ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

IMPLEMENTAZIONE DI INTERCETTAZIONE TELEMATICA IN RETI VIRTUALI DEFINITE VIA SOFTWARE

Tesi in
Reti di Calcolatori

Relatore
Prof. FRANCO CALLEGATI

Presentata da
GIULIO CRESTANI

Correlatore
Prof. WALTER CERRONI

SESSIONE III
ANNO ACCADEMICO 2015/2016

Alla mia famiglia ed ai miei amici.
Le persone a cui tengo di più.

PAROLE CHIAVE

SDN

NFV

OpenVirteX

Security

Indice

Abstract

Negli ultimi anni, all'interno delle reti, vengono effettuate delle azioni "di controllo" che, ormai, sono all'ordine del giorno. Per esempio, nel caso in cui venga posto in essere il sospetto di attività illecite da parte di un individuo, la polizia postale attua delle intercettazioni su un router che si attesta nella zona limitrofa rispetto a dove si trova questo individuo, piuttosto che sulla rete mobile. Le installazioni di questi dispositivi, oggigiorno, avvengono mediante l'intervento fisico che agisce negli switch o nei router i quali si interpongono tra i vari flussi di traffico della rete e che quindi, vengono ritenuti significativi. Su questi ultimi ci si connette fisicamente per poter abilitare certe azioni o in modo da fare delle operazioni di controllo. Queste sono tutte manovre che avvengono su un certo oggetto o su alcuni apparati sui quali, nel caso in cui ci sia bisogno di modificare qualche impostazione o configurazione, si deve ritornare a rimettere mano, sempre fisicamente, per modificarli.

Se si volesse estremizzare ulteriormente, si potrebbe dire che la gestione della rete in generale funziona come si è appena descritto. Ma, grazie al paradigma SDN si semplifica questa operazione perché, in pratica, la si remotizza. E quindi la domanda che ci si pone è: "*si possono egualmente remotizzare tutte quelle operazioni simili a quelle sopra descritte?*

Dietro a tali operazioni, vi sono questioni controverse come il fatto che tutte queste debbano essere autorizzate da qualcuno ed essere attuate rispettando certi termini e criteri. Concretamente, non si ritiene opportuno demandare ad un tecnico il compito di eseguire questo tipo di operazioni; piuttosto sarebbe più indicata una persona con un certo grado di competenza, esperienza e consapevolezza delle procedure. Se così non fosse potrebbe accadere che, nel caso di qualche tipo di errore da parte del tecnico, invece di rilevare il traffico di un impiegato, si vada a sniffare quello di un dirigente che, chiaramente, non è la stessa cosa.

Quindi, l'azione gestionale di una rete fatta giorno per giorno e l'azione

gestionale fatta con questi scopi di controllo, devono essere due processi completamente diversi. Ciò rimane invariato anche nel presente ma, stando ad una prospettiva futura in cui sopraggiungerà il paradigma SDN, saranno necessari maggiori analisi e controllo. Una possibile risposta a tutto questo potrebbe essere l'utilizzo, appunto, delle tecnologie abilitanti la virtualizzazione di rete come *FlowVisor* e *OpenVirteX*. Queste, permettono a più soggetti in parallelo (i controller in SDN, che sono logicamente simili), di gestire, come meglio credono, la rete sottostante. Più in particolare, si vuole fare in modo che un altro individuo possa eseguire azioni sul traffico gestito da uno di questi soggetti, in modo che non arrivi ad accorgersi della presenza dell'individuo. In pratica, all'interno della rete è presente una sorta di *superuser* che, nel caso ce ne fosse bisogno, può intervenire applicando azioni che normalmente gli altri utilizzatori non riescono a compiere come quelle di monitoraggio o di sicurezza.

Ovviamente questo non è un lavoro limitato solamente a queste funzionalità, ma può essere pensato come l'inizio di un progetto molto più vasto che arriverà ad includerne altre. Oltre alle funzionalità di sicurezza e controllo, questo *superuser* potrebbe implementare, sulla base della soluzione che viene discussa in questa tesi, funzionalità per *fault tolerance*, prevenzione dei guasti, *load balancing*, miglioramento delle performance della rete, consentendo l'uso di un numero minore di dispositivi di rete.

Ricapitolando, l'obiettivo centrale della tesi, è fare in modo di ottenere un sistema che riesca a controllare il flusso di dati di un particolare individuo all'interno di una rete SDN senza che nessun utilizzatore di quest'ultima ne sia a conoscenza. Per quanto riguarda la struttura del presente lavoro, nel Capitolo 1, si andranno ad introdurre i concetti principali delle *Software Defined Networks* e *Network Function Virtualization* che stanno alla base di tutto il futuro lavoro e, nel Capitolo 2, si illustrerà la tecnologia *OpenVirteX* la quale permette di creare e gestire uno scenario come quello descritto, implementando casi di studio ed analizzando il loro funzionamento. Nel Capitolo 3 si studierà in modo più approfondito questa tecnologia, considerando cosa avviene al suo interno e come si attua la virtualizzazione della rete. Si è voluto fare questo approfondimento anche per capire se sarà veramente possibile utilizzare questa tecnologia per lo scopo indicato. Successivamente nel Capitolo 4 sarà presentata *PySolv*, la soluzione che consentirà di mettere in atto le funzioni di controllo. Si esporrà il suo funzionamento con le relative modifiche apportate al sistema già presente per interfacciarsi con la soluzione. Infine, nel Capitolo 5, saranno presenti tutte le conclusioni che sono state tratte, indicando inoltre le scoperte fatte durante

lo sviluppo della tesi ed illustrando quali potrebbero essere eventuali evoluzioni in merito.

Capitolo 1

Introduzione

In questo primo capitolo si andranno ad illustrare due concetti centrali ovvero: *Software Defined Networks* (SDN) descritto nella Sezione 1.1 e *Network Function Virtualization* (NFV) descritto nella Sezione 1.2.

In particolare, il focus argomentativo riguarda l'ambito delle reti di calcolatori, le quali permettono lo scambio o la condivisione di dati e risorse tra diversi host. Come è noto a livello informatico, al giorno d'oggi l'architettura di una rete è composta da diversi componenti tra i quali spiccano *host*, *switch* e *router* interconnessi per mezzo di una qualche forma di collegamento o *link*. Questi componenti comunicano tra di loro grazie all'uso dei protocolli di rete che vengono visti come la parte software dell'architettura. Quindi, nel caso si presentasse un malfunzionamento all'interno di una rete o nel caso in cui si volesse cambiare la gestione di quest'ultima, quello che si dovrebbe fare è andare a mettere mano a questi dispositivi in modo da riprogrammarli per ottenere il risultato desiderato. Inoltre, in un mondo in continua evoluzione, crescono tutte quelle che sono le esigenze in termini di archiviazione, dinamicità e scalabilità nell'ambito del computing; ciò si traduce nella necessità di un ambiente più moderno. Questo può essere creato disaccoppiando e disassociando la parte del sistema che prende le decisioni circa l'indirizzamento del traffico dalle parti sottostanti che spediscono il traffico verso la destinazione desiderata.

SDN è un concetto che rivoluziona la concezione di reti di telecomunicazioni avuta fino a poco tempo fa. Esso permette di suddividere il piano di trasporto dei dati da quello di controllo, definito via software. Inoltre consente agli operatori di rendere le proprie reti facilmente gestibili, riprogrammabili e personalizzabili, oltre a liberarle dal cosiddetto "vendor lock-in", ossia la situazione di dipendenza

da un unico fornitore. Per tale motivo gli operatori, i produttori di apparati di rete e la comunità accademica stanno dedicando un'attenzione sempre maggiore a questo argomento. Una delle principali sfide tecnologiche dell'approccio SDN riguarda la centralizzazione della logica del controllo. Ciò abiliterebbe le applicazioni ad acquisire una vista astratta della rete, come se questa fosse governata da un piano di controllo concettualmente centralizzato; diventa quindi possibile implementare logiche di controllo, astraendo dalla complessità fisica della molteplicità degli apparati di rete.

L'approccio appena descritto si presta in modo ottimale per abilitare il secondo concetto: *Network Functions Virtualization* (NFV). Esso nasce dalla forte richiesta di virtualizzazione di tutti quei servizi di networking, a causa del costo e del mantenimento dell'hardware. Questo concetto cambia radicalmente il modo in cui l'insieme delle reti è implementato e gestito, aumentando l'efficienza e rendendo la rete più flessibile ed affidabile. Qualsiasi funzione di rete che richieda un hardware dedicato può essere virtualizzata e distribuita come un applicativo software (denominate VNF - *Virtual Network Function*), con la possibilità di spostarle da un server fisico ad un altro, oltre a fornire servizi personalizzati agli utenti semplicemente rimodulando il software di gestione degli apparati.

Quindi SDN ed NFV rappresentano due approcci complementari, e per molti versi interdipendenti, destinati a trarre beneficio da una loro integrazione nell'evoluzione della rete. Mentre il principale obiettivo di NFV è la realizzazione in modalità virtualizzata delle funzionalità di rete, le tecnologie SDN si candidano a giocare un ruolo fondamentale nel fornire la flessibilità nel controllo e nella programmazione della connettività della rete sottostante. Infatti, le soluzioni NFV richiedono di essere supportate da meccanismi potenti ed efficienti per la gestione dinamica della connettività, sia sul piano fisico che virtuale, per collegare tra di loro le funzionalità di rete virtualizzate (VNF). Questo è proprio il ruolo a cui la tecnologia SDN si presta naturalmente. Il controllo flessibile e dinamico della connettività e dell'inoltro del traffico attraverso la rete, può sfruttare la programmabilità introdotta dall'architettura, che consente di supportare in modo efficiente e generalizzato i requisiti di policy routing, ovvero la possibilità di controllare il percorso dei flussi di traffico.

1.1 Software Defined Networks

Negli ultimi anni diversi requisiti di rete (come ad esempio *QoS, VLANs, ACLs*) hanno reso il piano di controllo troppo complicato e poco agevole da aggiornare.

Secondo McKeown[1], l'obiettivo principale di SDN è ristrutturare l'architettura di networking, introducendo opportuni livelli di astrazione in grado di operare una trasformazione simile a quanto già avvenuto nel campo delle architetture elaborative. Nell'ambito del computing, infatti, ormai da molto tempo i programmatori sono in grado di implementare sistemi complessi senza dovere gestire le tecnicità dei singoli dispositivi coinvolti o senza interagire in linguaggio macchina, il tutto grazie all'introduzione di opportuni livelli di astrazione nell'architettura.

Essenzialmente si parte dall'idea di suddividere il piano di controllo e di gestione della rete da quello del processamento dei pacchetti tra diversi dispositivi, in contrasto con quanto accade nelle reti attuali, dove un router si occupa di popolare delle strutture dati con algoritmi e protocolli che verranno poi utilizzate dallo stesso per l'inoltro dei pacchetti.

A seguito di questa sezione si analizzerà quella che è l'architettura base del paradigma SDN, descrivendone i principali componenti e il relativo protocollo che sta alla base di questo concetto rivoluzionario.

1.1.1 Architettura

SDN è un approccio che consente agli amministratori di rete di gestire e controllare automaticamente e dinamicamente un gran numero di dispositivi di rete, di servizi, di topologie, di percorsi di traffico e gestione dei pacchetti, attraverso l'uso di linguaggi ad alto livello. Nella parte sopra della Figura 1.1 è possibile osservare come viene gestita una rete al ad oggi. È necessario che gli algoritmi che gestiscono il flusso dei pacchetti siano distribuiti ed eseguiti tra i vari switch vicini nella rete. Questo algoritmo di controllo risulta essere in alcuni casi complicato da gestire. Nella parte inferiore della figura, invece, viene mostrato come una rete complicata può essere semplificata attraverso l'uso dell'approccio SDN. In questa rete è presente un software che implementa un algoritmo general-purpose in esecuzione su un server che funge da *Network Operating System*. Questo software interagisce con ognuno degli switch della rete per determinare la topologia fisica.

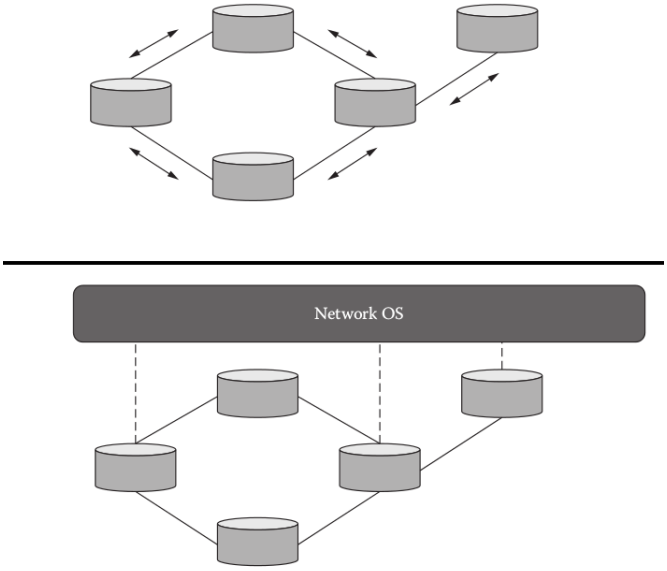


Figura 1.1: **Sopra**: meccanismi di controllo tradizionale. **Sotto**: SDN. Fonte: [3, p. 35].

Una volta osservata la topologia, il sistema operativo di rete è in grado di creare e fornire una vista globale della rete. Successivamente sarà presente un programma di controllo che lavorerà ad un livello di astrazione più alto in modo da impartire decisioni circa l'inoltro dei pacchetti (o altre funzionalità), facendo uso della conoscenza circa la topologia di rete sottostante. Questo programma di controllo sarà eseguito all'interno del Network Operating System ed userà delle API per interagire con la vista globale della rete. La Figura 1.2 riassume questo nuovo elemento.

Questo programma di controllo, denominato *controller* nel gergo SDN, è una delle componenti fondamentali di tutta l'architettura. Nella sottosezione 1.1.2 verrà descritto in maniera più approfondita insieme agli altri componenti che stanno alla base di questo paradigma.

1.1.2 Componenti principali

Sono presenti tre componenti principali: Controller, Switch e Host.

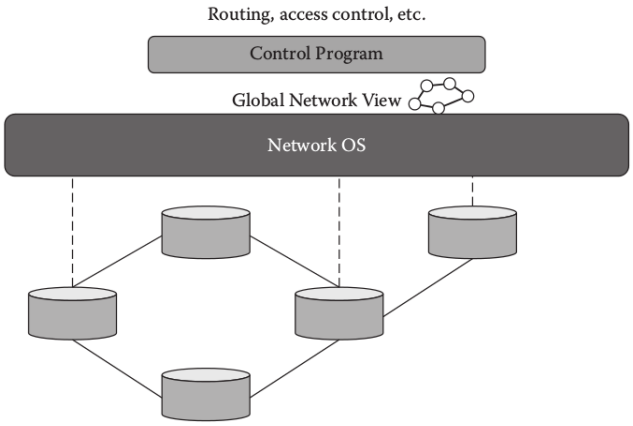


Figura 1.2: Riassunto architettura SDN completa. Fonte: [3, p. 36].

1. **Controller**: ha un ruolo fondamentale all'interno dell'architettura: governa la rete controllando se i flussi di traffico possono essere immessi e quali devono essere le loro caratteristiche. Fondamentalmente è un'applicazione software che si trova tra i dispositivi di rete e le applicazioni. Tutte le comunicazioni tra applicazioni e dispositivi devono passare attraverso il controller. Esso utilizza protocolli come OpenFlow per configurare i dispositivi di rete e scegliere il percorso ottimale per il traffico da e verso le applicazioni. Come accennato in precedenza, esso è a tutti gli effetti un sistema operativo di rete; esporta il piano di controllo fuori dai dispositivi e lo esegue come software. In questo modo facilita la gestione della rete automatizzandola e rendendola più facile da integrare e gestire da parte delle applicazioni di business. Un altro aspetto interessante è che le applicazioni vedono la rete come un'unica entità con la quale è possibile interagire attraverso delle API indipendenti dall'hardware.

2. **Switch**: sono i dispositivi di rete che inoltrano i vari flussi di pacchetti. Essi sono costituiti da delle *Flow Table* che contengono un insieme di entry usate per fare matching tra i vari campi dei pacchetti e per processare quelli che transitano. Oltre a queste tabelle è presente un canale sicuro che permette allo switch di comunicare con il controller e, quest'ultimo, lo gestisce permettendo ai comandi ed ai pacchetti di essere

scambiati tramite il protocollo OpenFlow (descritto nella sottosezione 1.1.3), che fornisce un'interfaccia standard di comunicazione. In breve, questo protocollo permette al controller di comunicare con gli switch permettendo di aggiungere rimuovere o modificare le entry delle varie flow table.

3. **Host**: normali terminali connessi alla rete. Più precisamente uno switch è connesso ad un certo numero di host e gestisce il traffico da e verso di essi.

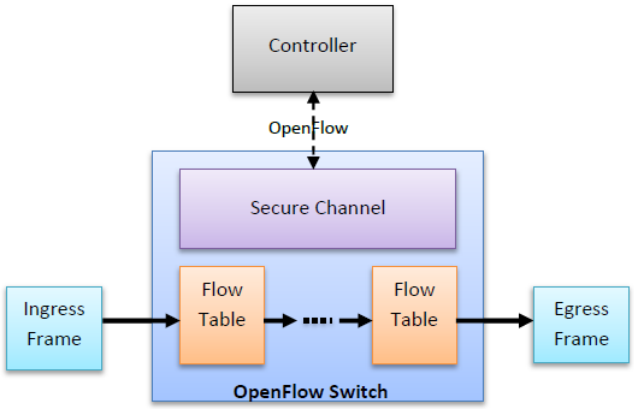


Figura 1.3: Struttura di uno switch.

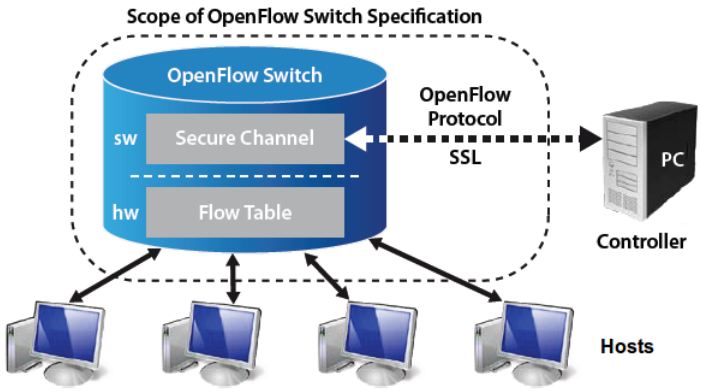


Figura 1.4: Struttura intera dell'architettura SDN.

1.1.3 Il protocollo OpenFlow

Questo protocollo è stato implementato principalmente per essere usato in coppia con la tecnologia SDN. Viene sfruttato come mezzo per l'astrazione del piano di controllo da quello dei dati, fornendo i livelli di astrazione richiesti tra i dispositivi di controllo e quelli di forwarding. Quindi, il protocollo OpenFlow si rivela uno dei pilastri nella definizione stessa del paradigma SDN.

Grazie all'uso di questo protocollo è possibile migliorare la Quality of Service attraverso il filtering del traffico oppure per motivi di network monitoring (ad esempio usare il controller o un altro host della rete come un dispositivo di monitoring).

L'interfaccia realizzata da OpenFlow si colloca al livello più basso di astrazione previsto dall'architettura SDN. Essa permette infatti di svincolarsi dall'hardware di inoltro dei pacchetti. Scopo di questo protocollo è quindi, quello di presentare all'esterno un modello di nodo generale e unificato, rendendo gli strati più alti dell'architettura di rete indipendenti dall'implementazione del particolare vendor delle tecnologie impiegate nel piano di forwarding. Risalendo alle origini della proposta, l'idea di base di OpenFlow è quella di rendere programmabili in senso generale, le tabelle di classificazione ed instradamento dei pacchetti presenti negli apparati di networking (siano essi router o switch); in questo modo il contenuto (le cosiddette *entry*), può essere configurato dalle applicazioni attraverso un piano di controllo esterno al dispositivo, mediante un'opportuna interfaccia. Quest'ultima, costituita appunto dal protocollo OpenFlow, permette di definirne in modo flessibile il contenuto, in funzione della logica di servizio da realizzare.

Da un punto di vista più applicativo, il protocollo OpenFlow è:

- Un modello per la gestione dei pacchetti all'interno degli switch. Per essere di tipo OpenFlow, l'implementazione dello switch deve essere in grado di sopperire le regole obbligatorie di tale modello.

- Un protocollo per informare su ciò che uno switch OpenFlow dovrebbe fare o che ha fatto con i pacchetti che sono passati o che passeranno attraverso di esso.

Il modello di gestione dei pacchetti non è altro che una sequenza di punti all'interno di una tabella che indicano un mapping tra una certa tipologia di pacchetti e la relativa azione da eseguire a fronte dell'arrivo di quella tipologia. Più precisamente, in ognuno di questi punti lo switch confronta un insieme di

valori dall'header dei pacchetti (e spesso anche l'*ingress port*), con una tabella di regole. Quando c'è un match con una di queste regole, la tabella indica allo switch quale azione eseguire come, ad esempio, inviare il pacchetto ad una certa *egress port* o rifiutare l'inoltro.

Nel modello del nodo OpenFlow questa tabella viene denominata *Flow Table* e specifica le regole associate a ciascun flusso di traffico. L'entità base con cui viene rappresentato e gestito il traffico in OpenFlow è per l'appunto il *flusso* di pacchetti (*flow*); quest'ultimo è ottenuto dalle regole di cui si parlava in precedenza, specificando il contenuto di opportuni campi dell'intestazione. Il protocollo OpenFlow permette quindi al piano di controllo di definire, in modo flessibile e dinamico, le regole di instradamento e di trattamento dei pacchetti appartenenti ai diversi flussi di traffico.

1.2 Network Function Virtualization

Prima di entrare nel merito della virtualizzazione delle reti si vogliono passare in rassegna i concetti principali analizzati nell'ambito della virtualizzazione di macchine fisiche.

Hardware Virtualization In informatica il termine macchina virtuale (VM) indica un software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale che emula tipicamente il comportamento di una macchina fisica grazie all'assegnazione di risorse hardware (porzioni di disco rigido, RAM e CPU), ed in cui alcune applicazioni possono essere eseguite come se interagissero con tale macchina; infatti se dovesse andare fuori uso il sistema operativo attivo sulla macchina virtuale, il sistema di base non ne risentirebbe affatto.

Grazie a queste macchine virtuali e al concetto più generale di virtualizzazione si possono ottenere diversi vantaggi tra i quali: *aumento dell'affidabilità del sistema*: grazie all'indipendenza delle macchine virtuali e quindi non saranno presenti conflitti tra i vari servizi; *consolidamento dei server*: riducendo il numero di macchine fisiche richieste grazie alla possibilità di ospitare più macchine virtuali all'interno di una fisica; *disaster recovery*: grazie alla possibilità di salvare il sistema operativo "guest" e ripristinarlo in un secondo momento, riducendo i periodi di indisponibilità; *esecuzione di applicazioni legacy*: nel caso in cui un'azienda utilizzi applicazioni sviluppate per sistemi operativi che girano su hardware ormai obsoleto, non supportato o addirittura introvabile.

Il componente più importante nel concetto della virtualizzazione è l'*HyperVisor*. Esso è un programma che permette alle diverse macchine virtuali di condividere l'hardware presente sul singolo host dove sono installate. Ogni singola macchina crede di avere CPU, memoria ed altre risorse tutte per sé e questo grazie al lavoro di questo componente. Esso controlla le risorse ed i processori della macchina fisica, allocandole in base alle richieste dalle varie macchine ospiti e facendo attenzione che non interferiscano tra loro.

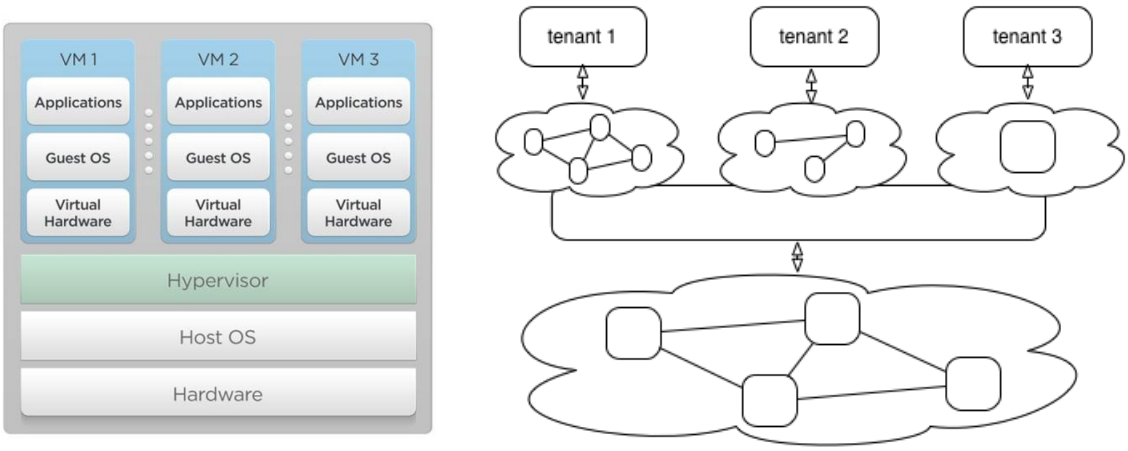
Network Virtualization L'incremento della richiesta di servizi da parte degli utenti della rete, come ad esempio la radicale crescita del traffico nelle reti di telecomunicazioni o come le nuove tecniche di comunicazione che fanno in modo di aumentare il numero di utenti sempre connessi (i cosiddetti *always-on users*), ha costretto gli operatori a cercare rapidamente delle nuove tipologie di servizi in modo da rinforzare l'efficienza e la qualità di questi ultimi, a discapito di una significativa riduzione in termini di CAPEX[1] e OPEX[2]. La nascita della NFV è quindi un diretto risultato di questa domanda.

NFV mira a trasformare il modo in cui gli operatori progettano le reti. Utilizza la tradizionale virtualizzazione dei server ma estendendo significativamente tale concetto. In questo caso i componenti che si vanno a virtualizzare sono i più diffusi dispositivi di rete. Piuttosto che avere hardware dedicato a fornire delle date funzioni, viene sfruttato un software eseguito su un server. Così facendo, un pull di macchine virtuali sarà utilizzato ad un livello superiore dei server fisici, fornendo a questi ultimi tutte le funzionalità di switch o router e le relative funzioni come ad esempio firewall o load balancing.

Si consideri un utente con il bisogno di una specifica funzione di rete. Prima dell'avvento della NFV, il tempo impiegato per soddisfare la sua richiesta era pressoché illimitato. Si doveva spedire il relativo hardware verso il posto indicato per poi installarlo in modo che funzionasse a dovere. Inoltre, se le richieste dell'utente fossero cambiate con l'aumentare del tempo e le funzioni del dispositivo non fossero più state richieste, allora il dispositivo si sarebbe rivelato inutile. Ora invece, è possibile sviluppare rapidamente una VNF all'interno di un server ed usare quest'ultima invece di agire fisicamente sulla

[1] *CAPital EXpenditure* ovvero i fondi che un'azienda impiega per acquistare delle risorse durevoli. Fanno parte di questa tipologia di risorse i fondi per gli apparati di rete, per la costruzione di edifici, aggiornamenti software, ecc.

[2] *OPerating EXpenditure* ovvero tutti quei costi per necessari per gestire un prodotto quindi i costi operativi e di gestione.



(a) Hardware virtualization. (b) Network virtualization.

Figura 1.5: Confronto tra due tipi di virtualizzazione.

rete. Questo approccio garantisce maggiore flessibilità, scalabilità e agilità assicurando inoltre minore CAPEX e OPEX.

Nella network virtualization si ha una singola topologia di rete fisica composta da nodi e switch, che viene partizionata tra le varie virtual networks, delegando la gestione di ognuna di esse ad una stessa o a diverse entità. Così come le varie virtual machine sono indipendenti tra loro e non hanno accesso l'una ai dati dell'altra, allo stesso modo l'amministratore di una singola virtual network non può avere accesso al traffico di un'altra virtual network che non controlla anche se condividono gli stessi collegamenti fisici della topologia di rete. Ovviamente, così come avviene nella virtualizzazione classica con l'hardware, l'utilizzatore della virtual network ha l'illusione di avere tutta la rete a sua disposizione. In Figura 1.5 è possibile confrontare le due strutture di virtualizzazione appena descritte.

La network virtualization, ovvero l'astrazione della rete la quale è disaccoppiata dall'infrastruttura fisica sottostante, è un caso d'uso di rilevante importanza per il paradigma SDN. Quest'ultimo permette alle diverse virtual network di venire eseguite sulla stessa infrastruttura ed ognuna di esse può avere una topologia più semplice rispetto alla rete fisica sottostante. Il punto focale che riguarda queste ricerche evidenzia il fatto che SDN può essere visto come la tecnologia abilitante la network virtualization. Questo perché, grazie al paradigma SDN, ai proprietari delle infrastrutture di rete viene fornita una

gestione più semplice della complessità delle reti e, allo stesso tempo, possono personalizzare le reti degli utilizzatori (i cosiddetti *tenant*), per meglio servire il loro fabbisogno.

Dopo aver introdotto questi due argomenti che stanno alla base del lavoro di ricerca che si è effettuato, si passerà alla descrizione della tecnologia studiata basata su SDN, che al contempo fornisce la base per abilitare i concetti della network virtualization.

Capitolo 2

SDN e NFV - OpenVirteX

Riprendendo l'argomento NFV, come già detto, esso si propone di sostituire i dispositivi di rete tradizionali con dei componenti software installati su commodity server eliminando così la necessità di dispositivi di rete dedicati come router, switch e firewall. Lo spostamento delle funzioni di rete dall'hardware ad hoc al server, consente alle aziende di evitare la proliferazione degli apparati e il sottoutilizzo delle risorse. Questa caratteristica permette al concetto di NFV di accoppiarsi perfettamente con il paradigma SDN. La sinergia di queste due soluzioni consente alla rete di raggiungere le migliori performance. Infatti, SDN fornisce a NFV i vantaggi di una connessione programmabile tra le funzioni di rete virtualizzate mentre, NFV, mette a disposizione del paradigma SDN la possibilità di implementare le funzioni di rete tramite software eseguiti su server. Si ha così la possibilità di virtualizzare più controller SDN in modo che possano essere presenti più *control plane* ognuno dei quali crede di utilizzare l'intera infrastruttura di rete che, invece, viene suddivisa tra gli utilizzatori.

Oggigiorno sono presenti progetti open-source che, basandosi su SDN, permettono di abilitare la network virtualization. In particolare se ne sono visti due: *FlowVisor* e *OpenVirteX* (OVX). Quello che si è deciso di scegliere per il lavoro di ricerca è il secondo. Entrambi i progetti hanno il compito principale di poter creare e saper gestire al meglio una Virtual Network ma, il motivo per cui si è scelto OpenVirteX è che, a differenza di FlowVisor il quale semplicemente suddivide l'intero *flowspace*[1] tra i vari *tenant*[2], questo fornisce ad ognuno di essi una rete completamente virtualizzata con la topologia

[1] L'insieme di flussi che appartengono ad una certa virtual network.
[2] L'utilizzatore della virtual network.

richiesta e con l'intero spazio degli header di pacchetto. Mentre in FlowVisor, quello che accade è che si suddividono i singoli header di pacchetto in diversi sottoinsiemi e vengono assegnati alle varie network. Quindi, questa suddivisione effettuata da FlowVisor è più limitata rispetto alle funzionalità di OpenVirteX. Questo perché tutte le *slice*[3] essenzialmente condividono lo stesso flusso o spazio degli indirizzi e quindi una slice non dispone di uno spazio degli indirizzi completamente separato ed indipendente. Inoltre, FlowVisor non permette ad una singola slice di avere una topologia di rete (virtuale) arbitraria, può offrire solamente un sottoinsieme della topologia di rete fisica.

