

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Laboratorio di Amministrazione di Sistemi T

**Analisi e sviluppo di un controller SDN per
applicazioni di controllo remoto di macchine
automatiche**

**CANDIDATO:
Federico Livi**

**RELATORE:
Chiar.mo Prof.
Marco Prandini**

**CORRELATORI:
Franco Callegati
Andrea Melis**

Anno Accademico 2015/2016

Sessione III

Indice

Capitolo 1: Introduzione	6
1.1 Le reti tradizionali	6
1.2 Il modello Software Defined Networking	6
1.3 Il protocollo OpenFlow	7
1.4 Il futuro delle SDN	8
Capitolo 2: Il progetto	9
2.1 Requisiti del prototipo	9
2.2 Casi d'uso	10
2.3 Analisi dei requisiti e studio di possibili tecniche risolutive	10
Capitolo 3: Gli strumenti	13
3.1 Il controller	13
3.2 Open vSwitch	16
3.3 Mininet	16
3.4 Effettiva configurazione degli strumenti utilizzati	17
Capitolo 4: La progettazione del controller	18
4.1 Struttura e aspetti dell'applicazione Ryu	18
4.2 Caricamento dei moduli web del controller	19
4.3 Aggiunta manuale di regole di flow	21
4.4 Visualizzazione delle statistiche della rete	26
4.5 Inserimento di regole in tempo reale	27
4.6 Script per l'effettivo avvio del controller	39
Capitolo 5: Testing del controller	41
5.1 L'ambiente di testing	41
5.2 La rete virtuale	42
5.3 La rete reale	46
Capitolo 6: Considerazioni finali	49
6.1 Conclusioni	49
6.2 Sviluppi futuri	50
Bibliografia	51
Ringraziamenti	52

Indice delle illustrazioni

Illustrazione 1: Modello SDN	7
Illustrazione 2: Il protocollo OpenFlow	7
Illustrazione 3: Crescita (in miliardi) del mercato SDN nel campo dei data center [8].....	8
Illustrazione 4: Casi d'uso del sistema	10
Illustrazione 5: Rete SDN con l'ausilio di chiamate REst.....	11
Illustrazione 6: Aggiunta di uno switch sulla rete.....	12
Illustrazione 7: Normale interazione tra switch e controller	12
Illustrazione 8: Architettura ONOS	13
Illustrazione 9: Interfaccia web offerta da ONOS	14
Illustrazione 10: Architettura Ryu.....	15
Illustrazione 11: Funzionalità di un Open vSwitch.....	16
Illustrazione 12: Architettura Ryu.....	18
Illustrazione 13: Output del controller all'avvio	20
Illustrazione 14: Homepage del controller: index.html	21
Illustrazione 15: Regola aggiunta correttamente	24
Illustrazione 16: Regola rimossa correttamente.....	24
Illustrazione 17: Visualizzazione delle flow entries installate sugli switch: stats.html	26
Illustrazione 18: Tratti di percorso tra due host collegati a due switch differenti	27
Illustrazione 19: Autorizzazione del traffico in tempo reale: live.html	28
Illustrazione 20: Situazione iniziale	33
Illustrazione 21: Host 3 inizia ad inviare pacchetti diretti a Host 4.....	33
Illustrazione 22: Traffico osservato anche nella seconda parte del percorso.....	34
Illustrazione 23: Traffico di ritorno	34
Illustrazione 24: Situazione della rete una volta installate tutte le regole.....	35
Illustrazione 25: Finestra di dialogo per autorizzare o meno il traffico.....	37
Illustrazione 26: Cartella con i file utilizzati dal controller Ryu.....	40
Illustrazione 27: Raspberry Pi 2 Model B.....	41
Illustrazione 28: Prima rete virtuale di test	42
Illustrazione 29: Prova di ping con Mininet.....	42
Illustrazione 30: Statistiche delle regole impostate sulla prima rete virtuale di test.....	43
Illustrazione 31: Inserimento manuale di regole nella prima rete virtuale di test.....	43
Illustrazione 32: Seconda rete virtuale di test	44
Illustrazione 33: Destinazione irraggiungibile	45
Illustrazione 34: Modifica alla seconda rete virtuale di test per evitare loop	45
Illustrazione 35: Rete reale di test.....	46

ABSTRACT

Con il concetto di Software Defined Networking si intende un nuovo modo di concepire la struttura di una rete, ora divisa nel piano dei dati e nel piano di controllo, in modo da raggiungere un più preciso monitoraggio della stessa e garantire maggiore sicurezza, scalabilità e affidabilità.

Questo lavoro di tesi ha come obbiettivo la creazione di un prototipo di rete gestita e monitorata da un controller, l'entità veramente innovativa nel campo delle SDN. Esso avrà funzionalità di firewalling e permetterà un controllo sicuro e una gestione precisa del traffico.

Tale rete verrà prima simulata virtualmente su una macchina per testare vari comportamenti e potenzialità della nuova astrazione, e poi sarà riprodotta realmente con una topologia in scala minore grazie anche all'utilizzo di un Raspberry.

Capitolo 1: Introduzione

1.1 Le reti tradizionali

Il Software Defined Networking è un approccio relativamente nuovo nell'amministrazione delle reti, mirato ad una gestione più precisa, scalabile e dinamica delle stesse, attraverso livelli di astrazione di funzionalità di base [1].

