle API di Ryu relative ad OpenFlow, presso il sito:
`http://ryu.readthedocs.org/en/latest/ofproto_ref.html`.

Un'applicazione di Ryu non è altro che un modulo Python in cui viene definita una sottoclasse di **ryu.base.app_manager.RyuApp**. Se due o più classi del genere sono definite in un modulo, la prima (in ordine di nome) verrà

selezionata dall'*app manager*, è supportata infatti solo una singola istanza, di un'applicazione data, per volta volta.

In questo caso, l'applicazione verrà chiamata *L2Switch*, erediterà dalla classe base **app_manager.RyuApp** e sarà definita così:

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls

class L2Switch(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)
```

Un'applicazione di Ryu può sia generare che ricevere un evento. Per crearne uno, viene chiamato il relativo metodo **ryu.base.app_manager** di RyuApp.

Un'applicazione può mettersi in ascolto riguardo eventi specifici grazie al decoratore **ryu.controller.handler.set_ev_cls**, che indica a ryu quando richiamare la funzione selezionata dal decoratore stesso.
Il primo argomento del decoratore **set_ev_cls** esprime l'evento di interesse, che innesca la seguente chiamata della funzione.
Il secondo argomento indica invece lo stato dello switch, ad esempio, se si volesse che l'applicazione ignori i messaggi di *packet-in* prima del termine della negoziazione tra il controller Ryu e lo switch, è necessario utilizzare **MAIN_DISPATCHER** come secondo argomento, che indica che la chiamata della seguente funzione avviene solamente dopo che la negoziazione sia stata completata.

In questo esempio, l'applicazione è interessata agli eventi di *packet-in* ed ogni volta che il controller ne riceve uno, una funzione responsabile della sua gestione viene chiamata. In aggiunta, questa applicazione è capace di inviare un pacchetto ricevuto a tutte le porte.

È possibile definire tali operazioni con:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    ofp_parser = dp.ofproto_parser

    actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
    out = ofp_parser.OFPPacketOut(
```

```
        datapath=dp, buffer_id=msg.buffer_id,
        in_port=msg.in_port,actions=actions)
    dp.send_msg(out)
```

Si osservi ora più in dettaglio la funzione **packet_in_handler**. Ogni volta che un messaggio di *packet-in* viene ricevuto, tale funzione viene chiamata, grazie alla API **set_ev_cls**.
La struttura di tale *packet-in* è rappresentata dall'oggetto **ev.msg**, grazie al quale, tramite **msg.datapath**, è possibile accedere al *data path*, ovvero all'identificativo dello switch.

A questo punto, l'oggetto **dp.ofproto** rappresenta il protocollo OpenFlow negoziato tra il controller e lo switch, mentre l'oggetto **ofproto_parser** analizza il messaggio di *packet-in* ricevuto.

Una volta ricevuto il messaggio di *packet-in*, si ha la necessità di reindirizzarlo verso tutte le porte dello switch tramite un messaggio di *packet-out*, appartenente alla classe **OFPActionOutput**, nel quale viene specificata la porta dalla quale si desidera inviare il pacchetto. Siccome, in questo caso, è d'interesse inviare tale pacchetto a tutte le porte, la porta prescelta sarà **OFPP_FLOOD**. Ora si utilizzerà quindi la classe **OFPPacketOut** per costruire il messaggio di *packet-out*.

Infine è necessario chiamare il metodo **send_msg**, della classe *data path*, con un oggetto della classe di messaggi OpenFlow, che fa costruire il messaggio al controller, per poi inviarlo allo switch.

Terminata perciò la scrittura dell'applicazione di Ryu, si è pronti ad avviarla. Siccome si tratta di uno script di Python, è possibile salvarla con qualsiasi nome ed estensione.

Considerando di aver salvato il codice precedente nel file **L2switch.py**, questa applicazione può essere lanciata tramite il seguente comando del Ryu manager:

```
% ryu-manager L2switch.py
```

Oppure, nel caso si riscontrino problemi col comando precedente, considerando di aver salvato tale file nella cartella **home/ryu/ryu/app** è possibile utilizzare:

```
% cd /home/UTENTE/ryu && ./bin/ryu-manager
    --verbose ryu/app/L2switch.py
```

Ottenendo il risultato nella seguente Figura 3.5

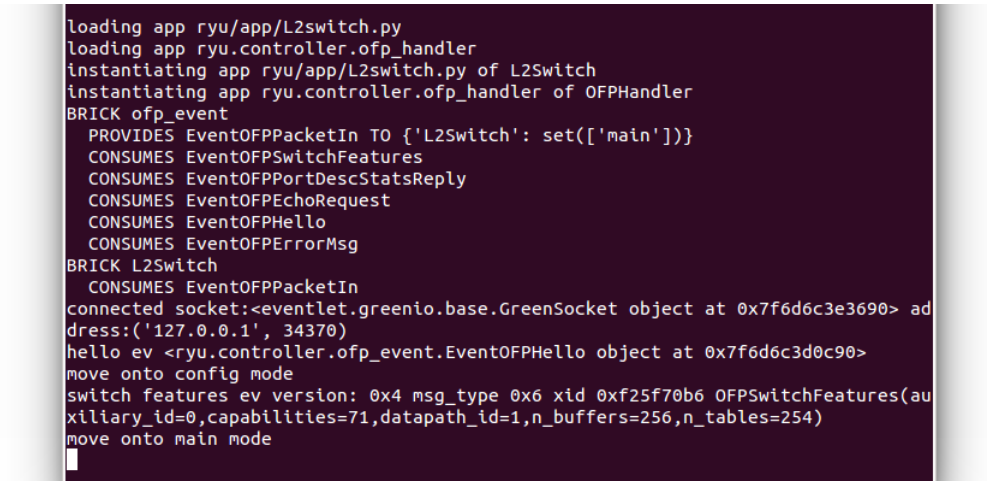

Figura 3.5: Risultato dell'avvio dell'applicazione di Ryu da terminale.

Il codice del controller appena utilizzato è presente, per intero, nell'Appendice A.1 **L2switch.py**.

3.5 Altri esempi di applicazioni di Ryu

In questa sezione verranno mostrati ulteriori esempi e modifiche del codice, in modo da fornire qualche conoscenza in più e, al tempo stesso, dare un'idea delle potenzialità di questo controller.

3.5.1 Simple switch

Un'evoluzione del codice precedentemente descritto può essere rappresentata dalla capacità di salvare le regole, relative ai flussi di pacchetti, nello switch, per evitare il ripresentarsi del messaggio di *packet-in*.

Verranno di seguito illustrate le principali modifiche effettuate, rispetto al codice già utilizzato. È possibile consultare il codice di tale applicazione nell'Appendice A.2 **simple_switch_13.py**.

- Creazione di una tabella in cui salvare gli indirizzi MAC per evitare, successivamente, il *FLOOD* dei pacchetti in uscita

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

- Definizione di una funzione di aggiunta di regole allo switch:

```
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)
```

- Aggiunta della regola relativa all'invio al controller, nel caso in cui il pacchetto non trovi alcun riscontro nella flow table (caso di *Table-miss*)

```
@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

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

- Aggiunta dell'indirizzo MAC, prelevato dal pacchetto ricevuto, nella tabella precedentemente creata (codice presente all'interno della funzione di gestione del messaggio di *packet-in*)

```
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)]
```

- Aggiunta della regola nella flow table ed invio del messaggio di *packet-out*

  (Le righe precedute dal carattere # sono commenti presenti nel codice)

```
# 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)
```

In questo modo, si è realizzato uno switch capace di salvare le regole relative al traffico, evitando perciò il presentarsi del messaggio di *packet-in* per i flussi già analizzati dal controller.

3.5.2 ARP e ICMP

Un'altra possibile aggiunta al codice del controller potrebbe essere l'installazione delle regole relative ai protocolli ARP e ICMP appena dopo l'instaurazione del collegamento con lo switch. Le seguenti linee di codice sono da inserire all'interno della funzione **switch_features_handler**, ovvero quella in cui è presente anche la 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)
```

In questo modo, tali regole verranno installate alla connessione con lo switch, evitando perciò il generarsi di messaggi di *packet-in* per quanto riguarda tali protocolli.

3.5.3 TCP e UDP

È possibile, e consigliato, effettuare l'analisi del messaggio di *packet-in* concentrandosi sul protocollo, in modo da poter inserire nello switch regole altamente selettive, che aumentino quindi la sicurezza e migliorino il funzionamento del sistema.

Le seguenti linee di codice sono da inserire all'interno della funzione di gestione del messaggio di *packet-in*:

```
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(
                           ofproto.OFPP_NORMAL)]
        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)
```

Questa è la principale modalità di aggiunta di regole alla flow table. Tramite la classe di corrispondenze **OFPMatch** è possibile utilizzare filtri più o meno selettivi per caratterizzare le regole inserite, facendo uso delle seguenti parole chiave in Tabella 3.1:

Argomento	**Valore**	**Descrizione**
in_port	Integer 32bit	Switch input port
in_phy_port	Integer 32bit	Switch physical input port
metadata	Integer 64bit	Metadata passed between tables
eth_dst	MAC address	Ethernet destination address
eth_src	MAC address	Ethernet source address
eth_type	Integer 16bit	Ethernet frame type
vlan_vid	Integer 16bit	VLAN id
vlan_pcp	Integer 8bit	VLAN priority
ip_dscp	Integer 8bit	IP DSCP (6 bits in ToS field)
ip_ecn	Integer 8bit	IP ECN (2 bits in ToS field)

ip_proto	Integer 8bit	IP protocol
ipv4_src	IPv4 address	IPv4 source address
ipv4_dst	IPv4 address	IPv4 destination address
tcp_src	Integer 16bit	TCP source port
tcp_dst	Integer 16bit	TCP destination port
udp_src	Integer 16bit	UDP source port
udp_dst	Integer 16bit	UDP destination port
sctp_src	Integer 16bit	SCTP source port
sctp_dst	Integer 16bit	SCTP destination port
icmpv4_type	Integer 8bit	ICMP type
icmpv4_code	Integer 8bit	ICMP code
arp_op	Integer 16bit	ARP opcode
arp_spa	IPv4 address	ARP source IPv4 address
arp_tpa	IPv4 address	ARP target IPv4 address
arp_sha	MAC address	ARP source hardware address
arp_tha	MAC address	ARP target hardware address
ipv6_src	IPv6 address	IPv6 source address
ipv6_dst	IPv6 address	IPv6 destination address
ipv6_flabel	Integer 32bit	IPv6 Flow Label
icmpv6_type	Integer 8bit	ICMPv6 type
icmpv6_code	Integer 8bit	ICMPv6 code
ipv6_nd_target	IPv6 address	Target address for ND
ipv6_nd_sll	MAC address	Source link-layer for ND
ipv6_nd_tll	MAC address	Target link-layer for ND
mpls_label	Integer 32bit	MPLS label
mpls_tc	Integer 8bit	MPLS TC
mpls_bos	Integer 8bit	MPLS BoS bit
pbb_isid	Integer 24bit	PBB I-SID
tunnel_id	Integer 64bit	Logical Port Metadata
ipv6_exthdr	Integer 16bit	IPv6 Extension Header pseudo-field

Tabella 3.1: Parole chiave della classe **OFPMatch**. [15]

Sono realizzati in questo modo anche i codici dei controller utilizzati nelle topologie L2 ed L3 nel Capitolo 5.

Capitolo 4

Dynamic Service Chaining

In questo capitolo viene presentato il concetto di Dynamic Service Chaining, illustrandone le basi e, soprattutto, gli aspetti operativi, introducendo quindi i casi di studio L2 ed L3 successivamente affrontati in modo più pratico nel Capitolo 5.

4.1 Dynamic Service Chaining

Il Dynamic Service Chaining fa riferimento ad un servizio, implementato tramite software, il cui fine è la gestione, in modo dinamico, del traffico di una rete condivisa da più utenti, analizzando, redirezionando e gestendo i flussi a seconda dei profili associati agli utenti, delle tipologie di servizi offerti ad essi e di altre caratteristiche. La sua applicazione permette quindi una maggiore ottimizzazione della rete, attraverso un migliore utilizzo delle risorse disponibili e grazie all'offerta di un servizio su misura in modo totalmente dinamico, senza effettuare alcuna modifica allo strato hardware del sistema. [16] Risulta perciò un ulteriore approccio per contrastare l'ossificazione attuale della rete, rendendola più flessibile e malleabile alle necessità del singolo utente.

Il Dynamic Service Chaining trova così applicazione sia nelle topologie standard della rete classica, sia in ambienti Cloud, dove più utenti accedono alle stesse risorse. In questo modo si ottengono migliori prestazioni, soprattutto in caso di congestione della rete, operando sui vari flussi attivi nel rispetto dei profili degli utenti connessi.

4.1.1 Funzionamento

Il Dynamic Service Chaining viene applicato ad una rete utilizzando un controller e quindi realizzando un'architettura SDN. Per descrivere il comportamento di tale servizio è sufficiente utilizzare la macchina a stati finiti in Figura 4.1, che illustra la sequenza di operazioni svolte per ogni flusso presente nella rete.

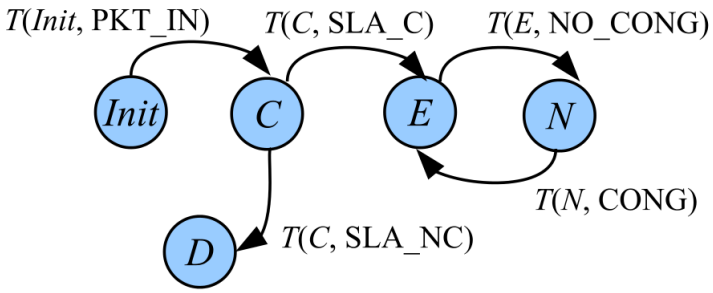

Figura 4.1: Diagramma della macchina a stati finiti rappresentante il funzionamento del controller durante il processo di un flusso. [17]

Nella macchina a stati finiti le transizioni di stato dipendono dallo stato corrente e dall'input ricevuto. Ogni transizione provoca quindi un'azione sul flusso in analisi.

Vengono considerate le seguenti definizioni:

- *stato s*, da un insieme finito, $s \in \{Init,\ C,\ E,\ N,\ D\}$, dove $s_0 = Init$ è lo stato iniziale;
- *input i*, da un insieme finito, $i \in$ {PKT_IN, SLA_C, SLA_NC, CONG, NO_CONG};
- *azioni A(f, s, i)*, definite secondo la tecnologia SDN utilizzata;
- *transizioni di stato T*, funzioni che fanno corrispondere una coppia *(s, i)* ad una coppia *(s', A(f, s, i))*.

Il controller può quindi trovarsi nei seguenti stati:

- *Init*, nel quale il controller installa, nel network node, regole generali di forwarding indipendenti dal flusso. Questo è lo stato da cui parte un nuovo processo ogni volta che un nuovo flusso f si presenta;

- *C*, nel quale il flusso f viene analizzato e classificato per determinarne il suo *Service Level Agreement (SLA)*;

- *E*, nel quale viene fatta rispettare la *Quality of Service (QoS)* per il flusso *f*, secondo il suo SLA;

- *N*, nel quale non viene fatta rispettare la QoS per il flusso *f*, perché la rete non è in un potenziale stato di congestione ed il flusso f può sfruttare un numero maggiore di risorse;

- *D*, nel quale il flusso f è sottoposto ad azioni di controllo (ad esempio i pacchetti possono venire scartati) poiché non è conforme allo SLA.

Il funzionamento è quindi il seguente. Quando un nuovo flusso f arriva, ad esempio con la ricezione di un packet-in (PKT_IN input), il controller crea un nuovo processo che entra nello stato *C*, mentre l'azione A(*f*, *Init*, PKT_IN) abilita il redirezionamento del flusso f ad un dispositivo che ne implementi la classificazione. A seconda del risultato, che sarà un input della macchina a stati, il flusso f verrà controllato, se non conforme (SLA_NC), o trattato a seconda dei requisiti di QoS, se conforme (SLA_C).

Per avere un approccio maggiormente cauto, ci si sposterà sullo stato *E*, in cui verrà fatto rispettare lo SLA, con una corretta azione di redirezionamento. In questa situazione, nel caso non sia presente una congestione nella rete e siano disponibili abbastanza risorse, il flusso f può essere processato da una catena di servizi che non esige il rispetto del QoS ed il sistema si posiziona allo stato *N*. A questo punto, si assume che sia presente un'entità separata, col compito di monitorare lo stato della rete e che interagisca col controller attraverso un'interfaccia di northbound per notificare la presenza (CONG) o meno (NO_CONG) di possibile congestione della rete, permettendo così al processo di passare dinamicamente dallo stato N allo stato E e viceversa.

Risulta perciò di fondamentale importanza la presenza di un dispositivo destinato alla classificazione dei flussi presenti nella rete. Tale compito sarà svolto nei due seguenti casi, L2 ed L3, dall'analizzatore del traffico, attuando la *Deep Packet Inspection (DPI)*, un metodo di filtraggio dei pacchetti della rete, in cui viene esaminata la parte dati (ed eventualmente anche l'header) di un pacchetto. [18]

Verranno ora illustrati i casi di studio L2 ed L3, due esempi di applicazione del Dynamic Service Chaining. Successivamente, nel Capitolo 5, verrà mostrato il lavoro riguardante la realizzazione pratica di tali casi.

4.1.2 Caso di studio L2

Il caso di studio di livello L2 è rappresentato dalla topologia in Figura 4.2.

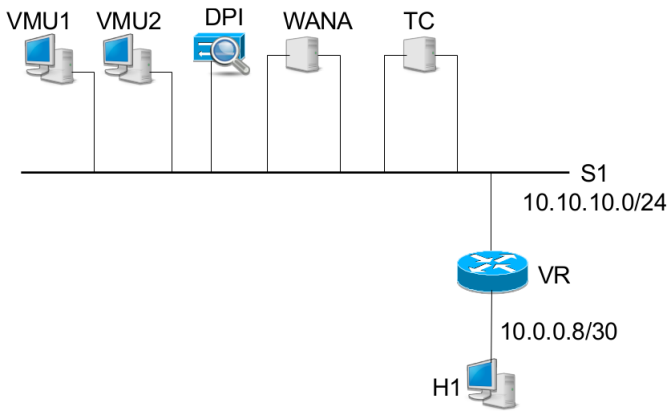

Figura 4.2: Topologia del caso L2. [19]

Questa topologia è caratterizzata dalla presenza di un singolo switch di rete, al quale sono collegati le seguenti virtual machines:

- *VMU1*, che rappresenta il Business User, ovvero l'utente col profilo a maggiore priorità, il cui traffico richiede, quando necessario, il passaggio tramite il *Wide Area Network Accelerator (WANA)* per rispettare lo SLA, altrimenti, nel caso sia disponibile una banda adeguata, vengono rispettate le tradizionali regole di indirizzamento;

- *VMU2*, che rappresenta il Residence User, ovvero l'utente col profilo a minore priorità, caratterizzato da un servizio *best-effort*, ovvero senza garanzia di qualità. Il traffico di tale utente segue le tradizionali regole di indirizzamento quando utenti a priorità maggiore non utilizzano la rete, inn caso contrario, il suo flusso attraversa una funzione di *shaping*, attuata dal *Traffic Control (TC)*, che ne modella la disponibilità di banda, riservandone perciò una certa quantità agli utenti più prioritari;

- *DPI*, che rappresenta l'analizzatore di traffico. Come espresso in precedenza si occupa di classificare il flusso generato da un utente, per poi procedere al reindirizzamento dello stesso a seconda del profilo a cui corrisponde;

- *WANA*, che rappresenta il *Wide Area Network Accelerator*. Questo dispositivo, implementato tramite software, ottimizza la disponibilità di banda di un utente operando sul flusso dati, effettuando la compressione e la deduplicazione (eliminazione dei dati presenti in duplice copia) dei pacchetti, riducendo perciò la quantità di dati che è necessario inviare;

- *TC*, che rappresenta il *Traffic Control*. Questo dispositivo implementa, tramite software, una funzione di riduzione della banda disponibile all'utente, rendendolo in questo modo "innocuo" nei confronti dei flussi di utenti a priorità maggiore;

- *VR*, che rappresenta il Router virtuale, ovvero il nodo di uscita della rete di livello L2;

- *H1*, connessa al VR, che rappresenta la destinazione per le comunicazioni lanciate dagli utenti *VMU1* e *VMU2*. Il nodo destinazione viene in questo caso implementato per comodità con un singolo host, non considerando la necessaria presenza di ulteriori dispositivi e servizi, come per il caso della decompressione dei pacchetti precedentemente trattati dal WANA.

DPI, WANA e TC sono funzioni di rete, offerte dal provider di servizi di Internet, in forma di virtual machines dedicate. Di queste, nel WANA e nel TC, essendo entrambe provviste di due interfacce connesse allo stesso switch, è necessario impostare un *bridge* interno per mantenere la connettività L2. Questa configurazione, però, può generare dei loop nella topologia ed è possibile porvi rimedio disabilitando lo *Spanning Tree Protocol (STP)*.

In questa situazione, la macchina a stati finiti in Figura 4.1 sarà composta dai seguenti stati:

- *Init*, nel quale vengono installate le regole relative al flusso di pacchetti ICMP e ARP, caratterizzate da azione di tipo NORMAL, che permette loro di attraversare lo switch liberamente, senza provocare messaggi di packet-in al controller;

- *C*, nel quale il flusso viene analizzato dal DPI ed, in contemporanea, inviato anche a destinazione;
- *E*, nel quale il flusso di VMU1 viene redirezionato al WANA e da esso verso la destinazione, mentre il flusso di VMU2 viene redirezionato al TC e da esso a destinazione (e viceversa per i flussi indirizzati a VMU1 e VMU2).
- *N*, nel quale sia il flusso di VMU1 che quello di VMU2 vengono mandati direttamente verso la destinazione;
- *D*, nel quale il flusso, non riconosciuto valido, viene scartato.

Lo studio di questa topologia viene quindi effettuato per fasi.

Caso L2: Prima fase

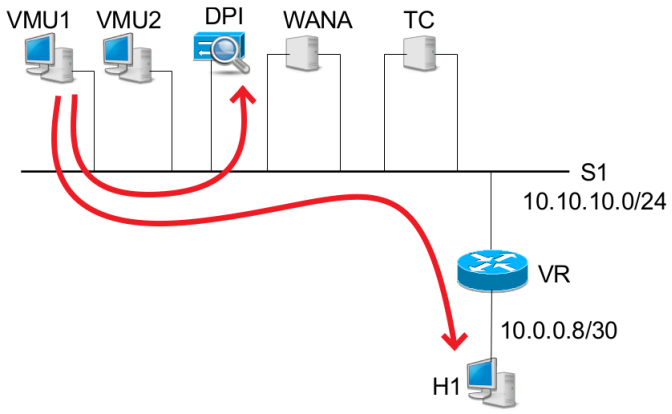

Figura 4.3: Prima fase del caso di studio L2.

La prima fase, denominata *fase di mirroring*, ha luogo quando l'utente VMU1 genera il traffico f_1, che, dopo la ricezione di un messaggio di *packet-in* da parte del controller, viene inviato contemporaneamente sia al DPI che al nodo di uscita, ovvero ad H1 tramite VR, e viceversa, per tutta la durata della fase. Si procede in questo modo all'analisi del flusso f_1, col compito di identificarne la provenienza.

Nella realizzazione pratica, presente nel Capitolo 5, tale fase ha una durata di 60 secondi.

Caso L2: Seconda fase

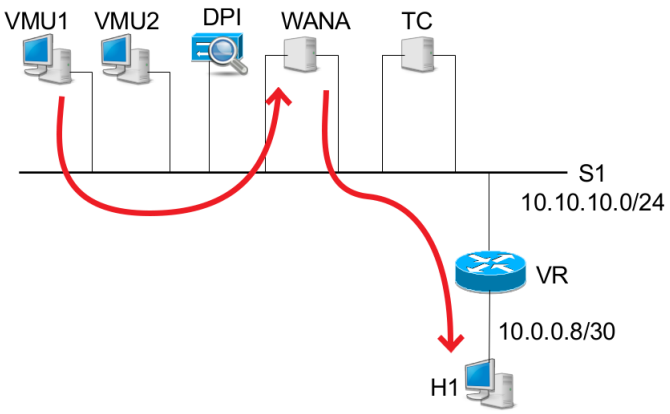

Figura 4.4: Seconda fase del caso di studio L2.

In questa fase momentanea, il flusso f_1 destinato al DPI viene rimosso, mentre quello diretto in uscita, verso VR, viene deviato e fatto passare tramite il WANA per poi procedere verso l'uscita, come previsto dal diagramma della macchina a stati in Figura 4.1.

Nella realizzazione pratica questa fase ha una durata di un secondo.

Caso L2: Terza fase

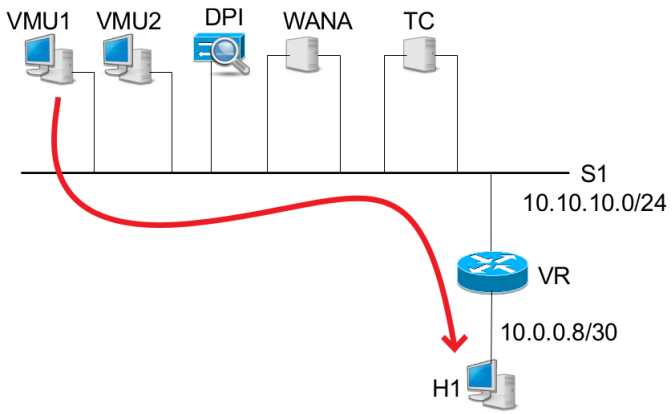

Figura 4.5: Terza fase del caso di studio L2.

In questa fase, siccome non sono presenti ulteriori utenti ad occupare la banda, il flusso f_1 viene indirizzato direttamente al nodo di uscita, quindi verso VR, senza alcun passaggio intermedio. Si utilizza, in tal modo, tutta la banda disponibile nella rete.

Il traffico di VMU1 seguira perciò questo percorso fino a quando il controller non riscontrerà una situazione di congestione nella rete.

Caso L2: Quarta fase

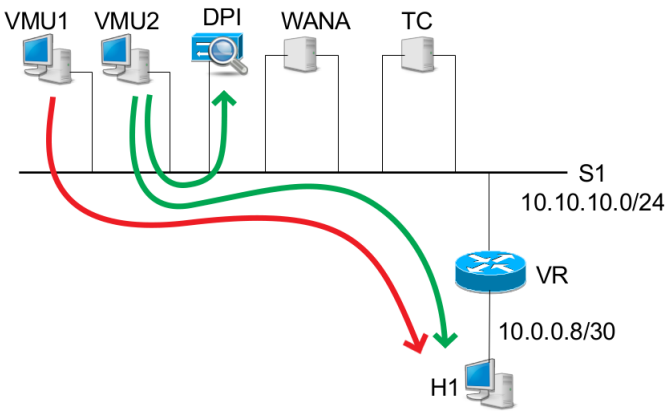

Figura 4.6: Quarta fase del caso di studio L2.

Si giunge a questa fase nel momento in cui l'utente VMU2 inizia a generare il traffico f_2, dando luogo ad un'altra *fase di mirroring* in cui il nuovo flusso f_2 viene inviato contemporaneamente sia al DPI che al nodo di uscita, ovvero ad H1 tramite VR, per tutta la durata della fase, mentre per il flusso f_1 di VMU1 permane la situazione della fase precedente, ovvero di indirizzamento diretto verso l'uscita. Si procede in questo modo all'analisi del flusso f_2, per identificarne la provenienza.

Come nella precedente fase di mirroring, nella realizzazione pratica tale fase ha una durata di 60 secondi.

Caso L2: Quinta fase

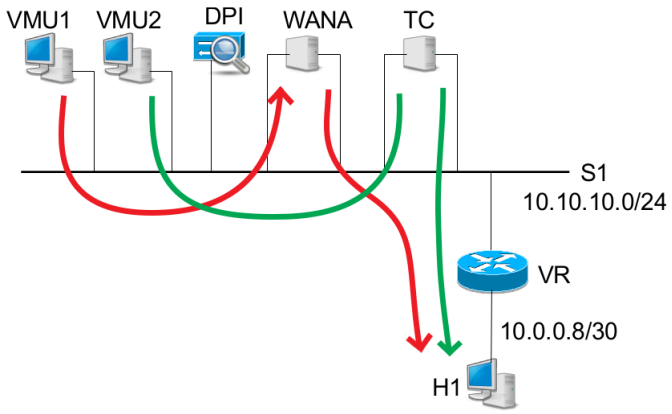

Figura 4.7: Quinta fase del caso di studio L2.

In quest'ultima fase il controller, riconoscendo la presenza di una congestione grazie all'analisi svolta dal DPI, redireziona il flusso f_1 da VMU1 al WANA e da esso al nodo di uscita VR, come successo nella seconda fase, ed il flusso f_2 da VMU2 al TC e da esso al nodo di uscita VR (e viceversa per i flussi indirizzati a VMU1 e VMU2).

In questo modo si contrasta la riduzione di banda disponibile all'utente maggiormente prioritario, causata dalla presenza di traffico proveniente da altri utenti.

Questa fase rappresenta il termine dello studio eseguito sul livello L2. Risulta necessaria, per continuare, la presenza di un'entità separata che monitori lo stato della rete e che interagisca col controller per informarlo di eventuali situazioni di non congestione,ad esempio quando uno dei due utenti attivi termina la comunicazione. In questo modo sarebbe concesso all'utente attivo rimasto lo sfruttamento della banda completa. A regime, il sistema transiterebbe perciò tra questa e le ultime due fasi a seconda della quantità di utenti nella rete, ottimizzandone in questo modo la disponibilità di banda.

Ogni azione di reindirizzamento precedentemente descritta viene realizzata da un controller appositamente programmato, di cui è analizzato un esempio nel Capitolo 5. Tale controller agisce sulla *flow-table* dello switch grazie a mirate configurazioni di priorità e timeout delle regole inserite.

4.1.3 Caso di studio L3

Il caso di studio di livello L3 è invece rappresentato dalla topologia in Figura 4.8.

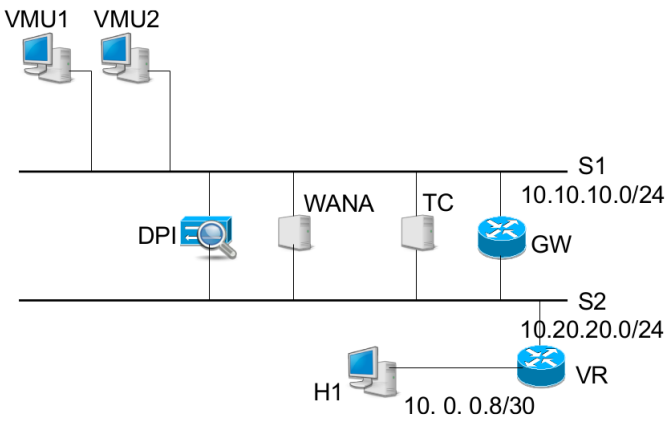

Figura 4.8: Topologia del caso L3. [19]

Questa topologia è caratterizzata dalla presenza di un due switch di rete, *S1* ed *S2*, ai quali sono collegati le virtual machines del caso L2:

- *VMU1*, collegata allo switch S1;
- *VMU2*, collegata allo switch S1;
- *DPI*, con due interfacce, una collegata allo switch S1, l'altra allo switch S2;
- *WANA*, con due interfacce, una collegata allo switch S1, l'altra allo switch S2;
- *TC*, con due interfacce, una collegata allo switch S1, l'altra allo switch S2;
- *VR*, nodo di uscita della rete di livello L3, collegata allo switch S2;
- *H1*, connessa al VR.

Oltre a tali virtual machines, già presentate nel caso precedente, è presente in questa topologia anche:

- *GW*, con due interfacce, una collegata allo switch S1, l'altra allo switch S2. Essa rappresenta il Gateway intermedio per i due switch ed è il Default Gateway per gli utenti VMU1 e VMU2, nella situazione normale della rete.

Anche in questo caso DPI, WANA e TC sono funzioni di rete, offerte dal provider di servizi di Internet, in forma di virtual machines dedicate. Siccome in questa topologia WANA e TC, non sono provviste di interfacce connesse allo stesso switch, non è necessario impostare un *bridge* interno, dato che la connettività di livello L3 non è compromessa.

In questa topologia, la macchina a stati finiti in Figura 4.1 avrà comportamento simile a quello del caso L2, sarà perciò composta dai seguenti stati:

- *Init*, nel quale vengono installate le regole relative al flusso di pacchetti ICMP e ARP, caratterizzate da azione di tipo NORMAL, che permette loro di attraversare lo switch liberamente, senza provocare messaggi di packet-in al controller;

- *C*, nel quale il flusso attraversa il DPI, dove viene analizzato, e da esso prosegue verso la destinazione;

- *E*, nel quale il flusso di VMU1 viene redirezionato al WANA e da esso verso la destinazione, mentre il flusso di VMU2 viene redirezionato al TC e da esso a destinazione (e viceversa per i flussi indirizzati a VMU1 e VMU2).

- *N*, nel quale sia il flusso di VMU1 che quello di VMU2 vengono mandati verso la destinazione tramite il loro Default Gateway (GW);

- *D*, nel quale il flusso, non riconosciuto valido, viene scartato.

Anche per il livello L3 lo studio di questa topologia viene effettuato per fasi.

Caso L3: Prima fase

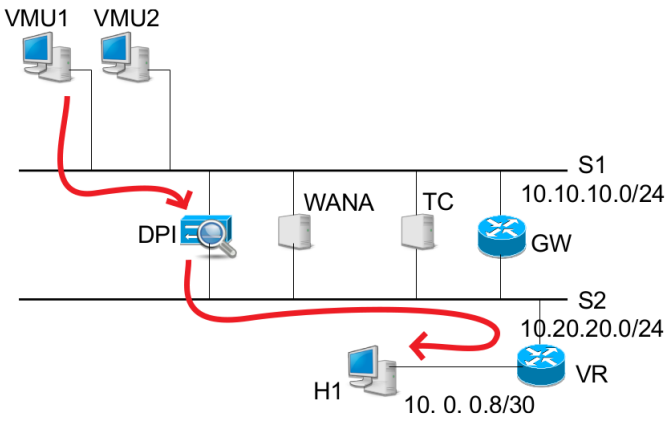

Figura 4.9: Prima fase del caso di studio L3.

La prima fase ha luogo quando l'utente VMU1 genera il traffico f_1, che, dopo la ricezione di un messaggio di *packet-in* da parte del controller, viene inviato al DPI e da esso al nodo di uscita, ovvero ad H1 tramite VR, e viceversa, per tutta la durata della fase. In questo modo si procede all'analisi del flusso f_1, così da identificarne la provenienza.

Come nel livello L2, nella realizzazione pratica presente nel Capitolo 5, tale fase di analisi ha una durata di 60 secondi.