A seguire verranno descritti in dettaglio la struttura ed il funzionamento di OpenVirteX, illustrando anche il funzionamento nell'ambito SDN con esempi testati e funzionanti.

2.1 OpenVirteX

OpenVirteX[6] è una piattaforma per la network virtualization che assicura all'operatore che la utilizza, la possibilità di creare e gestire delle reti SDN virtuali. Gli utilizzatori (i cosiddetti *tenants*), sono liberi di specificare la topologia, lo schema di indirizzamento della loro rete virtuale SDN ed eseguire il proprio NOS (*Network Operating System*) per controllarla. Quindi OpenVirteX funziona come un proxy OpenFlow posizionato tra la rete dell'utilizzatore e il sistema che la controlla.

L'obiettivo principale è dunque quello di disaccoppiare la rete dalla sua manifestazione fisica per fornire risorse virtualizzate cercando di offrire un forte isolamento tra quelle che sono le varie reti virtuali con l'abilità di migrare, di ottenere snapshot[4] e di personalizzare la topologia in real time.

Principali obiettivi

- Fornire la virtualizzazione degli indirizzi per mantenere separato il traffico dei vari tenant.
- Fornire la virtualizzazione della topologia di rete per fornire ai tenant la possibilità di personalizzarla come meglio credono.

[3] Una slice è una delle suddivisioni che OpenVirteX e FlowVisor sono in grado di gestire. Ogni slice ha un proprio controller che gestisce la sua virtual network.

[4] Un'istantanea dello stato di un sistema in un particolare momento. Può fare riferimento a una copia reale del sistema.

- Consegnare ogni virtual network ai NOS dei tenant come se fosse un'infrastruttura on-demand.

Quindi OpenVirteX permette ai tenant di usare il proprio NOS per controllare le loro risorse di rete corrispondenti alla loro virtual network. In altre parole, *OpenVirteX crea delle virtual SDN multiple che risiedono al di sopra della rete SDN fisica.*

2.1.1 Architettura

OpenVirteX è una piattaforma per la network virtualization in grado di riprodurre reti virtuali con l'utilizzo della tecnologia SDN. Queste reti virtuali devono avere una topologia ed uno schema degli indirizzi, che siano arbitrari e che possano essere configurati per le varie richieste degli utilizzatori. Queste richieste sono intercettate da un tool chiamato *Network Embedder*. Per prima cosa un utente specifica lo schema di indirizzamento e la topologia della rete virtuale all'embedder, il quale genera un mapping dal virtuale al fisico usando le informazioni da OpenVirteX. In seguito questo mapping viene passato a OpenVirteX che, a sua volta, istanzia la rete virtuale sulla topologia fisica (si veda Figura 2.1)

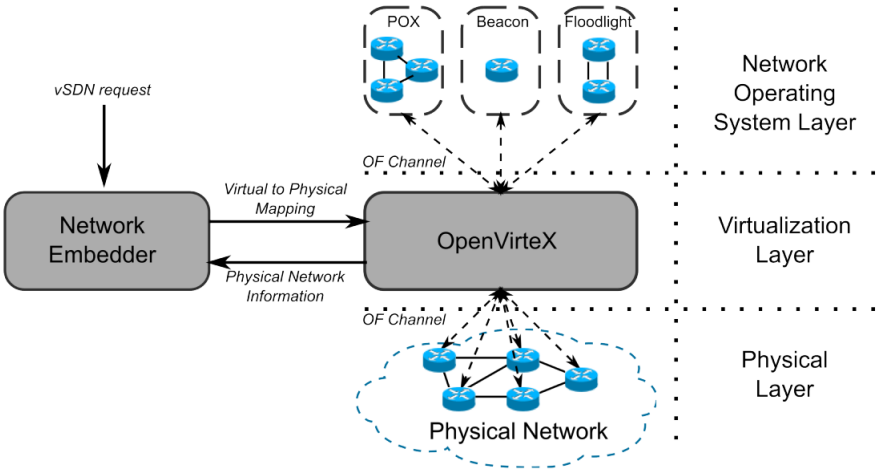


Figura 2.1: Architettura di OpenVirteX. Fonte: [6]

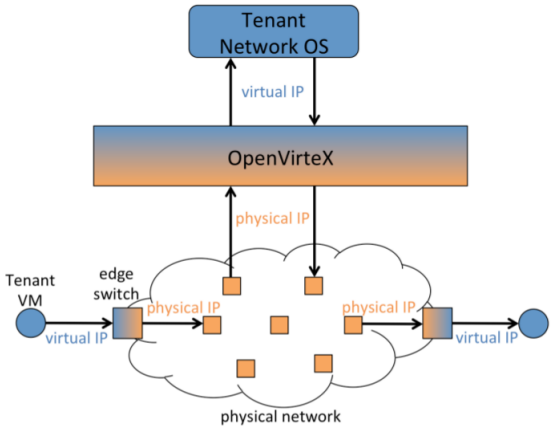
Quindi, OVX esegue un mapping tra il virtuale ed il fisico. Per fare ciò si ha bisogno di considerare alcuni accorgimenti essenziali per il corretto funzionamento. Questi fanno capo alla possibilità di realizzare una virtualizzazione della topologia, degli indirizzi e delle funzioni di controllo.

Virtualizzazione della topologia OpenVirteX permette ai vari utenti di specificare la loro topologia di rete arbitraria. Questa topologia potrebbe essere semplice (come per esempio un unico switch), o più elaborata con diversi percorsi, anche per casi di fault tolerance. In pratica, tutte queste topologie non devono necessariamente corrispondere alla topologia fisica presente nell'infrastruttura di rete fisica, ma piuttosto deve corrispondere a quello che l'utente desidera con l'*unica limitazione* di progetto che un singolo switch fisico non possa essere partizionato in diversi switch virtuali.

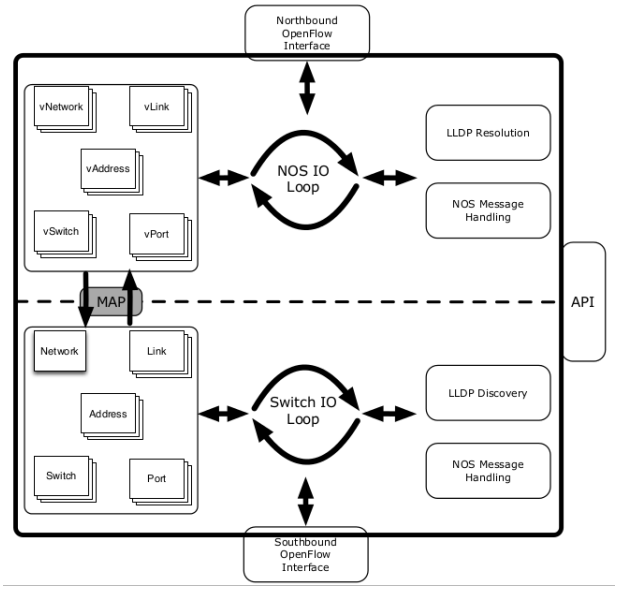
Virtualizzazione degli indirizzi OpenVirteX garantisce ai vari tenant la scelta dell'assegnamento degli indirizzi dei propri end host, permettendo delle sovrapposizioni multiple di indirizzi IP della rete fisica. Per differenziare l'appartenenza di questi host ad una particolare virtual network, OVX genera un ID univoco per ogni tenant e, per ogni host, associa un indirizzo fisico che codifica la sua appartenenza a quel particolare tenant (usando appunto questo ID). Le collisioni tra indirizzi vengono evitate grazie all'installazione di regole per i flussi, le quali riscrivono gli indirizzi negli switch ai confini tra la rete virtuale e quella fisica. Nel caso in cui ci si trovi in uno degli switch "entranti" nella virtual network, quella che viene eseguita è un traduzione dagli indirizzi assegnati ai tenant a quelli fisici e, viceversa, nel caso ci trovassimo nello switch "uscente" dalla virtual network. In Figura 2.2a viene descritto questo passaggio.

Questa riscrittura degli indirizzi IP dei pacchetti fa sorgere, ovviamente, un overhead sul piano dei dati che OVX deve saper gestire. Più in particolare OVX mantiene al suo interno, delle strutture dati in grado di memorizzare il mapping tra la parte fisica e quella virtuale (si veda Figura 2.2b). È evidente però, che uno scenario nel quale tutti gli utenti volessero usare l'intero spazio degli indirizzi IP, non sarebbe supportato. Questo perché alcuni bit dell'header del pacchetto sono usati per codificare l'ID dei tenant. La cosa più importante, però, è che tutta questa procedura sia invisibile agli occhi del NOS.

Virtualizzazione delle funzioni di controllo Ogni virtual network ha la possibilità di avere un proprio NOS. Questo si traduce nella responsabilità di



(a) Traduzione di indirizzi eseguita da OVX.



(b) Architettura interna di OVX.

Figura 2.2: Particolarità di OVX. Fonte [6].

adattamento che OVX ha, ovvero trovare un mapping tra le funzioni descritte nel NOS e le corrispondenti traduzioni sulla rete fisica. In alcuni casi c'è la possibilità che l'informazione richiesta sia relativamente semplice ma, traducendola per la rete fisica, si riveli una concatenazione di più azioni.

Proprietà

Il disaccoppiamento che viene a crearsi tra i componenti fisici e virtuali nella forma di mapping 1:N introduce un'aggiunta di flessibilità. Le libertà introdotte da questa flessibilità possono essere prese riguardo al modo in cui i componenti virtuali sono mappati nelle loro controparti fisiche. Queste libertà sono importanti per alcune proprietà chiave:

- **Personalizzazione della topologia**: un link virtuale potrebbe passare attraverso dei collegamenti multipli mentre gli switch virtuali potrebbero astrarre una parte o, addirittura, l'intera rete.
- **Resilienza**: un link o switch virtuale può essere mappato in diversi componenti fisici per assicurare ridondanza. Un link virtuale *resiliente*

è caratterizzato da percorsi fisici multipli tra i due endpoint della rete. Uno switch virtuale che astrae parte della della rete fisica, invece, sfrutta la ridondanza della topologia (con, ad esempio, percorsi multipli), per fornire diversi percorsi tra le sue porte.

- **Persistenza**: ogni rete ed ogni elemento della rete mantiene un insieme di attributi al suo interno. Questo abilita OpenVirteX a memorizzare delle informazioni in una zona persistente, in modo che la rete virtuale possa essere spenta e ricreata in un secondo momento. Inoltre, questa abilità di "salvare" e recuperare la rete è fondamentale per la funzionalità di snapshot.

2.1.2 Componenti principali

Una volta descritta l'architettura generale di OVX si passa più nel dettaglio alla spiegazione dei componenti che gestisce e di cui mantiene una riproduzione al suo interno. Le rappresentazioni delle reti di OVX (sia virtuale che fisica), non sono altro che un insieme di switch, porte, link, host e indirizzi. Dal punto di vista di un tenant, questi componenti appaiono del tutto uguali all'attuale configurazione del datapath, delle porte, dei link e degli host non avendo conoscenza della rete fisica sottostante.

La tecnologia OVX è implementata in Java ed i componenti sono definiti come classi. In particolare ogni componente è definito attraverso una classe base. Nella Tabella 2.1 vengono specificate, oltre alle classi base e quello che rappresentano, le classi delle parti fisiche e virtuali che implementano la classe base rispettiva. La convenzione che si nota nella maggior parte dei componenti è iniziare il nome con il prefisso `Physical`, nel caso di componenti fisici, e `OVX` nel caso dei virtuali.

OVX basa la sua conoscenza della rete fisica popolando l'istanza di `PhysicalNetwork`. Questa istanza è popolata da altre istanze dei componenti che ne fanno parte: switch, link e porte appartenenti all'infrastruttura. Le istanze di `OVXNetwork` sono create attraverso l'uso delle API utilizzate dall'operatore, le quali creano gli oggetti della rete e li mappa nei componenti dell'istanza di `PhysicalNetwork`.

Tabella 2.1: Componenti principali e classi.

Classe base	**Rappresentazione**	**Classe comp. fisici**	**Classi comp. virtuali**
Network	Intera topologia di rete	PhysicalNetwork	OVXNetwork
Switch	Uno switch	PhysicalSwitch	OVXSwitch OVXSingleSwitch OVXBigSwitch
Port	Porta di uno switch	PhysicalPort	OVXPort
Link	Connessione tra due porte	PhysicalLink	OVXLink SwitchRoute
Host	Host della rete	-	Host
IPAddress	Indirizzo IP	PhysicalIPAddress	OVXIPAddress

State Machine dei componenti

Gli elementi della rete sono associati e dipendono dai vari stati di altri elementi della rete stessa. Per esempio se uno switch della rete è spento, allora tutte le porte e tutti i link da e verso queste ultime sono "spente" cambiando la topologia di rete percepita da OVX. Ogni componente possiede 4 stati:

1. **INIT**: componente appena creato.
2. **INACTIVE**: estraneo agli eventi della rete.
3. **ACTIVE**: stato normale di operatività. Gli eventi sono gestiti come ci si aspetta.
4. **STOPPED**: componente distrutto. Il componente deve essere ricreato per essere usabile.

Mentre i metodi per passare da uno stato all'altro sono:

- **register()[INIT → INACTIVE]** Esempio: una nuova porta non può essere creata se lo switch al quale è associata non esiste.
- **boot()[INACTIVE → ACTIVE]** Esempio: attiva il link tra due porte se queste ultime sono attive.

- **teardown()[ACTIVE → INACTIVE]** Esempio: Se una porta viene disabilitata allora i link da e verso questa sono disabilitati.
- **unregister()[INACTIVE → STOPPED]** Esempio: Se un link non ha motivo di essere ancora attivo allora deve essere fermato se una o entrambe le porte che connette sono ferme.

Riassunto in Figura 2.3.

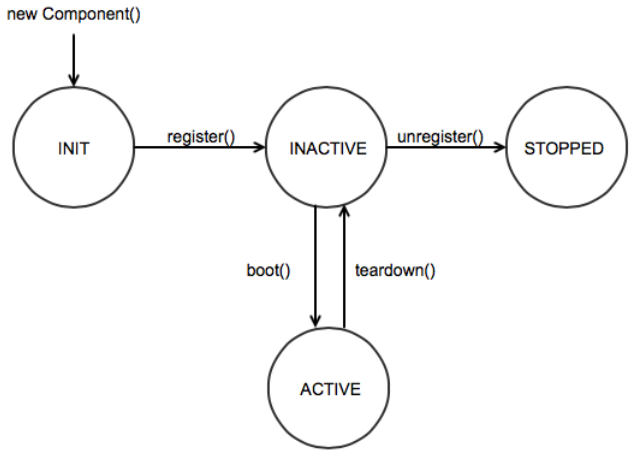


Figura 2.3: Diagramma degli stati con i vari passaggi.

Gli switch

Uno switch viene identificato univocamente attraverso uno switch ID (DPID). Inoltre è caratterizzato da un insieme di porte e dai suoi attributi e capacità. Dall'elemento base, OVX implementa 2 tipi principali di switch:

- *PhysicalSwitch*: rappresenta uno switch all'interno della rete fisica e che è connesso con OVX.
- *OVXSwitch*: è la rappresentazione di uno switch vituale interno a OVX, quindi gli switch che percepiscono i controller dei tenant. OVX mantiene al suo interno le flow table virtuali per ognuno di questi switch. Questa classe ha due classi figlie le quali rappresentano i due modi in cui è possibile virtualizzare switch:

  - *OVXSingleSwitch*: switch virtuale mappato in un singolo switch fisico.
  - *OVXBigSwitch*: switch virtuale mappato in più switch fisici. Esso mantiene al suo interno una *route table* per i percorsi fisici al suo interno.

Le porte

Rappresentate dalla classe base *Port*. Ha un attributo che si vuole evidenziare: `isEdge`. Questo booleano indica, se true, che la porta in questione non ha link[5] connessi ad essa e quindi è una porta connessa ad un host. OVX implementa le seguenti tipologie di porte:

- *PhysicalPort* rappresenta una porta all'interno di uno switch fisico. Il numero di porte e le loro proprietà sono indicate nel messaggio di *Feature Reply* che uno switch scambia con il controller (in questo caso OVX). Un parametro interessante in questo caso è `ovxPortMap` che mantiene il mapping tra sè stessa e le varie porte virtuali. Ogni PhysicalPort mappa al più una porta per ogni virtual network.
- *OVXPort*: è una porta virtuale che risiede all'interno di un OVXSwitch.

Link e percorsi

OVX rappresenta i *link* come tutte le interconnessioni tra switch e porte. Esso è definito da 2 endpoint: uno switch ed una porta sorgente ed uno switch ed una porta di destinazione. Sono presenti tre classi che estendono quella base e descritte di seguito.

- *PhysicalLink*: connette due PhysicalSwitch attraverso le PhysicalPort e rispecchiano i link della rete fisica.
- *OVXLink*: connette due OVXSwitch attraverso le OVXPort di una stessa rete virtuale. Un OVXLink potrebbe essere mappato all'interno di uno o più PhysicalLink adiacenti.
- *SwitchRoute*: connette due OVXPort all'interno dello stesso OVXBigSwitch. Uno SwitchRoute può essere mappato all'interno di uno o più

[5]Per link si intende una connessione tra due switch virtuali.

PhysicalLink adiacenti. In questo caso si vanno a definire due tipi di endpoint:

- *ingress/egress port*: OVXPort visibili al tenant come porta appartenente allo switch;
- *SwitchRoute endpoint*: sono PhysicalPort interne al BigSwitch ai capi di una catena di PhysicalLink.

In Figura 2.4 vengono riassunte graficamente le varie tipologie di *Link*. Nell'immagine in alto due *PhisicalLinks* connettono delle *PhysicalPort* (cerchi bianchi) attraverso tre *PhysicalSwitch* (`ps1`, `ps2`, `ps3`). Nell'immagine in mezzo un *OVXLink* connette due *OVXSwitch* (`vs1`, `vs2`), attraverso delle *OVXPort* (cerchi neri); l'*OVXLink* viene mappato attraverso due *PhysicalLink*. Infine, nell'immagine di sotto è presente uno *SwitchRoute* che connette due OVXPort all'interno dello stesso *BigSwitch*. Esso è caratterizzato da endpoint esterni (*OVXPort*) ed endpoint interni (*PhysicalPort*).

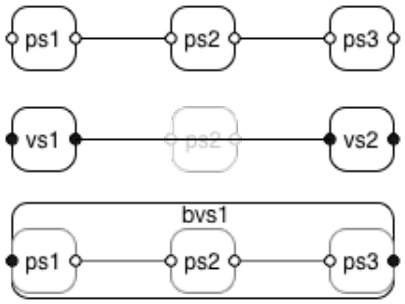


Figura 2.4: Tipologie di Link.

Gli indirizzi

Gli indirizzi IP sono rappresentati come interi formattati, associati ad un host della rete. OVX identifica due tipi di indirizzi IP usati nel processo di virtualizzazione ed indicato a seguire.

- *OVXIPAddress*: gli indirizzi IP associati agli host della rete virtuale e assegnati da un mezzo esterno ad OVX. Quindi questo indirizzo è unico all'interno di una rete virtuale, ma due reti virtuali possono utilizzare

questo stesso indirizzo. Ad esempio è permesso che due host distinti appartenenti a due reti diverse utilizzino l'indirizzo IP virtuale 10.0.0.5.

- *PhysicalIPAddress*: assegnati direttamente da OVX e sono unici all'interno di tutte le reti virtuali.

Gli host

Rappresentano gli endpoint dei flussi di traffico in una rete virtuale. Gli host di OVX sono rappresentazioni virtuali che non hanno la separazione tra fisico e virtuale. La rappresentazione fisica di un host è generata cercando l'indirizzo di rete ed il punto al quale è collegato fisicamente.

Le networks

Le network raccolgono insieme tutti questi componenti. La classe memorizza i vari mapping che descrivono le relazioni tra switch, link, porte ed host. I parametri con cui interagisce sono:

```
//T1: switch, T2: porte, T3: link

// set of all switches found in this network
protected final Set<T1> switchSet;

// set of all links within this network
protected final Set<T3> linkSet;

// mapping between switches and their DPIDs
protected final Map<Long, T1> dpidMap;

// mapping between ports that are endpoints of the same link
protected final Map<T2, T2> neighborPortMap;

// mapping between a switch and its adjacencies
protected final Map<T1, HashSet<T1>> neighborMap;
```

2.2 Esempio d'uso I

Nel seguente paragrafo si osserverà un prospettiva applicativa. Dopo aver illustrato l'architettura e aver descritto i componenti, si mettono in pratica

le funzionalità di OVX testando alcune reti virtuali. Successivamente viene decritta una topologia molto semplice illustrando tutti i vari comandi e passaggi svolti. In Appendice B, viene invece descritta una topologia più articolata, con un numero maggiore di switch. La spiegazione di tutti i comandi è possibile trovarla in [5].

Strumenti utilizzati per gli esempi Gli strumenti che si sono utilizzati per svolgere i test sono due: l'emulatore di reti *mininet*, con il quale è possibile sviluppare e sperimentare sistemi SDN e, ovviamente, OpenVirteX. Per facilitare l'utilizzatore, nella homepage del progetto OVX è possibile scaricare una macchina virtuale con mininet, tutti gli script e le utility per eseguire OVX già presenti ed installati. Tutti i comandi che seguono fanno riferimento alla macchina virtuale.

Come primo caso d'uso, si presenta una semplice topologia di rete formata da 4 switch e 9 host collegati tra loro come in Figura 2.5.

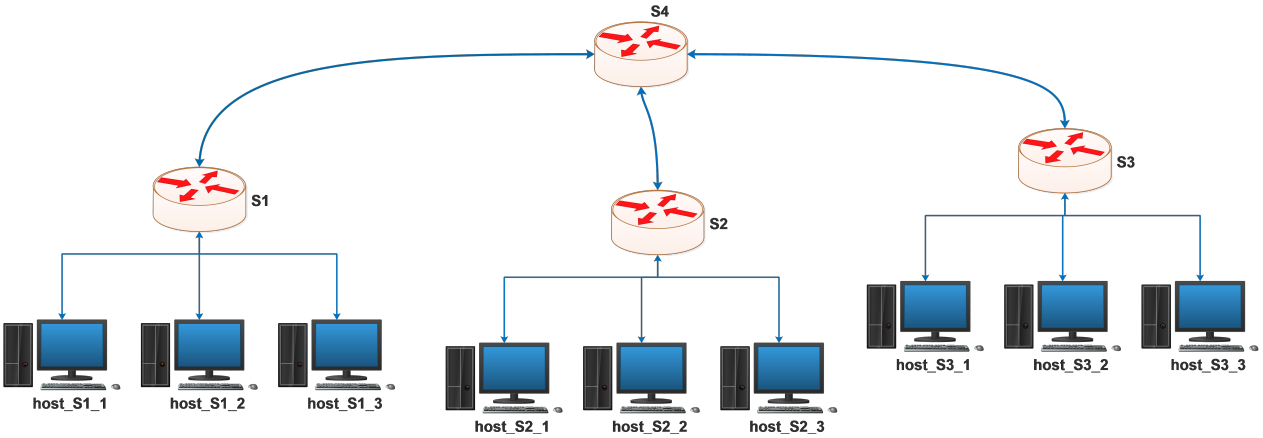


Figura 2.5: Topologia del case study I.

La rete virtuale che si vuole ottenere dopo le dovute considerazioni, mantiene inalterati gli switch ed i collegamenti tra essi ma utilizza solo 3 host: `host_S1_1`, `host_S2_2` e `host_S3_3`.

Il primo passo che si fa è quello di costruire la topologia con l'uso di mininet. In questa fase si vanno a specificare il numero di switch da utilizzare indicando,

in particolare, il DPID[6] di ognuno ed il *fanout* ovvero quanti host si vogliono collegare a ciascuno switch. Inoltre si attribuisce un nome e un indirizzo IP ai vari host. In Appendice A è presente il codice di riferimento in python.

Si esegue la topologia con il comando:

```
1 sudo python topology.py
```

e si abilita lo script di OVX, il quale rileva la topologia appena creata andando a popolare le sue strutture dati della parte fisica:

```
1 cd OpenVirteX/script/
2 sh ovx.sh
```

Successivamente si utilizzano le API di OVX per abilitare la network virtualization. In particolare si crea una rete virtuale andando a specificare gli switch virtuali, le porte associate a questi, i link tra queste ultime e le connessioni tra switch e host.

Per creare una virtual network si utilizza `ovxctl`, uno strumento a linea di comando che interagisce con OVX attraverso delle API. Per avere più informazioni è possibile utilizzare l'opzione `help`:

```
1 cd OpenVirteX/utils/
2 python ovxctl.py -- help
```

Istanziare la virtual network Il seguente comando crea una virtual network la quale ha un controller che comunica con essa tramite il protocollo TCP, il quale si trova in locale sulla porta 10000. Gli IP di questa rete saranno all'interno del range 10.0.0.0/16.

```
1 python ovxctl.py -n createNetwork tcp:localhost:10000
   ↪ 10.0.0.0 16
```

Il comando restituisce una tupla di valori dove il più importante è il *tenantID*, un numero intero crescente che parte da 1 e varia con l'aggiunta di reti virtuali. Questo valore è importante perché nei prossimi comandi verrà sempre richiesto per contraddistinguere la rete virtuale che si sta andando a manipolare.

[6] *Datapath ID*: identificatore univoco di ogni switch all'interno di una rete OpenFlow. Quindi, in questo caso, ogni rete virtuale vede lo stesso DPID per ogni switch.

Creare gli switch virtuali Si passa ora alla creazione degli switch virtuali. In questo caso, oltre ad indicare il tenantID al quale appartengono, si deve specificare a quali switch fisici viene mappato. Questo perché è anche possibile che uno switch virtuale sia composto da diversi switch fisici come nel secondo case study in Appendice B. In questo caso ogni switch virtuale corrisponde ad esattamente uno switch fisico.

```
1 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:01:00
2 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:02:00
3 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:03:00
4 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:04:00
```

Ad ogni switch viene assegnato un DPID virtuale attraverso il quale è possibile identificarlo unicamente. Questo DPID appare nel seguente formato: `00:a4:23:05:00:00:00:01` nel caso del primo switch creato e così via, incrementando il valore più a destra.

Creare le porte virtuali Le porte sono usate per connettere host e switch tra loro. Queste porte sono create specificando il tenantID della rete, il DPID fisico dello switch ed il numero di porta fisico.

```
1  python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 1
2  python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 4
3
4  python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 2
5  python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 4
6
7  python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 3
8  python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 4
9
10 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 4
11 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 5
12 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 6
```

Questa chiamata restituisce un identificatore della porta virtuale che si è creata. In questo caso il valore iniziale degli identificatori è 1 ed è incrementale per ogni switch. Quindi una porta viene identificata attraverso il suo ID e virtual switch a cui appartiene. Nei comandi si sono raggruppate le porte riguardanti i vari switch: le prime due porte create avranno ID rispettivamente pari a 1 e 2,

così come le porte riguardanti il secondo e terzo switch. Si evidenzia il discorso di questi ID perché nei prossimi comandi sono richiesti.

Creare link virtuali OVX interpreta i link virtuali come tutte le connessioni *solo* tra switch virtuali. Quindi, siccome si vuole che la rete virtuale rispecchi quella fisica, saranno presenti 3 virtual link: da `S1` a `S4`, da `S2` a `S4` e da `S3` a `S4`. Per ogni link si specifica il DPID virtuale e l'ID della porta virtuale di ognuno dei due switch facenti parte della connessione. Infine gli ultimi due parametri sono il *routing mode* (nel caso in questione si è indicato `spf` (*Shortest Path First*)), ed il numero di collegamenti di backup che si vogliono avere se, per esempio, il collegamento riscontra qualche problema (sempre nel caso si indichi spf come routing mode).

```
1 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:01 2
    ↪ 00:a4:23:05:00:00:00:04 1 spf 1
2 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:02 2
    ↪ 00:a4:23:05:00:00:00:04 2 spf 1
3 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:03 2
    ↪ 00:a4:23:05:00:00:00:04 3 spf 1
```

OVX restituisce un identificatore univoco per ogni virtual link creato.

Connessione tra host e switch Le porte sono usate anche per connettere i vari host. In questo caso si specificano, oltre al *tenantID* anche il DPID virtuale dello switch e l'ID della porta virtuale. Infine si specifica a quale host lo si vuole connettere attraverso l'indirizzo MAC dell'host (specificato in fase di creazione della topologia fisica).

```
1 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 1 00:00:00:00:01:01
2 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 1 00:00:00:00:02:02
3 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:03 1 00:00:00:00:03:03
```

Notare che l'indirizzo MAC `00:00:00:00:01:01` si riferisce al `host_S1_1` e così via.

Eseguire la virtual network Dopo che si sono inseriti tutti i dispositivi e che si sono formate tutte le connessioni utili, è possibile eseguire la rete virtuale semplicemente indicando il suo *tenantID*.

```
1 python ovxctl.py -n startNetwork 1
```

Verifica del funzionamento della virtual network Come verifica del fatto che la rete funziona si provi la seguente operazione. Prima di eseguire la virtual network con l'ultimo comando indicato sopra, si provi ad eseguire un ping tra l'host `host_S1_1` e l'host `host_S3_3` nella console di mininet. Quello che si noterà è che i due host non sono raggiungibili tra loro. In seguito, si faccia partire la rete virtuale si riprovi ad eseguire il ping, si noterà che andrà a buon fine.

Vincoli e limitazioni

Durante l'implementazione delle varie reti virtuali, sono sorti alcuni problemi di progettazione. In particolare sono dei vincoli che OpenVirteX impone alle varie reti virtuali. Qui di seguito vengono illustrati.

- Per poter fare in modo di mappare più di uno switch fisico all'interno di uno virtuale, si è dovuto impostare la topologia di rete fisica in modo che tutti gli switch fossero connessi tra loro. Inizialmente si era predisposta una topologia leggermente diversa per quel che riguarda il secondo case study ma, dopo aver capito quale fosse il problema, si sono dovute modificare alcune connessioni per poter permettere di eseguire la prova.
- Da una prima analisi, sembra che non si possa utilizzare lo stesso host in due reti virtuali differenti. Ovviamente questo viene fatto in modo da garantire uno *slicing* più accurato, senza rischiare che le varie parti si sovrappongano. Se, per esempio, si prova a connettere un host già in uso all'interno di una virtual network in un'altra, OpenVirteX risponderà con un messaggio di errore come segue: "`The specified MAC address is already in use: 00:00:00:00:01:01`". Da questo errore sembrerebbe che l'appartenenza ad una rete virtuale dipenda solo dall'indirizzo MAC dell'host.

Capitolo 3

Analisi avanzata

Dopo aver introdotto la tecnologia OpenVirteX e le sue caratteristiche, illustrato i componenti al suo interno ed aver testato le sue funzionalità con degli esempi, si è deciso di eseguire un'analisi più approfondita per scoprire come avviene lo *slicing* delle varie reti virtuali partendo da una topologia fisica.