Nelle reti tradizionali, il routing dei pacchetti è effettuato interamente dai vari device di rete: quando un pacchetto con una certa destinazione arriva su un dispositivo di rete, quale un semplice switch o un router, proprio tale device si occupa dell'instradamento del pacchetto attraverso la rete, effettuando scelte di routing e utilizzando vari algoritmi specifici per quel dispositivo.

Perciò i vari set di regole e scelte con cui un pacchetto viene instradato, vengono interamente decisi da quello specifico dispositivo a seconda del firmware montato su quel preciso hardware. Generalmente, a meno che non si tratti di router più costosi e meno utilizzati, i pacchetti destinati ad una certa destinazione vengono trattati tutti allo stesso modo: vengono utilizzate regole di routing statiche indistintamente e il traffico procede attraverso un percorso delineato dai vari dispositivi di rete [2]. Uno dei problemi evidenziato da questa metodologia è sicuramente la limitatezza di questi device durante situazioni di grande traffico sulla rete: tutto ciò porta a grossi problemi di performance. Inoltre, si presentano anche grosse perdite di flessibilità nel dover gestire reti che cambiano dinamicamente, introducendo problemi legati alla sicurezza, all'affidabilità e alla velocità. Di conseguenza, anche la modifica di un solo elemento appartenente alla rete può comportare la riconfigurazione manuale degli altri dispositivi connessi come ad esempio switch e router. Da ciò deriva anche che la manutenzione della rete è piuttosto complessa, in quanto ogni device è configurato in un certo modo con il suo firmware hardware-based ed è scritto in un certo linguaggio a seconda del produttore. Perciò risulta difficile modificare il comportamento desiderato per esempio di un certo switch piuttosto che un altro.

Infine, la configurazione attuale delle reti, si sta allontanando sempre di più da quella che era l'architettura convenzionale client-server, costituita da utenti che utilizzano dei servizi (client) e da componenti che li erogano (server). Per fare un esempio, il cloud Computing non si presta più ad una configurazione convenzionale: le enormi quantità di dati contenuti nei data center devono essere sempre disponibili in modo veloce e sicuro a tutti gli utenti che ne usufruiscono, e ciò deve essere realizzato in maniera efficiente anche se il sistema intanto scala in grandezza.

Sulla base di quanto detto sopra, si può affermare con certezza che in una rete tradizionale il passaggio dei pacchetti su un dispositivo di rete (piano dei dati) e il come essi effettivamente vengano diretti a destinazione (piano di controllo) sono comportamenti interamente decisi dallo specifico device. Ciò è la causa principale di tutti i problemi trattati, in quanto contribuisce ad una netta staticità di una rete.

1.2 Il modello Software Defined Networking

Nel campo del Software Defined Networking, la soluzione ai problemi riportati sopra, consiste nel separare i due piani ed astrarne i comportamenti. Più precisamente, è possibile dividere il modello SDN in tre livelli fondamentali [3]:

1) *Infrastructure Layer*: è la base di ogni rete e include tutti i dispositivi che trasmettono i pacchetti da un nodo all'altro. Contrariamente ad una rete tradizionale, i vari switch del livello non hanno nessun tipo di intelligenza algoritmica per quanto riguarda il forwarding dei pacchetti, ma sono costituiti meramente da agent predisposti a creare un canale di comunicazione con il secondo livello, il Control Layer.

2) *Control Layer*: è la parte centrale dell'architettura SDN ed è la vera differenza tra le reti tradizionali e le Software Defined Network. Si tratta di un livello adibito a gestire tutti i comportamenti e le

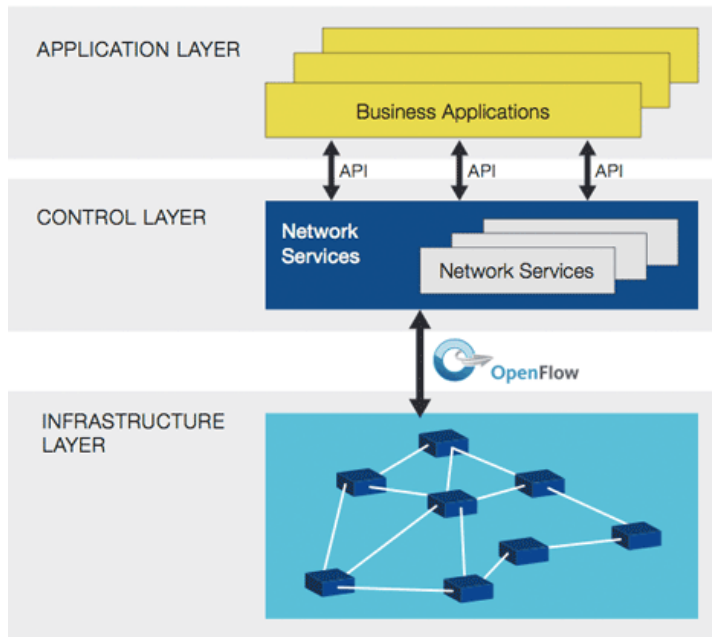


Illustrazione 1: Modello SDN

dinamiche del livello sottostante, prendendo decisioni di routing non più in maniera statica, ma dinamica e a seconda del traffico e della topologia della rete. Inoltre, se la rete scala (se per esempio vengono aggiunti dispositivi al layer sottostante), il piano di controllo può decidere di modificare scelte di routing precedentemente attuate. In questa maniera la rete può cambiare dinamicamente e adattarsi nel modo più completo e preciso alla nuova topologia.