Caso L3: Seconda fase

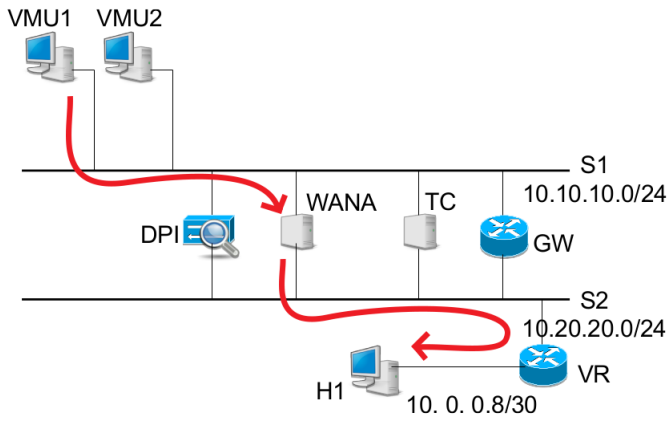

Figura 4.10: Seconda fase del caso di studio L3.

In questa fase momentanea, il flusso f_1 destinato al DPI viene redirezionato e fatto passare tramite il WANA, per poi procedere verso l'uscita, come previsto dal diagramma della macchina a stati in Figura 4.1.

Nella realizzazione pratica questa fase ha una durata di un secondo.

Caso L3: Terza fase

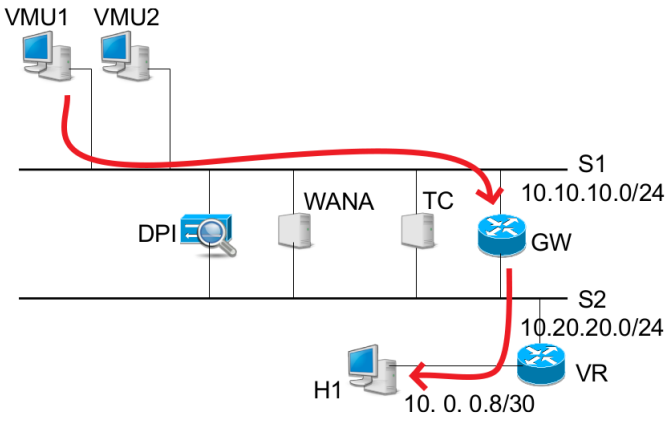

Figura 4.11: Terza fase del caso di studio L3.

In questa fase, siccome non sono presenti ulteriori utenti ad occupare la banda, il flusso f_1 viene indirizzato tramite il Gateway GW al nodo di uscita VR, quindi verso H1, e viceversa. Si utilizza, in tal modo, tutta la banda disponibile nella rete.

Il traffico di VMU1 seguira perciò questo percorso fino a quando il controller non riscontrerà una situazione di congestione nella rete.

Caso L3: Quarta fase

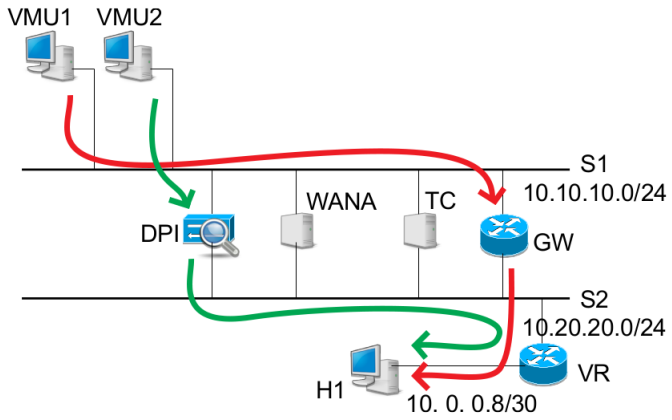

Figura 4.12: Quarta fase del caso di studio L3.

Si giunge a questa fase nel momento in cui l'utente VMU2 inizia a generare il traffico f_2, dando luogo ad un'altra fase di analisi in cui il nuovo flusso f_2 viene inviato al DPI, da esso al nodo di uscita, ovvero ad H1 tramite VR, e viceversa, per tutta la durata della fase, mentre per il flusso f_1 di VMU1 permane la situazione della fase precedente, ovvero di indirizzamento verso l'uscita tramite GW. Si procede in questo modo all'analisi del flusso f_2, per identificarne la provenienza.

Come nella precedente fase di analisi, nella realizzazione pratica tale fase ha una durata di 60 secondi.

Caso L3: Quinta fase

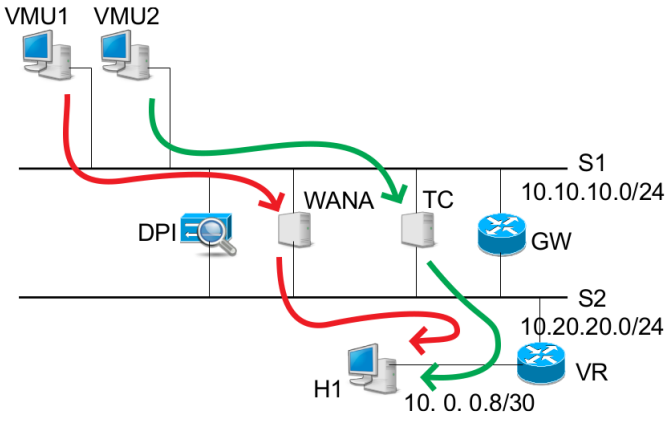

Figura 4.13: Quinta fase del caso di studio L3.

In quest'ultima fase, il controller, riconoscendo la presenza di una congestione grazie all'analisi svolta dal DPI, redireziona il flusso f_1 da VMU1 al WANA e da esso al nodo di uscita VR, come successo nella seconda fase, ed il flusso f_2 da VMU2 al TC e da esso al nodo di uscita VR (e viceversa per i flussi indirizzati a VMU1 e VMU2).

In questo modo si contrasta la riduzione di banda disponibile all'utente maggiormente prioritario, causata dalla presenza di traffico proveniente da altri utenti.

Questa fase rappresenta il termine dello studio eseguito sul livello L3. Risulta necessaria per continuare, anche in questo caso, la presenza di un'entità separata che monitori lo stato della rete e che interagisca col controller per informarlo di eventuali situazioni di non congestione,ad esempio quando uno dei due utenti attivi termina la comunicazione. In questo modo sarebbe concesso all'utente rimasto attivo lo sfruttamento della banda completa. Come nel caso L2, a regime, il sistema transiterebbe perciò tra questa e le ultime due fasi a seconda della quantità di utenti nella rete, ottimizzandone in questo modo la disponibilità di banda.

Capitolo 5

Emulazione del Dynamic Service Chaining

In questo capitolo viene affrontata la realizzazione pratica dei casi di studio delle topologie L2 ed L3 riguardanti il Dynamic Service Chaining ed introdotti in precedenza nel Capitolo 4, presentando ed analizzando quindi gli esperimenti svolti ed i risultati ottenuti.

5.1 Caso di studio L2

Il caso di studio di livello L2, la cui topologia è già stata mostrata nel Capitolo 4 in Figura 4.2, viene realizzato emulando virtualmente la rete ed azionando il controller Ryu appositamente realizzato. Tale emulazione è resa possibile grazie all'ambiente software Mininet, che permette di creare una rete virtuale, con kernel, switch e applicazioni su una singola macchina (Virtual Machine, cloud o nativa), in pochi secondi, utilizzando un codice scritto in linguaggio Python.

5.1.1 Codice Mininet

Per realizzare la rete del caso L2, si è fatto uso del codice presente nell'Appendice A.3 **test-l2-scenario.py**, di cui ora viene analizzata parte del contenuto.

- Una volta importate le librerie necessarie, si provvede a definire la funzione che genera la rete, configurando il controller come esterno e creando lo switch OpenFlow, al quale viene disabilitato lo *Spanning Tree Protocol (STP)* per mantenere la connettività L2 a causa della presenza di dispositivi, WANA e TC, con più interfacce connesse allo stesso switch

```
info("*** Create an empty network *** \n")
net = Mininet(controller=RemoteController,
              link=TCLink, build=False, xterms=True)
info("\n*** Controller will be external *** \n")
info("\n*** Creating Switch *** \n")
s1 = net.addSwitch('s1')
s1.cmd( 'ovs-vsctl del-br ' + s1.name )
s1.cmd( 'ovs-vsctl add-br ' + s1.name )
s1.cmd( 'ovs-vsctl set Bridge '+ s1.name +
        ' stp_enable=false protocols=OpenFlow13' )
```

- Si creano ora i dispositivi che compongono la rete, aggiungendo i collegamenti caratterizzati da una banda di 100 Mbit/s

```
info("\n*** Creating VM-User 1 *** \n")
vmu1 = net.addHost('VMU1')
info("\n*** Creating VM-User 2 *** \n")
vmu2 = net.addHost('VMU2')
info("\n*** Creating DPI *** \n")
dpi = net.addHost('DPI')
info("\n*** Creating WAN A. *** \n")
wana = net.addHost('WANA')
info("\n*** Creating TC *** \n")
tc = net.addHost('TC')
info("\n*** Creating Virtual Router *** \n")
vr = net.addHost('VR')
info("\n*** Creating External Host *** \n")
h1 = net.addHost('H1')
info("\n*** Creating Links *** \n")
net.addLink(vmu1, s1, bw=100)
net.addLink(vmu2, s1, bw=100)
net.addLink(dpi, s1, bw=100)
net.addLink(wana, s1, bw=100)
net.addLink(wana, s1, bw=100)
net.addLink(tc, s1, bw=100)
net.addLink(tc, s1, bw=100)
net.addLink(vr, s1, bw=100)
net.addLink(vr, h1, bw=100)
```

- Col seguente codice vengono configurati gli indirizzi MAC delle interfacce dei dispositivi presenti nella rete. Successivamente si procede alla disabilitazione del protocollo IPv6

```
vmu1.setMAC("00:00:00:00:00:01", vmu1.name + "-eth0")
vmu2.setMAC("00:00:00:00:00:02", vmu2.name + "-eth0")
dpi.setMAC("00:00:00:00:00:03", dpi.name + "-eth0")
wana.setMAC("00:00:00:00:00:04", wana.name + "-eth0")
wana.setMAC("00:00:00:00:00:05", wana.name + "-eth1")
tc.setMAC("00:00:00:00:00:06", tc.name + "-eth0")
tc.setMAC("00:00:00:00:00:07", tc.name + "-eth1")
vr.setMAC("00:00:00:00:00:08", vr.name + "-eth0")
vr.setMAC("00:00:00:00:00:09", vr.name + "-eth1")
h1.setMAC("00:00:00:00:00:0A", h1.name + "-eth0")

info('\n*** Disabling IPv6 ...\n')
vmu1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
```

- Si configura lo switch con la versione 1.3 del protocollo OpenFlow

```
net.start()
s1.cmd( 'ovs-vsctl set bridge ' + s1.name +
        ' protocols=OpenFlow13')
```

- Si procede quindi alla connessione dello switch al controller

```
s1.cmd('ovs-vsctl set-controller ' +  s1.name +
       ' tcp:127.0.0.1:6633')
```

- Il seguente codice è destinato all'inserimento di regole nel controller per evitare il fenomeno di *ARP storming*, scartando i pacchetti relativi al broadcast del protocollo ARP che darebbero vita a problemi nei dispositivi con due interfacce. Tali regole verranno inserite automaticamente nello switch

```
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=4,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=5,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=6,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=7,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
```

- Vengono ora rimosse le configurazioni di default riguardanti l'indirizzo IP delle interacce dei dispositivi, impostando gli indirizzi corretti e creando i bridge interni ai dispositivi con due interfacce connesse allo stesso switch, ovvero WANA e TC

```
for host in net.hosts:
   print 'Deleting ip address on ' + host.name +
       '-eth0 interface ...'
   host.cmd('ip addr del ' + host.IP(host.name +
       '-eth0') + '/8 dev ' + host.name + '-eth0')
   print 'Deleting entry in IP routing table on ' +
       host.name
   host.cmd('ip route del 10.0.0.0/8')
   print "Going to configure new IP"
   if host.name == 'WANA' or host.name == 'TC':
      print "Host with 2 interfaces: " + host.name
      host.cmd('brctl addbr br-' + host.name)
      host.cmd('brctl addif br-' + host.name + ' ' +
               host.name + '-eth0')
      host.cmd('brctl addif br-' + host.name + ' ' +
               host.name + '-eth1')
      host.cmd('ip addr add 10.10.10.' + str(count) +
               '/24 dev br-' + host.name)
      host.cmd('ip link set br-' + host.name + ' up')
      print "LB configured!"
```

```
        host.cmd('sysctl -w net.ipv4.ip_forward=1')
        print "IP Forwarding enabled!"
    elif host.name == 'H1':
        host.setIP("10.0.0." + str(count + 2), 30,
                   host.name + '-eth0')
        net.hosts[count - 2].setIP("10.0.0." +
                  str(count + 3), 30,
                  net.hosts[count - 2].name + "-eth1")
        print net.hosts[count - 2].name +
              "-eth1 interface has been configured!"
        print "[Checking VR IP] " +
              net.hosts[count - 2].IP('VR-eth1')
        net.hosts[count - 2].cmd(
                  'sysctl -w net.ipv4.ip_forward=1')
        print "On VR node: IP Forwarding enabled!"
    else:
        host.setIP("10.10.10." + str(count), 24,
                   host.name + "-eth0")
        print "[CURRENT-CHECK] IP: " +
              net.hosts[count - 1].IP(
                        net.hosts[count - 1].name +
                        '-eth0')
    count = count + 1
    print "\n"
```

- Ora si configura il Gateway su ogni host

```
print "Configuring default gw on each host.."
count = 1
for host in net.hosts:
    print "Adding default gw ..."
    if host.name != 'VR' and host.name != 'H1' and
                               host.name != 'WANA' and
                               host.name != 'TC':
        host.setDefaultRoute('dev ' + host.name +
                '-eth0 via ' +
                net.hosts[nhosts - 2].IP(
                    net.hosts[nhosts - 2].name +
                '-eth0'))
    elif host.name == 'TC' or host.name == 'WANA':
        print "Default GW manually configured"
        host.cmd('route add default gw ' +
                 net.hosts[nhosts - 2].IP(
                     net.hosts[nhosts - 2].name +
                '-eth0'))
```

```
    else:
        #H1 case
        host.setDefaultRoute('dev ' + host.name +
                '-eth0 via ' +
                net.hosts[nhosts - 2].IP(
                    net.hosts[nhosts - 2].name +
                '-eth1'))
```

- Infine si procede all'installazione del *TrafficShaper* sul TC, che provvede a limitare la banda del flusso che lo attraversa a 10 Mbit/s

```
info('\n*** Installing TrafficShaper on TC\n')
tc.cmd('tc qdisc del dev TC-eth1 root')
tc.cmd('tc qdisc add dev TC-eth1 root handle 1:
       cbq avpkt 1000 bandwidth 1000mbit')
tc.cmd('tc class add dev TC-eth1 parent 1:
       classid 1:1 cbq rate 10mbit allot 1500 prio 5
       bounded')
tc.cmd('tc filter add dev TC-eth1 parent 1:
       protocol ip prio 16 u32 match ip dst 10.0.0.9
       flowid 1:1')
tc.cmd('tc qdisc add dev TC-eth1 parent 1:1
       sfq perturb 10')
```

5.1.2 Codice Controller Ryu

Il codice Python del controller Ryu, il cui comportamento è stato descritto nel Capitolo 4, è simile a quelli precedentemente mostrati nel Capitolo 3, è consultabile interamente nell'Appendice A.4 **contrl2.py** ed è così composto.

- Avvenuta la connessione del controller allo switch, vengono subito installate le regole relative ai protocolli ARP ed ICMP per evitare il presentarsi dei messaggi di packet-in relativi

```
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)
```

- Viene quindi definita una variabile globale per la priorità delle regole inserite, oltre a due ulteriori variabili globali relative al riconoscimento avvenuto di flussi di utenti più e meno prioritari

```
#VARIABILE GLOBALE DELLA PRIORITA' DELLA REGOLA
global h
h = 1

#VARIABILI GLOBALI UTILI PER IL
    #RICONOSCIMENTO DELLA CONGESTIONE
global bu
bu=0
global ru
ru=0
```

- Successivamente, nella funzione di gestione del messaggio di packet-in, viene richiamata la variabile di priorità **h**, aumentandola di 1, in modo da inserire regole più prioritarie. Le regole presenti in questa sezione di codice, inserite solo nel caso in cui il messaggio di packet-in indichi la presenza di flusso TCP o UDP proveniente dai due utenti, realizzano la *fase di mirroring*, come analizzato nel Capitolo 4. Oltre all'inserimento di regole nella *flow table* dello switch, il controller procede ad avviare l'analisi del DPI, mantenendola attiva per 60 secondi, grazie ad un timer apposito, per poi concluderla richiamando la funzione **self.end_analysis**. È inoltre presente una variabile di timeout, grazie alla quale verranno rimosse le regole rimaste inattive per il tempo indicato, ovvero 50 secondi

```
#VARIABILE GLOBALE PRIORITA' DELLA REGOLA
global h
h = h+1

#REGOLE REDIREZIONAMENTO PACCHETTI
    #AL DPI E ALLA DESTINAZIONE
actions = []

if pkt_ipv4 and (in_port==1 or in_port==2 or in_port==8) :
   timeout = 50
   prio = h

   if in_port==1 or ( in_port==8 and
                      pkt_ipv4.dst == '10.10.10.1' ) :
      if pkt_ipv4.proto==6 :
         self.logger.debug("**** TCP pkt from VMU1
```

```
                    steered to DPI and port 8 ****")
            match = parser.OFPMatch(in_port = 1,
                        eth_type = eth.ethertype, ip_proto=6)
            actions = [parser.OFPActionOutput(3),
                        parser.OFPActionOutput(8)]
            self.add_flow(datapath, prio, match, actions,
                        timeout=timeout)

            self.logger.debug("**** TCP pkt to VMU1
                          steered to DPI and VMU1 ****")
            match = parser.OFPMatch(in_port = 8,
                        ipv4_dst = '10.10.10.1',
                        eth_type = eth.ethertype, ip_proto=6)
            actions = [parser.OFPActionOutput(3),
                        parser.OFPActionOutput(1)]
            self.add_flow(datapath, prio, match, actions,
                        timeout=timeout)

        elif pkt_ipv4.proto==17 :
            self.logger.debug("**** UDP pkt from VMU1
                           steered to DPI and port 8 ****")
            match = parser.OFPMatch(in_port = 1,
                        eth_type = eth.ethertype, ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                        parser.OFPActionOutput(8)]
            self.add_flow(datapath, prio, match, actions,
                        timeout=timeout)

            self.logger.debug("**** UDP pkt to VMU1
                          steered to DPI and VMU1 ****")
            match = parser.OFPMatch(in_port = 8,
                        ipv4_dst = '10.10.10.1',
                        eth_type = eth.ethertype, ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                        parser.OFPActionOutput(1)]
            self.add_flow(datapath, prio, match, actions,
                        timeout=timeout)

    elif in_port==2 or ( in_port==8 and
                        pkt_ipv4.dst == '10.10.10.2' ) :
        if pkt_ipv4.proto==6 :
            self.logger.debug("**** TCP pkt from VMU2
                          steered to DPI and port 8 ****")
            match = parser.OFPMatch(in_port = 2,
                        eth_type = eth.ethertype, ip_proto=6)
```

```
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(8)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)
            self.logger.debug("**** TCP pkt to VMU2
                              steered to DPI and VMU2 ****")
            match = parser.OFPMatch(in_port = 8,
                       ipv4_dst = '10.10.10.2',
                       eth_type = eth.ethertype, ip_proto=6)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(2)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

        elif pkt_ipv4.proto==17 :
            self.logger.debug("**** UDP pkt from VMU2
                              steered to DPI and port 8 ****")
            match = parser.OFPMatch(in_port = 2,
                       eth_type = eth.ethertype, ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(8)]
            self.add_flow(datapath, prio, match, actions,
                       timeout=timeout)

            self.logger.debug("**** UDP pkt to VMU2
                              steered to DPI and VMU2 ****")
            match = parser.OFPMatch(in_port = 8,
                       ipv4_dst = '10.10.10.2',
                       eth_type = eth.ethertype, ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(2)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

    #AVVIO nDPI
    self.logger.debug("[TIMER (%s)]",
        datetime.datetime.fromtimestamp(
            time.time()).strftime('%Y-%m-%d %H:%M:%S'))
    self.logger.debug("**start ndpiReader")
    os.chdir("/home/green/nDPI_Tool/nDPI/example")
    os.system("sudo ./ndpiReader -i s1-eth3 -v 2 -j
        /home/green/dpi-output/l2-test/netsoft-s1-eth3.json
        &")
    self.logger.debug("*nDPI ATTIVO*")
```

```
        #ATTESA DI 60 SECONDI E FINE ANALISI
        t=Timer(60.0, self.end_analysis)
        t.start()

    else :
        self.logger.debug("******FLUSSO NON AMMESSO******")
        match = parser.OFPMatch(in_port = in_port,
                    eth_type = eth.ethertype, eth_dst=dst)
        actions = []
        timeout=0
        prio=1
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, prio, match, actions,
                msg.buffer_id, timeout)
            return
        else:
            self.add_flow(datapath, prio, match, actions,
                timeout=timeout)
```

- Se il timer, presente nel precedente frammento di codice, viene avviato, dopo 60 secondi verrà richiamata la seguente funzione, in modo da arrestare l'analisi del DPI

```
def end_analysis(self):
    #STOP nDPI
    self.logger.debug("** STOP ndpiReader")
    os.system('sudo kill -2 $(pgrep ndpiReader)')
    time.sleep(0.10)
```

- Sempre nella stessa funzione, si procede alla lettura dell'esito dell'analisi, memorizzando gli indirizzi ed il numero di porta dei flussi TCP o UDP, relativi ai due utenti presenti nella rete

```
#LETTURA OUTPUT JSON (CLASSIFICAZIONE nDPI)
json_data = open('
    /home/green/dpi-output/l2-test/netsoft-s1-eth3.json')
data = json.load(json_data)
list_of_flows = data["known.flows"]
for i in list_of_flows :
    if ( i['protocol'] == "TCP" or
         i['protocol'] == "UDP" ) and (
         i["host_a.name"] == "10.10.10.1" or
         i["host_b.name"] == "10.10.10.1" or
         i["host_a.name"] == "10.10.10.2" or
```

```
        i["host_b.name"] == "10.10.10.2" ) :
      host_a = i["host_a.name"]
      host_b = i["host_b.name"]
      port_a = i["host_a.port"]
      port_b = i["host_n.port"]
```

- Nel caso sia effettivamente presente un flusso generato da uno dei due utenti, si procede all'avvio della fase di congestionamento. Vengono utilizzate due variabili globali per facilitare il riconoscimento dei casi di congestionamento e non congestionamento. Tali variabili rappresentano la possibile presenza di un'entità esterna e separata, destinata al monitoraggio, che informi il controller dello stato della rete

```
#AVVIO CASO DI CONGESTIONAMENTO
global bu
global ru

if host_a == "10.10.10.1" or host_b == "10.10.10.1" :
   if bu == 0 :
      bu=1
      self.congested_case()
   else :
      self.logger.debug("**** WARNING: BUSINESS USER'S
                         FLOW ALREADY ACTIVE ****")

if host_a == "10.10.10.2" or host_b == "10.10.10.2" :
   if ru == 0 :
      ru=1
      self.congested_case()
   else :
      self.logger.debug("**** WARNING: RESIDENCE USER'S
                         FLOW ALREADY ACTIVE ****")
```

- Successivamente, se la rete non è congestionata, viene avviato un timer di un secondo, al termine del quale viene chiamata la funzione relativa al caso di non congestione

```
nc=Timer(1.0, self.non_congested_case )
if (ru == 0 or bu == 0) :
   nc.start()

json_data.close()
```

- Infine il resto del codice è relativo alla definizione delle funzioni degli stati di congestione e non congestione del sistema che aggiungono, alla *flow-table* dello switch, regole sempre più prioritarie per poi eliminare le precedenti grazie al timeout impostato

```
def congested_case(self):
 ...

def non_congested_case(self):
 ...
```

Il controller è ora completo. Sono però necessarie ancora alcune operazioni preliminari per procedere al test del caso di studio.

5.1.3 Preparazione al test

Installazione nDPI

Per poter utilizzare la funzione del DPI durante il test, è necessario installare il software nDPI seguendo i seguenti semplici comandi:

- Si crea la cartella nDPI_Tool e vi si entra

```
mkdir nDPI_Tool
cd NDPI_Tool
```

- Si installa subversion

```
sudo apt-get install subversion
```

- Si esegue il download delle sorgenti

```
git clone https://github.com/ntop/nDPI
```

- Si entra nella cartella nDPI

```
cd nDPI
```

- Si esegue

```
./autogen.sh
make
```

Ora il DPI è pronto per essere richiamato dal codice del controller.

Installazione contatori

Per effettuare considerazioni più chiare e precise, è utile adoperare dei contatori durante il test della rete, i quali misureranno il numero di byte dei pacchetti inviati.

Si posizionano quindi i file **gettimestamp.c** e **rcvdatafw.sh**, entrambi consultabili nell'Appendice A.5 e A.6, nella cartella in cui è situato il codice della topologia Mininet.

Si compila dunque il file con estensione **.c** col seguente comando:

```
~$ gcc gettimestamp.c
```

Lo script con estensione **.sh** è invece da avviare con la seguente modalità:

```
./rcvdatafw.sh <OVS switch name> <sampling period in seconds>
```

Ad esempio, con:

```
./rcvdatafw.sh s1 1
```

I dati saranno raccolti dallo switch s1 ogni secondo. In questo modo, però, la misurazione effettuata viene mostrata solamente su schermo, visualizzando ogni secondo i contatori. Per redirezionare l'output dello schermo generato dallo script, è necessario utilizzare una pipeline (—) ed il comando tee. Ad esempio:

```
./rcvdatafw.sh s1 1 | tee contatori1.txt
```

In tal modo i risultati della misurazione saranno visibili sia sullo schermo, che su un file **contatori1.txt**, nel quale vengono aggiornati i dati.

Finite le misure, è sufficiente chiudere lo script per terminare il salvataggio dei dati su file.

Per poter essere utilizzati, questi dati necessitano di una elaborazione, affrontata al termine del test.

È ora possibile procedere col test della topologia.

5.1.4 Il Test

Definiti i codici della topologia, del controller ed effettuate le operazioni preliminari, si può procedere al test della rete. Tale test prevede l'avvio della funzione *iperf*, destinata a misurare le performance della connessione tra due host, dagli utenti VMU1 e VMU2 verso la destinazione H1, con un intervallo di tempo di ritardo di circa 70 secondi.

Innanzitutto si avvia la topologia precedentemente descritta tramite il comando:

```
~$ sudo python test-l2-scenario-tc.py 1
```

Successivamente la schermata di Mininet richiederà l'avvio del controller.

Si apre perciò una nuova scheda del terminale e si digita il seguente comando che richiama il codice del controller prescelto per questo test, ovvero quello mostrato in precedenza:

```
~$ cd /home/UTENTE/ryu &&
      ./bin/ryu-manager --verbose ryu/app/contrl2.py
```

Ottenendo:

```
loading app ryu/app/contrl2.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/contrl2.py of ControllerNetsoft
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ControllerNetsoft
CONSUMES EventOFPPacketIn
CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
PROVIDES EventOFPPacketIn TO {
    'ControllerNetsoft': set(['main'])}
PROVIDES EventOFPSwitchFeatures TO {
    'ControllerNetsoft': set(['config'])}
CONSUMES EventOFPSwitchFeatures
CONSUMES EventOFPPortDescStatsReply
CONSUMES EventOFPErrorMsg
CONSUMES EventOFPHello
CONSUMES EventOFPEchoRequest
connected socket:<eventlet.greenio.base.GreenSocket
    object at 0x7f399d081250> address:('127.0.0.1', 47339)
hello ev <ryu.controller.ofp_event.EventOFPHello
    object at 0x7f399d066850>
```

```
move onto config mode
EVENT ofp_event->ControllerNetsoft EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x8e06f1c3
OFPSwitchFeatures(auxiliary_id=0,capabilities=71,
    datapath_id=1,n_buffers=256,n_tables=254)
[TIMER (2015-05-12 10:32:37)]
**installo regole ARP
**installo regole ICMP
move onto main mode
```

Il controller si è ora connesso allo switch ed ha installato con successo le regole relative ai pacchetti ICMP e ARP.

Per visualizzare la lista delle regole attualmente installate sullo switch è possibile ricorrere, dal terminale, al comando:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Il risultato di tale richiesta è:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=149.727s, table=0, n_packets=14,
    n_bytes=4788, in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=149.714s, table=0, n_packets=16,
    n_bytes=5472, in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=149.72s, table=0, n_packets=16,
    n_bytes=5472, in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=149.707s, table=0, n_packets=16,
    n_bytes=5472, in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=150.577s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=150.577s, table=0, n_packets=0,
    n_bytes=0, priority=1,arp actions=NORMAL
cookie=0x0, duration=150.577s, table=0, n_packets=16,
    n_bytes=1460, priority=0
actions=CONTROLLER:65535
```

È possibile notare le due entry relative ad ARP e ICMP, l'ultima riguardante la connessione col controller e le quattro regole di DROP riguardanti l'*ARP storming*, presenti all'interno del codice della topologia.

Si avvia ora sull'host H1 della topologia il server dell'istruzione *iperf*, in modo da permettere la connessione da parte di client su altri hosts, inserendo sul terminale di H1 il comando:

```
~$ iperf -s
```

Dal terminale dell'utente VMU1, ovvero il Business User, si lancia il client (-c) della funzione *iperf*, con destinazione l'host H1 (con indirizzo IP 10.0.0.9), durata (-t) di 300 secondi e un intervallo di report su schermo (-i) di 10 secondi, in modo da monitorare al meglio il suo funzionamento:

```
~$ iperf -c 10.0.0.9 -t 300 -i 10
```

Il controller, avvenuta la connessione tra VMU1 e H1, mostrerà su schermo:

```
EVENT ofp_event->ControllerNetsoft EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:08 1
**** TCP pkt from VMU1 steered to DPI and port 8 ****
**** TCP pkt to VMU1 steered to DPI and VMU1 ****
[TIMER (2015-05-12 10:41:04)]
**start ndpiReader
*nDPI ATTIVO*
```

Il controller ha perciò ricevuto un messaggio di *packet-in* dallo switch col quale avverte della presenza di un pacchetto che cerca di transitare dalla porta 1 alla porta 8 dello switch stesso. Il controller installa quindi le regole di steering, in modo da permettere il flusso verso il Virtual Router (porta 8) e, in contemporanea, copiare ogni pacchetto anche sulla porta 3, ovvero quella connessa al DPI. Viene subito attivato il DPI che inizia ad analizzare il flusso.

Viene osservata nuovamente la tabella delle entries:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Col risultato:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=42.497s, table=0, n_packets=20407,
    n_bytes=1346870, idle_timeout=50, priority=18,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:3,output:1
cookie=0x0, duration=42.497s, table=0, n_packets=20470,
    n_bytes=487592348, idle_timeout=50, priority=18,
    tcp,in_port=1 actions=output:3,output:8
cookie=0x0, duration=548.295s, table=0, n_packets=29,
```

```
    n_bytes=9618, in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=548.282s, table=0, n_packets=30,
    n_bytes=9960, in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=548.288s, table=0, n_packets=29,
    n_bytes=9618, in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=548.275s, table=0, n_packets=30,
    n_bytes=9960, in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=549.145s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=549.145s, table=0, n_packets=6,
    n_bytes=252, priority=1,arp actions=NORMAL
cookie=0x0, duration=549.145s, table=0, n_packets=17,
    n_bytes=1534, priority=0, actions=CONTROLLER:65535
```

In questo caso, le prime due regole, sono le ultime inserite dal controller. Hanno entrambe un timeout in caso di inattività di 50 secondi, priorità 18 e sono di tipo tcp.

La prima considera i pacchetti entranti nella porta 8, ovvero il Virtual Router, aventi destinazione il VMU1 (indirizzo IP 10.10.10.1), con l'azione di permettere l'arrivo alla porta dell'host VMU1, ma anche l'invio dei pacchetti al DPI (porta 3). La seconda, invece, considera il flusso inverso, ovvero i pacchetti entranti dalla porta 1, il VMU1, verso la porta 8 del Virtual Router, permettendo tale percorso, inviando comunque una copia dei pacchetti al DPI per l'analisi (porta 3).

Successivamente, passati i 60 secondi di analisi, il terminale relativo al controller visualizzerà le seguenti informazioni:

```
** STOP ndpiReader
nDPI Memory statistics:
nDPI Memory (once):
91.02 KB
Flow Memory (per flow): 1.92 KB
Actual Memory:
1.56 MB
Peak Memory:
1.56 MB
[CONTROLLER - TIMER (2015-05-12 10:42:04)]
    Host 10.10.10.1:34060 is exchanging packets with Host
```

```
10.0.0.9:5001, via TCP
**** CONGESTED CASE at [TIMER (2015-05-12 10:42:04)]
**** STEERING BUSUSER FLOWS ****
**bu_c: 1/8 _ TCP pkt from VMU1 steered to WANA (LAN port) **
**bu_c: 2/8 _ UDP pkt from VMU1 steered to WANA (LAN port) **
**bu_c: 3/8 _ TCP pkt to VMU1 steered to WANA (WAN port) **
**bu_c: 4/8 _ UDP pkt to VMU1 steered to WANA (WAN port) **
**bu_c: 5/8 _ TCP pkt from WANA (WAN port) to destination **
**bu_c: 6/8 _ UDP pkt from WANA (WAN port) to destination **
**bu_c: 7/8 _ TCP pkt from WANA (LAN port) to VMU1 **
**bu_c: 8/8 _ UDP pkt from WANA (LAN port) to VMU1 **
```

Terminata l'analisi dei pacchetti, il DPI si ferma, indicando che l'host 10.10.10.1 (VMU1) sta scambiando pacchetti con l'host 10.0.0.9 (H1), col protocollo TCP.

Come previsto dall'algoritmo di gestione dei casi di congestionamento, la topologia viene considerata congestionata ed i flussi vengono redirezionati dal VMU1 al WANA, dal WANA alla destinazione (porta 8) e viceversa, sia per il protocollo TCP, che UDP.

```
**** NON CONGESTED CASE at [TIMER (2015-05-12 10:42:05)]
**** RESTORING BUSUSER FLOWS ****
**bu_nc: 1/4 _ TCP pkt from VMU1 to destination ****
**bu_nc: 2/4 _ UDP pkt from VMU1 to destination ****
**bu_nc: 3/4 _ TCP pkt to VMU1 ****
**bu_nc: 4/4 _ UDP pkt to VMU1 ****
```

Questa situazione si mantiene per un secondo, al termine del quale il controller si accorge della presenza dell'unico flusso di VMU1 (il Business User) e redireziona nuovamente i flussi, inserendo nuove regole che evitino il passaggio dei pacchetti dal WANA, permettendo quindi ora un percorso diretto dalla porta 1 (VMU1) alla porta 8 (il Virtual Router) e viceversa.