Per fare ciò, si è deciso di studiare come avvengono i vari scambi di pacchetti di rete tra OpenVirteX, i controller ed i vari switch. Più in particolare, tutto quello che riguarda la parte che sta tra OVX ed i vari controller viene detta *northbound*, mentre quella che sta tra OVX e la rete sottostante viene detta *southbound*. Per studiare lo scambio di pacchetti si è utilizzato uno dei più famosi software di sniffing, analisi di protocollo e *packet filtering*: Wireshark[1]. Questo software, molto spesso, viene già preinstallato all'interno delle distribuzioni Linux più famose.

Per prima cosa viene descritto in modo teorico come OVX processa i vari messaggi OpenFlow che si vede arrivare. Successivamente invece, si entra più nel merito di come avviene lo scambio dei pacchetti, mostrando catture e diagrammi che illustrano il comportamento dell'intero sistema.

3.1 Event-Loop di OpenVirteX

L'event-loop di OVX gestisce il processamento dei messaggi OpenFlow. I suoi compiti principali sono quelli di:

[1] `https://www.wireshark.org/`

- Svolgere l'handshake OpenFlow tra il datapath ed i vari controller delle reti virtuali. OVX è diviso in due parti principali:
  - *Southbound*: costruisce e mantiene la rappresentazione dell'infrastruttura e gestisce i canali OF tra OVX e il datapath.
  - *Northbound*: presenta ad ogni tenant la sua rete virtuale con i vari switch e link virtuali gestendo i canali OF tra questi switch ed i controller.

  OVX implementa l'handshake tra queste due parti per stabilire i canali di controllo tra esso, i datapath ed i controller.

- Virtualizzare o devirtualizzare i messaggi OF. Ogni messaggio che passa in OVX deve essere tradotto, quindi OVX deve essere in grado di risolvere correttamente le dovute traduzioni, andando a rilevare il corretto canale di comunicazione da usare verso il controller e riscrivere i campi del messaggio per fare in modo che la vista della rete rimanga consistente tra le varie prospettive dei tenant. OVX deve anche essere in grado di invertire questa procedura, andando a ricercare il datapath che dovrebbe ricevere il messaggio e riscrivendo i vari campi per adattarli a quella che è la rete fisica sottostante. Più in particolare il termine *virtualizzare* significa gestire tutti quei messaggi verso la parte *northbound* e provenienti dalla *southbound*, mentre con il termine *devirtualizzare* si intende il contrario.

- Gestire le funzioni di *keep-alive* da e verso il datapath ed i controller. I componenti scambiano dei messaggi di *echo request* e *reply* mentre sono in uno stato di *idle*. OVX deve essere in grado di gestire questi messaggi sulla base dei rispettivi canali di provenienza.

In Figura 3.1 viene proposta una descrizione ad alto livello del event-loop.

Rilevamento topologia e gestione pacchetti LLDP

I pacchetti LLDP sono usati dai dispositivi di rete per informare circa la propria identità, capacità e vicinanza con altri dispositivi. OVX utilizza questi ultimi per mantenere una vista aggiornata della rete fisica ed anche per altri compiti come, ad esempio, la categorizzazione delle porte interne agli switch suddividendole in *fast* o in *slow port*.

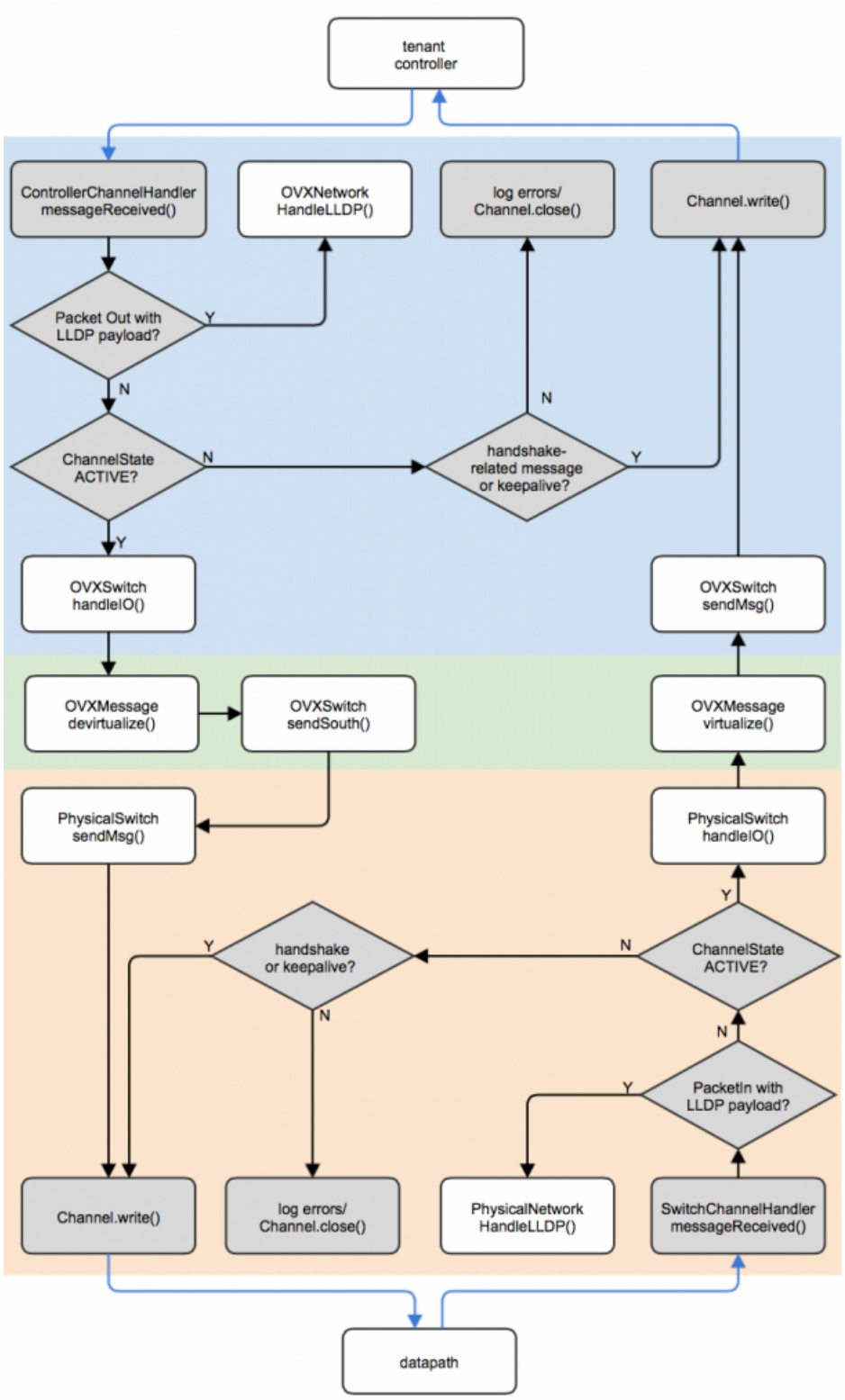


Figura 3.1: Event-loop di OVX.

Inoltre, OVX deve saper gestire i *PacketOut* contenenti al loro interno degli LLDP provenienti dai controller. In questo caso questi LLDP non vengono tradotti e inviati ai rispettivi switch, ma vengono gestiti internamente. Stessa cosa per gli LLDP che dagli switch sono inviati verso OVX, i quali non sono rispediti ai controller. Internamente, OVX ha una procedura in grado di gestire l'arrivo di questi pacchetti e di rispondere con le dovute informazioni per fare in modo che ogni controller veda esattamente la topologia che ha richiesto, senza venire a conoscenza della rete reale.

3.2 Analisi pacchetti con Wireshark

L'obiettivo di questa parte, è quello di studiare il comportamento che avviene sul piano del forwarding dei pacchetti, andando ad analizzare come lavora OpenVirteX e cercando di capire se e come avviene una qualche forma di traduzione al suo interno. Si è cercato quindi di chiarire se vengono rimappati tutti i pacchetti che partono dallo switch verso il controller, oppure se qualcuno di essi si ferma in OVX e viene rispedito indietro.

Per fare ciò, si è andata ad utilizzare la topologia di rete del primo case study (si veda Figura 2.5). Si sono create tre reti virtuali identiche, con l'unica differenza che utilizzano host diversi. Questo perché, come è indicato nel capitolo precedente, non si possono utilizzare degli host che sono già stati utilizzati in altre reti virtuali. Quindi, gli host utilizzati nelle tre virtual networks sono:

Tabella 3.1: Host utilizzati nelle varie virtual network.

Virtual Network	**Host usati**
1	host_S1_1, host_S2_2, host_S3_3
2	host_S1_2, host_S2_3, host_S3_1
3	host_S1_3, host_S2_1, host_S3_2

Come è possibile osservare dalla tabella, sono presenti tre controller, uno per ogni rete virtuale. Ognuno di essi ha la visione della propria topologia rete, che è la medesima per tutti e tre.

Dopo aver fatto partire queste reti virtuali si sono generati dei pacchetti per riuscire a distinguere, nel modo più chiaro possibile, le varie sorgenti e

destinazioni di questi ultimi con Wireshark. Ma si è capito fin da subito che, dopo aver analizzato questi indirizzi, i pacchetti non fanno altro che utilizzare l'indirizzo di localhost (127.0.0.1) per comunicare. Questo perché la comunicazione avviene tutta all'interno di una singola macchina. Quindi si evince che i vari controller, gli switch e OVX utilizzano diverse porte per comunicare tra di loro, ciò rende l'azione di sniffing più complicata.

Per risolvere questo problema si è cercato di adoperare una strategia alternativa: si sono utilizzate delle interfacce virtuali nella macchina che ospita mininet e OVX. Ognuna di queste interfacce ha un indirizzo IP diverso in modo da associare ognuno di questi indirizzi ad un diverso controller ed uno per OVX. Così facendo, si riescono a distinguere gli indirizzi sorgente e destinazione dei vari flussi di pacchetti che vengono scambiati nella parte *northbound* del sistema. Successivamente, si sono modificati i file di configurazione dei controller, agendo inoltre sui comandi che si danno a OVX, in modo da associarlo al corretto controller.

3.2.1 Creazione interfacce virtuali

Per creare le interfacce virtuali si è modificato il file `/etc/network/interfaces` il quale contiene la maggior parte delle impostazioni di rete. A questo file si sono aggiunte le seguenti quattro interfacce virtuali: `lo:ovx` per associarla a OpenVirteX, `lo:fl1`, `lo:fl2` e `lo:fl3` per associarle rispettivamente ai controller (FloodLight) delle reti virtuali 1, 2 e 3 (il codice che si è aggiunto si trova in Appendice C.1). Ad esse si sono associati i seguenti indirizzi:

Tabella 3.2: Indirizzi IP associati alle varie interfacce virtuali

	Interfaccia virtuale	**Indirizzo IP**
OVX	`lo:ovx`	`10.10.10.10`
Controller 1	`lo:fl1`	`10.10.10.1`
Controller 2	`lo:fl2`	`10.10.10.2`
Controller 2	`lo:fl3`	`10.10.10.3`

Dopo averli impostati, si sono modificati i file di configurazione di FloodLight, in modo da imporre la creazione dei controller a questi indirizzi.

Modifica dei file di configurazione di FloodLight

Queste modifiche vengono eseguite andando a manipolare i file di configurazione che vengono eseguiti allo startup del sistema operativo. Questa cosa è presente di default nella macchina virtuale che è stata usata per questi test. Più in particolare, la cartella contenente questi file è: `/home/ovx/ctrl` ed i file interessati sono: `ctrl1.floodlight` per quello che riguarda il controller 1, `ctrl2.floodlight` per il controller 2 e `ctrl3.floodlight` per il terzo.

A questi file si è aggiunta una semplice riga di codice, la quale specifica l'indirizzo in cui installare il controller:

```
net.floodlightcontroller.core.FloodlightProvider.openflowhost = 10.10.10.1
```

nel caso dell'indirizzo del controller 1.

Modifica dei comandi associati alle Virtual Networks

Infine, per fare in modo che le varie reti virtuali che si vanno a creare facciano riferimento ai controller appena indicati, si deve specificare l'indirizzo nel comando `createNetwork`. Di seguito si propone l'esempio per quanto riguarda il comando della prima rete virtuale:

```
python ovxctl.py -n createNetwork tcp:10.10.10.1:10000
   ↪ 10.0.0.0 16
```

Grazie a queste accortezze si è riusciti a ricostruire, tramite l'indirizzo IP sorgente o destinazione, lo scambio di pacchetti che avviene tra OVX ed i vari controller.

3.2.2 Analisi dei pacchetti - ping tra due host

Per capire il funzionamento interno e per vedere come i vari pacchetti si propagano tra i vari switch e controller fisici e virtuali, si è deciso di proporre un'esemplificazione, nella quale si effettua un semplice ping tra due host di una delle tre reti virtuali descritte all'inizio della sezione. Grazie alle accortezze dei paragrafi appena sopra, si è riusciti a distinguere i vari mittenti e riceventi dei pacchetti in modo chiaro.

Passando ai fatti pratici, dopo aver inizializzato la rete SDN con mininet e dopo aver creato le reti virtuali, è stato messo in ascolto Wireshark sull'interfaccia di loopback in modo da catturare tutti i pacchetti passanti.

Successivamente si è fatta partire la cattura e, nel terminale di mininet si è fatto partire il comando di ping:

```
mininet> host_S1_1 ping -c3 host_S3_3
```

Infine, dopo che sono terminati i tre ping, si è semplicemente arrestata la cattura. Per prima cosa, siccome alcuni pacchetti non sono codificati con i colori giusti, lo si va ad imporre attraverso un semplice comando di `Decode As`. Questo comando è utile nel caso in cui si vogliano fare esperimenti non comuni nella rete. Quello che succede è che si impone a Wireshark di trattare dei determinati pacchetti come se fossero appartenenti ad un certo protocollo che, nel nostro caso, è OFP (*OpenFlow Protocol*). Si clicca con il destro sul pacchetto e, dopo aver selezionato il comando, si passa alla finestra *Transport* ed infine si seleziona la voce OFP[2].

Una volta effettuato questo cambiamento si applica un filtro `of` alla cattura. Quello che succede è che rimangono nella cattura solo i pacchetti OpenFlow (Figura 3.2).

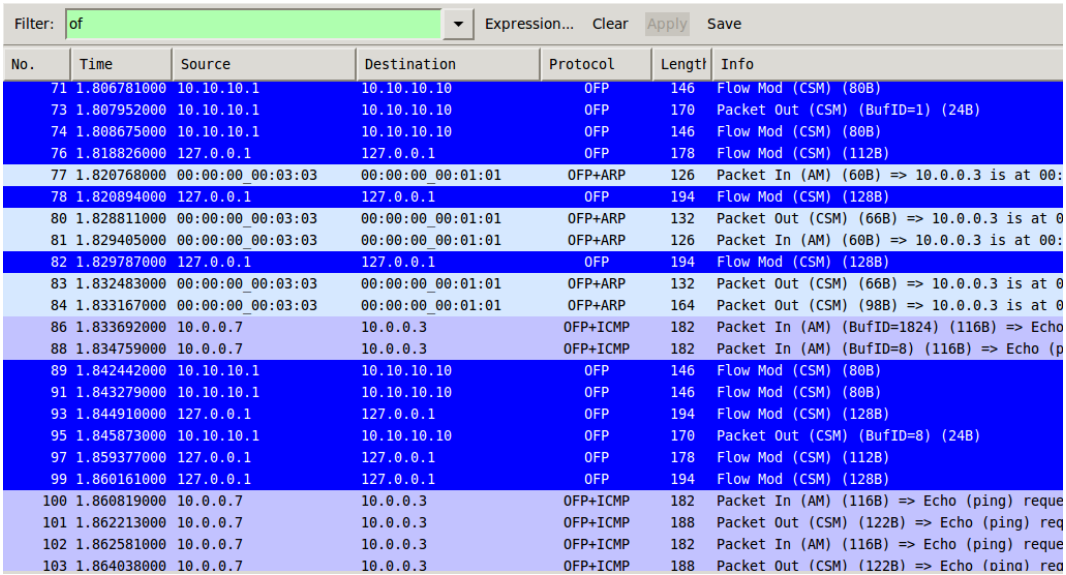


Figura 3.2: Filtro per i pacchetti OpenFlow.

[2]Per poter fare queste azioni bisogna che siano installati i *dissector* di OpenFlow per Wireshark. Nella macchina virtuale utilizzata sono già preinstallati.

Come si può vedere da questa figura, gli indirizzi sorgente e destinazione dei pacchetti 76, 78 e 82 sono identici. Questi pacchetti si riferiscono a scambi che avvengono nella parte southbound della rete; più in particolare, tra i vari switch ed il controller. Ma, per capire a quale switch si riferiscono, l'unico modo sarebbe quello di analizzare tutti i pacchetti iniziali di *Feature Request/Reply* che gli switch scambiano con il protocollo OF in modo da ricondurli ad uno di questi.

Semplificare la lettura delle catture Per semplificare questo procedimento si può procedere in maniera circa similare per attribuire degli indirizzi IP ai vari controller. In pratica quello che si può fare per riuscire a distinguere il traffico OpenFlow tra switch e OpenVirteX, è fare in modo che ogni switch si connetta a quest'ultimo usando un diverso indirizzo. Quindi si devono assegnare tanti alias al loopback quanti sono gli switch creati nella rete mininet (nel caso in questione sono altre quattro interfacce di rete). Si è deciso di utilizzare un alias del tipo 10.20.20.X/24. In Appendice C.2 è presente l'aggiunta da inserire nel file delle impostazioni di rete.

Come ultimo passaggio, dopo che è stata effettuata l'aggiunta ed aver riavviato la macchina (per permettere l'attivazione di questi nuovi alias), si deve modificare il file utilizzato per avviare la topologia di rete con mininet. La presente aggiunta fa in modo che i vari switch si colleghino ai diversi indirizzi appena specificati. Per fare ciò, si è aggiunto il seguente codice dopo che si è fatta partire la rete:

```
1  ...
2  net.addController(controllerGiulio2)
3  net.start()
4
5  var = 0
6  for switch in net.switches:
7    var = var + 1
8    s = str(var)
9    ipControllerBinding =
       ↪ str.replace("tcp:10.20.20.X:6633", 'X', s)
10   switch.cmd('sudo ovs-vsctl set-controller '
11               + switch.name + ' ' + ipControllerBinding)
12 ...
```

dove nell'array `net.switches` sono presenti tutte le istanze degli switch, una volta che si è fatto lo start della rete. Inoltre la stringa `ipControllerBinding` è una stringa che deve variare per ogni switch in modo da poterli mappare correttamente. Si è notato che l'array viene popolato secondo l'ordine con il quale si aggiungono gli switch tramite il comando `addSwitch(...)` quindi si sono associati i seguenti indirizzi agli switch della rete:

Switch	**Indirizzo IP**
S1	10.20.20.1
S2	10.20.20.2
S3	10.20.20.3
S4	10.20.20.4

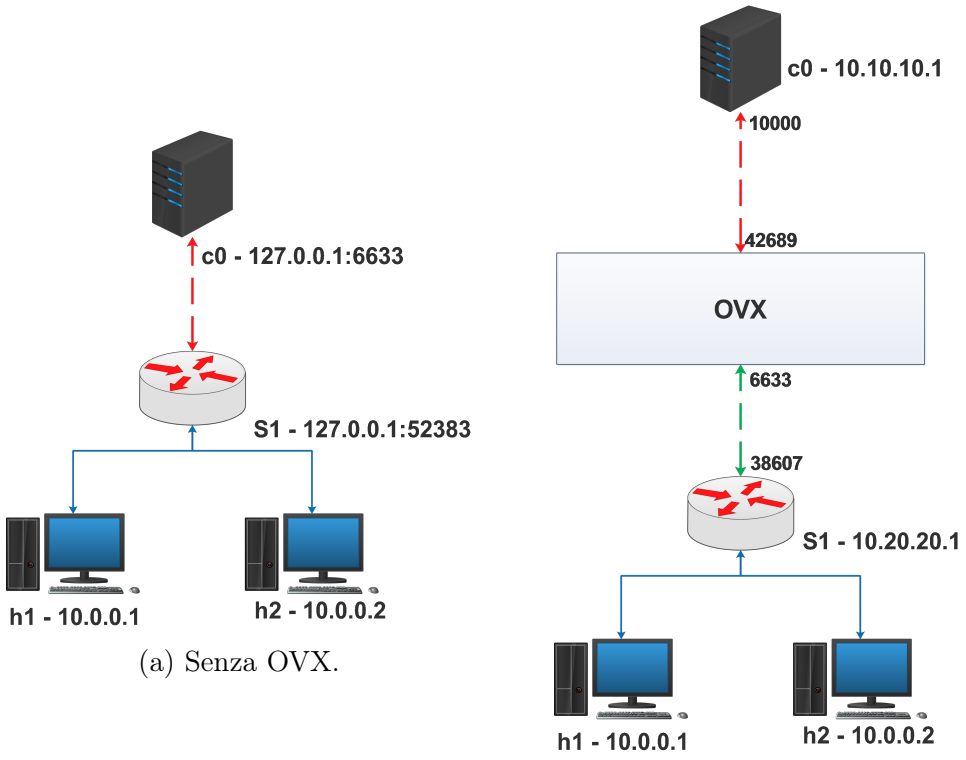
Una volta salvato il file, si fa ripartire tutta la rete e si ripercorrono tutti i passaggi per fare il ping. Con questa nuova cattura, si è ora in grado di identificare gli switch attraverso un loro indirizzo IP.

3.3 Relazione pacchetti northbound e southbound

Il passo successivo è stato quello di attribuire ad ogni pacchetto dalla *southbound* una relativa controparte verso la *northbound* e viceversa. Finora è stato osservato un insieme di pacchetti che fluisce da una parte all'altra senza dare molta importanza al contenuto di questi ultimi né al loro comportamento.

Ora, invece, quello che si vuole fare è analizzare un intero scambio di pacchetti in modo da capire, una volta per tutte, il funzionamento "ai morsetti", utilizzando la tecnologia OVX come una *black box* e cercando di chiarire come si comporta. Quello che viene fatto nel prossimo capitolo sarà, invece, andare a mettere mano a questa scatola nera alterandone il comportamento per permettere azioni che, per ora, non è in grado di fare.

Per fare ciò, si sono analizzate due semplici topologie per rendere minimo lo scambio di pacchetti e per rendere possibile un maggiore focus sull'obiettivo. Per ogni topologia, si è eseguita un'analisi di una cattura di un ping tra due host su una *rete senza OVX* e, quindi, con la presenza di un solo controller per capire in modo approfondito come avviene uno scambio tra componenti fisici reali. Successivamente, si è aggiunto OVX creando una rete virtuale uguale alla fisica, ed andando ad analizzare sempre lo stesso ping tra gli stessi due host.



(a) Senza OVX.

(b) Con OVX.

Figura 3.3: Topologia 1.

Riassumendo quindi, si è aggiunto un nuovo componente alla comunicazione e si è analizzato il comportamento, andando a riportare i vari scambi in un diagramma delle interazioni.

Topologia 1

Nella prima prova si è utilizzata la topologia più semplice esistente, descritta in Figura 3.3.

Come si può osservare, l'immagine a sinistra presenta la topologia senza OVX, quindi con lo switch connesso direttamente al controller. L'immagine a destra invece, è caratterizzata dalla presenza di OVX che si interpone alla comunicazione tra controller e switch. Successivamente si è eseguito un ping tra i due host presenti, andando a catturare i vari pacchetti che si scambiano.

In Figura 3.4 sono presenti tutti i passaggi, partendo dal primo *ARP request* nel quale l'host `h1` richiede di comunicare con `h2`, per finire nell'ultimo *echo ping reply* di `h2` verso `h1`.

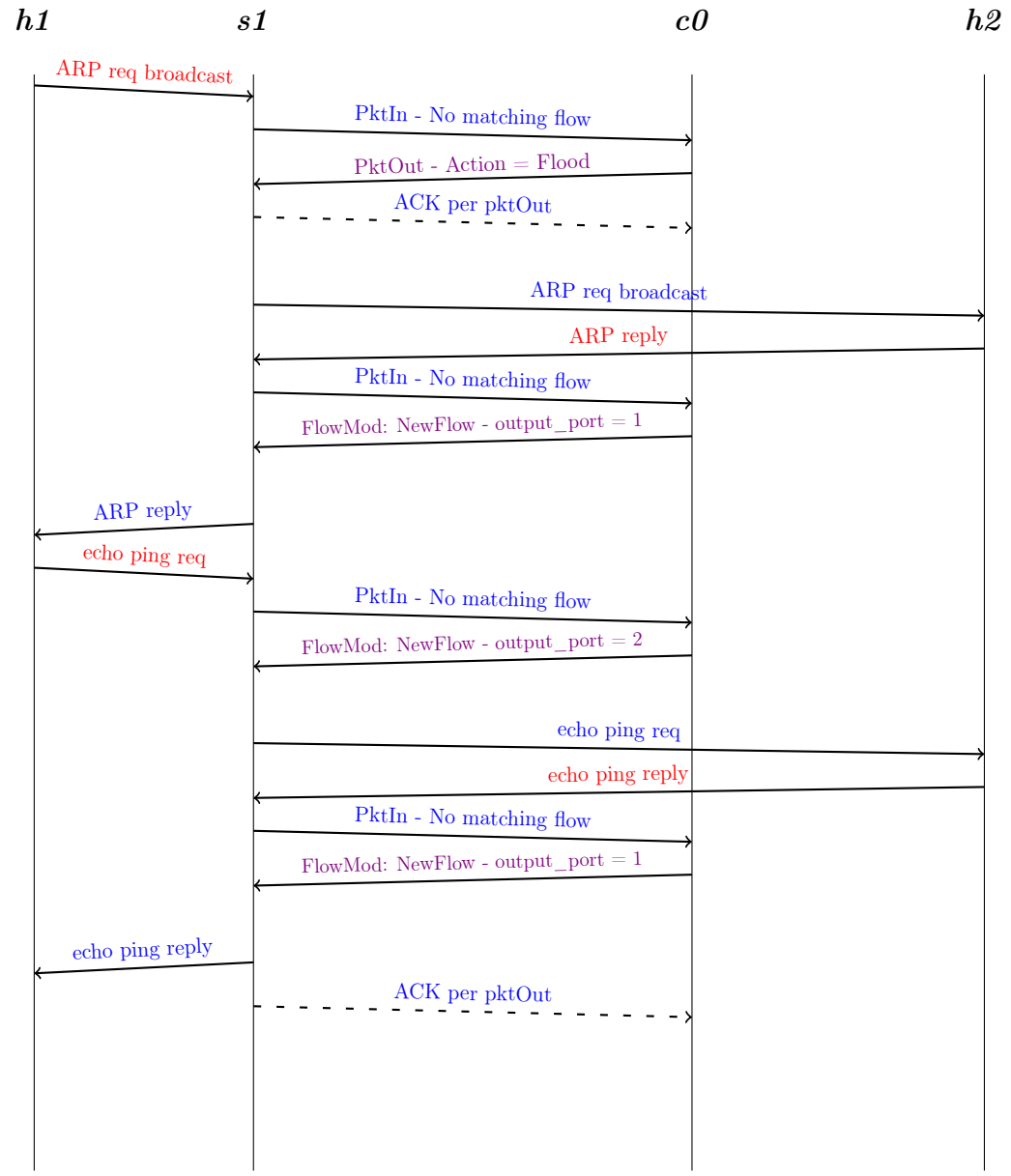


Figura 3.4: Diagramma scambio di pacchetti tra i vari componenti I.

Spiegazione scambio pacchetti senza OVX

- Il primo passaggio è l'invio di un *ARP request* che `h1` manda in broadcast e che serve per sapere dove trovare `h2`.
- Questo pacchetto viene intercettato dallo switch `S1` che, non avendo regole installate al suo interno per l'inoltro di questa tipologia di pacchetto, genera un *PacketIn* verso il controller.
- Il controller risponde con un *PacketOut*, nel quale indica l'azione da attuare a fronte dell'arrivo di quella tipologia di pacchetto.
- Successivamente `h1` riesce a raggiungere `h2` con il suo *ARP request*, nel frattempo `h2` a fronte dell'arrivo di questo pacchetto non fa altro che rispondere con le sue informazioni per comunicare con esso.
- Questa risposta genera un *ARP reply* che `S1` intercetta e che, non avendo regole installate al suo interno per l'inoltro di questa tipologia di pacchetto proveniente da `h2`, genera un altro *PacketIn* sempre verso il controller.
- Il controller risponde con un pacchetto di tipo *FlowMod*, nel quale è presente l'installazione di una nuova regola per il flusso con *output_port = 1*. Questo significa spedire il pacchetto in questione attraverso la porta 1 dello switch; questa porta è connessa ad `h1`.
- Successivamente `h2` spedisce il suo *ARP reply* riuscendo a raggiungere `h1`, che ora ha tutte le informazioni per comunicare con il suo mittente. Quindi quello che fa è inviare un pacchetto di *echo ping request* verso `h2`.
- Questo pacchetto però, come già successo, viene intercettato da `S1` che, non avendo nessuna regola per gestire questo tipo di pacchetto, genera l'ennesimo *PacketIn* verso il controller.
- Il controller risponde con un FlowMod nel quale specifica che quel tipo di pacchetto deve essere inoltrato verso la porta numero 2 dello switch (*output_port = 2*). Quindi ora `h1` può raggiungere `h2` con il suo *echo request*.
- Una volta raggiunto `h2`, questo risponde con l'ultimo passaggio per ultimare l'azione di ping, ovvero generando un *echo ping reply*.

- Questo pacchetto però, non raggiunge direttamente `h1` ma si ferma in `S1` perché, ancora una volta, non è presente nessuna regola in grado di processare tale pacchetto. Quindi viene generato l'ennesimo *PacketIn* verso il controller che risponde con un FlowMod.

- In questo FlowMod è presente la regola che specifica allo switch la porta di uscita per questo tipo di pacchetto: la numero 1.

- Infine, `h2` riesce a raggiungere `h1` con il pacchetto di *echo reply*, terminando così lo scambio di pacchetti per il comando di ping.

Tutti questi scambi avvengono solo ed esclusivamente per un ping in una topologia semplice. Quindi si provi ad immaginare anche solo con la presenza di qualche switch ed host in più, come si espanda il tutto. Per questo motivo si è scelto di utilizzare topologie che siano più semplici possibile. Inoltre, come si vedrà qui di seguito, con l'aggiunta di OVX il numero di pacchetti scambiati aumenta ulteriormente. Infatti, aggiungendo OVX alla topologia e quindi andando ad aggiungere un nuovo attore al diagramma degli scambi di pacchetti, il comportamento è quello illustrato in Figura 3.5. I componenti h1, h2 e S1 fanno parte della southbound, il controller c0 fa parte della northbound mentre OVX fa da tramite a queste due parti.

Spiegazione scambio pacchetti con OVX Quello che accade è che tutto il traffico da o verso la *southbound* o la *northbound* deve necessariamente passare per OVX. Infatti, come illustrato, il controller comunica solo ed esclusivamente con esso; stessa cosa per quanto riguarda `S1`.

Ogni volta che lo switch comunica al controller (per mezzo dei *PacketIn*), OVX intercetta questo pacchetto e, per mezzo di funzioni di mapping, va a rimappare e forgiare dei nuovi dati, alterando anche gli indirizzi di destinazione e sorgente per fare in modo che il particolare controller abbia la visione della sua struttura virtuale di rete.

Nella seguente tabella sono indicati i vari indirizzi che i componenti utilizzano per risalire agli host della rete. In questo caso, per essere più chiari, si fa riferimento a due reti virtuali agenti sulla medesima rete fisica. Ad ognuna di queste appartengono due host.

