Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Software Defined Networking - Technology and Application Scenarios

Studente/i

Davide Ganna

Relatore

Angelo Consoli

Correlatore

Cristian Bovino Ottaviano

Committente

Angelo Consoli

Corso di laurea

Ingegneria Elettronica

Modulo

M00002 - Progetto di Diploma

Anno

2019

Data

6 settembre 2019

STUDENTSUPSI

Indice

1 Introduzione **1**

2 Virtualizzazione Server **3**

2.1 Server Hardware 3

2.2 Basi di Virtualizzazione 4

2.2.1 Virtual Machine (VM) 5

2.2.2 VMware ESXi 6

2.3 Networking con Switches Virtuali su Host Virtualizzati 7

2.3.1 Virtual Switch 8

2.3.2 Cisco Nexus 1000v 9

2.3.2.1 Virtual Ethernet Module 9

2.3.2.2 Virtual Supervisor Module 10

2.4 Vantaggi della Virtualizzazione 11

2.4.1 Scalabilità 11

2.4.1.1 Scalabilità Verticale 11

2.4.1.2 Scalabilità Orizzontale 12

2.4.2 Quality of Service (QoS) 12

2.4.3 Riduzione del Costo di Hardware 13

STUDENTSUPSI

2.4.4 Tempi di Network Provisioning e di Distribuzione Inferiori 13
2.4.5 Recupero di Downtime Migliorato 13
2.4.6 Risparmio Sui Costi Statici 14
2.4.7 Produttività Più Elevata 14
2.5 Virtualizzazione - Laboratorio 14
2.5.1 Firewall 15
2.5.2 LAN 16
2.5.3 DMZ 16
2.6 Arista EOS e vEOS 17
2.7 vEOS - Laboratorio 19

3 SDN e Programmabilità di Rete 21

3.1 Data, Control e Management Planes 22
3.1.1 Data Plane 22
3.1.2 Control Plane 23
3.1.3 Management Plane 25
3.1.4 Data Plane di Uno Switch - Dettaglio 25
3.2 Controllori e Architettura di Rete 27
3.2.1 Controllori e Controllo Centralizzato 27
3.2.2 Southbound Interface 29
3.2.3 Northbound Interface 30
3.3 Sommario dell'Architettura SDN 32
3.4 Esempio di SDN - OpenFlow 32
3.4.1 Open SDN e OpenFlow 32

STUDENTSUPSI

3.4.1.1 OpenDayLight Controller 33

4 GNS3 **35**

4.1 Netmiko 38

4.2 NAPALM 41

4.2.1 Esempio di Utilizzo: Configurazione di ACLs 42

4.3 Ottimizzazione Del Codice 44

4.3.1 Criptare Le Passwords Dei Dispositivi 45

4.3.2 Multithreading 46

4.4 OpenDayLight e Mininet 50

5 MPLS **53**

5.1 MPLS VPN 56

5.1.1 Layer 3 VPN - VPRN 56

5.2 Realizzazione di VPN MPLS 57

5.2.1 Abilitare MPLS 58

5.2.2 Creare ed Assegnare VRFs 60

5.2.3 Configurare MP-BGP sui PE Routers 61

5.2.4 Configurare PE-CE OSPF 62

5.2.5 Configurare la Ridistribuzione dei Percorsi 63

5.2.6 Fase di Test e Conferma 64

5.3 Integrazione Virtuale - Fisica 66

6 VPN over MPLS con Controller SDN **71**

7 Conclusioni **77**

STUDENTSUPSI

STUDENTSUPSI

Capitolo 1

Introduzione

Questa tesi vuole introdurre il lettore ai concetti di *Rete Virtualizzata, Software Defined Networking* e *VPN over MPLS*.

Il documento è così suddiviso:

- Una prima parte dedicata al concetto di virtualizzazione server, i vantaggi che questa opzione offre e come utilizzarla per soddisfare i propri bisogni;
- La seconda parte, dedicata al concetto di Software Defined Networking, illustra i benefici che SDN porta ad una rete moderna.
- La terza ed ultima macrosezione vuole spiegare il concetto di MPLS, per poi analizzare la realizzazione di una rete privata basata su MPLS (VPN over MPLS).

L'ultimo capitolo di questa tesi racchiude il lavoro delle prime tre parti viste precedentemente, con un caso pratico relativo ad una esigenza reale. Verrà integrata una rete VPN over MPLS gestita da un controllore SDN, il tutto virtualizzato all'interno di un server.

Uno switch che lavora a livello 2 della pila ISO/OSI verrà abbreviato con la sigla *L2 Switch*. Lo stesso discorso si applica per gli switches di livello 3.

Il capitolo seguente vuole introdurre il lettore al concetto di *virtualizzazione server*, illustrando dapprima le basi, andando poi ad esaminare i vantaggi che questa tecnologia offre. Seguirà un caso pratico in cui realizzo una rete completamente virtualizzata e connessamediante switch virtualizzati.

STUDENTSUPSI

STUDENTSUPSI

Capitolo 2

Virtualizzazione Server

Sebbene multiple definzioni della parola "server" siano accettate, in questo documento ci riferiamo ad un server come un *luogo dove eseguire applicazioni*, in cui gli utenti della rete possono connettersi ed usare le suddette applicazioni in remoto.

Non è sbagliato pensare ad un server come un computer con tanta RAM, oppure ad uno con una CPU molto veloce. Un server può inoltre essere un componente software e non obbligatoriamente hardware. Avendo a disposizione i servers fisici di laboratorio, questa tesi si riferisce esplicitamente a quelli descritti nella sezione che segue.

2.1 Server Hardware

Come spiegato nella sezione precedente, gli utenti che necessitano di eseguire applicazioni si connettono ad un computer centrale. Questo viene considerato un luogo di accesso condiviso, ed è racchiuso in un luogo definito *data center*. Non avendo bisogno di connettersi fisicamente al server (ogni utente si connette da remoto), non vi è necessità di collegare nè mouse nè tastiere a questo computer centrale.

Dato che una ditta spesso necessita di più servers, vi è il bisogno di costruirli in modo tale che siano facilmente componibili e sovrapponibili. Per i seguenti motivi, i servers moderni sono dispositivi molto simili a quelli rappresentati nella figura 2.1:

STUDENTSUPSI

Figura 2.1: HP Gateway Servers: B-Series (Blade)

La figura mostra un blocco composto da 4 servers denominati *blades*. Anche se non visibili, ai lati vi sono dei buchi: questi sono stati concepiti per permettere ai servers di essere montati nei cosiddetti *racks*, ovvero scaffali dedicati.

2.2 Basi di Virtualizzazione

Tradizionalmente, l'idea di un server è quella di ospitare un sistema operativo (OS) al suo interno. L'hardware è composto da una CPU, memoria RAM, memoria allo stato solido (SSD) ed una o più schede di rete (NICs). Il sistema operativo può utilizzare tutte queste risorse hardware e gestire una o più applicazioni. La figura 2.2 illustra il concetto:

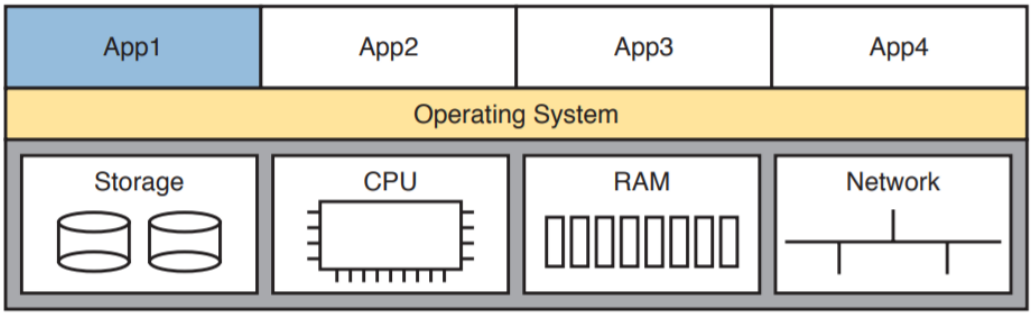


Figura 2.2: Modello Interno di un Server: Hardware Fisico, OS e Applicazioni

Fino a poco tempo fa, ogni server eseguiva un sistema operativo, e quell'OS utilizzava tutto l'hardware in quel server. La tendenza di oggi è invece quella di *virtualizzare* i servers.

STUDENTSUPSI

Quello che viene definito *virtualized data center* è il frutto della virtualizzazione dei server. Una ditta moderna compra servers hardware, li installa sui racks e poi tratta tutta la CPU, la RAM e la memoria come capacità del data center. In seguito, ogni istanza del sistema operativo viene disaccoppiata dall'hardware, ed è difatti virtuale (e non più fisica). Ogni pezzo di hardware che prima avremmo considerato come un server ora ospita più sistemi operativi. Ogni istanza virtuale di un OS è definita *Virtual Machine* (VM).

2.2.1 Virtual Machine (VM)

Una Virtual Machine, ovvero un'istanza di un OS che è disaccoppiata dal server fisico, deve essere comunque eseguita sull'hardware. Ogni VM ha una configurazione che specifica il numero minimo di vCPU (virtual CPU) che richiede, la RAM minima, e così via. Il sistema di virtualizzazione fa poi partire la VM nel server fisico in modo che sia garantita la capacità hardware necessaria per supportare tutte le VMs su quel determinato host. Quindi, in qualsiasi momento ogni VM è attiva su un server fisico usando una frazione di CPU, di RAM, di memoria e di interfacce di rete su quel server. La figura 2.3 illustra graficamente il concetto, con quattro VMs separate attive su un server fisico.

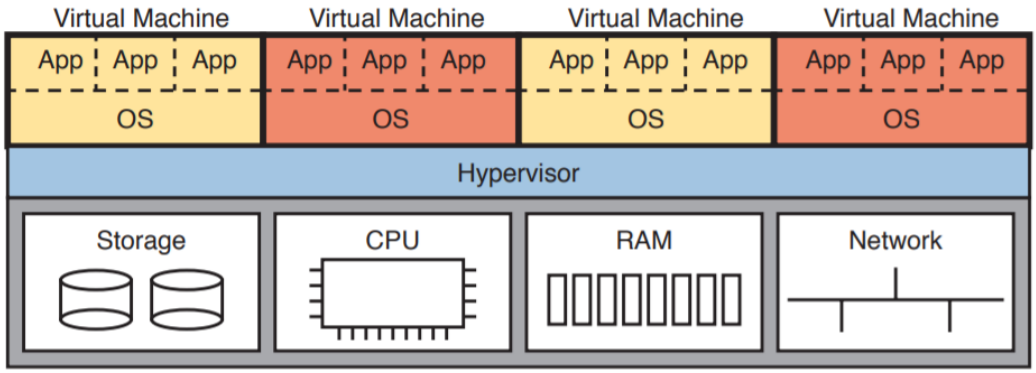


Figura 2.3: Quattro VMs Attive su Un Host: L'Hypervisor Gestisce le Risorse Hardware

Quello che ne risulta è che un singolo host fisico (server) spesso possiede più potenza di processamento di quella richiesta da un sistema operativo. Pensando ai processori, le CPU dei server moderni sono *multi-cores*, ovvero possiedono più processori in un singolo chip. Ogni core è capace di eseguire più processi contemporaneamente, caratteristica definita *multithreading*. Per esempio, un processore Intel con 8 cores multithread (di norma 2 threads/core) è capace di eseguire 16 differenti programmi *simultaneamente*. L'hypervisor (che verrà introdotto a breve) può poi trattare ogni thread disponibile come una CPU virtuale (vCPU) e dare ad ogni VM un numero *n* di vCPUs (16 nell'esempio precedente).

Per far sì che la virtualizzazione funzioni, ogni server fisico (chiamato *host* nel mondo di

STUDENTSUPSI

virtualizzazione server) utilizza un *hypervisor*. L'hypervisor gestisce ed alloca l'hardware dell'host (CPU, RAM, ecc.) per ogni VM basandosi sui settaggi di quella Virtual Machine. Ogni VM è attiva come se fosse contenuta in una frazione del server fisico, con un numero specificato di CPUs, RAM, memoria e NICs. Supponiamo di avere una VM che necessita di 4 CPUs e 8 GB di RAM: l'hypervisor alloca le parti specifiche dell'hardware necessario di modo tale che una frazione di queste agisca come un componente indipendente, ovvero la nostra Virtual Machine.

Esistono tante ditte che forniscono sistemi di virtualizzazione come quello appena elencato. Per effettuare i test in laboratorio ho utilizzato *VMware ESXi*. La sezione seguente si focalizza sulle caratteristiche offerte da questo prodotto.

2.2.2 VMware ESXi

VMware ESXi (acronimo di Elastic Sky X integrated) è un hypervisor di tipo-1 sviluppato da VMware che permette la virtualizzazione di servers. Essendo di tipo-1, ESXi non è un'applicazione che viene installata su un sistema operativo, bensì include ed integra componenti OS vitali (come il kernel di sistema).

Il kernel di ESXi è un *micro-kernel*, ovvero la minima quantità di software possibile per fornire alll'OS servizi di indirizzamento, gestione dei thread e comunicazione fra processi (IPC). Il micro-kernel ESXi possiede tre interfacce:

- Interfaccia hardware
- Interfaccia di sistema ospite
- Interfaccia di console OS

STUDENTSUPSI

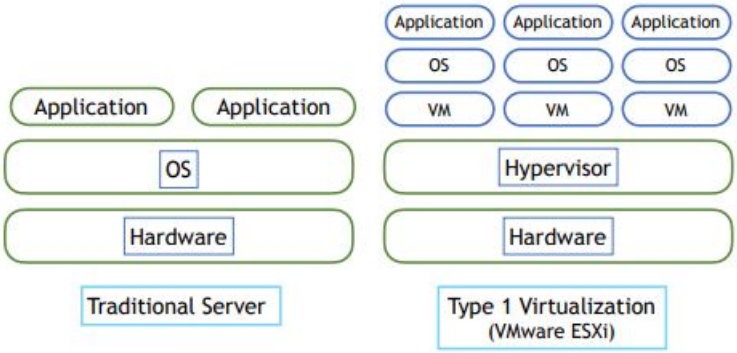


Figura 2.4: Differenza tra Un Server Tradizionale e Un Server Virtualizzato di Tipo-1

Il principale vantaggio di ESXi è quello di poter essere installato direttamente sul server fisico. Come evidenziato in figura 2.4, grazie alla possibilità di accedere alle risorse sottostanti e di controllarle direttamente, VMware ESXi partiziona l'hardware in modo efficace per consolidare le applicazioni e ridurre i costi.

2.3 Networking con Switches Virtuali su Host Virtualizzati

Gli strumenti per la virtualizzazione dei servers forniscono una grande varietà di opzioni per connettere le virtual machines alle reti. Prima di entrare nel dettaglio, è utile discutere le basi che permettono a piu VMs di comunicare fra loro all'interno di un server virtualizzato.

Per prima cosa, ogni server possiede almeno una Network Interface Card (NIC) che permette di connettersi alla rete. Le velocità tipiche sono di 10 Gbps, con alcune NIC che raggiungono i 40 Gbps.

Successivamente, pensiamo alle VMs. Il lettore è abituato a pensare che ogni OS abbia la sua NIC (a volte più di una) e questo è corretto. Essendo una VMs uno spazio virtualizzato che ospita un sistema operativo, anche questo spazio virtualizzato dovrà avere la propria (o le proprie) NIC/NICs. In questo caso, la NIC non esiste fisicamente (in quanto le uniche interfacce di rete fisiche presenti sono quelle presenti sul server) ed è quindi necessario astrarre il concetto ad una NIC virtuale, altrimenti conosciuta come *Virtual NIC*, o vNIC.

Per ultimo, il server deve combinare l'idea di una NIC fisica con una vNIC: il flusso di dati da e per le varie VMs avviene in modo simile allo switching Ethernet. Dovrà esistere un dispositivo che ossa permettere di indirizzare il traffico basandosi su quale VM lo ha generato. Anche in questo caso, tale componente non esiste fisicamente: stiamo parlando di uno switch virtuale, altrimenti conosciuto come *virtual Switch*, o vSwitch.

STUDENTSUPSI

Il capitolo seguente descrive nel dettaglio le funzioni di uno switch virtuale.

2.3.1 Virtual Switch

Un virtual switch (vSwitch) è un'applicazione software che permette a più virtual machines di comunicare fra loro. Il suo ruolo non è solamente quello di inoltrare i pacchetti, ma dirigere intelligentemente la comunicazione sulla rete controllando i frame prima di inviarli a destinazione.

Tipicamente un virtual switch è racchiuso all'interno di un software installato, ma può anche essere incluso in un server hardware come parte del firmware. Un vSwitch è completamente virtuale e può connettersi ad un'interfaccia di rete (NIC). Come vedremo in seguio, la possibilità di unire più switches fisici in uno logico lo rende un componente fondamentale per implementare SDN: grazie ad un virtual switch possiamo infatti aumentare la larghezza di banda e creare una mesh attiva tra server e switches.

Un virtual switch viene concepito per fornire un meccanismo per ridurre la complessità della configurazione di rete. Questo è permesso riducendo il numero di switches che hanno bisogno di essere gestiti dopo aver considerato la grandezza della rete, la dimensione dei pacchetti e l'architettura della rete. Dato che un virtual switch è un componente intelligente, può inoltre assicurare l'integrità del profilo della virtual machine che include impostazioni di rete e di sicurezza. Listo di seguito alcuni vantaggi offerti da uno switch virtuale:

- Sviluppo e migrazione di macchine virtuali semplificata
- Possibilità di gestire port groups attraverso un hypervisor
- Comparato ad uno switch fisico, in uno virtuale è più semplice implementare nuove funzioni sia a livello hardware che a livello software

La figura 2.5 illustra un host virtualizzato, al cui all'interno sono installate 4 VMs. La gestione del flusso di rete è affidata ad un virtual switch.

Si noti come ogni VM possa condividere la stessa VLAN oppure utilizzarne una privata. La virtualizzazione è gestita via software e permette di spostare le VMs tra più servers, riservare più spazio per una e meno per un'altra, aumentare oppure diminuire i cores e molto altro ancora.

Abbiamo due possibilità per utilizzare un virtual switch:

- Utilizzare il vSwitch fornito dall'hypervisor

STUDENTSUPSI

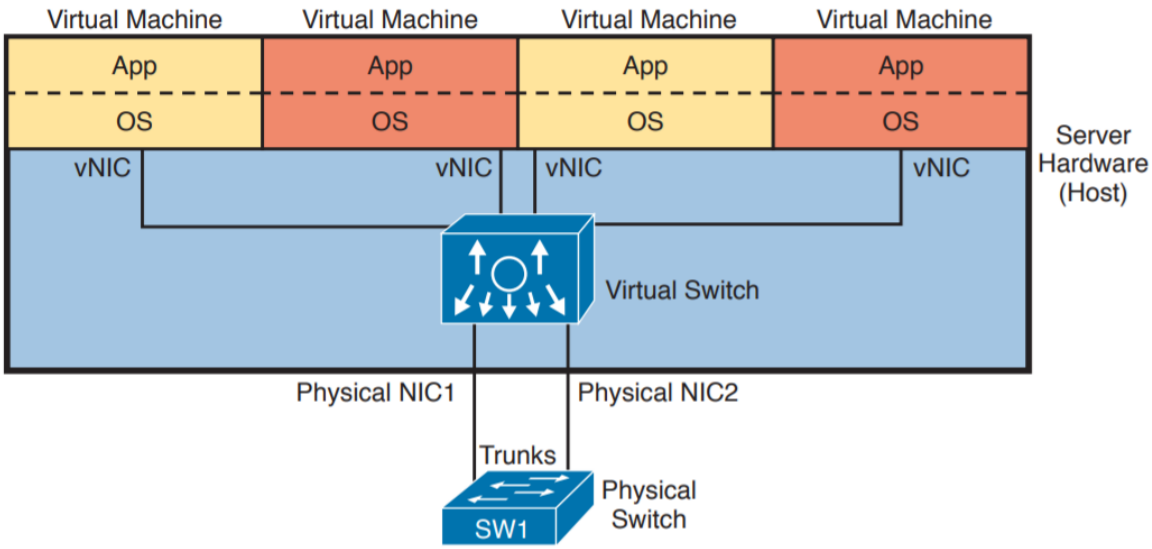


Figura 2.5: Networking in un Host Virtualizzato con Uno Switch Virtuale

- Utilizzare un vSwitch di terze parti

Nel mio caso ho scelto la prima opzione. Un esempio di vSwitch di terze parti è il modello *Nexus 1000v*, fornito da Cisco. La sezione seguente elenca le caratteristiche di questo dispositivo.

2.3.2 Cisco Nexus 1000v

Cisco Nexus 1000v è un vSwitch destinato all'utilizzo in ambienti virtuali, come VMware vSphere e Microsoft Hyper-V. Questo dispositivo è divisibile in due parti principali:

- Virtual Ethernet Module (VEM)
- Virtual Supervisor Module (VSM)

La seguente sezione spiega il comportamento dei due moduli appena citati.