Viene osservata nuovamente la tabella delle entries, sempre col comando:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Ottenendo:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=5.89s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50, priority=19,udp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=5.89s, table=0, n_packets=448,
```

```
    n_bytes=29568, idle_timeout=50, priority=19,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=4.883s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50, priority=20,udp,
    in_port=8,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=5.89s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=19,udp,
    in_port=8,nw_dst=10.10.10.1 actions=output:5
cookie=0x0, duration=4.883s, table=0, n_packets=2432,
    n_bytes=160512, idle_timeout=50,priority=20,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=5.89s, table=0, n_packets=560,
    n_bytes=36960, idle_timeout=50,priority=19,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:5
cookie=0x0, duration=66.073s, table=0, n_packets=31944,
    n_bytes=2108312, idle_timeout=50,priority=18,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:3,output:1
cookie=0x0, duration=5.89s, table=0, n_packets=466,
    n_bytes=12098388, idle_timeout=50,priority=19,tcp,
    in_port=5 actions=output:8
cookie=0x0, duration=5.89s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=19,udp,
    in_port=5 actions=output:8
cookie=0x0, duration=4.883s, table=0, n_packets=2417,
    n_bytes=68086650, idle_timeout=50,priority=20,tcp,
    in_port=1 actions=output:8
cookie=0x0, duration=5.89s, table=0, n_packets=560,
    n_bytes=15802784, idle_timeout=50,priority=19,tcp,
    in_port=1 actions=output:4
cookie=0x0, duration=66.074s, table=0, n_packets=32009,
    n_bytes=693290810, idle_timeout=50,priority=18,tcp,
    in_port=1 actions=output:3,output:8
cookie=0x0, duration=4.883s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=20,udp,
    in_port=1 actions=output:8
cookie=0x0, duration=5.89s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=19,udp,
    in_port=1 actions=output:4
cookie=0x0, duration=575.272s, table=0, n_packets=11,
    n_bytes=3462,in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=575.253s, table=0, n_packets=14,
    n_bytes=4488,in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=575.266s, table=0, n_packets=12,
```

```
    n_bytes=3804,in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=575.245s, table=0, n_packets=14,
    n_bytes=4488,in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=575.643s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=575.643s, table=0, n_packets=8,
    n_bytes=336, priority=1,arp actions=NORMAL
cookie=0x0, duration=575.643s, table=0, n_packets=17,
    n_bytes=1534, priority=0, actions=CONTROLLER:65535
```

In questo momento sono presenti più regole relative al flusso tra VMU1 e H1, ma, osservandole, si nota una diversa priorità, che rende alcune di esse inutilizzate, per poi rimuoverle, dopo un periodo di inattività di 50 secondi, grazie alla configurazione dell'*idle_timeout*.

Le regole con priorità 18 sono le precedenti, quelle di invio al DPI. Le regole con priorità 19 sono quelle utilizzate nel secondo in cui il sistema era nello stato di congestione. Si può notare, nelle azioni, l'invio alla porta 4 e 5 del WANA. Le regole con priorità 20 sono quelle utilizzate attualmente, nello stato di non congestione. Nelle azioni, infatti, si osserva l'output diretto alla porta 8 (il Vitual Router) e viceversa alla porta 1 (VMU1).

Dopo circa 70 secondi dallo start dell'iperf su VMU1, si esegue la stessa istruzione anche sul terminale del VMU2, questa volta con una durata (-t) minore, ovvero di 200 secondi:

```
~$ iperf -c 10.0.0.9 -t 200 -i 10
```

Tale azione provocherà:

```
EVENT ofp_event->ControllerNetsoft EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:08 2
**** TCP pkt from VMU2 steered to DPI and port 8 ****
**** TCP pkt to VMU2 steered to DPI and VMU2 ****
[TIMER (2015-05-12 10:42:23)]
**start ndpiReader
*nDPI ATTIVO*
```

Il controller riceve un altro mmessaggio di *packet-in* dallo switch col quale avverte della presenza di un pacchetto che cerca di transitare dalla porta 2 alla porta 8 dello switch stesso. Il controller installa perciò le regole di steering, come fatto in precedenza per VMU1, in modo da permettere il flusso verso

il Virtual Router (porta 8) e copiare ogni pacchetto anche sulla porta 3, ovvero quella connessa al DPI. Viene attivato nuovamente il DPI che inizia ad analizzare il nuovo flusso.

Viene osservata nuovamente la tabella delle entries dello switch:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Ottenendo:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=61.018s, table=0, n_packets=50594,
    n_bytes=3491488, idle_timeout=50,priority=20,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=43.257s, table=0, n_packets=31719,
    n_bytes=2197394, idle_timeout=50,priority=21,tcp,
    in_port=8,nw_dst=10.10.10.2 actions=output:3,output:2
cookie=0x0, duration=43.257s, table=0, n_packets=33768,
    n_bytes=123833328, idle_timeout=50,priority=21,tcp,
    in_port=2 actions=output:3,output:8
cookie=0x0, duration=61.02s, table=0, n_packets=53227,
    n_bytes=629734638, idle_timeout=50,priority=20,tcp,
    in_port=1 actions=output:8
cookie=0x0, duration=628.066s, table=0, n_packets=41,
    n_bytes=13422,in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=628.053s, table=0, n_packets=40,
    n_bytes=13080,in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=628.059s, table=0, n_packets=39,
    n_bytes=12738,in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=628.046s, table=0, n_packets=39,
    n_bytes=12738,in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=628.916s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=628.916s, table=0, n_packets=16,
    n_bytes=672, priority=1,arp actions=NORMAL
cookie=0x0, duration=628.916s, table=0, n_packets=18,
    n_bytes=1608, priority=0, actions=CONTROLLER:65535
```

Sono state aggiunte le regole relative alla fase di mirroring dell'utente VMU2. Si può inoltre notare che le regole relative al flusso di VMU1 con priorità minore di 20 sono state rimosse per inattività.

Passati i 60 secondi di analisi, il terminale relativo al controller visualizzerà le seguenti informazioni:

```
** STOP ndpiReader
nDPI Memory statistics:
nDPI Memory (once):
91.02 KB
Flow Memory (per flow): 1.92 KB
Actual Memory:
1.56 MB
Peak Memory:
1.56 MB
[CONTROLLER - TIMER (2015-05-12 10:43:23)]
    Host 10.10.10.2:53875 is exchanging packets with Host
10.0.0.9:5001, via TCP
**** CONGESTED CASE at [TIMER (2015-05-12 10:43:23)]
**** STEERING BUSUSER FLOWS ****
**bu_c: 1/8 _ TCP pkt from VMU1 steered to WANA (LAN port) ****
**bu_c: 2/8 _ UDP pkt from VMU1 steered to WANA (LAN port) ****
**bu_c: 3/8 _ TCP pkt to VMU1 steered to WANA (WAN port) ****
**bu_c: 4/8 _ UDP pkt to VMU1 steered to WANA (WAN port) ****
**bu_c: 5/8 _ TCP pkt from WANA (WAN port) to destination ****
**bu_c: 6/8 _ UDP pkt from WANA (WAN port) to destination ****
**bu_c: 7/8 _ TCP pkt from WANA (LAN port) to VMU1 ****
**bu_c: 8/8 _ UDP pkt from WANA (LAN port) to VMU1 ****

**** STEERING RESUSER FLOWS ****
**ru_c: 1/8 _ TCP pkt from VMU2 steered to TC ****
**ru_c: 2/8 _ UDP pkt from VMU2 steered to TC ****
**ru_c: 3/8 _ TCP pkt to VMU2 steered to TC (2nd port) ****
**ru_c: 4/8 _ UDP pkt to VMU2 steered to TC (2nd port) ****
**ru_c: 5/8 _ TCP pkt from TC (2nd port) to destination ****
**ru_c: 6/8 _ UDP pkt from TC (2nd port) to destination ****
**ru_c: 7/8 _ TCP pkt from TC to VMU2 ****
**ru_c: 8/8 _ UDP pkt from TC to VMU2 ****
```

Anche in questo caso, dopo 60 secondi di analisi dei pacchetti, il DPI si ferma e indica che l'host 10.10.10.2 (VMU2) sta scambiando pacchetti con l'host 10.0.0.9 (H1), col protocollo TCP.

Il controller, accorgendosi però della presenza sia del flusso di VMU1 che di VMU2, considera la topologia congestionata ed i flussi vengono redirezionati dal VMU1 al WANA, dal WANA alla destinazione (porta 8), dal VMU2 al TC, dal TC alla destinazione e viceversa, sia per il protocollo TCP, che UDP.

A questo punto, viene osservata nuovamente la tabella delle entries:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Il risultato di tale operazione è:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=8,nw_dst=10.10.10.2 actions=output:7
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=8,nw_dst=10.10.10.1 actions=output:5
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=6,nw_dst=10.10.10.2 actions=output:2
cookie=0x0, duration=27.497s, table=0, n_packets=27143,
    n_bytes=1844862, idle_timeout=50,priority=22,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=27.497s, table=0, n_packets=29407,
    n_bytes=1994286, idle_timeout=50,priority=22,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:5
cookie=0x0, duration=105.495s, table=0, n_packets=61579,
    n_bytes=4273014, idle_timeout=50,priority=20,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=27.497s, table=0, n_packets=10479,
    n_bytes=819830, idle_timeout=50,priority=22,tcp,
    in_port=6,nw_dst=10.10.10.2 actions=output:2
cookie=0x0, duration=27.497s, table=0, n_packets=11294,
    n_bytes=873620, idle_timeout=50,priority=22,tcp,
    in_port=8,nw_dst=10.10.10.2 actions=output:7
cookie=0x0, duration=87.734s, table=0, n_packets=41239,
    n_bytes=2853094, idle_timeout=50,priority=21,tcp,
    in_port=8,nw_dst=10.10.10.2 actions=output:3,output:2
cookie=0x0, duration=27.497s, table=0, n_packets=27281,
    n_bytes=281767002, idle_timeout=50,priority=22,tcp,
    in_port=5 actions=output:8
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=5 actions=output:8
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
```

```
    in_port=2 actions=output:6
cookie=0x0, duration=27.497s, table=0, n_packets=10873,
    n_bytes=27037914, idle_timeout=50,priority=22,tcp,
    in_port=7 actions=output:8
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=7 actions=output:8
cookie=0x0, duration=27.497s, table=0, n_packets=14171,
    n_bytes=34762014, idle_timeout=50,priority=22,tcp,
    in_port=2 actions=output:6
cookie=0x0, duration=87.734s, table=0, n_packets=43976,
    n_bytes=170194352, idle_timeout=50,priority=21,tcp,
    in_port=2 actions=output:3,output:8
cookie=0x0, duration=27.5s, table=0, n_packets=29952,
    n_bytes=309756472, idle_timeout=50,priority=22,tcp,
    in_port=1 actions=output:4
cookie=0x0, duration=105.497s, table=0, n_packets=64105,
    n_bytes=763775738, idle_timeout=50,priority=20,tcp,
    in_port=1 actions=output:8
cookie=0x0, duration=27.497s, table=0, n_packets=0,
    n_bytes=0, idle_timeout=50,priority=22,udp,
    in_port=1 actions=output:4
cookie=0x0, duration=672.543s, table=0, n_packets=45,
    n_bytes=14790,in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=672.53s, table=0, n_packets=43,
    n_bytes=14106,in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=672.536s, table=0, n_packets=42,
    n_bytes=13764,in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=672.523s, table=0, n_packets=42,
    n_bytes=13764,in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=673.393s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=673.393s, table=0, n_packets=20,
    n_bytes=840, priority=1,arp actions=NORMAL
cookie=0x0, duration=673.393s, table=0, n_packets=39,
    n_bytes=33402, priority=0,actions=CONTROLLER:65535
```

La situazione attuale presenta nuove regole con priorità 22, che riguardano i flussi di VMU1 e VMU2 passanti da WANA e TC, che rendono inutilizzate le precedenti regole a priorità 20 (caso non congestionato di VMU1) e priorità

21 (traffico di VMU2 redirezionato al DPI e alla destinazione).

Dopo 50 secondi di inattività delle vecchie entries, esse verranno rimosse, infatti, circa dieci secondi prima della chiusura dell'iperf, osserviamo nuovamente la tabella dello switch:

```
~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

Ottenendo:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=136.103s, table=0, n_packets=124201,
    n_bytes=8417026, idle_timeout=50,priority=22,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=136.103s, table=0, n_packets=126465,
    n_bytes=8566450, idle_timeout=50,priority=22,tcp,
    in_port=8,nw_dst=10.10.10.1 actions=output:5
cookie=0x0, duration=136.103s, table=0, n_packets=47615,
    n_bytes=3642886, idle_timeout=50,priority=22,tcp,
    in_port=6,nw_dst=10.10.10.2 actions=output:2
cookie=0x0, duration=136.103s, table=0, n_packets=48430,
    n_bytes=3696676, idle_timeout=50,priority=22,tcp,
    in_port=8,nw_dst=10.10.10.2 actions=output:7
cookie=0x0, duration=136.103s, table=0, n_packets=125660,
    n_bytes=1459357384, idle_timeout=50,priority=22,tcp,
    in_port=5 actions=output:8
cookie=0x0, duration=136.103s, table=0, n_packets=50366,
    n_bytes=130672612, idle_timeout=50,priority=22,tcp,
    in_port=7 actions=output:8
cookie=0x0, duration=136.103s, table=0, n_packets=57962,
    n_bytes=145561276, idle_timeout=50,priority=22,tcp,
    in_port=2 actions=output:6
cookie=0x0, duration=136.106s, table=0, n_packets=128331,
    n_bytes=1487346854, idle_timeout=50,priority=22,tcp,
    in_port=1 actions=output:4
cookie=0x0, duration=781.149s, table=0, n_packets=45,
    n_bytes=14790,in_port=4,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=781.136s, table=0, n_packets=43,
    n_bytes=14106,in_port=6,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=781.142s, table=0, n_packets=42,
    n_bytes=13764,in_port=5,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=781.129s, table=0, n_packets=42,
```

```
    n_bytes=13764,in_port=7,
    dl_dst=ff:ff:ff:ff:ff:ff actions=drop
cookie=0x0, duration=781.999s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=781.999s, table=0, n_packets=32,
    n_bytes=1344, priority=1,arp actions=NORMAL
cookie=0x0, duration=781.999s, table=0, n_packets=39,
    n_bytes=33402, priority=0,actions=CONTROLLER:65535
```

Tutte le regole inutilizzate sono state rimosse, rimangono quindi solamente quelle TCP relative al caso congestionato di VMU1 e VMU2, quelle di ARP STORMING e le tre iniziali di ARP, ICMP e connessione al controller.

Visualizzazione dei terminali degli Host

Terminato il test, è possibile visualizzare il resoconto delle performance di banda delle connessioni direttamente dai terminali degli host.

Tali risultati, saranno visualizzabili in modo più chiaro grazie ai grafici realizzati nella prossima sezione.

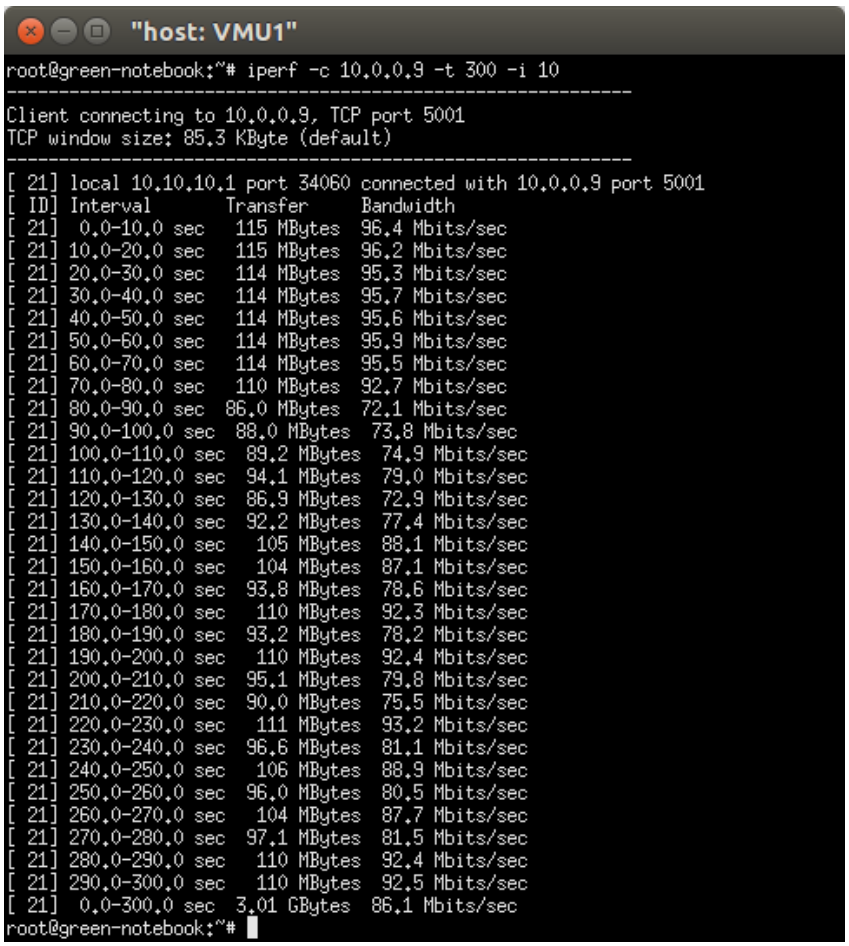

Figura 5.1: Terminale dell'host VMU1.

Nell'host VMU1, in Figura 5.1, si nota l'attacco subito alla banda da parte del flusso di VMU2, principalmente avvenuto dall'ottantesimo secondo in poi,

per recuperare in seguito parte della banda, grazie all'azione di shaping del TC.

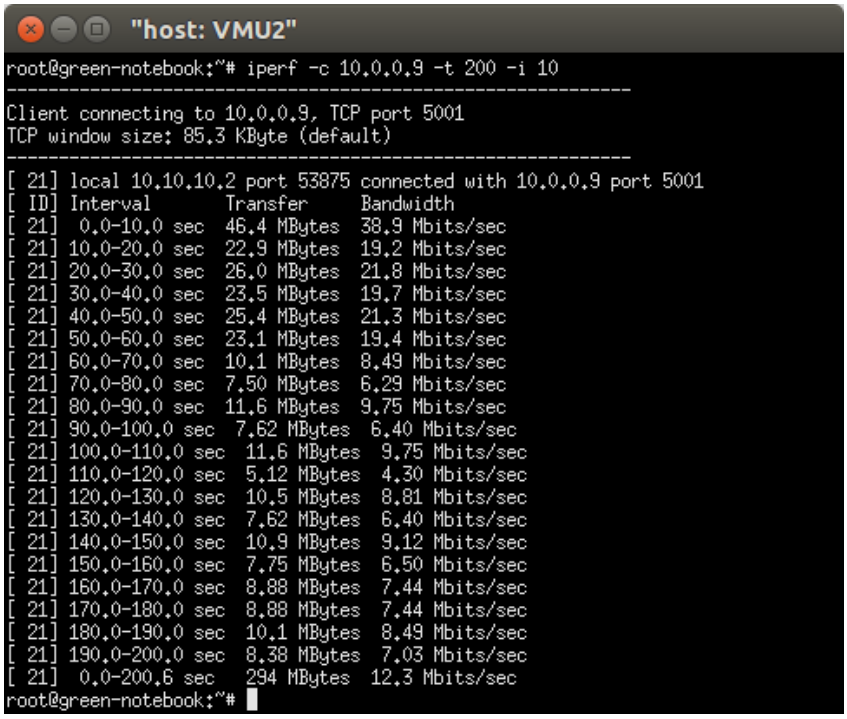

Figura 5.2: Terminale dell'host VMU2.

Nell'host VMU2, in Figura 5.2, si osserva il periodo iniziale in cui ha utilizzato maggiore banda, attaccando il flusso di VMU1. In seguito all'analisi e al redirezionamento al TC, la banda viene visibilmente ridotta sotto ai 10 Mbits/sec.

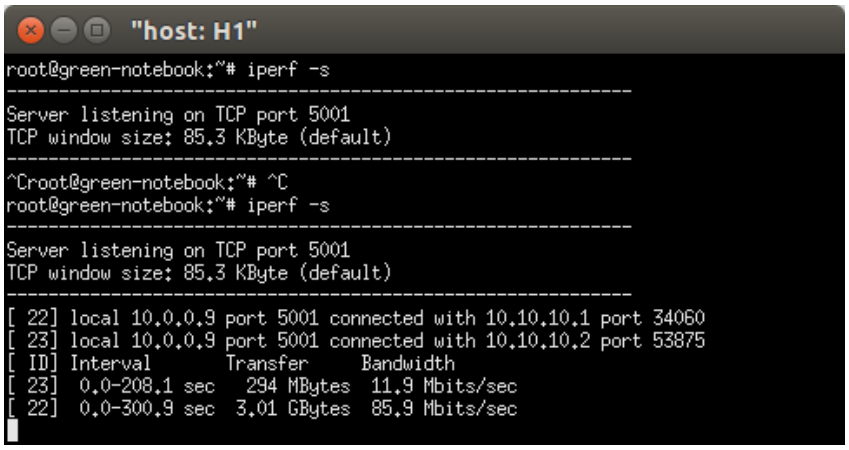

Figura 5.3: Terminale dell'host H1.

In Figura 5.3 è possibile osservare il report finale del test, mostrato dall'host H1.

5.1.5 Elaborazione della rilevazione dei contatori

Una volta giunti a conclusione del test e finito perciò il conteggio dei byte dei pacchetti inviati, è necessario scegliere quale dato sia di utilità, tra quelli raccolti. A tal scopo sono da selezionare i valori in grassetto nella Tabella 5.1:

s1eth1byterx	s1eth1bytetx	VMU1
s1eth2byterx	s1eth2bytetx	VMU2
s1eth3byterx	**s1eth3bytetx**	DPI
s1eth4byterx	**s1eth4bytetx**	WANA (LAN port)
s1eth5byterx	s1eth5bytetx	WANA (WAN port)
s1eth6byterx	**s1eth6bytetx**	TC
s1eth7byterx	s1eth7bytetx	TC (2nd port)
s1eth8byterx	**s1eth8bytetx**	VR

Tabella 5.1: Legenda del file prodotto dalla misurazione svolta dai contatori.

Si raccomanda di prestare attenzione a considerare rx e tx dal punto di vista dello switch. Per maggiore chiarezza si osservi la Figura 5.4:

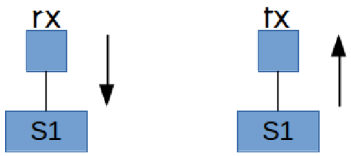

Figura 5.4: Visualizzazione di rx e tx secondo lo switch.

Per la corretta lettura da parte del software di elaborazione, ovvero un foglio di calcolo, nel file generato dal contatore sono stati sostituiti tutti i punti con le virgole, in modo tale da sistemare la colonna del tempo e rimuovere l'offset.

Siccome la lettura dei contatori viene eseguita in byte, è necessario effettuare qualche calcolo sui dati:

$$\Delta B \cdot 8 \cdot \frac{1536}{1500} \cdot \frac{1}{\Delta t} \tag{5.1}$$

Considerando $\Delta B = byte - byte_{\text{precedenti}}$

Successivamente, è necessario selezionare le colonne di interesse, per poi copiarle in un file chiamato **testL2.dat** , sostituendo questa volta tutte le virgole con i punti, per adattarle alla successiva elaborazione.

Si esegue quindi l'installazione del software destinato alla generazione dei grafici con:

```
sudo apt-get install gnuplot
```

Si avvia la shell di tale applicativo tramite il comando *gnuplot*.

Viene qui mostrato un esempio di realizzazione di grafico dalla shell:

```
set xlabel \TIME (sec)"                    (imposta variabile x)
set ylabel \THROUGHPUT (Mbit/s)"           (imposta variabile y)
plot \testl2.dat" u 1:($4/1000000.0)
     w lp t \DPI in"                       (es. di configurazione dato)
set logscale y                             (scala logaritmica nelle y)
set key right bottom                       (legenda in basso a destra)
set yrange [0.1:1000]                      (cambia il range delle y)
```

Il risultato delle misurazioni svolte durante il test è il seguente:

- Nel grafico in Figura 5.5 viene visualizzato il traffico generato da VMU1, VMU2 e quello entrante nel Virtual Router. Si nota perfettamente la perdita di banda di VMU1 all'avvio di VMU2 ed il ripristino della stessa durante la fase di congestionamento, una volta ridotta la banda del secondo User a circa 10 Mbits/sec

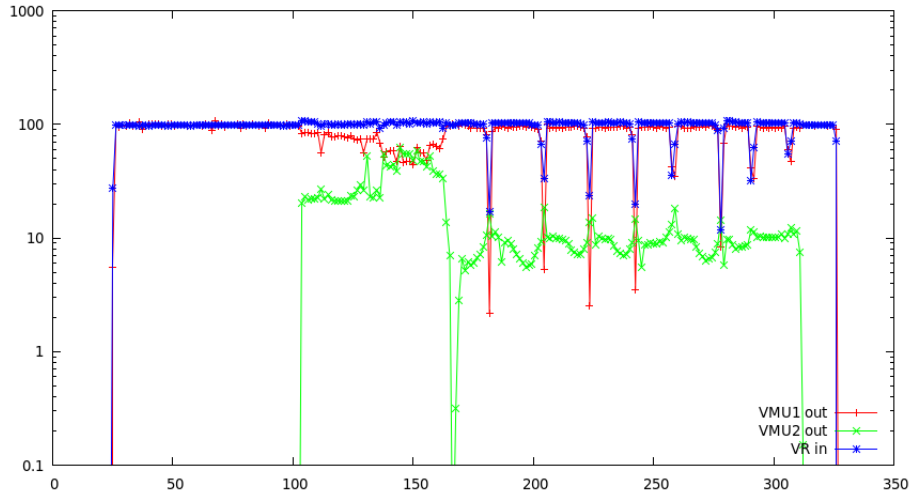

Figura 5.5: Traffico generato da VMU1, VMU2 ed entrante in VR.

- Nel grafico in Figura 5.6 viene mostrato il lavoro di identificazione del traffico, svolto dal DPI per 60 secondi ad ogni avvio di User.

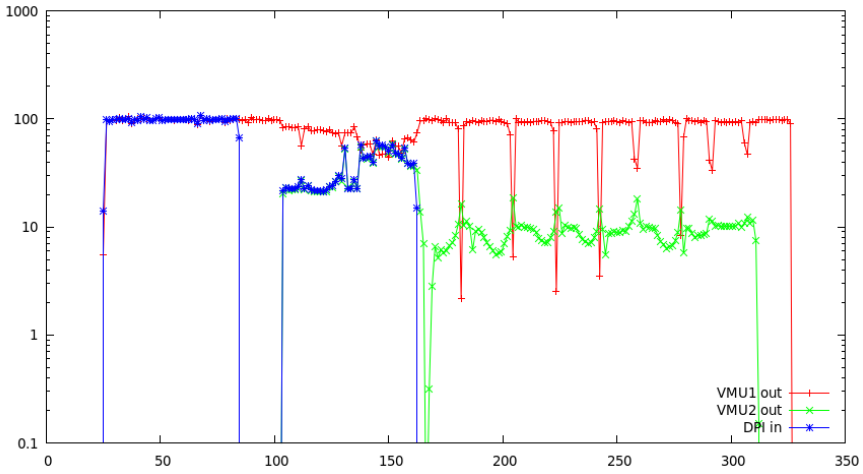

Figura 5.6: Traffico generato da VMU1, VMU2 ed analisi del DPI.

- Nel grafico in Figura 5.7 si nota la compressione di banda, del traffico di VMU2, eseguita dal TC.

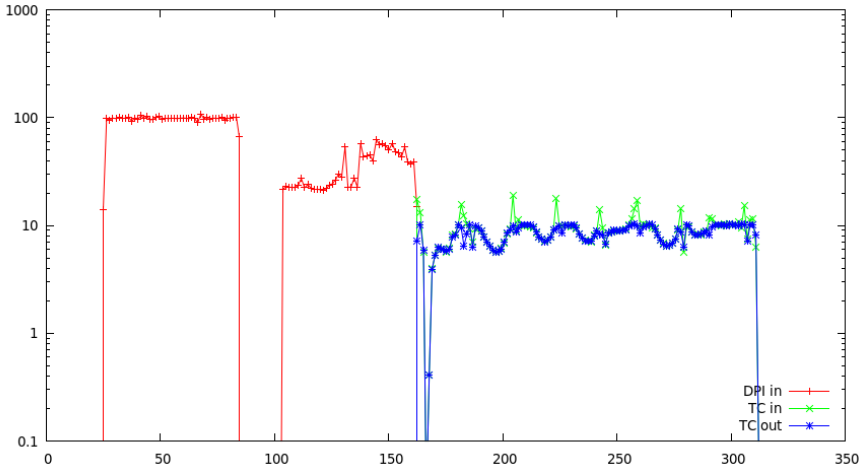

Figura 5.7: Traffico analizzato da DPI, entrante ed uscente da TC.

- Infine, nel grafico in Figura 5.8, è mostrato il lavoro svolto dalle funzioni di rete presenti nella topologia, ovvero il DPI, il WANA ed il TC.

  Si nota l'attivazione momentanea del WANA al termine della prima fase di identificazione del traffico, per poi procedere all'attivazione definitiva all'avvio della fase di congestionamento, assieme al TC.

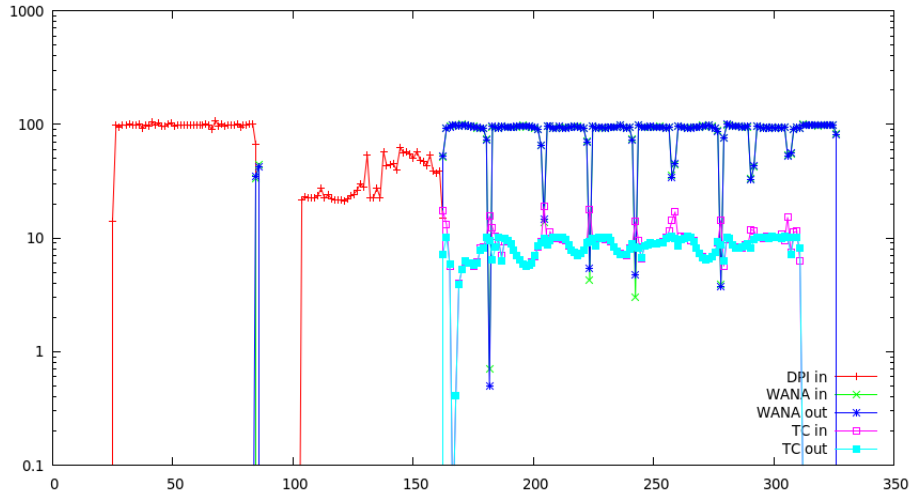

Figura 5.8: Traffico analizzato da DPI, entrante ed uscente da WANA e TC.

Termina qui l'analisi svolta per la topologia di livello L2. Di seguito è presentato lo studio svolto sulla topologia di livello L3.

5.2 Caso di studio L3

Come per il caso precedente, anche il caso di studio di livello L3, la cui topologia è già stata mostrata nel Capitolo 4 in Figura 4.8, viene realizzato emulando virtualmente la rete tramite Mininet ed azionando il controller Ryu appositamente realizzato.

Viene di seguito analizzato il codice Python relativo alla topologia del caso di studio.

5.2.1 Codice Mininet

Siccome la topologia è differente rispetto al caso precedente, è necessario effettuare qualche modifica al codice.

Si elencano ora le principali variazioni apportate al codice Python della rete di livello L2 per ottenere il caso L3. Esso è consultabile per intero nell'Appendice A.7 **test-l3-scenario.py**.

- Innanzitutto si aggiunge uno switch, denominato *S2*

```
s2 = net.addSwitch('s2')
s2.cmd('ovs-vsctl del-br ' + s2.name )
s2.cmd('ovs-vsctl add-br ' + s2.name )
s2.cmd('ovs-vsctl set Bridge '+ s2.name +
       ' stp_enable=false protocols=OpenFlow13' )
```

- Successivamente si aggiunge un ulteriore host, ovvero il gateway *GW*, si modificano le connessioni degli host ai due switch e si impostano gli indirizzi MAC relativi alle nuove interfacce

```
info("\n*** Creating GateWay *** \n")
gw = net.addHost('GW')
...

net.addLink(vmu1, s1, bw=100)
net.addLink(vmu2, s1, bw=100)
net.addLink(dpi, s1, bw=100)
net.addLink(dpi, s2, bw=100)
net.addLink(wana, s1, bw=100)
net.addLink(wana, s2, bw=100)
net.addLink(tc, s1, bw=100)
net.addLink(tc, s2, bw=100)
net.addLink(gw, s1, bw=100)
```

```
net.addLink(gw, s2, bw=100)
net.addLink(vr, s2, bw=100)
net.addLink(vr, h1, bw=100)

...
dpi.setMAC("00:00:00:00:00:04", dpi.name + "-eth1")
...
gw.setMAC("00:00:00:00:00:09", gw.name + "-eth0")
gw.setMAC("00:00:00:00:00:0A", gw.name + "-eth1")
```

- Si procede quindi all'aggiunta delle interfacce anche per il secondo switch, come fatto per il primo, per poi configurarlo secondo il protocollo OpenFlow 1.3 ed impostarne la connessione al controller

```
for intf in s2.intfs.values():
    s2.cmd('ovs-vsctl add-port ' + s2.name + ' %s' % intf)
    print "Eseguito comando: ovs-vsctl add-port s2 ", intf
...