I controller decidono quali indirizzi usare, in fase di creazione della rete virtuale. Essi utilizzano sempre questi indirizzi per riferirsi agli host, mentre la traduzione ai veri indirizzi fisici viene demandata a OVX. Quest'ultimo,

VN1	Host 1	Host 3
Controller 1	10.0.0.1	10.0.0.2
OVX	1.0.0.1	1.0.0.3
Rete fisica	10.0.0.1	10.0.0.3

VN2	Host 2	Host 4
Controller 2	20.0.0.1	20.0.0.2
OVX	2.0.0.2	2.0.0.4
Rete fisica	10.0.0.2	10.0.0.4

Tabella 3.3: Indirizzi che i vari componenti utilizzano per gestire la rete.

al suo interno, mappa gli indirizzi degli host in base alla rete virtuale a cui appartengono. Quindi se un particolare controller volesse comunicare con un suo host della rete, OVX è in grado di risalire a questo host partendo dal controller che ha richiesto la comunicazione e dall'indirizzo che è presente nel pacchetto.

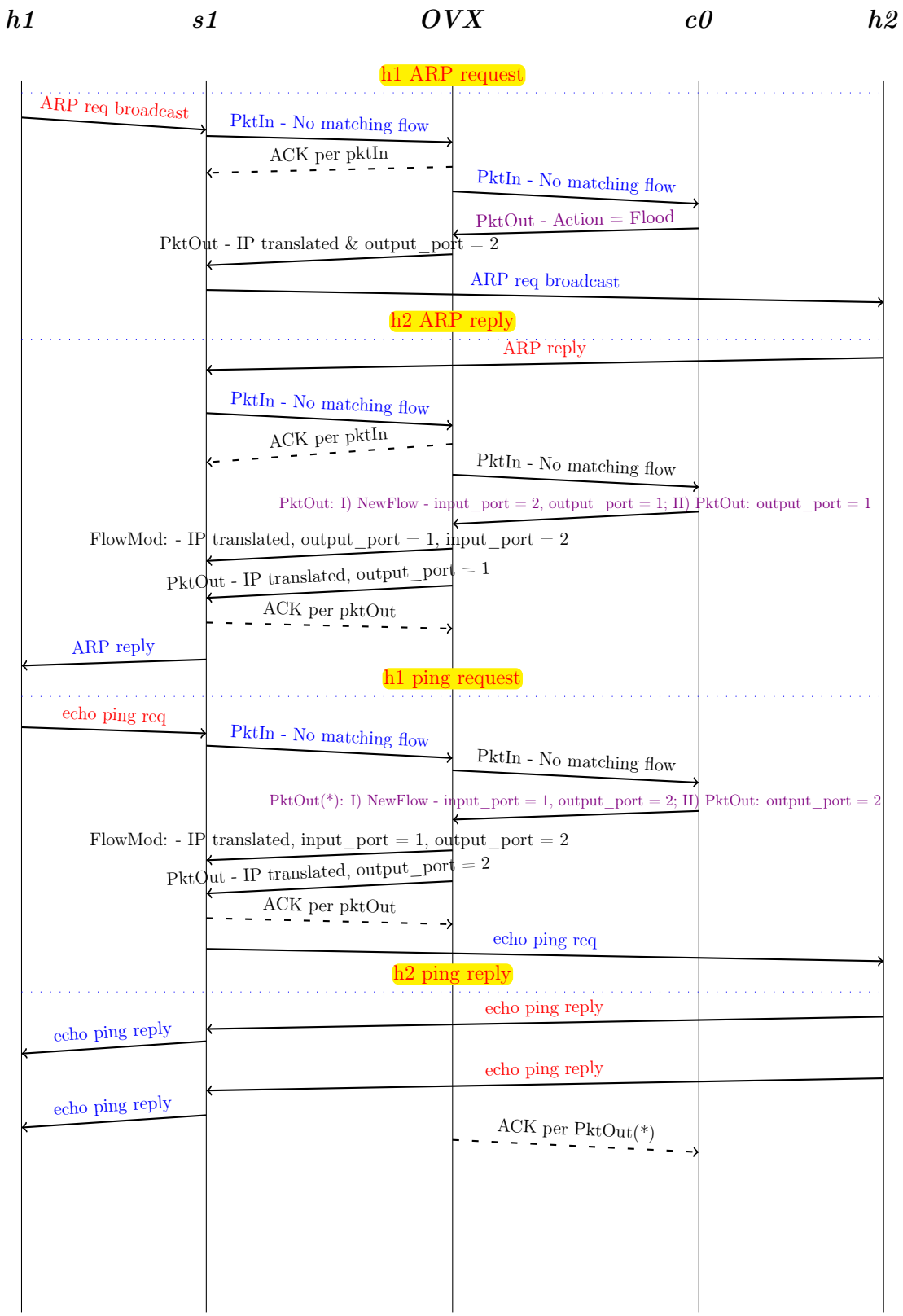


Figura 3.5: Diagramma scambio di pacchetti tra i vari componenti II.

Topologia 2

In questa seconda topologia, si è voluto testare una funzionalità di OVX: mappare più switch fisici all'interno di uno virtuale. Per fare ciò si è usata, anche in questo caso, una topologia abbastanza semplice che permettesse di concentrarsi il più possibile sui pacchetti che vengono scambiati. Questa topologia, a differenza della precedente, ha un numero maggiore di switch perché, per mappare più switch fisici all'interno di uno logico, ne servono almeno due. Inoltre sono presenti più host, in modo da mapparne due all'interno di una prima virtual network e due in un'altra. Servono quindi almeno 4 host perché, come indicato nel capitolo 2.2, un singolo host fisico può appartenere al più ad una rete virtuale.

La topologia utilizzata è quella illustrata in Figura 3.6. Come è possibile osservare, sono presenti due reti virtuali ognuna delle quali contenente un proprio switch virtuale (`Virtual Switch 1` e `Virtual Switch 2`), mappato all'interno di due switch fisici (`S1` e `S2`). Più in particolare, i due switch fisici sono utilizzati per entrambe le reti virtuali. Infine, gli host `h1` e `h3` fanno parte della prima rete virtuale mentre `h2` e `h4` fanno parte della seconda.

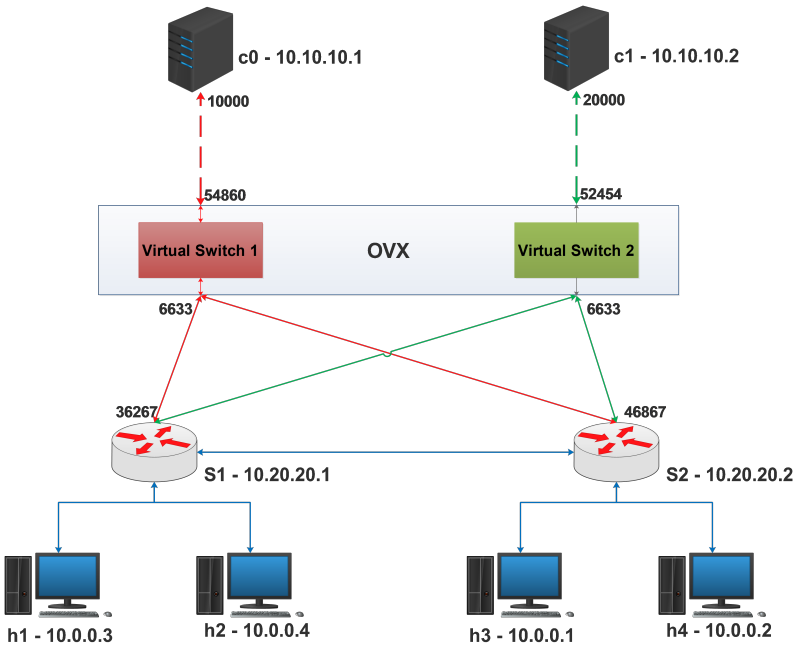


Figura 3.6: Topologia 2.

Come nella Topologia 1, anche in questo caso si è eseguito un ping tra due host di una stessa rete virtuale e, sniffando i pacchetti, si è riusciti a capire come agisce OVX per gestire più switch fisici all'interno di uno virtuale.

Nel caso in cui uno switch fisico si veda recapitare un pacchetto del quale non disponga di regole per la gestione, esso lo rimanda al controller (passando per OVX che eseguirà le dovute traduzioni). Il controller vedrà quindi arrivare un *PacketIn* che avrà come mittente non lo switch fisico ma, bensì, lo switch virtuale all'interno del quale è presente, fra tutti, anche lo switch fisico. Successivamente il controller risponde con un *PacketOut* verso OVX, con destinatario lo switch virtuale. Una volta raggiunto OVX, che ha al suo interno tutti i mapping per gestire le varie virtual network, esso genera un numero di *PacketOut* verso determinati switch fisici lungo i quali si vogliono installare regole per i relativi pacchetti.

Quindi, se si volesse mandare un pacchetto da `h1` a `h3`, quello che avviene è che il controller `c0` manda un solo *PacketOut* verso OVX (perché lui vede solo un unico switch virtuale), mentre OVX manda due *PacketOut*: uno verso `S1` ed uno verso `S2` installando delle regole relative alla parte fisica della rete ed indicando le relative porte di input e di output.

Per concludere, OVX genera verso gli switch un numero di *PacketOut* pari al numero di switch che ha mappato in un suo link interno per il raggiungimento della destinazione voluta.

Capitolo 4

Contributo sulla sicurezza: PySolv

OpenVirteX insieme alla tecnologia SDN, permette di facilitare l'azione gestionale della rete, mettendo in campo un livello di astrazione tra il *data plane* ed il *control plane*, che manipola i vari pacchetti passanti al suo interno. Come già detto, grazie ad esso è possibile garantire una separazione delle reti dei vari utilizzatori in modo che, ognuno di essi, abbia la certezza di utilizzare tutta la rete disponibile, personalizzandola come desidera anche se, come ormai è stato appurato, ogni rete virtuale è una totale astrazione di quella che è la vera rete fisica.

Riprendendo l'argomento principale della tesi introdotto in Abstract, per poter eseguire delle azioni di controllo delle varie reti virtuali, si devono mettere in campo delle accortezze che permettano di manipolare i vari flussi di traffico (e quindi le varie regole nelle Flow Table degli switch), in modo del tutto trasparente ai vari utilizzatori della rete. Tutto ciò non può essere fatto partendo da uno dei controller gestiti dai tenant o, generalizzando, partendo dalla parte *northbound* dell'architettura OVX. Infatti, nel caso in cui si attui una strategia di questo tipo, il requisito che la soluzione proposta deve rispettare è la capacità di capire come sono formate tutte le reti virtuali e quali flussi le attraversano. Quindi è chiaro che per avere questa visione esterna delle varie reti, non è possibile che si imposti la soluzione come se fosse un ulteriore controller collegato ad OVX. Il problema di questo tipo di soluzione è la presenza dello "strato" OVX tra sé stessa e la rete fisica da andare a manipolare; ciò implica che c'è di mezzo l'astrazione che esso fornisce della rete.

Quello di cui la soluzione richiesta ha bisogno, è il fatto di avere un accesso che sia il più diretto possibile alla rete fisica sottostante e perciò, deve cercare

di svolgere il proprio compito in modo più parallelo possibile ad OVX. Finora lo studio della tecnologia OVX, con l'ulteriore analisi dei messaggi e dello scambio dei pacchetti, si è svolto principalmente con due obiettivi. Il primo è quello di capire il funzionamento di questa nuova tecnologia, mentre il secondo è analizzare dove ed in che maniera è possibile subentrare, ottenendo un punto di partenza per ulteriori sviluppi circa la lettura dei vari pacchetti scambiati da OVX sia con la parte *northbound* che con la *southbound*.

Le due possibilità che si sono pensate per subentrare nel sistema, dopo aver scartato a priori quella di posizionare la soluzione nella parte *northbound* dell'architettura OVX, sono le seguenti:

- Creare un ulteriore strato di astrazione nella parte *southbound* dell'architettura.
- Affiancare ad OVX la soluzione, in modo che si integrino tra loro per raggiungere l'obiettivo richiesto.

Nel primo caso si è pensato di creare un altro strato simile ad OVX ma che si inserisse tra i messaggi del *data plane* e OVX, come mostrato in Figura 4.1a. La particolarità di questa soluzione è che entra direttamente a contatto con gli switch fisici ed i loro pacchetti, con la possibilità di manipolare i vari flussi iniettando apposite regole per la particolare evenienza.

Nel secondo caso la scelta è ricaduta sulla possibilità di inserire la soluzione parallelamente ad OVX, affinché non alteri l'architettura standard di quest'ultimo. Quindi si è pensato di "affiancare" la soluzione in modo che, nel caso in cui si dovessero eseguire dei cambiamenti di qualunque tipo, non si vada a modificare il comportamento principale del sistema. In Figura 4.1b viene mostrata la soluzione.

Tra le due, la prima soluzione è sicuramente la più efficace sotto l'aspetto del reperimento dei pacchetti. Del resto, l'aggiunta del livello d'astrazione si traduce in un aumento sostanziale della complessità del sistema che deve essere gestita. Essendo che questo livello verrebbe aggiunto tra due parti principali dell'architettura, qualsiasi tipo di guasto o di errore derivante dalla soluzione, manderebbe in crash l'intero sistema, non fornendo più connettività ai vari utilizzatori. Nel secondo caso invece, la soluzione fornita non entra direttamente a contatto con i pacchetti fisici degli switch lasciando però l'architettura inalterata. Quindi, questa scelta ha il vantaggio di non modificare il funzionamento del sistema; questo vantaggio si traduce inoltre in un maggior overhead di

gestione delle interazioni che avvengono tra OVX e la soluzione, in modo che vengano reperiti i pacchetti che circolano.

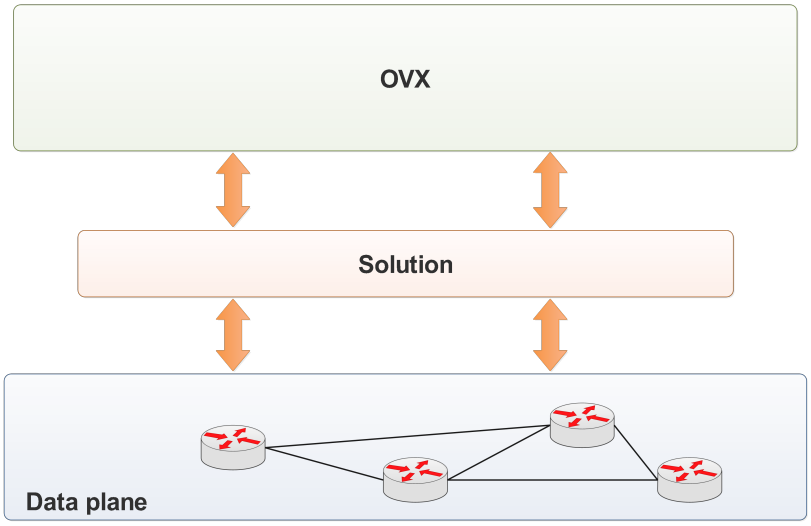
La scelta perciò è ricaduta sul secondo caso (come mostrato in Figura 4.1b), in quanto si vogliono mantenere il più inalterate possibili le funzioni di base che OVX fornisce. In questo modo, il principio di funzionamento di un'architettura OVX "pura" è lo stesso di quello con la soluzione installata, in cui le parti *southbound* e *northbound* interagiscono con lo strato OVX senza che nessuna entità si interponga tra essi. Un altro motivo per cui si è scelta la seconda soluzione sta nel fatto che viene ritenuta meno dispendiosa in termini di carico computazionale a cui è sottoposta. La prima soluzione deve essere in grado di ricevere il pacchetto dagli switch fisici, filtrarli ed elaborarli come richiesto e rinviarli verso OVX. Inoltre, nel caso in cui si ritenesse che quei flussi possano essere di interesse per un'eventuale intercettazione, la soluzione deve anche saper inviare ai rispettivi switch, dei particolari pacchetti che permettano questa azione di controllo. Nel caso della seconda soluzione invece, l'azione di filtraggio e di gestione viene fatta in parallelo alle azioni standard di OVX e, sempre nel caso di un'eventuale intercettazione, si userebbe lo stesso canale che utilizza OVX per interagire con gli switch. In questo modo non si va ad aggiungere l'overhead che ci sarebbe nel primo caso.

Successivamente, siccome lo scopo è quello di mantenere sotto controllo una parte delle comunicazioni che un eventuale host della rete ha, si pensa che il metodo migliore sia quello di ottenere l'accesso a queste comunicazioni grazie all'installazione di regole nella Flow Table di alcuni switch della rete.

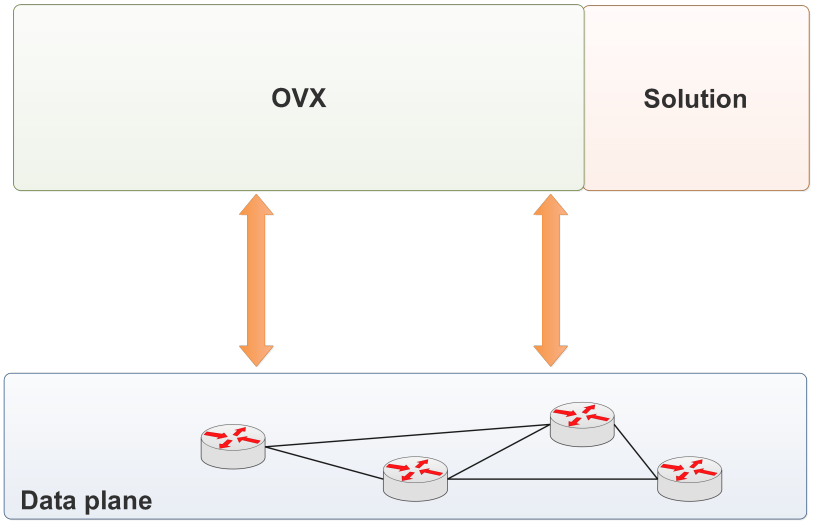
Nelle sezioni che seguiranno, si farà sempre riferimento alla seconda soluzione come base di partenza per tutti gli sviluppi che si sono compiuti.

4.1 Analisi della soluzione proposta

Una volta stabilito dove posizionare la soluzione in modo che svolga il suo compito parallelamente ad OVX, ci si imbatte in una nuova scelta architetturale circa il fatto di dove inserire (in senso astratto), l'applicativo che successivamente si andrà a sviluppare. Si desidera che la soluzione richiesta sia accessibile solamente agli individui autorizzati, nel caso in cui fosse presente del traffico da manipolare per scopi di controllo. Ad esempio, se un giudice autorizza un tecnico ad eseguire delle operazioni per intercettare un certo dispositivo,



(a)



(b)

Figura 4.1: Possibili soluzioni a confronto I.

quest'ultimo dovrà avere accesso all'applicativo in modo da poter reperire le varie comunicazioni nel minor tempo possibile.

La scelta, in questo caso, riguarda il comprendere in che punto dell'architettura installare la soluzione per ottenere la fruibilità massima. Le possibilità sondate sono due:

- Inserire la soluzione all'interno del codice di OVX.
- L'applicativo ed OVX separati ma comunicanti tra di loro.

In Figura 4.2 sono riassunte graficamente.

Nel primo caso (Figura 4.2a) la soluzione fa parte del codice stesso di OVX. Quindi, sostanzialmente, siccome OVX è scritto in linguaggio Java, essa apparirà come un insieme di classi e metodi che interagiscono direttamente all'interno di OVX, utilizzando anche oggetti e metodi preesistenti. In questo caso, un eventuale operatore che volesse manipolare del traffico ha due scelte per comunicare con questo applicativo:

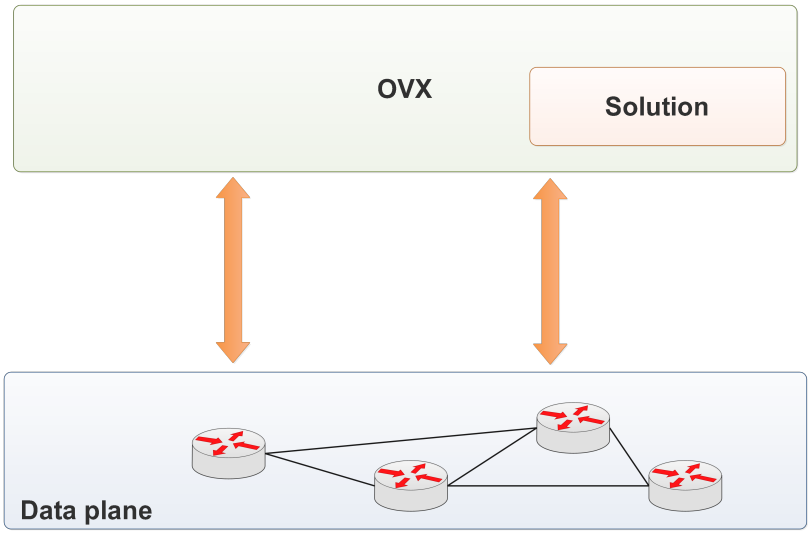
- Modificare direttamente il codice all'interno di OVX.
- Inviare particolari comandi che vengono impartiti dall'esterno con una determinata sintassi.

Inoltre, nel caso in cui si volesse modificare il codice della soluzione, si dovrebbe riavviare l'intero sistema per fare in modo di attuare le modifiche effettuate.

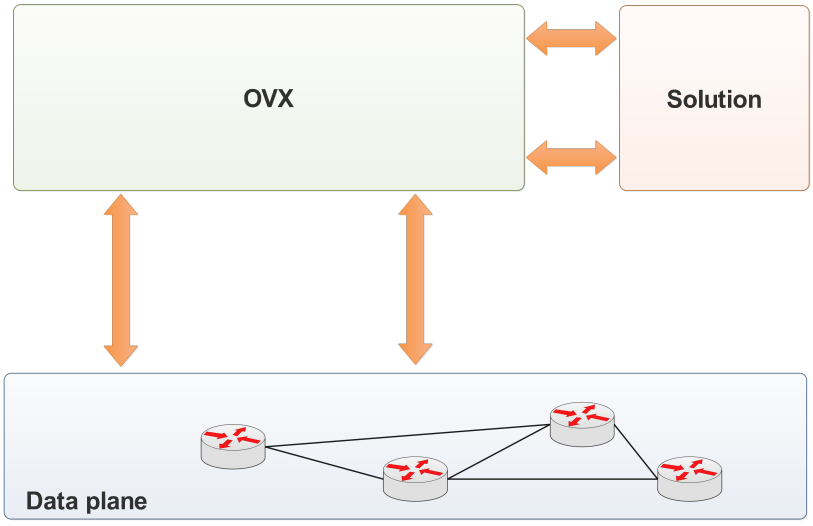
Il secondo caso (Figura 4.2b) mantiene le due entità maggiormente separate, se non del tutto. L'applicativo risiede esternamente all'intera architettura e, siccome non è dipendente da essa, può essere sviluppata anche in linguaggi differenti rispetto a quello utilizzato per OVX. Quindi le comunicazioni tra le due entità sono di primaria importanza per poter svolgere le giuste azioni sull'obiettivo scelto. In questo caso, un'eventuale modifica del codice applicativo non implica un riavvio del sistema, che potrebbe benissimo essere fatto successivamente, nel caso ve ne fosse bisogno.

La scelta fatta è ricaduta sulla seconda soluzione.

Una volta illustrata la soluzione proposta, si passa alla spiegazione delle principali accortezze che sono state prese e di eventuali problemi riscontrati nello sviluppo.



(a)



(b)

Figura 4.2: Possibili soluzioni a confronto II.

4.2 Implementazione

In questa sezione si illustrano i vari passi fatti per arrivare alla soluzione proposta. Dopo aver capito dove posizionare la soluzione in modo da non infierire massicciamente sul funzionamento del sistema, si passa a capire come quest'ultima si deve comportare partendo dallo scopo per cui è stata pensata.

Ricapitolando, i principali componenti di questo tipo di architettura sono gli switch fisici. All'interno di essi sono presenti delle tabelle (Flow Table) che hanno il compito di indicare allo switch, il comportamento che deve assumere a fronte dell'arrivo di una certa tipologia di pacchetto. Si definisce *flusso* (*flow*) una particolare classificazione ottenuta specificando il contenuto di opportuni campi dell'intestazione dei pacchetti che fluiscono negli switch. A fronte dell'arrivo di un determinato pacchetto che non combacia con nessuna delle regole già presenti nella Flow Table, lo switch richiede al piano di controllo il comportamento che deve assumere attraverso lo scambio di particolari tipologie di pacchetto. In risposta, il controller manda un'altra tipologia di pacchetto, chiamata FlowMod, all'interno del quale sono presenti tutte le informazioni utili per installare nello switch una nuova regola per quel particolare flusso. Nel caso della tecnologia OVX, invece, tra il piano di controllo e gli switch si interpone uno strato che permette l'astrazione delle virtual networks. In questo caso, quindi, a fronte dell'arrivo in uno switch di un pacchetto che non combacia con nessuna regola nella tabella, esso invia una richiesta ad OVX che, attraverso le dovute traduzioni, genera una richiesta forgiata *ad hoc* per il particolare controller che governa quella particolare rete virtuale.

Lo scopo è quello di riuscire ad estrapolare varie informazioni che fluiscono all'interno di OVX, per ottenere un controllo costante riguardo un dato obiettivo. La soluzione deve quindi essere in grado di ricevere certe informazioni circa i pacchetti che fluiscono, utili per riuscire a prendere, nel minor tempo possibile, tutte le decisioni che si vogliono adottare a fronte di un particolare evento. Si è pensato che, oltre a comunicare al giusto switch la scelta che deve prendere, OVX debba anche essere in grado di comunicare con l'applicativo esterno, in modo che quest'ultimo riesca a controllare gli indirizzi e verificare l'appartenenza di un determinato flusso ad un obiettivo sensibile e da intercettare. Successivamente l'applicativo deve poter rispondere ad OVX in modo che quest'ultimo alteri (creando nuovi flussi o modificandoli), la FlowTable degli switch, generando un comportamento del quale l'utilizzatore non è minimamente a conoscenza. Questo è quello che avviene già al giorno d'oggi quando

si effettua un'intercettazione: semplicemente colui che è intercettato non è a conoscenza di questo fatto perché si va ad agire a livello dei dispositivi che permettono il forwarding dei pacchetti.

Fatto questo appunto, come primo passo si sono analizzati i vari sorgenti di OVX, andando a ricercare uno o più punti dai quali partire per riuscire a comunicare con l'esterno. Tutti i componenti dell'architettura (sia fisici che virtuali), come già visto, sono rappresentati da classi Java. Dopo un'attenta analisi, la scelta è ricaduta su due classi principali:

- `RoleManager.java`[1] che istanzia oggetti utilizzati dai `OVXSwitch` (ovvero gli switch virtuali) per comunicare con i vari controller connessi all'architettura.

- `PhysicalSwitch.java`[2] rappresenta tutti gli switch fisici e gestisce tutte le comunicazioni verso di essi.

All'interno di queste classi sono presenti due particolari metodi: `checkAndSend(...)` nel primo e `sendMsg(...)` nel secondo. Questi metodi permettono di comunicare rispettivamente con i vari controller e con gli switch fisici. Quindi si avranno tanti PhysicalSwitch quanti sono gli switch fisici della rete e tanti RoleManager quanti sono i vari OVXSwitch di tutte le reti virtuali. Ovviamente i pacchetti che questi due oggetti inviano sono diversi. Più in particolare, entrambi devono essere pacchetti del protocollo OpenFlow, in modo che sia gli switch che i controller possano leggerli correttamente ma, come già detto in precedenza, i pacchetti verso la *southbound* hanno indirizzi differenti rispetto a quelli verso la *northbound*.

Successivamente si è voluto analizzare cosa, i due oggetti, riescono ad inviare per capire che tipo di informazioni si riesce a reperire. Quindi si è creata una semplice funzionalità in python che permettesse di ricevere dati rimanendo in ascolto su una porta. Infine si è creata una socket in ognuno dei due oggetti e la

[1] Presente nella directory: `/OpenVirteX/src/main/java/net/onrc/openvirtex/elements/ /datapath/role/RoleManager.java`

[2] Presente nella directory: `OpenVirteX/src/main/java/net/onrc/openvirtex/elements/ /datapath/PhysicalSwitch.java`

si è connessa alla funzionalità appena descritta[3]. In Figura 4.3 viene riassunto quanto si è appena descritto.

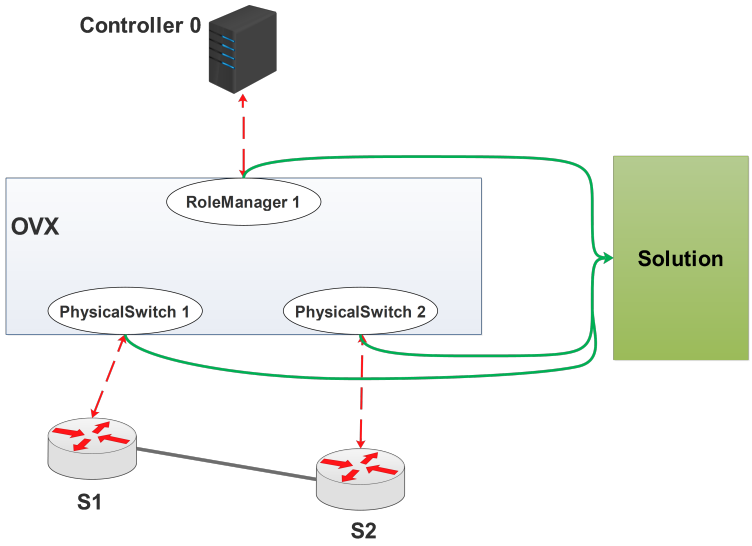


Figura 4.3: Primo step: invio dati da OVX.

Inizialmente si sono semplicemente replicati tutti i messaggi che OVX scambia sia con il controller che con gli switch fisici. Con l'avanzare del progetto si è giunti alla struttura finale del messaggio che viene inviato da OVX.

Struttura del messaggio da OVX verso la soluzione

All'inizio ed alla fine del messaggio si hanno due delimitatori. Successivamente, come primo dato importante si ha la *lunghezza* del messaggio che segue. Questa lunghezza è formata da 10 caratteri che, se troppi per contenere il numero di byte del messaggio che segue, viene riempita di zeri (es: 0000000806). Proseguendo si ha il corpo vero e proprio del messaggio formato da 3 parti principali:

[3]Per ricompilare il codice di OpenVirteX si deve prima di tutto eliminare il file `/OpenVirteX/target/OpenVirteX.jar`. In questo modo, una volta che si lancia il comando `sh ovx.sh` per avviare OpenVirteX, non trovandosi il file appena eliminato, ricompilerà tutto il codice. Se non sono presenti errori di sintassi allora il comando farà partire OpenVirteX con le nuove modifiche aggiunte.

- JSON dei campi del pacchetto che circola in OVX;
- *localAddress* e *remoteAddress* sono gli indirizzi che OVX utilizza per collegarsi ai vari switch;
- Stringa generata dal metodo `toString()` dell'oggetto specificato nel primo punto. Da questa stringa possono essere dedotte altre informazioni utili e che, dal JSON del primo punto, non sono deducibili perché i valori vengono alterati.

Tra questi campi sono presenti dei delimitatori in modo da capire dove inizia una certa informazione. Questi sono specificabili nel file di configurazione utilizzato per il progetto e spiegato durante il proseguimento del capitolo. Per quello che riguarda questi delimitatori, nel file è possibile specificare le tre tipologie presenti nella seguente figura. In particolare il campo:

- ***delim_ init*** specifica il delimitatore iniziale del messaggio;
- ***delim_ final*** specifica il delimitatore finale del messaggio;
- ***delim_ inter*** specifica i delimitatori intermedi del messaggio.