2.3.2.1 Virtual Ethernet Module

Il *Virtual Ethernet Module* (VEM) è attivo come parte del software hypervisor. Il VEM ottiene informazioni di configurazione dal VSM e opera switching di livello 2 insieme a funzioni di rete più avanzate.

STUDENTSUPSI

Nel caso di perdita di comunicazione con il VSM, VEM ha la capacità di continuare a inoltrare il traffico basandosi sulla configurazione più recente memorizzata (*Non Stop Forwarding*, o NSF).

2.3.2.2 Virtual Supervisor Module

La gestione dei molteplici VEMs avviene da parte del *Virtual Supervisor Module* (VSM). Il controllo è effettuato sotto forma di uno *Switch Modulare Virtuale Distribuito*, ovvero un vSwitch che percmette la gestione di più moduli (VEMs nel nostro caso). La configurazione è effettuata dal VSM e viene propagata ai vari VEMs.

La figura sottostante illustra il concetto.

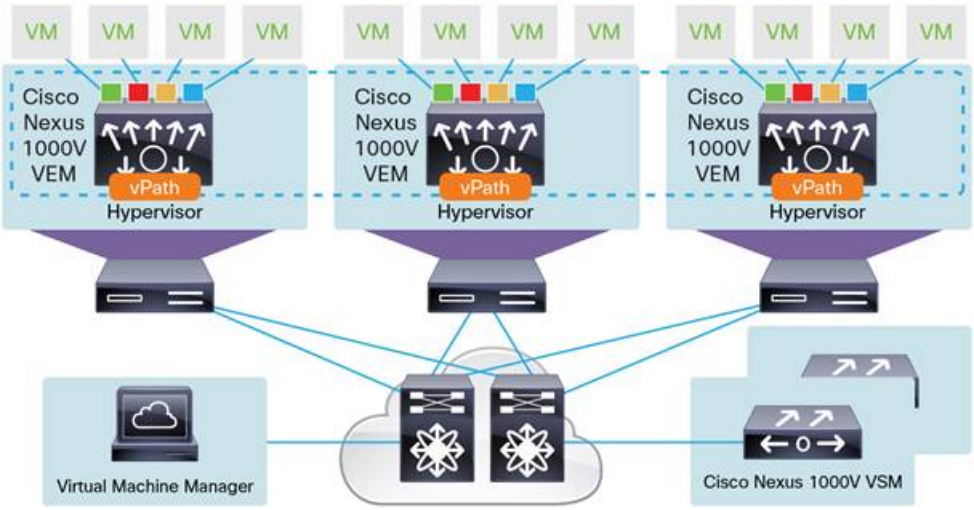


Figura 2.6: Gestione di VEMs da Parte di Un VSM

Nel complesso, il VSM agisce come control plane del vSwitch mentre il VEM rappresenta il data plane.

STUDENTSUPSI

2.4 Vantaggi della Virtualizzazione

Negli anni, l'elettronica dei microprocessori si è evoluta per offrire più prestazioni occupando meno spazio. Allo stesso modo, i softwares cercano di utilizzare il minor numero di risorse possibili per svolgere il proprio lavoro.

Elenco di seguito i benefici della virtualizzazione di servers:

1. Scalabilità
2. Quality of Service (QoS)
3. Riduzione del costo di hardware
4. Tempi di network provisioning e di distribuzione inferiori
5. Recupero di downtime migliorato
6. Risparmio sui costi di elettricità e raffreddamento del data center
7. Produttività più elevata

Le seguenti pagine vogliono spiegare nel dettaglio i vantaggi appena elencati.

2.4.1 Scalabilità

Viene definita *scalabilità* la proprietà di un sistema di gestire un ammontare di lavoro crescente aggiungendo risorse al sistema stesso.

In ottica cloud computing ad esempio, è di fondamentale importanza gestire il desiderio di un utente di ottenere più spazio virtuale: questo è permesso dalla capacità della VM di aumentare le proprie dimensioni, ovvero di essere per l'appunto *scalabile*.

La scalabilità di un sistema può essere di due tipi: verticale e orizzontale.

2.4.1.1 Scalabilità Verticale

Con *scalabilità verticale* (Up/Down) si indica l'aggiunta di risorse ad un singolo nodo. Tipicamente le risorse coinvolte sono CPU, RAM o memoria.

Come vedremo in seguito, scalare un sistema verticalmente è vantaggioso fino ad un determinato punto, dopo il quale è più conveniente scalare orizzontalmente.

STUDENTSUPSI

2.4.1.2 Scalabilità Orizzontale

Al contrario di quella verticale, la *scalabilità orizzontale* (Out/In) permette di aggiungere risorse ad un sistema aumentando il numero di nodi dello stesso. Un esempio è l'aggiunta di un secondo computer (nodo) in modo tale da aumentare la produttività del sistema.

La figura 2.7 mostra l'efficienza relativa ad una scalatura verticale in contrasto ad una orizzontale:

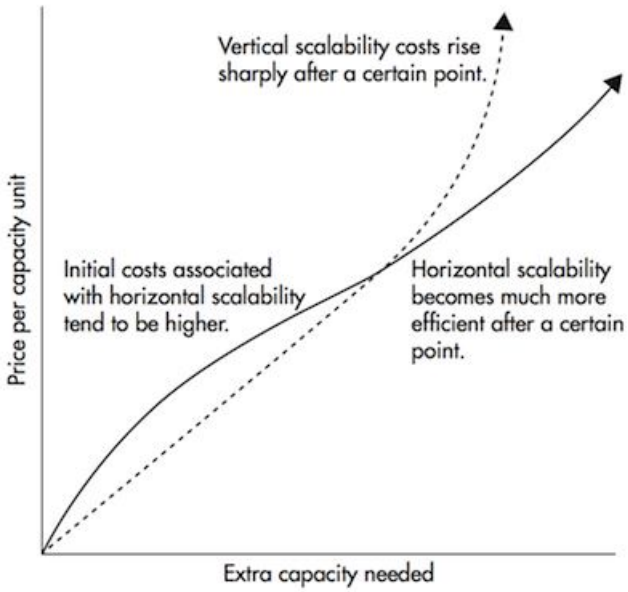


Figura 2.7: Confronto tra Scalabilità Verticale e Orizzontale

Come facilmente visibile dalla figura soprastante, la scalabilità orizzontale diventa significativamente più economica dopo una certa soglia.

2.4.2 Quality of Service (QoS)

Quality of Service (QoS) è la descrizione della performance globale di una determinata prestazione, come ad esempio una telefonata oppure un servizio di cloud computing.

Dato che spesso la performance è vista dal punto di vista degli utilizzatori della rete (ovvero in maniera qualitativa), per misurare la QoS quantitativamente vengono misurati tanti aspetti, tra cui:

- Packet Loss

STUDENTSUPSI

- Bit Rate
- Throughput
- Latenza
- Disponibilità

In questa tesi si vuole focalizzare l'attenzione sulla QoS relativa alla prioritizzazione del traffico, ovvero prediligere alcuni pacchetti rispetto ad altri. È possibile apprezzare questa caratteristica quando il traffico in rete è elevato (congestione della rete) dove si nota che i pacchetti con priorità vengono consegnati con un *loss time* garantito, mentre quelli senza priorità presentano un tempo di perdita variabile.

2.4.3 Riduzione del Costo di Hardware

Quando un server viene virtualizzato, si sfruttano tutte le risorse disponibili, evitando di sprecare potenza computazionale che rimarrebbe inutilizzata. Non vi è più il bisogno di installare molteplici servers sui racks e nemmeno la necessità di comprare i cavi necessari a collegarli fra loro.

2.4.4 Tempi di Network Provisioning e di Distribuzione Inferiori

Con il termine *network provisioning* si indica il tempo necessario per preparare ed equipaggiare una rete e renderla pronta per essere utilizzata. Virtualizzando un server, il tempo di messa a punto della suddetta rete è inferiore se comparato ad un server non virtualizzato, così come la distribuzione della stessa.

2.4.5 Recupero di Downtime Migliorato

Ci si riferisce al termine *downtime* come il periodo di tempo durante il quale un sistema è inutilizzabile. Ponendo per esempio il caso di un guasto ad un server, tutte le applicazioni attive su di esso sarebbero inutilizzabili. Uno dei principali benefici della virtualizzazione è la possibilità di ridurre il downtime spostando le VMs su uno server differente, e di conseguenza renderle nuovamente disponibili.

STUDENTSUPSI

2.4.6 Risparmio Sui Costi Statici

In relazione al punto #1 di questo elenco, un minor numero di hardware significa meno elettricità richiesta. Questo implica sia i costi di alimentazione dell'hardware sia quelli relativi al raffreddamento dell'ambiente che lo circonda (data center). In accordo con VMware, la virtualizzazione dei servers può ridurre i costi fino all'80%.

2.4.7 Produttività Più Elevata

Avere meno servers fisici significa doverne mantenere e gestire di meno. In relazione al punto #2 di questa lista, un periodo inferiore di provisioning e di gestione permettono ai managers di rete di focalizzarsi su altri compiti, risparmiando tempo.

2.5 Virtualizzazione - Laboratorio

Dopo aver spiegato i concetti di virtualizzazione server, analizziamo un caso pratico sviluppato in laboratorio. Consideriamo quindi la seguente topologia di rete:

La figura 2.8 mostra un blocco principale (il server fisico) che è connesso alla WAN del laboratorio (IP: 193.246.121.0/24). La WAN si connette ad Internet tramite il firewall di laboratorio.

Vogliamo focalizzare l'attenzione sul blocco server collegato alla WAN del laboratorio. Notiamo che i nodi al suo interno sono:

- Un firewall
- Un client Windows
- Un client Linux
- Un server Windows
- Un server Web

STUDENTSUPSI

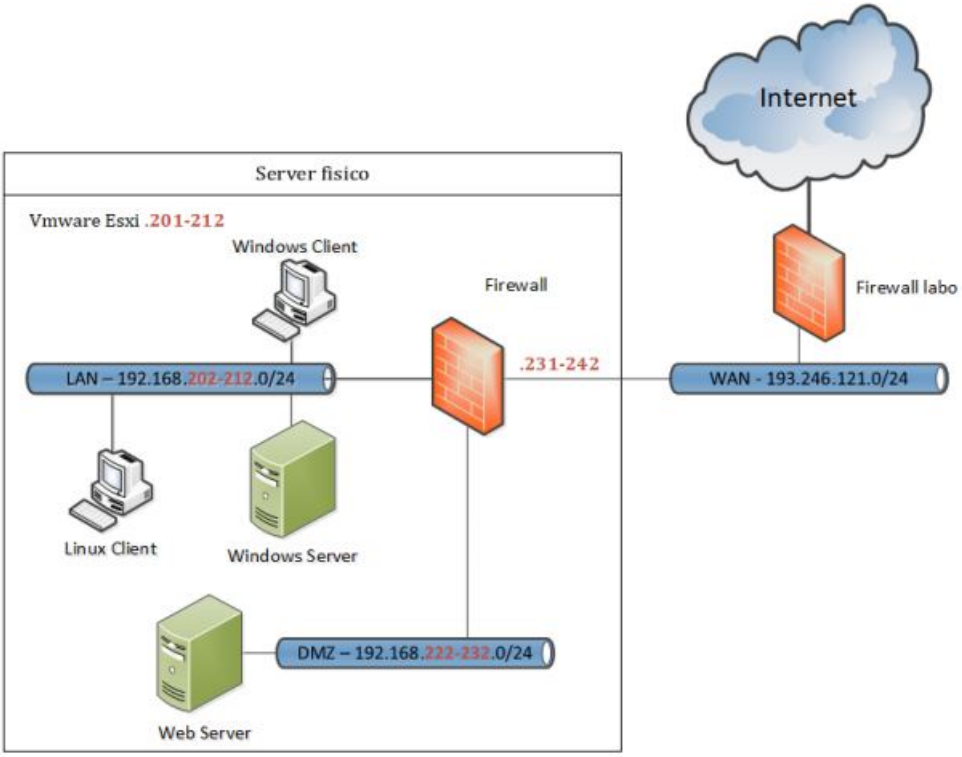


Figura 2.8: Topologia di Rete da Realizzare All'Interno di Un Server Fisico

2.5.1 Firewall

La topologia di rete rappresentata in figura 2.8 è definita *modello a tre gambe*, o a singolo firewall. Tutto il traffico proveniente dalla LAN e dalla DMZ deve obbligatoriamente passare da questo componente.

Nel mio caso ho scelto di utilizzare il firewall open source *pfSense*, conosciuto per la sua sicurezza, potenza e elevata possibilità di essere configurato.

Figura 2.9: Logo del Firewall pfSense

Come anticipato precedentemente, ponendosi come inframezzo tra la rete LAN e la rete DMZ, è logico pensare che tutto il traffico generato dalle suddette reti debba passare prima per il firewall. Il comportamento è quindi quello di un *default gateway*, a cui ho impostato un

STUDENTSUPSI

indirizzo IP diverso per ogni sottorete a cui è connesso:

- Per la LAN: 192.168.202.1/24
- Per la DMZ: 192.168.222.1/24
- Per la WAN: 193.246.121.40/24

2.5.2 LAN

La LAN presente dietro il firewall appena descritto è composta dai due clients (Windows e Linux) assieme ad un server Windows. Questi tre nodi sono interconnessi fra loro mediante un vSwitch che ho chiamato *vSwitch - LAN*, e relativo port group: *LAN - Port Group*.

Il server Windows è stato concepito per fornire servizi DHCP e DNS ai clients della LAN. A questo nodo ho associato un indirizzo IP statico (192.168.202.2/24), mentre gli altri dispositivi otterranno un indirizzo IP nel range che va da 192.168.202.128/24 fino a 192.168.202.254/24.

2.5.3 DMZ

La *DeMilitarized Zone* (o DMZ), è una sottorete isolata che permette agli utenti esterni ad un'azienda di sfruttare dei servizi offerti dalla stessa.

La particolarità di una DMZ è il fatto di potere essere accessibile sia da reti esterne all'azienda (WAN) che da reti interne (Intranet), ponendo quindi tutti quei servizi che devono essere condivisi con l'esterno in questa zona e lasciando i servizi privati in un'altra sottorete. In questo modo un dispositivo esterno potrà accedere solamente ai servizi offerti dall'azienda, senza mettere a rischio l'intera rete aziendale in caso di attacchi informatici.

All'interno della DMZ ho sviluppato un web server che potesse permettere agli utenti esterni di visualizzare le informazioni relative alla ditta in questione attraverso il web browser. Ho scelto di utilizzare il server web Linux-based più famoso in commercio: *Apache HTTP Server*.

STUDENTSUPSI

Figura 2.10: Logo del Web Server Apache

Si tratta di un progetto open source multipiattaforma che offre molti servizi, tra cui il più importante è sicuramente quello di risolvere le queries dei clients alle pagine web, e lo fa utilizzando il protocollo HTTP (*HyperText Transfer Protocol*).

Come nel caso della LAN, anche nella DMZ ho creato uno switch virtuale (*vSwitch - DMZ*) assieme ad un *DMZ - Port Group*, a cui ho collegato il web server. L'indirizzo IP che ho impostato per questo nodo è: 192.168.222.2/24.

2.6 Arista EOS e vEOS

Ora che i concetti di virtualizzazione server sono stati esaminati nel dettaglio, si vuole realizzare una configurazione che possa permettere di utilizzare un vSwitch non ESXi - based. Tutti i nodi che andrò a costruire saranno gestiti da VMware ESXi (sezione 2.2.2), all'interno del server di laboratorio.

Come già anticipato nel capitolo 1, per poter programmare uno switch è necessario che quest'ultimo sia effettivamente configurabile mediante script, caratteristica che non tutti possiedono. Solo negli ulimi tempi ci si è accorti dei vantaggi che SDN fornisce, e quindi la tendenza dei produttori di dispositivi di networking è quella di fornire pieno supporto all'automazione e la programmabilità di rete. Tra questi produttori ho scelto di utilizzare una soluzione di *Arista Networks*, una ditta che negli ultimi tempi sta prendendo sempre più spazio nel mercato.

Il prodotto scelto prende il nome di *Arista vEOS*. Si tratta di una soluzione open source, i cui dettagli vengono analizzati nella prossima sezione.

STUDENTSUPSI

Arista EOS (*Extensible Operative System*) è una piattaforma software concepita per fornire supporto ai bisogni aziendali dei data centers di nuova generazione e le reti cloud based. Si tratta di un sistema operativo a switch modulare, con un'unica architettura di condivisone che permette di separare lo stato dello switch dalla logica di applicazione e processamento dei protocolli. Essendo costruito su un kernel Linux standard, tutti i processi EOS sono trattati in zone di memoria dedicate e protette. I processi condividono le informazioni mediante un database, rendendo questa struttura estremamente robusta, affidabile, stabile ed estensibile.

La seguente figura mostra il concetto appena descritto: si noti come tutti i processi siano separati fra loro. L'unico modo che questi hanno per scambiarsi informazioni è la presenza di *NetDB*, il quale rappresenta il database condiviso dai processi.

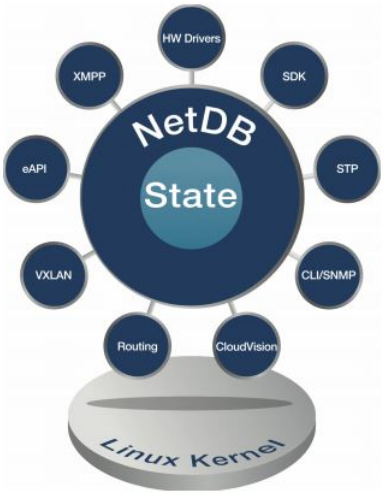


Figura 2.11: Arista vEOS Nel Dettaglio

Arista offre la possibilità di testare EOS in un ambiente virtuale come VMware ESXi: il prodotto prende il nome di *Arista vEOS*. Essendo un dispositivo emulato all'interno di un server, valgono tutti i principi elencati nel capitolo 3.4: Vantaggi della Virtualizzazione.

La particolarità di vEOS è la sua unicità: ogni switch di Arista è infatti basato su EOS, ed è quindi possibile simulare qualsiasi tipo di switch all'interno di questo ambiente virtuale (al contrario di Cisco in cui ogni componente richiede la sua immagine dedicata).

Per l'analisi nel dettaglio di Arista vEOS rimando a `https://www.arista.com/assets/data/pdf/Datasheets/vEOS_Datasheet.pdf`.

STUDENTSUPSI

2.7 vEOS - Laboratorio

La topologia di rete che andrò a costruire in laboratorio è rappresentata in figura 2.12.

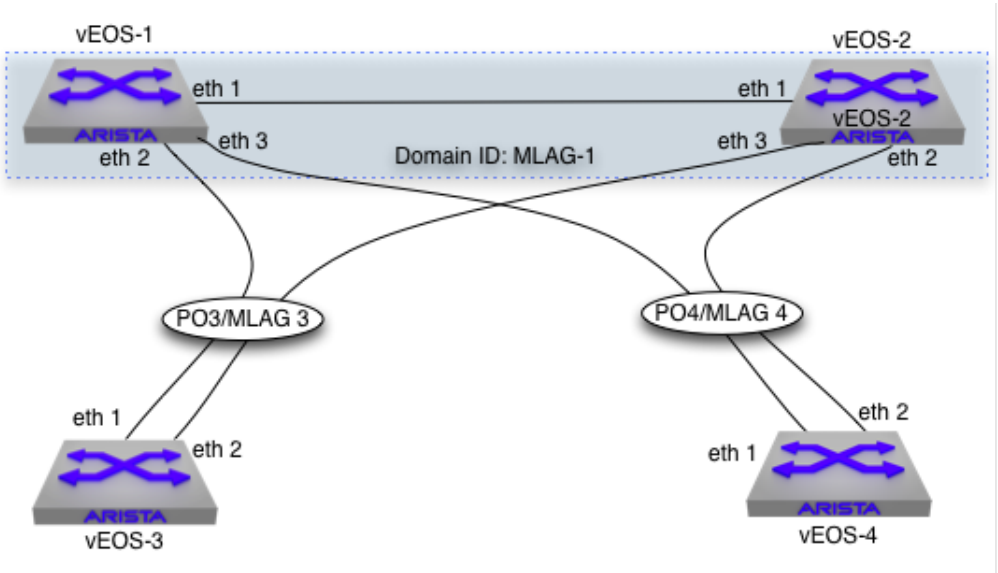


Figura 2.12: Quattro Arista vEOS Switches in MLAG

Si tratta della topologia di rete più usata nei data centers moderni: vEOS-3 e vEOS-4 rappresentano la parte *Leaf* mentre vEOS-1 e vEOS-2 corrispondono alla parte *Spine*. Si notino i collegamenti diretti tra vEOS-1 e vEOS-4 e quelli tra vEOS-2 e vEOS-3: normalmente questi collegamenti protrebbero essere tralasciati (dato che vEOS-1 e vEOS-2 sono connessi tra loro). Arista sfrutta invece questi link per ottenere una QoS più elevata: questa tecnica è definita MLAG (*Multi Link AGgregation*), come si può notare in figura. Poniamo per esempio il caso in cui un collegamento dovesse essere interrotto (link tra vEOS-1 e vEOS-3): se non ci fosse il collegamento tra vEOS-3 e vEOS-2, come facilmente intuibile vEOS-3 sarebbe inutilizzabile. Con MLAG invece, vEOS-3 può continuare a comunicare con vEOS-1 dato che è fisicamente connesso con vEOS-2, e quindi può inviare un pacchetto a vEOS-1 sfruttando il collegamento con vEOS-3. Per queste ragioni, MLAG riduce esponenzialmente il rischio di downtime del sistema.