Questo piano è gestito da un'entità solitamente chiamata controller, che ha il compito di monitorare tutti i cambiamenti della rete, di filtrare traffico indesiderato e di attuare particolari protocolli per garantire maggiore sicurezza.

3) *Application Layer*: è il livello più intelligente in una rete SDN. Astrae dai livelli sottostanti e consiste in una varietà di strumenti ed interfacce grafiche dedicate all'utente finale. In tal modo si riesce a gestire il reale funzionamento e monitoraggio della rete attraverso il *control layer*.

1.3 Il protocollo OpenFlow

Utilizzando il concetto di SDN, gli amministratori di rete posso controllare i vari flussi di rete e cambiare i comportamenti degli switch, anche in maniera del tutto dinamica e soprattutto centralizzata: non occorre più trattare ogni singolo switch individualmente.

Il protocollo di rete utilizzato per gestire le comunicazioni tra *Infrastructure Layer* e *Control Layer* è OpenFlow.

Gestito dall'Open Networking Foundation (ONF), questo protocollo agisce a livello 2 del modello OSI, garantendo l'accesso al piano di inoltro di router e switch [3] e permettendo la modifica dinamica via software delle tabelle di routing. Un concetto fondamentale di questo protocollo è quello di flow (flusso), inteso come porzione di traffico di una certa tipologia tra due specifici host.

All'interno di ogni switch della rete, questo protocollo prevede l'utilizzo di flow tables e cioè tabelle che raccolgono le flow entries, ovvero tutte le regole di traffico impostate precedentemente. Quando un pacchetto arriva inizialmente sullo switch e non c'è nessuna regola per il suo instradamento nelle flow tables, esso viene inviato di default al controller attraverso un canale dedicato. Quest'ultimo perciò ha la responsabilità di decidere dove dovrà essere inviato il pacchetto e potrà o ignorarlo eliminandolo, o impostare una regola appropriata nello switch.

Il prossimo pacchetto entrante nello switch perciò seguirà semplicemente le indicazioni impostate dal controller nella tabella.

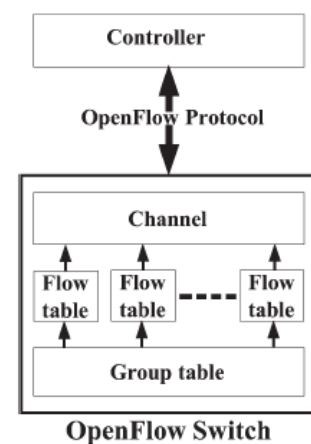


Illustrazione 2: Il protocollo OpenFlow

Infine una group table è una ulteriore astrazione che permette di raccogliere flow entries identiche ma con diverse destinazioni in modo da gestirne meglio il comportamento voluto [5].

All'interno di uno switch o di un router normale, le tabelle di routing sono scritte e pensate in maniera diversa a seconda del produttore, ma esistono set di funzioni comuni a tutti i dispositivi di rete. Proprio qua interviene OpenFlow, sfruttandoli per creare un aggancio comune a tutti.

In questo modo, grazie a questo protocollo, i ricercatori per esempio possono testare nuovi protocolli di routing, nuove tipologie di indirizzi alternativi a quelli IP [7] e anche nuovi modelli di sicurezza. Infatti, risulta evidente che utilizzando OpenFlow è molto semplice separare sulla stessa rete particolari percorsi dedicati solamente ad una certa tipologia di traffico.

Perciò possono essere creati prototipi di controller più sofisticati che dinamicamente rimuovono e aggiungono flow entries a seconda dell'esperimento considerato in quel momento.

È stato inoltre dimostrato che un controller centralizzato su una rete con molti switch interconnessi utilizzando OpenFlow, è abbastanza veloce da processare nuove flow entries e programmare gli switch stessi. Infatti, un controller basato su Ethane, un prototipo di interfaccia che poi diede vita a OpenFlow, venne installato su un desktop pc economico e riuscì a processare più di 10000 flow entries al secondo, abbastanza per un campus universitario di grosse dimensioni [7].

Un aspetto molto positivo di questo protocollo è che per implementare le funzionalità di OpenFlow su vecchi dispositivi di rete non c'è bisogno di nessuna modifica hardware: gli switch configureranno un canale sicuro verso il controller utilizzando solamente il loro processore già esistente.

Perciò OpenFlow è anche un ottimo compromesso per gli amministratori di rete e per i ricercatori: esso infatti rende molto facile la sperimentazione di nuovi protocolli in configurazioni di rete molto eterogenee. Inoltre i produttori di device di rete quali router e switch, non hanno bisogno di esporre logiche interne dei propri dispositivi perché OpenFlow sfrutta set di regole comuni a tutti e hardware già esistente.

1.4 Il futuro delle SDN

I possibili utilizzi di questa nuova tecnologia sono veramente tanti e tutti sottoposti a prototipazioni di vario genere alla ricerca di uno standard ben definito.