...
s2.cmd('ovs-vsctl set bridge ' + s2.name +
       ' protocols=OpenFlow13')

...
s2.cmd('ovs-vsctl set-controller ' +  s2.name +
       ' tcp:127.0.0.1:6633')
```

- Data la diversità della topologia, in questo caso non è necessario impostare delle regole che prevengano l'*ARP storming*. Le linee di codice riguardanti tali regole vengono perciò rimosse

- Infine si correggono le configurazioni di indirizzo IP e default gateway degli host. Per tale modifica, si consiglia di consultare direttamente il codice presente nell'appendice.

5.2.2 Codice Controller Ryu

Il codice Python del controller Ryu, il cui comportamento è stato descritto nel Capitolo 4, è simile a quello precedentemente mostrato per il caso L2 ed è anch'esso consultabile interamente nell'Appendice A.8 **contrl3.py**. Tale codice è stato realizzato solamente per gestire il traffico TCP, per implementare

anche il protocollo UDP è sufficiente duplicare le regole, modificando in esse il parametro **ip_proto**.

Si riportano di seguito le principali differenze di tale codice rispetto a quello del caso L2:

- Innanzitutto è presente la definizione di una nuova funzione, destinata a fornire il *data path* appartenente allo switch di cui viene inserito il nome come argomento nella chiamata della funzione stessa. Questa aggiunta è di fondamentale importanza, in una topologia con più switch, perchè permette di inserire regole specifiche per ogni switch

```
def connectionForBridge(self, bridge):
    return self.connections_name_dpid[bridge]
```

- Sarà perciò necessario specificare in quale switch inserire le regole, prima di procedere alla funzione *add_flow*. Si utilizza per tale scopo il seguente codice:

```
dp1 = self.connectionForBridge("s1")
dp2 = self.connectionForBridge("s2")
```

- Il resto del codice riprende quasi del tutto quello visto in precedenza per il caso L2, ovviamente con la presenza di più regole, in parte destinate anche al secondo switch presente nella topologia. Si consiglia perciò di consultare l'intero codice nell'appendice.

Si considera già installato l'analizzatore DPI, come mostrato in precedenza. Per lo studio di questo caso, a differenza del precedente, non verranno utilizzati i contatori.
È ora possibile procedere col test della topologia L3.

5.2.3 Test del caso L3

Una volta definiti i codici della topologia e del controller, si procede al test della rete. Come per il precedente, questo test prevede l'avvio della funzione *iperf* dagli utenti VMU1 e VMU2 verso la destinazione H1, con un intervallo di tempo di ritardo di circa 70 secondi.

Innanzitutto si avvia la topologia del caso L3 tramite il comando:

```
~$ sudo python test-l3-scenario.py 1
```

Successivamente verrà richiesto, dalla schermata di Mininet, l'avvio del controller.

Si apre quindi una nuova scheda del terminale e si digita il seguente comando che richiama il codice del controller prescelto per questo test, ovvero quello mostrato in precedenza:

```
~$ cd /home/UTENTE/ryu &&
      ./bin/ryu-manager --verbose ryu/app/contrl3.py
```

Ottenendo:

```
loading app ryu/app/contrl3.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/contrl3.py of
    BasicOpenStackL3Controller

...

   move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
EVENT ofp_event->BasicOpenStackL3Controller EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath
    object at 0x7efbf4092850>

...

[TIMER (2015-09-12 16:17:58)] [SC-HANDLER] Switch s2
    registered on the controller, datapath.id  =  2
**Installo regole ARP
**Installo regole ICMP
move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
EVENT ofp_event->BasicOpenStackL3Controller EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath
    object at 0x7efbf40928d0>
[TIMER (2015-09-12 16:17:58)] [SC-HANDLER] Switch s1
    registered on the controller, datapath.id  =  1
**Installo regole ARP
**Installo regole ICMP
```

Il controller si è ora connesso ai due switch ed ha installato, ad entrambi, le regole relative ai pacchetti ICMP e ARP.

Per visualizzare la lista delle regole attualmente installate sugli switch è possibile ricorrere, dal terminale, al comando:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Il risultato di tale richiesta è, per lo switch S1:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=5.642s, table=0, n_packets=0,
    n_bytes=0, priority=1,arp actions=NORMAL
cookie=0x0, duration=5.642s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=5.642s, table=0, n_packets=2,
    n_bytes=128, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=5.212s, table=0, n_packets=0,
    n_bytes=0, priority=1,arp actions=NORMAL
cookie=0x0, duration=5.212s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=5.212s, table=0, n_packets=1,
    n_bytes=70, priority=0 actions=CONTROLLER:65535
```

È possibile notare, per ogni switch, le due entry relative ad ARP e ICMP, e l'ultima riguardante la connessione col controller. A differenza del caso precedente, non sono presenti le quattro regole di DROP riguardanti l'*ARP storming*, data l'assenza all'interno del codice della topologia.

Si avvia ora sull'host H1 della topologia il server dell'istruzione *iperf*, in modo da permettere la connessione tramite client da parte di altri hosts, inserendo sul terminale di H1 il comando:

```
~$ iperf -s
```

Esattamente come nel test precedente, dal terminale dell'utente VMU1, ovvero il Business User, si lancia il client (-c) della funzione *iperf*, con destinazione l'host H1 (con indirizzo IP 10.0.0.9), durata (-t) di 300 secondi e un intervallo di report su schermo (-i) di 10 secondi, in modo da monitorare al meglio il suo funzionamento:

```
~$ iperf -c 10.0.0.9 -t 300 -i 10
```

Il controller, avvenuta la connessione tra VMU1 e H1, mostrerà su schermo:

```
EVENT ofp_event->BasicOpenStackL3Controller EventOFPPacketIn
[PACKET-IN]  (2015-09-12 14:10:20) in port: 1 , datapath id: 1
[PKT-HANDLER]  (2015-09-12 14:10:20)
    Number of detected protocols: 1
[PKT-HANDLER] (2015-09-12 14:10:20)
    PacketIn from DPID = 1 - in_port=1 dl_type=2048
        eth_src=00:00:00:00:00:01
eth_dst=00:00:00:00:00:09 [PKT-HANDLER] (2015-09-12 14:10:20)
    TCP packets detected
[PKT-HANDLER]  (2015-09-12 14:10:20) CHECKING protocol detected
    ip_proto=6 ip_src=10.10.10.1 ip_dst=10.0.0.9 inet_PROTO=6
[TIMER (2015-09-12 14:10:20)]
**start ndpiReader
*nDPI ATTIVO*
```

Il controller ha perciò ricevuto un messaggio di *packet-in* dallo switch **S1** col quale avverte della presenza di un pacchetto che cerca di transitare dalla porta 1 dello switch stesso, con protocollo TCP. Il controller installa quindi le regole di steering, in modo da redirezionare il flusso uscente verso la porta 3 dello switch S1, ovvero quella connessa al DPI, ed il flusso entrante verso la porta 1 di S2. Viene subito attivato il DPI che inizia ad analizzare il flusso.

Viene osservata nuovamente la tabella delle entries:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Col risultato, per lo switch S1:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=11.926s, table=0, n_packets=7269,
    n_bytes=110910058, idle_timeout=40, priority=2,tcp,
    in_port=1,nw_src=10.10.10.1,nw_dst=10.0.0.9
    actions=set_field:00:00:00:00:00:03->eth_dst,output:3
cookie=0x0, duration=11.926s, table=0, n_packets=7258,
    n_bytes=479036, idle_timeout=40, priority=2,tcp,
    in_port=3,nw_src=10.0.0.9,nw_dst=10.10.10.1
    actions=output:1
cookie=0x0, duration=32.935s, table=0, n_packets=4,
    n_bytes=168, priority=1,arp actions=NORMAL
cookie=0x0, duration=32.935s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
```

```
cookie=0x0, duration=32.935s, table=0, n_packets=4,
    n_bytes=256, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=11.261s, table=0, n_packets=7258,
    n_bytes=479036, idle_timeout=40, priority=2,tcp,
    in_port=5,nw_src=10.0.0.9 ,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:04->eth_dst,output:1
cookie=0x0, duration=11.261s, table=0, n_packets=7268,
    n_bytes=110891168, idle_timeout=40, priority=2,tcp,
    in_port=1,nw_src=10.10.10.1,nw_dst=10.0.0.9
    actions=output:5
cookie=0x0, duration=32.505s, table=0, n_packets=4,
    n_bytes=168, priority=1,arp actions=NORMAL
cookie=0x0, duration=32.505s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=32.505s, table=0, n_packets=1,
    n_bytes=70, priority=0 actions=CONTROLLER:65535
```

In questo caso, in entrambi gli switch, le prime due regole sono le ultime inserite dal controller. Hanno entrambe un timeout in caso di inattività di 40 secondi, priorità 2 e sono di tipo tcp.

Nello switch S1, la prima considera i pacchetti entranti nella porta 1, il VMU1, aventi destinazione H1 (indirizzo IP 10.0.0.9), direzionandoli alla porta 3 del medesimo switch, procedendo prima ad un cambio di indirizzo MAC di destinazione. La seconda regola considera il flusso inverso, destinando alla porta 1 dello switch S1 tutti i pacchetti uscenti dal DPI (porta 3), con destinazione VMU1 (indirizzo IP 10.10.10.1).

Nello switch S2, invece, la prima regola considera il flusso proveniente da H1 (porta 5 di S2 ed indirizzo IP 10.0.0.9), direzionandolo alla porta 1 del medesimo switch, sempre modificandone prima l'indirizzo MAC di destinazione. La seconda regola si occupa del flusso inverso, ovvero quello diretto ad H1, destinandolo alla porta 5 di S2.

Successivamente, passati i 60 secondi di analisi, il terminale relativo al controller visualizzerà le seguenti informazioni:

```
** STOP ndpiReader

nDPI Memory statistics:
        nDPI Memory (once):      91.02 KB
```

```
        Flow Memory (per flow):  1.92 KB
        Actual Memory:           1.56 MB
        Peak Memory:             1.56 MB
[CONTROLLER - TIMER (2015-09-12 14:11:20)]
    Host 10.10.10.1:55488 is exchanging packets
    with Host 10.0.0.9:5001, via TCP
**** CONGESTED CASE at [TIMER (2015-09-12 14:11:20)]
**** STEERING BUSUSER FLOWS ****
**bu_c: 1 _ TCP pkt from VMU1 steered to WANA (LAN port) **
**bu_c: 2 _ TCP pkt from WANA (LAN port) to VMU1 **
**bu_c: 3 _ TCP pkt to VMU1 steered to WANA (WAN port) **
**bu_c: 4 _ TCP pkt from WANA (WAN port) to destination **
```

Terminata l'analisi dei pacchetti, il DPI si ferma, indicando che l'host 10.10.10.1 (VMU1) sta scambiando pacchetti con l'host 10.0.0.9 (H1), col protocollo TCP.

Come previsto dall'algoritmo di gestione dei casi di congestionamento, la topologia viene considerata momentaneamente congestionata ed i flussi vengono redirezionati dal VMU1 al WANA, dal WANA alla destinazione (porta 8) e viceversa.

```
**** NON CONGESTED CASE at [TIMER (2015-09-12 14:11:21)]
**** RESTORING BUSUSER FLOWS ****
**bu_nc: 1 _ TCP pkt from VMU1 to destination ****
**bu_nc: 2 _ TCP pkt from VMU1 to destination ****
**bu_nc: 3 _ TCP pkt to VMU1 ****
**bu_nc: 4 _ TCP pkt to VMU1 ****
```

Questa situazione si mantiene per un secondo, al termine del quale il controller si accorge della presenza dell'unico flusso di VMU1 (il Business User) e redireziona nuovamente i flussi, inserendo nuove regole che evitino il passaggio dei pacchetti dal WANA, permettendo quindi ora il loro passaggio tramite il Gateway GW, da esso al Virtual Router VR e viceversa.

Viene osservata nuovamente la tabella delle entries, sempre col comando:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Ottenendo, per lo switch S1:

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=5.524s, table=0, n_packets=2717,
    n_bytes=179322, idle_timeout=40, priority=4,tcp,
    in_port=6,nw_dst=10.10.10.1 actions=output:1
```

```
cookie=0x0, duration=5.524s, table=0, n_packets=3237,
    n_bytes=71956250, idle_timeout=40, priority=4,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:09->eth_dst,
    output:6
cookie=0x0, duration=6.532s, table=0, n_packets=568,
    n_bytes=37488, idle_timeout=40, priority=3,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=6.532s, table=0, n_packets=593,
    n_bytes=11990930, idle_timeout=40, priority=3,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:05->eth_dst,
    output:4
cookie=0x0, duration=66.698s, table=0, n_packets=7269,
    n_bytes=110910058, idle_timeout=40, priority=2,tcp,
    in_port=1,nw_src=10.10.10.1,nw_dst=10.0.0.9
    actions=set_field:00:00:00:00:00:03->eth_dst,output:3
cookie=0x0, duration=66.698s, table=0, n_packets=7258,
    n_bytes=479036, idle_timeout=40, priority=2,tcp,
    in_port=3,nw_src=10.0.0.9,nw_dst=10.10.10.1
    actions=output:1
cookie=0x0, duration=87.862s, table=0, n_packets=12,
    n_bytes=504, priority=1,arp actions=NORMAL
cookie=0x0, duration=87.862s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=87.862s, table=0, n_packets=4,
    n_bytes=256, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
cookie=0x0, duration=4.703s, table=0, n_packets=2184,
    n_bytes=48778120, idle_timeout=40, priority=4,tcp,
    in_port=4 actions=output:5
cookie=0x0, duration=4.703s, table=0, n_packets=2688,
    n_bytes=177408, idle_timeout=40, priority=4,tcp,
    in_port=5,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:0a->eth_dst,output:4
cookie=0x0, duration=5.711s, table=0, n_packets=583,
    n_bytes=12097422, idle_timeout=40, priority=3,tcp,
    in_port=2 actions=output:5
cookie=0x0, duration=5.711s, table=0, n_packets=592,
    n_bytes=39072, idle_timeout=40, priority=3,tcp,
    in_port=5,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:06->eth_dst,output:2
cookie=0x0, duration=65.876s, table=0, n_packets=31036,
    n_bytes=2048384, idle_timeout=40, priority=2,tcp,
    in_port=5,nw_src=10.0.0.9,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:04->eth_dst,output:1
```

```
cookie=0x0, duration=65.876s, table=0, n_packets=31596,
    n_bytes=697245552, idle_timeout=40, priority=2,tcp,
    in_port=1,nw_src=10.10.10.1,nw_dst=10.0.0.9
    actions=output:5
cookie=0x0, duration=87.342s, table=0, n_packets=10,
    n_bytes=420, priority=1,arp actions=NORMAL
cookie=0x0, duration=87.342s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=87.342s, table=0, n_packets=3,
    n_bytes=230, priority=0 actions=CONTROLLER:65535
```

In questo momento sono presenti più regole relative al flusso tra VMU1 e H1, ma, osservandole, si nota una diversa priorità, che rende alcune di esse inutilizzate, per poi rimuoverle, dopo un periodo di inattività di 40 secondi, grazie alla configurazione dell'*idle_timeout*.

Le regole con priorità 2 sono le precedenti, quelle di invio al DPI. Le regole con priorità 3 sono quelle utilizzate nel secondo in cui il sistema era nello stato di congestione. Si può notare, nelle azioni, l'invio alla porta relativa al WANA (4 di S1 e 2 di S2). Le regole con priorità 4 sono quelle utilizzate attualmente, nello stato di non congestione. Nelle azioni, infatti, si osserva l'output al Gateway (porta 6 di S1 per il traffico uscente e porta 4 di S2 per quello entrante).

COme nel caso precedente, dopo circa 70 secondi dallo start dell'iperf su VMU1, si esegue la stessa istruzione anche sul terminale del VMU2, questa volta con una durata (-t) minore, ovvero di 200 secondi:

```
~$ iperf -c 10.0.0.9 -t 200 -i 10
```

Tale azione provocherà:

```
EVENT ofp_event->BasicOpenStackL3Controller EventOFPPacketIn
[PACKET-IN]  (2015-09-12 14:11:37) in port: 2 , datapath id: 1
[PKT-HANDLER]  (2015-09-12 14:11:37)
    Number of detected protocols: 1
[PKT-HANDLER] (2015-09-12 14:11:37)
    PacketIn from DPID = 1 - in_port=2 dl_type=2048
    eth_src=00:00:00:00:00:02 eth_dst=00:00:00:00:00:09
[PKT-HANDLER] (2015-09-12 14:11:37) TCP packets detected
[PKT-HANDLER]  (2015-09-12 14:11:37)
    CHECKING protocol detected - ip_proto=6
    ip_src=10.10.10.2 ip_dst=10.0.0.9 inet_PROTO=6
[TIMER (2015-09-12 14:11:37)]
```

```
**start ndpiReader
*nDPI ATTIVO*
```

Il controller riceve un altro mmessaggio di *packet-in* dallo switch col quale avverte della presenza di un pacchetto che cerca di transitare dalla porta 2 di S1. Il controller installa perciò le regole di steering, come fatto in precedenza per VMU1, in modo da redirezionare il flusso uscente verso la porta 3 dello switch S1, ovvero quella connessa al DPI, ed il flusso entrante verso la porta 1 di S2. Viene attivato nuovamente il DPI che inizia ad analizzare il nuovo flusso.

Viene osservata nuovamente la tabella delle entries dello switch:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Ottenendo, per lo switch S1:

```
cookie=0x0, duration=4.538s, table=0, n_packets=2074,
    n_bytes=6335804, idle_timeout=40, priority=5,tcp,
    in_port=2,nw_src=10.10.10.2,nw_dst=10.0.0.9
    actions=set_field:00:00:00:00:00:03->eth_dst,output:3
cookie=0x0, duration=4.535s, table=0, n_packets=1908,
    n_bytes=125936, idle_timeout=40, priority=5,tcp,
    in_port=3,nw_src=10.0.0.9,nw_dst=10.10.10.2
    actions=output:2
cookie=0x0, duration=54.064s, table=0, n_packets=29432,
    n_bytes=1942716, idle_timeout=40, priority=4,tcp,
    in_port=6,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=54.064s, table=0, n_packets=30058,
    n_bytes=642748444, idle_timeout=40, priority=4,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:09->eth_dst,
    output:6
cookie=0x0, duration=137.247s, table=0, n_packets=18,
    n_bytes=756, priority=1,arp actions=NORMAL
cookie=0x0, duration=137.247s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=137.247s, table=0, n_packets=5,
    n_bytes=330, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
cookie=0x0, duration=4.635s, table=0, n_packets=2982,
    n_bytes=196820, idle_timeout=40, priority=5,tcp,
    in_port=5,nw_src=10.0.0.9 ,nw_dst=10.10.10.2
    actions=set_field:00:00:00:00:00:04->eth_dst,output:1
```

```
cookie=0x0, duration=4.685s, table=0, n_packets=3225,
    n_bytes=9745066, idle_timeout=40, priority=5,tcp,
    in_port=1,nw_src=10.10.10.2,nw_dst=10.0.0.9
    actions=output:5
cookie=0x0, duration=54.216s, table=0, n_packets=30941,
    n_bytes=640767938, idle_timeout=40, priority=4,tcp,
    in_port=4 actions=output:5
cookie=0x0, duration=54.216s, table=0, n_packets=30961,
    n_bytes=2043630, idle_timeout=40, priority=4,tcp,
    in_port=5,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:0a->eth_dst,output:4
cookie=0x0, duration=136.896s, table=0, n_packets=16,
    n_bytes=672, priority=1,arp actions=NORMAL
cookie=0x0, duration=136.896s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=136.896s, table=0, n_packets=3,
    n_bytes=230, priority=0 actions=CONTROLLER:65535
```

Sono state aggiunte le regole relative alla fase di mirroring dell'utente VMU2, con priorità 5. Si può inoltre notare che le regole relative al flusso di VMU1 con priorità minore di 4 sono state rimosse per inattività.

Passati i 60 secondi di analisi, il terminale relativo al controller visualizzerà le seguenti informazioni:

```
** STOP ndpiReader

nDPI Memory statistics:
        nDPI Memory (once):      91.02 KB
        Flow Memory (per flow):  1.92 KB
        Actual Memory:           1.55 MB
        Peak Memory:             1.55 MB
[CONTROLLER - TIMER (2015-09-12 14:12:37)]
    Host 10.10.10.2:51973 is exchanging packets
    with Host 10.0.0.9:5001, via TCP
**** CONGESTED CASE at [TIMER (2015-09-12 14:12:37)]
**** STEERING BUSUSER FLOWS ****
**bu_c: 1 _ TCP pkt from VMU1 steered to WANA (LAN port) **
**bu_c: 2 _ TCP pkt from WANA (LAN port) to VMU1 ****
**bu_c: 3 _ TCP pkt to VMU1 steered to WANA (WAN port) **
**bu_c: 4 _ TCP pkt from WANA (WAN port) to destination **
**** STEERING RESUSER FLOWS ****
**ru_c: 1 _ TCP pkt from VMU2 steered to TC **
**ru_c: 2 _ TCP pkt from TC to VMU2 **
**ru_c: 3 _ TCP pkt from TC (2nd port) to destination **
```

```
**ru_c: 4 _ TCP pkt to VMU2 steered to TC (2nd port) **
```

Anche in questo caso, dopo 60 secondi di analisi dei pacchetti, il DPI si ferma e indica che l'host 10.10.10.2 (VMU2) sta scambiando pacchetti con l'host 10.0.0.9 (H1), col protocollo TCP.

Il controller, accorgendosi però della presenza sia del flusso di VMU1 che di VMU2, considera la topologia congestionata ed i flussi vengono redirezionati dal VMU1 al WANA, dal WANA alla destinazione (ovvero al Virtual Router), dal VMU2 al TC, dal TC alla destinazione e viceversa.

A questo punto, viene osservata nuovamente la tabella delle entries:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Il risultato di tale operazione è, per lo switch S1:

```
cookie=0x0, duration=10.664s, table=0, n_packets=2195,
    n_bytes=148470, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.2 actions=output:2
cookie=0x0, duration=10.664s, table=0, n_packets=3000,
    n_bytes=8217024, idle_timeout=40, priority=6,tcp,
    in_port=2 actions=set_field:00:00:00:00:00:07->eth_dst,
    output:5
cookie=0x0, duration=10.664s, table=0, n_packets=10373,
    n_bytes=684618, idle_timeout=40, priority=6,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=10.664s, table=0, n_packets=12442,
    n_bytes=140635708, idle_timeout=40, priority=6,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:05->eth_dst,
    output:4
cookie=0x0, duration=70.846s, table=0, n_packets=48900,
    n_bytes=170622024, idle_timeout=40, priority=5,tcp,
    in_port=2,nw_src=10.10.10.2,nw_dst=10.0.0.9
    actions=set_field:00:00:00:00:00:03->eth_dst,output:3
cookie=0x0, duration=70.849s, table=0, n_packets=45759,
    n_bytes=3146462, idle_timeout=40, priority=5,tcp,
    in_port=3,nw_src=10.0.0.9,nw_dst=10.10.10.2
    actions=output:2
cookie=0x0, duration=120.375s, table=0, n_packets=85834,
    n_bytes=5854760, idle_timeout=40, priority=4,tcp,
    in_port=6,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=120.375s, table=0, n_packets=87246,
    n_bytes=1152050404, idle_timeout=40, priority=4,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:09->eth_dst,
```

```
    output:6
cookie=0x0, duration=203.558s, table=0, n_packets=30,
    n_bytes=1260, priority=1,arp actions=NORMAL
cookie=0x0, duration=203.558s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=203.558s, table=0, n_packets=5,
    n_bytes=330, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
cookie=0x0, duration=19.333s, table=0, n_packets=4677,
    n_bytes=12366178, idle_timeout=40, priority=6,tcp,
    in_port=3 actions=output:5
cookie=0x0, duration=19.333s, table=0, n_packets=5117,
    n_bytes=361766, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.2
    actions=set_field:00:00:00:00:00:08->eth_dst,output:3
cookie=0x0, duration=19.333s, table=0, n_packets=17456,
    n_bytes=198927176, idle_timeout=40, priority=6,tcp,
    in_port=2 actions=output:5
cookie=0x0, duration=5.711s, table=0, n_packets=19503,
    n_bytes=1311446, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.1
    actions=set_field:00:00:00:00:00:06->eth_dst,output:2
cookie=0x0, duration=79.513s, table=0, n_packets=45301,
    n_bytes=3116234, idle_timeout=40, priority=5,tcp,
    in_port=5,nw_src=10.0.0.9 ,nw_dst=10.10.10.2
    actions=set_field:00:00:00:00:00:04->eth_dst,output:1
cookie=0x0, duration=79.513s, table=0, n_packets=49456,
    n_bytes=172335504, idle_timeout=40, priority=5,tcp,
    in_port=1,nw_src=10.10.10.2,nw_dst=10.0.0.9
    actions=output:5
cookie=0x0, duration=129.044s, table=0, n_packets=88901,
    n_bytes=1159454658, idle_timeout=40, priority=4,tcp,
    in_port=4 actions=output:5
cookie=0x0, duration=129.044s, table=0, n_packets=84615,
    n_bytes=5774306, idle_timeout=40, priority=4,tcp,
    in_port=5,nw_dst=10.10.10.1
cookie=0x0, duration=203.017s, table=0, n_packets=30,
    n_bytes=1260, priority=1,arp actions=NORMAL
cookie=0x0, duration=203.017s, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=203.017s, table=0, n_packets=3,
    n_bytes=230, priority=0 actions=CONTROLLER:65535
```

La situazione attuale presenta nuove regole con priorità 6, che riguardano

i flussi di VMU1 e VMU2 passanti da WANA e TC, che rendono inutilizzate le precedenti regole a priorità 4 (caso non congestionato di VMU1) e priorità 5 (traffico di VMU2 redirezionato al DPI e da esso alla destinazione).

Dopo 40 secondi di inattività delle vecchie entries, esse verranno rimosse, infatti, circa dieci secondi prima della chiusura dell'iperf, osserviamo nuovamente la tabella dello switch:

```
~$ sudo ovs-ofctl dump-flows NOMESWITCH -O OpenFlow13
```

Ottenendo, per lo switch S1:

```
cookie=0x0, duration=43.923s, table=0, n_packets=14627,
    n_bytes=1115458, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.2 actions=output:2
cookie=0x0, duration=43.923s, table=0, n_packets=17794,
    n_bytes=43573292, idle_timeout=40, priority=6,tcp,
    in_port=2 actions=set_field:00:00:00:00:00:07->eth_dst,
    output:5
cookie=0x0, duration=43.923s, table=0, n_packets=43231,
    n_bytes=2910278, idle_timeout=40, priority=6,tcp,
    in_port=4,nw_dst=10.10.10.1 actions=output:1
cookie=0x0, duration=43.923s, table=0, n_packets=45430,
    n_bytes=483767684, idle_timeout=40, priority=6,tcp,
    in_port=1 actions=set_field:00:00:00:00:00:05->eth_dst,
    output:4
cookie=0x0, duration=236.817, table=0, n_packets=34,
    n_bytes=1428, priority=1,arp actions=NORMAL
cookie=0x0, duration=236.817, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=236.817, table=0, n_packets=5,
    n_bytes=330, priority=0 actions=CONTROLLER:65535
```

Per lo switch S2:

```
cookie=0x0, duration=43.848s, table=0, n_packets=15167,
    n_bytes=38376798, idle_timeout=40, priority=6,tcp,
    in_port=3 actions=output:5
cookie=0x0, duration=43.848s, table=0, n_packets=15085,
    n_bytes=1145686, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.2
    actions=set_field:00:00:00:00:00:08->eth_dst,output:3
cookie=0x0, duration=43.848s, table=0, n_packets=43615,
    n_bytes=465200374, idle_timeout=40, priority=6,tcp,
    in_port=2 actions=output:5
cookie=0x0, duration=43.848s, table=0, n_packets=44961,
```

```
    n_bytes=3024458, idle_timeout=40, priority=6,tcp,
    in_port=5,nw_dst=10.10.10.1
cookie=0x0, duration=236.321, table=0, n_packets=36,
    n_bytes=1512, priority=1,arp actions=NORMAL
cookie=0x0, duration=236.321, table=0, n_packets=0,
    n_bytes=0, priority=1,icmp actions=NORMAL
cookie=0x0, duration=236.321, table=0, n_packets=3,
    n_bytes=230, priority=0 actions=CONTROLLER:65535
```

Tutte le regole inutilizzate sono state rimosse, rimangono quindi, ad entrambi gli switch, solamente le entries TCP relative al caso congestionato di VMU1 e VMU2 e le tre iniziali di ARP, ICMP e connessione al controller.

Visualizzazione dei terminali degli Host

Terminato il test, è possibile visualizzare il resoconto delle performance di banda delle connessioni direttamente dai terminali degli host. Tali risultati sono del tutto simili a quelli del caso precedente.

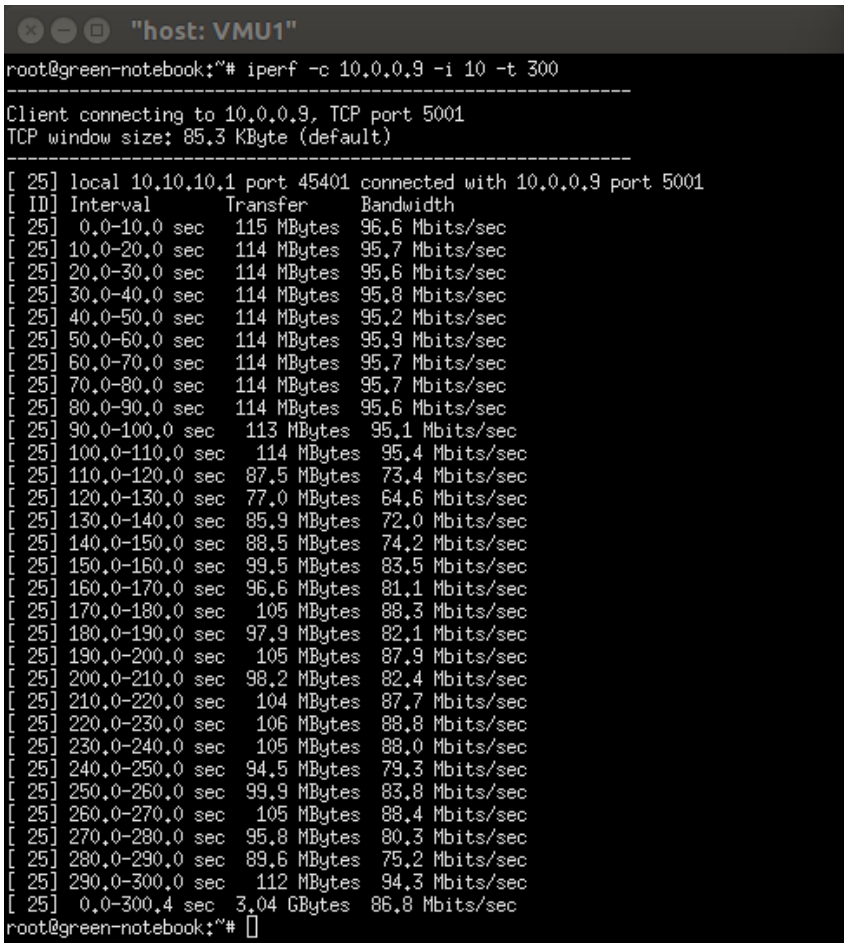

Figura 5.9: Terminale dell'host VMU1.

Nell'host VMU1, in Figura 5.9, si nota l'attacco subito alla banda da parte del flusso di VMU2, principalmente avvenuto dal centodecimo secondo in poi, per recuperare in seguito parte della banda, grazie all'azione di shaping del TC.

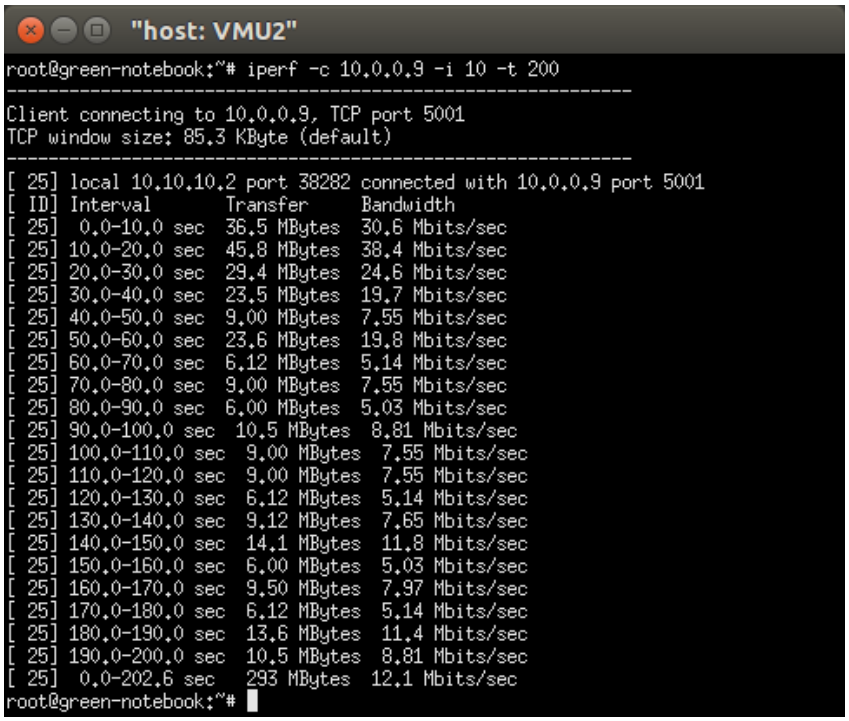

Figura 5.10: Terminale dell'host VMU2.

Nell'host VMU2, in Figura 5.10, si osserva il periodo iniziale in cui ha utilizzato maggiore banda, attaccando il flusso di VMU1. In seguito all'analisi e al redirezionamento al TC, la banda viene visibilmente ridotta sotto ai 10 Mbits/sec.

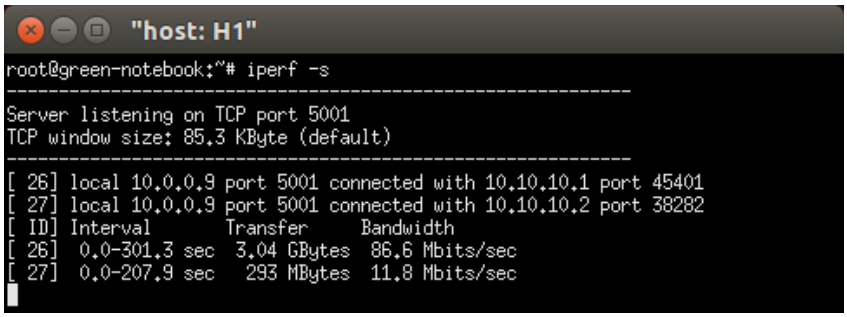

Figura 5.11: Terminale dell'host H1.

In Figura 5.11 è possibile osservare il report finale del test, mostrato dall'host H1.

Capitolo 6

Generalizzazione del Controller

Questo capitolo tratta l'implementazione di un controller Ryu generalizzato, facendo riferimento ad una topologia di rete simile a quella L3, vista nel Capitolo 5, ma più complessa riguardo alla struttura della destinazione.

Tale controller, a differenza dei precedenti, è predisposto al funzionamento con più utenti, permettendo la gestione di più flussi dati per ciascun utente tramite l'archiviazione delle informazioni ad essi relative ed è in grado di constatare l'eventuale congestione o non congestione della rete in modo autonomo, per poi applicare regole che permettano di sfruttare al meglio le risorse disponibili.

6.1 Topologia di riferimento

Prima di procedere all'analisi del codice del controller è necessario considerare la struttura della rete a cui esso si rivolge, tenendo presente la possibilità di avere più utenti, indipendentemente dalla loro priorità.