Ciò che accade nel caso cadesse un link in questo sistema viene descritto graficamente in figura 2.13:

La comunicazione è volutamente ridondante, soluzione conosciuta anche come *Multipath-TCP*.

Per enfatizzare il concetto, si noti la ridondanza di collegamenti tra leaf e spine nella figura seguente:

STUDENTSUPSI

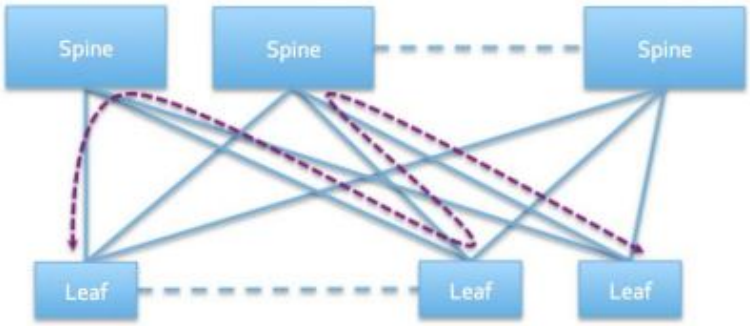


Figura 2.13: Recupero di Un Link Tramite un Percorso Alternativo

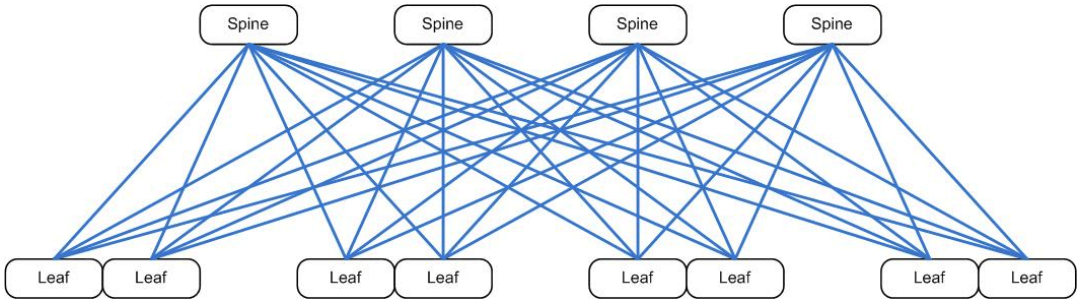


Figura 2.14: Configurazione Leaf Spine in MLAG

In una rete di questo tipo il downtime dovuto alla rottura di un componente è stimato essere prossimo allo 0%, dato che esisterà sempre un percorso alternativo per il pacchetto.

Il capitolo che segue vuole introdurre il lettore ai concetti di programmabilità di rete e SDN, fornendo una visione a largo spettro dei concetti generali, per poi entrare nel dettaglio nei capitoli successivi.

STUDENTSUPSI

Capitolo 3

SDN e Programmabilità di Rete

Il vasto argomento denominato *Software Defined Networking*, sinonimo di *Network Programmability*, introduce un nuovo modo di costruire le reti. La possibilità di programmare le reti cambia il modello operazionale richiesto, richiedendo ora nuovi metodi di controllo dei dispositivi per abilitare i nuovi schemi.

Il termine stesso *Network Programmability* si riferisce al controllo via software della rete, in modo tale che questa possa essere modificata più facilmente per adattarsi all'ambiente in continua evoluzione che ci circonda: recentemente le Virtual Machines (VM) vengono implementate nei Data Centers, il pattern del traffico di dati viene modificato, e così via. Lo stesso termine *Software Defined Networking* (SDN) enfatizza questo concetto: il controllo della rete avviene via software, e non più a livello statico.

Va ricordato che i dispositivi di rete inoltrano dati sotto forma di messaggi, tipicamente frames a livello data link (dei quali il più comune di questi è il frame Ethernet).

La programmabilità di rete e SDN non si discostano da questa idea di base: entrambe analizzano i princìpi e cercano un modo per migliorare le prestazioni delle reti odierne, riassemblando questi concetti in un nuovo modo che possa permettere ai dispositivi in rete di continuare ad inoltrare i pacchetti; ciò che cambia è la modalità con cui questi verranno inviati.

La prima sezione di questo capitolo vuole introdurre il lettore ai concetti principali che si trovano dietro ad SDN e alla programmabilità di rete. Partendo dalle caratteristiche di una rete tradizionale, vengono analizzati i dispositivi che la caratterizzano. Successivamente viene descritto il software di controllo centralizzato (denominato *Controller*) che è il responsabile dell'architettura di rete programmata.

STUDENTSUPSI

3.1 Data, Control e Management Planes

Ogni funzione che un dispositivo di rete svolge può essere associata ad uno fra tre gruppi distinti, denominati *planes*:

- Data Plane
- Control Plane
- Management Plane

Nei paragrafi seguenti andremo ad analizzare nel dettaglio la funzione di ogni piano. Tutte le descrizioni sono accompagnate da un esempio che illustra il concetto spiegato.

3.1.1 Data Plane

Il termine *Data Plane* si riferisce a tutti i compiti che un dispositivo di rete deve svolgere per inoltrare un messaggio.

In altre parole, qualsiasi funzione che coinvolga la ricezione, il processamento o l'inoltro del frame è parte del data plane.

Per esempio, pensiamo al lavoro svolto da un router per inoltrare pacchetti IP (figura 3.1). Focalizzandoci a L3, l'host invia il pacchetto (Step 1) al suo router di default R1. R1, R2 e R3 eseguono lo stesso lavoro svolto da R1: prendono decisioni di routing e inoltrano il pacchetto (Step 2, 3 e 4).

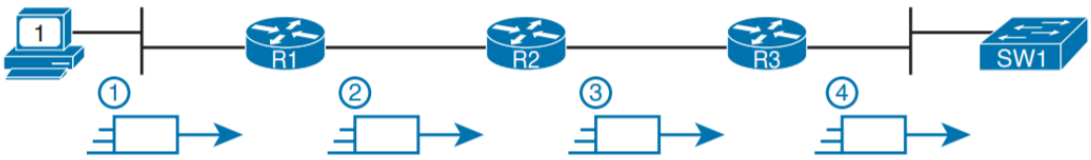


Figura 3.1: Data Plane Processing

Il concetto appena descritto può essere applicato anche ad uno switch dato che anch'esso riceve, processa ed inoltra il messaggio. Chiaramente la decisione di inoltrare il dato (processo di *forwarding*) è parte della logica. Per queste ragioni, il data plane è spesso chiamato *forwarding plane*.

STUDENTSUPSI

La seguente lista elenca alcune tra le più comuni azioni che un dispositivo di rete svolge che appartengono al data plane:

- De-encapsulare e re-incapsulare un pacchetto in un frame data link (Routers, L3 Switches)
- Aggiungere o rimuovere l'header di trunking 802.1Q
- Confrontare la destinazione del MAC address con la MAC address table (L2 Switches)
- Confrontare l'indirizzo IP del destinatario con la tabella di routing IP (Routers, L3 Switches)
- Criptare i dati e aggiungere un nuovo header IP (Per processamento VPN)
- Cambiare la sorgente o la destinazione di un indirizzo IP (Per processamento NAT)
- Scartare un messaggio a seguito di un filtro (ACLs, sicurezza sulle porte)

Tutti gli elementi listati precedentemente fanno parte del data plane.

3.1.2 Control Plane

Una volta descritto il data plane dobbiamo porci la seguente domanda: che tipo di informazioni necessita questo piano per inoltrare correttamente i dati? I routers hanno bisogno dei percorsi IP (memorizzati nella tabella di routing), gli switches di L2 necessitano della propria MAC address table prima di poter decidere su che porta inoltrare i frame Ethernet appena processati. Per evitare che un pacchetto entri in un loop infinito, gli switches devono altresì limitare il numero di interfacce che possono essere usate per inoltrare i frames, come descritto dal protocollo STP (Spanning Tree Protocol).

Possiamo quindi affermare che le informazioni fornite al data plane controllano il lavoro che esso svolge. Pensiamo per esempio ad un router che non ha nessun percorso memorizzato nella propria tabella di routing: questo dispositivo non potrà inoltrare pacchetti. Eppure il data plane esiste fisicamente, semplicemente nessuno ha fornito al router le informazioni necessarie per poter inoltrare il pacchetto. In questo caso il router cerca di trovare il percorso dove inoltrare il pacchetto, ma dato che la tabella è vuota (nessun percorso esistente) il frame viene scartato. Se la tabella non fosse vuota, il data plane potrebbe a quel punto inoltrare il pacchetto. A questo punto la domanda sorge spontanea: chi controlla il contenuto della tabella di routing?

STUDENTSUPSI

Qualsiasi azione di controllo del data plane è associata al *control plane*. La maggior parte di queste azioni hanno a che fare con la creazione di tabelle usate dal data plane, come la tabella di routing IP, la tabella ARP, quella di MAC address e così via. Aggiungendo, togliendo e modificando gli ingressi delle tabelle usate dal data plane, il control plane gestisce il lavoro del data plane.

I protocolli di rete tradizionali separano control plane e data plane all'interno dello stesso dispositivo in due entità separate, come mostrato in figura 2.2. In questo caso, il protocollo a livello di controllo OSPF è adottato dai tre routers (il protocollo è *distribuito* tra tutti i routers). OSPF aggiunge, rimuove e modifica la tabella di routing IP di ogni router. Una volta che la tabella è popolata da percorsi utili, il data plane (anch'esso distribuito fra ogni router) può inoltrare i pacchetti in ingresso, come mostrato da sinistra a destra nella figura sottostante:

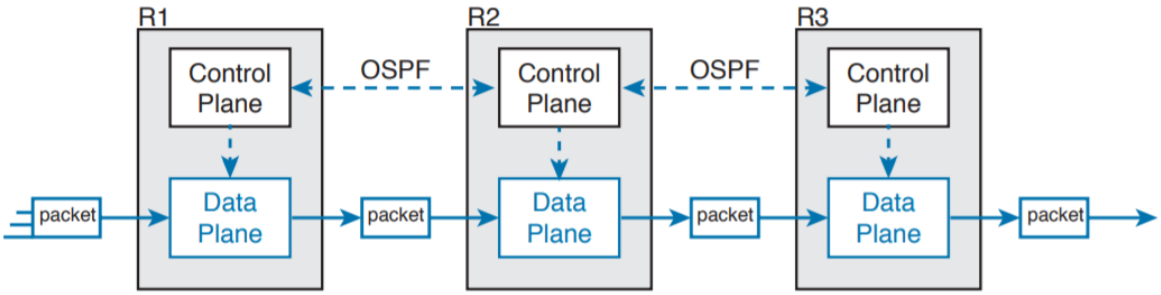


Figura 3.2: Control and Data Planes di Tre Routers

La seguente lista elenca tanti dei più comuni protocolli di controllo:

- Protocolli di routing (OSPF, EIGRP, RIP, BGP)
- IPv4 ARP
- IPv6 NDP
- Memorizzazione dei nuovi MAC addresses di uno switch
- STP

Senza i protocolli e le attività del control plane, il data plane non può funzionare a dovere. Un router con una tabella di routing vuota non è che un dispostivo inutile, come uno switch senza una sua MAC address table. Certo, lo switch potrebbe inoltrare ogni pacchetto in rete floodando il frame (L1 hub), ma ciò implicherebbe un traffico esagerato sulla LAN con conseguenti ritardi in ricezione. Il data plane si affida quindi al control plane per svolgere il proprio lavoro nel miglior modo possibile.

STUDENTSUPSI

3.1.3 Management Plane

Il terzo ed ultimo piano che andremo ad analizzare è relativo alla gestione del dispositivo. Il *management plane* non impatta il lavoro del data plane, ma include protocolli che permettono agli ingegneri di rete di gestire in modo più semplice il sistema.

I due protocolli di gestione più ovvi sono Telnet ed SSH. Per enfatizzare la differenza con i protocolli del control plane, pensiamo a due routers, uno configurato per permettere Telnet e SSH, l'altro che non supporta queste tecnologie. Entrambi possono inoltrare pacchetti, sia che supportino Telnet e SSH o meno.

La figura 2.3 mostra alcuni dei più comuni protocolli a livello di management plane:

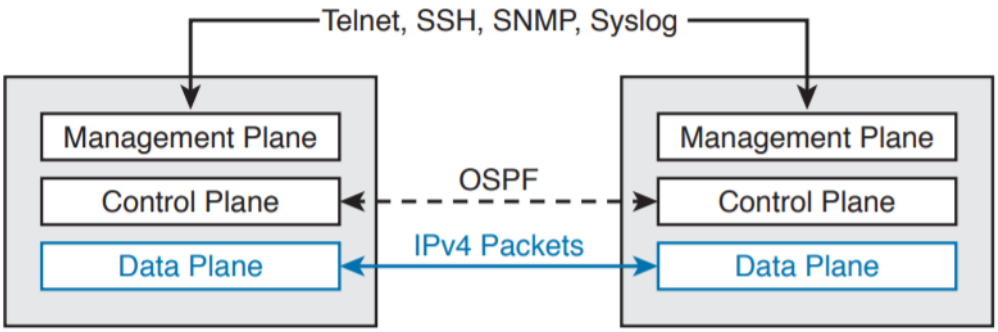


Figura 3.3: Management Plane per la Configurazione del Control Plane e del Data Plane

3.1.4 Data Plane di Uno Switch - Dettaglio

Per capire meglio SDN e la programmabilità di rete, è utile pensare alla composizione interna di uno switch. Il seguente capitolo si occupa di questo concetto.

Sin dagli albori dei primi switches LAN, l'hardware all'interno di questi dispositivi era costruito specificatamente per gestire un numero elevatissimo di fps (frames per secondo) in ingresso. Per dare un'idea del volume di pacchetti che uno switch deve processare, consideriamo la grandezza minima di un frame Ethernet, il numero di porte di uno switch e la velocità di queste. Persino gli switches di fascia bassa devono essere pronti a processare *milioni* di frames per secondo. Prendiamo per esempio uno switch con queste caratteristiche:

- Porte: 24
- Velocità di ogni porta: 100 Mbps ognuna

STUDENTSUPSI

- Tipo di porte: full duplex (in qualsiasi momento lo switch può ricevere e/o trasmettere)
- Dimensione di un frame: 125 bytes ognuno

Calcoliamo la velocità di una porta in Bytes al secondo:

$$\frac{100\ Mbps}{8} = 1.25 \cdot 10^6\ Bps \tag{3.1}$$

Dato che la dimensione di ogni frame di 125 Bytes, la porta riceverà:

$$\frac{1.25 \cdot 10^6\ Bps}{125\ Bytes} = 10^5\ fps \tag{3.2}$$

Essendo lo switch costituito da 24 porte, il traffico totale in ingresso sarà:

$$10^5\ fps \cdot 24 = 2.4 \cdot 10^6\ fps \tag{3.3}$$

Un numero così elevato di pacchetti da processare deve essere gestito da un hardware dedicato e non dalla CPU dello switch. Questo dispositivo è denominato *Application Specific Integrated Circuit* (ASIC). Un ASIC è un circuito costruito specificatamente per far fronte ad un determinato problema, come nel nostro caso la gestione di un volume molto elevato di pacchetti in ingresso.

Secondariamente, l'ASIC ha bisogno di consultare la tabella di MAC address dello switch nel minor tempo possibile. Per fare ciò, lo switch utilizza una memoria dedicata, chiamata *Ternary Content Addressable Memory* (TCAM) per memorizzare il contenuto equivalente della MAC address table. La TCAM non ha bisogno che l'ASIC cerchi nella tabella di MAC address: essa ritorna l'indirizzo corrispondente all'input immesso, senza nessun algoritmo di ricerca.

Si noti in figura 3.4 come uno switch presenti sia una CPU che una RAM. Il sistema operativo viene gestito dalla CPU ed usa la memoria RAM, come anche il control plane ed il management plane. Per quanto riguarda il data plane, questo è associato all'ASIC e alla TCAM.

In modo simile agli switches, anche alcuni routers utilizzano dispositivi hardware per svolgere funzioni a livello data plane. L'idea alla base per entrambi i dispositivi è la stessa: usare un chip integrato dedicato per la logica di forwarding (ASIC) ed una TCAM per massimizzare la velocità di comparazione degli indirizzi.

STUDENTSUPSI

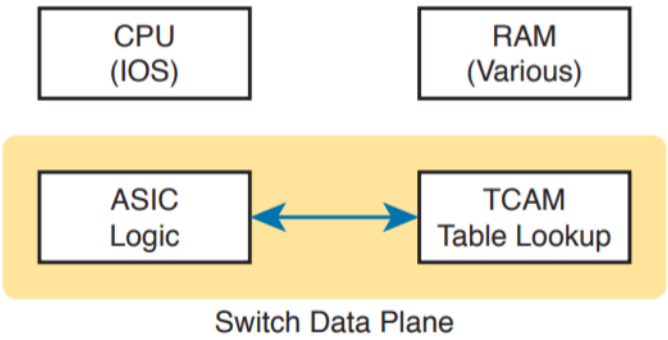


Figura 3.4: Switch - Processi Interni e Registri

3.2 Controllori e Architettura di Rete

Negli ultimi anni le reti moderne si sono evolute notevolmente. Nuove tecniche hanno permesso di cambiare alcune delle funzionalità del control plane in modo sostanziale. La tendenza di oggi è quella di *centralizzare* il piano di controllo a livello software, creando un software di controllo centralizzato denominato *controller*. Questo capitolo spiega i concetti relativi al controller, le interfacce e i dispositivi che si trovano al di sotto di esso e i programmi che ne fanno uso.

3.2.1 Controllori e Controllo Centralizzato

Oggi, la maggior parte dei processi tradizionali a livello di control plane utilizza un'architettura distribuita. Con questo si intende un control plane distribuito, attivo su più dispositivi in real time. Per esempio, ogni router possiede il proprio processo di controllo OSPF. Per svolgere il proprio lavoro insieme, questi processi di controllo distribuito utilizzano messaggi per comunicare fra di essi. Come risultato, il control plane delle suddette reti viene definito *distributed control plane*.

Chi ha creato i moderni concetti di control plane avrebbe potuto scegliere di utilizzare un control plane centralizzato invece di uno distributo, e conseguentemente gestire la logica in un unico dispositivo (per esempio un server). Il software centralizzato avrebbe potuto utilizzare messaggi per imparare le informazioni dai dispositivi, ma con il cuore dell'architettura di processamento in un posto centralizzato. Invece, è stato scelto di sviluppare un'architettura distribuita.

Ci sono pro e contro nel costruire reti ad architettura distribuita o centralizzata. Tante funzioni del control plane funzionano bene in un'architettura distribuita, e spesso quando qualcosa

STUDENTSUPSI

lavora come dovrebbe si è restii a volerla cambiare. Tuttavia, un'applicazione centralizzata può essere più semplice da scrivere rispetto ad una distributa, perché questa raccoglie tutti i dati in un unico posto. Questo mondo emergente della programmabilità di rete e SDN spesso utilizza un'architettura centralizzata, con un control plane centralizzato, e i suoi fondamenti racchiusi in un servizio definito controller.

Un *controller*, o *SDN controller*, centralizza il controllo dei dispositivi di rete. Il grado con il quale il controllo è centralizzato può variare, dal controllore che esegue tutte le funzioni del control plane, fino a quello che monitora solamente la rete.

Per capire meglio questa idea del controller, consideriamo il caso in figura 3.5, in cui un controller SDN centralizza tutte le funzioni importanti del control plane. Per prima cosa, il controllore viene posto in un punto da cui può essere raggiunto da tutti i dispositivi in rete. Ogni dispositivo in rete ha ancora il proprio data plane, tuttavia, si noti come ad ognuno manchi il control plane. In ottica SDN (come mostrato in figura 3.5) il controller (oppure un programma che ne fa uso) programma direttamente gli ingressi del data plane nelle tabelle di ogni dispositivo. I routers e gli switches non riempiono più le proprie tabelle con i protocolli a livello di controllo tradizionali.