Si parla di numerosi progetti in corso d'opera, dall'utilizzo di SDN per pura ricerca e testing di nuovi

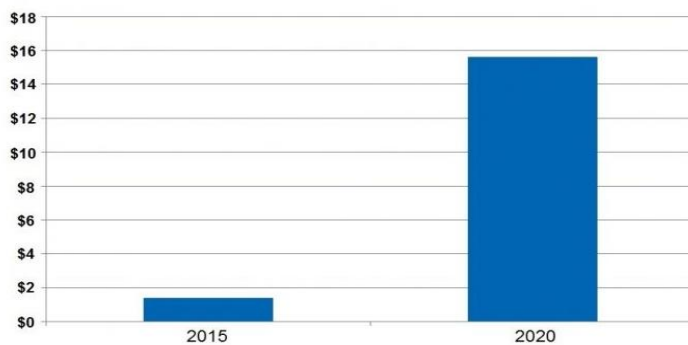


Illustrazione 3: Crescita (in miliardi) del mercato SDN nel campo dei data center [8]

protocolli, alla netta semplificazione delle reti in maniera da garantire copertura migliore di rete anche nelle aree rurali, separando l'ISP business dal vero e proprio sviluppo della logica di rete.

Inoltre anche dal punto di vista delle grandi multinazionali, le SDN rappresentano un risparmio netto di denaro: in un data center con moltissimi switch, essi ora possono essere tutti controllati e configurati in un'unica locazione [2].

A prova di questo, si sa che il mercato dello sviluppo e della ricerca di applicazioni SDN ha raggiunto \$1.1 miliardi di dollari nella prima metà del 2016 (un miglioramento del 42% rispetto alla prima metà del 2015). Inoltre una significativa crescita è prevista per il 2017 a riprova che il processo di standardizzazione delle SDN è appena iniziato ma avrà notevoli novità in un futuro molto prossimo [6].

Capitolo 2: Il progetto

2.1 Requisiti del prototipo

Il sistema da realizzare deve essere in grado di regolare e gestire il traffico scambiato tra i vari host presenti su una rete. Per fare ciò nella logica del Software Defined Networking, si utilizza un controller che dovrà essere sviluppato e modellato per realizzare comportamenti specifici sul traffico della rete.

In definitiva, il controller, una volta attivato, sarà in grado di effettuare le seguenti operazioni:

- 1) Riconoscere ogni entità presente sulla rete, qualunque sia il numero e qualunque sia la topologia con la quale esse sono interconnesse. In particolare dovrà essere a conoscenza di tutti gli switch, di tutti gli host e di tutti i collegamenti tra essi presenti sulla rete;
- 2) Permettere l'inserimento da parte dell'utente di regole di traffico attraverso la modifica delle flow table degli switch. Esse possono essere più o meno specifiche: la più generale prevede il passaggio di qualunque tipologia di traffico tra due host, mentre più specificamente è anche possibile determinare con quale protocollo tra i due host sia possibile lo scambio di pacchetti. È possibile identificare l'host (sia sorgente che destinatario) attraverso l'utilizzo di un indirizzo IP, di un MAC address o di entrambi. Successivamente deve essere anche possibile l'eliminazione di tali regole.
- 3) Controllare lo stato attuale della rete. Infatti l'utente potrà visualizzare tutte le statistiche per le flow degli switch presenti sulla rete. Per ogni regola egli potrà osservare il numero totale di pacchetti transitati nella rete, i byte trasferiti e la durata del collegamento (cioè da quanto tempo è stata effettivamente inserita tale regola di flow). Occorrerà che di default tali informazioni vengano aggiornate ogni tot di tempo per rendere il sistema effettivamente attendibile, ma per questioni di performance sarà anche possibile interrompere tale refresh automatico.
- 4) Effettuare operazioni in tempo reale simili a quelle di un firewall. Infatti solamente certo traffico potrà passare nella rete. Ogni volta che il controller rileverà che su uno degli switch della topologia c'è un pacchetto in attesa di essere inviato al destinatario, esso dovrà bloccare quel tipo di traffico e notificare l'utente. Egli, se valuterà attendibile il mittente e il tipo di traffico destinato al ricevente, potrà decidere di autorizzare quel traffico e perciò impostare una regola di flow tra sorgente e destinazione. In caso contrario, i pacchetti verranno eliminati ed eventuali richieste future, sempre dallo stesso mittente e con la stessa tipologia di traffico, verranno ignorate. Ovviamente il controller non notificherà l'utente se, anche dopo un certo lasso di tempo, passerà lo stesso traffico precedentemente autorizzato.

I percorsi autorizzati da questa funzione di firewall possono essere eliminati in qualsiasi momento utilizzando i servizi descritti nel secondo punto.

Per testare il controller occorrerà procedere in due modi:

- 1) Simulare più topologie di rete su una macchina virtuale, in modo da verificare il corretto funzionamento del controller in molte situazioni e configurazioni di essa;
- 2) Procedere alla creazione di una singola topologia di rete in scala minore, ma testarla con componenti reali (simulando però gli switch): è previsto l'utilizzo di un Raspberry come contenitore del controller.