Si presti particolare attenzione a come si generano questi delimitatori perché si potrebbe incorrere in un errato split delle varie informazioni, dovuto all'utilizzo di delimitatori che coincidano con parti del messaggio. Per essere praticamente sicuri di non insorgere in errori di questo tipo, si consiglia di utilizzare dei caratteri speciali ripetuti, come per esempio # ripetuto dieci volte. Per ulteriori approfondimenti sull'uso di questi delimitatori si rimanda alla Sezione 4.7.

Proseguendo, si è subito resi conto che tutta la parte interagente con la *northbound* è inutile, perciò è stata eliminata. Questo perché, i pacchetti interessanti sono quelli aventi indirizzi non ancora tradotti in virtuali e quindi tutti quelli che circolano a stretto contatto con la parte *southbound* dell'architettura.

Comunicazione dal progetto ad OVX

Successivamente si è evoluta la soluzione permettendole di comunicare con il mittente (ovvero OVX). Questo si traduce in un aumento della complessità lato python e nell'aggiunta di codice all'interno della classe `PhysicalSwitch.java`. Più in particolare, dal lato del PhysicalSwitch si è scelto di gestire questa complessità attraverso la creazione di un apposito thread, il quale sfrutta la

Figura 4.4: Struttura messaggio da OpenVirteX alla soluzione proposta.

stessa porta che viene utilizzata per comunicare con la soluzione. In questo modo vengono adoperate limitate risorse nel sistema e la comunicazione tra le parti avviene parallelamente al funzionamento principale dell'architettura, suddividendo così i vari compiti. Nel codice, la classe di questo thread prende il nome di *InputStreamThread*. In Figura 4.5 viene riportata la soluzione fino a questo punto. Lato python, le varie parti del messaggio si sono ottenute attraverso lo *split* dei byte del messaggio letto. Si ricorda che il numero di byte letti è specificato come primo campo del messaggio. I campi interessanti del pacchetto che dall'esterno si riesce a leggere o a dedurre sono i seguenti.

- ***mac_ src***: indirizzo MAC dell'host che invia il particolare pacchetto.
- ***mac_ dst***: indirizzo MAC dell'host che riceve il particolare pacchetto.
- ***ip_ src***: indirizzo IP dell'host che invia il particolare pacchetto.
- ***ip_ dst***: indirizzo IP dell'host che riceve il particolare pacchetto.
- ***switch***: switch verso il quale il pacchetto è diretto. In questo caso si è mantenuta la denominazione che si è utilizzata dall'inizio. Quindi lo switch 1 ha nome `S1`, lo switch 2 ha nome `S2` e così via.
- ***virtual network***: numero della particolare rete virtuale alla quale appartiene il pacchetto. In base a questa vengono tradotti gli indirizzi

internamente ad OVX. In questo caso il numero si è dedotto dal campo *cookie* del pacchetto. Questo campo viene generato da OVX in base a quale virtual network appartiene il pacchetto.

- ***port_ in***: numero di porta dello switch dal quale è arrivato il pacchetto.
- ***port_ out***: numero di porta dello switch dal quale deve uscire il pacchetto.

Una volta che la soluzione rileva un pacchetto che soddisfa certi requisiti, questa genera un JSON da restituire a OVX in modo che lo riceva e, nel più breve tempo possibile, prenda una decisione circa le azioni da attuare.

Siccome si vuole che il traffico venga intercettato a fini di controllo, si è pensato di fare in modo che la soluzione rilevi dei pacchetti FlowMod (utilizzati per installare le regole nella Flow Table degli switch), ed in questo caso, inviare ad OVX un JSON contenente tutte le informazioni utili per riuscire a generare un nuovo pacchetto FlowMod. All'interno di questo pacchetto sono specificati tutti i dati circa il nuovo indirizzo di destinazione per il traffico proveniente da un certo host. Così facendo si replica tutto il traffico proveniente da un particolare individuo ritenuto rischioso in modo che, oltre ad installare nello switch il flusso corretto verso la normale destinazione del traffico, si vada ad installare anche questa nuova regola che devia questo flusso verso un altro particolare dispositivo che lo intercetta.

In seguito verranno fatte considerazioni in merito alla struttura ed alla creazione del pacchetto FlowMod appena descritto.

Generazione FlowMod

La macchina virtuale contenente OVX sul quale si è sviluppato il progetto supporta la versione del protocollo OpenFlow 1.0. Con questa versione, i campi contenuti nel pacchetto sono quelli della Figura 4.6.

Il campo *match* contiene un insieme di altri campi che lo switch utilizza per categorizzare i pacchetti entranti. I campi più importanti che si citano sono: ***dl_ src*** (*DataLayer Source* ovvero l'indirizzo MAC sorgente), ***dl_ dst*** (*DataLayer Destination* ovvero l'indirizzo MAC destinazione), ***dl_ type*** (*DataLayer Type* ovvero il tipo a cui il pacchetto appartiene), ***nw_ src*** (*Network Source* ovvero l'indirizzo IP sorgente), ***nw_ dst*** (*Network Destination* ovvero l'indirizzo IP destinazione) e ***in_ port*** (*Input Port* ovvero la porta dello switch dalla quale è arrivato il pacchetto). Come è possibile vedere, la maggior parte di questi campi sono gli stessi che la soluzione riesce a leggere dal JSON che

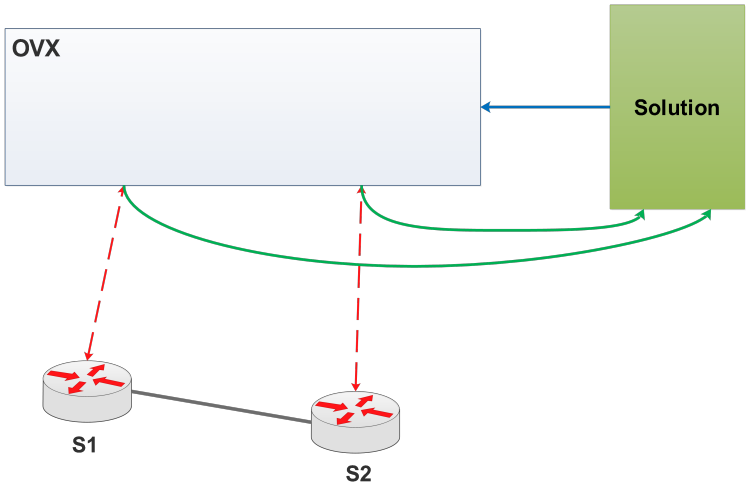


Figura 4.5: Secondo step: risposta verso OVX.

gli arriva da OVX. Questo facilita molto la creazione del campo *match* del pacchetto. Quindi ogni pacchetto entrante nello switch viene controllato per accertarsi che coincida con qualche regola di matching. Nel caso in cui non ci sia una corrispondenza con nessun campo *match* delle varie regole, lo switch richiede l'azione da attuare al rispettivo controller, come già discusso.

Oltre a questo campo ne sono presenti altri specificati di seguito [8, p. 27]:

- ***idle_timeout***: quantità di tempo (in secondi), che determina quanto un flusso in uno switch rimane fintanto che non arriva nessuna corrispondenza con i pacchetti entranti.
- ***hard_timeout***: quantità di tempo che determina quanti secondi un flusso può esistere in uno switch. Dopo questa durata il flusso viene rimosso anche a fronte del continuo arrivo di pacchetti che combaciano con il campo *match*.
- ***priority***: livello di priorità della flow entry (0 valore minimo).
- ***action[]***: insieme di azioni che lo switch deve attuare per gestire il flusso. Queste azioni specificano tutte le informazioni utili circa il comportamento

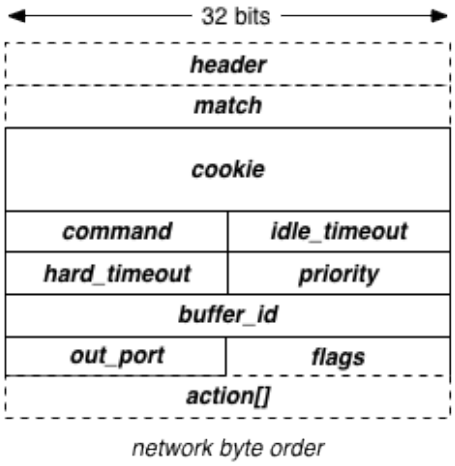


Figura 4.6: Struttura FlowMod.

che deve assumere lo switch. Per esempio possono specificare gli indirizzi di destinazione verso cui inviare il pacchetto.

Tutti questi campi sono stati forgiati in OVX per generare il FlowMod parallelamente a tutti i pacchetti che transitano abitualmente e che mantengono il comportamento standard della tecnologia. Quindi, nel caso in cui si volessero intercettare le comunicazioni di un host `X`, il FlowMod che verrà creato avrà come *match* tutte le informazioni utili per fare in modo che lo switch abbia la corrispondenza con dei particolari pacchetti da o verso `X`. Infine si inseriscono le varie azioni da applicare per replicare quei flussi verso una destinazione diversa in modo da controllarli.

Dopo aver introdotto il funzionamento base della soluzione pensata, verrà presentato il progetto che si è sviluppato: ***PySolv***.

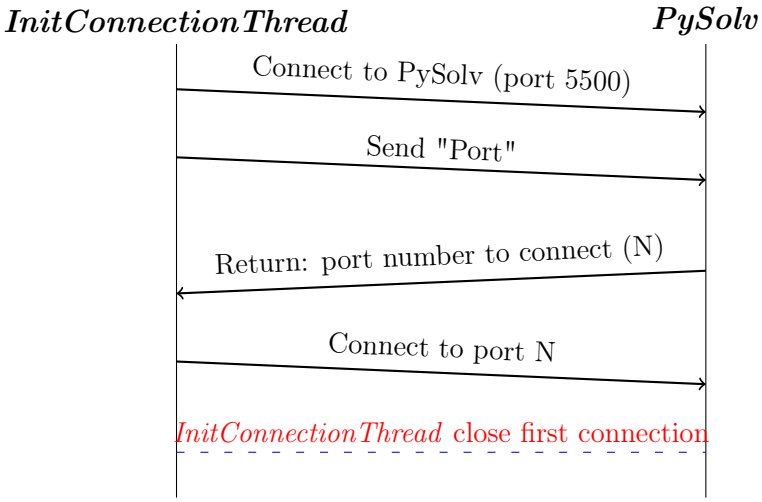
4.3 PySolv

Il nome che si è deciso di dare alla soluzione descritta nelle sezioni precedenti è *PySolv*.

Quello che verrà esposto di seguito riguarda la gestione dei vari sottoprocessi all'interno di OVX e di PySolv, eseguiti in modo concorrente al normale funzionamento di OVX.

Gestione e comportamento dei thread in OVX

Come già detto in precedenza, all'interno di OVX si è agito sulla classe `PhysicalSwitch.java` che rappresenta gli switch fisici della rete. Oltre al normale funzionamento di questa classe, si sono creati due thread: il primo per gestire lo scambio di informazioni con PySolv ed il secondo per inviare al relativo switch fisico il FlowMod. Il primo thread che si crea è chiamato *InitConnectionThread* ed ha la funzione di instaurare la connessione con PySolv. Più in particolare lo scambio iniziale che avviene tra questo thread e PysSolv è il seguente:

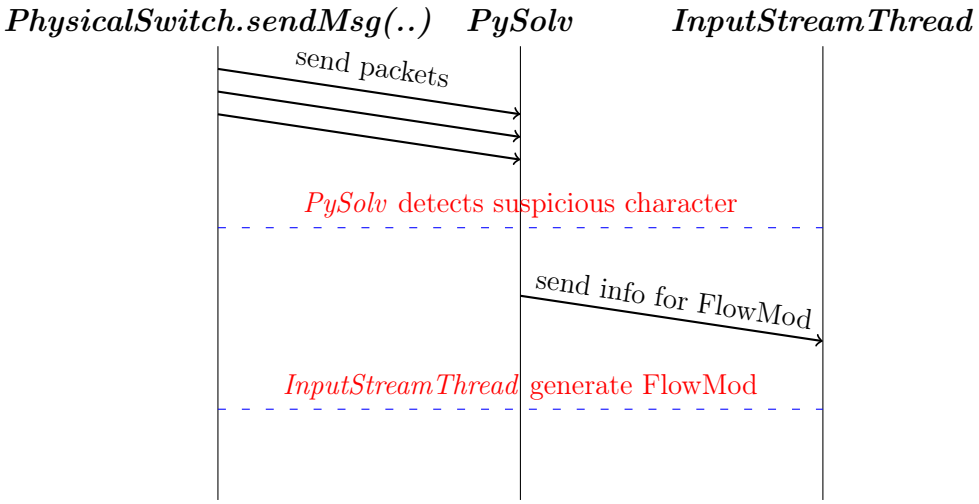


Inizialmente il thread si connette ad una porta prefissata di PySolv (5500 in questo caso), invia la stringa "Port" e PySolv risponde con un numero intero a quattro cifre contenente il numero di porta esclusivo per quel particolare switch. Infine il thread si connette a questa porta e, da questo punto in poi, restituisce la socket già connessa che PhysicalSwitch utilizzerà per comunicare direttamente con PySolv. Tutto questo avviene mentre l'intero sistema esegue il suo normale comportamento. Quindi, ogni volta che un PhysicalSwitch viene creato, si crea anche un *InitConnectionThread* con il compito di instaurare una connessione affidabile con PySolv. Creata quest'ultima, il thread viene terminato.

Una volta stabilita la connessione con PySolv sulla nuova porta, è possibile inviare tutti i pacchetti FlowMod che PhysicalSwitch invia agli switch fisici, anche verso la soluzione. OpenVirteX utilizza la nuova connessione che

InitConnectionThread restituisce, per inviare dati a PySolv. La struttura del messaggio è stata discussa nella Sezione 4.2. Sempre all'interno della classe PhysicalSwitch è presente il metodo `sendMsg(...)`; questo metodo è utilizzato per inviare i pacchetti che arrivano dalla *nortbound* verso il piano dei dati. Quindi, se è presente la connessione con PySolv, ogni volta che il funzionamento standard della tecnologia invoca questo metodo, viene generato anche un flusso di informazioni verso l'esterno per fare in modo che venga controllato.

Tutte le volte che PySolv rileva un individuo da intercettare, crea un JSON da restituire a OVX contenente tutti i campi necessari per generare il FlowMod. Quest'azione di ricezione da parte di OVX viene fisicamente effettuata da un altro thread denominato *InputStreamThread*. Esso ha il compito di gestire tutti i dati che provengono da PySolv in modo da non interferire con il comportamento standard di OVX. L'ultima azione che esegue *InitConnectionThread* prima di terminare è creare questo thread. Ciò viene fatto in modo da consegnare a *InputStreamThread* la socket già connessa verso PySolv. Successivamente quello che avviene è descritto nella seguente immagine:



- Tutte le volte che un pacchetto FlowMod sta per essere inviato al rispettivo switch fisico, il metodo `sendMsg(...)` invia anche una copia di questo pacchetto (con anche altre informazioni), verso PySolv.
- PySolv analizza queste informazioni:

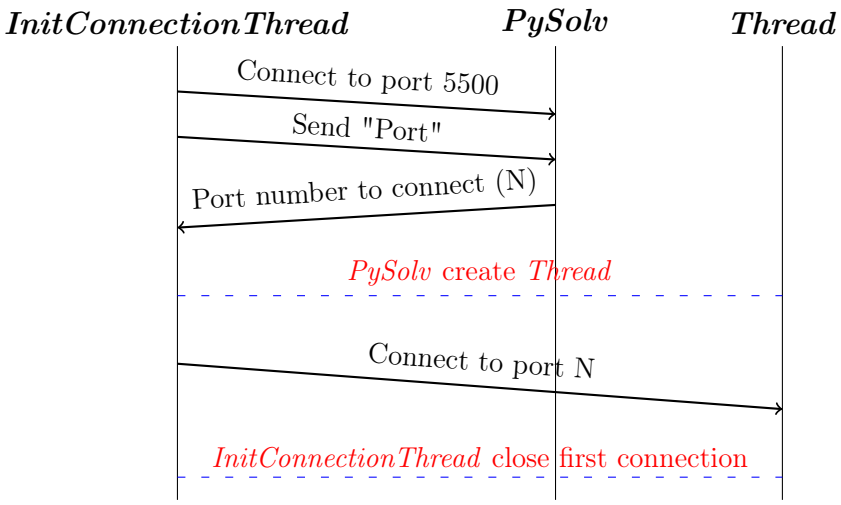
  - Nel caso in cui queste informazioni non siano di interesse per gli scopi di controllo, allora non viene restituito nulla al *InputStreamThread*.
  - Nell'altro caso invece, PySolv crea un JSON da restituire al thread con tutte le informazioni per intercettare il traffico.

- PySolv risponde al *InputStreamThread* nel caso in cui ci sia da generare un pacchetto FlowMod da inviare allo switch. In questo caso è stato creato un metodo apposta per generarlo chiamato `createFlowMod(...)`[4].

Gestione e comportamento dei thread in PySolv

Dal lato di PySolv, il comportamento tenuto è il seguente. Dopo che è stata fatta partire la soluzione, le viene assegnato un indirizzo locale su una determinata porta per rimanere in attesa che qualcuno si connetta. Come già discusso in precedenza, all'interno di OVX è presente il thread *InitConnectionThread*, il quale si connette a questa porta per ricevere un altro numero di porta a cui successivamente si andrà a connettere per proseguire il funzionamento del sistema. PySolv gestisce ogni connessione attraverso la creazione di un thread, ogni volta che un *InitConnectionThread* richiede la porta alla quale connettersi successivamente. In questo modo tutti i thread che vogliono connettersi a PySolv vengono reindirizzati su una porta diversa, gestita da un thread separato ed eseguito in modo concorrente a tutti gli altri.

[4]Successivamente sono stati creati diversi metodi da utilizzare nella gestione di una particolare tipologia di pacchetti. Per maggiori informazioni si veda la Sezione 4.4



Quindi tutti i futuri scambi di informazioni avvengono tra questo il nuovo thread interno a PySolv, *InputStreamThread* che riceve dati da quest'ultimo e *PhysicalSwitch* (attraverso il metodo `sendMsg(...)`).

4.3.1 Come avviene l'intercettazione

Per fare in modo che i flussi appartenenti ad un host vengano intercettati si ha bisogno di un dispositivo (chiamato *sniffer*), che li riceva e li elabori. La parte dell'elaborazione dei flussi esula da quello che è lo scopo della tesi ma, per completezza, si è deciso di fornire una soluzione per lo sniffer adibito all'intercettazione. Quest'ultimo deve essere un dispositivo connesso alla rete (in modo che riceva tutti i pacchetti del flusso), ed il suo indirizzo deve essere noto a PySolv. Questo perché è proprio PySolv che invia ad OVX tutti i dati per poi creare il FlowMod e, fra questi dati, ci deve essere anche l'indirizzo dello sniffer. Per implementare ciò, colui che andrà ad agire sull'infrastruttura per fare in modo che i dati vengano replicati, dovrà essere a conoscenza di tutti gli indirizzi degli host e di tutte le loro connessioni con i vari switch. Questo perché si fornisce a PySolv un file di configurazione (*config.json*) dentro al quale sono presenti tutte le informazioni per i dispositivi che devono ricevere i dati intercettati di tutte le virtual networks.

I campi principali di questo file di configurazione e riguardanti gli sniffer sono:

- ***ip_ sniff***: indirizzo IP dello sniffer.
- ***mac_ sniff***: indirizzo MAC dello sniffer.
- ***port_ sniff***: porta dello switch alla quale è connesso lo sniffer.

Da questi campi si deduce che si deve essere profondamente a conoscenza della rete e di tutti i suoi dispositivi, perché se solo si vanno a cambiare alcuni indirizzi di alcuni host, possono nascere conflitti con questi ultimi.

Gli altri campi presenti in questo file sono usati internamente da PySolv e da OVX in modo che si possa configurarli senza andare a modificare direttamente il codice sorgente. Questi sono ad esempio l'indirizzo al quale si devono *bindare* e la porta su cui rimanere in ascolto delle richieste che arrivano o che partono da OVX in base al relativo scopo. Per il momento si è implementato tutto il progetto in modo che funzioni in locale ovvero con indirizzo `127.0.0.1`.

Vincoli

Al giorno d'oggi se si volesse tenere sotto controllo l'intero traffico, basterebbe avere un singolo sniffer che, connesso alla stessa rete degli host, si interponga al normale flusso di pacchetti per ottenere i dati desiderati. La presenza di un singolo sniffer nella tecnologia OVX non potrebbe esistere perché, se così fosse, questo sniffer dovrebbe riuscire a vedere tutto il traffico di tutte le reti virtuali e questo non è possibile essendo la separazione dei vari flussi un obiettivo di primaria importanza.

Quindi, nel caso limite in cui si volesse tener sotto controllo tutta la rete in un'ottica OpenVirteX, ogni virtual network dovrebbe avere almeno uno sniffer al suo interno. A differenza delle reti odierne, in questo caso c'è bisogno di avere più dispositivi connessi ed ognuno dei quali facente parte di una virtual network differente. Inoltre, nel caso fossero presenti dei *big switch* ovvero aggregati di switch che l'utilizzatore interpreta come un unico switch, si dovrebbe pensare ad un modo per posizionare lo sniffer nel posto più conveniente.

4.4 Caso di studio

In questa sezione viene presentato un caso di studio, per mettere in pratica tutto quello che finora è stato descritto ed illustrato. Si è pensato di mostrare un esempio semplice in quanto quello che si spera è che si capisca il funzionamento

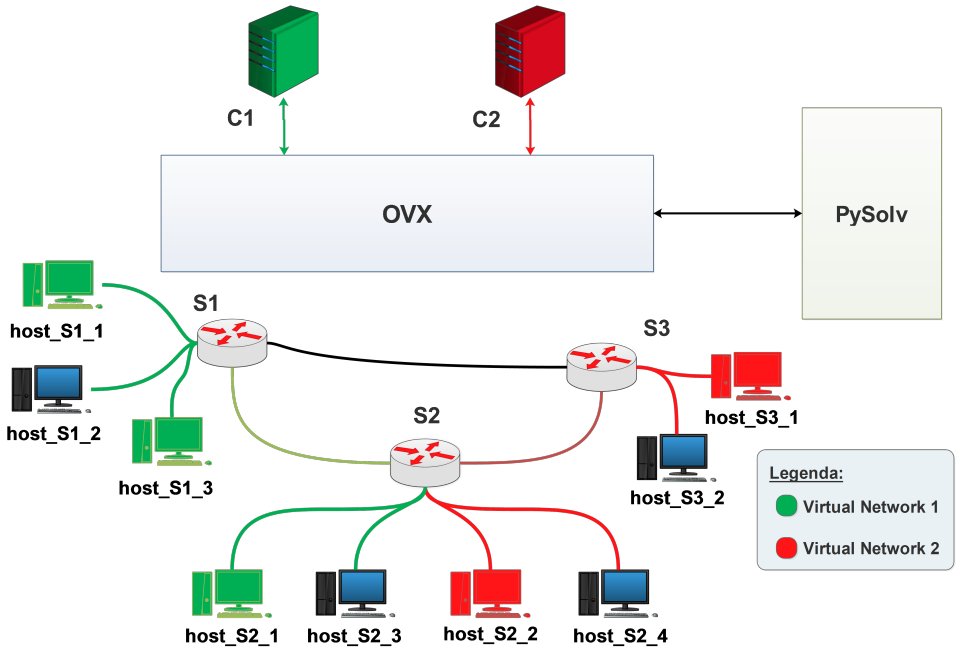


Figura 4.7: Topologia di rete del caso di studio.

base del progetto. Per aumentare la difficoltà non si deve fare altro che ingrandire la topologia di rete ed aggiungere altre reti virtuali. Si ricorda che, ogni volta in cui si aggiungono reti virtuali o si ingrandisce la topologia, si deve aggiornare il file di configurazione sopra citato, perché c'è la possibilità che i vari indirizzi cambino.

La topologia di rete che si presenta (Figura 4.7), è formata da tre switch (`S1`, `S2` e `S3`), ognuno dei quali ha cinque host connessi[5]. Al di sopra di questa infrastruttura si posizionano due controller che comandano le due rispettive reti virtuali. Ognuna di queste è formata da due switch e cinque host per la prima e quattro host per la seconda.

Come è possibile notare, nella Tabella 4.1 è presente il campo *Host sniffer*. Questo è dovuto al fatto che, siccome non si hanno dei dispositivi fisici da connettere alla rete che svolgano la funzione di sniffer, si è scelto di nominare degli host interni alla rete. Questo perché gli host, proprio come un normale sniffer di rete, sono caratterizzati da un indirizzo IP e un indirizzo MAC, e

[5]Denominati come già in precedenza descritto. Nel caso degli host connessi allo switch `S1`, essi avranno i seguenti nomi: `host_S1_1`, `host_S1_2`, `host_S1_3`, `host_S1_4` e `host_S1_5`.

Tabella 4.1: Composizione delle reti virtuali nel caso di studio.

Virtual Network	Switch	Host usati	Host sniffer
1	S1, S2	host_S1_1, host_S1_2, host_S1_3, host_S2_1, host_S2_3	host_S1_2, host_S2_3
2	S2, S3	host_S3_1, host_S3_2, host_S2_2, host_S2_4	host_S2_4, host_S3_2

quindi possono essere usati come base per un futuro sviluppo in cui, al posto di questi ultimi, è possibile posizionarci un vero e proprio dispositivo che raccolga informazioni e prenda decisioni circa il traffico degli individui intercettati. Per creare queste reti virtuali non si è fatto altro che seguire i passi descritti nella Sezione 2.2, con anche la creazione della topologia in un file python dal quale si è fatto partire l'emulatore Mininet.

Nota sugli sniffer Si è deciso di inserire in ogni switch un numero di sniffer pari al numero di reti virtuali che utilizzano questo switch. Quindi, se per esempio si osserva lo switch `S2`, si noterà che esso viene utilizzato sia dalla virtual network 1 che dalla virtual network 2 e perciò si sono connessi due sniffer. Una seconda soluzione viene presentata nella Sezione 4.4.5, mentre un possibile sviluppo viene analizzato nella Sezione 5.1.

Un'altra tabella degna di nota è la Tabella 4.2. In questa sono presenti tutte le informazioni circa i vari indirizzi fisici e virtuali di tutti gli host della rete fisica.

Dal nome di un host si può dedurre facilmente la porta fisica alla quale è connesso al relativo switch. Per esempio l'host `host_S1_3` è connesso alla porta 3 del relativo switch (ovvero `S1`). Inoltre, per gli indirizzi virtuali non si sono inserite le informazioni circa il valore esatto dell'indirizzo. Questo perché variano tutte le volte che si avvia OpenVirteX e quindi non sono predicibili a priori.

Prima di proseguire, è necessario inserire i dati degli sniffer insieme ad altri campi, all'interno del file di configurazione *config.json*. Se per esempio, si volesse venire a conoscenza dell'indirizzo IP di un particolare host nella rete, è

Tabella 4.2: Informazioni indirizzi degli host del caso di studio.

Host	MAC address	Physical IP address	Virtual IP Address
host_S1_1	00:00:00:00:01:01	10.0.0.11	1.0.0.?
host_S1_2	00:00:00:00:01:02	10.0.0.12	1.0.0.?
host_S1_3	00:00:00:00:01:03	10.0.0.13	1.0.0.?
host_S2_1	00:00:00:00:02:01	10.0.0.6	1.0.0.?
host_S2_3	00:00:00:00:02:03	10.0.0.8	1.0.0.?
host_S2_2	00:00:00:00:02:02	10.0.0.7	2.0.0.?
host_S2_4	00:00:00:00:02:04	10.0.0.9	2.0.0.?
host_S3_1	00:00:00:00:03:01	10.0.0.1	2.0.0.?
host_S3_2	00:00:00:00:03:02	10.0.0.2	2.0.0.?

possibile scoprirlo aprendo un terminale di quest'ultimo partendo dal terminale di mininet e digitando il seguente comando:

```
mininet> xterm host_S1_1
```

Una volta che si è aperta la nuova finestra, si digiti il comando `ifconfig` per stampare tutte le informazioni di un host tra cui anche l'indirizzo IP. Si è impostato il file di configurazione in modo che gli host `host_S1_2`, `host_S2_3`, `host_S2_4` e `host_S3_2` siano trattati come gli sniffer delle virtual network a cui appartengono.

Successivamente si eseguono OVX e PySolv in modo che inizino a scambiarsi messaggi. Se si lascia funzionare il sistema senza eseguire nessuna particolare azione tra gli host (quindi senza invii di FlowMod da parte di OVX), si noterà che gli unici pacchetti che vengono scambiati sono quelli appartenenti al protocollo LLDP, usati dagli switch per informare circa la propria identità, capacità e vicinanza con altri dispositivi. Una volta impostato il file di configurazione ed avviato il sistema, è possibile vedere il funzionamento di tutte le procedure che si sono spiegate nei paragrafi precedenti.

Attuare l'intercettazione

Di seguito sono descritti tutti i passi messi in atto dal sistema per intercettare un particolare host della rete. Prima di addentrarsi, si deve fare una chiara

distinzione per quanto riguarda i vari flussi installati all'interno di uno switch. Questa distinzione è dettata dal fatto che questi flussi vengono installati in maniera diversa; più precisamente le azioni sono installate secondo un certo ordine che varia a seconda di due valori fondamentali. Questi due sono i valori della porta di input (*input port*), del pacchetto che ha generato il FlowMod e la conseguente porta di output (*output port*), specificata tra le azioni all'interno dello stesso pacchetto. In base a questi due valori si sono dovute intraprendere strade diverse per quanto riguarda la creazione dei FlowMod generati da PySolv. Più in particolare si sono considerate quattro casistiche fondamentali[6]:

Caso I $in\_port \leq fanout$, $out\_port \leq fanout$; caso in cui il FlowMod inviato da OVX gestisca dei pacchetti in arrivo da un host connesso direttamente allo switch, e installa un'azione verso un altro host sempre connesso al medesimo switch.

Caso II $in\_port > fanout$, $out\_port \leq fanout$; caso in cui il FlowMod inviato da OVX gestisca dei pacchetti in arrivo da un altro switch della rete (essendo che `in_port > fanout`), e installa un'azione verso un host connesso allo switch che riceve questo FlowMod.

Caso III $in\_port \leq fanout$, $out\_port > fanout$; caso in cui il FlowMod inviato da OVX gestisca dei pacchetti in arrivo da un host connesso direttamente allo switch, e installa un'azione verso un altro switch della rete connesso a quello che riceve il FlowMod.

Caso IV $in\_port > fanout$, $out\_port > fanout$; caso in cui il FlowMod inviato da OVX gestisca dei pacchetti in arrivo da uno switch della rete connesso allo switch interessato, e installa un'azione verso un altro switch sempre connesso quest'ultimo.

Sono state fatte queste categorizzazioni perché in fase di implementazione, ci si è accorti che analizzando le azioni inserite nelle Flow Table da OVX, l'ordine di queste ultime cambia in base al valore delle due porte.

Si passa ora alla descrizione dettagliata di questi quattro casi d'interesse.

[6]Si ricorda che il campo *fanout* (specificato anch'esso nel file di configurazione), è il numero massimo di host connessi ad ogni switch. Le prime *fanout* porte di ogni switch vengono riservate per la connessione agli host, mentre tutte quelle maggiori di questo valore sono utilizzate per le varie connessioni tra switch.