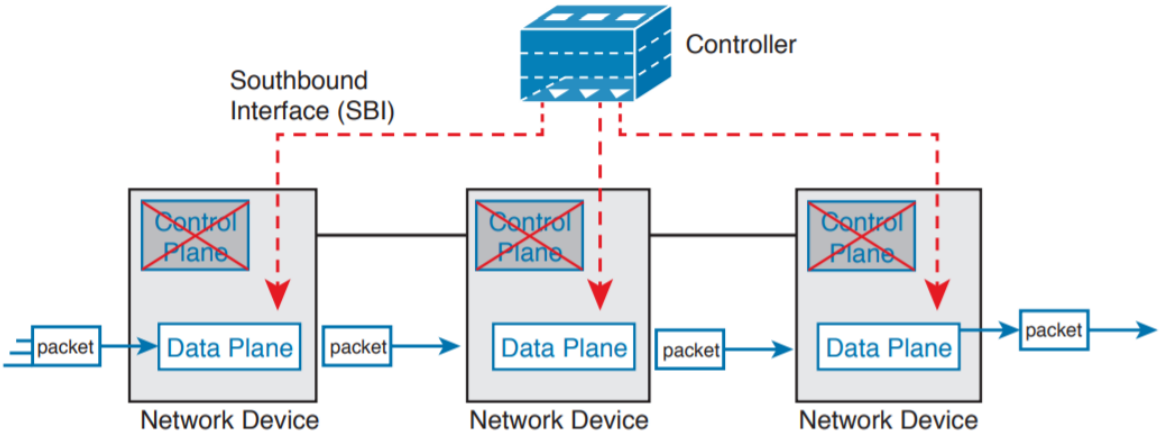


Figura 3.5: Control Plane Centralizzato e Data Plane Distribuito

La figura 3.5 mostra un modello per la programmabilità di rete e SDN, ma non tutte le architetture sono di questo tipo. Una volta esposta l'idea generale di centralizzazione possiamo approfondire l'argomento. Nel successivo capitolo andremo ad analizzare due concetti importanti: Southbound Interface e Northbound Interface.

STUDENTSUPSI

3.2.2 Southbound Interface

Nelle architetture di rete basate su un controller vi è la necessità che quest'ultimo comunichi con i dispositivi connessi ad esso. Nella maggior parte delle rappresentazioni visive delle architetture, questi dispositivi si trovano al di sotto del controller, come mostrato in figura 3.5. Analizzandola nel dettaglio, notiamo che tra il controller e i dispositivi vi è un'interfaccia, denominata *Southbound Interface*, altrimenti conosciuta come SBI.

Tenendo in mente che il nostro obiettivo finale sarà la possibilità di programmare la rete, SBI deve svolgere altre funzioni oltre a quelle a livello di protocollo. Nella maggior parte delle architetture infatti SBI contiene al suo interno un protocollo che permetta al controller di comunicare con i dispositivi, ma spesso include una *Application Programming interface*, o API.

Una API è un metodo che permette ad una applicazione di scambiare informazioni con un'altra. Per descrivere l'idea sotto un altro punto di vista, una API è un'interfaccia destinata ad un programma applicativo. I programmi processano i dati, quindi l'API permette lo scambio di questi ultimi. Mentre un protocollo esiste come documento, API esiste come codice (variabili, funzioni e strutture di dati) che possono essere utilizzate da un programma per scambiare dati con un altro in rete.

Quindi, ritornando al termine *SBI*: è un'interfaccia tra un programma (il controller) ed un altro (sul dispositivo in rete) che lascia che i due programmi comunichino, con l'intento di permettere al controller di programmare le tabelle di forwarding per il data plane del dispositivo su cui opera.

Come facilmente intuibile, in un'architettura costruita per permettere la programmabilità della rete, le capacità della SBI e delle API ci dicono cosa un'architettura può o non può fare. Per esempio alcuni controllori possono supportare una sola SBI per uno specifico scopo, mentre altre possono supportarne di più e quindi hanno la possibilità di scegliere quale SBI utilizzare. Di seguito ne elenco alcune tra le più utilizzate:

- OpenFlow
- OpFlex
- CLI (Telnet/SSH) e SNMP

STUDENTSUPSI

3.2.3 Northbound Interface

Come detto nella sezione precedente in riferimento alla figura 3.5, il controller può aggiungere ingressi alle tabelle di forwarding dei dispositivi connessi ad esso. Quello che andremo ora ad analizzare è il *come* queste tabelle vengono aggiornate. Come fa il controller a sapere che persorsi aggiungere? Come sceglie questi percorsi? Che tipo di informazioni abbiamo bisogno di raccogliere prima che il controller possa modificare una tabella di routing? Per rispondere a queste domande è fondamentale conoscere:

- La lista di tutti i nodi in rete
- La capacità di ogni dispositivo
- Le interfacce/porte di ognuno
- Lo stato corrente di ogni porta
- La topologia: quali dispositivi si connettono con quali altri, su quale interfaccia
- Configurazione dei dispositivi: indirizzi IP, VLANs, relazioni

In un modello di controllo centralizzato, un controller svolge molto del lavoro richiesto dal control plane. Raccoglie qualsiasi sorta di informazione utile della rete, come gli elementi listati precedentemente. Al suo interno tutte queste informazioni sono memorizzate in un database.

La *Northbound Interface* (NBI) si occupa di fornire accesso al controller da parte di software che sono considerati di un livello gerarchico più alto. Le informazioni necessarie possono quindi essere estratte dai processi che ne richiedono l'accesso.

Per vedere dove risiede la NBI, pensiamo alla struttura del controller: stiamo parlando di un software che corre su un server (sia esso una VM o un server fisico). Sullo stesso server possiamo avere anche un'applicazione come il controller che utilizza una API in modo tale che i due possano comunicare.

La figura 3.6 illustra il concetto: il grande riquadro nella figura rappresenta il sistema dove risiede il software del controllore. Questo particolare controller (che nell'esempio è scritto in Java) possiede una API nativa. Qualsiasi persona può scrivere una nuova API e sostituire quella di default con la nuova. Utilizzando questa API per scambiare dati con il controller, l'applicazione può imparare informazioni sulla rete. L'applicazione può inoltre programmare il flusso in rete, ovvero chiedere al controller di aggiungere un nuovo percorso ad una tabella di forwarding.

STUDENTSUPSI

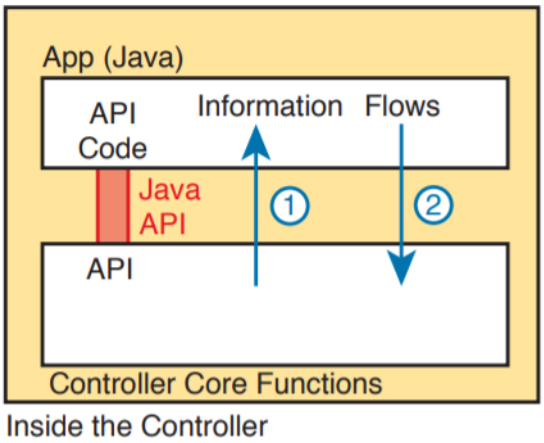


Figura 3.6: Java API: L'Applicazione Java Comunica con il Controller

Prima di terminare il capitolo spiego brevemente il concetto di REST APIs, ovvero quelle applicazioni che permettono di trasferire dati tra hosts differenti. *REpresentational State Transfer* (REST) è un tipo di APIs che permettono la comunicazione tramite messaggi HTTP tra applicazioni che si trovano su sistemi diversi. In figura 3.6 questo bisogno non c'è, dato che l'applicazione e il controller fanno parte dello stesso sistema.

La figura 3.7 illustra il concetto. L'applicazione vuole ottenere informazioni su una struttura di dati che ha bisogno di memorizzare e successivamente processare.

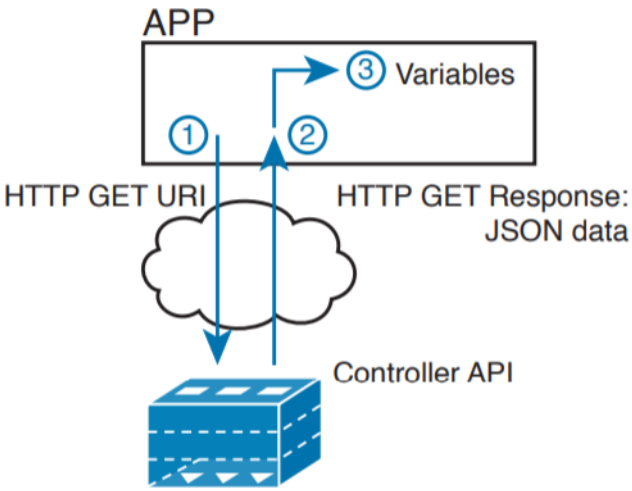


Figura 3.7: Esempio di Processo GET Utilizzando una REST API

Allo Step 1 l'applicazione richiede tramite chiamata HTTP (HTTP GET Request) una particolare informazione di cui ha bisogno (per esempio una struttura dati). Lo Step 2 rappresenta

STUDENTSUPSI

la risposta del controller: la struttura dati viene inviata all'applicazione che l'aveva precedentemente richiesta. Seguendo l'esempio di figura 3.6, un codice scritto in JavaScript fornisce una risposta di tipo JavaScript Object Notation (JSON).

3.3 Sommario dell'Architettura SDN

SDN e la programmabilità di rete introduce un nuovo modo di costruire le reti moderne. I dispositi di rete continuano ad esistere, inoltrano i frame come hanno sempre fatto, ma le funzioni del control plane e la sua locazione possono cambiare drammaticamente. Il controller centralizzato rappresenta il punto focale, di modo che almeno una tra le tante funzioni del control plane si spostino da un modello distribuito ad un modello centralizzato.

Tuttavia, il mondo della programmabilità di rete ed SDN include un vasto elenco di possibilità e soluzioni. Alcune opzioni inseriscono la maggior parte delle funzioni del control plane nel controller, mentre altre ne inseriscono solo un paio. La prossima sezione introduce il lettore alla soluzione più spesso utilizzata: Open SDN e OpenFlow.

3.4 Esempio di SDN - OpenFlow

Questa sezione vuole introdurre il lettore alla soluzione SDN attualmente più utilizzata, illustrandone le peculiarità e le caratteristiche.

3.4.1 Open SDN e OpenFlow

La forma più comune di SDN è implementata da Open Networking Foundation (ONF) ed è conosciuta come *Open SDN*. ONF agisce come promotore della tecnologia SDN sul mercato, ed è sostenuto sia da normali utenti che da venditori esperti. Parte del loro lavoro implica la realizzazione di protocolli, SBIs, NBIs e qualsiasi strumento che possa aiutare le persone ad ampliare la propria visione riguardo SDN.

Il modello ONF di SDN implementa OpenFlow come SBI. Parte di OpenFlow definisce un protocollo IP-based usato tra il controller e i dispositivi di rete. Inoltre, OpenFlow definisce un'idea standard riguardo le capacità degli switch basandosi su ASICs e TCAMs utilizzati nei dispositivi odierni (questa idea standardizzata del lavoro di uno switch prende il nome di *switch abstraction*). Uno switch OpenFlow può agire come L2 Switch, L3 Switch oppure in modi differenti e con grande flessibilità rispetto ad uno tradizionale.

STUDENTSUPSI

Il modello Open SDN centralizza la maggior parte delle funzioni del control plane, con il controllo della rete effettuato dal controller oltre a qualsiasi applicazione che utilizzi NBIs. Facciamo di nuovo riferimento alla figura 2.5, il cui scopo era di mostrare i dispositivi di rete senza un control plane: in questo esempio la maggior parte delle funzioni del modello ONF di SDN sono centralizzate. Le applicazioni posso sfruttare qualsiasi APIs (NBIs) supportate dalla piattaforma del controller. Tuttavia, viene richiesto il protocollo OpenFlow a livello SBI, e gli switches devono necessariamente supportare OpenFlow.

Dato che il modello Open SDN sviluppato da ONF presuppone che il controller sia strettamente legato con OpenFlow SBI, è facile intuire come il controllore giochi un ruolo chiave nella rete. Le prossime pagine forniscono più dettagli circa uno dei più comuni controllers attualmente in commercio: OpenDayLight Controller.

3.4.1.1 OpenDayLight Controller

Quando SDN era ancora in fase sperimentale, tante aziende hanno provato a sviluppare un proprio controller che permettesse di assolvere alle varie funzioni che la rete potesse richiedere. Dato che la maggior parte di queste funzioni erano le medesime nonostante le aziende fossero diverse, si è pensato che rendendo il software Open Source, tutti ne avrebbero tratto beneficio. Nasce quindi *OpenDayLIght Controller*, un progetto di Linux Foundation che ad oggi è sostenuto economicamente da moltissime aziende.

La figura 3.8 mostra una versione generalizzata dell'architettura ODL. Si noti in particolare le varietà di SBIs disponibili: OpenFlow, NetConf, PCEP, BGP-LS e OVSDB. Avendo così tanti sostenitori, ODL include tutte queste varianti di protocolli SBI, non solo OpenFlow.

STUDENTSUPSI

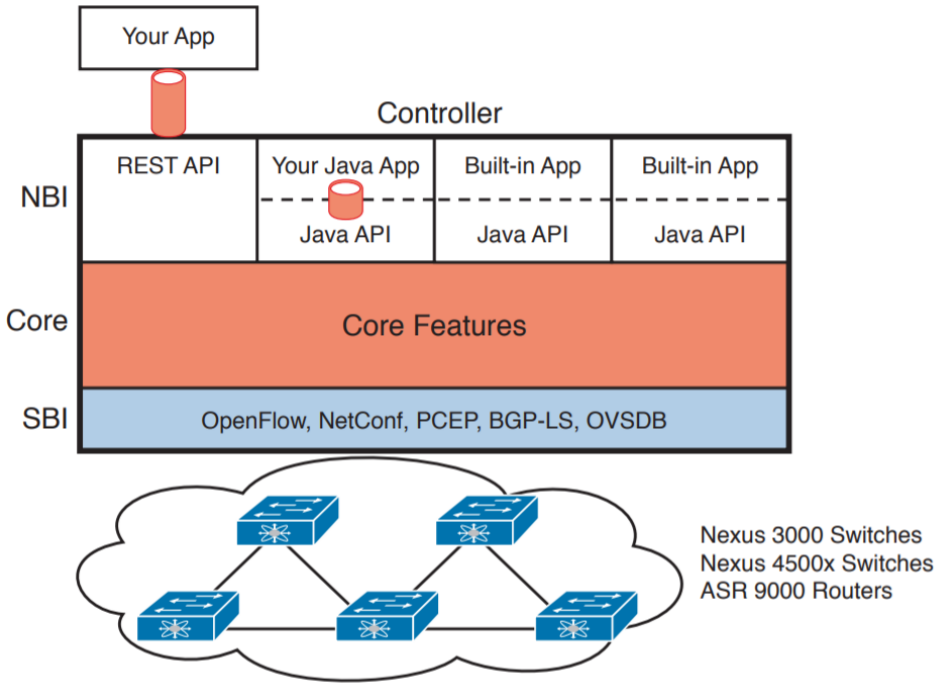


Figura 3.8: Architettura della NBI, Interni del Controller e SBI in Relazione ai Network Devices

ODL svolge tante funzioni con molte SBIs e possiede altrettante caratteristiche chiave. Per informazioni aggiuntive riguardo ODL, la lista dei controllori che fanno parte di questa famiglia e qualsiasi altra curiosità, rimando al sito web del progetto https://www.opendaylight.org.

Ora che sia la virtalizzazione server che la tecnologia SDN sono state illustrate nel dettaglio, possiamo vedere come implementare questa soluzione ad un caso pratico. Il prossimo capitolo introduce il software utilizzato per la realizzazione delle topologie di rete all'interno del server di laboratorio.

STUDENTSUPSI

Capitolo 4

GNS3

La seguente sezione vuole analizzare il lavoro svolto per emulare il concetto di SDN tramite il software *Graphical Network Simulator 3*, altrimenti conosciuto come *GNS3*. Questo software permette di combinare dispositivi reali e virtuali in modo da creare topologie di rete complesse altrimenti difficili da realizzare in laboratorio con dispositivi fisici.

Figura 4.1: Logo di GNS3

Ogni dispositivo di rete che si vuole analizzare viene emulato dal software proprietario a Command Line Interface (CLI).

GNS3 è un software open source, e per motivi di copyright le immagini dei dispositivi che si vogliono utilizzare nelle topologie di rete devono essere acquistate a parte. In una prima fase ho scelto di utilizzare la soluzione open source già analizzata in precedenza: Arista vEOS. Nella seconda parte del documento mi focalizzerò di più verso routers Cisco Systems, in quanto la documentazione online di questi ultimi è molto più vasta di quella di Arista Networks.

STUDENTSUPSI

Per iniziare ho configurato il modello leaf-spine come rappresentato in figura:

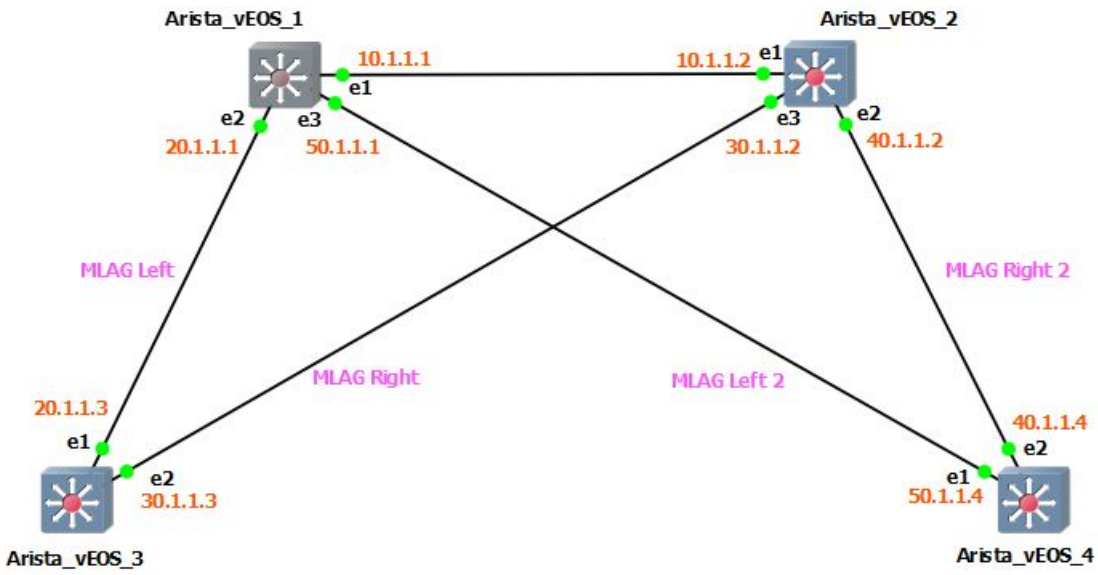


Figura 4.2: Configurazione Leaf Spine - GNS3

Aggiungendo un terminale Linux è possibile connettersi via Telnet agli switches, e da qui impartire comandi da remoto. Questo rappresenta il processo chiave con cui si automatizza una rete: i terminali figurano nel data center mentre l'utente si trova fisicamente in un altro posto: non abbiamo bisogno di essere presenti davanti al computer perché possiamo connetterci via client da remoto.

Una volta stabilita una connessione in remoto verso un dispositivo abbiamo pieno accesso a questo: possiamo impartire comandi di configurazione e revisionare lo stato dello switch.

Essendo Arista EOS un sistema operativo programmabile, è possibile creare degli scripts che automatizzino la configurazione. Per iniziare, esaminiamo queste linee di codice:

```
# Entering global configuration mode
connection.write('configure terminal \n')
time.sleep(1)

# Configuring 5 VLANs
connection.write('vlan 2 \n')
connection.write('name VLAN_2 \n')
connection.write('vlan 3 \n')
connection.write('name VLAN_3 \n')
connection.write('vlan 4 \n')
connection.write('name VLAN_4 \n')
connection.write('vlan 5 \n')
connection.write('name VLAN_5 \n')
```

STUDENTSUPSI

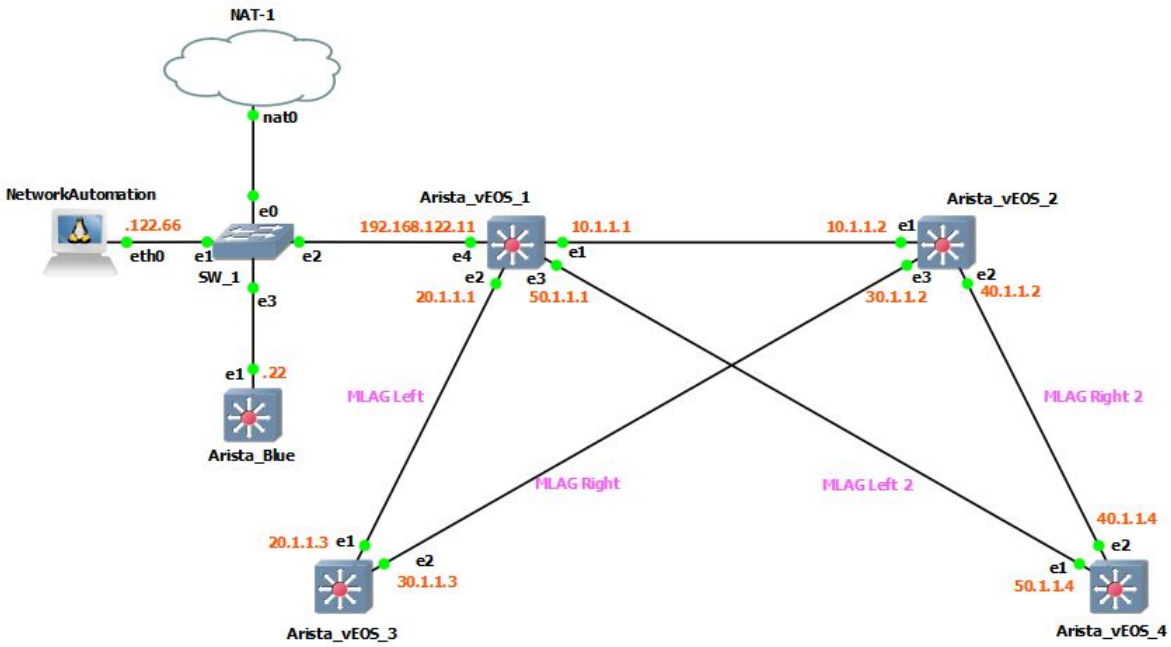


Figura 4.3: Configurazione Leaf Spine - GNS3 con Terminale Linux

```
connection.write('vlan 6 \n')
connection.write('name VLAN_6 \n')
connection.write('vlan 7 \n')
connection.write('name VLAN_7 \n')
connection.write('vlan 8 \n')
connection.write('name VLAN_8 \n')
connection.write('vlan 9 \n')
connection.write('name VLAN_9 \n')
connection.write('vlan 10 \n')
connection.write('name VLAN_10 \n')
```

Questo semplice script configura 10 VLANs nel nostro switch, assegnando ad ognuna un nome. Nel caso volessimo crearne di più, questo metodo non è più efficace, e si preferisce ricorrere alle iterazioni. La figura sottostante configura cento (!) VLANs, e attribuisce un nome ad ognuna:

```
for n in range (2, 102)
connection.write('vlan ' + str(n) + '\n')
connection.write('name VLAN_' + str(n) + '\n')
```

Si noti la brevità del codice: in solo tre righe abbiamo impostato gli stessi comandi quivalenti a quelle che sarebbero state duecento righe di configurazione nel caso standard (CLI da terminale).