La topologia di riferimento del controller in analisi in questo capitolo è costituita da tre switch ed è rappresentata in Figura 6.1

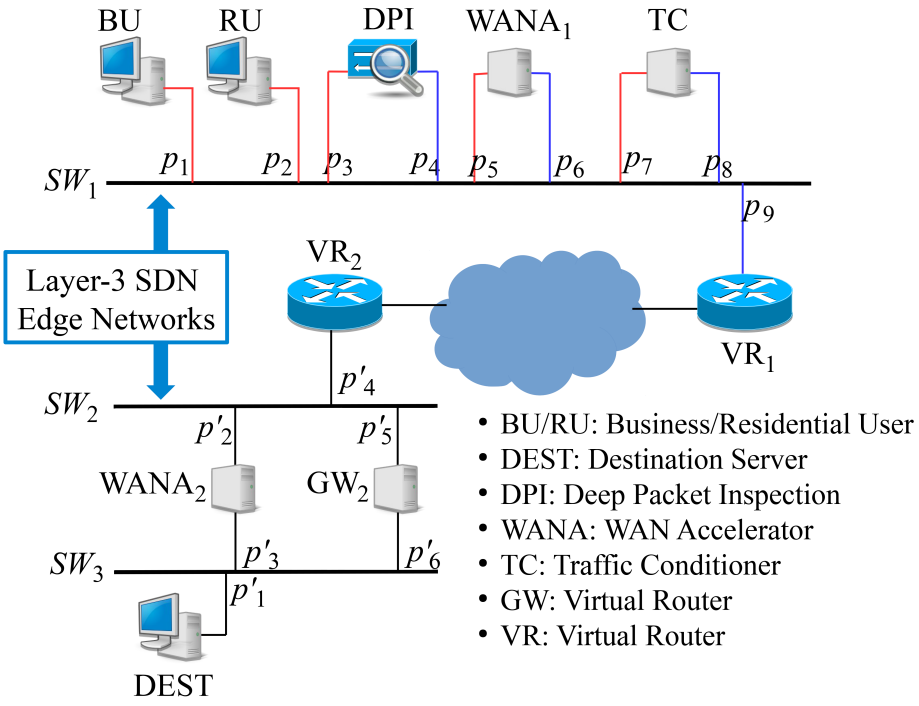

Figura 6.1: Topologia del controller Ryu generalizzato. [20]

Tale topologia presenta alcune fondamentali modifiche, rispetto alle precedenti reti di livello L2 e L3, che rispecchiano più fedelmente un caso reale di applicazione del controller:

- Nel primo switch viene realizzata una topologia simile a quella del caso di studio L3, affrontato nel Capitolo 5, mediante l'utilizzo di *VLAN*. Sono quindi stati utilizzati due VLAN ID differenti per virtualizzare la presenza di un secondo switch.

  Nella Figura 6.1, tale suddivisione di VLAN è rappresentata dal diverso colore dei collegamenti alle interfacce dei dispositivi dello switch SW_1, rosso per indicare la rete interna, blu per la rete di gateway.

  Lo switch SW_1 è denominato, nel controller, **br-int**.

- Tra il secondo ed il terzo switch, rispettivamente SW_2 e SW_3, sono presenti due host, il WanaDec (o $WANA_2$) ed un gateway (GW_2).

Nel WanaDec viene realizzata una funzionalità di rete complementare a quella del Wana presente nel primo switch, in quanto tale dispositivo è destinato alla decompressione dei paccchetti, permettendone perciò la lettura all'utente finale.

Il gateway GW_2 viene posizionato in modo da realizzare, per i pacchetti che non necessitano del passaggio tramite il WanaDec, un percorso verso la destinazione, ovvero l'host Dest.

- Nel controller, lo switch SW_2 è denominato **br3**, lo switch SW_3 è invece denominato **br4**. Tali switch sono entrambi gestiti dal controller, come accade per SW_1.

Terminata l'analisi della topologia, è possibile concentrarsi sul codice del controller.

6.2 Il codice

Il codice del controller Ryu descritto in questa sezione è rivolto ad un approccio più generalizzato, ed è quindi destinabile a più reti della stessa tipologia senza effettuare alcuna modifica. Si tratta perciò di un'evoluzione dei codici precedentemente realizzati.

Tale controller fa affidamento alla macchina a stati presentata nel Capitolo 4, adottando più fedelmente possibile gli algoritmi ad esso associati. [17]

Vengono di seguito riportati i frammenti di codice salienti, illustrando quanto possibile il criterio alla base della generalizzazione a cui aspira questo controller.

Il codice del controller è consultabile interamente nell'Appendice A.9 **controllerl3_final.py**.

- Una volta importate le librerie, si inizializzano le variabili globali presenti nel codice.

```
# INITIALIZE GLOBAL VARIABLES
global flow_id
flow_id=0
```

La variabile *flow_id* rappresenta l'identificativo univoco di ogni singolo flusso, viene aumentato di uno ogni volta che un flusso viene analizzato

e ,tramite esso, è possibile gestire i flussi attivi nella rete ed installare regole specifiche per ogni flusso.

```
global flows_state
flows_state = [] # List of flows
```

La lista *flow_state* rappresenta l'archivio in cui vengono salvate tutte le informazioni relative ai flussi presenti nella rete. Tutte le informazioni riferite ad un singolo flusso vengono inserite in un dizionario e la variabile *flow_id* funge in questo caso da indice per la lista.

L'elenco delle informazioni presenti per ogni flusso è descritto nella definizione della funzione *_memFlow*.

```
global active_flows
active_flows = [] # List of active flows
```

La lista *active_flows* contiene tutti i flow_id dei flussi attivi nella rete. I flussi vengono considerati tali e quindi aggiunti ad essa dal momento in cui inizia la classificazione, quando viene perciò eclusa la condizione di non conformità del flusso.

```
global classified_flows
classified_flows = [] #List of classified flows
                      #(after the preventive enforcement)
```

La lista *classified_flows* contiene tutti i flow_id dei flussi attivi classificati dal DPI, vengono considerati tali e quindi aggiunti ad essa i flussi che subiscono le azioni di reindirizzamento relative al caso di congestionamento preventivo. Questa lista viene creata con lo scopo di gestire tutti i flussi classificati attraverso le fasi di *Enforcement* o *Not Enforcement*, senza agire sui flussi attivi che non sono ancora stati classificati con successo.

```
global  hipriousers
hipriousers = [] #List of high priority users
```

La lista *hipriousers* contiene tutti gli indirizzi IP degli utenti a maggiore priorità e viene utilizzata per distinguere i flussi durante la fase di *Enforcement*.

```
global flowlimit
flowlimit = 3   #Max flow in Not Enforcement case
```

La variabile *flowlimit* rappresenta il limite di flussi attivi presenti nella rete che discrimina il caso di congestionamento da quello di non congestionamento.

```
global wanaproto
wanaproto = [200, 201, 202]
            #Protocols of compressed packets
```

Infine, la lista *wanaproto*, include i protocolli corrispondenti ai pacchetti compressi dal WANA.

- La caratteristica principale di questo controller è l'indipendenza dagli indirizzi IP, MAC e dai numeri di porta dei dispositivi della rete, per questo motivo, essi sono presenti solamente nella parte iniziale del codice, che rappresenta un possibile risultato di un'entità esterna denominata *Topology Manager*, destinata alla scoperta e gestione di indirizzi e numeri di porta dei dispositivi connessi alla rete.

  In questo frammento di codice viene perciò ipotizzata la conoscenza delle informazioni relative ai dispositivi presenti nella rete, in forma di variabili semplici o di dizionari, in modo da potervi accedere successivamente quando necessario, lasciando quindi inalterato il resto del codice. Questa implementazione è la chiave della generalizzazione del codice del controller.

```
def __init__(self, *args, **kwargs):
        super(BasicOpenStackL3Controller,
            self).__init__(*args, **kwargs)
        self.dpset = kwargs['dpset']
        #VARIABLES
        self.switch_dpid_name = {}
        self.connections_name_dpid = {}
        #Set of general parameters
        self.net_topo = ['br-int', 'br3', 'br4']
        self.users = ['BusUser', 'ResUser']
        self.net_func = ['DPI', 'TC', 'Wana', 'WanaDec',
                         'gw_dest', 'vr', 'vr_dest'] # NF

    #self.int_network_vid = 9
    #self.gw_network_vid = 10
    self.outport = 1
    self.outport_dest = 1
    self.sink = 3
```

```
self.bususer_port={'port1': 88, 'port2': None}
self.resuser_port={'port1': 87, 'port2': None}
self.wana_port={'port1': 113, 'port2': 114}
self.tc_port={'port1': 103, 'port2': 104}
self.dpi_port={'port1': 106, 'port2': 107}
#self.gw_port={'port1': , 'port2': }
self.wana_dest_port={'port1': 2, 'port2':1 }
self.gw_dest_port={'port1': 3, 'port2': 2}

self.bcast = "ff:ff:ff:ff:ff:ff"
self.ip_resuser = "192.168.8.35"
self.ip_bususer = "192.168.8.36"
#self.ip_gw = "10.10.0.1"
#self.ip_router_vr = "10.0.0.1"
self.ip_nat_bu = "10.250.0.115"
self.ip_nat_ru = "10.250.0.116"
self.ip_sink = "10.30.0.2"

    # HIGH PRIORITY USERS LIST
    global hipriousers
hipriousers=[self.ip_bususer]

    # MAC DICTIONARY
self.dpi_mac={'eth1':"fa:16:3e:42:9c:51",
            'eth2':"fa:16:3e:4f:00:e5"}
self.wana_mac={'eth1':"fa:16:3e:fe:76:34",
            'eth2':"fa:16:3e:8d:6f:d9"}
self.tc_mac={'eth1':"fa:16:3e:d9:5b:b9",
            'eth2':"fa:16:3e:75:3a:e7"}
#self.gw_mac={'eth1':"fa:16:3e:f1:cc:7b",
#             'eth2':"fa:16:3e:a4:16:7a"}
self.wana_dest_mac={'eth1':"52:54:00:a9:08:67",
            'eth2':"52:54:00:8c:55:14"}
self.gw_dest_mac={'eth1':"52:54:00:bb:3c:d1", '
            eth2':"52:54:00:12:5f:5f"}


    self.dpiExePath = '/home/ubuntu/nDPI_Tool/
        nDPI/example/ndpiReader'
    self.dpiCapPath = '/tmp/ndpiresult_'
    self.a = 0
    self.b = 0
    self.DPIcredentials='ubuntu@192.168.122.64'
```

- Per poter accedere alle informazioni precedentemente ottenute è necessario definire alcune funzioni che forniscano tali dati durante il funzionamento del controller.

```
# GET MAC ADDRESS
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]
```

La funzione *get_in_mac_address* permette di recuperare l'indirizzo MAC relativo all'interfaccia di un dispositivo di rete, anche dotato di più interfacce, grazie alla specifica di direzione del flusso, ovvero *inbound* per i flussi diretti verso l'utente o *outbound* per i flussi opposti.

```
# GET IN PORT
def get_in_port(self, host, direction):
   result_tuple = []

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

```
            result_tuple.append('br-int')
            result_tuple.append(self.wana_port['port2'])
        elif host =='TC' :
            if direction == 'outbound':
                result_tuple.append('br-int')
                result_tuple.append(self.tc_port['port1'])
            elif direction == 'inbound':
                result_tuple.append('br-int')
                result_tuple.append(self.tc_port['port2'])
        #elif host =='GW' :
            #if direction == 'outbound':
                #result_tuple.append('br-int')
                #result_tuple.append(self.gw_port['port1'])
            #elif direction == 'inbound':
                #result_tuple.append('br-int')
                #result_tuple.append(self.gw_port['port2'])
        elif host =='WanaDec' :
            if direction == 'outbound':
                result_tuple.append('br3')
                result_tuple.append(self.wana_dest_port[
                                    'port1'])
            elif direction == 'inbound':
                result_tuple.append('br4')
                result_tuple.append(self.wana_dest_port[
                                    'port2'])
        elif host =='GWDest' :
            if direction == 'outbound':
                result_tuple.append('br3')
                result_tuple.append(self.gw_dest_port[
                                    'port1'])
            elif direction == 'inbound':
                result_tuple.append('br4')
                result_tuple.append(self.gw_dest_port[
                                    'port2'])
        return result_tuple
```

La funzione *get_in_port* permette di recuperare il numero di porta relativo all'interfaccia di un dispositivo di rete, anche dotato di più interfacce, specificando la direzione del flusso.

```
def _get_ports_info(self, dpid):
    return self.dpset.get_ports(dpid)

def _get_port_name(self, dpid, port):
    return self.dpset.get_port(dpid, port).name
```

```
def connectionForBridge(self, bridge):
    if bridge in self.connections_name_dpid:
        return self.connections_name_dpid[bridge]
    else:
        return -1
```

Le funzioni *_get_ports_info*, *_get_port_name* e *connectionForBridge* sono le stesse presenti nel controller del caso di studio L3 e permettono la gestione di più switch della stessa rete.

- Un'importante novità di questo controller è la capacità di controllare i flussi attivi presenti nella rete. Tale funzionalità è permessa grazie alla definizione di due funzioni.

```
# ANALIZE ACTIVE FLOWS REQUEST
def _request_stats(self):
    self.logger.debug('FLOW CONTROL request')
    datapath = self.connectionForBridge("br-int")
    threading.Timer(30.0, self._request_stats).start()
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)
```

La prima funzione, *_request_stats*, genera una richiesta di statistiche relative ai flussi allo switch indicato dal datapath, in questo caso il primo. Tale definizione contiene la linea:

```
threading.Timer(30.0, self._request_stats).start()
```

che permette di chiamare ciclicamente la funzione stessa, con un intervallo di tempo di 30 secondi tra una chiamata e l'altra.

Tale richiesta verrà soddisfatta dallo switch mediante un messaggio di risposta, gestito ed analizzato dalla seguente funzione, composta di due parti:

```
# 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
self.logger.debug('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 >= 34500 ],
               key=lambda flow: (flow.priority)):
      try:
         if (stat.match['ipv4_src'] ==
               flows_state[flow_id]['ip_src'])
               and (stat.match['tcp_src'] ==
               flows_state[flow_id]['port_src']):
            valid=1
      except:
         pass
      try:
         if (stat.match['ipv4_dst'] ==
               flows_state[flow_id]['ip_src'])
               and (stat.match['tcp_dst'] ==
               flows_state[flow_id]['port_src']):
            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)
```

La prima parte della funzione gestisce il messaggio ricevuto dallo switch, cercando la presenza di ogni flusso attivo tra le regole inserite nella flow table dello switch stesso. Tale ricerca viene effettuata confrontando le informazioni relative ad indirizzo IP sorgente e porta sorgente, reperibili nella lista di dizionari *flows_state*, di ogni flusso attivo, il cui *flow_id* è contenuto nella lista *active_flows*, con le corrispondenze di indirizzo IP e porta, sorgente o destinazione, presenti nelle regole inserite nella flow table dello switch.

Se le informazioni prescelte di un flusso non vengono trovate tra le regole presenti nello switch, tale flusso viene considerato inattivo e perciò rimosso dalle liste di *active_flows* e *classified_flows*.

Ciò è reso possibile dalla configurazione, nelle regole, di un tempo di timeout di inattività, dopo il quale avviene la rimozione della regola rimasta inutilizzata per il tempo previsto. In questo modo è possibile capire quali flussi sono terminati, una volta rimossa la regola ad essi relativa.

In questa funzione si utilizza il comando *try* poichè la ricerca di parole chiave non presenti in una determinata regola inserita nello switch genererebbe un'eccezione, non permettendo quindi la corretta esecuzione dell'applicazione. Allo stesso modo accade per la rimozione di un flusso dalla lista *classified_flows*, nel caso il flusso terminato non sia stato ancora classificato con successo dal controller.

```
# ANALIZE IF CONGESTION ARISE
if len(active_flows) <= flowlimit :
        self.logger.debug('Not Enforcement State')
        self._handle_NotEnforcement_N_State()
else :
        self.logger.debug('WARNING! -
             Too many active flows:
             Enforcement State')
        self._handle_Enforcement_E_State()
```

La seconda parte della funzione è destinata al controllo delle eventuali congestioni della rete. Viene perciò confrontato il numero di flussi

attivi, ovvero la lunghezza della lista *active_flows*, col limite *flowlimit* precedentemente impostato.

Nel caso il numero di flussi attivi sia inferiore al limite, viene chiamata la funzione relativa allo stato di *Not Enforcement* della macchina a stati del controller, in caso contrario, col numero di flussi attivi superiore al limite, viene chiamata la funzione opposta alla precedente, ovvero quella relativa allo stato di *Enforcement*.

In questo modo viene realizzato un controllo costante dello stato della rete, gestendo ogni eventuale congestione e sfruttando nel miglior modo ogni risorsa disponibile nella rete.

- In seguito, vengono definite le funzioni che realizzano l'avvio ed il recupero dell'output del DPI. Nei codici precedenti, tali funzionalità erano già state implementate all'interno di altre funzioni, in questo caso si è preferito definire delle funzioni specifiche in modo da strutturare in modo più ordinato il codice del controller.

```
def startDPI(self, flow_id):
   os.system("ssh ubuntu@192.168.10.32 'sudo nohup
          %s -i eth1 -f \"ip host %s\" -v 2 -s 10
          -j %s%s.json  > foo.out 2> foo.err <
          /dev/null & '" % (self.dpiExePath,
          flows_state[flow_id]['ip_src'],
          self.dpiCapPath,
          flows_state[flow_id]['ip_src']))
   self.logger.debug("[CONTROLLER - TIMER (%s)]
                DPI started!\n\n\n\n",
                datetime.datetime.fromtimestamp(
                    time.time()).strftime(
                         '%Y-%m-%d %H:%M:%S'))

def obtain_DPI_output(self, flow_id):
   result = None
   try:
      result = subprocess.check_output(['ssh',
          'ubuntu@192.168.10.32',
          'sudo cat %s%s.json' %
          self.dpiCapPath,
          flows_state[flow_id]['ip_src'] ])
   except subprocess.CalledProcessError as e:
      result = e.output
   return result
```

- Viene definita la funzione di memorizzazione delle informazioni relative ad un flusso all'interno dei dizionari presenti nella lista *flows_state*. Tramite tale funzione è possibile creare un nuovo dizionario o aggiornarne uno già esistente, modificando anche una sola informazione, semplicemente utilizzando le parole chiave durante la chiamata.

```
def _memFlow(self, flow_id, ip_src="0.0.0.0",
             port_src=0, ip_dst="0.0.0.0",
             port_dst=0, in_port=0, ip_proto=0,
             state="X", rules=[]) :

   global flows_state
   global active_flows

   if flow_id in active_flows :
      if ip_src != "0.0.0.0" :
         flows_state[flow_id]['ip_src'] = ip_src
      if port_src != 0 :
         flows_state[flow_id]['port_src'] = port_src
      if ip_dst != "0.0.0.0" :
         flows_state[flow_id]['ip_dst'] = ip_dst
      if port_dst != 0 :
         flows_state[flow_id]['port_dst'] = port_dst
      if in_port != 0 :
         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 != [] :
         flows_state[flow_id]['rules'] = rules
   else :       #NEW FLOW
      flow={}
      flow['flow_id'] = flow_id
      flow['ip_src'] = ip_src
      flow['port_src'] = port_src
      flow['ip_dst'] = ip_dst
      flow['port_dst'] = port_dst
      flow['in_port'] = in_port
      flow['ip_proto'] = ip_proto
      flow['state'] = state
      flow['rules'] = rules
      if ip_src in self.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)

    #UPDATE LOG
    self.logger.debug("**FLOW UPGRADED** : %s",
                      flows_state[flow_id])
```

Nel caso si cerchi di memorizzare un nuovo flusso, e quindi sia risultato assente il *flow_id* nella lista *active_flows*, viene creato un nuovo dizionario e l'identificativo del flusso stesso viene inserito nella lista dei flussi attivi.

Le informazioni raccolte all'interno del dizionario corrispondente al flusso sono:

- *flow_id*, identificativo del flusso, così da visualizzarlo in eventuali stampe su schermo;
- *ip_src*, indirizzo IP sorgente;
- *port_src*, numero di porta sorgente;
- *ip_dst*, indirizzo IP destinazione;
- *port_dst*, numero di porta destinazione;
- *in_port*, numero di porta in ingresso dello switch;
- *ip_proto*, protocollo del flusso, nel nostro caso è utile per capire se i pacchetti che transitano dallo switch sono TCP o UDP;
- *state*, stato del sistema, indica in che stato della macchina a stati si trova attualmente il flusso considerato. Viene utilizzato anche per discriminare i flussi durante l'applicazione delle regole di Enforcement e Not Enforcement;
- *rules*, lista dei dizionari relativi alle regole installate nello switch, appartenenti al flusso considerato;
- *ip_nat*, indirizzo del Floating IP associato all'utente, permette di poter installare le regole anche nei dispositivi successivi al NAT.

- Per installare le regole relative ai protocolli ARP e ICMP, viene inserita la chiamata alla funzione dello stato iniziale Init, della macchina a stati, all'interno della funzione relativa all'installazione della regola di *Table*

miss, ovvero quella per cui viene inviato al controller ogni pacchetto che non trova corrispondenze nella flow table.

```
# Tell the switch to send msgs to the
    # Controller in case of table miss
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(
           ofproto.OFPP_CONTROLLER,
           ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)

# ARP and ICMP
self._handle_Initial_Init_State(datapath)
```

- Per rispettare la macchina a stati descritta in precedenza, la funzione di gestione del packet-in si limita a richiamare la funzione dello stato di classificazione *_handle_Classification_C_State* per poi chiamare la funzione destinata a gestire la conclusione dell'analisi *dpi_analysis_finished* dopo un'attesa di 15 secondi, durante la quale avviene l'analisi del flusso.

```
#Handle PacketIn event
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):

   global flow_id
   self.logger.debug("[DEBUG] FLOW-ID %s", flow_id)

   msg = ev.msg
   pkt = packet.Packet(msg.data)

   header = pkt.get_protocol(ipv4.ipv4)
   self.logger.debug("[DEBUG] ip_proto=%s ip_src=%s
                        ip_dst=%s in_port=%d",
                     header.proto, header.src,
                     header.dst, in_port)

   np = len(pkt.get_protocols(ethernet.ethernet))
   self.logger.debug("[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)

            #ATTESA DI 15 SECONDI E FINE ANALISI
            t=Timer(15.0, self.dpi_analysis_finished(flow_id))
            t.start()

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

- La funzione *dpi_analysis_finished*, chiamata alla fine dell'analisi, è strutturata in due sezioni, la prima si occupa caricare l'output del DPI, in previsione della successiva lettura.

```
def dpi_analysis_finished(self, flow_id):

    self.logger.debug("[DEBUG - READING JSON FILE] ")
    # Step 1: read json output file (DPI classification)
    json_data = self.obtain_DPI_output(flow_id)
    self.logger.debug("[DEBUG - JSON] %s", str(json_data))
    data = json.loads(json_data)
    list_of_flows = data["known.flows"]
```

La seconda sezione effettua una ricerca, all'interno dell'output, di flussi aventi protocollo TCP o UDP, di cui salva tutte le informazioni relative agli indirizzi IP e numeri di porta di sorgente e destinazione.

```
    # 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.debug("[CONTROLLER - TIMER (%s)]
                        Host %s:%s is exchanging packets
                        with Host %s:%s, via %s ",
                        datetime.datetime.fromtimestamp(
                        time.time()).strftime(
                        '%Y-%m-%d %H:%M:%S'),
                        i["host_a.name"], i["host_a.port"],
```

```
                    i["host_b.name"], i["host_n.port"],
                    i['protocol'])

        # UPDATING FLOW INFORMATIONS
        if host_a == self.ip_bususer or
                     host_a == self.ip_resuser or
                     host_b == self.ip_bususer or
                     host_b == self.ip_resuser :
           self._memFlow(flow_id, ip_src=host_a,
                         port_src=port_a, ip_dst=host_b,
                         port_dst=port_b)
           time.sleep(0.10)
           self._handle_PreventiveEnforcement_E_State(
                                                 flow_id)
```

Successivamente, se i flussi analizzati risultano generati dagli utenti della rete, aggiorna le informazioni relative ai flussi stessi prima di procedere all'avvio dello stato di Enforcement preventivo, chiamando la funzione *_handle_PreventiveEnforcement_E_State*.

- Infine, rimane la definizione delle funzioni degli stati del sistema. Tali funzioni installeranno nello switch regole specifiche per indirizzare ogni flusso come previsto dal Capitolo 4.

  Viene di seguito descritto qualche frammento, in modo da comprenderne la struttura. Per una visione più completa, si consiglia di osservare il codice presente per intero nell'appendice.

  Il seguente estratto rappresenta l'inserimento di una regola appartenente allo stato di classificazione del flusso.

```
# 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')
mac_addr = self.get_in_mac_address('DPI', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append(parser.OFPActionSetField(
              eth_dst=mac_addr))
action.append(parser.OFPActionOutput(
```

```
                switch_port[1]))
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(hard_timeout=270,
             priority=34500, match=parser.OFPMatch(
             in_port = coming_port, eth_type = 2048,
             ip_proto=pkt_ipv4.proto, ipv4_src = nw_src,
             tcp_src = pkt_tcp.src_port),
             instructions=inst )
dp.send_msg(msg)
```

La funzione *get_in_port* inserisce nella lista *switch_port* due valori, nella posizione 0 è presente il nome dello switch, mentre nella posizione 1 è presente il numero di porta richiesto nella chiamata della funzione stessa.

In questo caso la regola deve sostituire l'indirizzo MAC destinazione per permettere il passaggio del flusso tramite il DPI, realizzando così una corretta azione di reindirizzamento. Per questo motivo, viene utilizzata la funzione *get_in_mac_address* per reperire l'indirizzo MAC dell'interfaccia desiderata.

Ogni regola presenta, nella sezione *Match*, le corrispondenze relative a indirizzo IP e numero di porta sorgente, o destinazione, per identificare univocamente il flusso dati. Questo risulta necessario dal momento in cui vi è la possibilità di generare più flussi dallo stesso utente nella rete.

Ogni regola, inoltre, è progettata a predisporsi, in fase di installazione, al protocollo TCP o UDP a seconda della natura del flusso considerato.

La priorità relativa allo stato di classificazione del flusso è 34500.

In seguito ad ogni regola è sempre presente l'archiviazione delle informazioni relative alla regola appena definita. Tali informazioni vengono memorizzate sotto forma di dizionario, inserito in una lista contenente tutti i dizionari delle regole appartenenti ad un unico flusso.

```
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_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)
tmp_dict_actions.clear()

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

Infine, si effettua la pulizia delle liste e dei dizionari temporanei utilizzati durante l'archiviazione delle informazioni della regola, per procedere all'installazione, e alla raccolta delle informazioni, della regola successiva.

```
tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

Finito il codice corrispondente all'inserimento delle regole di uno stato, è sempre presente la chiamata alla funzione *_memFlow* per memorizzare tutte le informazioni raccolte, fino a tal punto, all'interno della lista di dizionari *flows_state*.

```
# SAVING CURRENT FLOW IN flows_state
self._memFlow(flow_id, state = 'C', rules = rules_list)
```

In generale, tutti i codici che realizzano l'installazione delle regole sono strutturati in questa modalità.

Nello stato di classificazione del flusso, è inoltre presente la chiamata alla funzione che avvia l'analisi del DPI.

```
# START DPI
self.startDPI(flow_id)
```

Al termine dell stato di Enforcement preventivo è invece presente la chiamata alla funzione che avvia il controllo ciclico del congestionamento della rete. Tale chiamata viene eseguita solamente al primo flusso, poichè successivamente si ripete automaticamente in modo autonomo.

Sempre in questo stato, avviene l'aumento del *flow_id*, permettendo così la gestione e l'archiviazione del flusso successivo.

```
# FLOWS CONTROL START WHEN FIRST FLOW
if f_id=0 :
   self._request_stats()

# UPDATE FLOW-ID
time.sleep(0.10)
global flow_id
flow_id=f_id+1
```

Per quanto riguarda gli stati di Enforcement e Not Enforcement, i flussi vengono gestiti grazie alla lista *classified_flows* che permette di applicare, ad un flusso per volta, tutte le regole previste dallo stato mediante il codice:

```
for flow in classified_flows :
```

In entrambi gli stati, prima di inserire le regole nello switch, si effettua un controllo relativo allo stato di appartenenza dei singoli flussi, in modo da non applicare più volte le stesse regole, nel caso il flusso si trovi già nello stato attuale del sistema. Tale funzionalità è realizzata con:

```
   if flows_state[flow]['state'] != 'E' :
```

Nello stato di Enforcement, si effettua inoltre un controllo supplementare per verificare la priorità dell'utente che genera ogni flusso, in modo da applicare il corretto indirizzamento, come previsto dallo stato di gestione di situazioni di congestionamento.

```
      if flows_state[flow]['ip_src'] in hipriousers :
```

Negli stati stati di Preventive Enforcement, Enforcement e Not Enforcement, le informazioni inserite nelle regole sono prelevate direttamente dalla lista di dizionari *flows_state*, accedendovi con questa modalità:

```
msg = parser.OFPFlowMod( idle_timeout=60,
         priority = 34501,
         match = parser.OFPMatch(in_port=switch_port[1],
             eth_type=2048,
             ip_proto=flows_state[flow]['ip_proto'],
             ipv4_dst=flows_state[flow]['ip_src'],
             tcp_dst = flows_state[flow]['port_src']),
         instructions=inst )
```

La priorità relativa alle regole dello stato di Enforcement è 34501, mentre per lo stato di Not Enforcement è 34502.

Per poter tornare allo stato di Enforcement, una volta passati per quello di Not Enforcement, è necessario rimuovere tutte le regole precedentemente inserite.

Tale rimozione è realizzata riutilizzando tutte le configurazioni già inserite per ogni singola regola nello stato di Not Enforcement, aggiungendo però un comando che indichi la rimozione della regola considerata. Tale aggiunta è il comando `command=ofproto.OFPFC_DELETE`, posizionato all'interno della costruzione del messaggio da inviare allo switch.

```
msg = parser.OFPFlowMod(idle_timeout=60,
         priority=34502,
         match=parser.OFPMatch(
             in_port = flows_state[flow]['in_port'],
             eth_type = 2048,
             ip_proto=flows_state[flow]['ip_proto'],
             ipv4_src = flows_state[flow]['ip_src'],
             tcp_src = flows_state[flow]['port_src'] ),
         instructions=inst,
         command=ofproto.OFPFC_DELETE  )
```

Infine, siccome la compressione applicata ai pacchetti dal WANA provoca un cambio di protocollo, è necessario impostare, nelle regole che riguardano il tragitto tra WANA e WANADEC, un protocollo corretto, differente da quello TCP o UDP. I protocolli da utilizzare in questo caso, contenuti all'interno della lista *wanaproto*, vengono prelevati tramite il seguente codice, che racchiude le regole destinate a tale modifica:

```
for protocol in wanaproto:
   # br-int Gateway network, inbound traffic,
   # from br4 to Wana (WAN port), MAC_DST is changed
```

```
...

# br-int Gateway network, outbound traffic,
# from Wana (WAN port) to br4

...

# br4, outbound traffic, to WanaDec

...

# br4, inbound traffic, from WanaDec

...
```

Il codice del controller è quindi terminato e rappresenta una notevole evoluzione rispetto a quelli visti in precedenza nel Capitolo 5, rendendo il sistema maggiormente dinamico, versatile e non più dedicato univocamente ad una rete specifica.

Capitolo 7

Conclusioni

In questo documento è stato descritto lo studio svolto nell'ambito del Service Chaining dinamico mediante l'utilizzo del controller SDN Ryu, applicando quindi il protocollo OpenFlow a casi pratici, ottimizzando la gestione delle risorse di rete e facendo riferimento all'ambiente del Cloud Computing.

Tutti i casi pratici sono stati svolti simulando le reti e gli utenti in esse presenti grazie al software Mininet, in previsione di possibili future implementazioni in reti ed architetture reali.

Risultato di tali esperimenti è stata certamente la conferma dei vantaggi presentati dai paradigmi messi in pratica, facendo risultare, di conseguenza, la collaborazione di SDN e NFV come un'ottima possibile soluzione da sviluppare per risolvere le problematiche di staticità e dipendenza dai fornitori delle *middle-boxes* della rete Internet tradizionale.

I possibili sviluppi futuri in questo ambito sono principalmente rappresentati dalla realizzazione su reti reali, dotate di piattaforma OpenStack, di test simili a quelli effettuati nelle simulazioni affrontate con Mininet, applicandovi i controller Ryu descritti nel corso delle sperimentazioni svolte in questo documento in modo da procedere quindi alla vera e propria implementazione finale.

Di fondamentale importanza, per raggiungere tale scopo, è la creazione e l'utilizzo di un'entità esterna al controller, il *Topology manager*, addetta a fornire al controller tutte le informazioni relative alla rete di applicazione, adattandone quindi in modo dinamico ed autonomo il funzionamento e permettendo così la stesura di un unico codice per più topologie di rete.

Capitolo 8

Ringraziamenti

Innanzitutto vorrei ringraziare Walter Cerroni, Chiara Contoli e Francesco Foresta per la massima disponibilità, cortesia e per tutto l'aiuto fornitomi durante le attività di tirocinio e di tesi.

Ringrazio di cuore la mia famiglia, senza la quale non avrei potuto intraprendere un percorso simile, fonte di costante supporto, aiuto, motivazione e consigli sinceri per affrontare ogni ostacolo presente lungo il cammino. Grazie quindi ai miei genitori, Simone e Cristina, alla mia sorellina Anna, alla zia Giulia, a Maria Grazia e ai miei nonni, ai quali rivolgo un pensiero profondo.

Grazie anche ad Ilaria, per la compagnia ed il continuo supporto morale rivolto in questi ultimi mesi di studio.

Infine, ma non di minore importanza, grazie ai miei compagni di studio e ai miei amici, chi presente da sempre, chi invece soltanto da qualche mese o anno, per ogni momento con loro condiviso e grazie ai quali ho potuto confrontarmi e crescere, giungendo finalmente a questo traguardo.

Appendice A

Codici utilizzati

A.1 L2switch.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/L2switch.py
```

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3

class L2Switch(app_manager.RyuApp):
   OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

   def __init__(self, *args, **kwargs):
      super(L2Switch, self).__init__(*args, **kwargs)

   @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
   def packet_in_handler(self,ev):
      msg=ev.msg
      dp=msg.datapath
      ofp=dp.ofproto
      ofp_parser = dp.ofproto_parser

      actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
      out= ofp_parser.OFPPacketOut(
```

```
            datapath=dp, buffer_id=msg.buffer_id,
            in_port=msg.in_port, actions=actions)
        dp.send_msg(out)
```

A.2 simple_switch_13.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/simple_switch_13.py
```

```
# Copyright (C) 2011 Nippon Telegraph and
#                    Telephone Corporation.
#
# Licensed under the Apache License,
# Version 2.0 (the "License");
# you may not use this file except in compliance
# with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to
# in writing, software distributed under the
# License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
# either express or implied.
# See the License for the specific language governing
# permissions and limitations under the License.