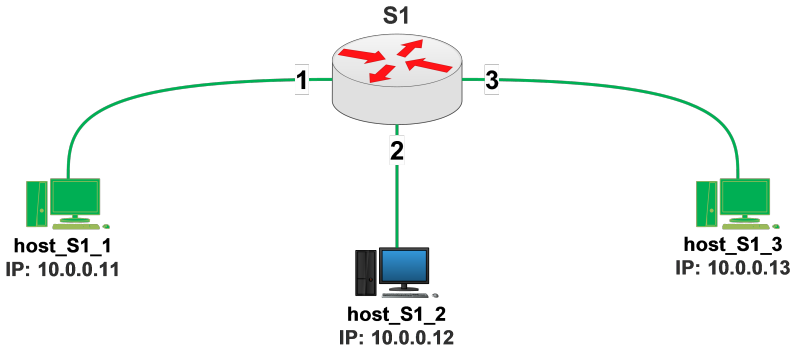


Figura 4.8: Caso I: comunicazione *intraswitch*.

4.4.1 Analisi Caso I: intra-switch

Inizialmente si mostra il caso su un singolo switch per poi ampliarlo negli altri casi con la presenza di più switch. In questo caso, come già indicato, si ha che i due valori di porta soddisfano il seguente requisito:

$$in\_port \leq fanout,\ out\_port \leq fanout$$

Questo requisito si traduce fisicamente in una configurazione di rete come quella presente nello switch `S1` e riportata in maniera più chiara in Figura 4.8.

Si sono analizzati i vari campi dei FlowMod che OVX invia per consentire una comunicazione di questo tipo. In particolare i campi di interesse sono quelli relativi al matching e alle azioni che vengono inserite.

Per quanto riguarda i campi del *matching*, si sono osservati i FlowMod generati da OVX e che sono giunti a PySolv. Si ricorda che a PySolv arriva un determinato messaggio con una struttura nota e all'interno del quale sono presenti i campi che OVX utilizza per confrontare tutti i pacchetti entranti nello switch. Si sono considerati questi stessi campi e si sono rispediti ad OVX insieme a tutte le informazioni per poter creare le azioni corrette e ordinate proprio come il FlowMod originale.
Per ottenere le *azioni* corrette da inviare insieme ai campi del matching, una volta eseguito OpenVirteX e create le reti virtuali si aspetta a far partire PySolv in modo che venga mantenuto solo il comportamento standard di OVX. Successivamente si esegue un semplice ping tra i due host `S1_1` e `S1_3` in modo

che vengano installate le varie azioni all'interno dello switch. Fatto ciò, da un altro terminale si esegue il comando per gestire i vari flussi negli OpenVSwitch[7]:

```
sudo ovs-ofctl dump-flows S1
```

.[8] L'output di questo comando è la lista di tutti i flussi installati all'interno dello switch specificato e che di seguito viene riportato formattato in modo da renderlo più leggibile:

Listing 4.1: Output comando nel Caso I.

```
cookie=0x100000003, duration=1.468s, table=0, n_packets=0,
n_bytes=0, idle_timeout=5, idle_age=1, priority=0,
    ↪ vlan_tci=0x0000,
/* Campi per Match */
  in_port=3,
  dl_src=00:00:00:00:01:03,
  dl_dst=00:00:00:00:01:01
/* Azioni del FlowMod */
actions=
  mod_nw_dst:1.0.0.4,
  mod_nw_src:1.0.0.3,
  mod_nw_src:10.0.0.13,
  mod_nw_dst:10.0.0.11,
  output:1

cookie=0x100000002, duration=1.579s, table=0, n_packets=0,
n_bytes=0, idle_timeout=5, idle_age=1, priority=0,
    ↪ vlan_tci=0x0000,
/* Campi per Match */
  in_port=1,
  dl_src=00:00:00:00:01:01,
  dl_dst=00:00:00:00:01:03
/* Azioni del FlowMod */
actions=
  mod_nw_dst:1.0.0.3,
  mod_nw_src:1.0.0.4,
  mod_nw_src:10.0.0.11,
  mod_nw_dst:10.0.0.13,
```

[7] Link alla documentazione del comando: `http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt`.

[8] La durata delle azioni che OpenVirteX installa all'interno degli switch è di 5 secondi dall'ultimo momento nel quale vengono utilizzate. Quindi si consiglia di mantenere attivo il ping tra i due host in modo che le azioni rimangano installate per più tempo.

```
output:3
```

L'analisi di questo output ha permesso di inserire nella sequenza corretta le varie azioni da PySolv. Si è tenuto conto dell'ordine e della tipologia di indirizzi presenti. Si noti che per questa tipologia le azioni *ordinate* da inserire nel FlowMod sono:

1. **mod_nw_dst**: indirizzo IP ***virtuale*** dell'host di destinazione;
2. **mod_nw_src**: indirizzo IP ***virtuale*** dell'host sorgente;
3. **mod_nw_src**: indirizzo IP ***fisico*** dell'host sorgente;
4. **mod_nw_dst**: indirizzo IP ***fisico*** dell'host di destinazione;
5. **output**: porta fisica attraverso la quale far uscire il pacchetto da gestire.

A queste azioni si sono aggiunte quelle relative all'intercettazione. Per l'aggiunta di queste azioni si veda la Sezione 4.4.4.

4.4.2 Analisi Caso II e III: inter-switch

Entrambi questi casi sono caratterizzati dalla presenza di due switch. Nel caso precedente, il pacchetto FlowMod che giungeva a PySolv doveva gestire dei pacchetti entranti da una certa porta connessa ad un host e spedire verso un'altra porta sempre connessa ad un host connesso direttamente con lo switch. In questi due casi, invece, si ha che il pacchetto FlowMod che PySolv si vede arrivare deve saper gestire pacchetti che arrivano da una porta che non è connessa direttamente ad un host e spedirli verso una porta connessa ad un host, o viceversa. Quindi i requisiti che, rispettivamente, devono verificare sono i seguenti:

$$\text{Caso II: } in\_port > fanout,\ out\_port \leq fanout$$
$$\text{Caso III: } in\_port \leq fanout,\ out\_port > fanout$$

.

Per un esempio grafico si veda la Figura 4.9. Questo caso copre tutti i FlowMod che gestiscono, per esempio, tutti i pacchetti che da un host connesso a `S1` vanno verso un host connesso a `S2`. Si tenga presente che il valore di *fanout* di questi esempi è 5.

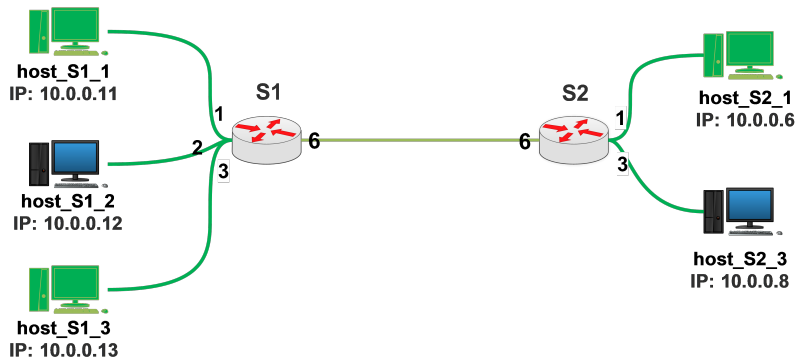


Figura 4.9: Caso II: comunicazione *interswitch*.

L'output del comando che si è usato anche nel caso precedente e che, grazie a questo, si è riusciti a dedurre le azioni da immettere nel nuovo FlowMod, è il seguente:

Listing 4.2: Output comando nel Caso II.

```
cookie=0x100000002, duration=0.712s, table=0, n_packets=0,
n_bytes=0, idle_timeout=5, idle_age=0, priority=0,
    ↪ vlan_tci=0x0000,
/* Campi per Match */
  dl_src=a4:23:05:01:00:00,
  dl_dst=a4:23:05:10:00:0d,
  in_port=6
/* Azioni del FlowMod */
actions=
  mod_nw_src:10.0.0.11,
  mod_nw_dst:10.0.0.6,
  mod_dl_src:00:00:00:00:01:01,
  mod_dl_dst:00:00:00:00:02:01,
  output:1
```

dove, per quanto riguarda le azioni, si ha:

1. **mod_nw_src**: indirizzo IP ***fisico*** dell'host sorgente;
2. **mod_nw_dst**: indirizzo IP ***fisico*** dell'host destinazione;
3. **mod_dl_src**: indirizzo MAC ***fisico*** dell'host sorgente;
4. **mod_dl_dst**: indirizzo MAC ***fisico*** dell'host di destinazione;

5. **output**: porta fisica attraverso la quale far uscire il pacchetto da gestire.

Listing 4.3: Output comando nel Caso III.

```
cookie=0x100000003, duration=0.556s, table=0, n_packets=0,
n_bytes=0, idle_timeout=5, idle_age=0, priority=0,
   ↪ vlan_tci=0x0000,
/* Campi per Match */
  dl_src=00:00:00:00:02:01,
  dl_dst=00:00:00:00:01:01,
  in_port=1
/* Azioni del FlowMod */
actions=
  mod_nw_dst:1.0.0.2,
  mod_nw_src:1.0.0.3,
  mod_dl_src:a4:23:05:01:00:00,
  mod_dl_dst:a4:23:05:10:00:08,
  output:6
```

dove:

1. **mod_nw_dst**: indirizzo IP ***virtuale*** dell'host di destinazione;
2. **mod_nw_src**: indirizzo IP ***virtuale*** dell'host sorgente;
3. **mod_dl_src**: indirizzo MAC ***virtuale*** dello switch sorgente;
4. **mod_dl_dst**: indirizzo IP ***virtuale*** dello switch di destinazione;
5. **output**: porta fisica attraverso la quale far uscire il pacchetto da gestire.

Anche in questo caso, oltre a queste azioni si sono aggiunte quelle relative all'intercettazione. Si veda la Sezione 4.4.4 per queste azioni.

4.4.3 Analisi Caso IV: switch multipli

Ultimo caso che si è gestito. Rispetto ai precedenti, questa casistica è caratterizzata dalla presenza di soli switch nello scambio di pacchetti. Questo perché può capitare che un pacchetto raggiunga uno switch ma che non abbia come destinazione un host connesso a quest'ultimo. Quindi il pacchetto che transita, entra da una porta connessa ad uno switch ed esce da un'altra connessa ad un

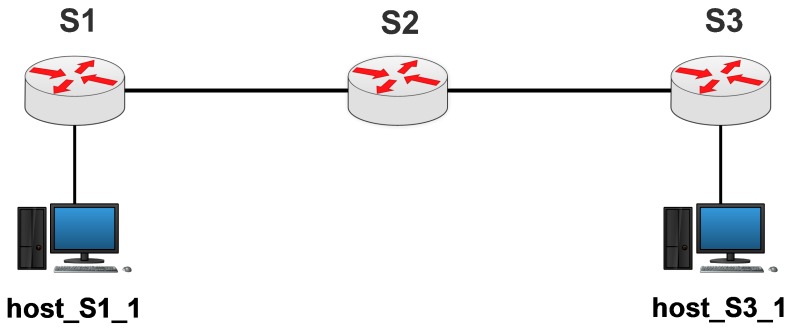


Figura 4.10: Caso IV: "transito" in uno switch

altro switch grazie al FlowMod che verrà installato. Tradotto in maniera più ridotta e con meno giri di parole, questa casistica gestisce tutti i casi in cui:

$$in\_port > fanout,\ out\_port > fanout$$

In Figura 4.10 viene presentato un esempio grafico. Si consideri lo switch `S2`. A fronte di uno scambio tra i due host presenti, questo switch non fa altro che essere il tramite tra i due switch connessi agli host. Quindi verrà installata un'azione da un pacchetto FlowMod contenente, sia nell'area per il matching che nell'area delle azioni, due porte con valore strettamente maggiore di *fanout*[9]. Di seguito è presente l'output del comando `sudo ovs-ofctl dump-flows S2` relativo allo switch in questione.

Listing 4.4: Output comando nel Caso IV.

```
cookie=0x100000002, duration=0.333s, table=0, n_packets=0,
n_bytes=0, idle_timeout=5, idle_age=0, priority=0,
    ↪ vlan_tci=0x0000,
/* Campi per Match */
  dl_src=a4:23:05:01:00:00,
  dl_dst=a4:23:05:10:00:02,
  in_port=6
/* Azioni del FlowMod */
actions=
  mod_dl_src:a4:23:05:01:00:00,
  mod_dl_dst:a4:23:05:20:00:02,
  output:7
```

[9]Si ricorda che i numeri di porta compresi tra $1 \leq n_{port} \leq fanout$ sono intese essere connesse agli host del relativo switch. Quindi le porte con valore maggiore di *fanout* sono da intendere essere connesse ad altri switch della rete.

con:

1. **mod_dl_src**: indirizzo MAC ***virtuale*** dello switch sorgente;
2. **mod_dl_dst**: indirizzo IP ***virtuale*** dello switch di destinazione;
3. **output**: porta fisica attraverso la quale far uscire il pacchetto da gestire.

4.4.4 Aggiunta sniffer

Una volta illustrati i vari campi per il matching (da impostare nei FlowMod che si andranno a creare insieme alla tipologia), e discusso dell'ordine con cui si devono inserire le azioni da indicare nello switch interessato, si spiega come si sono attuate le intercettazioni dei pacchetti. Non si sono fatte distinzioni sulle tipologie di pacchetto da intercettare, semplicemente si sono inserite tutte quelle azioni utili in modo che tutti i pacchetti che fluiscono in un determinato switch e che hanno come indirizzo sorgente o indirizzo destinazione quello da intercettare, vengano spediti anche ad uno sniffer tra quelli specificati nel file di configurazione. Ovviamente tutte queste manipolazioni non sono viste dai vari controller che, ignari di ciò, proseguono con il loro normale scambio di pacchetti.

Come già discusso, ogni volta che ad OVX viene imposto (da un controller risiedente nella *northbound*), di generare un pacchetto FlowMod per un particolare switch, esso "inoltra" le informazioni di questo pacchetto a PySolv. Quest'ultimo le elabora e, nei casi di interesse, rispedisce ad OVX tutte le informazioni per generare un nuovo FlowMod per permettere l'intercettazione.

Le caratteristiche di questo FlowMod sono simili a quelle dei FlowMod installati da OVX ma con qualche aggiunta e modifica. Per prima cosa si sono utilizzati gli stessi campi per fare il matching. Questo viene fatto in modo da assicurarsi che solo ed esclusivamente i pacchetti con certe caratteristiche (come indirizzi sorgente e destinazione) siano effettivamente intercettati e rispediti verso lo sniffer. Successivamente, siccome si vuole che all'arrivo dei pacchetti da intercettare lo switch scelga le nuove azioni che si andranno ad installare con questo FlowMod, si attribuisce un valore maggiore del campo *priority* in modo che lo switch sia costretto a scegliere queste azioni[10]. Oltre a questo vengono

[10]Lo switch decide le azioni da prendere in base a questo campo *priority*. A fronte di un pacchetto che combacia con il campo *match* di due azioni, quella che verrà applicata sarà quella più prioritaria tra queste due.

inserite le stesse identiche azioni che erano presenti nel vecchio pacchetto; questo perché si vuole che i pacchetto in questione raggiungano la corretta destinazione senza intoppi. Infine si è aggiunta l'azione che permette la fuoriuscita di queste tipologie di pacchetti verso una porta alla quale è connessa uno sniffer. i valori di queste porte sono inviati da PySolv e sono contenuti nel file di configurazione.

Approfondimento sulle azioni aggiunte A questo punto si è di fronte ad una scelta per quanto riguarda le azioni da inserire. La prima opzione è stata quella di inserirle in modo che i pacchetti che giungono allo sniffer avessero l'indirizzo destinazione modificato in quello dello sniffer. Questo si può fare aggiungendo delle azioni che permettano di riscrivere gli indirizzi mantenendo il resto dei campi inalterato. Più in particolare queste azioni sono `mod_nw_src`, `mod_nw_dst`, `mod_dl_src` e `mod_dl_dst` e sono state usate anche da OVX per camuffare i veri indirizzi fisici dei pacchetti. Ma questa non sarebbe la forma più corretta di intercettazione in quanto, per una buona intercettazione si dovrebbero mantenere il più inalterati possibile tutti i campi del pacchetto. La seconda scelta, invece, non modifica gli indirizzi sorgente e destinazione in modo da recapitare allo sniffer tutti i pacchetti con i reali indirizzi IP sia del mittente che del destinatario. In questo modo lo sniffer può eseguire maggiori analisi circa i traffici di dati. Ovviamente si è scelta questa seconda opzione ma si è voluto informare dell'eventualità dell'altra scelta.

Quindi, ricapitolando, si sono eseguiti i seguenti passaggi per i FlowMod creati:

- Stessi campi di matching;
- Maggiore priorità rispetto ai FlowMod creati dai controller di OVX;
- Aggiunta stesse azioni del FlowMod iniziale;
- Aggiunta di azioni per inoltrare il pacchetto verso lo sniffer connesso allo switch.

In Figura 4.11 vengono rappresentati in maniera grafica le differenze tra i pacchetti che invia OVX verso la southbound e quelli generati con PySolv.

Si è scelto questo comportamento perché è sembrato il più giusto da applicare. Quello che viene fatto è aggiungere una serie di azioni più prioritarie che devono

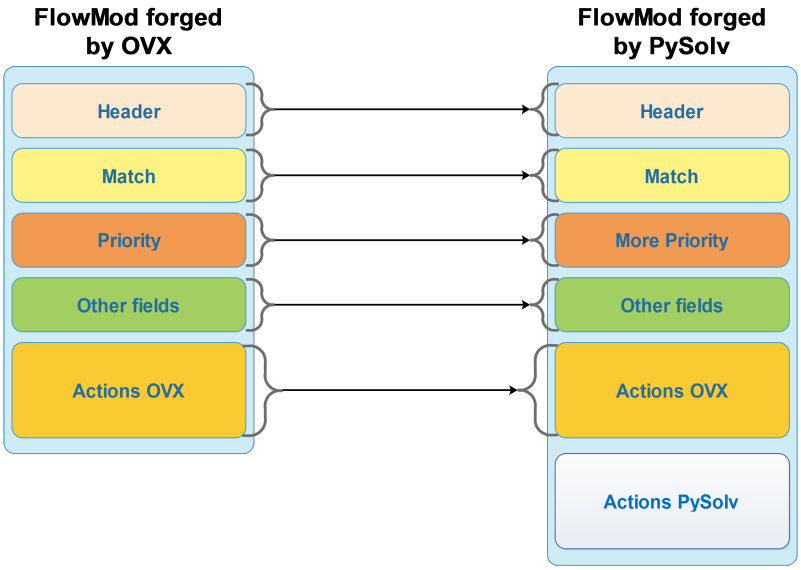


Figura 4.11: Differenza FlowMod generati da OVX e da PySolv.

essere scelte rispetto alle normali azioni installate da OVX. Quindi le azioni normali della tecnologia sono sempre presenti anche se non utilizzate.

Un'altra scelta procedurale poteva essere quella di *eliminare il FlowMod* precedentemente inviato da OVX inviando un pacchetto con all'interno specificati particolari comandi del tipo `DELETE` o `DELETE_STRICT` [8, pp. 27-29]. In questo modo, così come si è usato un pacchetto FlowMod per aggiungere una entry nella FlowTable, il pacchetto per rimuovere determinate azioni avrebbe dovuto contenere la descrizione dei pacchetti interessati. Questa scelta, però, è sembrata essere troppo invasiva. La scelta di questa opzione si sarebbe tradotta non in una sostituzione delle azioni utilizzando quelle più prioritarie ma proprio in una sostituzione radicale di queste ultime. Così facendo si è pensato di andare a modificare troppo quello che è il funzionamento di OVX e quindi si è abbandonata l'idea.

4.4.5 Altro approccio per l'intercettazione

Nella sezione precedente è stata presentata quella che è la soluzione per cercare di mantenere inalterata la proprietà più importante delle tecnologie di *slicing*: la separazione del traffico dei vari tenant. Questa è stata messa in atto

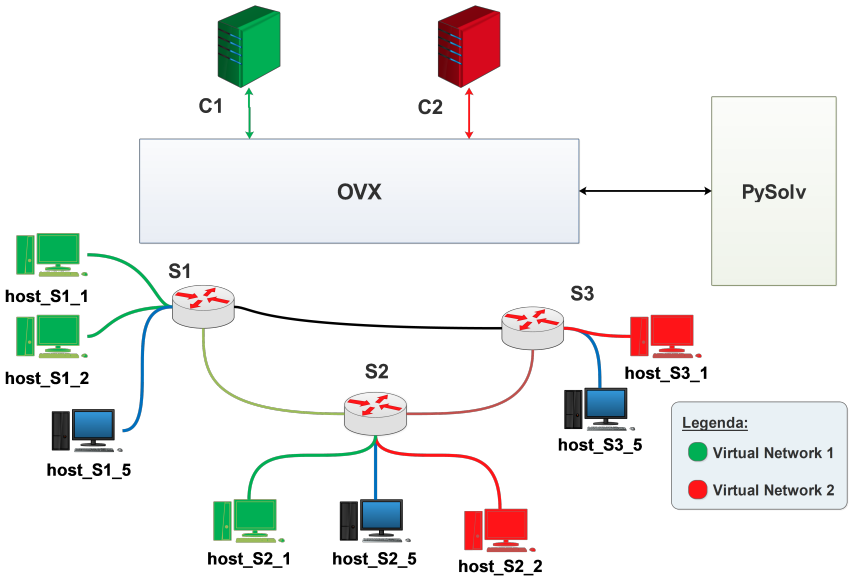


Figura 4.12: Topologia rivista del caso di studio.

semplicemente agendo sul numero degli sniffer connessi ad ogni switch. Come già detto, si è attribuito ad ogni switch un numero di sniffer pari al numero di virtual networks che agiscono su di esso. In questo modo il traffico rimane circoscritto all'interno di tutti e soli quei dispositivi connessi alla stessa rete virtuale.

L'altro approccio che si vuole descrivere e che si è testato, va a violare la proprietà della separazione. In sostanza quello che viene fatto è connettere un solo sniffer ad ogni switch, indipendentemente dal numero di reti virtuali al quale appartiene, con la conseguente deviazione del traffico verso quest'ultimo. In questo modo viene ridotto il numero di sniffer utilizzati; essi ricevono i pacchetti da tutte le reti virtuali presenti.

Passando più sul pratico, in Figura 4.12 viene illustrata la topologia di rete con questo approccio. I passi che sono stati fatti per giungere a questa soluzione alternativa sono praticamente gli stessi della soluzione precedente, tranne per alcune differenze. Per prima cosa si sono strutturati alcuni campi del file di configurazione in maniera diversa. Quello della soluzione precedente ha i vari sniffer raggruppati in base alla virtual network alla quale appartengono, in modo da gestire correttamente i vari invii di FlowMod. Il file di configurazione dell'attuale soluzione è reso più semplice dal fatto che gli sniffer non devono

"appartenere" ad una determinata virtual network; in sostanza si devono specificare meno informazioni a riguardo. Come svantaggio si ha che lo sniffer deve saper gestire la presenza di più reti virtuali. Questo si traduce nella capacità di differenziare i vari pacchetti entranti, in modo da saperli categorizzare per eventuali elaborazioni future. Nell'esempio proposto si sono connessi tutti gli sniffer alla porta 5 di ogni switch. Ogni volta che PySolv si accorge della presenza (negli indirizzi sorgente o di destinazione), di un host da intercettare, non si fa altro che inviare le stesse informazioni a OVX con la porta dello sniffer pari a 5.

Gestione BigSwitch

In tutti i casi di studio esaminati si è fatto sempre riferimento a topologie di rete virtuali, le quali rispecchiano quella che è la rete fisica. Questo implica che per ogni switch virtuale di ogni rete virtuale, è possibile risalire a quello fisico corrispondente. Si ha quindi una corrispondenza univoca tra l'insieme degli switch virtuali e quello degli switch fisici. In questi casi, esaminando i FlowMod in arrivo a PySolv da OVX si è in grado di risalire a tutte le informazioni utili del percorso, ed in questo modo è possibile gestire i pacchetti da creare *ad hoc* per deviare il traffico verso gli sniffer.

Caso diverso si ha quando al posto di uno switch virtuale mappato in ogni switch fisico, sono presenti più switch fisici mappati all'interno di uno virtuale o, come lo chiamano gli sviluppatori, *BigSwitch*[11]. OpenVirteX gestisce questi BigSwitch in modo che l'utilizzatore veda un unico switch. Anche in questo caso si sono esaminati i FlowMod che vengono scambiati nella parte *southbound*; si è notato che sono leggermente diversi da quelli generati per gli switch virtuali, associati ad un solo switch fisico. Anche in questo caso si è gestita questa eventualità. In particolare si è notato che, a differenza dei casi considerati in precedenza, nel caso in cui $in\_port > fanout$, $out\_port \leq fanout$ o $in\_port \leq fanout$, $out\_port > fanout$, le azioni che OVX invia agli switch non contengono i campi riguardanti il *data layer*: `set_dl_src` e `set_dl_dst`. Quindi si è agito in modo che a fronte dell'arrivo di un messaggio da PySolv, si controlli se le azioni di questo tipo sono presenti, cosicché si vada a creare il corretto pacchetto da inviare nella *southbound*.

[11] Rappresentato in java con la classe *OVXBigSwitch.java*

Per testare quest'ultimo caso si è creata una virtual network partendo dalla topologia di rete fisica dei casi precedenti e usando un BigSwitch contenente tutti e tre quelli fisici presenti.

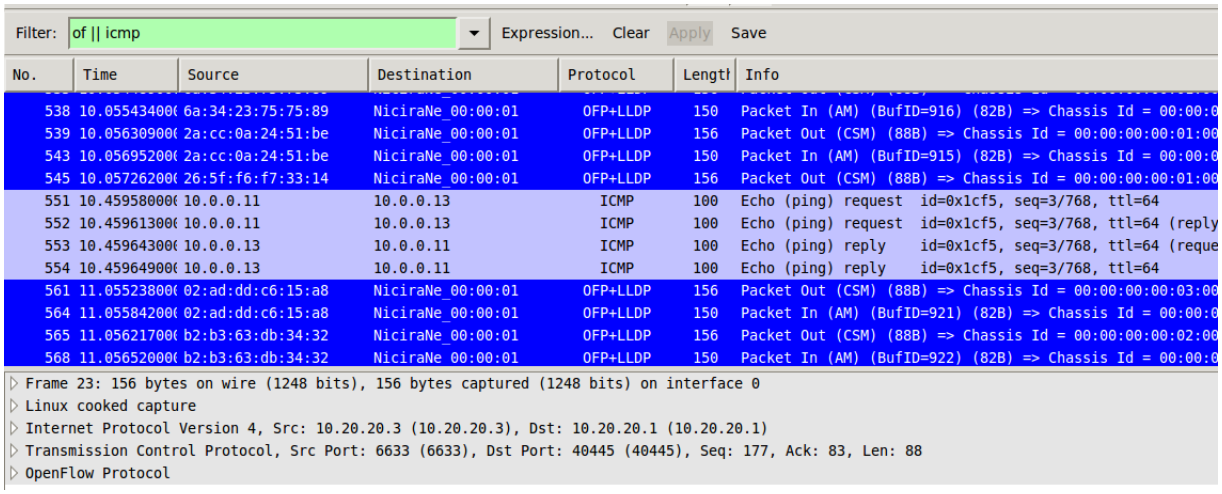
4.5 Prove dell'intercettazione

Per avere le prove certe che l'intercettazione sia avvenuta con successo è stato usato Wireshark. Infatti con l'aiuto di questo sniffer di rete, si è potuto verificare che tutte le operazioni fatte per creare un nuovo pacchetto e per inviarlo verso la *southbound* fossero corrette. Grazie a Wireshark è possibile posizionarsi in qualsiasi punto della rete dal quale fluiscono i pacchetti ed analizzarli. Quindi, oltre ad utilizzare l'interfaccia `any` per poter vedere tutti i pacchetti che circolano, si è posizionato Wireshark all'ingresso delle varie interfacce di ogni switch per osservare solo i pacchetti entranti o uscenti da quel punto.

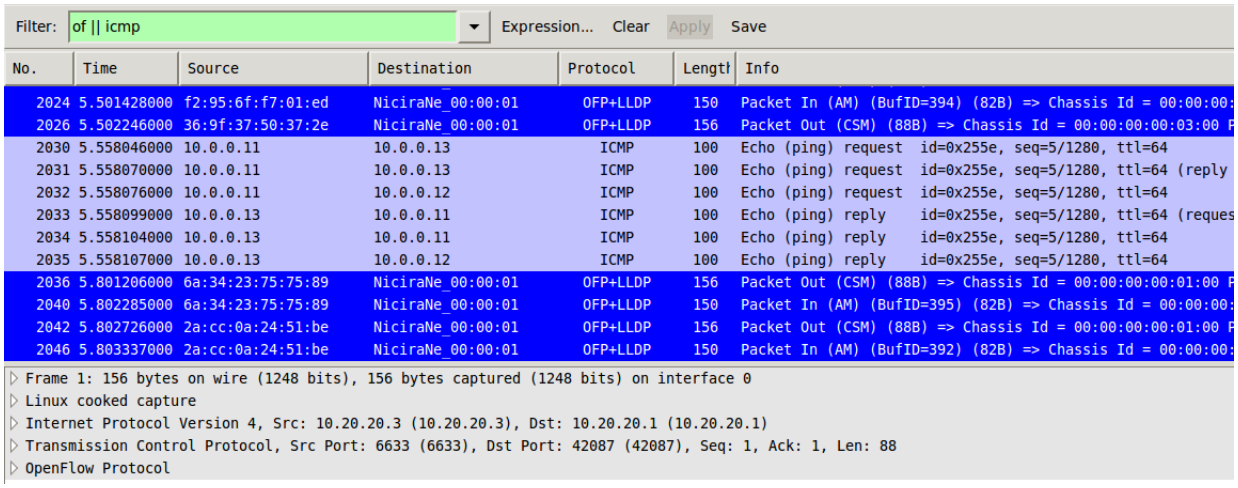
Se non si modificano le varie soluzioni descritte, negli indirizzi sorgente e destinazione si vedranno solo quelli dei due host che stanno comunicando. Questo fatto, come già discusso, è il modo corretto di agire siccome non vengono modificati i vari indirizzi dei pacchetti. Di conseguenza, però, non si capisce quando un pacchetto è diretto verso la giusta destinazione o verso lo sniffer. Quindi per chiarire meglio il funzionamento del sistema, si andranno a modificare questi indirizzi[12] per fare in modo che nelle immagini si riesca a distinguere la vera destinazione.

Le seguenti catture sono state eseguite partendo dalla topologia del caso di studio e servono per chiarire tutte le casistiche descritte in precedenza. Successivamente si descriveranno come sono state fatte, spiegando i risultati ottenuti.

[12] Gli indirizzi verranno modificati attraverso le azioni `mod_nw_src` e `mod_nw_dst` già descritte.