STUDENTSUPSI

4.1 Netmiko

Netmiko è una libreria Python open-source che permette di semplificare le connessioni SSH verso i dispositivi di rete. Netmiko ha quattro funzioni principali:

- Stabilire una connessione SSH verso il dispositivo
- Semplificare l'esecuzione del comando *show* e il relativo output a schermo
- Semplificare l'esecuzione dei comandi di configurazione del dispositivo
- Sviluppare le precedenti su diverse piattaforme attualmente in commercio

Il primo script che ho creato permette di stabilire una connessione remota e sicura con un dispositivo nella nostra rete. Chiedendo in input l'indirizzo IP dello switch che vogliamo configurare ci possiamo direttamente connetere a questo e applicare la configurazione desiderata.

```
from netmiko import ConnectHandler

arista_vEOS = {
'device_type': 'arista_eos',
'ip': deviceIP,
'username': 'davide',
'password': 'tesi'
}

print('Insert the IP of the Arista vEOS device:')
deviceIP = input()

print('----- Trying to establish a SSH connection with ' + arista_vEOS['
    ip'] + '... -----')
net_connect = ConnectHandler(**arista_vEOS)
print('SSH connection successfully created!')
output = net_connect.send_command('sh ip int brief')
print('Show IP Interfaces: ' + output)

config_commands = [
'int loop 0',
'ip address 1.1.1.1 255.255.255.0'
]
net_connect.enable()
net_connect.config_mode()
output = net_connect.send_config_set(config_commands)
print('Configuration commands:' + output)
```

STUDENTSUPSI

```
27
28 for n in range (2,11):
29 print ("Creating VLAN " + str(n))
30 config_commands = ['vlan ' + str(n), 'name VLAN_' + str(n)]
31 output = net_connect.send_config_set(config_commands)
32 print (output)
33
34 net_connect.disconnect()
```

L'esecuzione di questo script mostra le interfacce dello switch (*show ip interfaces brief*), configura un'interfaccia di loopback (chiamata *loopback 0*) a cui assegna il valore 1.1.1.1/24 e crea 10 VLANs con relativo nome. Possiamo connetterci a qualsiasi dispositivo ed eseguire questo codice e il risultato sarà identico per tutti! Perché allora non creare uno script che possa programmare TUTTI i nodi della nostra rete?

Il risultato è quello mostrato qui:

```
1 from netmiko import ConnectHandler
2
3 loopback_counter = 1
4 deviceIP = ''
5
6 arista_vEOS_1 = {
7 'device_type': 'arista_eos',
8 'ip': '192.168.122.11',
9 'username': 'davide',
10 'password': 'tesi'
11 }
12
13 arista_vEOS_2 = {
14 'device_type': 'arista_eos',
15 'ip': '10.1.1.2',
16 'username': 'davide',
17 'password': 'tesi'
18 }
19
20 arista_vEOS_3 = {
21 'device_type': 'arista_eos',
22 'ip': '20.1.1.3',
23 'username': 'davide',
24 'password': 'tesi'
25 }
26
27 arista_vEOS_4 = {
28 'device_type': 'arista_eos',
29 'ip': '40.1.1.4',
30 'username': 'davide',
```

STUDENTSUPSI

```
31 'password': 'tesi'
32 }
33
34 # List - All my devices are grouped here
35 myDevices = [
36 arista_vEOS_1,
37 arista_vEOS_2,
38 arista_vEOS_3,
39 arista_vEOS_4
40 ]
41
42 for deviceName in myDevices:
43 print('----- Trying to establish a SSH connection with ' + deviceName['ip
    '] + '... -----')
44 net_connect = ConnectHandler(**deviceName)
45 print('SSH connection successfully created!')
46 output = net_connect.send_command('sh ip int brief')
47 print('Show IP Interfaces:')
48 print(output)
49
50 config_commands = [
51 'int loop 0',
52 'ip address ' + str(loopback_counter) + '.' +  str(loopback_counter) + '.
    ' + str(loopback_counter) + '.' + str(loopback_counter) + '
    255.255.255.0'
53 ]
54
55 loopback_counter = loopback_counter + 1
56 net_connect.enable()
57 net_connect.config_mode()
58 output = net_connect.send_config_set(config_commands)
59 print('Configuration commands:')
60 print(output)
61
62 for n in range (2,12):
63 print ("Creating VLAN " + str(n))
64 config_commands = ['vlan ' + str(n), 'name VLAN_' + str(n)]
65 output = net_connect.send_config_set(config_commands)
66 print (output)
67
68 net_connect.disconnect()
```

In meno di 70 righe di codice ho programmato tutti gli switches in configurazione leaf-spine di figura 4.3. Lo script precedente, se salvato all'interno del computer Linux rappresenta la configurazione di base (backup configuration) in caso di errori da parte dell'utente. Questo file è la soluzione migliore nell'eventualità che un ingegnere sbagli un comando e comprometta il funzionamento di uno switch. Lo script viene lanciato e in pochi secondi la rete

STUDENTSUPSI

torna alla configurazione di default, evitando di perdere tempo a cercare il problema causato dall'errore umano.

4.2 NAPALM

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor Support) è una libreria Python che implementa un set di funzioni per interagire con differenti Sistemi Operativi dei dispositivi di rete, utilizzando una API universale.

NAPALM supporta diversi metodi di connessione ai dispositivi, permette di manipolare le configurazioni o recuperare i dati persi.

Figura 4.4: NAPALM -Logo

Tramite le funzioni integrate di NAPALM possiamo ottenere informazioni dai nostri dispositivi, salvarli in un file di backup e visualizzarli in modo ordinato tramite JSON (Java Script Object Notation).

Per la lista delle funzioni disponibili rimando al link: `https://napalm.readthedocs.io/en/latest/support/index.html`

Vogliamo ora definire una lista in cui vengono definiti dei parametri che i pacchetti devono rispettare per essere processati dai nostri switches. Questo elenco è definito ACL, acronimo di *Access Control List*.

L'ACL che ho creato è definita di seguito:

STUDENTSUPSI

```
access-list 100 permit icmp any any
access-list 100 permit tcp any any eq domain
access-list 100 permit tcp any any eq www
access-list 100 permit tcp any any eq 443
```

Al suo interno contiene quattro regole che permettono di filtrare i pacchetti se questi non sono richieste di tipo:

- PING (icmp any any)
- DNS (tcp any any eq domain)
- WWW (tcp any any eq www)
- HTTPS (tcp any any eq 443)

In tutti i casi in cui i pacchetti non rispecchiano queste caratteristiche, essi vengono eliminati (quindi non processati).

Vogliamo ora implementare questa lista su tutti i nostri switches. Ovviamente, non andremo a configurare ogni switch singolarmente! Sfruttando i concetti che ho esaminato al capitolo precedente possiamo programmare tutta la nostra rete sviluppando uno script dedicato.

4.2.1 Esempio di Utilizzo: Configurazione di ACLs

Chiamiamo *ACL1* la lista degli accessi definita precedentemente. Per risparmiare tempo importante in fase di configurazione, introduciamo un controllo sulle ACLs già presenti nel dispositivo:

```
differences = ios_l3.compare_config()
if len(differences) > 0:
print(differences)
ios_l3.commit_config()
print('Device successfully configured!')
else:
print('The device already has this configuration.')
ios_l3.discard_config()
ios_l3.close()
```

Listing 4.1: Aggiunta di ACL su un nodo

In questo modo se la configurazione che vogliamo impostare è già presente, evitiamo di riconfigurare il dispositivo con gli stessi dati! Lanciando lo script su un nodo senza nessuna impostazione predefinita, il risultato è il seguente:

STUDENTSUPSI

```
root@Network_Automation:~# python3 napalm2.py
Accessing the device...
+access-list 100 permit icmp any any
+access-list 100 permit tcp any any eq domain
+access-list 100 permit tcp any any eq www
+access-list 100 permit tcp any any eq 443
Device successfully configured!
```

Il simbolo '+' indica che quel determinato parametro è stato aggiunto. Spostandoci ora sul nodo in questione, notiamo che la CLI ci indica che il dispositivo è stato configurato da me tramite 192.168.122.248 (IP del client Network_Automation):

```
*Jul 15 12:01:29.220: %GRUB-5-CONFIG_WRITING: GRUB configuration is being
    updated on disk. Please wait...
*Jul 15 12:01:30.016: %GRUB-5-CONFIG_WRITTEN: GRUB configuration was
    written to disk successfully.
*Jul 15 12:01:33.403: %SYS-5-CONFIG_I: Configured from console by davide
    on vty0 (192.168.122.248)
Leaf-1(config)#
```

Listing 4.2: Output di un Nodo - Leaf-1

All'interno dello switch, con il comando *show access-lists* vediamo che la lista degli accessi è stata popolata.

```
Leaf-1#sh access-lists
Extended IP access list 100
10 permit icmp any any
20 permit tcp any any eq domain
30 permit tcp any any eq www
40 permit tcp any any eq 443
```

Listing 4.3: Leaf-1 - Show Access-Lists

Cosa succede nel caso volessimo rilanciare lo script su questo nodo (che presenta la stessa ACL che vorremmo configurare)? Come facilmente intuibile:

```
Accessing the device...
The device already has this configuration.
root@Network_Automation:~#
```

Listing 4.4: Configurare un Dispositivo Con Una ACL Già Presente

Non bisogna pensare in piccolo: in una rete costituita da migliaia di dispositivi, risparmiare un secondo ogni volta che l'ACL è la stessa permette di rendere la configurazione nettamente più veloce ed i costi di calcolo inferiori.

STUDENTSUPSI

Ora che abbiamo visto come impostare le ACL su uno switch, per settare i controlli su tutti gli altri dispositivi non dobbiamo fare altro che inserire un loop:

```
import json
from napalm import get_network_driver

driver = get_network_driver('ios')
ios_leaf1 = driver('192.168.122.40', 'davide', 'tesi')
ios_leaf2 = driver('192.168.122.50', 'davide', 'tesi')
ios_leaf3 = driver('192.168.122.60', 'davide', 'tesi')

leafList = [ios_leaf1, ios_leaf2, ios_leaf3]

for devices in leafList:
devices.open()
print('Accessing the device' + str(devices))
devices.load_merge_candidate(filename = 'ACL1.cfg')
differences = devices.compare_config()

if len(differences) > 0:
print(differences)
devices.commit_config()
print('Device successfully configured!')
else:
print('The device already has this configuration.')
devices.discard_config()
devices.close()
```

Listing 4.5: Configurare Più Dispositivi Attraverso Uno Script Unico

A titolo di esempio ho scelto di applicare la configurazione ACL ai tre switches a stretto contatto con i clients (Leaf-1, Leaf-2 e Leaf-3). Si può applicare la stessa anche per gli altri al livello soprastante, oppure optare una configurazione diversa.

4.3 Ottimizzazione Del Codice

Una volta automatizzata la rete con il codice esaminato nei capitoli precedenti, passo ad un lavoro di ottimizzazione. Oltre ai loops ed agli if-statements, due concetti offrono migliorie ad alto livello:

- Criptare le passwords dei dispositivi
- Multithreading

STUDENTSUPSI

Nelle sezioni che seguono esamino entrambi nel dettaglio.

4.3.1 Criptare Le Passwords Dei Dispositivi

Se vogliamo automatizzare il processo di configurazione dei dispositivi, dobbiamo evitare di inserire le passwords manualmente. Diventa quindi necessario presalvare queste in un file che possa essere letto dal nostro script. Ovviamente, salvare le password in chiaro non è sicuro. Per aggiungere un livello di sicurezza supplementare andrò ora a criptare queste password e le salverò in un file indecifrabile senza chiave, di modo che le credenziali siano protette.

Per fare ciò utilizzo una libreria Python di nome *SimpleCrypt*, che permette sia la cifratura che la decifratura delle parti di testo che ci interessa nascondere.

Il file delle password in chiaro l'ho chiamato leafCredentials, ed è costituito dagli indirizzi IP degli switches a livello client, oltre all'username *davide* associato alla password *tesi*:

```
192.168.122.40, davide, tesi
192.168.122.50, davide, tesi
192.168.122.60, davide, tesi
```

Listing 4.6: Password in Chiaro - leafCredentials

```
192.168.122.40, cisco-ios, Leaf-1
192.168.122.50, cisco-ios, Leaf-2
192.168.122.60, cisco-ios, Leaf-3
```

Listing 4.7: Indirizzo IP, Tipo di Device e Hostname

Con questo script che ho scritto genero un file che contiene al suo interno le password criptate:

```
from simplecrypt import encrypt, decrypt
from pprint import pprint
import csv
import json

#---- Read in pertinent information from user
dc_in_filename = input('\nInput Credentials File (leafCredentials): ') or
    'leafCredentials'
key            = input('Encryption key (encryptionKey): ') or '
    encryptionKey'

#---- Read in the device credentials from CSV file into list of device
    credentials
with open( dc_in_filename, 'r' ) as dc_in:
```


STUDENTSUPSI

```
12 device_creds_reader = csv.reader( dc_in )
13 device_creds_list = [device for device in device_creds_reader]
14
15 print ('\n----- Device Credentials ------')
16 pprint( device_creds_list )
17
18 #---- The credentials are encrypted into a file whose name is chosen by
    me
19 encrypted_dc_out_filename = input('\nOutput encrypted filename (
    leafCredentialsEncrypted): ') or 'leafCredentialsEncrypted'
20
21 with open( encrypted_dc_out_filename, 'wb' ) as dc_out:
22 dc_out.write( encrypt( key, json.dumps( device_creds_list ) ) )
23
24 print ("----- File Successfully Encrypted -----")
25
26 print ('\n... Getting credentials ...\n')
27 with open( encrypted_dc_out_filename, 'rb') as device_creds_file:
28 device_creds_json = decrypt( key, device_creds_file.read() )
29
30 device_creds_list = json.loads( device_creds_json.decode('utf-8'))
31 pprint( device_creds_list )
32
33 print ('\n----- Converting To Dictionary -------')
34
35 #---- Convert to dictionary of lists using dictionary comprehension
36 device_creds = { dev[0]:dev for dev in device_creds_list }
37 pprint( device_creds )
38
39 print ('---------- Done! ----------\n')
```

Listing 4.8: Creazione di un File Contenente le Password Criptate

Abbiamo ottenuto un file (nome di default: *leafCredentialsEncrypted*) che contiene le nostre credenziali criptate con la chiave *encryptionKey*. Nella prossima sezione vediamo come sfruttare questo file assieme al concetto di multithreading.

4.3.2 Multithreading

Il concetto di *multithreading* (altrimenti conosciuto come *parallelizzazione*) permette di sfruttare meglio le risorse hardware a nostra disposizione per diminuire i tempi di calcolo. Nello specifico, come evidenziato nel capitolo 2.2.1 di questa tesi, le CPU moderne sono caratterizzate dal possedere più cores, e spesso ogni core supporta due threads. Ogni thread rappresenta un processo a parte, che può essere eseguito in parallelo.

STUDENTSUPSI

La programmazione parallela (a dispetto di quella sequenziale) permette di velocizzare notevolmente i tempi di esecuzione di uno script, perché più cores sono coinvolti nell'esecuzione del programma.

```
from simplecrypt import encrypt, decrypt
from pprint import pprint
from netmiko import ConnectHandler
import json
from time import time

import threading

#-----------------------
def read_devices( devices_filename ):

devices = {}  # create our dictionary for storing devices and their info

with open( devices_filename ) as devices_file:

for device_line in devices_file:

device_info = device_line.strip().split(',')  #extract device info from line

device = {'ipaddr': device_info[0],
'type':   device_info[1],
'name':   device_info[2]}  # create dictionary of device objects ...

devices[device['ipaddr']] = device  # store our device in the devices dictionary
# note the key for devices dictionary entries is ipaddr

print ('\n----- devices -------------------------')
pprint( devices )

return devices

#-----------------------
def read_device_creds( device_creds_filename, key ):

print ('\n... getting credentials ...\n')
with open( device_creds_filename, 'rb') as device_creds_file:
device_creds_json = decrypt( key, device_creds_file.read() )

device_creds_list = json.loads( device_creds_json.decode('utf-8'))

print ('\n----- device_creds ----------------------')

```

STUDENTSUPSI

```
# convert to dictionary of lists using dictionary comprehension
device_creds = { dev[0]:dev for dev in device_creds_list }
pprint( device_creds )

return device_creds

#-----------------------
def config_worker( device, creds ):

#---- Connect to the device ----
if   device['type'] == 'junos-srx': device_type = 'juniper'
elif device['type'] == 'cisco-ios': device_type = 'cisco_ios'
elif device['type'] == 'cisco-xr':  device_type = 'cisco_xr'
else:                               device_type = 'cisco_ios'    # attempt Cisco IOS as default

print ('---- Connecting to device {0}, username={1}, password={2}'.format( device['ipaddr'],
creds[1], creds[2] ))

#---- Connect to the device
session = ConnectHandler( device_type=device_type, ip=device['ipaddr'],
username=creds[1], password=creds[2] )
#session = ConnectHandler( device_type=device_type, ip='172.16.0.1',  # Faking out IP address for now
#                                                    username=creds[1], password=creds[2] )

if device_type == 'juniper':
#---- Use CLI command to get configuration data from device
print ('---- Getting configuration from device')
session.send_command('configure terminal')
config_data = session.send_command('show configuration')

if device_type == 'cisco_ios':
#---- Use CLI command to get configuration data from device
print ('---- Getting configuration from device')
config_data = session.send_command('show run')


if device_type == 'cisco_xr':
#---- Use CLI command to get configuration data from device
print ('---- Getting configuration from device')
config_data = session.send_command('show configuration running-config')

#---- Write out configuration information to file
config_filename = 'config-' + device['ipaddr']  # Important - create unique configuration file name
```

STUDENTSUPSI

```
86
87 print ('---- Writing configuration: ', config_filename)
88 with open( config_filename , 'w' ) as config_out:  config_out.write(
       config_data )
89
90 session.disconnect()
91
92 return
93
94 #=====================================#
95 # ---- Main: Get Configuration ----  #
96 #=====================================#
97
98 devices = read_devices( 'spineList' )
99 creds   = read_device_creds( 'leafCredentialsEncrypted', 'encryptionKey'
       )
100
101 starting_time = time()
102
103 config_threads_list = []
104 for ipaddr,device in devices.items():
105
106 print ('Creating thread for: ', device)
107 config_threads_list.append( threading.Thread( target=config_worker, args
       =(device, creds[ipaddr] ) ) )
108
109 print ('\n---- Begin get config threading ----\n')
110 for config_thread in config_threads_list:
111 config_thread.start()
112
113 for config_thread in config_threads_list:
114 config_thread.join()
115
116 print ('\n----- End get config threading, elapsed time=', time() -
       starting_time -----)
```

Listing 4.9: Programmazione Parallela - Multithreading

Come possiamo notare nel main viene sfruttato il file *leafCredentialsEncrypted*, a cui viene fornita la chiave per decifrare il codice criptato.