2.2 Casi d'uso

I requisiti descritti nel precedente sotto paragrafo possono essere sintetizzati schematicamente con la rappresentazione che segue:

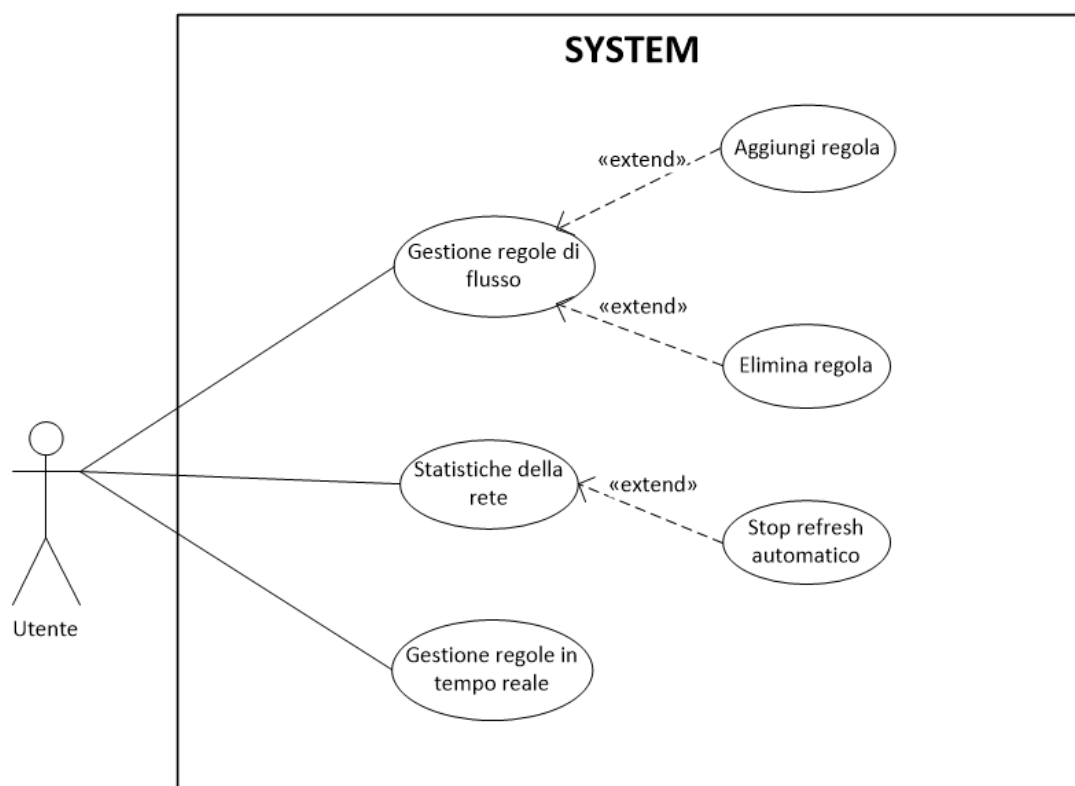


Illustrazione 4: Casi d'uso del sistema

L'attore del sistema è solamente uno: l'utente finale. Egli potrà effettuare una molteplicità di servizi di monitoraggio e controllo del sistema in base alle proprie esigenze.

È importante sottolineare che tale schema è anche di estrema utilità per provare già ad esprimere una sorta di interfaccia grafica a cui dovrà tendere la progettazione del prototipo.

2.3 Analisi dei requisiti e studio di possibili tecniche risolutive

Dalla lettura dei requisiti risulta ovvio che il pilastro fondamentale di un prototipo del genere è sicuramente il controller: in una rete SDN, esso prende tutte le decisioni di traffico, ne esegue il monitoraggio e descrive le statistiche generali di tutti i componenti di rete. Il focus principale perciò deve essere posto sul secondo livello, *il Control Layer*, ovviamente strettamente connesso all'*Infrastructure Layer* tramite OpenFlow.

Inoltre, è chiaro che allo scopo di realizzare un controller del genere risulti di grande importanza che esso sia dotato di un'interfaccia grafica affinché l'utente finale possa effettuare operazioni sul traffico in modo semplice, senza toccare in alcun modo la complessità delle operazioni che vengono eseguite a più basso livello. Per fare ciò è necessario realizzare quindi anche un componente di livello *Application Layer* del modello SDN.

In effetti la gestione del traffico deve tener conto di tutte le regole del protocollo OpenFlow e per questo il controller dovrà esserne perfettamente integrato.

Il modo più completo per realizzare ciò è fare in modo che le impostazioni e la gestione del controller siano accessibili tramite browser. Perciò tale controller dovrà essere anche costituito da una parte di applicazione web.

Nasce quindi l'esigenza di trovare un controller adatto tra le varie proposte che supporti anche questa funzionalità: deve prevedere l'utilizzo di un'interfaccia web per la gestione da parte dell'utente o comunque dei modi per realizzarla.

Per rendere totalmente gestibile e controllabile la realizzazione dell'applicazione web si potrebbe pensare di utilizzare delle REst API.

L'interfaccia di programmazione di un'applicazione REst (REpresentational State Transfer) utilizza richieste HTTP (GET, PUT, POST, DELETE) per gestire dei dati. Si tratta di un approccio molto spesso utilizzato nell'ambito dei servizi web ed è generalmente preferito al più complesso e robusto Simple Object Access Protocol (SOAP) perché utilizza meno larghezza di banda e quindi meno risorse [9].

Attraverso le REst API è quindi possibile ricevere, modificare, creare ed eliminare risorse e tutto ciò viene fatto in modo stateless, senza perciò lasciare traccia di interazioni passate.

Questa modalità è ideale nel mondo del web perché componenti stateless possono essere facilmente modificabili se qualcosa fallisce. Possono inoltre scalare per garantire il bilanciamento del traffico.

Nella logica di questo prototipo, le REst API possono essere utilizzate per ricevere e modificare risorse del controller e degli switch, nonché richiedere il caricamento di pagine web.