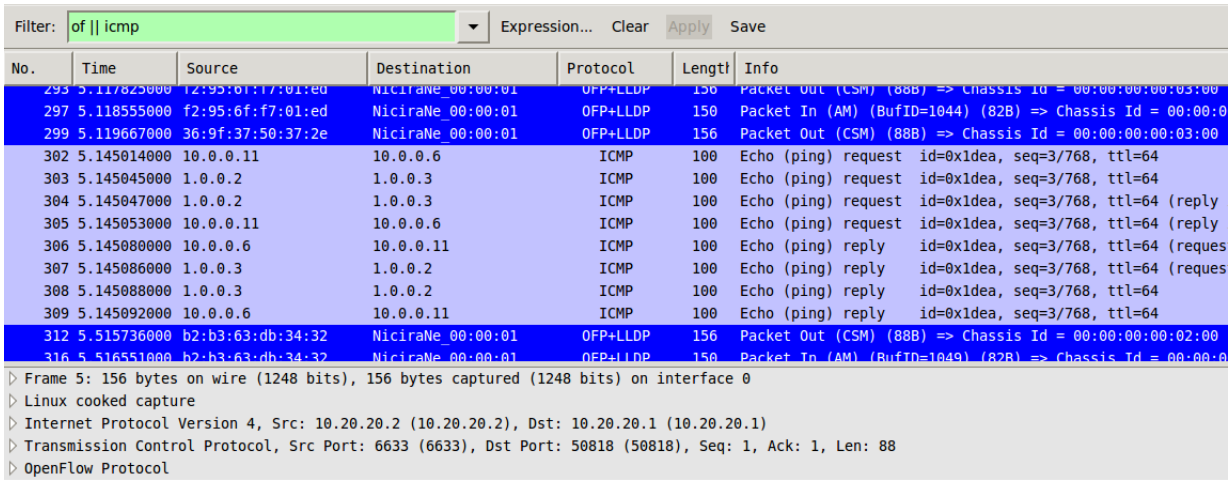


(a) Solo con OVX.

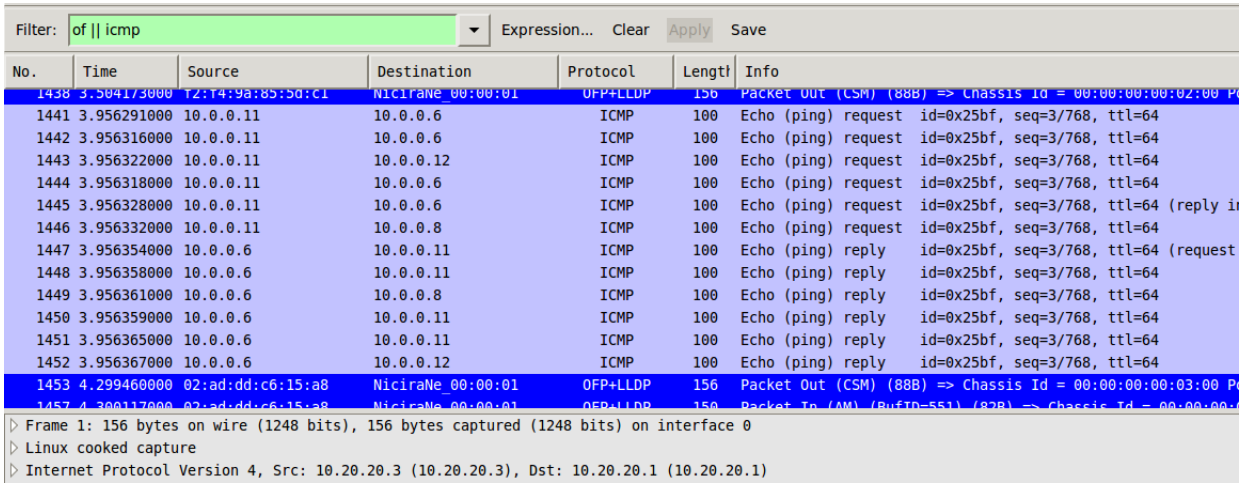


(b) Sia con OVX che con PySolv

Figura 4.13: Catture Caso I: $in\_port \leq fanout$, $out\_port \leq fanout$.

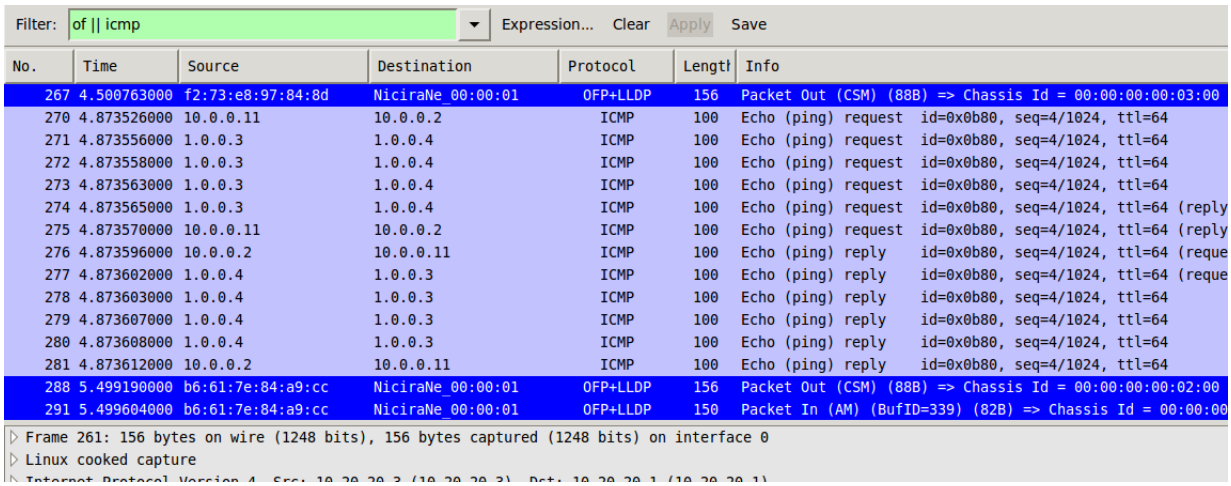


(a) Solo con OVX.



(b) Sia con OVX che con PySolv

Figura 4.14: Catture Caso II e III: $in\_port > fanout,\ out\_port \leq fanout$ e $in\_port \leq fanout,\ out\_port > fanout$.

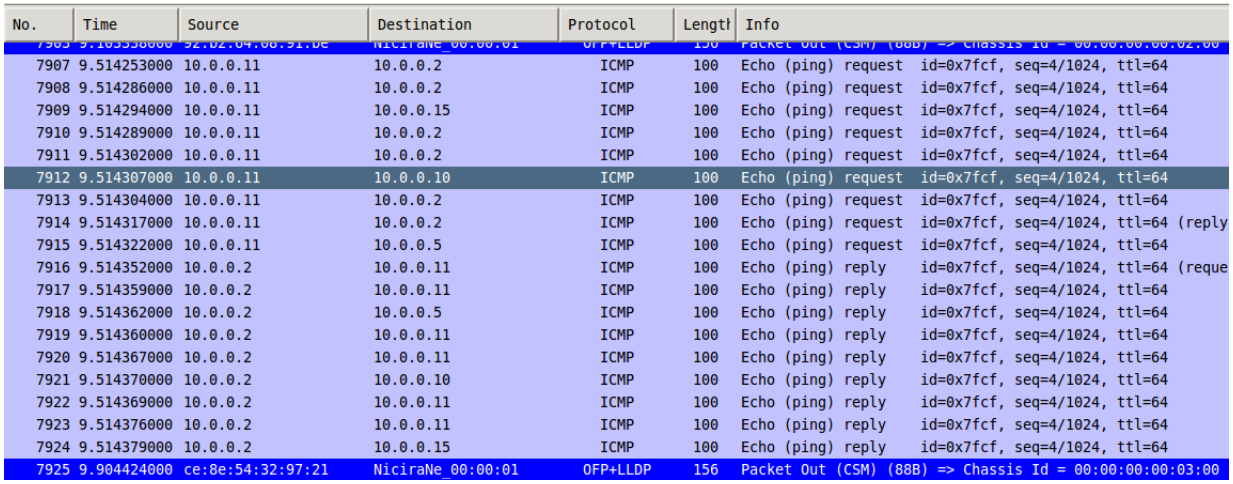


Filter: of || icmp — Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
267	4.500763000	f2:73:e8:97:84:8d	NiciraNe_00:00:01	OFP+LLDP	156	Packet Out (CSM) (88B) => Chassis Id = 00:00:00:00:03:00
270	4.873526000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64
271	4.873556000	1.0.0.3	1.0.0.4	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64
272	4.873558000	1.0.0.3	1.0.0.4	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64
273	4.873563000	1.0.0.3	1.0.0.4	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64
274	4.873565000	1.0.0.3	1.0.0.4	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64 (reply
275	4.873570000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x0b80, seq=4/1024, ttl=64 (reply
276	4.873596000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64 (reque
277	4.873602000	1.0.0.4	1.0.0.3	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64 (reque
278	4.873603000	1.0.0.4	1.0.0.3	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64
279	4.873607000	1.0.0.4	1.0.0.3	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64
280	4.873608000	1.0.0.4	1.0.0.3	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64
281	4.873612000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x0b80, seq=4/1024, ttl=64
288	5.499190000	b6:61:7e:84:a9:cc	NiciraNe_00:00:01	OFP+LLDP	156	Packet Out (CSM) (88B) => Chassis Id = 00:00:00:00:02:00
291	5.499604000	b6:61:7e:84:a9:cc	NiciraNe_00:00:01	OFP+LLDP	150	Packet In (AM) (BufID=339) (82B) => Chassis Id = 00:00:00

▷ Frame 261: 156 bytes on wire (1248 bits), 156 bytes captured (1248 bits) on interface 0
▷ Linux cooked capture

(a) Solo con OVX.



No.	Time	Source	Destination	Protocol	Length	Info
7907	9.514253000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7908	9.514286000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7909	9.514294000	10.0.0.11	10.0.0.15	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7910	9.514289000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7911	9.514302000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7912	9.514307000	10.0.0.11	10.0.0.10	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7913	9.514304000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7914	9.514317000	10.0.0.11	10.0.0.2	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64 (reply
7915	9.514322000	10.0.0.11	10.0.0.5	ICMP	100	Echo (ping) request id=0x7fcf, seq=4/1024, ttl=64
7916	9.514352000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64 (reque
7917	9.514359000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7918	9.514362000	10.0.0.2	10.0.0.5	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7919	9.514360000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7920	9.514367000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7921	9.514370000	10.0.0.2	10.0.0.10	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7922	9.514369000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7923	9.514376000	10.0.0.2	10.0.0.11	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7924	9.514379000	10.0.0.2	10.0.0.15	ICMP	100	Echo (ping) reply id=0x7fcf, seq=4/1024, ttl=64
7925	9.904424000	ce:8e:54:32:97:21	NiciraNe_00:00:01	OFP+LLDP	156	Packet Out (CSM) (88B) => Chassis Id = 00:00:00:00:03:00

(b) Sia con OVX che con PySolv

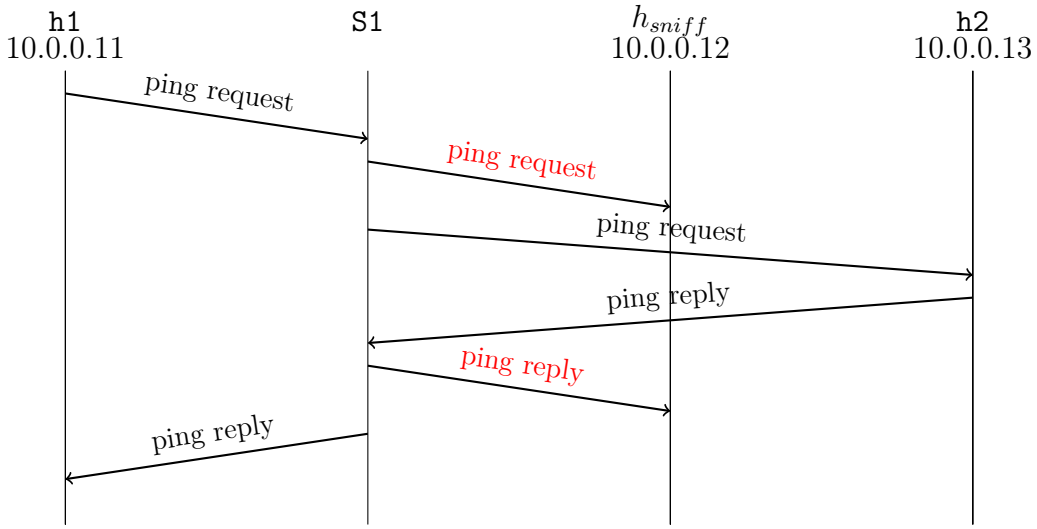
Figura 4.15: Catture Caso IV: $in\_port > fanout,\ out\_port > fanout$.

I passaggi per avviare il sistema e per catturare i vari pacchetti sono descritti di seguito. Inizialmente si fa partire la topologia con mininet e si esegue OpenVirteX creando le reti virtuali in questione. Si lancia PySolv, il quale si connette a OVX rimanendo in attesa di un qualche FlowMod, ed infine si iniziano le catture provando ad eseguire dei ping tra gli host delle reti virtuali. Una volta che si sono catturati i pacchetti interessanti si arresta e si applica un filtro per visualizzare solo i pacchetti OpenFlow e ICMP. Il filtro è:

```
of || icmp
```

4.5.1 Spiegazione caso I

Nel caso della Figura 4.13 viene eseguito un ping tra due host connessi allo stesso switch: `host_S1_1` e `host_S1_3`. Più in particolare in Figura 4.13a è rappresentata una cattura sull'interfaccia `any` del ping fatto senza l'ausilio di PySolv, mentre in Figura 4.13b è illustrata la stessa cattura con la presenza di PySolv nel sistema. Come si può notare, nella prima immagine sono presenti meno pacchetti ICMP; i mancanti (presenti nella seconda immagine), sono quelli relativi all'inoltro verso lo sniffer. Perciò si è voluto verificare il funzionamento di PySolv nel Caso I. Nel file di configurazione è stato indicato come host da intercettare, `host_S1_1` che ha come indirizzo IP 10.0.0.11 e indirizzo MAC 00:00:00:00:01:01. In questo caso, tutto il traffico da e verso questo host dovrebbe venire deviato verso lo sniffer connesso al medesimo switch e rappresentato da `host_S1_2` (IP: 10.0.0.12 e sempre specificato nel file di configurazione). A fronte dell'arrivo dei FlowMod generati da OVX, PySolv crea un messaggio di risposta accorgendosi della presenza dell'host da intercettare. In questo messaggio sono contenute tutte le informazioni dello sniffer, le quali verranno usate per replicare il traffico verso la sua porta, connessa allo switch. I vari passaggi eseguiti sono illustrati nel diagramma seguente e descritti di seguito.



Passaggi Caso I

Si illustrano ora gli scambi dei pacchetti dopo che il FlowMod generato da PySolv è stato correttamente installato. Per semplicità si indica con `h1` l'host di partenza del ping (quello che invia la *request*), e con `h2` l'host di destinazione (quello che invia la *reply*). h_{sniff} sarà invece l'host che intercetta il traffico. I numeri in fondo ad ogni punto della lista sono i relativi pacchetti illustrati nella seconda cattura.

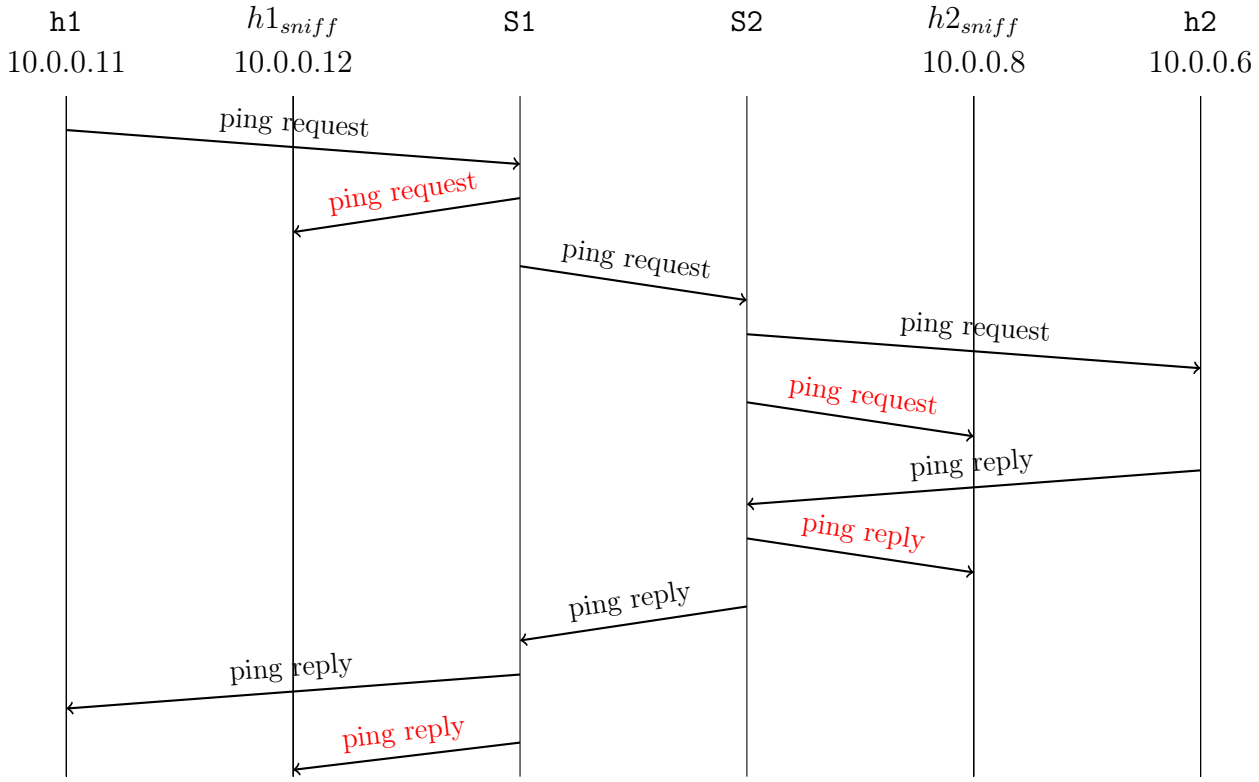
1. `h1` invia il primo *request* verso lo switch `S1` (2030);
2. `S1` riceve questa request e la inoltra ad `h2` (2031);
3. `S1`, oltre ad inoltrarla verso la vera destinazione la inoltra anche a h_{sniff} (2032);
4. `h2`, che ha ricevuto la request, risponde con una reply (2033);
5. `S1` riceve questa reply e la inoltra ad `h1` (2034);
6. `S1`, oltre ad inoltrarla verso la vera destinazione, la inoltra anche a h_{sniff} (2035).

In questo modo tutte le volte che lo switch rileva la presenza dell'indirizzo da tenere sotto controllo, tra gli indirizzi sorgente o di destinazione del pacchetto, inoltra lo stesso pacchetto a h_{sniff}.

4.5.2 Spiegazione caso II e III

Questo caso è più articolato. Viene eseguito un ping tra `host_S1_1` e `host_S2_1` con conseguente inoltro dei pacchetti verso i due sniffer `host_S1_2` e `host_S2_3`. Anche in questo caso l'host che si vuole intercettare è sempre `host_S1_1` (IP: 10.0.0.11 e MAC 00:00:00:00:01:01). In Figura 4.14 sono raffigurate le catture riguardanti i casi II e III. Questi casi vengono inseriti insieme perché si presentano in contemporanea; se un pacchetto è uscente da uno switch verso un altro switch, nel primo switch si avrà $in\_port \leq fanout$, $out\_port > fanout$, mentre nel secondo si avrà $in\_port > fanout$, $out\_port \leq fanout$.

Nel diagramma seguente si riportano tutti gli scambi di pacchetti che avvengono in presenza dei FlowMod già installati con l'ausilio di PySolv.



Passaggi caso II e III

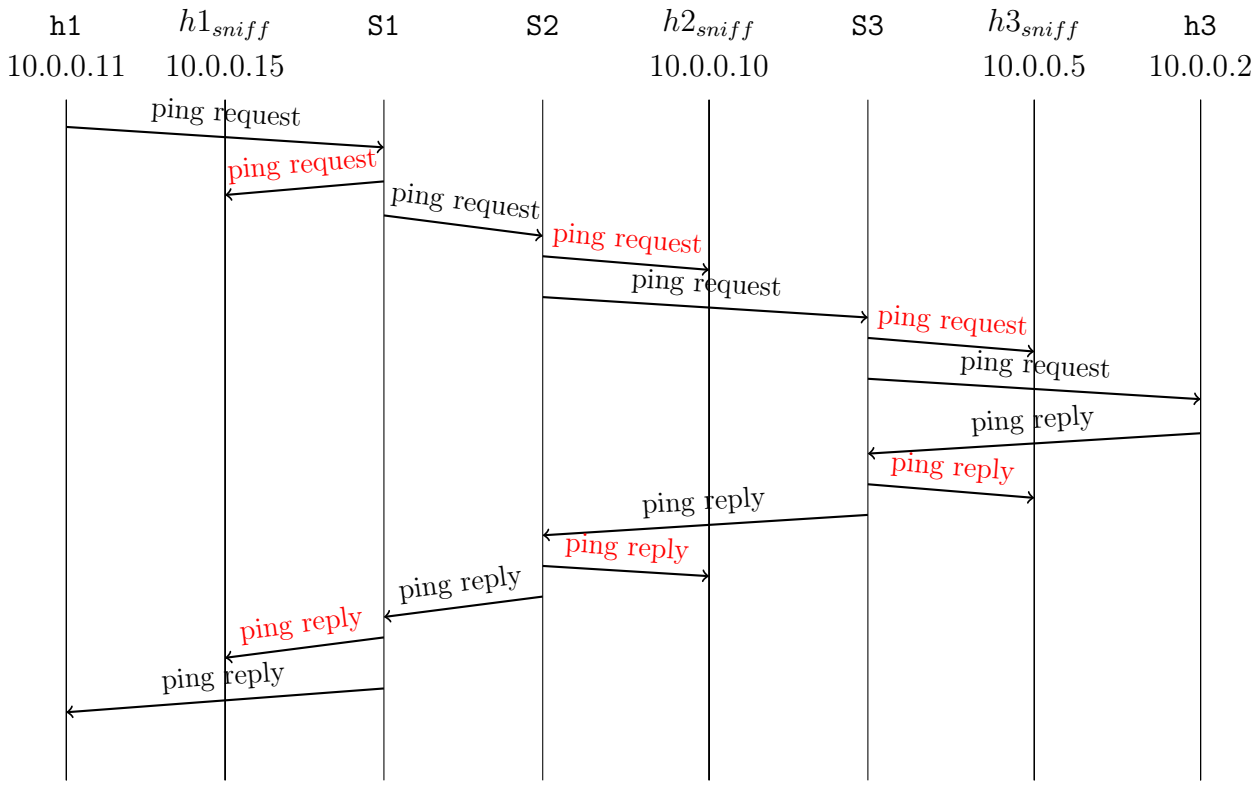
Per semplicità si indica con `h1` l'host di partenza del ping (quello che invia la *request*), e con `h2` l'host di destinazione (quello che invia la *reply*). $h1_{sniff}$ sarà invece l'host che intercetta il traffico connesso allo switch `S1` e $h2_{sniff}$ quello per lo switch `S2`. I numeri in fondo ad ogni punto della lista sono i relativi pacchetti illustrati nella seconda cattura.

1. `h1` invia il primo *request* verso lo switch `S1` (1441);
2. `S1` riceve questa *request* e la inoltra ad `S2` (1442);
3. `S1`, oltre ad inoltrare il pacchetto verso lo switch, lo inoltra anche a $h1_{sniff}$ (1443);
4. `S2` riceve il pacchetto e lo inoltra ad `h2` (1444, 1445);
5. `S2`, oltre ad inoltrarlo verso la corretta destinazione, lo inoltra anche a $h2_{sniff}$ (1446);
6. `h2`, che ha ricevuto la *request*, risponde con una *reply* (1447);
7. `S2` riceve questa *reply* e la inoltra verso lo switch `S1` (1448);
8. `S2`, oltre ad inoltrarla verso lo switch, la inoltra anche a $h2_{sniff}$ (1449);
9. `S1` riceve questa *reply* e la inoltra ad `h1` (1450, 1451);
10. `S1`, oltre ad inoltrarla verso l'host corretto, inoltra anche a $h1_{sniff}$ (1452).

4.5.3 Spiegazione caso IV

Per questa spiegazione si utilizza la topologia di rete simile a quella presentata in Figura 4.10 con qualche rivisitazione. In particolare gli host che dialogano sono `host_S1_1` e `host_S3_2`, mentre gli sniffer sono connessi alla porta 5 di ogni switch. Anche in questo caso l'host che si vuole intercettare è sempre `host_S1_1` (IP: 10.0.0.11 e MAC 00:00:00:00:01:01). In Figura 4.15 sono presenti le catture di questo particolare caso. Come già discusso, si è deciso di inoltrare i pacchetti da controllare anche agli sniffer connessi a switch che sono solo "di passaggio", come in questo caso `S2`.

Nel diagramma seguente si riportano tutti gli scambi di pacchetti che avvengono in presenza dei FlowMod già installati con l'ausilio di PySolv.



Passaggi caso IV

Anche per questo caso si indica con `h1` l'host di partenza del ping (quello che invia la *request*) e con `h3` l'host di destinazione (quello che invia la *reply*). $h1_{sniff}$ sarà invece l'host che intercetta il traffico connesso allo switch `S1`, $h2_{sniff}$ quello per lo switch `S2` ed infine $h3_{sniff}$ quello per lo switch `S3`. I numeri in fondo ad ogni punto della lista sono i relativi pacchetti illustrati nella seconda cattura.

1. `h1` invia il primo *request* verso lo switch `S1` (7907);
2. `S1` riceve questa *request* e la inoltra ad `S2` (7908);
3. `S1`, oltre ad inoltrare il pacchetto verso lo switch, lo inoltra anche a $h1_{sniff}$ (7909);
4. `S2` riceve il pacchetto e lo inoltra ad `S3` (7910, 7911);

5. `S2`, oltre ad inoltrarlo verso lo switch, lo inoltra anche a $h2_{sniff}$ (7912);
6. `S3` riceve il pacchetto e lo inoltra ad `h3` (7913, 7914);
7. `S3`, oltre ad inoltrarlo verso la corretta destinazione, lo inoltra anche a $h3_{sniff}$ (7915);
8. `h3`, che ha ricevuto la *request*, risponde con una *reply* (7916);
9. `S3` riceve questa *reply* e la inoltra verso lo switch `S2` (7917);
10. `S3`, oltre ad inoltrarla verso lo switch, la inoltra anche a $h3_{sniff}$ (7918);
11. `S2` riceve questa *reply* e la inoltra verso lo switch `S1` (7919, 7920);
12. `S2`, oltre ad inoltrarla verso questo switch, inoltra anche a $h2_{sniff}$ (7921);
13. `S1` riceve la *reply* e la inoltra verso l'host `h1` (7922, 7923);
14. `S1`, oltre ad inoltrarla verso la corretta destinazione, inoltra anche a $h1_{sniff}$ (7924).

4.6 Prerequisiti e limitazioni di progetto

Si vuole dedicare questa sezione a tutti i futuri sviluppatori, utilizzatori o semplicemente ai lettori. Verranno esposti i requisiti per poter utilizzare OVX affiancato da PySolv. Nel caso in cui si utilizzasse la macchina virtuale presente nella homepage di OpenVirteX (presente in [5]), molti dei seguenti requisiti sono già presenti, soprattutto quelli riguardanti la parte infrastrutturale di Mininet ed OpenFlow.

I requisiti basilari che si devono avere sono i seguenti.

- *Sistema operativo Unix-like*: per poter installare tutto l'occorrente si deve poter avere un sistema operativo di questo tipo con il compilatore python e java installati.
- *Emulatore Mininet*: usato per creare reti SDN con gli OpenVSwitch ed i vari controller.

- *Protocollo OpenFlow versione 1.0*: per abilitare la comunicazione tra gli switch ed il controller. Si raccomanda questa versione essendo quella supportata da OpenVirteX. Nel caso in cui si utilizzasse un versione più recente non si garantisce il corretto funzionamento della tecnologia.

- *sorgenti di OpenVirteX*[13]: per poter utilizzare OVX nel sistema.

Tutti e quattro questi punti sono già presenti nella macchina virtuale fornita dalla homepage di OpenVirteX. Si raccomanda di non aggiornare la distribuzione fornita e nemmeno i vari pacchetti attraverso i vari comandi `sudo apt-get update` e `sudo apt-get upgrade`. Questo perché, come è stato testato, si va ad aggiornare alla versione più recente il protocollo OpenFlow, risultando incompatibile con l'implementazione attuale di OVX.

Altri requisiti che si devono avere per poter utilizzare il progetto sono i *dissector OpenFlow* per Wireshark e l'aggiunta di alcune dipendenze Java a quelle di OpenVirteX. Per quanto riguarda le funzioni dissector per OpenFlow, esse vengono usate per analizzare e riconoscere i pacchetti appartenenti a questo protocollo, in modo da rappresentarli come tutti gli altri pacchetti riconosciuti, ovvero mostrando i vari campi e valori contenuti.
L'aggiunta delle dipendenze in Java è stata fatta per gestire degli oggetti JSON. Questi, come già descritto, vengono utilizzati nella comunicazione tra OVX e PySolv. Queste dipendenze sono state aggiunte al file `pom.xml`, contenuto nella cartella iniziale di OpenVirteX. Di seguito vengono illustrate le righe di codice da aggiungere:

```
...
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20090211</version>
</dependency>

<dependency>
  <groupId>com.googlecode.json-simple</groupId>
  <artifactId>json-simple</artifactId>
  <version>1.1.1</version>
</dependency>
...
```

[13]Scaricabili dal sito `https://github.com/opennetworkinglab/OpenVirteX`.

Tutti i requisiti elencati finora riguardano le caratteristiche che deve avere la macchina per poter eseguire, senza problemi, l'intero progetto. Si passa ora ad un altro tipo di requisito ovvero la conoscenza che deve avere colui che vuole costruire il sistema. Ricordando sempre il fine di controllo per cui si vorrebbe utilizzare questo progetto, tale individuo potrebbe essere il tecnico che, incaricato da un giudice, mette in atto le azioni per intercettare un certo host della rete. Come già discusso, al giorno d'oggi questo tecnico dovrebbe fisicamente mettere mano ai dispositivi di rete per manovrare i vari flussi di pacchetti per deviarli verso uno sniffer. Con l'introduzione di OVX e PySolv, quello che invece viene richiesto a questo tecnico è una conoscenza profonda di tutti quelli gli indirizzi, sia degli sniffer che degli host che vuole intercettare. In particolare sono richiesti sia gli indirizzi IP che MAC di questi ultimi, così come le porte fisiche alle quali sono connessi al relativo switch. Si ricorda che una qualsiasi variazione della topologia di rete (ad esempio l'aggiunta di host), si può ripercuotere in numerose modifiche di indirizzi che devono essere ben gestite.

Passando alle limitazioni di questo progetto, la più importante che si vuole mettere in evidenza è quella riguardante la presenza dei cosiddetti *BigSwitch*. Questa particolare tipologia di switch, come già indicato, è uno switch virtuale mappato in più switch fisici. Quindi dal punto di vista del controller, viene categorizzato come un unico switch connesso ad altri switch virtuali, delegando ad OVX la gestione interna circa il forwarding dei vari pacchetti tra gli switch fisici interni a questo BigSwitch. La soluzione presentata da PySolv è in grado di gestire solo la presenza di un unico BigSwitch contenente tutti gli switch fisici della topologia.
Un'altra limitazione riguarda il numero di host che è possibile intercettare con PySolv. Può capitare che si debba tenere sotto controllo non solo un singolo individuo della rete ma anche diversi. Per il momento questa soluzione offre solo la possibilità di avere un singolo obiettivo da controllare.