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


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 = {}

@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
    #
    # We specify NO BUFFER to max_len of the output
    # action due to OVS bug. At this moment, if we
    # specify a lesser number, e.g., 128, OVS will
    # send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot
    # output packets correctly.
    # The bug has been fixed in OVS v2.1.0.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(
                      ofproto.OFPP_CONTROLLER,
                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)

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)
```

```
@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']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s",
                     dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    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)]

    # 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)
```

A.3 test-l2-scenario.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/test-l2-scenario.py
```

Il codice di tale topologia è stato realizzato dalla Ing. Chiara Contoli, ad esso è stato necessario apportare alcune modifiche per poterlo utilizzare con la versione 1.3 di OpenFlow, .

```
#!/usr/bin/python

'''
TOPOLOGY USED IN NETSOFT 2015 TEST

Emulation of l2-scenario for the OpenStack case. Nodes:
- Host  VM-User1: one interface
- Host  VM-User2: one interface
- Host DPI: one interface
- Host WAN Accellerator (WANA): 2 interfaces,
  both connected to Linux Bridge
  (eth0 and eth1 are bridged with a LB, br-WANA)
- Host TC: 2 interfaces, both connected to Linux Bridge
  (eth0 and eth1 are bridged with a LB, br-TC)
- Host VR: 2 interfaces
- Host H1: one interfaces
- An Open vSwitch

1) sudo python test-l2-scenario.py
'''
```

```
from mininet.net import Mininet
from mininet.node import Node
from mininet.node import Host
from mininet.link import TCLink
from mininet.link import Intf
from mininet.log import setLogLevel, info
from mininet.cli import CLI
from mininet.node import Controller
from mininet.node import RemoteController
from mininet.util import quietRun

from time import sleep
import os
import sys

def defineNetwork():
   if len(sys.argv) < 2:
      print "Missing paramenter:
             python test-l2-scenario.py <debug=1|0>"
      sys.exit()
   #commento
   debug = sys.argv[1] #print some usefull information

   info("*** Create an empty network and
        add nodes and switch to it *** \n")
   net = Mininet(controller=RemoteController,
                 link=TCLink, build=False, xterms=True)
   info("\n*** Adding Controller: Controller will be
        external *** \n")
   info("\n*** Creating Switch *** \n")
   s1 = net.addSwitch('s1')
   s1.cmd('ovs-vsctl del-br ' + s1.name )
   s1.cmd('ovs-vsctl add-br ' + s1.name )
   s1.cmd('ovs-vsctl set Bridge '+ s1.name +
          ' stp_enable=false protocols=OpenFlow13' )
          # Disabling STP

   info("\n*** Creating VM-User 1 *** \n")
   vmu1 = net.addHost('VMU1')
   info("\n*** Creating VM-User 2 *** \n")
   vmu2 = net.addHost('VMU2')
   info("\n*** Creating DPI *** \n")
   dpi = net.addHost('DPI')
   info("\n*** Creating WAN A. *** \n")
```

```
wana = net.addHost('WANA')
info("\n*** Creating TC *** \n")
tc = net.addHost('TC')
info("\n*** Creating Virtual Router *** \n")
vr = net.addHost('VR')
info("\n*** Creating External Host *** \n")
h1 = net.addHost('H1')
info("\n*** Creating Links *** \n")
net.addLink(vmu1, s1, bw=100)
net.addLink(vmu2, s1, bw=100)
net.addLink(dpi, s1, bw=100)
net.addLink(wana, s1, bw=100)
net.addLink(wana, s1, bw=100)
net.addLink(tc, s1, bw=100)
net.addLink(tc, s1, bw=100)
net.addLink(vr, s1, bw=100)
net.addLink(vr, h1, bw=100)

#Trying to assign MAC address to each node of the topology
vmu1.setMAC("00:00:00:00:00:01", vmu1.name + "-eth0")
vmu2.setMAC("00:00:00:00:00:02", vmu2.name + "-eth0")
dpi.setMAC("00:00:00:00:00:03", dpi.name + "-eth0")
wana.setMAC("00:00:00:00:00:04", wana.name + "-eth0")
wana.setMAC("00:00:00:00:00:05", wana.name + "-eth1")
tc.setMAC("00:00:00:00:00:06", tc.name + "-eth0")
tc.setMAC("00:00:00:00:00:07", tc.name + "-eth1")
vr.setMAC("00:00:00:00:00:08", vr.name + "-eth0")
vr.setMAC("00:00:00:00:00:09", vr.name + "-eth1")
h1.setMAC("00:00:00:00:00:0A", h1.name + "-eth0")

#Disabling IPv6
info('\n*** Disabling IPv6 ...\n')
vmu1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
```

```
tc.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')

for intf in s1.intfs.values():
   s1.cmd( 'ovs-vsctl add-port ' + s1.name + ' %s' % intf )
   print "Eseguito comando: ovs-vsctl add-port s1 ", intf

info("\n*** Starting Network using Open vSwitch and
     remote controller*** \n")
# Creating a Linux Bridge on each host
nhosts = len(net.hosts)
print 'Total number of hosts: ' + str(nhosts)
count = 1

net.start()
s1.cmd('ovs-vsctl set bridge ' + s1.name +
       ' protocols=OpenFlow13') #OpenFlow 1.3
# Set the controller for the switch
print "Switch name: ", s1.name
s1.cmd('ovs-vsctl set-controller ' +  s1.name +
       ' tcp:127.0.0.1:6633')
info( '\n*** Waiting for switch to connect to controller' )
while 'is_connected' not in quietRun( 'ovs-vsctl show' ):
   sleep( 1 )
   info( '.' )
info('\n')

# Add some static rules (avoid ARP storm -
# these rules will be moved to Controller code) ...
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=4,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=5,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=6,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
s1.cmd('ovs-ofctl -O OpenFlow13 add-flow ' + s1.name +
       ' in_port=7,dl_dst=FF:FF:FF:FF:FF:FF,actions=drop')
info('\n*** Going to take down default configuration ...\n')
info('\n*** ... and creating Linux bridge on WANA and TC,
```

```
      as well as configuring interfaces \n')
for host in net.hosts:
   print 'Deleting ip address on ' + host.name +
         '-eth0 interface ...'
   host.cmd('ip addr del ' + host.IP(host.name + '-eth0') +
            '/8 dev ' + host.name + '-eth0')
   print 'Deleting entry in IP routing table on ' +
         host.name
   host.cmd('ip route del 10.0.0.0/8')
   print "Going to configure new IP"
   if host.name == 'WANA' or host.name == 'TC':
      print "Host with 2 interfaces: " + host.name
      host.cmd('brctl addbr br-' + host.name)
      host.cmd('brctl addif br-' + host.name + ' ' +
               host.name + '-eth0')
      host.cmd('brctl addif br-' + host.name + ' ' +
               host.name + '-eth1')
      host.cmd('ip addr add 10.10.10.' + str(count) +
               '/24 dev br-' + host.name)
      host.cmd('ip link set br-' + host.name + ' up')
      print "LB configured!"
      host.cmd('sysctl -w net.ipv4.ip_forward=1')
      print "IP Forwarding enabled!"
   elif host.name == 'H1':
      host.setIP("10.0.0." + str(count + 2), 30,
                 host.name + '-eth0')
      net.hosts[count - 2].setIP("10.0.0." + str(count + 3),
                 30, net.hosts[count - 2].name + "-eth1")
      print net.hosts[count - 2].name +
            "-eth1 interface has been configured!"
      print "[Checking VR IP] " +
            net.hosts[count - 2].IP('VR-eth1')
      net.hosts[count - 2].cmd('sysctl -w
                               net.ipv4.ip_forward=1')
      print "On VR node: IP Forwarding enabled!"
   else:
      host.setIP("10.10.10." + str(count), 24,
                 host.name + "-eth0")
      print "[CURRENT-CHECK] IP: " +
           net.hosts[count - 1].IP(
               net.hosts[count - 1].name + '-eth0')
   count = count + 1
   print "\n"
print "Configuring default gw on each host.."
count = 1
```

```
for host in net.hosts:
   print "Adding default gw ..."
   if host.name != 'VR' and host.name != 'H1' and
                            host.name != 'WANA' and
                            host.name != 'TC':
      host.setDefaultRoute('dev ' + host.name +
          '-eth0 via ' +
          net.hosts[nhosts - 2].IP(
              net.hosts[nhosts - 2].name + '-eth0'))
   elif host.name == 'TC' or host.name == 'WANA':
      print "Default GW manually configured"
      host.cmd('route add default gw ' +
          net.hosts[nhosts - 2].IP(
              net.hosts[nhosts - 2].name + '-eth0'))
   else:
      #H1 case
      host.setDefaultRoute('dev ' + host.name +
          '-eth0 via ' +
          net.hosts[nhosts - 2].IP(
              net.hosts[nhosts - 2].name + '-eth1'))

#installing TrafficShaper on TC
info('\n*** Installing TrafficShaper on TC\n')
tc.cmd('tc qdisc del dev TC-eth1 root')
tc.cmd('tc qdisc add dev TC-eth1 root handle 1: cbq
       avpkt 1000 bandwidth 1000mbit')
tc.cmd('tc class add dev TC-eth1 parent 1: classid
       1:1 cbq rate 10mbit allot 1500 prio 5 bounded')
tc.cmd('tc filter add dev TC-eth1 parent 1: protocol
       ip prio 16 u32 match ip dst 10.0.0.9 flowid 1:1')
tc.cmd('tc qdisc add dev TC-eth1 parent 1:1 sfq perturb 10')


if debug:
   print "***************** DEBUG MODE ON ****************"
   print "[SWITCH] ", s1, " Number of interfaces is ",
         len(s1.intfs)
   print "List of hosts:"
   for host in net.hosts:
      info( host.name + '\n' )
      print "[HOST " + host.name + " - Interfaces]"
      print host.cmd('ip a')
      print "[HOST " + host.name + " - Routing table]"
      print host.cmd('route -n')
      print "[HOST " + host.name + " - IPv6 status]"
```

```
        print host.cmd(
              'cat /proc/sys/net/ipv6/conf/all/disable_ipv6')
   info('... running CLI \n***')
   CLI(net)
   info('\n')
   info('... stopping Network ***\n')
   net.stop()

#Main
if __name__ == '__main__':
   setLogLevel('info')
   defineNetwork()
```

A.4 contrl2.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/contrl2.py

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 arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from threading import Timer
import os
import json
import time
import datetime

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

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

```
@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
   #
   # We specify NO BUFFER to max_len of the output
   # action due to OVS bug.
   # At this moment, if we specify a lesser number, e.g.,
   # 128, OVS will send Packet-In with invalid buffer_id
   # and truncated packet data.
   # In that case, we cannot output packets correctly.
   # The bug has been fixed in OVS v2.1.0.
   match = parser.OFPMatch()
   actions = [parser.OFPActionOutput(
                    ofproto.OFPP_CONTROLLER,
                    ofproto.OFPCML_NO_BUFFER)]
   self.add_flow(datapath, 0, match, actions)

   self.logger.debug("[TIMER (%s)]",
        datetime.datetime.fromtimestamp(
        time.time()).strftime('%Y-%m-%d %H:%M:%S'))

   self.logger.debug("**installo regole ARP")
        #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")
        #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)

#VARIABILE GLOBALE DELLA PRIORITA' DELLA REGOLA
global h
   h = 1

   # VARIABILI GLOBALI UTILI PER IL RICONOSCIMENTO
   # DELLA CONGESTIONE
```

```
        global bu
        bu=0
        global ru
        ru=0


    def add_flow(self, datapath, priority, match, actions,
                 buffer_id=None, timeout=0):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(
                ofproto.OFPIT_APPLY_ACTIONS, actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath,
                                    idle_timeout=timeout,
                                    buffer_id=buffer_id,
                                    priority=priority,
                                    match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath,
                                    idle_timeout=timeout,
                                    priority=priority,
                                    match=match,
                                    instructions=inst)
          datapath.send_msg(mod)

    @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)
        #ANALISI PACKET_IN
        msg = ev.msg

        global datapath

        datapath = msg.datapath
        ofproto = datapath.ofproto

        global parser
```

```
parser = datapath.ofproto_parser
in_port = msg.match['in_port']

pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]

dst = eth.dst
src = eth.src

pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
    #utile per estrarre gli indirizzi IP

dpid = datapath.id
self.mac_to_port.setdefault(dpid, {})

self.logger.info("packet in %s %s %s %s", dpid, src,
                 dst, in_port)

#VARIABILE GLOBALE PRIORITA' DELLA REGOLA
global h
h = h+1

#REGOLE REDIREZIONAMENTO PACCHETTI AL DPI
# E ALLA DESTINAZIONE
actions = []

if pkt_ipv4 and (in_port==1 or in_port==2 or
                 in_port==8) :
   timeout = 50
   prio = h

   if in_port==1 or (in_port==8 and
                     pkt_ipv4.dst == '10.10.10.1') :
      if pkt_ipv4.proto==6 :
         self.logger.debug("**** TCP pkt from VMU1
                     steered to DPI and port 8 ****")
         match = parser.OFPMatch(in_port = 1,
                       eth_type = eth.ethertype,
                       ip_proto=6)
         actions = [parser.OFPActionOutput(3),
                    parser.OFPActionOutput(8)]
         self.add_flow(datapath, prio, match, actions,
                       timeout=timeout)
```

```
                self.logger.debug("**** TCP pkt to VMU1
                            steered to DPI and VMU1 ****")
                match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.1',
                                eth_type = eth.ethertype,
                                ip_proto=6)
                actions = [parser.OFPActionOutput(3),
                           parser.OFPActionOutput(1)]
                self.add_flow(datapath, prio, match, actions,
                              timeout=timeout)

            elif pkt_ipv4.proto==17 :
                self.logger.debug("**** UDP pkt from VMU1
                            steered to DPI and port 8 ****")
                match = parser.OFPMatch(in_port = 1,
                                eth_type = eth.ethertype,
                                ip_proto=17)
                actions = [parser.OFPActionOutput(3),
                           parser.OFPActionOutput(8)]
                self.add_flow(datapath, prio, match, actions,
                              timeout=timeout)

                self.logger.debug("**** UDP pkt to VMU1
                            steered to DPI and VMU1 ****")
                match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.1',
                                eth_type = eth.ethertype,
                                ip_proto=17)
                actions = [parser.OFPActionOutput(3),
                           parser.OFPActionOutput(1)]
                self.add_flow(datapath, prio, match, actions,
                              timeout=timeout)

        elif in_port==2 or (in_port==8 and
                              pkt_ipv4.dst == '10.10.10.2' ) :
            if pkt_ipv4.proto==6 :
                self.logger.debug("**** TCP pkt from VMU2
                            steered to DPI and port 8 ****")
                match = parser.OFPMatch(in_port = 2,
                                eth_type = eth.ethertype,
                                ip_proto=6)
                actions = [parser.OFPActionOutput(3),
                           parser.OFPActionOutput(8)]
                self.add_flow(datapath, prio, match, actions,
                              timeout=timeout)
```

```
            self.logger.debug("**** TCP pkt to VMU2
                        steered to DPI and VMU2 ****")
            match = parser.OFPMatch(in_port = 8,
                           ipv4_dst = '10.10.10.2',
                           eth_type = eth.ethertype,
                           ip_proto=6)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(2)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

        elif pkt_ipv4.proto==17 :
            self.logger.debug("**** UDP pkt from VMU2
                        steered to DPI and port 8 ****")
            match = parser.OFPMatch(in_port = 2,
                           eth_type = eth.ethertype,
                           ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(8)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**** UDP pkt to VMU2
                        steered to DPI and VMU2 ****")
            match = parser.OFPMatch(in_port = 8,
                           ipv4_dst = '10.10.10.2',
                           eth_type = eth.ethertype,
                           ip_proto=17)
            actions = [parser.OFPActionOutput(3),
                       parser.OFPActionOutput(2)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

    #AVVIO nDPI
    self.logger.debug("[TIMER (%s)]",
        datetime.datetime.fromtimestamp(
                time.time()).strftime(
                    '%Y-%m-%d %H:%M:%S'))
    self.logger.debug("**start ndpiReader")
    os.chdir("/home/green/nDPI_Tool/nDPI/example")
    os.system("sudo ./ndpiReader -i s1-eth3 -v 2 -j
       /home/green/dpi-output/l2-test/netsoft-s1-eth3.json
       &")
    self.logger.debug("*nDPI ATTIVO*")
```

```
            #ATTESA DI 60 SECONDI E FINE ANALISI
            t=Timer(60.0, self.end_analysis)
            t.start()

        else :
            self.logger.debug("******FLUSSO NON AMMESSO******")
        match = parser.OFPMatch(in_port = in_port,
                                eth_type = eth.ethertype,
                                eth_dst=dst)
        actions = []
        timeout=0
        prio=1
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, prio, match, actions,
                          msg.buffer_id, timeout)
            return
        else:
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

        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)


    def end_analysis(self):
        #STOP nDPI
        self.logger.debug("** STOP ndpiReader")
        os.system('sudo kill -2 $(pgrep ndpiReader)')
        time.sleep(0.10)

        #INIZIALIZZAZIONE VARIABILI HOST_A E HOST_B
        host_a="0.0.0.0"
        host_b="0.0.0.0"

        #LETTURA OUTPUT JSON (CLASSIFICAZIONE nDPI)
        json_data = open(
            '/home/green/dpi-output/l2-test/netsoft-s1-eth3.json')
```

```
data = json.load(json_data)
list_of_flows = data["known.flows"]
for i in list_of_flows :
   if (i['protocol'] == "TCP" or
       i['protocol'] == "UDP") and
           (i["host_a.name"] == "10.10.10.1" or
            i["host_b.name"] == "10.10.10.1" or
            i["host_a.name"] == "10.10.10.2" or
            i["host_b.name"] == "10.10.10.2" ) :
      host_a = i["host_a.name"]
      host_b = i["host_b.name"]
      port_a = i["host_a.port"]
      port_b = i["host_n.port"]
      self.logger.debug("[CONTROLLER - TIMER (%s)]
                  Host %s:%s is exchanging packets
                  with Host %s:%s, via %s ",
                  datetime.datetime.fromtimestamp(
                           time.time()).strftime(
                           '%Y-%m-%d %H:%M:%S'),
                  i["host_a.name"], i["host_a.port"],
                  i["host_b.name"], i["host_n.port"],
                  i["protocol"])

#AVVIO CASO DI CONGESTIONAMENTO
global bu
global ru



if host_a == "10.10.10.1" or host_b == "10.10.10.1" :
                          #CODICE RELATIVO AL BUSUSER
   if bu == 0 :
      bu=1
      self.congested_case()
           #AVVIO CASO DI CONGESTIONAMENTO
   else :
      self.logger.debug("**** WARNING: BUSINESS USER'S
                        FLOW ALREADY ACTIVE ****")

if host_a == "10.10.10.2" or host_b == "10.10.10.2" :
                          #CODICE RELATIVO AL RESUSER
   if ru == 0 :
      ru=1
      self.congested_case()
           #AVVIO CASO DI CONGESTIONAMENTO
```

```
            else :
               self.logger.debug("**** WARNING: RESIDENCE USER'S
                                  FLOW ALREADY ACTIVE ****")

         # ATTESA DI 1 SECONDO
         # E POSSIBILE FASE DI NON CONGESTIONAMENTO

         nc=Timer(1.0, self.non_congested_case )
         if (ru == 0 or bu == 0) :
            nc.start()

         json_data.close()


      def congested_case(self):
         self.logger.debug("**** CONGESTED CASE at [TIMER (%s)]",
                           datetime.datetime.fromtimestamp(
                                    time.time()).strftime(
                                         '%Y-%m-%d %H:%M:%S'))

         global ru
         global bu
         global datapath
         global parser

         #AUMENTO PRIORITA'
         global h
         h = h+1

         timeout = 50
         prio = h

         if bu == 1 :
            self.logger.debug("**** STEERING BUSUSER FLOWS ****")

            #REGOLE STEERING PACCHETTI BU AL WANA
            self.logger.debug("**bu_c: 1/8 _ TCP pkt from VMU1
                               steered to WANA (LAN port) ****")
            match = parser.OFPMatch(in_port=1, eth_type = 2048 ,
                                    ip_proto=6)
            actions = [parser.OFPActionOutput(4)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**bu_c: 2/8 _ UDP pkt from VMU1
```

```
                    steered to WANA (LAN port) ****")
match = parser.OFPMatch(in_port=1, eth_type = 2048 ,
                            ip_proto=17)
actions = [parser.OFPActionOutput(4)]
self.add_flow(datapath, prio, match, actions,
              timeout=timeout)

self.logger.debug("**bu_c: 3/8 _ TCP pkt to VMU1
                    steered to WANA (WAN port) ****")
match = parser.OFPMatch(in_port = 8,
                ipv4_dst = '10.10.10.1',
                eth_type = 2048 , ip_proto=6)
actions = [parser.OFPActionOutput(5)]
self.add_flow(datapath, prio, match, actions,
              timeout=timeout)

self.logger.debug("**bu_c: 4/8 _ UDP pkt to VMU1
                    steered to WANA (WAN port) ****")
match = parser.OFPMatch(in_port = 8,
                            ipv4_dst = '10.10.10.1',
                            eth_type = 2048 , ip_proto=17)
actions = [parser.OFPActionOutput(5)]
self.add_flow(datapath, prio, match, actions,
              timeout=timeout)

#STEERING DAL WANA ALLA DESTINAZIONE
self.logger.debug("**bu_c: 5/8 _ TCP pkt from
                    WANA (WAN port) to destination **")
match = parser.OFPMatch(in_port = 5,
                            eth_type = 2048 , ip_proto=6)
actions = [parser.OFPActionOutput(8)]
self.add_flow(datapath, prio, match, actions,
              timeout=timeout)

self.logger.debug("**bu_c: 6/8 _ UDP pkt from
                    WANA (WAN port) to destination **")
match = parser.OFPMatch(in_port = 5, eth_type = 2048 ,
                            ip_proto=17)
actions = [parser.OFPActionOutput(8)]
self.add_flow(datapath, prio, match, actions,
              timeout=timeout)

self.logger.debug("**bu_c: 7/8 _ TCP pkt from
                    WANA (LAN port) to VMU1 ****")
match = parser.OFPMatch(in_port=4,
```

```
                                ipv4_dst = '10.10.10.1',
                                eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**bu_c: 8/8 _ UDP pkt from
                              WANA (LAN port) to VMU1 ****")
            match = parser.OFPMatch(in_port=4,
                                ipv4_dst = '10.10.10.1',
                                eth_type = 2048 , ip_proto=17)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

        if ru == 1 :
            self.logger.debug("**** STEERING RESUSER FLOWS ****")

            #REGOLE STEERING PACCHETTI RU AL TC
            self.logger.debug("**ru_c: 1/8 _ TCP pkt from
                              VMU2 steered to TC ****")
            match = parser.OFPMatch(in_port=2,
                                eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionOutput(6)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**ru_c: 2/8 _ UDP pkt from
                              VMU2 steered to TC ****")
            match = parser.OFPMatch(in_port=2,
                                eth_type = 2048 , ip_proto=17)
            actions = [parser.OFPActionOutput(6)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**ru_c: 3/8 _ TCP pkt to VMU2
                              steered to TC (2nd port) ****")
            match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionOutput(7)]
            self.add_flow(datapath, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**ru_c: 4/8 _ UDP pkt to VMU2
```

```
                    steered to TC (2nd port) ****")
        match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 ,
                                ip_proto=17)
        actions = [parser.OFPActionOutput(7)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        #STEERING DAL TC ALLA DESTINAZIONE
        self.logger.debug("**ru_c: 5/8 _ TCP pkt from
                    TC (2nd port) to destination ****")
        match = parser.OFPMatch(in_port = 7,
                                eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(8)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_c: 6/8 _ UDP pkt from
                    TC (2nd port) to destination ****")
        match = parser.OFPMatch(in_port = 7,
                                eth_type = 2048 , ip_proto=17)
        actions = [parser.OFPActionOutput(8)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_c: 7/8 _ TCP pkt from
                    TC to VMU2 ****")
        match = parser.OFPMatch(in_port = 6,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(2)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_c: 8/8 _ UDP pkt from
                    TC to VMU2 ****")
        match = parser.OFPMatch(in_port = 6,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 , ip_proto=17)
        actions = [parser.OFPActionOutput(2)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

    def non_congested_case(self):
```

```
self.logger.debug("* NON CONGESTED CASE at [TIMER (%s)]",
                  datetime.datetime.fromtimestamp(
                           time.time()).strftime(
                                '%Y-%m-%d %H:%M:%S'))

global ru
global bu
global datapath
global parser

#AUMENTO PRIORITA'
global h
h = h+1

timeout = 50
prio = h

if bu == 1 :
   self.logger.debug("**** RESTORING BUSUSER FLOWS ****")

   #REGOLE PACCHETTI BU
   self.logger.debug("**bu_nc: 1/4 _ TCP pkt from
                     VMU1 to destination ****")
   match = parser.OFPMatch(in_port=1,
                           eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionOutput(8)]
   self.add_flow(datapath, prio, match, actions,
                 timeout=timeout)

   self.logger.debug("**bu_nc: 2/4 _ UDP pkt from
                     VMU1 to destination ****")
   match = parser.OFPMatch(in_port=1,
                           eth_type = 2048 , ip_proto=17)
   actions = [parser.OFPActionOutput(8)]
   self.add_flow(datapath, prio, match, actions,
                 timeout=timeout)

   self.logger.debug("**bu_nc: 3/4 _ TCP pkt to VMU1 **")
   match = parser.OFPMatch(in_port = 8,
                           ipv4_dst = '10.10.10.1',
                           eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionOutput(1)]
   self.add_flow(datapath, prio, match, actions,
                 timeout=timeout)
```

```
        self.logger.debug("**bu_nc: 4/4 _ UDP pkt to VMU1 **")
        match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.1',
                                eth_type = 2048 , ip_proto=17)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

    if ru == 1 :
        self.logger.debug("**** RESTORING RESUSER FLOWS ****")

        #REGOLE PACHETTI RU
        self.logger.debug("**ru_nc: 1/4 _ TCP pkt from
                           VMU2 to destination ****")
        match = parser.OFPMatch(in_port=2,
                                eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(8)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_nc: 2/4 _ UDP pkt from
                           VMU2 to destination ****")
        match = parser.OFPMatch(in_port=2,
                                eth_type = 2048 , ip_proto=17)
        actions = [parser.OFPActionOutput(8)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_nc: 3/4 _ TCP pkt to VMU2 **")
        match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(2)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_nc: 4/4 _ UDP pkt to VMU2 **")
        match = parser.OFPMatch(in_port = 8,
                                ipv4_dst = '10.10.10.2',
                                eth_type = 2048 , ip_proto=17)
        actions = [parser.OFPActionOutput(2)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)
```

A.5 gettimestamp.c

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/gettimestamp.c
```

Il codice è stato fornito da Francesco Foresta

```
#include <sys/time.h>
#include <stdio.h>

main (void) {

        struct timeval currentTime;

        gettimeofday(&currentTime,NULL);
        printf("%.6f\n",(double)currentTime.tv_sec+
               currentTime.tv_usec/1000000.0);

}
```

A.6 rcvdatafw.sh

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/rcvdatafw.sh
```

Il codice è stato fornito da Francesco Foresta

```
#! /bin/bash

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

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

ovsname=$1
```

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

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

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

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

        sleep $period
done
```

A.7 test-l3-scenario.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/test-l3-scenario.py

#!/usr/bin/python
```

```
'''
TOPOLOGY USED IN NETSOFT 2015 TEST

Emulation of l3-scenario for the OpenStack case. Nodes:
- Host  VM-User1: one interface
- Host  VM-User2: one interface
- Host DPI: 2 interfaces
- Host WAN Accelerator (WANA): 2 interfaces
  (eth0 connected to s1 and eth1 connected to s2)
- Host TC: 2 interfaces
  (eth0 connected to s1 and eth1 connected to s2)
- Host GW: 2 interfaces
- Host VR: 2 interfaces
- Host h1: one interfaces
- Two Open vSwitch

1) sudo python test-l3-scenario.py
'''

from mininet.net import Mininet
from mininet.node import Node
from mininet.node import Host
from mininet.link import TCLink
from mininet.link import Intf
from mininet.log import setLogLevel, info
from mininet.cli import CLI
from mininet.node import Controller
from mininet.node import RemoteController
from mininet.util import quietRun

from time import sleep
import os
import sys

def defineNetwork():
   if len(sys.argv) < 2:
      print "Missing paramenter: python test-l2-scenario.py
            <debug=1|0>"
      sys.exit()
   #commento
   debug = sys.argv[1] #print some useful information

   info("*** Create an empty network and add nodes and
        switch to it *** \n")
   net = Mininet(controller=RemoteController, link=TCLink,
```

```
                build=False, xterms=True) #MyPOXController
    info("\n*** Adding Controller: Controller will be
         external *** \n")

    #Creazione Open vSwitch
    info("\n*** Creating Switch *** \n")
    s1 = net.addSwitch('s1')
    s1.cmd('ovs-vsctl del-br ' + s1.name )
    s1.cmd('ovs-vsctl add-br ' + s1.name )
    s1.cmd('ovs-vsctl set Bridge '+ s1.name +
           ' stp_enable=false protocols=OpenFlow13' )
           # Disabling STP
    s2 = net.addSwitch('s2')
    s2.cmd('ovs-vsctl del-br ' + s2.name )
    s2.cmd('ovs-vsctl add-br ' + s2.name )
    s2.cmd('ovs-vsctl set Bridge '+ s2.name +
           ' stp_enable=false protocols=OpenFlow13' )
           # Disabling STP

    #Creazione Host
    info("\n*** Creating VM-User 1 *** \n")
    vmu1 = net.addHost('VMU1')
    info("\n*** Creating VM-User 2 *** \n")
    vmu2 = net.addHost('VMU2')
    info("\n*** Creating DPI *** \n")
    dpi = net.addHost('DPI')
    info("\n*** Creating WAN A. *** \n")
    wana = net.addHost('WANA')
    info("\n*** Creating TC *** \n")
    tc = net.addHost('TC')
    info("\n*** Creating GateWay *** \n")
    gw = net.addHost('GW')
    info("\n*** Creating Virtual Router *** \n")
    vr = net.addHost('VR')
    info("\n*** Creating External Host *** \n")
    h1 = net.addHost('H1')
    info("\n*** Creating Links *** \n")

    #Creazione Link
    net.addLink(vmu1, s1, bw=100)
    net.addLink(vmu2, s1, bw=100)
    net.addLink(dpi, s1, bw=100)
    net.addLink(dpi, s2, bw=100)
    net.addLink(wana, s1, bw=100)
    net.addLink(wana, s2, bw=100)
```

```
net.addLink(tc, s1, bw=100)
net.addLink(tc, s2, bw=100)
net.addLink(gw, s1, bw=100)
net.addLink(gw, s2, bw=100)
net.addLink(vr, s2, bw=100)
net.addLink(vr, h1, bw=100)

#Trying to assign MAC address to each node of the topology
vmu1.setMAC("00:00:00:00:00:01", vmu1.name + "-eth0")
vmu2.setMAC("00:00:00:00:00:02", vmu2.name + "-eth0")
dpi.setMAC("00:00:00:00:00:03", dpi.name + "-eth0")
dpi.setMAC("00:00:00:00:00:04", dpi.name + "-eth1")
wana.setMAC("00:00:00:00:00:05", wana.name + "-eth0")
wana.setMAC("00:00:00:00:00:06", wana.name + "-eth1")
tc.setMAC("00:00:00:00:00:07", tc.name + "-eth0")
tc.setMAC("00:00:00:00:00:08", tc.name + "-eth1")
gw.setMAC("00:00:00:00:00:09", gw.name + "-eth0")
gw.setMAC("00:00:00:00:00:0A", gw.name + "-eth1")
vr.setMAC("00:00:00:00:00:0B", vr.name + "-eth0")
vr.setMAC("00:00:00:00:00:0C", vr.name + "-eth1")
h1.setMAC("00:00:00:00:00:0D", h1.name + "-eth0")

#Disabling IPv6
info('\n*** Disabling IPv6 ...\n')
vmu1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vmu2.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
dpi.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
wana.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
tc.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
gw.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
gw.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
gw.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
vr.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')
```

```
h1.cmd('sysctl -w net.ipv6.conf.all.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.default.disable_ipv6=1')
h1.cmd('sysctl -w net.ipv6.conf.lo.disable_ipv6=1')


#Aggiunta interfacce agli switch
info('\n*** Adding interfaces to switches ***\n')
for intf in s1.intfs.values():
   s1.cmd( 'ovs-vsctl add-port ' + s1.name + ' %s' % intf )
   print "Eseguito comando: ovs-vsctl add-port s1 ", intf

for intf in s2.intfs.values():
   s2.cmd( 'ovs-vsctl add-port ' + s2.name + ' %s' % intf )
   print "Eseguito comando: ovs-vsctl add-port s2 ", intf



info("\n*** Starting Network using Open vSwitch and
     remote controller*** \n")
# Creating a Linux Bridge on each host
nhosts = len(net.hosts)
print 'Total number of hosts: ' + str(nhosts)
count = 1

#net.build() #Allow to build xterm for each node
net.start()

#Configurazione Open vSwitch al protocollo 1.3
s1.cmd('ovs-vsctl set bridge ' + s1.name +
       ' protocols=OpenFlow13') #OpenFlow 1.3
s2.cmd('ovs-vsctl set bridge ' + s2.name +
       ' protocols=OpenFlow13') #OpenFlow 1.3

# Set the controller for the switch
print "Switch name: ", s1.name, s2.name
s1.cmd('ovs-vsctl set-controller ' +  s1.name +
       ' tcp:127.0.0.1:6633')
s2.cmd('ovs-vsctl set-controller ' +  s2.name +
       ' tcp:127.0.0.1:6633')
info( '\n*** Waiting for switch to connect to controller' )
while 'is_connected' not in quietRun( 'ovs-vsctl show' ):
   sleep( 1 )
   info( '.' )
info('\n')
```

```
#IP Configuration
info('\n*** Going to take down default configuration ...\n')
info('\n*** ... and configuring interfaces \n')
for host in net.hosts:
   print 'Deleting ip address on ' + host.name +
         '-eth0 interface ...'
   host.cmd('ip addr del ' + host.IP(host.name + '-eth0') +
            '/8 dev ' + host.name + '-eth0')
   print 'Deleting entry in IP routing table on ' +
         host.name
   host.cmd('ip route del 10.0.0.0/8')
   print "Going to configure new IP"
   if host.name == 'VMU1' or host.name == 'VMU2' :
      host.setIP("10.10.10." + str(count), 24, host.name +
                 "-eth0")
      print "[CURRENT-CHECK] IP eth0: " +
         net.hosts[count - 1].IP(net.hosts[count - 1].name +
         '-eth0')
   elif host.name == 'DPI'  or host.name == 'WANA' or
        host.name == 'TC' or host.name == 'GW' :
      host.setIP("10.10.10." + str(count), 24, host.name +
                 "-eth0")
      print "[CURRENT-CHECK] IP eth0: " +
         net.hosts[count - 1].IP(net.hosts[count - 1].name +
         '-eth0')
      host.setIP("10.20.20." + str(count-2), 24, host.name +
                 "-eth1")
      print "[CURRENT-CHECK] IP eth1: " +
         net.hosts[count - 1].IP(net.hosts[count - 1].name +
         '-eth1')
      host.cmd('sysctl -w net.ipv4.ip_forward=1')
      print "IP Forwarding enabled!"
   elif host.name == 'H1':
      host.setIP("10.0.0." + str(count + 1), 30, host.name +
                 '-eth0')
      print "[CURRENT-CHECK] H1 IP eth0: " +
         net.hosts[count - 1].IP(net.hosts[count - 1].name +
         '-eth0')
      net.hosts[count - 2].setIP("10.0.0." + str(count + 2),
          30, net.hosts[count - 2].name + "-eth1")
      print net.hosts[count - 2].name +
          "-eth1 interface has been configured!"
      print "[Checking VR IP] eth1: " +
          net.hosts[count - 2].IP('VR-eth1')
      net.hosts[count - 2].cmd('sysctl -w
```

```
            net.ipv4.ip_forward=1')
         print "On VR node: IP Forwarding enabled!"
      else:
         host.setIP("10.20.20." + str(count-2), 24, host.name +
                    "-eth0")
         print "[CURRENT-CHECK] IP eth0: " +
            net.hosts[count - 1].IP(net.hosts[count - 1].name +
            '-eth0')
      count = count + 1
      print "\n"

   #Gateway Configuration
   print "Configuring gateway on each host.."
   count=1
   for host in net.hosts:
      print "Adding gateway ..."
      if host.name == 'VMU1' or host.name == 'VMU2' :
         host.setDefaultRoute('dev ' + host.name +
           '-eth0 via ' +
           net.hosts[nhosts - 3].IP(net.hosts[nhosts - 3].name
           + '-eth0'))
      elif host.name == 'VR' :
         host.setDefaultRoute('dev ' + host.name +
           '-eth0 via ' +
           net.hosts[nhosts - 3].IP(net.hosts[nhosts - 3].name
           + '-eth1'))
      elif host.name == 'GW' :
         host.setDefaultRoute('dev ' + host.name +
           '-eth1 via ' +
           net.hosts[nhosts - 2].IP(net.hosts[nhosts - 2].name
           + '-eth0'))
      elif host.name == 'H1' :
         host.setDefaultRoute('dev ' + host.name +
           '-eth0 via ' +
           net.hosts[nhosts - 2].IP(net.hosts[nhosts - 2].name
           + '-eth1'))
   elif host.name == 'DPI' or host.name == 'WANA' or
        host.name == 'TC' :
      host.cmd('route add -net 10.0.0.8 netmask 255.255.255.252
               gw 10.20.20.5')

   #installing TrafficShaper on TC
   info('\n*** Installing TrafficShaper on TC\n')
   tc.cmd('tc qdisc del dev TC-eth1 root')
   tc.cmd('tc qdisc add dev TC-eth1 root handle 1: cbq avpkt
```

```
            1000 bandwidth 1000mbit')
    tc.cmd('tc class add dev TC-eth1 parent 1: classid 1:1 cbq
            rate 10mbit allot 1500 prio 5 bounded')
    tc.cmd('tc filter add dev TC-eth1 parent 1: protocol ip prio
            16 u32 match ip dst 10.0.0.9 flowid 1:1')
    tc.cmd('tc qdisc add dev TC-eth1 parent 1:1 sfq perturb 10')

    #Modalita' di debug

    if debug:
       print "*************** DEBUG MODE ON ***************"
       print "[SWITCH] ", s1, " Number of interfaces is ",
              len(s1.intfs)
       print "[SWITCH] ", s2, " Number of interfaces is ",
              len(s2.intfs)
       print "List of hosts:"
       for host in net.hosts:
          info( host.name + '\n' )
          print "[HOST " + host.name + " - Interfaces]"
          print host.cmd('ip a')
          print "[HOST " + host.name + " - Routing table]"
          print host.cmd('route -n')
          print "[HOST " + host.name + " - IPv6 status]"
          print host.cmd(
                'cat /proc/sys/net/ipv6/conf/all/disable_ipv6')
    info('... running CLI \n***')
    CLI(net)
    info('\n')
    info('... stopping Network ***\n')
    net.stop()

#Main
if __name__ == '__main__':
    setLogLevel('info')
    defineNetwork()
```