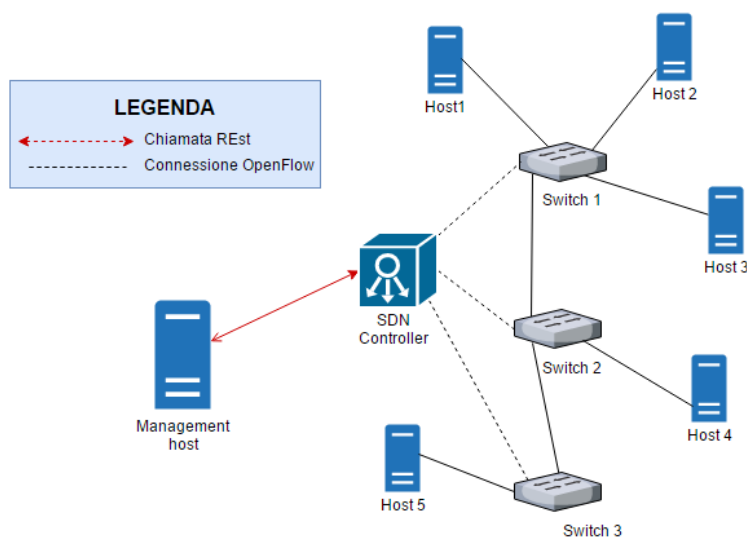


Illustrazione 5: Rete SDN con l'ausilio di chiamate REst

Perciò per esempio, risulterebbe molto semplice risolvere il primo quesito del problema attraverso una REst API che tramite richieste HTTP informi gli switch della rete di notificare il controller del loro stato. In questo modo è possibile vedere se ci sono state aggiunte o rimozioni di questi dispositivi nella rete.

Si potrebbe pensare infatti di eseguire quella particolare chiamata REst ciclicamente ogni tot di tempo per osservare gli eventuali cambiamenti della rete.

Le REst API nell'ottica di questo problema quindi risulterebbero dei mezzi per risolvere il problema della comunicazione tra applicazione web e layer sottostanti.

Per quanto riguarda gli altri punti del problema, sarebbe ottimo utilizzare JQuery o Javascript per gestire l'interazione dell'utente con il browser e collegare direttamente i loro metodi a chiamate REst per manipolare i comportamenti del controller sottostante. Si pensi al secondo e al terzo punto dove basterebbe creare delle pagine HTML arricchite con Javascript o JQuery per raccogliere i dati dell'utente (nel secondo punto) o per informarlo sullo stato dei livelli sottostanti (terzo punto), tramite anche l'interazione con OpenFlow.

Il quarto punto risulta il più complicato: occorre pensare ad un modo di intercettare in tempo reale tutti i pacchetti che arrivano sugli switch della rete in modo da bloccarli e chiedere all'utente se devono essere autorizzati a raggiungere la destinazione.

Per capire il miglior modo per realizzare questo comportamento, è necessario comprendere come il protocollo OpenFlow lavori a basso livello.

Quando uno switch viene connesso nella rete, deve configurarsi, notificando inizialmente il controller della sua presenza. Realizza ciò tramite invio di messaggi di OFPT_HELLO (per negoziare la versione del protocollo OpenFlow da utilizzare), OFPT_FEATURES_REQUEST e OFPT_FEATURES_REPLY (per richiedere e accettare l'identità e le capacità di base di uno switch) e OFPT_SET_CONFIG (per settare dei parametri di configurazione allo switch).

Nel normale funzionamento di uno switch nella rete, attraverso messaggi di OFPT_MULTIPART_REQUEST e OFPT_MULTIPART_REPLY, ottimizzati anche per processare grandi quantità di informazioni, il controller e gli switch si scambiano informazioni su stato del componente, delle sue porte, delle flow e su tanti altri comportamenti.

Quando un nuovo pacchetto raggiunge uno switch, esso viene inviato al controller tramite OFPT_PACKET_IN, il quale risponderà a sua volta installando una nuova flow entry nella flow table dello switch tramite messaggio di OFPT_FLOW_MOD.

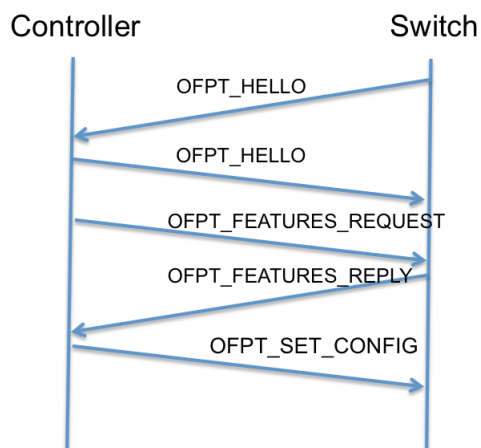


Illustrazione 6: Aggiunta di uno switch sulla rete

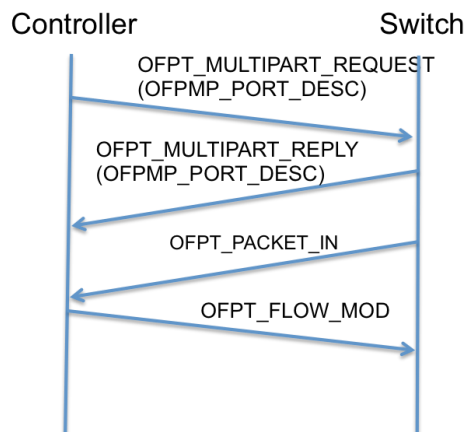


Illustrazione 7: Normale interazione tra switch e controller

Risulta evidente che per realizzare la caratteristica richiesta dal quarto punto dei requisiti, occorrerà intercettare in qualche modo i messaggi di packetIn diretti al controller e chiedere all'utente se autorizzarli o meno a passare sulla rete e ad arrivare a destinazione.