Ho inserito i tempi di calcolo per dare una idea al lettore della grande differenza tra programmazione sequenziale e parallela:

Sequenziale:

```
1 ----- End get config sequential, elapsed time= 25.407458066940308 -----
```

Listing 4.10: Tempo di Esecuzione - Programmazione Sequenziale

STUDENTSUPSI

Parallela:

```
---- End get config threading, elapsed time= 8.784448146820068 -----
```

Listing 4.11: Tempo di Esecuzione - Programmazione Parallela

Solamente con tre dispositivi, notiamo che parallelizzare i calcoli ha permesso di ridurre il tempo di esecuzione del codice del 68%!

4.4 OpenDayLight e Mininet

Come già illustrato nel Capitolo 2, OpenDayLight rappresenta una soluzione open-source per implementare un controller di rete. Se associato all'emulatore di rete *Mininet*, OpenDayLight permette di creare topologie di rete gestibili e visualizzabili graficamente tramite web browser.

Ho scelto di gestire sia ODL che Mininet tramite VirtualBox sul mio computer. Affinchè le due VMs possano comunicare fra loro, è necessario introdurre due interfacce di rete aggiuntive (una per host) di modo tale che ODL e Mininet possano scambiarsi informazioni. Nello specifico, ho scelto di assegnare a Mininet l'indirizzo IP 192.168.56.4/24, e ad OpenDayLight 192.168.56.5/24.

STUDENTSUPSI

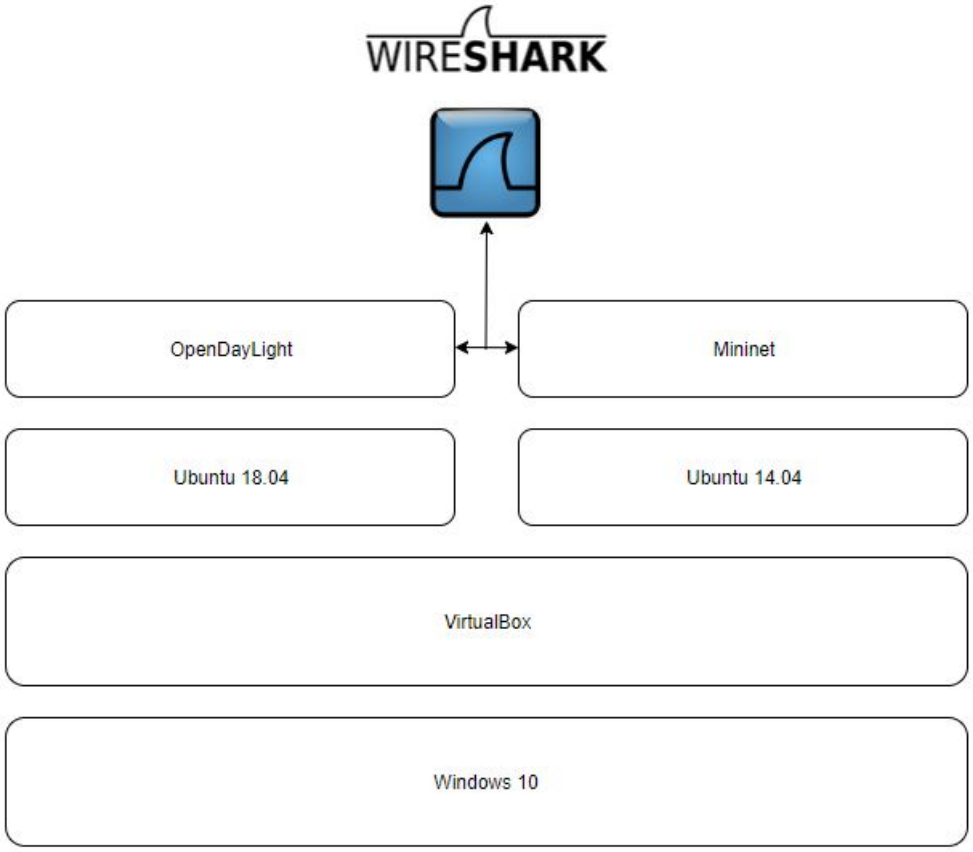


Figura 4.5: Trasmissione di Pacchetti tra ODL e Mininet

Il protocollo con cui le due VMs comunicano fra loro è *OpenFlow* (definito dalle porte 6633 e 6653).

Dopo aver creato una topologia di rete per effettuare qualche semplice test, sfruttando Wireshark ho potuto vedere come le due VMs si trasmettano correttamente i messaggi:

Come si nota sotto la voce *Source*, OpenDayLight (192.168.56.5) invia richieste a Mininet (*Destination:* 192.168.56.4) circa il traffico di dati, statistiche dei pacchetti in coda, tabelle, e così via. Ad ogni richiesta di informazioni vi è sempre una risposta da parte di Mininet, evidenziando come il sistema stia lavorando nel modo corretto.

STUDENTSUPSI

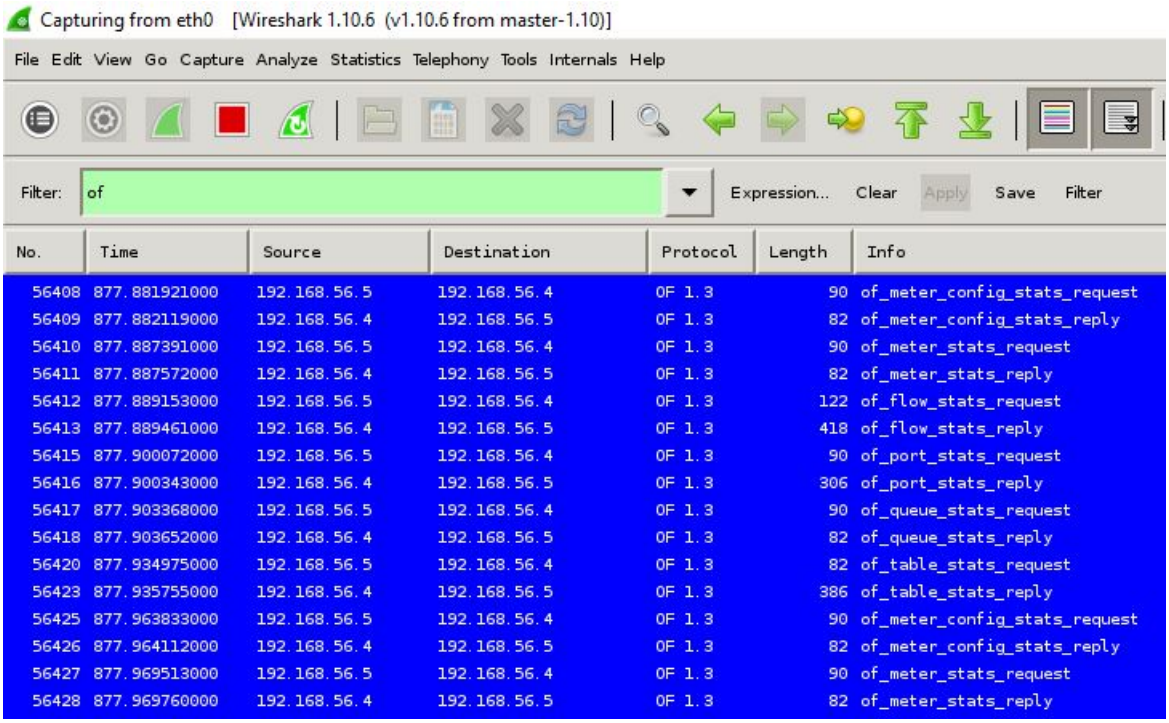


Figura 4.6: Trasmissione di Pacchetti OpenFlow tra ODL e Mininet

STUDENTSUPSI

Capitolo 5

MPLS

MPLS, acronimo di *Multi Protocol Label Switching*, è una tecnologia moderna tipicamente utilizzata dai Service Providers (SPs) per velocizzare la trasmissione dei pacchetti all'interno del core della struttura. Visti i notevoli vantaggi che MPLS offre, recentemente anche le più grandi aziende si stanno spostando verso l'utlizzo di questa tecnologia.

MPLS viene implementato aggiungendo una etichetta (*label*) di 32 bit tra il frame ethernet e l'header IP. Per questa ragione, alla tecnologia MPLS viene spesso associato il termine "Layer 2.5", ovvero una zona che rimane tra il livello di collegamento (L2 - Switching) e il livello di rete (L3 - Routing).

La seguente figura mostra una tipica implementazione di MPLS da parte di un ISP:

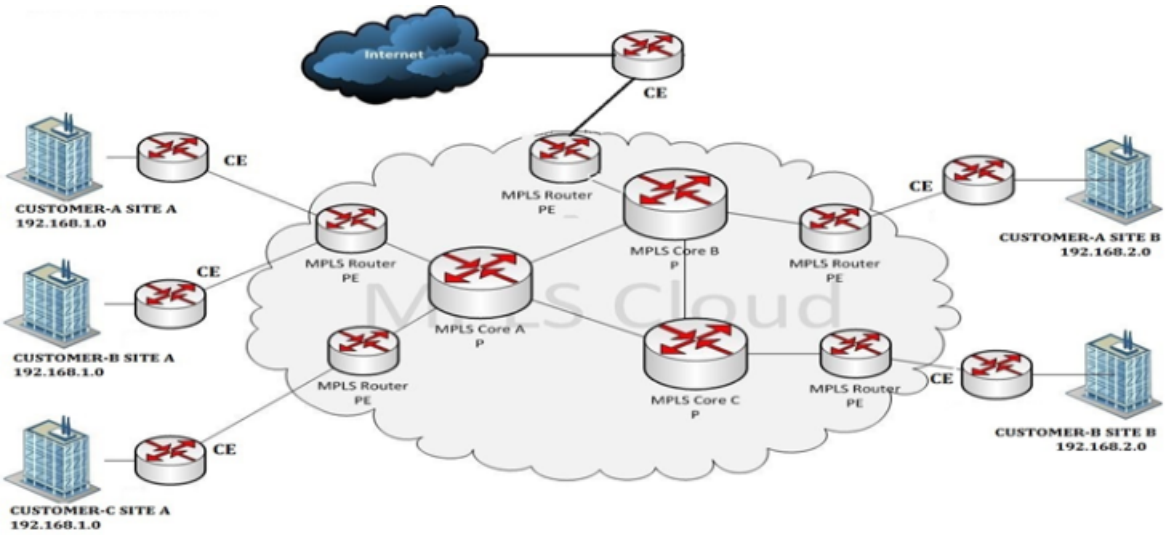


Figura 5.1: IPv4 over MPLS in un ISP

STUDENTSUPSI

Si notano tre tipi di routers differenti:

- CE: Customer Equipment
- PE: Provider Edge
- CP: Core Provider

CE: I router caratterizzati dalla sigla *CE* sono tutti quelli che rappresentano l'ultimo stadio nella fase di routing, prima di passare al livello inferiore (L2) interno all'azienda.

PE: Viene definito *Provider Edge* un router che fornisce connessioni tra un Core Provider ed un Customer Equipment. La tipica posizione esterna che questo dispositivo assume viene appunto definita "edge", ovvero "sul bordo".

CP: I routers più interni sono definiti *Core Providers*, e sovente presentano connessioni all'interno del cloud MPLS con altri CPs.

In riferimento alla figura 6.1, gli unici routers che lavorano in MPLS sono quelli rappresentati all'interno della nuvola denominata *MPLS Cloud*. Quando un pacchetto IPv4 in uscita da uno dei CE entra nel cloud MPLS, ad esso viene assegnata una etichetta (processo denominato *push*). Attraverso il suo percorso, i router MPLS effettueranno una operazione di scambio etichette (*label swap*). L'ultimo PE effettuerà un *pop* dell'etichetta, riportando di fatto il pacchetto ad essere puramente IPv4.

Le etichette MPLS aggiunte ai pacchetti da parte dei PEs sono divise in classi: proprio come un centro postale smista le lettere sulla base dei destinatari, MPLS assegna una etichetta in base al CE destinatario. Queste label diventano quindi un numero nettamente inferiore se comparato con le righe delle tabelle di routing di un Core Provider. Meno elementi tra cui scegliere si traduce quindi in una ricerca più veloce ed efficace.

Anche se i componenti hardware odierni (ASICs) consultano le tabelle di routing molto più vlocemente del passato, MPLS rimane uno dei più importanti standard per quanto riguarda la trasmissione WAN. Bisogna infatti notare come l'etichetta MPLS non indichi solamente il destinatario, bensì può essere aggiunta anche una prioritizzazione del traffico. Specialmente per quanto riguarda il Voice over IP (VoIP), MPLS gioca un ruolo determinante: assegnando in fatti la priorità massima a questo tipo di traffico possiamo assicurarci che la voce subisca il minimo ritardo nell'attraversare la rete. Viceversa, possiamo assegnare alle email una priorità inferiore dato che nella maggior parte dei casi un ritardo di qualche secondo di un messaggio nella casella di posta non rappresenta un problema.

STUDENTSUPSI

Un altro beneficio di MPLS è la possibilità di controllare il percorso di un pacchetto all'interno della rete. Per capire meglio questo concetto, introduciamo la sigla *LSP*, acronimo di *Label Switched Path*. Un LSP è un percorso specifico che inizia in un router MPLS di ingresso e termina in un router MPLS di uscita (ingress ed egress routers). Quando al pacchetto in ingresso viene aggiunta un'etichetta MPLS, viene altresì definito un determinato LSP associato ad essa. È importante notare che ogni pacchetto che possiede lo stesso LSP seguirà il medesimo percorso all'interno della rete.

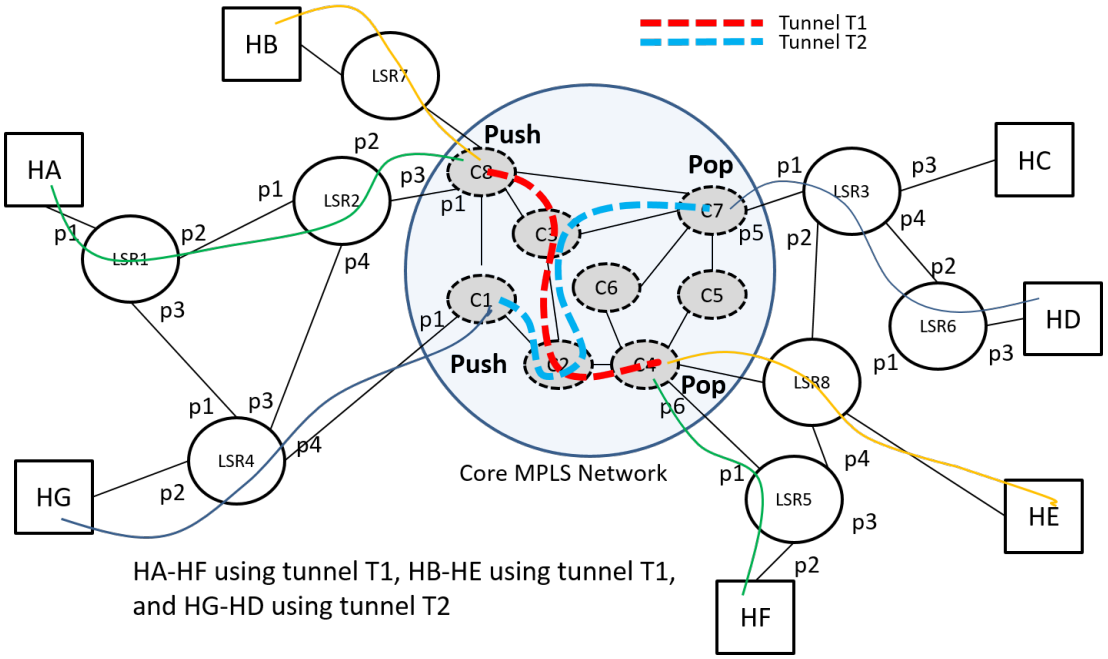


Figura 5.2: MPLS Tunneling

La figura 6.2 illustra questo concetto fondamentale: il cuore della rete MPLS è la zona in cui avvengono la maggior parte dei processi di computazione hardware, e diminuire questi associando più percorsi ad un'unica etichetta diminuisce il tempo di calcolo e di conseguenza aumenta la velocità di trasmissione.

Il prossimo acronimo che andremo a definire si chiama *LDP*, ovvero *Label Distribution Protocol*. Lo scopo di questo protocollo è la distribuzione delle etichette ai pacchetti in ingresso (processo di push), lo scambio (swap) e la rimozione in uscita (pop). LDP si appoggia interamente ai protocolli di routing per gestire l'assegnazione delle etichette, dato che è solo grazie alle tabelle di routing che possiamo calcolare il percorso migliore per un pacchetto. È quindi sbagliato pensare che MPLS possa sostituire un protocollo di routing, dato che senza di esso non potremmo ottenere i percorsi dei pacchetti e di conseguenza non potremmo assegnare nessuna etichetta.

Per i concetti appena descritti, negli anni MPLS ha assunto un ruolo determinante nella

STUDENTSUPSI

realizzazione di VPN (*Virtual Private Networking*). Il seguente capitolo spiega il concetto in modo dettagliato.

5.1 MPLS VPN

MPLS VPN è un metodo utilizzato per creare reti virtuali private sfruttando il cloud MPLS. All'interno di questo è possibile trasportare diversi tipi di traffico dati, come per esempio voce, streaming video e audio, pagine web, e altro ancora.

Possiamo categorizzare tre tipi di MPLS VPN:

- Point to Point (Pseudowire)
- Layer 2 VPN (VPLS)
- Layer 3 VPN (VPRN)

In questa tesi viene analizzato il terzo tipo di MPLS VPN: *Virtual Private Routed Network* (VPRN).

5.1.1 Layer 3 VPN - VPRN

VPRN, acronimo di Virtual Private Routed Network, utilizza VRF (VPN Routing and Forwarding) per segmentare le tabelle di routing per ogni customer che utilizza il servizio. Viene creata una tabella di routing indipendente per ogni cliente che desidera utilizzare VPRN.

Per rendere più chiara l'idea, ho implementato una topologia in GNS3 che possa simulare il comportamento di VRF:

STUDENTSUPSI

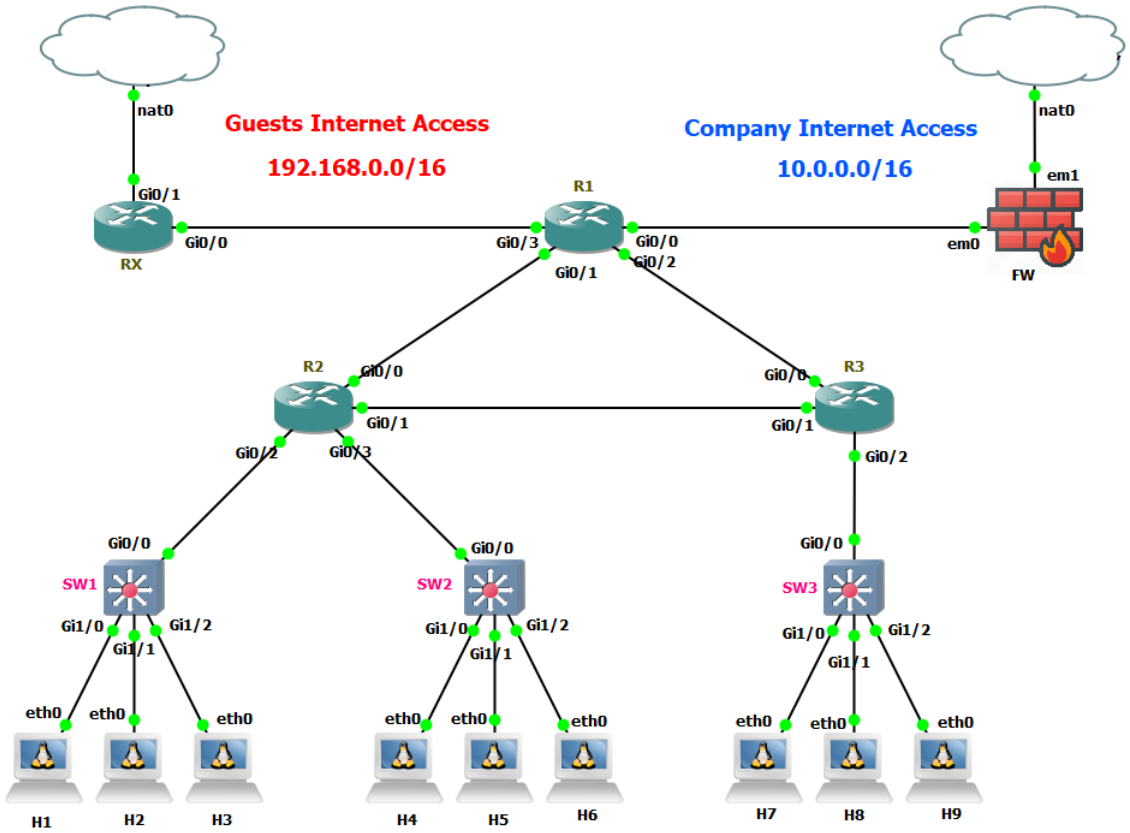


Figura 5.3: VPN Routing and Forwarding

Assumiamo che la seguente topologia di rete sia quella di una ditta. Come ci si può immaginare, tutto il traffico che proviene dall'esterno della ditta deve passare attraverso il firewall (Company Internet Access, indirizzo 10.0.0.0/16) ed è considerato quindi traffico sicuro. Nel caso la ditta avesse ospiti, possiamo definire una rete privata (Guests Internet Access, con indirizzo 192.168.0.0/16) riservata al traffico privato. In questo modo possiamo separare le due reti, permettendo ai visitatori di navigare in internet, ma sfruttando una connessione diversa da quella utilizzata dagli impiegati della ditta.