A.8 contrl3.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/contrl3.py
```

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller import dpset
from ryu.controller.handler import CONFIG_DISPATCHER,
                    MAIN_DISPATCHER, DEAD_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 arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from ryu.ofproto import inet
from ryu.lib.packet import ether_types as ether
from threading import Timer
import os
import json
import time
import datetime

class BasicOpenStackL3Controller(app_manager.RyuApp):
   OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
   _CONTEXTS = {'dpset': dpset.DPSet}

   # VARIABILI GLOBALI UTILI PER IL RICONOSCIMENTO
   # DELLA CONGESTIONE
   global bu
   bu=0
   global ru
   ru=0
   global prio
   prio=1

   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
```

```
# FUNCTIONS
def _get_ports_info(self, dpid): #Return information about
            # all port on a switch
   return self.dpset.get_ports(dpid)

def _get_port_name(self, dpid, port): #Return the name
            # associated to the specified port number
            # on the specified switch
   return self.dpset.get_port(dpid, port).name

# Handle reception of StateChange message
# (NOTE. the message is sent whenever a switch
# performs the handshake with controller)
@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def state_change_handler(self, ev):
datapath = ev.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

if ev.state == MAIN_DISPATCHER:
   port_name = self._get_port_name(datapath.id, 4294967294)
   if not datapath.id in self.switch_dpid_name:
      self.switch_dpid_name[datapath.id] = port_name
      self.logger.debug("[TIMER (%s)] [SC-HANDLER]
            Switch %s registered on the controller,
            datapath.id  =  %s",
            datetime.datetime.fromtimestamp(
                     time.time()).strftime(
                           '%Y-%m-%d %H:%M:%S'),
            str(port_name), datapath.id)
      self.connections_name_dpid[str(port_name)] = datapath

      # Tell the eswitch to send msgs to the Controller
      # in case of table miss
      match = parser.OFPMatch()
      actions = [parser.OFPActionOutput(
                ofproto.OFPP_CONTROLLER,
                ofproto.OFPCML_NO_BUFFER)]
      self.add_flow(datapath, 0, match, actions)

   # 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 ARP")


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

    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.switch_dpid_name:
            self.logger.debug("[TIMER (%s)] [SC-HANDLER]
                    Switch %s disconnected",
                    datetime.datetime.fromtimestamp(
                            time.time()).strftime(
                                 '%Y-%m-%d %H:%M:%S'),
            self.switch_dpid_name[datapath.id])
            del self.connections_name_dpid[
                     self.switch_dpid_name[datapath.id]]
            del self.switch_dpid_name[datapath.id]


def connectionForBridge(self, bridge):
    return self.connections_name_dpid[bridge]

#Add flow
def add_flow(self, datapath, priority, match, actions,
             buffer_id=None, timeout=0):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, actions)]

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

```
                    instructions=inst)

        datapath.send_msg(mod)

    #Handle PacketIn event
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = ev.msg.datapath
        global ofproto
        ofproto = datapath.ofproto
        global parser
        parser = datapath.ofproto_parser
        pkt = packet.Packet(msg.data)

        in_port = msg.match['in_port']

        self.logger.debug("[PACKET-IN] (%s) in port: %s ,
                           datapath id: %s",
                           datetime.datetime.fromtimestamp(
                                    time.time()).strftime(
                                         '%Y-%m-%d %H:%M:%S'),
                           in_port, datapath.id)

        np = len(pkt.get_protocols(ethernet.ethernet))
        self.logger.debug("[PKT-HANDLER] (%s)
                           Number of detected protocols: %d",
                           datetime.datetime.fromtimestamp(
                               time.time()).strftime(
                                    '%Y-%m-%d %H:%M:%S'), np)
        if np == 1:
            eth = pkt.get_protocols(ethernet.ethernet)[0]
            eth_type = eth.ethertype
            eth_src = eth.src
            eth_dst = eth.dst
            self.logger.debug("[PKT-HANDLER] (%s)
                    PacketIn from DPID = %s - in_port=%d
                    dl_type=%s eth_src=%s eth_dst=%s",
                    datetime.datetime.fromtimestamp(
                        time.time()).strftime(
                             '%Y-%m-%d %H:%M:%S'),
                    ev.msg.datapath.id, in_port,
                    str(eth_type), str(eth_src),
                    str(eth_dst))
```

```
if eth_type == ether.ETH_TYPE_IP: # IP packets ...
   ip_pkt = pkt.get_protocol(ipv4.ipv4)
   ip_proto = ip_pkt.proto
   ip_src = ip_pkt.src
   ip_dst = ip_pkt.dst
   if ip_proto == 6 and (in_port==1 or
                         in_port==2): # ..TCP
      self.logger.debug("[PKT-HANDLER] (%s)
               TCP packets detected ",
               datetime.datetime.fromtimestamp(
                   time.time()).strftime(
                        '%Y-%m-%d %H:%M:%S'))
      self.logger.debug("[PKT-HANDLER] (%s)
               CHECKING protocol detected -
               ip_proto=%s ip_src=%s
               ip_dst=%s inet_PROTO=%s",
               datetime.datetime.fromtimestamp(
                   time.time()).strftime(
                        '%Y-%m-%d %H:%M:%S'),
               ip_proto, ip_src,
               ip_dst, inet.IPPROTO_TCP)

      #TIMEOUT GLOBALE
      global timeout
      timeout=40

      #PRIORITA'
      global prio
      prio=prio+1

      # STEERING TCP TRAFFIC
      match = parser.OFPMatch(in_port = in_port,
                eth_type = ether.ETH_TYPE_IP,
                ip_proto = inet.IPPROTO_TCP,
                ipv4_src = ip_src,
                ipv4_dst = ip_dst)
      actions = [parser.OFPActionSetField(
                   eth_dst="00:00:00:00:00:03"),
                 parser.OFPActionOutput(3)]
      self.add_flow(datapath, prio, match,
               actions, timeout=timeout)

      dp = self.connectionForBridge("s2")
      match = parser.OFPMatch(in_port = 1,
                eth_type = ether.ETH_TYPE_IP,
```

```
                ip_proto = inet.IPPROTO_TCP,
                ipv4_src = ip_src,
                ipv4_dst = ip_dst)
        actions = [parser.OFPActionOutput(5)]
        self.add_flow(dp, prio, match, actions,
                      timeout=timeout)

        #Handle bidirectional flows
        match = parser.OFPMatch(in_port = 3,
                eth_type = ether.ETH_TYPE_IP,
                ip_proto = inet.IPPROTO_TCP,
                ipv4_src = ip_dst,
                ipv4_dst = ip_src)
        actions = [parser.OFPActionOutput(in_port)]
        self.add_flow(datapath, prio, match, actions,
                      timeout=timeout)

        match = parser.OFPMatch(in_port = 5,
                eth_type = ether.ETH_TYPE_IP,
                ip_proto = inet.IPPROTO_TCP,
                ipv4_src = ip_dst,
                ipv4_dst = ip_src)
        actions = [parser.OFPActionSetField(
                   eth_dst="00:00:00:00:00:04"),
                   parser.OFPActionOutput(1)]
        self.add_flow(dp, prio, match, actions,
                      timeout=timeout)

        #AVVIO nDPI
        self.logger.debug("[TIMER (%s)]",
                 datetime.datetime.fromtimestamp(
                     time.time()).strftime(
                          '%Y-%m-%d %H:%M:%S'))
        self.logger.debug("**start ndpiReader")
        os.chdir("/home/green/nDPI_Tool/nDPI/example")
           os.system("sudo
           ./ndpiReader -i s1-eth3 -v 2 -j
           /home/green/dpi-output/l2-test/
                        netsoft-s1-eth3.json &")
        self.logger.debug("*nDPI ATTIVO*")

        #ATTESA DI 60 SECONDI E FINE ANALISI
        t=Timer(60.0, self.end_analysis)
        t.start()
    else:
```

```
            self.logger.debug("[PKT-HANDLER]
                    Other protocol detected: %s",
                    str(eth_type))
        else:
            self.logger.debug("More than one protocol detected")

    def end_analysis(self):
        #STOP nDPI
        self.logger.debug("** STOP ndpiReader")
        os.system('sudo kill -2 $(pgrep ndpiReader)')
        time.sleep(0.10)

        #INIZIALIZZAZIONE VARIABILI HOST_A E HOST_B
        host_a="0.0.0.0"
        host_b="0.0.0.0"

        #LETTURA OUTPUT JSON (CLASSIFICAZIONE nDPI)
        json_data = open(
            '/home/green/dpi-output/l2-test/netsoft-s1-eth3.json
            ')
        data = json.load(json_data)
        list_of_flows = data["known.flows"]
        for i in list_of_flows :
            if ( i['protocol'] == "TCP" or
                 i['protocol'] == "UDP" ) and (
                      i["host_a.name"] == "10.10.10.1" or
                      i["host_b.name"] == "10.10.10.1" or
                      i["host_a.name"] == "10.10.10.2" or
                      i["host_b.name"] == "10.10.10.2" ) :
                host_a = i["host_a.name"]
                host_b = i["host_b.name"]
                port_a = i["host_a.port"]
                port_b = i["host_n.port"]
                self.logger.debug("[CONTROLLER - TIMER (%s)]
                      Host %s:%s is exchanging packets
                      with Host %s:%s, via %s ",
                      datetime.datetime.fromtimestamp(
                          time.time()).strftime(
                          '%Y-%m-%d %H:%M:%S'),
                      i["host_a.name"], i["host_a.port"],
                      i["host_b.name"], i["host_n.port"],
                      i["protocol"])
        #AVVIO CASO DI CONGESTIONAMENTO
        global bu
        global ru
```

```
        if host_a == "10.10.10.1" or host_b == "10.10.10.1":
                                    #CODICE RELATIVO AL BUSUSER
            if bu == 0 :
                bu=1

        if host_a == "10.10.10.2" or host_b == "10.10.10.2":
                                    #CODICE RELATIVO AL RESUSER
            if ru == 0 :
                ru=1

    #FASE DI CONGESTIONAMENTO
    if bu==1 or ru==1 :
        self.congested_case()
             #AVVIO CASO DI CONGESTIONAMENTO

        # ATTESA DI 1 SECONDO E
        # POSSIBILE FASE DI NON CONGESTIONAMENTO
        nc=Timer(1.0, self.non_congested_case )
        if (ru == 0 or bu == 0) :
            nc.start()

        json_data.close()

    def congested_case(self):
        self.logger.debug("** CONGESTED CASE at [TIMER (%s)]",
                    datetime.datetime.fromtimestamp(
                        time.time()).strftime(
                             '%Y-%m-%d %H:%M:%S'))

        global ru
        global bu
        global parser
        global ofproto

        #AUMENTO PRIORITA'
        global prio
        prio = prio+1

        global timeout

        #DATAPATH SWITCH 1 e 2
        dp1 = self.connectionForBridge("s1")
        dp2 = self.connectionForBridge("s2")
```

```
if bu == 1 :
   self.logger.debug("**** STEERING BUSUSER FLOWS ****")

   #REGOLE STEERING SWITCH 1
   self.logger.debug("**bu_c: 1 _ TCP pkt from
          VMU1 steered to WANA (LAN port) ****")
   match = parser.OFPMatch(in_port=1,
                  eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionSetField(
               eth_dst="00:00:00:00:00:05"),
              parser.OFPActionOutput(4)]
   self.add_flow(dp1, prio, match, actions,
                 timeout=timeout)

   self.logger.debug("**bu_c: 2 _ TCP pkt from
          WANA (LAN port) to VMU1 ****")
   match = parser.OFPMatch(in_port=4,
                  ipv4_dst = '10.10.10.1',
                  eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionOutput(1)]
   self.add_flow(dp1, prio, match, actions,
                 timeout=timeout)

   #REGOLE STEERING SWITCH 2
   self.logger.debug("**bu_c: 3 _ TCP pkt to
          VMU1 steered to WANA (WAN port) ****")
   match = parser.OFPMatch(in_port = 5,
                  ipv4_dst = '10.10.10.1',
                  eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionSetField(
               eth_dst="00:00:00:00:00:06"),
              parser.OFPActionOutput(2)]
   self.add_flow(dp2, prio, match, actions,
                 timeout=timeout)

   self.logger.debug("**bu_c: 4 _ TCP pkt from
          WANA (WAN port) to destination ****")
   match = parser.OFPMatch(in_port = 2,
                  eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionOutput(5)]
   self.add_flow(dp2, prio, match, actions,
                 timeout=timeout)
if ru == 1 :
   self.logger.debug("**** STEERING RESUSER FLOWS ****")
```

```
        #REGOLE STEERING SWITCH 1
        self.logger.debug("**ru_c: 1 _ TCP pkt from
               VMU2 steered to TC ****")
        match = parser.OFPMatch(in_port=2,
                       eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionSetField(
                     eth_dst="00:00:00:00:00:07"),
                   parser.OFPActionOutput(5)]
        self.add_flow(dp1, prio, match, actions,
                       timeout=timeout)

        self.logger.debug("**ru_c: 2 _ TCP pkt from
               TC to VMU2 ****")
        match = parser.OFPMatch(in_port = 5,
                       ipv4_dst = '10.10.10.2',
                       eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(2)]
        self.add_flow(dp1, prio, match, actions,
                       timeout=timeout)

        #STEERING DAL TC ALLA DESTINAZIONE
        self.logger.debug("**ru_c: 3 _ TCP pkt from
               TC (2nd port) to destination ****")
        match = parser.OFPMatch(in_port = 3,
                       eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(5)]
        self.add_flow(dp2, prio, match, actions,
                       timeout=timeout)

        self.logger.debug("**ru_c: 4 _ TCP pkt to
               VMU2 steered to TC (2nd port) ****")
        match = parser.OFPMatch(in_port = 5,
                       ipv4_dst = '10.10.10.2',
                       eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionSetField(
                     eth_dst="00:00:00:00:00:08"),
                   parser.OFPActionOutput(3)]
        self.add_flow(dp2, prio, match, actions,
                       timeout=timeout)

    def non_congested_case(self):
        self.logger.debug(
                "** NON CONGESTED CASE at [TIMER (%s)]",
                datetime.datetime.fromtimestamp(
                    time.time()).strftime(
```

```
                    '%Y-%m-%d %H:%M:%S'))

global ru
global bu
global parser
global ofproto

#AUMENTO PRIORITA'
global prio
prio = prio+1

global timeout

#DATAPATH SWITCH 1 e 2
dp1 = self.connectionForBridge("s1")
dp2 = self.connectionForBridge("s2")

if bu == 1 :
   self.logger.debug("**** RESTORING BUSUSER FLOWS ****")

   #REGOLE PACCHETTI BU
   self.logger.debug("**bu_nc: 1 _ TCP pkt from
                      VMU1 to destination ****")
   match = parser.OFPMatch(in_port=1,
                   eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionSetField(
                eth_dst="00:00:00:00:00:09"),
              parser.OFPActionOutput(6)]
   self.add_flow(dp1, prio, match, actions,
                  timeout=timeout)

   self.logger.debug("**bu_nc: 2 _ TCP pkt from
                      VMU1 to destination ****")
   match = parser.OFPMatch(in_port=6,
                   ipv4_dst = '10.10.10.1',
                   eth_type = 2048 , ip_proto=6)
   actions = [parser.OFPActionOutput(1)]
   self.add_flow(dp1, prio, match, actions,
                  timeout=timeout)

   self.logger.debug("**bu_nc: 3 _ TCP pkt to
                      VMU1 ****")
   match = parser.OFPMatch(in_port = 5,
                   ipv4_dst = '10.10.10.1',
                   eth_type = 2048 , ip_proto=6)
```

```
            actions = [parser.OFPActionSetField(
                        eth_dst="00:00:00:00:00:0A"),
                       parser.OFPActionOutput(4)]
            self.add_flow(dp2, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**bu_nc: 4 _ TCP pkt to
                              VMU1 ****")
            match = parser.OFPMatch(in_port = 4,
                           eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionOutput(5)]
            self.add_flow(dp2, prio, match, actions,
                          timeout=timeout)

        if ru == 1 :
            self.logger.debug("**** RESTORING RESUSER FLOWS ****")

            #REGOLE PACHETTI RU
            self.logger.debug("**ru_nc: 1 _ TCP pkt from
                              VMU2 to destination ****")
            match = parser.OFPMatch(in_port=2,
                           eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionSetField(
                        eth_dst="00:00:00:00:00:09"),
                       parser.OFPActionOutput(6)]
            self.add_flow(dp1, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**ru_nc: 2 _ TCP pkt from
                              VMU2 to destination ****")
            match = parser.OFPMatch(in_port=6,
                           ipv4_dst = '10.10.10.2',
                           eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(dp1, prio, match, actions,
                          timeout=timeout)

            self.logger.debug("**ru_nc: 3 _ TCP pkt to
                              VMU2 ****")
            match = parser.OFPMatch(in_port = 5,
                           ipv4_dst = '10.10.10.2',
                           eth_type = 2048 , ip_proto=6)
            actions = [parser.OFPActionSetField(
                        eth_dst="00:00:00:00:00:0A"),
                       parser.OFPActionOutput(4)]
```

```
        self.add_flow(dp2, prio, match, actions,
                      timeout=timeout)

        self.logger.debug("**ru_nc: 4 _ TCP pkt to
                          VMU2 ****")
        match = parser.OFPMatch(in_port = 4,
                       eth_type = 2048 , ip_proto=6)
        actions = [parser.OFPActionOutput(5)]
        self.add_flow(dp2, prio, match, actions,
                      timeout=timeout)
```

A.9 controllerl3_final.py

Codice disponibile presso:

```
https://github.com/luigiponti/laureatriennale/
        blob/master/controllerl3_final.py
```

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller import dpset
from ryu.controller.handler import CONFIG_DISPATCHER,
                    MAIN_DISPATCHER, DEAD_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 arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from ryu.lib.packet import tcp
from ryu.ofproto import inet
from ryu.lib.packet import ether_types as ether
from threading import Timer
import os
import json
import time
import datetime
import subprocess
import threading

class BasicOpenStackL3Controller(app_manager.RyuApp):
   OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

```
_CONTEXTS = {'dpset': dpset.DPSet}

# INITIALIZE GLOBAL VARIABLES
global flow_id
flow_id=0
global flows_state
flows_state = [] # List of flows
global active_flows
active_flows = [] # List of active flows
global classified_flows
classified_flows = [] # List of classified flows
            #(after the preventive enforcement)
global  hipriousers
hipriousers = [] #List of high priority users
global flowlimit
flowlimit = 3        #Max flow in Not Enforcement case
global wanaproto
wanaproto = [200, 201, 202]
            #Protocols of compressed packets

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
   #Set of general parameters
   self.net_topo = ['br-int', 'br4', 'br3'] # NT
   self.users = ['BusUser', 'ResUser'] # U
   self.net_func = ['DPI', 'TC', 'Wana', 'WanaDec',
            'gw_dest', 'vr', 'vr_dest'] # NF

   #self.int_network_vid = 9
   #self.gw_network_vid = 10
   self.outport = 1
   self.outport_dest = 1
   self.sink = 3

   self.bususer_port={'port1': 88, 'port2': None}
   self.resuser_port={'port1': 87, 'port2': None}
```

```
        self.wana_port={'port1': 113, 'port2': 114}
        self.tc_port={'port1': 103, 'port2': 104}
        self.dpi_port={'port1': 106, 'port2': 107}
        #self.gw_port={'port1': , 'port2': }
        self.wana_dest_port={'port1': 2, 'port2':1 }
        self.gw_dest_port={'port1': 3, 'port2': 2}

        self.bcast = "ff:ff:ff:ff:ff:ff"
        self.ip_resuser = "192.168.8.35"
        self.ip_bususer = "192.168.8.36"
        #self.ip_gw = "10.10.0.1"
        #self.ip_router_vr = "10.0.0.1"
        self.ip_nat_bu = "10.250.0.115"
        self.ip_nat_ru = "10.250.0.116"
        self.ip_sink = "10.30.0.2"

        # HIGH PRIORITY USERS LIST
        global hipriousers
        hipriousers=[self.ip_bususer]

        # MAC DICTIONARY
        self.dpi_mac={'eth1':"fa:16:3e:42:9c:51",
                    'eth2':"fa:16:3e:4f:00:e5"}
        self.wana_mac={'eth1':"fa:16:3e:fe:76:34",
                    'eth2':"fa:16:3e:8d:6f:d9"}
        self.tc_mac={'eth1':"fa:16:3e:d9:5b:b9",
                    'eth2':"fa:16:3e:75:3a:e7"}
        #self.gw_mac={'eth1':"fa:16:3e:f1:cc:7b",
        #             'eth2':"fa:16:3e:a4:16:7a"}
        self.wana_dest_mac={'eth1':"52:54:00:a9:08:67",
                    'eth2':"52:54:00:8c:55:14"}
        self.gw_dest_mac={'eth1':"52:54:00:bb:3c:d1", '
                    eth2':"52:54:00:12:5f:5f"}


        self.dpiExePath = '/home/ubuntu/nDPI_Tool/
                           nDPI/example/ndpiReader'
        self.dpiCapPath = '/tmp/ndpiresult_'
        self.a = 0
        self.b = 0
        self.DPIcredentials='ubuntu@192.168.122.64'

    # GET MAC ADDRESS
    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]

    # GET IN PORT
    def get_in_port(self, host, direction):
        result_tuple = []

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

```
            #result_tuple.append(self.gw_port['port1'])
        #elif direction == 'inbound':
            #result_tuple.append('br-int')
            #result_tuple.append(self.gw_port['port2'])
    elif host =='WanaDec' :
        if direction == 'outbound':
            result_tuple.append('br3')
            result_tuple.append(self.wana_dest_port[
                                'port1'])
        elif direction == 'inbound':
            result_tuple.append('br4')
            result_tuple.append(self.wana_dest_port[
                                'port2'])
    elif host =='GWDest' :
        if direction == 'outbound':
            result_tuple.append('br3')
            result_tuple.append(self.gw_dest_port['port1'])
        elif direction == 'inbound':
            result_tuple.append('br4')
            result_tuple.append(self.gw_dest_port['port2'])
    return result_tuple

# ANALIZE ACTIVE FLOWS REQUEST
def _request_stats(self):
    self.logger.debug('FLOW CONTROL request')
    datapath = self.connectionForBridge("br-int")
    threading.Timer(30.0, self._request_stats).start()
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)

# 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
    self.logger.debug('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 >= 34500 ],
                       key=lambda flow: (flow.priority)):
           try:
              if (stat.match['ipv4_src'] ==
                      flows_state[flow_id]['ip_src']) and
                    (stat.match['tcp_src'] ==
                      flows_state[flow_id]['port_src']):
                 valid=1
           except:
              pass
           try:
              if (stat.match['ipv4_dst'] ==
                      flows_state[flow_id]['ip_src']) and
                    (stat.match['tcp_dst'] ==
                      flows_state[flow_id]['port_src']):
                 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.debug('Not Enforcement State')
        self._handle_NotEnforcement_N_State()
     else :
        self.logger.debug('WARNING! - Too many
```

```
                active flows: Enforcement State')
        self._handle_Enforcement_E_State()

    # DPI FUNCTIONS
    def startDPI(self, flow_id):
        os.system("ssh ubuntu@192.168.10.32 'sudo nohup
                %s -i eth1 -f \"ip host %s\" -v 2 -s 10
                -j %s%s.json  > foo.out 2> foo.err <
                /dev/null & '" % (self.dpiExePath,
                flows_state[flow_id]['ip_src'],
                self.dpiCapPath,
                flows_state[flow_id]['ip_src']))
        self.logger.debug("[CONTROLLER - TIMER (%s)]
                    DPI started!\n\n\n\n",
                    datetime.datetime.fromtimestamp(
                        time.time()).strftime(
                            '%Y-%m-%d %H:%M:%S'))

    def obtain_DPI_output(self, flow_id):
        result = None
        try:
            result = subprocess.check_output(['ssh',
                'ubuntu@192.168.10.32',
                'sudo cat %s%s.json' %
                self.dpiCapPath,
                flows_state[flow_id]['ip_src'] ])
        except subprocess.CalledProcessError as e:
            result = e.output
        return result

    def _get_ports_info(self, dpid):
            #Return information about all port on a switch
        return self.dpset.get_ports(dpid)

    def _get_port_name(self, dpid, port):
            #Return the name associated to the specified
            # port number on the specified switch
        return self.dpset.get_port(dpid, port).name

    def connectionForBridge(self, bridge):
        if bridge in self.connections_name_dpid:
            return self.connections_name_dpid[bridge]
        else:
            return -1
```

```
#function executed after DPI analysis
def dpi_analysis_finished(self, flow_id):

   ## Step 1: stop DPI
   #self.stopDPI()
   #time.sleep(0.15)

   self.logger.debug("[DEBUG - READING JSON FILE] ")
   # Step 2: read json output file (DPI classification)
   json_data = self.obtain_DPI_output(flow_id)
   self.logger.debug("[DEBUG - JSON] %s", str(json_data))
   data = json.loads(json_data)
   list_of_flows = data["known.flows"]

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

         # UPDATING FLOW INFORMATIONS
         if host_a == self.ip_bususer or
              host_a == self.ip_resuser or
              host_b == self.ip_bususer or
              host_b == self.ip_resuser :
            self._memFlow(flow_id, ip_src=host_a,
                 port_src=port_a, ip_dst=host_b,
                 port_dst=port_b)
            time.sleep(0.10)
            self._handle_PreventiveEnforcement_E_State(
```

```
                            flow_id)

    def add_flow(self, datapath, priority, match, actions,
                 h_timeout=0, buffer_id=None, timeout=0):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(
                ofproto.OFPIT_APPLY_ACTIONS, actions)]

        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath,
                  idle_timeout=timeout, hard_timeout=h_timeout,
                  buffer_id=buffer_id, priority=priority,
                  match=match, instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath,
                  idle_timeout=timeout, hard_timeout=h_timeout,
                  priority=priority, match=match,
                  instructions=inst)

        datapath.send_msg(mod)

    # UTILITY
    def _memFlow(self, flow_id, ip_src="0.0.0.0", port_src=0,
                 ip_dst="0.0.0.0", port_dst=0, in_port=0,
                 ip_proto=0, state="X", rules=[]) :

        global flows_state
        global active_flows

        if flow_id in active_flows :
            if ip_src != "0.0.0.0" :
                flows_state[flow_id]['ip_src'] = ip_src
            if port_src != 0 :
                flows_state[flow_id]['port_src'] = port_src
            if ip_dst != "0.0.0.0" :
                flows_state[flow_id]['ip_dst'] = ip_dst
            if port_dst != 0 :
                flows_state[flow_id]['port_dst'] = port_dst
            if in_port != 0 :
                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 != [] :
                flows_state[flow_id]['rules'] = rules
        else:     #NEW FLOW
            flow={}
            flow['flow_id'] = flow_id
            flow['ip_src'] = ip_src
            flow['port_src'] = port_src
            flow['ip_dst'] = ip_dst
            flow['port_dst'] = port_dst
            flow['in_port'] = in_port
            flow['ip_proto'] = ip_proto
            flow['state'] = state
            flow['rules'] = rules
            if ip_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)

        #UPDATE LOG
        self.logger.debug("**FLOW UPGRADED** : %s",
                          flows_state[flow_id])

    #Handle reception of StateChange message (NOTE.
    #the message is sent whenever a switch performs
    #the handshake with controller)
    @set_ev_cls(ofp_event.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def state_change_handler(self, ev):
        datapath = ev.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        if ev.state == MAIN_DISPATCHER:
            port_name = self._get_port_name(datapath.id,
                                            4294967294)
            if not datapath.id in self.switch_dpid_name:
                self.switch_dpid_name[datapath.id] = port_name
                self.logger.debug("[TIMER (%s)] [SC-HANDLER]
                      Switch %s registered on the controller,
                      datapath.id  =  %s",
                      datetime.datetime.fromtimestamp(
                          time.time()).strftime(
                               '%Y-%m-%d %H:%M:%S'),
```

```
                str(port_name), datapath.id)
            self.connections_name_dpid[
                str(port_name)] = datapath

            #Tell the eswitch to send msgs to the
            # Controller in case of table miss
            match = parser.OFPMatch()
            actions = [parser.OFPActionOutput(
                        ofproto.OFPP_CONTROLLER,
                        ofproto.OFPCML_NO_BUFFER)]
            self.add_flow(datapath, 0, match, actions)

        # ARP and ICMP
        self._handle_Initial_Init_State(datapath)

    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.switch_dpid_name:
            self.logger.debug("[TIMER (%s)] [SC-HANDLER]
                    Switch %s disconnected",
                    datetime.datetime.fromtimestamp(
                        time.time()).strftime(
                            '%Y-%m-%d %H:%M:%S'),
                    self.switch_dpid_name[datapath.id])
            del self.connections_name_dpid[
                    self.switch_dpid_name[datapath.id]]
            del self.switch_dpid_name[datapath.id]

#Handle PacketIn event
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):

    global flow_id
    self.logger.debug("[DEBUG] FLOW-ID %s", flow_id)

    msg = ev.msg
    pkt = packet.Packet(msg.data)

    header = pkt.get_protocol(ipv4.ipv4)
    self.logger.debug("[DEBUG] ip_proto=%s ip_src=%s
                ip_dst=%s in_port=%d", header.proto,
                header.src, header.dst, in_port)

    np = len(pkt.get_protocols(ethernet.ethernet))
    self.logger.debug("[PKT-HANDLER]  (%s) Number of
              datetime.datetime.fromtimestamp(
```

```
                time.time()).strftime(
                     '%Y-%m-%d %H:%M:%S'), np)
    if np == 1:

        self._handle_Classification_C_State(flow_id, msg)

        #ATTESA DI 15 SECONDI E FINE ANALISI
        t=Timer(15.0, self.dpi_analysis_finished(flow_id):)
        t.start()

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



# STATE MACHINE ----------------------------------------


# ----------------------INITIAL STATE--------------------

def _handle_Initial_Init_State(self, dp):
    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # ARP packets
    switch_port = []
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
    inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, action)]
    msg = parser.OFPFlowMod( instructions=inst,
         priority = 32000,
         match = parser.OFPMatch(eth_type = 2054) )
    dp.send(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['priority'] = msg.priority
    # Matching rule
    tmp_dict_match['eth_type'] = msg.match.eth_type
    # Actions
    switch_port.append('OFPP_NORMAL')
```

```
    tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
    tmp_dict_actions['port'] = switch_port[0]
    tmp_list_actions.append(tmp_dict_actions)
              # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # ICMP packets
    switch_port = []
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
    inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, action)]
    msg = parser.OFPFlowMod( instructions=inst,
         priority = 32000,
         match = parser.OFPMatch(eth_type = 2048,
                                 ip_proto=1) )
    dp.send(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['priority'] = msg.priority
    # Matching rule
    tmp_dict_match['eth_type'] = msg.match.eth_type
    tmp_dict_match['ip_proto'] = msg.match.ip_proto
    # Actions
    switch_port.append('OFPP_NORMAL')
    tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
    tmp_dict_actions['port'] = switch_port[0]
    tmp_list_actions.append(tmp_dict_actions)
              # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

# ------------------CLASSIFICATION STATE------------------

def _handle_Classification_C_State(self, flow_id,
```

```
                                        match_from_packet):
    global active_flows
    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    rules_list = []

    datapath = match_from_packet.datapath
    parser = datapath.ofproto_parser
    coming_port =  match_from_packet.match['in_port']

    pkt = packet.Packet(match_from_packet.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    eth_type = eth.ethertype

    if eth_type == ether.ETH_TYPE_IP :     # TCP or UDP CASE

       pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
       nw_src = pkt_ipv4.src
       nw_dst = pkt_ipv4.dst
       pkt_tcp = pkt.get_protocol(tcp.tcp)

       # 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')
       mac_addr = self.get_in_mac_address('DPI', 'outbound')
       dp=connectionForBridge(switch_port[0])
       ofproto = dp.ofproto
       parser = dp.ofproto_parser
       action.append(parser.OFPActionSetField(
                      eth_dst= mac_addr))
       action.append(parser.OFPActionOutput( switch_port[1]))
       inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, action)]
       msg = parser.OFPFlowMod( hard_timeout=270,
            priority=34500,
            match=parser.OFPMatch(
                 in_port = coming_port, eth_type = 2048,
                 ip_proto=pkt_ipv4.proto, ipv4_src = nw_src,
                 tcp_src = pkt_tcp.src_port),
            instructions=inst )
```

```
dp.send_msg(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear()  # Empty dict_actions
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()  # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int internal network, inbound traffic,
# from DPI to User
switch_port = []
switch_port = self.get_in_port('DPI', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
```

```
        priority = 34500,
        match = parser.OFPMatch(
            in_port = switch_port[1], eth_type = 2048,
            ip_proto=pkt_ipv4.proto, ipv4_dst = nw_src,
            tcp_dst = pkt_tcp.src_port),
        instructions=inst )
    dp.send(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['hard_timeout'] = msg.hard_timeout
    tmp_dict_opts['priority'] = msg.priority
    # 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_dst'] = msg.match.ipv4_dst
    tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
    # Actions
    switch_port.append('OFPP_NORMAL')
    tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
    tmp_dict_actions['port'] = switch_port[2]
    tmp_list_actions.append(tmp_dict_actions)
            # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions

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

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # br-int Gateway network, inbound traffic,
    # from br3 to DPI, MAC_DST is changed
    switch_port = []
    action=[]
    switch_port = self.get_in_port('DPI', 'inbound')
    mac_addr = self.get_in_mac_address('DPI', 'inbound')
    dp=connectionForBridge(switch_port[0])
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
```

```
action.append(parser.OFPActionSetField(
                eth_dst= mac_addr ) )
action.append(parser.OFPActionOutput(switch_port[1]))
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority=34500, match=parser.OFPMatch(
          in_port = self.outport, eth_type = 2048,
          ip_proto=pkt_ipv4.proto, ipv4_dst = nw_src,
          tcp_dst = pkt_tcp.src_port ),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