Una ulteriore difficoltà sta poi nel realizzare il concetto di flow completo da sorgente a destinazione. Ormai è chiaro che il controller sarà notificato ogni qualvolta un pacchetto entrerà in uno switch, ma non ha molto senso avvisarlo per pacchetti che sono già stati autorizzati nel tratto precedente.

Questo succede perché una semplice flow è una regola che dice come trattare un pacchetto nell'ambito di un singolo switch che arriva su una determinata porta e che deve uscire da un'altra. Perciò la flow entry, in sé per sé, non rappresenta assolutamente l'intero cammino da host mittente a host destinatario, ma ne determina solamente una parte.

Quindi, se in una rete per arrivare a destinazione occorre attraversare per esempio 10 switch, l'utente dovrà autorizzare 20 tratte di percorso. Risultano il doppio in quanto è necessario autorizzare le tratte mittente-destinatario ma anche destinatario-mittente.

Ciò ovviamente è assurdo perché non ha senso autorizzare tutti i singoli pezzi di percorso se l'utente ha già autorizzato il passaggio di quel pacchetto sul primo switch. Infatti, solamente analizzando il primo pacchetto, egli è già a conoscenza dell'indirizzo del mittente, del destinatario e del tipo di traffico che è in attesa di circolare sulla rete.

Per cui ha già tutte le informazioni per decidere se autorizzarne o meno il passaggio.

Dopo l'interazione con l'utente, le autorizzazioni delle altre tratte di percorso devono essere perciò rese trasparenti. Di conseguenza, risulta fondamentale pensare di realizzare scelte algoritmiche che chiedano all'utente di autorizzare solamente il percorso una volta che i pacchetti cominciano a fluire nel primo switch verso destinazione. Tutte le regole delle altre tratte devono inserirsi autonomamente nelle flow tables degli switch.

Ovviamente fanno eccezione protocolli che comunicano nelle due direzioni in modo diverso: per esempio il comando di rete iperf può funzionare in un verso tramite passaggio di pacchetti TCP o UDP e nell'altro tramite risposte di tipo ICMP. In tal caso ovviamente dovrà essere notificato all'utente il diverso tipo di traffico pronto a circolare verso destinazione.

Capitolo 3: Gli strumenti

3.1 Il controller

La scelta del controller non deve essere sottovalutata, perché occorre considerare il componente più valido possibile per la realizzazione del prototipo richiesto.

In questo senso è fondamentale che soprattutto esso permetta la creazione di un *Application Layer* per l'interazione con l'utente e che sia il più versatile possibile per la realizzazione del comportamento richiesto dal quarto punto dei requisiti: deve perciò permettere un controllo totale e in seguito automatico delle flow impostate nella rete.

Attualmente esistono diversi controller [11] per la gestione di una Software Defined Network, ma i più interessanti per il nostro prototipo sono sicuramente quelli open source.

Tra questi si distinguono quelli che sicuramente sono stati i pionieri in questo settore:

- 1) Opendaylight: si tratta di un progetto open source fondato nel 2013 e seguito da The Linux Foundation che sviluppa tecnologie non solo nell'ambito delle SDN ma anche delle NFV (processo di virtualizzazione delle funzionalità di rete svolte da apparati di telecomunicazione fisici). È sviluppato in Java. [12]
- 2) FloodLight: open standard nel campo delle SDN gestito dall'ONF. Può gestire numerosi apparati di rete senza risentirne in performance ed è facilmente installabile con la minima quantità di dipendenze. È sviluppato in Java. [13]
- 3) ONOS: è un vero e proprio sistema operativo di rete che garantisce grande scalabilità ed efficienza grazie all'utilizzo di numerose astrazioni per creare applicazioni e servizi. È strutturato in moduli software, è scritto in Java ed è molto utilizzato nelle reti SDN odierne.
- 4) RYU: è un framework SDN modulare che prevede numerosi API per gli sviluppatori. È scritto in Python.

Si è scelto di approfondire lo studio di due controller: ONOS e Ryu.

Il primo è stato considerato inizialmente per l'intuitività generale e per i molteplici servizi offerti.