4.7 Approfondimenti

In questa parte si andranno ad approfondire tutti quegli argomenti che si reputano importanti ed interessanti, con l'intento di eliminare i dubbi o semplicemente per chiarire alcune problematiche che sono sorte durante lo sviluppo del progetto.

Uso dei delimitatori

Come illustrato nella Sezione 4.2, la struttura di un messaggio inviato da OVX a PySolv è caratterizzata da un delimitatore iniziale ed uno finale (preferibilmente diversi e formati da più caratteri uguali). Inizialmente questi delimitatori non erano presenti ma a volte, capitava che un messaggio giungesse a PySolv in maniera non corretta, ovvero con dei caratteri strani e non tutti perfettamente al loro posto. Quindi si è cercato di risolvere questo problema scongiurando l'uscita del programma e quindi la conseguente terminazione di PySolv. La soluzione che si è pensata è semplice ed utilizza proprio questi delimitatori. Nel caso in cui giunga un messaggio non consono alla struttura che PySolv si aspetta, non si fa altro che attivare una funzionalità di supporto che permetta di leggere tutti i byte ancora presenti nel buffer in ricezione e appartenenti al messaggio in questione, fino a quando non si incontra il delimitatore finale. Così facendo si va a scartare tutto il contenuto del messaggio in modo che non venga successivamente letto da PySolv, generando un errore.

Il delimitatore iniziale viene utilizzato per un motivo simile. Oltre ad essere un modo per verificare che il messaggio in arrivo è proprio quello che ci si aspetta, può capitare a volte che giunga un messaggio contenente al suo interno un altro messaggio. Questo accade perché in OVX circolano dei pacchetti contenenti al loro interno degli altri pacchetti e che, conseguentemente, vengano tradotti in un messaggio per PySolv, contenente un altro messaggio con la stessa struttura di quello che lo contiene. Anche questa particolare tipologia di messaggi genera un errore alla funzione che traduce i dati in formato JSON e, come tale, deve essere gestito. Questo avviene semplicemente come si è spiegato poco sopra. Se PySolv verifica che la struttura del messaggio non è consona a quella standard, si scartano i byte in arrivo e, nel caso in cui ci si accorge che il messaggio è del tipo appena descritto (ovvero con un messaggio contenuto in un altro), si deve scartare tutto il messaggio interno e tutto quello esterno, tramite l'utilizzo di entrambi i delimitatori.

Questi casi non sono frequenti e quindi si è scelta l'opzione più immediata, ovvero quella di eliminare direttamente il messaggio in arrivo. Questa azione è stata decisa anche a fronte della non utilità di questi tipi di messaggi, in quanto praticamente illeggibili da una macchina.

File di configurazione mancante

Nel caso in cui il file di configurazione fosse mancante o non sia leggibile perché scritto in maniera errata, si sono prese delle precauzioni a riguardo. La prima di queste è che, siccome in questo file sono presenti tutte le informazioni circa gli sniffer e circa i modi per poter deviare i vari flussi della rete, si è deciso di non inviare nulla da PySolv ad OVX anche a fronte dell'arrivo di un FlowMod. In questi casi, di solito, si risolve preimpostando dei dati standard da utilizzare nel caso in cui si incappasse in un qualche tipo di errore nella lettura del file. Ma in questo caso si è deciso saggiamente di non inserire questi dati standard. Tale decisione è stata presa siccome le informazioni interne a questo file possono cambiare alla minima modifica della topologia di rete e quindi non è possibile impostare delle informazioni standard circa gli indirizzi degli sniffer.

Invece, per quanto riguarda le informazioni che non implicano gli sniffer e la rete fisica in generale (quindi per esempio i delimitatori), si sono impostati dei valori predefiniti in modo da essere gli stessi sia per OVX che per PySolv. Così facendo, essi possono connettersi e comunicare ugualmente anche se con grosse limitazioni dovute al fatto che PySolv non potrà comunicare con OVX ma solo viceversa.

Gestione startup progetto e gestione di errori

Capita a volte che un programma per funzionare abbia bisogno di altri programmi in esecuzione e, nel caso non li trovasse, ritorni un messaggio di errore. Quello che si è pensato per questo progetto è fare in modo di scongiurare la nascita di eventuali errori, i quali rendano la *user experience* sgradevole. Per quanto riguarda l'inizializzazione del progetto, essa può avvenire nelle seguenti due modalità.

- La tecnologia OpenVirteX è già in esecuzione e PySolv viene eseguito in un secondo momento. In questo caso, grazie all'utilizzo del thread *InitConnectionThread*, si riesce ad utilizzare OVX anche senza il bisogno di avere PySolv attivo. A questo thread si delega la gestione della connessione con PySolv, senza in alcun modo andare a modificare il normale comportamento della tecnologia OVX. Quindi tutti i controller possono tranquillamente utilizzare la tecnologia come meglio credono, senza preoccuparsi di eventuali errori dovuti al sopraggiungere di PySolv.

- La tecnologia OpenVirteX non è ancora in esecuzione ma PySolv è già in esecuzione. Questo viene gestito internamente a PySolv con l'utilizzo di una semplice socket che, finché OVX non si connette, non viene invocata e perciò rimane in attesa senza eseguire nient'altro. Successivamente, dopo che OVX è stato eseguito, esso andrà a connettersi alla porta dov'è in ascolto questa socket, scatenando il successivo comportamento di PySolv già decritto.

Quindi, come si è appena discusso, non ci sono problemi di esecuzione tra OVX e PySolv.

Un'altra eventualità in cui si può incorrere, e che quindi deve essere gestita, è quella riguardante l'improvvisa terminazione di PySolv. Si è deciso di gestire questo imprevisto sempre in modo da non alterare il funzionamento standard di OVX. Semplicemente, quando ci si accorge di questa evenienza, quello che viene fatto è una sorta di *reset*, ritornando al comportamento iniziale dell'interazione di OVX con PySolv. Questo permette di ripartire dall'inizio con la nuova istanza di PySolv che sopraggiungerà, senza dover riavviare OVX. È possibile sfruttare questa funzionalità anche per azioni di debugging di PySolv, facendo rimanere sempre attivo OVX.

Infine si vuole aggiungere un'ultima azione che viene fatta allo startup di PySolv. Se nel file di configurazione non viene specificato neanche uno sniffer o se non è presente nessuno obiettivo da controllare, non ha senso che PySolv venga attivato. Quindi, per evitare ciò, è stato inserito un controllo che, se si presentano uno dei casi appena citati, non permette al primo thread di PySolv di connettersi a OVX. Successivamente si attendono 10 secondi per poi riprovare a leggere il file di configurazione, nella speranza che venga modificato con l'aggiunta dei campi necessari.

Capitolo 5

Conclusioni

Il lavoro svolto aveva come obiettivo quello di capire se - ed in che termini - fosse possibile costruire un'infrastruttura che creasse la possibilità di sniffare il traffico di un host all'interno di una rete, il tutto definendo delle procedure via software. Giunti a questo punto è possibile ripercorrere le fasi che si sono attuate per raggiungere il risultato prefissato. In questa tesi, oltre ad aver individuato alcuni strumenti, si sono analizzate possibili architetture per poter arrivare all'obiettivo stabilito ed, alla fine, ne è stata identificata una che soddisfa i requisiti richiesti. Tali requisiti sono:

- Consentire tale funzione di sniffing;
- Disaccoppiare totalmente la gestione delle reti dalla gestione degli sniffer;
- Mettere in atto la suddetta funzione in modo che l'utenza non sia informata di quest'ultima. Così facendo, l'utente finale non ha nessun modo per accorgersi di quello che sta realmente succedendo.

Questi tre sono i requisiti fondamentali di una qualsiasi architettura di controllo.

Per prima cosa si è scelta l'infrastruttura che permettesse di abilitare le funzionalità che si volevano ottenere: si è scelto OpenVirteX per tutte le sue potenzialità in termini di *slicing* delle varie reti virtuali. Inoltre si è deciso di utilizzarlo anche per la presenza di una discreta documentazione, la quale è stata d'aiuto soprattutto nelle fasi iniziali, quando si è dovuto apprendere come vengono gestite le varie reti virtuali e come crearle. Sono state eseguite varie prove per comprendere appieno i concetti e come vengono mappati tutti gli elementi propri di queste reti. Successivamente ci si è chiesti

come questa architettura potesse essere usata per il fine stabilito. Per fare ciò si è dovuto studiare approfonditamente il funzionamento interno, analizzando i vari pacchetti e le interazioni tra *northbound* e *southbound*. In seguito, dopo aver capito come dialogano le due parti dell'infrastruttura, sono state fatte alcune considerazioni su come e dove inserire la nuova funzionalità di sniffing senza compromettere l'andamento generale del sistema, cercando di incidere il meno possibile sulle risorse e sulle prestazioni che normalmente caratterizzano l'architettura. Sono state trovate più alternative, che si sono dimostrate tutte valide. Tra queste è stata scelta quella che rispecchia maggiormente i requisiti appena indicati. Questa soluzione, rinominata *PySolv*, è stata implementata e descritta minuziosamente; i messaggi scambiati tra PySolv e OVX, il funzionamento dei vari thread ed i FlowMod che si sono creati, sono stati descritti ed illustrati interamente. PySolv è stata sperimentata su diverse topologie di rete, in modo da poter affermare con relativa certezza, che essa svolge il compito che era stato preposto.

Al termine del tempo previsto per lo svolgimento del progetto di tesi, si sono tratte le dovute conclusioni sul lavoro svolto. L'obiettivo preposto è stato raggiunto? E se no, perché non è stato possibile? In questo caso l'obiettivo è stato raggiunto con risultati ottimi e si può ritenere lo svolgimento del progetto totalmente soddisfacente. L'abilitazione delle funzionalità di sniffing, disaccoppiando la gestione delle reti degli utilizzatori da quella degli sniffer e agendo in modo del tutto trasparente ai vari utenti, è stata implementata correttamente. Questo può essere visto come un punto di partenza per ulteriori analisi mirate ad una gestione più efficiente di tutti gli apparati di rete, anche per abilitare funzionalità diverse da quelle di controllo discusse in questo documento. Ad esempio sarebbe possibile ideare alcune soluzioni, basate su reti programmabili SDN, che consentano azioni di *fault tolerance* o di *load balancing*.

Alcune direzioni che si potrebbero intraprendere per proseguire e migliorare il lavoro presentato, sono descritte nella seguente ultima sezione.

5.1 Sviluppi futuri

Siccome questo progetto potrebbe essere sviluppato ulteriormente, si è deciso di lasciare a coloro che volessero farlo, alcuni spunti da cui partire, non appena siano state pienamente comprese l'architettura e le funzionalità di questa

tecnologia. Si tenga conto che quelli indicati di seguito non rappresentano necessariamente gli sviluppi immediatamente successivi al lavoro svolto, ma tendono ad essere un *vademecum* per quelli che si pensa siano i più importanti da attuare per migliorare in maniera profonda tutto il progetto.

Gestione degli sniffer

Il progetto presentato ha un insieme di requisiti abbastanza stringenti per quanto riguarda la connessione dei vari sniffer agli switch. Come trattato in precedenza, nella soluzione in cui viene mantenuta l'indipendenza tra le varie reti virtuali, ci si serve di un elevato numero di sniffer. Più in particolare all'interno di un certo switch si devono connettere un numero di sniffer pari al numero di reti virtuali che "circolano" all'interno di questo switch. Quindi è possibile trovare un numero di sniffer che varia da 0 al massimo numero di reti virtuali presenti. Invece, nella soluzione dove gli sniffer sono indipendenti dalla particolare rete virtuale, è stato connesso un solo sniffer per ogni switch indipendentemente dal numero di reti virtuali presenti.

Una possibile evoluzione di queste due soluzioni, potrebbe essere l'utilizzo di un numero inferiore di sniffer, da posizionare in punti strategici e noti della rete fisica. Si potrebbe utilizzare questa evoluzione sia nel caso in cui si voglia mantenere il più separato possibile i vari flussi delle reti virtuali, sia nell'altro caso; indubbiamente, in quest'ultimo, verrebbero utilizzati meno sniffer. In entrambe le situazioni, comunque, è necessario che vengano memorizzate, oltre ai loro indirizzi, anche le modalità per raggiungerli. Per spiegare più chiaramente, sarà esposto un semplice esempio che rispecchi le reti fisiche odierne. Si consideri una topologia di rete molto complessa, formata da centinaia di switch interconnessi tra loro e con decine di host connessi ad ognuno di essi. Le soluzioni presentate sarebbero inammissibili in questo caso, perché si dovrebbero utilizzare centinaia (se non migliaia) di sniffer. Come si diceva, la possibile evoluzione sarebbe quella di posizionare un numero molto ristretto di sniffer in punti noti della rete. In questo modo si riduce drasticamente il numero di dispositivi utilizzati, ma si incorre anche in un rilevante svantaggio: non appena PySolv si accorge della presenza di un host da intercettare, deve essere in grado di installare regole all'interno degli switch che permettano di recapitare i pacchetti interessati anche ad uno sniffer. Pertanto, è necessario saper gestire tutti i collegamenti interswitch.

Un rischio circa l'uso di un numero ridotto di sniffer in reti di grandi dimensioni, è quello di incorrere in un numero di pacchetti troppo ingente per le capacità del dispositivo e, conseguentemente, non riuscire ad elaborare tutte le informazioni richieste. Questa considerazione non è argomento proprio di questa tesi, ma si è ritenuto opportuno informare della presenza di tale soluzione alternativa.

Evoluzione in rete

Durante lo sviluppo del progetto, PySolv è sempre stato testato in locale sulla macchina virtuale dov'era eseguito anche OVX. È stato tralasciato lo scenario che vede la presenza di PySolv e OVX in due macchine fisiche distinte. Questo aspetto è stato gestito in maniera minima, infatti nel file di configurazione sono presenti tutti i campi atti a proiettare sia PySolv che OVX in rete attraverso la modifica degli indirizzi IP e delle porte associate. Tuttavia, questo aspetto non è mai stato testato sul campo, pertanto è stato deciso di inserire la questione all'interno di questa sezione. In particolare, come possibile evoluzione, si potrebbe pensare di utilizzare PySolv su una macchina fisica, OVX su un'altra assieme alla topologia di rete fisica reale (non emulata con mininet), contenente switch ed host fisici. In questo modo è possibile innanzitutto testare se la soluzione proposta sia funzionante anche su dispositivi fisici e capire se il funzionamento possa persistere anche in reti più vaste. Un altro aspetto dell'evolvere l'intero progetto in rete è rivolto all'utilizzo di quelle accortezze che, lavorando in locale, si tenderebbe a non utilizzare. Per esempio, si adoperano funzionalità di sicurezza per rendere i messaggi scambiati illeggibili da terze parti. In particolare la questione diviene importante per quanto riguarda lo scambio di messaggi tra PySolv e OVX perché, se un malintenzionato ne venisse in possesso, sarebbe in grado di intromettersi e di gestire tutti i flussi di traffico di suo interesse.

Altri sviluppi

Altri possibili sviluppi relativi all'implementazione di PySolv e ai thread creati in OVX sono discussi di seguito. Più nello specifico, riguardano alcuni casi particolari in cui è possibile imbattersi e che devono essere gestiti al meglio.

L'implementazione di PySolv rilasciata con questa tesi è in grado di gestire tutti i FlowMod che arrivano da OVX e che hanno un solo valore di *output*

port nelle action. Questo significa che i vari FlowMod che fanno match con un pacchetto lo inoltrano verso una ed una sola porta d'uscita. Successivamente, nel caso in cui l'host sia da intercettare, sarà compito di PySolv aggiungere la seconda porta d'uscita verso il relativo sniffer. Quindi, con la versione attuale, non è possibile gestire il caso in cui all'interno di un FlowMod che da OVX giunge a PySolv, siano presenti già due o più *output port*.

Sempre nell'implementazione attuale è possibile intercettare solamente un singolo host, quindi una possibile evoluzione da attuare può essere mirata a tenere sotto controllo più di un host alla volta. Soprattutto per quanto riguarda le grosse reti, si ritiene che questa evoluzione diventi inevitabile; con il crescere del numero di host ci si imbatterà nel bisogno di intercettare più di un individuo della rete.

Appendice A

Codice topologia del case study I

Nome file: `topology.py`

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import lg, setLogLevel
from mininet.cli import CLI
from mininet.node import RemoteController

CORES = {
  'S1': {'dpid': '000000000000010%s'},
  'S2': {'dpid': '000000000000020%s'},
  'S3': {'dpid': '000000000000030%s'},
  'S4': {'dpid': '000000000000040%s'},
  }

FANOUT = 3 # numero di host connessi ad ogni switch

class I2Topo(Topo):

  def __init__(self, enable_all = True):
    "Create Giulio topology."

    # Add default members to class.
    super(I2Topo, self).__init__()

```

```
26      # Add core switches
27      self.cores = {}
28      for switch in CORES:
29        self.cores[switch] = self.addSwitch(switch,
            ↪ dpid=(CORES[switch]['dpid'] % '0'))
30      temp = 0
31      # Add hosts and connect them to their core switch
32      for switch in CORES:
33        for count in xrange(1, FANOUT + 1):
34          temp = temp + 1
35          # Add hosts
36          host = 'host_%s_%s' % (switch, count)
37          ip = '10.0.0.%s' % temp
38          mac = CORES[switch]['dpid'][4:] % count
39          h = self.addHost(host, ip=ip, mac=mac)
40          # Connect hosts to core switches
41          self.addLink(h, self.cores[switch])
42
43      # Connect core switches
44      self.addLink(self.cores['S1'], self.cores['S4'])
45      self.addLink(self.cores['S2'], self.cores['S4'])
46      self.addLink(self.cores['S3'], self.cores['S4'])
47
48
49  if __name__ == '__main__':
50      topo = I2Topo()
51      ip = '127.0.0.1'
52      port = 6633
53      controllerGiulio2 = RemoteController('GiulioCtrl2',
          ↪ ip=ip, port=port)
54      net = Mininet(topo=topo, autoSetMacs=True,
          ↪ xterms=False, controller=None)
55      net.addController(controllerGiulio2)
56      net.start()
57      print "Hosts configured"
58      CLI(net)
59      net.stop()
```

Appendice B

Esempio d'uso II

In questo secondo case study sono presenti un numero maggiore di switch e di host in modo da poter capire meglio come più reti virtuali possono convivere. In dettaglio le reti virtuali sono:

- Una prima rete virtuale che rispecchia la topologia di rete fisica sottostante (come nel case study I);
- Una seconda rete virtuale composta da due soli switch i quali racchiudono al loro interno rispettivamente 3 switch fisici nel primo virtuale e 2 fisici nel secondo virtuale.

In questo caso si è scelto 4 come *fanout* (numero di host connessi a ciascuno switch). In Figura B.1 è indicata la topologia fisica creata da mininet. Si sono specificati solo i vari switch con i relativi link per comodità.

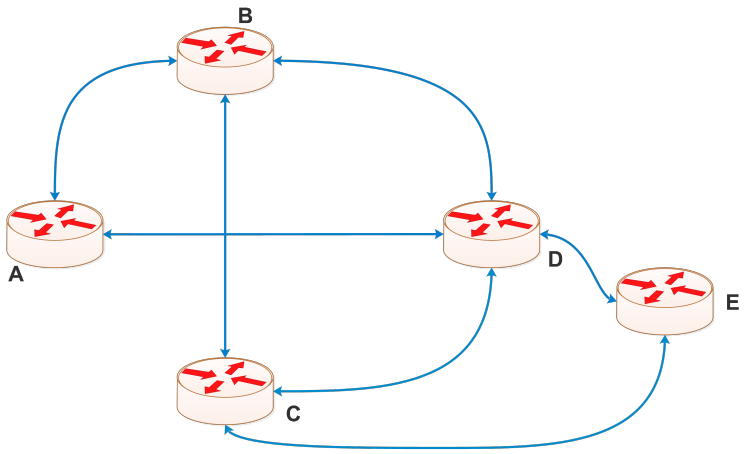


Figura B.1: Topologia di rete del secondo case study.

B.1 Virtual Network I

Come prima virtual network si è deciso di agire come nel case study precedente ovvero mappando su ogni switch fisico, uno virtuale ed associando ad ogni link fisico, uno virtuale. In fondo a questa appendice sono presenti i comandi utilizzati, tutti più o meno sulla falsa riga di quelli descritti nell'esempio precedente. Ovviamente, siccome il numero di nodi e di host è maggiore, saranno presenti più comandi; in questo caso è molto importante non sbagliarsi con i vari identificatori di porte perché è facile dimenticarsi i valori.

Il numero di host connessi ad ogni switch varia ed è specificato nella Tabella B.1 mentre i loro nomi sono simili al caso di studio precedente. Per esempio l'host numero 3 connesso allo switch B avrà il seguente nome: `host_B_3`. Quindi è possibile risalire facilmente al nome di tutti gli host desiderati.

B.1.1 Comandi per Virtual Network I

```
1 python ovxctl.py -n createNetwork tcp:localhost:10000 10.0.0.0 16
2
3 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:01:00
4 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:02:00
5 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:03:00
6 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:04:00
7 python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:05:00
8
9 python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 5
```

Tabella B.1: Numero di host connessi ad ogni switch.

Nome switch	**Numero host connessi**
A	2
B	3
C	1
D	3
E	2

```
10 python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 6
11 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 5
12 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 6
13 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 7
14 python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 5
15 python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 6
16 python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 7
17 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 5
18 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 6
19 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 7
20 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 8
21 python ovxctl.py -n createPort 1 00:00:00:00:00:00:05:00 5
22 python ovxctl.py -n createPort 1 00:00:00:00:00:00:05:00 6
23
24 python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 1
25 python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 2
26 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 1
27 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 2
28 python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 3
29 python ovxctl.py -n createPort 1 00:00:00:00:00:00:03:00 1
30 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 1
31 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 2
32 python ovxctl.py -n createPort 1 00:00:00:00:00:00:04:00 3
33 python ovxctl.py -n createPort 1 00:00:00:00:00:00:05:00 1
34 python ovxctl.py -n createPort 1 00:00:00:00:00:00:05:00 2
35
36 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:01 1
       ↪ 00:a4:23:05:00:00:00:02 1 spf 1
37 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:01 2
       ↪ 00:a4:23:05:00:00:00:04 3 spf 1
38 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:02 2
       ↪ 00:a4:23:05:00:00:00:03 2 spf 1
39 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:02 3
       ↪ 00:a4:23:05:00:00:00:04 4 spf 1
40 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:03 1
       ↪ 00:a4:23:05:00:00:00:04 2 spf 1
41 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:04 1
       ↪ 00:a4:23:05:00:00:00:05 1 spf 1
42 python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:03 3
       ↪ 00:a4:23:05:00:00:00:05 2 spf 1
43
```

```
44 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 3 00:00:00:00:01:01
45 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 4 00:00:00:00:01:02
46 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 4 00:00:00:00:02:01
47 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 5 00:00:00:00:02:02
48 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 6 00:00:00:00:02:03
49 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:03 4 00:00:00:00:03:01
50 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:04 5 00:00:00:00:04:01
51 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:04 6 00:00:00:00:04:02
52 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:04 7 00:00:00:00:04:03
53 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:05 3 00:00:00:00:05:01
54 python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:05 4 00:00:00:00:05:02
55
56 python ovxctl.py -n startNetwork 1
```

B.2 Virtual Network II

Si crea ora la seconda rete virtuale sempre sulla topologia fisica indicata all'inizio del caso di studio. In particolare quello che si fa è andare a mappare tutti gli switch fisici all'interno di due grandi switch virtuali. Gli switch A,B e D all'interno dello switch virtuale denominato X, mentre gli switch C ed E all'interno dello switch virtuale Y. In Figura B.2 è presente una rappresentazione della rete.

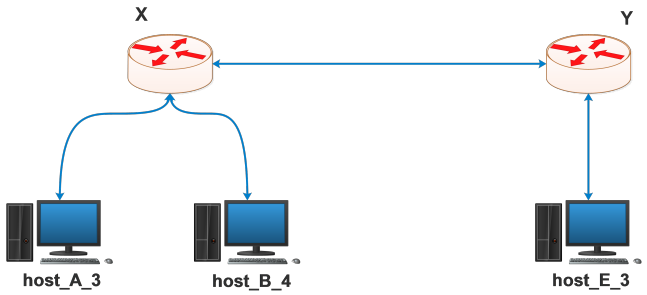


Figura B.2: Topologia di rete virtuale mappata sopra la rete fisica del secondo case study.

Si sono connessi due host al primo switch virtuale ed un host al secondo in modo da verificare, una volta fatta partire questa seconda rete virtuale, il fatto che riescano a comunicare tra loro.

I comandi che si sono utilizzati per creare questa seconda virtual network non differiscono molto da quelli indicati nel primo case study.

Istanziare la virtual network Per creare la virtual network si utilizza lo stesso comando iniziale ma cambiando la porta sulla quale può comunicare con il controller.

```
1 python ovxctl.py -n createNetwork tcp:localhost:20000
    ↪ 10.0.0.0 16
```

In questo caso il *tenantID* avrà valore 2, essendoci già una rete virtuale attiva.

Creare gli switch virtuali In questo caso si devono mappare 3 (o 2) switch all'interno di uno virtuale. Per fare ciò si indicano i 3 (2) DPID fisici degli switch interessati.

```
1 python ovxctl.py -n createSwitch 2 00:00:00:00:00:00:01:00,
    ↪ 00:00:00:00:00:00:02:00, 00:00:00:00:00:00:04:00
python ovxctl.py -n createSwitch 2
    ↪ 00:00:00:00:00:00:03:00,00:00:00:00:00:00:05:00
```

Anche in questo caso viene assegnato un DPID virtuale ad entrambi gli switch.

Creare le porte virtuali Per quanto riguarda le porte, quelle che si devono indicare devono appartenere agli switch fisici. Quindi quelli che si specificano sono i DPID fisici degli switch fisici ed il rispettivo numero di porta che si vuole utilizzare.

```
1 python ovxctl.py -n createPort 2 00:00:00:00:00:00:01:00 3
2 python ovxctl.py -n createPort 2 00:00:00:00:00:00:02:00 4
3 python ovxctl.py -n createPort 2 00:00:00:00:00:00:04:00 5
4
5 python ovxctl.py -n createPort 2 00:00:00:00:00:00:03:00 5
6 python ovxctl.py -n createPort 2 00:00:00:00:00:00:05:00 3
```

I primi tre comandi si riferiscono alle porte interne del primo switch virtuale (X) mentre gli ultimi due si riferiscono al secondo switch (Y). In questo caso gli identificativi delle porte vengono attribuiti in base ai vari switch virtuali ai quali appartengono. Quindi in questo caso le prime tre porte avranno ID pari a 1, 2 e 3 essendo tutte e tre appartenenti al primo switch virtuale, così come le ultime due porte avranno anch'esse ID 1 e 2 ma in riferimento al secondo switch.

Creare link virtuali Questi link virtuali si comportano in modo identico al primo caso di studio. Si devono indicare i DPID virtuali degli switch interessati e l'ID delle porte che si vogliono connettere.

```
1 python ovxctl.py -n connectLink 2 00:a4:23:05:00:00:00:01 3
    ↪ 00:a4:23:05:00:00:00:02 1 spf 1
```

Connessione tra host e switch Anche la connessione con gli host avviene come nel primo caso. Si specificano il DPID virtuale dello switch che si vuole connettere all'host e l'indirizzo MAC dell'host.

```
1 python ovxctl.py -n connectHost 2 00:a4:23:05:00:00:00:01 1 00:00:00:00:01:03
2 python ovxctl.py -n connectHost 2 00:a4:23:05:00:00:00:01 2 00:00:00:00:02:04
3 python ovxctl.py -n connectHost 2 00:a4:23:05:00:00:00:02 2 00:00:00:00:05:03
```

Eseguire la virtual network Ora che si sono inseriti tutti i dispositivi e che si sono formate tutte le connessioni utili, è possibile eseguire la rete virtuale semplicemente indicando il suo *tenantID*.

```
1 python ovxctl.py -n startNetwork 2
```

Per garantire l'isolamento del traffico, OVX traduce l'indirizzo IP di ogni pacchetto di una rete virtuale sotto forma di un altro indirizzo IP. Nel caso della seconda rete virtuale, gli indirizzi IP sono mappati in 2.0.0.0/8 che assicurano una chiara separazione da quelli della prima rete che sono mappati negli indirizzi IP 1.0.0.0.

Appendice C

Modifica impostazioni di rete

C.1 Alias usati per i controller e per OVX

Modifica effettuate al file `/etc/network/interfaces` per aggiungere delle interfacce di rete virtuali:

Listing C.1: Aggiunta delle 4 interfacce virtuali.

```
auto lo:ovx
iface lo:ovx inet static
name lo-ovx
address 10.10.10.10
netmask 255.255.255.0
broadcast 10.10.10.255
network 10.10.10.0

auto lo:fl1
iface lo:fl1 inet static
name lo-fl1
address 10.10.10.1
netmask 255.255.255.0
broadcast 10.10.10.255
network 10.10.10.0

auto lo:fl2
iface lo:fl2 inet static
name lo-fl2
```

```
address 10.10.10.2
netmask 255.255.255.0
broadcast 10.10.10.255
network 10.10.10.0

auto lo:fl3
iface lo:fl3 inet static
name lo-fl3
address 10.10.10.3
netmask 255.255.255.0
broadcast 10.10.10.255
network 10.10.10.0
```

C.2 Alias utilizzati per gli switch

Di seguito sono presenti le interfacce di rete, una per ogni switch presente nella rete. Queste modifiche sono state sempre apportate al file `/etc/network/interfaces`.

```
auto lo:s1
iface lo:s1 inet static
name lo:s1
address 10.20.20.1
netmask 255.255.255.0
broadcast 10.20.20.255
network 10.20.20.0

auto lo:s2
iface lo:s2 inet static
name lo:s2
address 10.20.20.2
netmask 255.255.255.0
broadcast 10.20.20.255
network 10.20.20.0

auto lo:s3
iface lo:s3 inet static
name lo:s3
address 10.20.20.3
netmask 255.255.255.0
```

```
broadcast 10.20.20.255
network 10.20.20.0

auto lo:s4
iface lo:s4 inet static
name lo:s4
address 10.20.20.4
netmask 255.255.255.0
broadcast 10.20.20.255
network 10.20.20.0
```

Bibliografia

[1] Nick McKeown, *How SDN will Shape Networking*, Open Networking Summit 2011.

[2] Thomas D. Madeau and Ken Gray, *SDN: Software Defined Networks*, O'Reilly, 1ª edizione (2013).

[3] Patricia A. Morreale and James M. Anderson. 2014. *Software Defined Networking: Design and Deployment.* CRC Press, Inc., Boca Raton, FL, USA.

[4] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. *OpenFlow: enabling innovation in campus networks.* SIGCOMM Comput. Commun. Rev. 38, 2 (March 2008), 69-74.

[5] Homepage del progetto OpenVirteX: `http://ovx.onlab.us/`.

[6] Ali Al-Shabibi, Marc De Leenheer, Ayaka Koshibe, Guru Parulkar, Bill Snow, Matteo Gerola, and Elio Salvadori, *OpenVirteX: Make Your Virtual SDNs Programmable*, in ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), Chicago, IL, USA, August 2014.

[7] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, William Snow, Guru Parulkar, *OpenVirteX: A Network Hypervisor*, in Open Networking Summit, Santa Clara, CA, USA, March 2014.

[8] OpenFlow Switch Consortium and Others. *OpenFlow Switch Specification Version 1.0.0.* 2009, Disponibile all'indirizzo: `http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf`.

Elenco delle figure