```
# br-int Gateway network, outbound traffic,
# from DPI to br3
switch_port = []
switch_port = self.get_in_port('DPI', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority = 34500, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=pkt_ipv4.proto, ipv4_src = nw_src,
          tcp_src = pkt_tcp.src_port),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

```
# br3, outbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority = 34500, match = parser.OFPMatch(
          eth_type = 2048, ip_proto=pkt_ipv4.proto,
          ipv4_src = flows_state[flow_id]['ip_nat'],
          tcp_src = pkt_tcp.src_port),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

```
# br3, inbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority = 34500, match = parser.OFPMatch(
          eth_type = 2048, ip_proto=pkt_ipv4.proto,
          ipv4_dst = flows_state[flow_id]['ip_nat'],
          tcp_dst = pkt_tcp.src_port),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, outbound traffic
```

```
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority = 34500, match = parser.OFPMatch(
          eth_type = 2048, ip_proto=pkt_ipv4.proto,
          ipv4_src = flows_state[flow_id]['ip_nat'],
          tcp_src = pkt_tcp.src_port),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
```

```
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( hard_timeout=270,
     priority = 34500, match = parser.OFPMatch(
          eth_type = 2048, ip_proto=pkt_ipv4.proto,
          ipv4_dst = flows_state[flow_id]['ip_nat'],
          tcp_dst = pkt_tcp.src_port),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['hard_timeout'] = msg.hard_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# START DPI
self.startDPI(flow_id)
```

```
        else :      # D CASE

            self._handle_NotCompliant_D_State(flow_id, datapath)

        # SAVING CURRENT FLOW IN flows_state
        self._memFlow(flow_id, state = 'C', rules = rules_list)

    # ----------------------NOT COMPLIANT STATE--------------

    def _handle_NotCompliant_D_State(self, flow_id, datapath):
        tmp_list_opts = []
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}

        rules_list = []

        global flows_state

        # packet dropping rules
        switch_port = []
        action=[]
        ofproto = dp.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(
                ofproto.OFPIT_APPLY_ACTIONS, action)]
        msg = parser.OFPFlowMod(priority=30000,
             match=parser.OFPMatch(
                  in_port = flows_state[flow_id]['in_port'],
                  eth_type = eth.ethertype,
                  ip_proto=flows_state[flow_id]['ip_proto'],
                  ipv4_src = flows_state[flow_id]['ip_src'],
                  tcp_src = flows_state[flow_id]['port_src'] ),
             instructions=inst)
        datapath.send_msg(msg)
        # Add the previous rule to internal memory
        # Options
        tmp_dict_opts['priority'] = msg.priority
        # 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['tcp_src'] = msg.match.tcp_src
        # Actions
```

```
        switch_port.append('OFPP_F')
        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() # Empty dict_actions

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

        tmp_list_opts = []
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}

        # SAVING CURRENT FLOW IN flows_state
        self._memFlow(flow_id, state = 'D', rules = rules_list)


    # ----------------PREVENTIVE ENFORCEMENT STATE------------

    def _handle_PreventiveEnforcement_E_State(self, f_id):
        tmp_list_opts = []
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}

        rules_list = []

        global flows_state
        global classified_flows
        global hipriousers
        global wanaproto


        if flows_state[f_id]['ip_src'] in hipriousers :
                                        #HIGH PRIORITY CASE

            # br-int internal network, outbound traffic,
            # from HiPrioUser to Wana (LAN port),
            # MAC_DST is changed
            switch_port = []
```

```
action=[]
switch_port = self.get_in_port('Wana', 'outbound')
mac_addr = self.get_in_mac_address('Wana', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
               eth_dst= mac_addr))
action.append( parser.OFPActionOutput(
               switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(idle_timeout=60,
     priority=34501, match=parser.OFPMatch(
          in_port = flows_state[f_id]['in_port'],
          eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_src = flows_state[f_id]['ip_src'],
          tcp_src = flows_state[f_id]['port_src'] ),
     instructions=inst )
dp.send_msg(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

rule = {'switch': switch_port[0].id, 'op': 'ADD',
          'options': tmp_dict_opts,
```

```
            'match': tmp_dict_match,
            'actions': tmp_list_actions}
rules_list.append(rule)

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int internal network, inbound traffic,
# from Wana (LAN port) to HiPrioUser
switch_port = []
switch_port = self.get_in_port('Wana', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority = 34501, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_dst = flows_state[f_id]['ip_src'],
          tcp_dst = flows_state[f_id]['port_src']),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
```

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

for protocol in wanaproto:

   # br-int Gateway network, inbound traffic,
   # from br3 to Wana (WAN port), MAC_DST is changed
   switch_port = []
   action=[]
   switch_port = self.get_in_port('Wana', 'inbound')
   mac_addr = self.get_in_mac_address('Wana',
                                      'inbound')
   dp=connectionForBridge(switch_port[0])
   ofproto = dp.ofproto
   parser = dp.ofproto_parser
   action.append( parser.OFPActionSetField(
                  eth_dst= mac_addr ) )
   action.append( parser.OFPActionOutput(
                  switch_port[1] ) )
   inst = [parser.OFPInstructionActions(
           ofproto.OFPIT_APPLY_ACTIONS, action)]
   msg = parser.OFPFlowMod( idle_timeout=60,
        priority=34501, match=parser.OFPMatch(
           in_port = self.outport, eth_type = 2048,
           ip_proto=protocol,
           ipv4_dst = flows_state[f_id]['ip_src'],
           tcp_dst = flows_state[f_id]['port_src']),
        instructions=inst )
   dp.send(msg)
   # Add the previous rule to internal memory
   # Options
   tmp_dict_opts['idle_timeout'] = msg.idle_timeout
   tmp_dict_opts['priority'] = msg.priority
   # 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, outbound traffic,
# from Wana (WAN port) to br3
switch_port = []
switch_port = self.get_in_port('Wana', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority = 34501, match = parser.OFPMatch(
        in_port = switch_port[1], eth_type = 2048,
        ip_proto=protocol,
        ipv4_src = flows_state[f_id]['ip_src'],
        tcp_src = flows_state[f_id]['port_src']),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
```

```
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, outbound traffic, to WanaDec
switch_port = []
action=[]
switch_port = self.get_in_port('WanaDec',
                               'outbound')
mac_addr = self.get_in_mac_address('WanaDec',
                               'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
               eth_dst= mac_addr ) )
action.append( parser.OFPActionOutput(
               switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority=34501, match=parser.OFPMatch(
```

```
            eth_type = 2048, ip_proto=protocol,
            ipv4_src = flows_state[f_id]['ip_nat'],
            tcp_src = flows_state[f_id]['port_src']),
        instructions=inst )
    dp.send_msg(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['idle_timeout'] = msg.idle_timeout
    tmp_dict_opts['priority'] = msg.priority
    # Matching rule
    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['tcp_src'] = msg.match.tcp_src
    # Actions
    tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
    tmp_dict_actions['dl_addr'] = mac_addr
    tmp_list_actions.append(tmp_dict_actions)
              # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions
    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() # Empty dict_actions

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

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # br3, inbound traffic, from WanaDec
    switch_port = []
    switch_port = self.get_in_port('WanaDec',
                                   'outbound')
    dp=connectionForBridge(switch_port[0])
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
```

```
    inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, action)]
    msg = parser.OFPFlowMod(idle_timeout=60,
       priority = 34501, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=protocol,
          ipv4_dst = flows_state[f_id]['ip_nat'],
          tcp_dst = flows_state[f_id]['port_src']),
       instructions=inst )
    dp.send(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['idle_timeout'] = msg.idle_timeout
    tmp_dict_opts['priority'] = msg.priority
    # 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_dst'] = msg.match.ipv4_dst
    tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
    # Actions
    switch_port.append('OFPP_NORMAL')
    tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
    tmp_dict_actions['port'] = switch_port[2]
    tmp_list_actions.append(tmp_dict_actions)
              # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions

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

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

# br4, outbound traffic, from WanaDec
switch_port = []
switch_port = self.get_in_port('WanaDec', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
```

```
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(idle_timeout=270,
     priority = 34501, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_src = flows_state[f_id]['ip_nat'],
          tcp_src = flows_state[f_id]['port_src']),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
action=[]
switch_port = self.get_in_port('WanaDec', 'inbound')
mac_addr = self.get_in_mac_address('WanaDec',
```

```
                                        'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
               eth_dst= mac_addr ) )
action.append( parser.OFPActionOutput(
               switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority=34501, match=parser.OFPMatch(
          eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_dst = flows_state[f_id]['ip_nat'],
          tcp_dst = flows_state[f_id]['port_src'] ),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

```
            tmp_list_opts = []
            tmp_list_actions = []
            tmp_dict_match = {}
            tmp_dict_actions = {}

        else :      #LOW PRIORITY CASE

            # TCP : br-int internal network, outbound traffic,
            # from LowPrioUser to TC (1st port),
            # MAC_DST is changed
            switch_port = []
            action=[]
            switch_port = self.get_in_port('TC', 'outbound')
            mac_addr = self.get_in_mac_address('TC', 'outbound')
            dp=connectionForBridge(switch_port[0])
            ofproto = dp.ofproto
            parser = dp.ofproto_parser
            action.append( parser.OFPActionSetField(
                           eth_dst= mac_addr ) )
            action.append( parser.OFPActionOutput(
                           switch_port[1] ) )
            inst = [parser.OFPInstructionActions(
                    ofproto.OFPIT_APPLY_ACTIONS, action)]
            msg = parser.OFPFlowMod(idle_timeout=60,
                 priority=34501, match=parser.OFPMatch(
                      in_port = flows_state[f_id]['in_port'],
                      eth_type = 2048,
                      ip_proto=flows_state[f_id]['ip_proto'],
                      ipv4_src = flows_state[f_id]['ip_src'],
                      tcp_src = flows_state[f_id]['port_src'] ),
                 instructions=inst )
            dp.send_msg(msg)
            # Add the previous rule to internal memory
            # Options
            tmp_dict_opts['idle_timeout'] = msg.idle_timeout
            tmp_dict_opts['priority'] = msg.priority
            # 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['tcp_src'] = msg.match.tcp_src
            # Actions
            tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
            tmp_dict_actions['dl_addr'] = mac_addr
```

```
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions
rule = {'switch': switch_port[0].id, 'op': 'ADD',
          'options': tmp_dict_opts,
          'match': tmp_dict_match,
          'actions': tmp_list_actions}
rules_list.append(rule)
tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# TCP : br-int internal network, inbound traffic,
# from TC (1st port) to LowPrioUser
switch_port = []
switch_port = self.get_in_port('TC', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority = 34501, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_dst = flows_state[f_id]['ip_src'],
          tcp_dst = flows_state[f_id]['port_src']),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
```

```
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# TCP : br-int Gateway network, inbound traffic,
# from br3 to TC (2nd port), MAC_DST is changed
switch_port = []
action=[]
switch_port = self.get_in_port('Wana', 'inbound')
mac_addr = self.get_in_mac_address('Wana', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
               eth_dst= mac_addr ) )
action.append( parser.OFPActionOutput(
               switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority=34501, match=parser.OFPMatch(
          in_port = self.outport, eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
          ipv4_dst = flows_state[f_id]['ip_src'],
          tcp_dst = flows_state[f_id]['port_src'] ),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
```

```
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
          # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, outbound traffic,
# from TC (2nd port) to br3
switch_port = []
switch_port = self.get_in_port('TC', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority = 34501, match = parser.OFPMatch(
          in_port = switch_port[1], eth_type = 2048,
          ip_proto=flows_state[f_id]['ip_proto'],
```

```
            ipv4_src = flows_state[f_id]['ip_src'],
            tcp_src = flows_state[f_id]['port_src']),
     instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
            # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# TCP : br3, outbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
     priority = 34501, match = parser.OFPMatch(
            eth_type = 2048,
```

```
            ip_proto=flows_state[f_id]['ip_proto'],
            ipv4_src = flows_state[f_id]['ip_nat'],
            tcp_src = flows_state[f_id]['port_src']),
      instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# TCP : br3, inbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
          ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
      priority = 34501, match = parser.OFPMatch(
            eth_type = 2048,
```

```
                ip_proto=flows_state[f_id]['ip_proto'],
                ipv4_dst = flows_state[f_id]['ip_nat'],
                tcp_dst = flows_state[f_id]['port_src']),
           instructions=inst )
      dp.send(msg)
      # Add the previous rule to internal memory
      # Options
      tmp_dict_opts['idle_timeout'] = msg.idle_timeout
      tmp_dict_opts['priority'] = msg.priority
      # Matching rule
      tmp_dict_match['eth_type'] = msg.match.eth_type
      tmp_dict_match['ip_proto'] = msg.match.ip_proto
      tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
      tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
      # Actions
      switch_port.append('OFPP_NORMAL')
      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() # Empty dict_actions

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

      tmp_list_opts = []
      tmp_list_actions = []
      tmp_dict_match = {}
      tmp_dict_actions = {}

      # TCP : br4, outbound traffic
      switch_port = []
      switch_port.append('br4')
      dp=connectionForBridge(switch_port[0])
      ofproto = dp.ofproto
      parser = dp.ofproto_parser
      action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
      inst = [parser.OFPInstructionActions(
              ofproto.OFPIT_APPLY_ACTIONS, action)]
      msg = parser.OFPFlowMod( idle_timeout=60,
           priority = 34501, match = parser.OFPMatch(
                eth_type = 2048,
```

```
            ip_proto=flows_state[f_id]['ip_proto'],
            ipv4_src = flows_state[f_id]['ip_nat'],
            tcp_src = flows_state[f_id]['port_src']),
      instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# TCP : br4, inbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
          ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
      priority = 34501, match = parser.OFPMatch(
            eth_type = 2048,
```

```
                ip_proto=flows_state[f_id]['ip_proto'],
                ipv4_dst = flows_state[f_id]['ip_nat'],
                tcp_dst = flows_state[f_id]['port_src']),
            instructions=inst )
        dp.send(msg)
        # Add the previous rule to internal memory
        # Options
        tmp_dict_opts['idle_timeout'] = msg.idle_timeout
        tmp_dict_opts['priority'] = msg.priority
        # Matching rule
        tmp_dict_match['eth_type'] = msg.match.eth_type
        tmp_dict_match['ip_proto'] = msg.match.ip_proto
        tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
        tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
        # Actions
        switch_port.append('OFPP_NORMAL')
        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() # Empty dict_actions

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

        tmp_list_opts = []
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}

    # SAVING CURRENT FLOW IN flows_state
    self._memFlow(f_id, state = 'E', rules = rules_list)

    # ADD THE FLOW TO THE CLASSIFIED LIST
    classified_flows.append(f_id)

    # FLOWS CONTROL START WHEN FIRST FLOW
    if f_id==0 :
        self._request_stats()

    # UPDATE FLOW-ID
    time.sleep(0.10)
```

```
      global flow_id
      flow_id=f_id+1

   # ----------------------ENFORCEMENT STATE------------------

   def _handle_Enforcement_E_State(self):
      tmp_list_opts = []
      tmp_list_actions = []
      tmp_dict_match = {}
      tmp_dict_actions = {}

      rules_list = []

      global flows_state
      global classified_flows
      global hipriousers
      global wanaproto

      for flow in classified_flows :
         if flows_state[flow]['state'] != 'E' :
            if flows_state[flow]['ip_src'] in hipriousers :
                                    #HIGH PRIORITY CASE

               # br-int internal network, outbound traffic,
               # from HiPrioUser to Wana (LAN port),
               # MAC_DST is changed
               switch_port = []
               action=[]
               switch_port = self.get_in_port('Wana',
                                       'outbound')
               mac_addr = self.get_in_mac_address('Wana',
                                       'outbound')
               dp=connectionForBridge(switch_port[0])
               ofproto = dp.ofproto
               parser = dp.ofproto_parser
               action.append( parser.OFPActionSetField(
                                    eth_dst= mac_addr ) )
               action.append( parser.OFPActionOutput(
                                    switch_port[1] ) )
               inst = [parser.OFPInstructionActions(
                        ofproto.OFPIT_APPLY_ACTIONS, action)]
               msg = parser.OFPFlowMod(idle_timeout=60,
                  priority=34501, match=parser.OFPMatch(
                     in_port = flows_state[flow]['in_port'],
                     eth_type = 2048,
```

```
            ip_proto=flows_state[flow]['ip_proto'],
            ipv4_src = flows_state[flow]['ip_src'],
            tcp_src = flows_state[flow]['port_src']),
        instructions=inst )
    dp.send_msg(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['idle_timeout'] = msg.idle_timeout
    tmp_dict_opts['priority'] = msg.priority
    # 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['tcp_src'] = msg.match.tcp_src
    # Actions
    tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
    tmp_dict_actions['dl_addr'] = mac_addr
    tmp_list_actions.append(tmp_dict_actions)
            # Add actions to the list of actions
    tmp_dict_actions.clear() # Empty dict_actions
    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() # Empty dict_actions

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

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # br-int internal network, inbound traffic,
    # from Wana (LAN port) to HiPrioUser
    switch_port = []
    switch_port = self.get_in_port('Wana',
                                   'outbound')
    dp=connectionForBridge(switch_port[0])
```

```
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501,
   match = parser.OFPMatch(
      in_port = switch_port[1],
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

```
for protocol in wanaproto:

   # br-int Gateway network, inbound traffic,
#from br3 to Wana (WAN port), MAC_DST is changed
   switch_port = []
   action=[]
   switch_port = self.get_in_port('Wana',
                                  'inbound')
   mac_addr = self.get_in_mac_address('Wana',
                                  'inbound')
   dp=connectionForBridge(switch_port[0])
   ofproto = dp.ofproto
   parser = dp.ofproto_parser
   action.append( parser.OFPActionSetField(
                        eth_dst= mac_addr))
   action.append( parser.OFPActionOutput(
                        switch_port[1]))
   inst = [parser.OFPInstructionActions(
           ofproto.OFPIT_APPLY_ACTIONS, action)]
   msg = parser.OFPFlowMod( idle_timeout=60,
      priority=34501, match=parser.OFPMatch(
         in_port = self.outport, eth_type =2048,
         ip_proto=protocol,
         ipv4_dst = flows_state[flow]['ip_src'],
         tcp_dst=flows_state[flow]['port_src']),
      instructions=inst )
   dp.send(msg)
   # Add the previous rule to internal memory
   # Options
   tmp_dict_opts['idle_timeout']=
            msg.idle_timeout
   tmp_dict_opts['priority'] = msg.priority
   # 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_dst']=msg.match.ipv4_dst
   tmp_dict_match['tcp_dst']=msg.match.tcp_dst
   # Actions
   tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
   tmp_dict_actions['dl_addr'] = mac_addr
   tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
   tmp_dict_actions.clear() # Empty dict_actions
   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() # Empty dict_actions

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

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

    # br-int Gateway network, outbound traffic,
#from Wana (WAN port) to br3
    switch_port = []
    switch_port = self.get_in_port('Wana',
                                   'inbound')
    dp=connectionForBridge(switch_port[0])
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    action=parser.OFPActionOutput(
                 ofp.OFPP_NORMAL)
    inst = [parser.OFPInstructionActions(
            ofproto.OFPIT_APPLY_ACTIONS, action)]
    msg = parser.OFPFlowMod( idle_timeout=60,
       priority = 34501, match = parser.OFPMatch(
         in_port = switch_port[1],eth_type =2048,
         ip_proto=protocol,
         ipv4_src = flows_state[flow]['ip_src'],
         tcp_src =flows_state[flow]['port_src']),
       instructions=inst )
    dp.send(msg)
    # Add the previous rule to internal memory
    # Options
    tmp_dict_opts['idle_timeout'] =
             msg.idle_timeout
    tmp_dict_opts['priority'] = msg.priority
    # 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
    # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, outbound traffic, to WanaDec
switch_port = []
action=[]
switch_port = self.get_in_port('WanaDec',
                        'outbound')
mac_addr = self.get_in_mac_address('WanaDec',
                        'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
                    eth_dst= mac_addr))
action.append( parser.OFPActionOutput(
                    switch_port[1]))
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority=34501, match=parser.OFPMatch(
     eth_type =2048, ip_proto=protocol,
```

```
     ipv4_src = flows_state[flow]['ip_nat'],
     tcp_src=flows_state[flow]['port_src']),
   instructions=inst )
dp.send_msg(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] =
                          msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
    # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, inbound traffic, from WanaDec
switch_port = []
switch_port = self.get_in_port('WanaDec',
                  'outbound')
```

```
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(
               ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
     in_port = switch_port[1],eth_type =2048,
     ip_proto=protocol,
     ipv4_dst = flows_state[flow]['ip_nat'],
     tcp_dst=flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] =
         msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst']=msg.match.ipv4_dst
tmp_dict_match['tcp_dst']=msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
    # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}
```

```
# br4, outbound traffic, from WanaDec
switch_port = []
switch_port = self.get_in_port('WanaDec',
                  'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(idle_timeout=270,
   priority = 34501, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
```

```
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
action=[]
switch_port = self.get_in_port('WanaDec',
                               'inbound')
mac_addr = self.get_in_mac_address('WanaDec',
                               'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
                     eth_dst= mac_addr ) )
action.append( parser.OFPActionOutput(
                     switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority=34501, match=parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src'] ),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

            tmp_list_opts = []
            tmp_list_actions = []
            tmp_dict_match = {}
            tmp_dict_actions = {}

        else :    #LOW PRIORITY CASE

            # br-int internal network, outbound traffic,
            #from LowPrioUser to TC (1st port),
            # MAC_DST is changed
            switch_port = []
            action=[]
            switch_port = self.get_in_port('TC', 'outbound')
            mac_addr = self.get_in_mac_address('TC',
                                       'outbound')
            dp=connectionForBridge(switch_port[0])
            ofproto = dp.ofproto
            parser = dp.ofproto_parser
            action.append( parser.OFPActionSetField(
                                 eth_dst= mac_addr ) )
            action.append( parser.OFPActionOutput(
                                       switch_port[1] ) )
            inst = [parser.OFPInstructionActions(
                    ofproto.OFPIT_APPLY_ACTIONS, action)]
            msg = parser.OFPFlowMod(idle_timeout=60,
               priority=34501, match=parser.OFPMatch(
                  in_port = flows_state[flow]['in_port'],
                  eth_type = 2048,
                  ip_proto=flows_state[flow]['ip_proto'],
                  ipv4_src = flows_state[flow]['ip_src'],
                  tcp_src = flows_state[flow]['port_src'] ),
               instructions=inst )
            dp.send_msg(msg)
            # Add the previous rule to internal memory
```

```
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions
rule = {'switch': switch_port[0].id,
       'op': 'ADD',
       'options': tmp_dict_opts,
       'match': tmp_dict_match,
       'actions': tmp_list_actions}
rules_list.append(rule)
tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int internal network, inbound traffic,
#from TC (1st port) to LowPrioUser
switch_port = []
switch_port = self.get_in_port('TC', 'outbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
```

```
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, inbound traffic,
#from br3 to TC (2nd port), MAC_DST is changed
switch_port = []
action=[]
switch_port = self.get_in_port('TC', 'inbound')
mac_addr = self.get_in_mac_address('TC',
                                   'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action.append( parser.OFPActionSetField(
```

```
                        eth_dst= mac_addr ) )
action.append( parser.OFPActionOutput(
                        switch_port[1] ) )
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority=34501, match=parser.OFPMatch(
      in_port = self.outport, eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src'] ),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
tmp_dict_actions['dl_addr'] = mac_addr
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
```

```
tmp_dict_actions = {}

# br-int Gateway network, outbound traffic,
#from TC (2nd port) to br3
switch_port = []
switch_port = self.get_in_port('TC', 'inbound')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_src'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
tmp_dict_actions['port'] = switch_port[2]
tmp_list_actions.append(tmp_dict_actions)
       # Add actions to the list of actions
tmp_dict_actions.clear() # Empty dict_actions

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

```
tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, outbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

```
tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, inbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

```
rules_list.append(rule)

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, outbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

rule = {'switch': switch_port[0].id,
       'op': 'ADD',
       'options': tmp_dict_opts,
       'match': tmp_dict_match,
```

```
       'actions': tmp_list_actions}
rules_list.append(rule)

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34501, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

rule = {'switch': switch_port[0].id,
       'op': 'ADD',
       'options': tmp_dict_opts,
```

```
            'match': tmp_dict_match,
            'actions': tmp_list_actions}
    rules_list.append(rule)

    tmp_list_opts = []
    tmp_list_actions = []
    tmp_dict_match = {}
    tmp_dict_actions = {}

# DELETE NOT ENFORCEMENT STATE RULES-----

# br-int internal network, outbound traffic,
   #from User to gw
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod(idle_timeout=60,
  priority=34502, match=parser.OFPMatch(
      in_port = flows_state[flow]['in_port'],
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_src'],
      tcp_src = flows_state[flow]['port_src'] ),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send_msg(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int internal network, inbound traffic,
   #from gw to User
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
```

```
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, inbound traffic,
   #from br3 to gw
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority=34502, match=parser.OFPMatch(
      in_port = self.outport, eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src'] ),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, outbound traffic,
   #from gw to br3
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_src'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, outbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
```

```
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, inbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
```

```
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, outbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE )
dp.send(msg)
# Add the previous rule to internal memory
```

```
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst,
   command=ofproto.OFPFC_DELETE )
dp.send(msg)
```

```
            # Add the previous rule to internal memory
            # Options
            tmp_dict_opts['idle_timeout'] = msg.idle_timeout
            tmp_dict_opts['priority'] = msg.priority
            # Matching rule
            tmp_dict_match['eth_type'] = msg.match.eth_type
            tmp_dict_match['ip_proto'] = msg.match.ip_proto
            tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
            tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
            # Actions
            switch_port.append('OFPP_NORMAL')
            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() # Empty dict_actions

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

            tmp_list_opts = []
            tmp_list_actions = []
            tmp_dict_match = {}
            tmp_dict_actions = {}

            # SAVING CURRENT FLOW IN flows_state
            self._memFlow(flow, state = 'E',
                          rules = rules_list)


    # ----------------------NOT ENFORCEMENT STATE---------------

    def _handle_NotEnforcement_N_State(self):
        tmp_list_opts = []
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}

        rules_list = []

        global flows_state
        global classified_flows
```

```
for flow in classified_flows :
   if flows_state[flow]['state'] != 'N' :

      # br-int internal network, outbound traffic,
         #from User to gw
      switch_port = []
      switch_port.append('br-int')
      dp=connectionForBridge(switch_port[0])
      ofproto = dp.ofproto
      parser = dp.ofproto_parser
      action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
      inst = [parser.OFPInstructionActions(
              ofproto.OFPIT_APPLY_ACTIONS, action)]
      msg = parser.OFPFlowMod(idle_timeout=60,
         priority=34502, match=parser.OFPMatch(
            in_port = flows_state[flow]['in_port'],
           eth_type = 2048,
            ip_proto=flows_state[flow]['ip_proto'],
            ipv4_src = flows_state[flow]['ip_src'],
            tcp_src = flows_state[flow]['port_src'] ),
         instructions=inst  )
      dp.send_msg(msg)
      # Add the previous rule to internal memory
      # Options
      tmp_dict_opts['idle_timeout'] = msg.idle_timeout
      tmp_dict_opts['priority'] = msg.priority
      # 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['tcp_src'] = msg.match.tcp_src
      # Actions
      switch_port.append('OFPP_NORMAL')
      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() # Empty dict_actions

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

```
rules_list.append(rule)

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int internal network, inbound traffic,
   #from gw to User
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

rule = {'switch': switch_port[0].id, 'op': 'ADD',
          'options': tmp_dict_opts,
```

```
          'match': tmp_dict_match,
          'actions': tmp_list_actions}
rules_list.append(rule)

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, inbound traffic,
   #from br3 to gw
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority=34502, match=parser.OFPMatch(
      in_port = self.outport, eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_src'],
      tcp_dst = flows_state[flow]['port_src'] ),
   instructions=inst  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions
```

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br-int Gateway network, outbound traffic,
   #from gw to br3
switch_port = []
switch_port.append('br-int')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      in_port = switch_port[1], eth_type =2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_src'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# 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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, outbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br3, inbound traffic
switch_port = []
switch_port.append('br3')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst  )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, outbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_src = flows_state[flow]['ip_nat'],
      tcp_src = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
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['tcp_src'] = msg.match.tcp_src
# Actions
switch_port.append('OFPP_NORMAL')
```

```
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# br4, inbound traffic
switch_port = []
switch_port.append('br4')
dp=connectionForBridge(switch_port[0])
ofproto = dp.ofproto
parser = dp.ofproto_parser
action=parser.OFPActionOutput(ofp.OFPP_NORMAL)
inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS, action)]
msg = parser.OFPFlowMod( idle_timeout=60,
   priority = 34502, match = parser.OFPMatch(
      eth_type = 2048,
      ip_proto=flows_state[flow]['ip_proto'],
      ipv4_dst = flows_state[flow]['ip_nat'],
      tcp_dst = flows_state[flow]['port_src']),
   instructions=inst )
dp.send(msg)
# Add the previous rule to internal memory
# Options
tmp_dict_opts['idle_timeout'] = msg.idle_timeout
tmp_dict_opts['priority'] = msg.priority
# Matching rule
tmp_dict_match['eth_type'] = msg.match.eth_type
tmp_dict_match['ip_proto'] = msg.match.ip_proto
tmp_dict_match['ipv4_dst'] = msg.match.ipv4_dst
tmp_dict_match['tcp_dst'] = msg.match.tcp_dst
# Actions
```

```
switch_port.append('OFPP_NORMAL')
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() # Empty dict_actions

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

tmp_list_opts = []
tmp_list_actions = []
tmp_dict_match = {}
tmp_dict_actions = {}

# SAVING CURRENT FLOW IN flows_state
self._memFlow(flow, state = 'N',
               rules = rules_list)
```

Bibliografia

[1] Wikipedia, Cloud Computing, disponibile:

`https://it.wikipedia.org/wiki/Cloud\_computing`

[2] Telecom Italia, Notiziario tecnico: SDN e NFV: quali sinergie? , disponibile:

`http://www.telecomitalia.com/tit/it/notiziariotecnico/numeri/2014-2/capitolo-05.html`

[3] Wikipedia, Software-defined networking, disponibile:

`https://en.wikipedia.org/wiki/Software-defined\_networking`

[4] Open Networking Foundation. OpenFlow Switch Specification. Technical report. Dicembre 2009. Disponibile:

`https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`

[5] Open Networking Foundation. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Technical report, ONF White Paper, Ottobre 2012. Disponibile:

`http://portal.etsi.org/NFV/NFV\_White\_Paper.pdf`

[6] F. Callegati, W. Cerroni, C. Contoli, G. Santandrea. Performance of Network Virtualization in Cloud Computing Infrastructures: The OpenStack Case.

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

[8] Wikipedia, OpenStack, disponibile:

`https://it.wikipedia.org/wiki/OpenStack`

[9] F. Foresta, Composizione dinamica di funzioni di rete virtuali in ambienti Cloud. 2015.

[10] Struttura del cluster di OpenStack, disponibile:

`https://keithtenzer.files.wordpress.com/2015/01/openstack\_multinode\_4\_architecture.jpg`

[11] Logo del controller Ryu, disponibile:

`http://osrg.github.io/ryu/css/images/LogoSet02.png`

[12] Architettura del controller Ryu, disponibile:

`http://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/}`

[13] Installazione di Ryu, disponibile:

`http://osrg.github.io/ryu/`

[14] Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.

[15] Ryu 3.24 documentation: OpenFlow v1.3 Messages and Structures, disponibile:

`http://ryu.readthedocs.org/en/latest/ofproto\_v1\_3\_ref.html`

[16] Gabriel Brown, White Paper: Service Chaining in Carrier Networks, disponibile:

`http://www.qosmos.com/wp-content/uploads/2015/02/Service-Chaining-in-Carrier-Networks\_WP\_Heavy-Reading\_Qosmos\_Feb2015.pdf`

[17] F. Callegati, W. Cerroni, C. Contoli, G. Santandrea, SDN Controller Design for Dynamic Chaining of Virtual Network Functions.

[18] Wikipedia, Deep packet inspection, disponibile:

`https://en.wikipedia.org/wiki/Deep\-packet\-inspection`

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

[20] C. Contoli.