VRF è molto simile al concetto di VLAN, con la differenza che quest'ultima lavora a L2, mentre VRF a L3. Con questo in mente, possiamo ora passare all'implementazione di una rete VPN basata su MPLS.

5.2 Realizzazione di VPN MPLS

Dopo aver analizzato il concetto di VRF, possiamo ora implementare una Virtual Private Network che lavori in MPLS. Per fare ciò, mi servo di GNS3 e creo la seguente rete:

STUDENTSUPSI

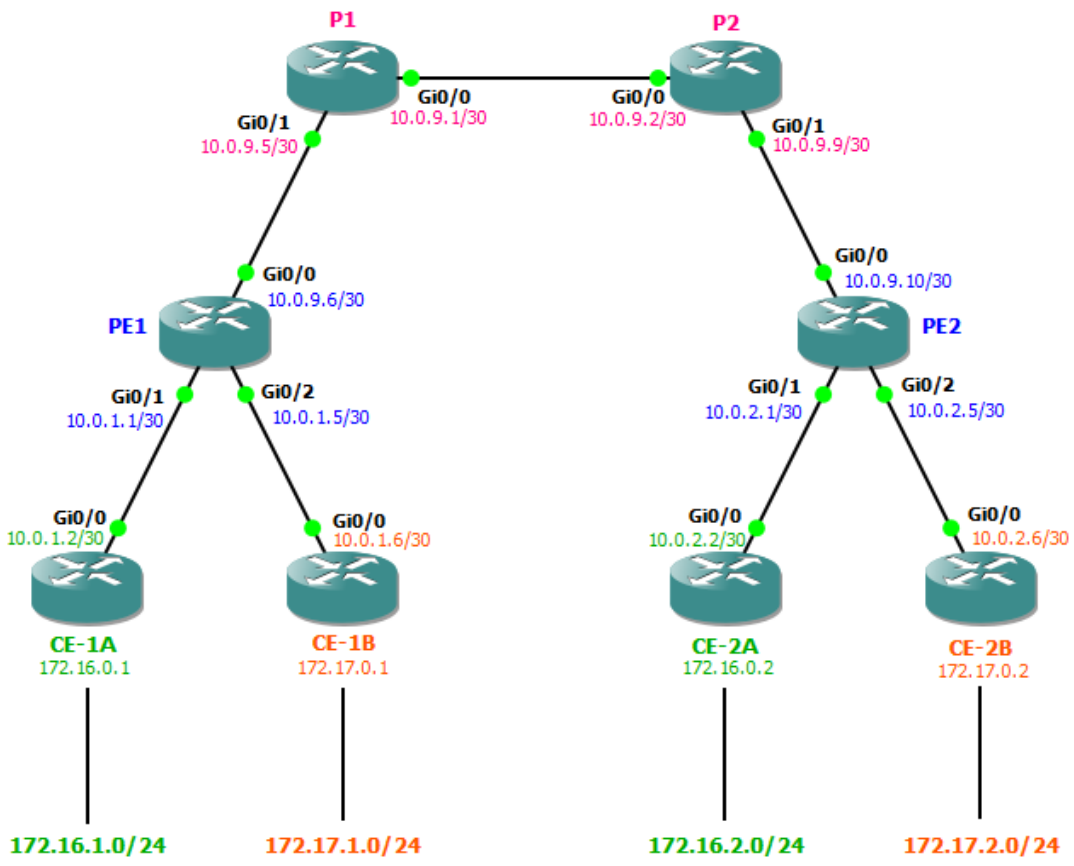


Figura 5.4: VPN - MPLS

Per ottenere una VPN MPLS dobbiamo svolgere 5 punti fondamentali:

- Abilitare MPLS a livello Provider (P1, P2)
- Creare VRFs e assegnare le interfacce desiderate
- Configurare MP-BGP tra i Provider Edge routers
- Configurare OSPF tra ogni Provider Edge e il relativo Customer Edge router
- Abilitare la ridistribuzione dei percorsi tra Customer Edge e Core Provider

5.2.1 Abilitare MPLS

Come già evidenziato nella sezione 5.1, essendo MPLS basato su IGP, abbiamo bisogno di specificare che tipo di protocollo interno utilizzare. Nel mio caso ho scelto OSPF, che deve essere attivo su tutti i Core Providers ed i Provider Edge routers.

STUDENTSUPSI

Come si evince dai seguenti listati, tutti i routers interessati sono stati configurati in modo corretto. MPLS è quindi attivo sulle seguenti interfacce:

```
P1#show mpls interfaces
Interface              IP            Tunnel    BGP  Static  Operational
GigabitEthernet0/0     Yes (ldp)     No        No   No      Yes
GigabitEthernet0/1     Yes (ldp)     No        No   No      Yes
```

Listing 5.1: P1 - Show MPLS Interfaces

```
P2#show mpls interfaces
Interface              IP            Tunnel    BGP  Static  Operational
GigabitEthernet0/0     Yes (ldp)     No        No   No      Yes
GigabitEthernet0/1     Yes (ldp)     No        No   No      Yes
```

Listing 5.2: P2 - Show MPLS Interfaces

```
PE1#show mpls interfaces
Interface              IP            Tunnel    BGP  Static  Operational
GigabitEthernet0/0     Yes (ldp)     No        No   No      Yes
```

Listing 5.3: PE1 - Show MPLS Interfaces

```
PE2#show mpls interfaces
Interface              IP            Tunnel    BGP  Static  Operational
GigabitEthernet0/0     Yes (ldp)     No        No   No      Yes
```

Listing 5.4: PE2 - Show MPLS Interfaces

Anche LDP risulta attivo: il seguente listato mostra le adiacenze LDP relative al Core Provider 1. Per evitare ridondanze, viene mostrato solo questo listato; gli altri routers sono stati testati nello stesso modo e verificati correttamente.

```
P1#sh mpls ldp neighbor
Peer LDP Ident: 10.0.0.3:0; Local LDP Ident 10.0.0.1:0
  TCP connection: 10.0.0.3.23109 - 10.0.0.1.646
  State: Oper; Msgs sent/rcvd: 39/40; Downstream
  Up time: 00:25:51
  LDP discovery sources:
  GigabitEthernet0/1, Src IP addr: 10.0.9.6
  Addresses bound to peer LDP Ident:
  10.0.9.6        10.0.0.3
Peer LDP Ident: 10.0.0.2:0; Local LDP Ident 10.0.0.1:0
  TCP connection: 10.0.0.2.20041 - 10.0.0.1.646
  State: Oper; Msgs sent/rcvd: 22/22; Downstream
  Up time: 00:11:08
  LDP discovery sources:
  GigabitEthernet0/0, Src IP addr: 10.0.9.2
  Addresses bound to peer LDP Ident:
```

STUDENTSUPSI

```
  10.0.9.2          10.0.9.9          10.0.0.2
```

Listing 5.5: P1 - Show MPLS LDP Neighbor

Ora che il primo compito è stato eseguito possiamo passare alla creazione di VRFs, come descritto nel prossimo capitolo.

5.2.2 Creare ed Assegnare VRFs

Il prossimo step consiste nel creare VRFs sui nostri Provider Edge routers. Una volta fatto questo, assegneremo alle interfacce utente (Customer Edge) una Route Distinguisher (RD), per identificare unicamente i prefissi come appartenenti a quella VRF, insieme ad una o più Route Targets (RTs) per specificare come i percorsi debbano essere importati ed esportati dalla VRF.

Per brevità, riporto solamente la configurazione e l'output del Provider Edge 1, dato che i comandi sono gli stessi nel caso del PE2:

```
PE1(config)# ip vrf Customer_A
PE1(config-vrf)# rd 65000:1
PE1(config-vrf)# route-target both 65000:1
PE1(config-vrf)# ip vrf Customer_B
PE1(config-vrf)# rd 65000:2
PE1(config-vrf)# route-target both 65000:2
```

Listing 5.6: PE1 - Creazione VRF

```
PE1(config)# interface g0/1
PE1(config-if)# ip vrf forwarding Customer_A
% Interface GigabitEthernet0/1 IP address 10.0.1.1 removed due to
    enabling VRF Customer_A
PE1(config-if)# ip address 10.0.1.1 255.255.255.252
PE1(config-if)# interface g0/2
PE1(config-if)# ip vrf forwarding Customer_B
% Interface GigabitEthernet0/2 IP address 10.0.1.5 removed due to
    enabling VRF Customer_B
PE1(config-if)# ip address 10.0.1.5 255.255.255.252
PE1(config-if)# end
PE1#sh ip vrf int
Interface              IP-Address      VRF                    Protocol
Gi0/1                  10.0.1.1        Customer_A             up
Gi0/2                  10.0.1.5        Customer_B             up
```

Listing 5.7: PE1 - Assegnazione VRF

STUDENTSUPSI

Come è possibile notare, sono state create due tabelle di routing distinte: una relativa al Customer 1A, l'altra al Customer 1B. Lo stesso è stato eseguito per CE-2A e CE-2B.

5.2.3 Configurare MP-BGP sui PE Routers

Per permettere che due Provider Edge routers riconoscano i percorsi VRF tra loro, dobbiamo configurare Multi Protocol BGP (MP-BGP). MP-BGP è leggermente diverso dal BGP tradizionale: supporta più famiglie di indirizzi (IPv4, IPv6) in un'adiacenza comune. Supporta inoltre la segnalazione di percorsi VPN, i quali sono più lunghi di un normale percorso (questo è dovuto al fatto di avere aggiunto un Route Distinguisher di 64-bit nella fase di configurazione VRF).

MP-BGP viene configurato solamente nei routers PE: i Providers dipendono interamente da IGP (OSPF nel nostro caso) e MPLS per poter inoltrare il traffico all'interno della rete di providers. I routers CE non hanno nessuna conoscenza circa i percorsi che non riguardano il proprio VRF.

Di seguito riporto la configurazione di MP-BGP per PE1 e PE2:

```
PE1(config)# router bgp 65000
PE1(config-router)# neighbor 10.0.0.4 remote-as 65000
PE1(config-router)# neighbor 10.0.0.4 update-source loopback 0
PE1(config-router)# address-family vpnv4
PE1(config-router-af)# neighbor 10.0.0.4 activate
```

Listing 5.8: PE1 - Configurazione MP-BGP

```
PE2(config)# router bgp 65000
PE2(config-router)# neighbor 10.0.0.3 remote-as 65000
PE2(config-router)# neighbor 10.0.0.3 update-source loopback 0
PE2(config-router)# address-family vpnv4
PE2(config-router-af)# neighbor 10.0.0.3 activate
```

Listing 5.9: PE2 - Configurazione MP-BGP

```
PE1#show bgp vpnv4 unicast all summary
BGP router identifier 10.0.0.3, local AS number 65000
BGP table version is 9, main routing table version 9
[...]
Neighbor        V           AS MsgRcvd MsgSent   TblVer  InQ OutQ
10.0.0.4        4        65000     146     143        9    0
```

Listing 5.10: PE1 - Show BGP VPNv4 Unicast All Summary

Come è possibile vedere, il router PE1 (10.0.0.3/32) ha una relazione di vicinanza con PE2 (10.0.0.4/32). I due routers presentano ora una adiacenza MP-BGP, come desiderato.

STUDENTSUPSI

5.2.4 Configurare PE-CE OSPF

Abbiamo appena configurato MP-BGP tra i due Provider Edges. Ora abbiamo bisogno di configurare un IGP tra ogni PE ed i relativi CEs, in modo da poter scambiare informazioni tra i Customer Edges. In questo caso utilizzo OSPF, ma avrei potuto farlo con qualsiasi altro IGP (EIGRP, RIPv2).

OSPF ad area singola è già stato implementato sui CEs: tutte le interfacce fanno parte dell'area 0. Da sottolineare è il fatto che anche se stiamo utilizzando OSPF tra ognuno dei routers CE ed il relativo PE, questi processi sono isolati dalla topologia OSPF dei Providers. La seguente figura illustra questo concetto:

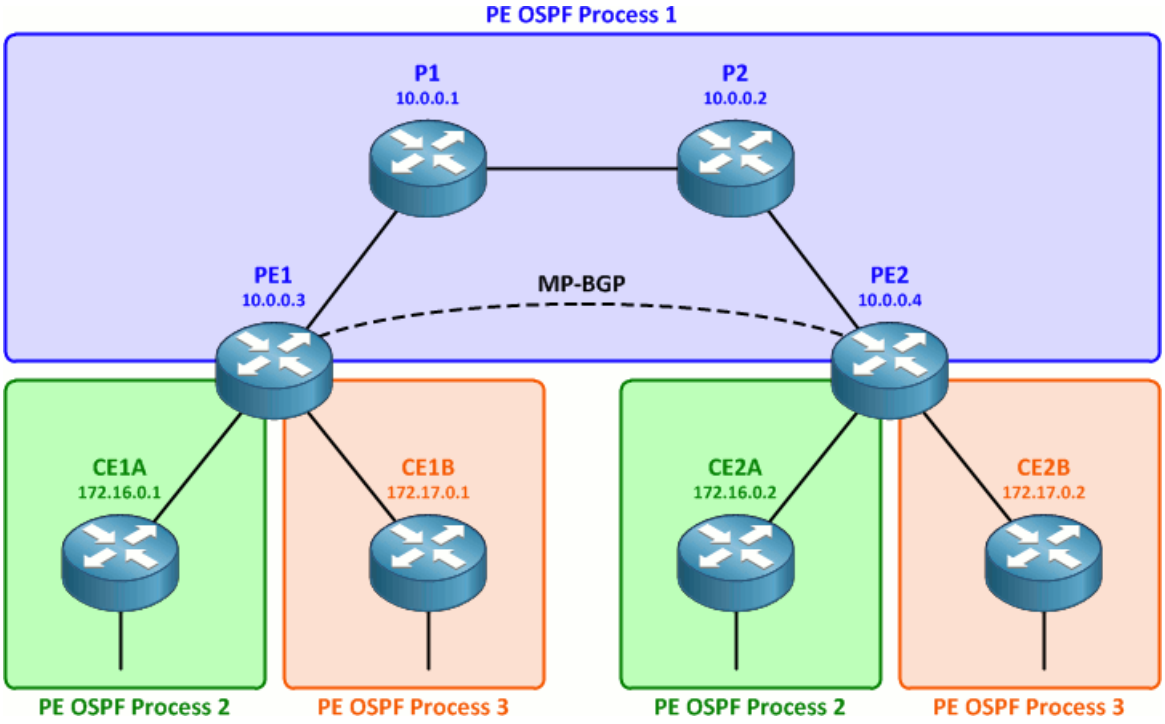


Figura 5.5: Differenti Processi OSPF per Providers e CEs

Il processo OSPF è già stato configurato sui routers PE come processo 1. Configuro ora un processo OSPF addizionale per ogni CE a cui un PE è collegato. Ogni router PE avrà quindi tre processi OSPF in totale: uno per la rete P, uno per CE1 ed uno per CE2. Come riporto ora nei seguenti listati, mentre il processo OSPF del Provider esiste nella tabella globale di routing, i due processi CE verranno assegnati al rispettivo VRF (Customer A e Customer B).

```
PE1(config)# router ospf 2 vrf Customer_A
PE1(config-router)# router-id 10.0.1.1
PE1(config-router)# interface f0/0
PE1(config-if)# ip ospf 2 area 0
```

STUDENTSUPSI

```
PE1(config-if)# router ospf 3 vrf Customer_B
PE1(config-router)# router-id 10.0.1.5
PE1(config-router)# interface f0/1
PE1(config-if)# ip ospf 3 area 0
```

Listing 5.11: PE1 - Configurazione PE-CE OSPF

```
PE2(config)# router ospf 2 vrf Customer_A
PE2(config-router)# router-id 10.0.2.1
PE2(config-router)# interface f0/0
PE2(config-if)# ip ospf 2 area 0
PE2(config-if)# router ospf 3 vrf Customer_B
PE2(config-router)# router-id 10.0.2.5
PE2(config-router)# interface f0/1
PE2(config-if)# ip ospf 3 area 0
```

Listing 5.12: PE2 - Configurazione PE-CE OSPF

5.2.5 Configurare la Ridistribuzione dei Percorsi

A questo punto abbiamo il protocollo MP-BGP attivo e funzionante tra PE1 e PE2. OSPF è altresì attivo in 3 processi, come visibile in figura 6.5. L'ultimo step è l'integrazione del protocollo OSPF con quello MP-BGP: questo processo viene definito *Route Redistribution*.

Configuro quindi PE1 e PE2 (i due routers interessati da MP-BGP) in modo tale che possano ridistribuire i percorsi imparati tramite OSPF utilizzando MP-BGP:

```
PE1(config)# router bgp 65000
PE1(config-router)# address-family ipv4 vrf Customer_A
PE1(config-router-af)# redistribute ospf 2
PE1(config-router-af)# address-family ipv4 vrf Customer_B
PE1(config-router-af)# redistribute ospf 3
```

Listing 5.13: PE1 - Route Redistribution (OSPF in MP-BGP)

```
PE2(config)# router bgp 65000
PE2(config-router)# address-family ipv4 vrf Customer_A
PE2(config-router-af)# redistribute ospf 2
PE2(config-router-af)# address-family ipv4 vrf Customer_B
PE2(config-router-af)# redistribute ospf 3
```

Listing 5.14: PE2 - Route Redistribution (OSPF in MP-BGP)

Possiamo verificare che i percorsi imparati dal Customer A (172.16.0.0/24) e dal Customer B (172.17.0.0/16) appaiono ora nelle tabele BGP per ogni rispettivo VRF:

```
PE1#sh ip bgp vpnv4 vrf Customer_A
```

STUDENTSUPSI

```

[...]

Network          Next Hop            Metric LocPrf Weight Path
Route Distinguisher: 65000:1 (default for vrf Customer_A)
*>   10.0.1.0/30      0.0.0.0                  0         32768 ?
*>i  10.0.2.0/30      10.0.0.4                 0    100      0 ?
*>   172.16.0.1/32    10.0.1.2                 2         32768 ?
*>i  172.16.0.2/32    10.0.0.4                 2    100      0 ?
*>   172.16.1.0/24    10.0.1.2                 2         32768 ?
*>i  172.16.2.0/24    10.0.0.4                 2    100      0 ?
```

Listing 5.15: PE1 - Show IP BGP VPNv4 VRF Customer_A

Da notare i percorsi aggiunti (identificati dalla lettera *i*):

- 10.0.2.0/30 è il collegamento tra PE2 e CE1A
- 172.16.1.0/24 è l'indirizzo di Loopback di CE1A
- 172.16.2.0/24 è la rete al di sotto della quale CE2A rappresenta il default gateway

Prima abbiamo integrato OSPF in MP-BGP, ma essendo la redistribuzione un processo biunivoco, abbiamo bisogno ora di effettuare l'operazione contraria: integrare MP-BGP in OSPF:

```
PE1(config)# router ospf 2
PE1(config-router)# redistribute bgp 65000 subnets
PE1(config-router)# router ospf 3
PE1(config-router)# redistribute bgp 65000 subnets
```

Listing 5.16: PE1 - Route Redistribution (MP-BGP in OSPF)

```
PE2(config)# router ospf 2
PE2(config-router)# redistribute bgp 65000 subnets
PE2(config-router)# router ospf 3
PE2(config-router)# redistribute bgp 65000 subnets
```

Listing 5.17: PE2 - Route Redistribution (MP-BGP in OSPF)

5.2.6 Fase di Test e Conferma

A questo punto CE1A e CE2A devono essere capaci di comunicare fra loro, lo stesso vale per CE1B e CE2B.

STUDENTSUPSI

```
CE1A#ping 172.16.0.2
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.0.2, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 9/10/15 ms
```

Listing 5.18: CE1A verso CE2A

```
CE1B#ping 172.17.0.2
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.17.0.2, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 9/10/14 ms
```

Listing 5.19: CE1B verso CE2B

Entrambe le comunicazioni funzionano alla perfezione: i due Customers della rete A si scambiano messaggi, e lo stesso vale per i Customers della rete B. Essendo su due reti separate, i Customers della rete A non devono essere capaci di comunicare con i Customers della rete B, e viceversa:

```
CE1A#ping 172.17.0.1
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.17.0.1, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
```

Listing 5.20: CE1A verso CE1B

```
CE1A#ping 172.17.0.2
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.17.0.2, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
```

Listing 5.21: CE1A verso CE2B

Per ultimo, analizziamo tramite il comando *traceroute* il percorso dei pacchetti in rete:

```
CE1A#traceroute 172.16.0.2
Type escape sequence to abort.
Tracing the route to 172.16.0.2
VRF info: (vrf in name/id, vrf out name/id)
1 10.0.1.1 4 msec 3 msec 4 msec
2 10.0.9.5 [MPLS: Labels 17/23 Exp 0] 12 msec 9 msec 9 msec
3 10.0.9.2 [MPLS: Labels 17/23 Exp 0] 10 msec 8 msec 9 msec
4 10.0.2.1 [MPLS: Label 23 Exp 0] 9 msec 8 msec 7 msec
5 10.0.2.2 10 msec 8 msec 8 msec
```

Listing 5.22: Traceroute - CE1A su CE2A

STUDENTSUPSI

```
CE1B#traceroute 172.17.0.2
Type escape sequence to abort.
Tracing the route to 172.17.0.2
VRF info: (vrf in name/id, vrf out name/id)
1 10.0.1.5 9 msec 4 msec 4 msec
2 10.0.9.5 [MPLS: Labels 17/25 Exp 0] 13 msec 9 msec 10 msec
3 10.0.9.2 [MPLS: Labels 17/25 Exp 0] 10 msec 9 msec 9 msec
4 10.0.2.5 [MPLS: Label 25 Exp 0] 10 msec 8 msec 8 msec
5 10.0.2.6 10 msec 8 msec 11 msec
```

Listing 5.23: Traceroute - CE1B su CE2B

Questi due outputs sono di particolare interesse per più motivi:

- Il Provider Edge di ingreso effettua un label push, quello di uscita un label pop;
- Come ci aspettiamo, MPLS agisce solo tra Providers e Core Providers;
- L'etichetta apposta dal LDP è la stessa sia per la rete A che per la rete B, dato che entrambe condividono lo stesso percorso PE-P;
- Viene apposta l'etichetta *23* per la rete A e *25* per la rete B, corrispondente alle due VPNs create;
- Exp 0 è un riferimento al QoS del pacchetto: come precedentemente indicato, alcuni pacchetti devono essere prioritizzati rispetto ad altri (VoIP).

5.3 Integrazione Virtuale - Fisica

Ora che abbiamo costruito la nostra rete virtualizzata e abbiamo effettuato un test all'interno del software GNS3, lo step successivo sarà quello di *integrare* la parte fisica con quella virtuale.

La parte fisica consiste in tutti quei dispositivi che possono essere utilizzati per trasmettere dati a L3 (e.g. i dispositivi di rete che possiedono un indirizzo IPv4).

Essendo connessi ad uno switch, i dispositivi facenti parte delle varie reti CE1A, CE1B, CE2A e CE2B devono necessariamente fare parte di VLANs differenti, o in alternativa avere maschere di sottorete diverse. Chiaramente, essendo la prima opzione migliore della seconda, io ho inizialmente optato per questa. Purtroppo, ESXi è limitato ad una sola VLAN per ogni port group e di conseguenza questa opzione è infattibile. Il dispositivo che permette di integrare più VLANs in un solo port group è Cisco Nexus 1000v, un dispositivo di cui ho parlato nel Capitolo 2.

STUDENTSUPSI

Discutendo con il Professore, abbiamo optato per un cambio della subnet mask, in quanto il costo di Cisco Nexus 1000v era decisamente troppo elevato. La maschera di rete è passata quindi da /24 a /29. Di seguito riporto una tabella che riassume il range di indirizzi di una rete /29:

Network #	IP Range	Broadcast
.0	.1-.6	.7
.8	.9-.14	.15
.16	.17-.22	.23
.24	.25-.30	.31
.32	.33-.38	.39
.40	.41-.46	.47
.48	.49-.54	.55
.56	.57-.62	.63
.64	.65-.70	.71
.72	.73-.78	.79

Nel mio caso ho optato per 5 devices, di cui elenco le caratteristiche ed i relativi indirizzi IP:

- Raspberry 1: IPv4: 193.246.121.51/29 Default Gateway: .49/29
- Raspberry 2: IPv4: 193.246.121.58/29 Default Gateway: .57/29
- Windows Client: IPv4: 193.246.121.66/29 Default Gateway: .65/29
- Ubuntu Client: IPv4 193.246.121.74/29 Default Gateway: .73/29

La seguente figura mostra la realizzazione del progetto finale:

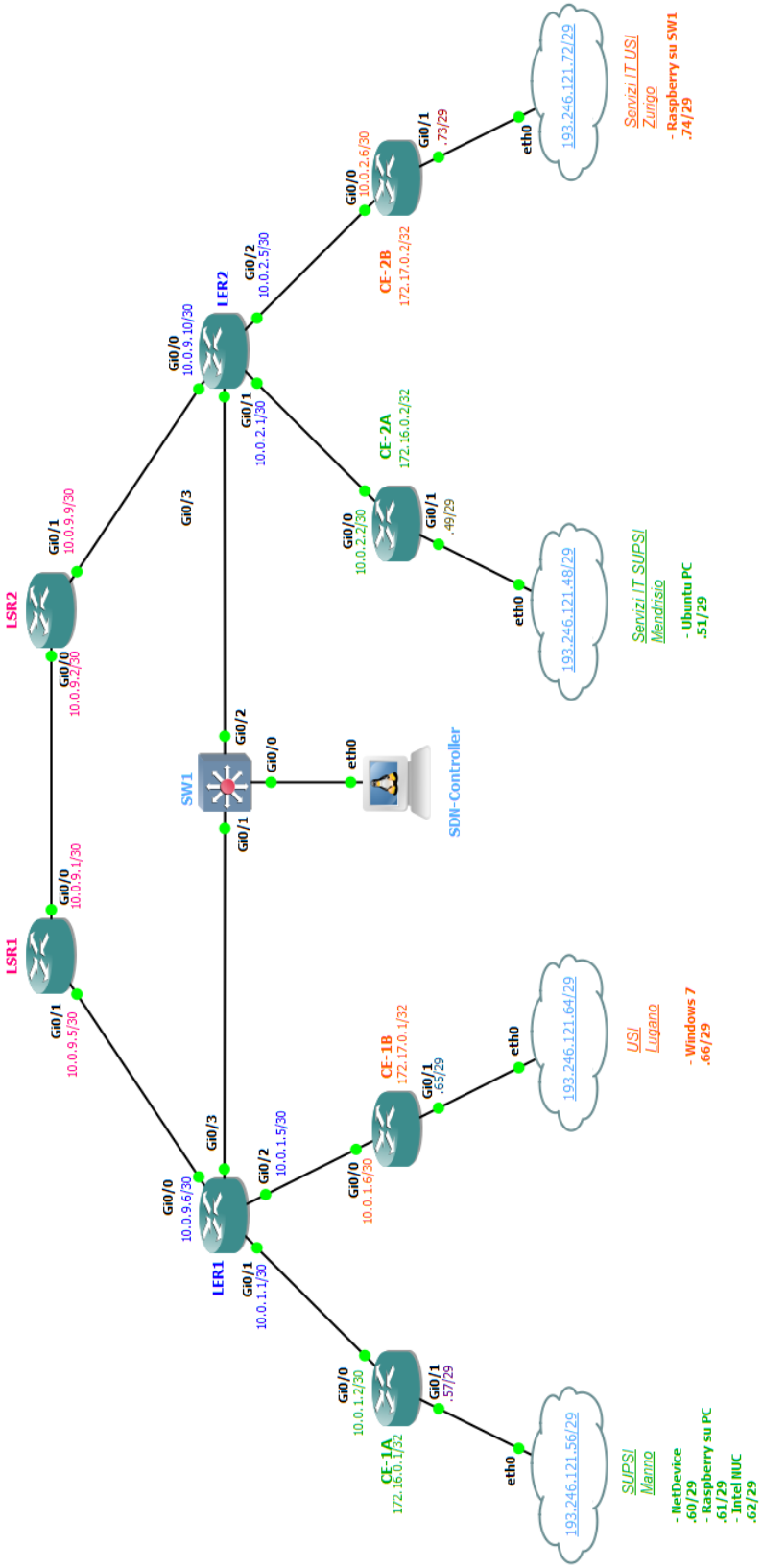
STUDENTSUPSI

Per enfatizzare il concetto di VPN over MPLS, vengono rappresentate quattro zone fisicamente distanti fra loro:

- SUPSI (Manno)
- Servizi IT SUPSI (Mendrisio)
- USI (Lugano)
- Servizi IT USI (Bellinzona)

I customers SUPSI devono potersi connettere ai servizi IT SUPSI, rispettivamente gli utenti USI si connetteranno ai servizi IT USI. La linea MPLS è condivisa, ma le reti private sono differenti.

STUDENTSUPSI



STUDENTSUPSI

STUDENTSUPSI

Capitolo 6

VPN over MPLS con Controller SDN

La parte finale di questa tesi vuole esaminare le potenzialità che SDN fornisce nella gestione del traffico all'interno di una rete VPN over MPLS.

Questo ultimo capitolo rappresenta l'integrazione delle prime due macrosezioni della mia tesi. Andrò ad implementare un controllore SDN che verrà applicato alla rete MPLS analizzata nel capitolo precedente: in questo modo sarà possibile gestire via software l'appartenenza dei customers alla rete privata A, piuttosto che alla B.

Nella topologia rappresentata in figura 5.1 è stato aggiunto alla nota rete MPLS un controllore SDN, rappresentato dal docker Ubuntu posizionato al centro del diagramma:

A sua volta, il docker Ubuntu è connesso ad uno switch Cisco IOSv L2.

L'obiettivo finale è la possibilità di aggiungere, togliere e modificare l'appartenenza di un client da una rete ad un'altra. Per fare ciò ho scritto degli script in Python che mi permettessero di modificare la mia topologia di rete a piacere.

Connettendo il controller SDN ai due Provider Edge Routers (LER1 e LER2), posso raggiungere entrambi e di conseguenza programmarli via SSH.

Il codice che ho scritto è il seguente:

```
from netmiko import ConnectHandler

LER1 = {
'device_type': 'cisco_ios',
'ip': '193.246.121.41',
'username': 'davide',
'password': 'ganna'
}

LER2 = {
'device_type': 'cisco_ios',
'ip': '193.246.121.42',
'username': 'davide',
'password': 'ganna'
}

routerToMove = raw_input('Quale dispositivo vuoi spostare? [CE1A, CE2A,
    CE1B, CE2B]: ')
place = raw_input('In quale VPN? [A, B] ')

if routerToMove == 'CE1A':
  print('--- Moving CE1A under the selected VPN... ---')
  net_connect = ConnectHandler(**LER1)

  if place == 'A':
    config_commands = [
    'interface GigabitEthernet0/1',
    'ip vrf forwarding Customer_A',
    'ip address 10.0.1.1 255.255.255.252',
    'no shut',
    'ip ospf 2 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE1A successfully moved under VPN A! ---')
  if place == 'B':
    config_commands = [
```

STUDENTSUPSI

```
    'interface GigabitEthernet0/1',
    'ip vrf forwarding Customer_B',
    'ip address 10.0.1.1 255.255.255.252',
    'no shut',
    'ip ospf 3 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE1A successfully moved under VPN B! ---')

if routerToMove == 'CE1B':
  print('--- Moving CE1B under the selected VPN... ---')
  net_connect = ConnectHandler(**LER1)

  if place == 'A':
    config_commands = [
    'interface GigabitEthernet0/2',
    'ip vrf forwarding Customer_A',
    'ip address 10.0.1.5 255.255.255.252',
    'no shut',
    'ip ospf 2 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE1B successfully moved under VPN A! ---')
  if place == 'B':
    config_commands = [
    'interface GigabitEthernet0/2',
    'ip vrf forwarding Customer_B',
    'ip address 10.0.1.5 255.255.255.252',
    'no shut',
    'ip ospf 3 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE1B successfully moved under VPN B! ---')

if routerToMove == 'CE2A':
  print('--- Moving CE2A under the selected VPN... ---')
  net_connect = ConnectHandler(**LER2)

  if place == 'A':
    config_commands = [
    'interface GigabitEthernet0/1',
    'ip vrf forwarding Customer_A',
    'ip address 10.0.2.1 255.255.255.252',
    'no shut',
    'ip ospf 2 area 0',
```

STUDENTSUPSI

```
    'duplex auto',
    'speed auto'
    ]
    print('--- CE2A successfully moved under VPN A! ---')
  if place == 'B':
    config_commands = [
    'interface GigabitEthernet0/1',
    'ip vrf forwarding Customer_B',
    'ip address 10.0.2.1 255.255.255.252',
    'no shut',
    'ip ospf 3 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE2A successfully moved under VPN B! ---')

if routerToMove == 'CE2B':
  print('--- Moving CE2B under the selected VPN... ---')
  net_connect = ConnectHandler(**LER2)

  if place == 'A':
    config_commands = [
    'interface GigabitEthernet0/2',
    'ip vrf forwarding Customer_A',
    'ip address 10.0.2.5 255.255.255.252',
    'no shut',
    'ip ospf 2 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE2B successfully moved under VPN A! ---')
  if place == 'B':
    config_commands = [
    'interface GigabitEthernet0/2',
    'ip vrf forwarding Customer_B',
    'ip address 10.0.2.5 255.255.255.252',
    'no shut',
    'ip ospf 3 area 0',
    'duplex auto',
    'speed auto'
    ]
    print('--- CE2B successfully moved under VPN B! ---')

net_connect.enable()
net_connect.config_mode()
output = net_connect.send_config_set(config_commands)

```

STUDENTSUPSI

```
net_connect.disconnect()
```

Listing 6.1: VPN over MPLS - SDN Script

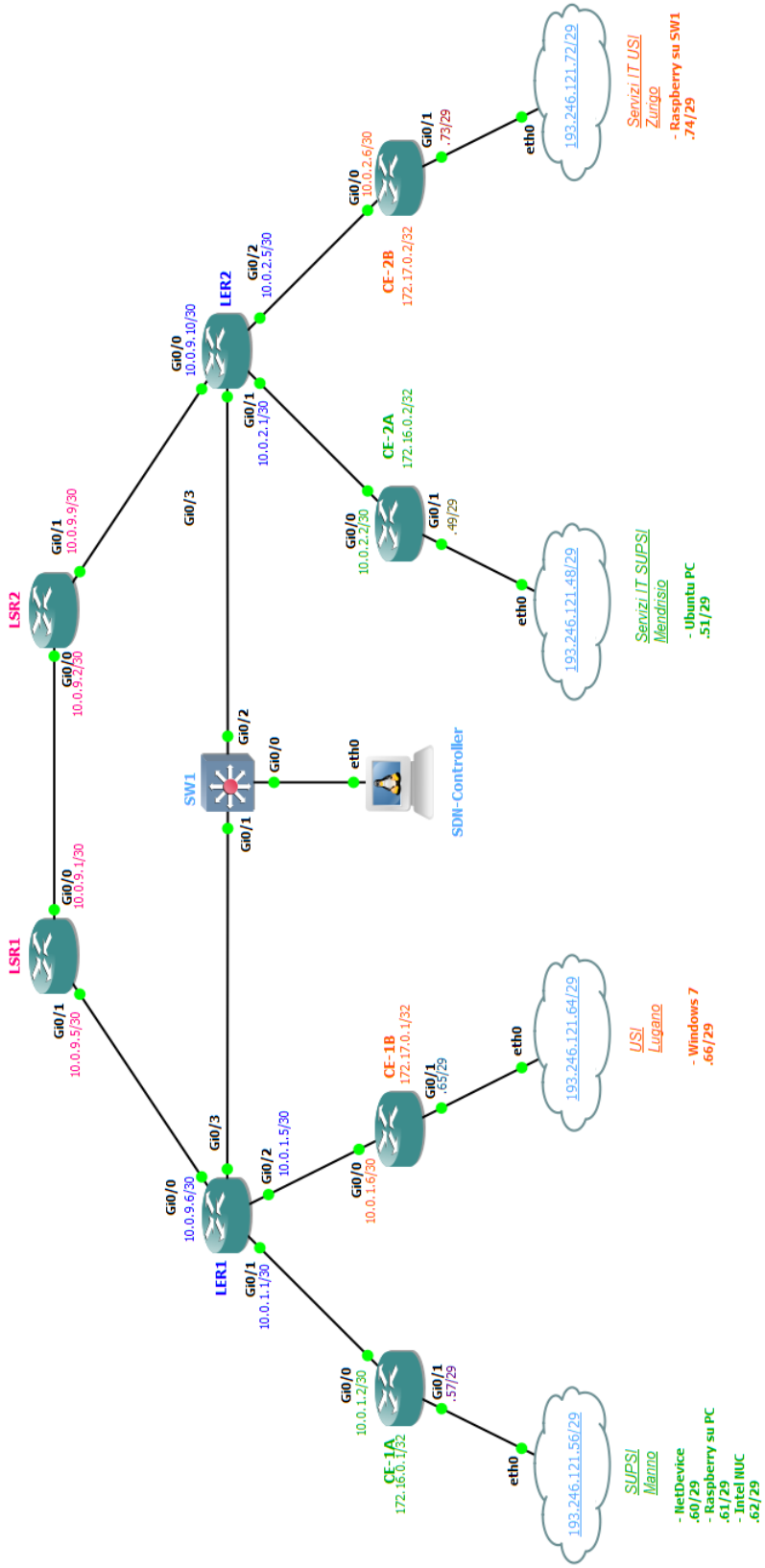
Esso chiede all'utente di specificare quale customer si desidera spostare (*routerToMove*), e in quale sottorete (*place*). Dopo aver ricevuto l'input, tramite la struttura condizionale si entra in uno degli otto casi disponibili.

Come già detto nel capitolo precedente, VRF crea due tabelle di routing distinte, e lo spostamento del customer A nella rete B (o viceversa) porta all'aggiornamento della routing table introducendo i percorsi imparati dal router aggiunto, e/o eliminando quelli del router tolto. Questo processo di aggiornamento dei percorsi viene definito *re-routing*.

A dipendenza della potenza del processore del server, della RAM dedicata al router e dei percorsi da aggiornare, questo procedimento può richiedere più o meno tempo. Nei miei test, il processo di re-routing impiega circa 30 secondi prima di essere completato.

Per visualizzare meglio questo cambiamento, ho implementato una dimostrazione di streaming video mediante VLC in laboratorio. In qualsiasi momento, tramite lo script precedente è possibile cambiare l'appartenenza di un client ad una VPN piuttosto che all'altra. Se il customer viene spostato in una rete privata a cui è destinato lo streaming potrà accedere ad esso. Analogamente, nel caso non facesse più parte della VPN a cui lo streaming era destinato, il client non riceverà più i pacchetti video.

STUDENTSUPSI



STUDENTSUPSI

Capitolo 7

Conclusioni

Come esposto intensamente in questa tesi, la tendenza a virtualizzare un sistema presenta enormi vantaggi. La possibilità di integrare le parti virtuali con quelle fisiche suggerisce di spostarsi al più presto verso questo mondo. Per questioni di spazio e costi, la possibilità di virtualizzare un sistema viene sfruttata dalle grandi aziende per implementare soluzioni efficaci, scalabili e ad alta QoS, le quali permettono di effettuare le stesse operazioni di un sistema fisico all'interno di un server. Sapere in che modo lavora un hypervisor come VMware ESXi è di fondamentale importanza considerando che con il passare degli anni sempre più dispositivi verranno emulati virtualmente all'interno di un server.

Con il passare del tempo la tecnologia si evolve. Sebbene l'elettronica dei componenti di un sistema (come quella presente in uno switch oppure in un router) occupi sempre meno spazio, le funzioni a livello data plane rimangono le stesse. Quello che cambia è *la modalità* con la quale questi compiti vengono eseguiti. Se alcuni anni fa ogni dispostivo prendeva delle decisioni per conto proprio, la tendenza di oggi è invece quella di spostarsi verso un sistema centralizzato che possa controllare più unità connesse fra loro.

Come esposto nel secondo capitolo, programmare una rete non permette solamente di risparmiare una quantità ingente di tempo, ma fornisce anche un piano di backup nel caso venga applicata una configurazione sbagliata per errore umano. I molteplici vantaggi offerti dai numerosi script in Python di cui questo documento è ricco, offrono al lettore uno spunto ulteriore per considerare il concetto di Network Programmability ed implementarlo nella propria rete aziendale. Al giorno d'oggi, configurare una rete complessa manualmente non è più un'opzione.

La possibilità di implementare topologie di rete complesse all'interno di un sistema virtuale come GNS3 permette di testare soluzioni multiple sfruttando dispostivi di cui non si dispo-

STUDENTSUPSI

ne fisicamente. Per queste ragioni, GNS3 si rivela uno strumento ottimo per configurare, testare e validare switches e routers di brands diversi, come Arista Networks e Cisco Systems.

Nell'ultima sezione si è esaminata una rete privata che utilizza la tecnologia MPLS per inoltare i pacchetti IPv4. Sfruttando le tecnologie descritte nei primi capitoli, tale topologia di rete è stata costruita in modo dinamico: qualsiasi customer (*fisico*) può essere spostato nella rete privata (*virtuale*) desiderata. In questo modo vi è la possibilità di gestire da remoto una rete reale che in base alle esigenze può cambiare, e questo cambiamento avviene ora via software.

STUDENTSUPSI