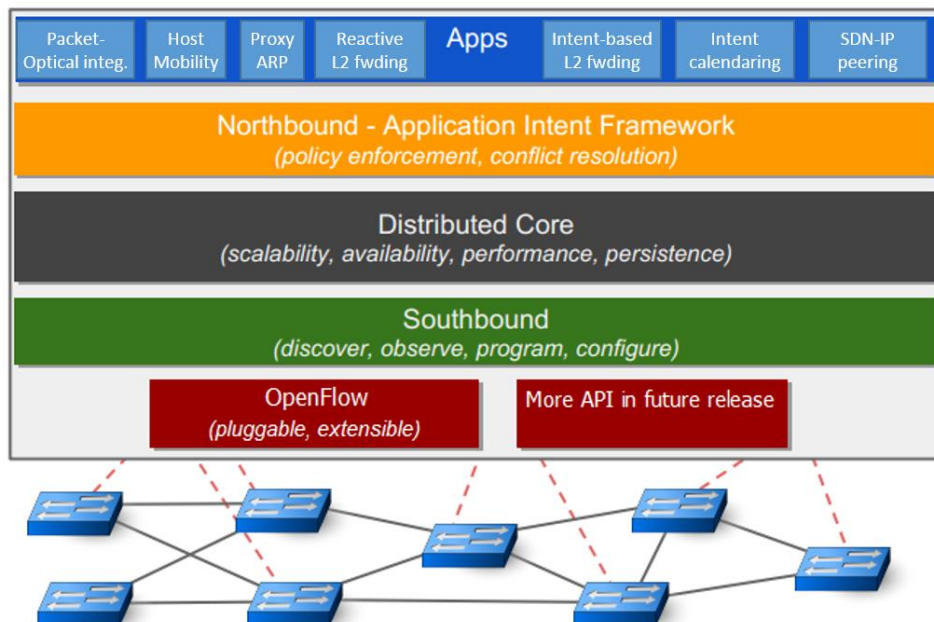


Illustrazione 8: Architettura ONOS

Costruito a livelli e programmato in Java, ONOS risolverebbe i problemi iniziali per la realizzazione del prototipo. Infatti include un'interfaccia web del tutto completa per la gestione del controller, garantendo numerose funzionalità attraverso un sistema modulare.

Ciò che non serve può essere disattivato tramite specifici comandi.

Tramite una pagina web possono essere visualizzate tutte le informazioni della rete, dei percorsi e delle regole installate tramite un'interfaccia grafica dell'intera topologia:

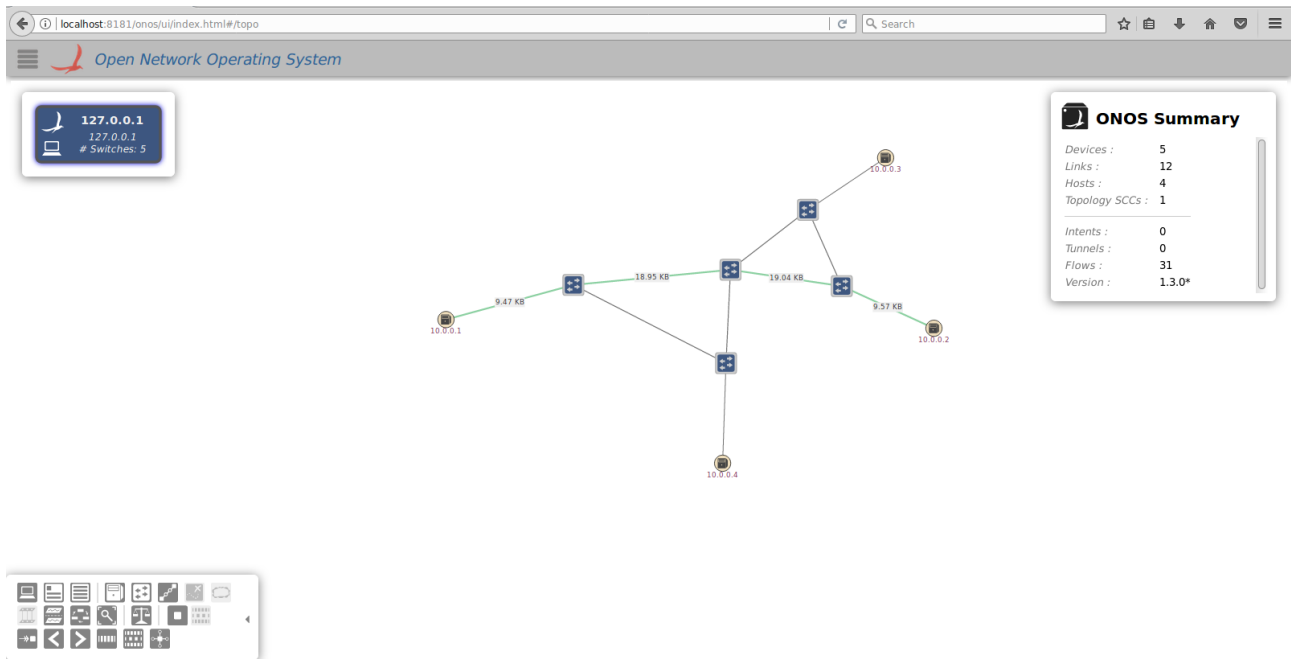


Illustrazione 9: Interfaccia web offerta da ONOS

Possono essere visualizzate inoltre statistiche degli switch e degli host connessi sulla rete.

Come possiamo notare dall'illustrazione appena sopra, il framework offre di default anche servizi per la risoluzione di topologie circolari e multipath.

Infatti, nella topologia considerata, costituita da 5 switch e 4 host, sono presenti più possibilità di percorso per raggiungere la stessa destinazione. Con efficienti algoritmi di shortest path e spanning tree, ONOS evita loop e stabilisce la strada migliore per giungere a destinazione.

La cosa veramente rivoluzionaria però sta nel concetto di intent offerto da ONOS.

Si tratta di un'astrazione che permette alle applicazioni di specificare dei comportamenti desiderati attraverso direttive simili a policy [13]. È previsto inoltre un vero e proprio framework per la gestione e la creazione di intent. Nel campo del nostro prototipo, questa funzionalità di ONOS si tradurrebbe nella semplicità con cui possono essere aggiunti tratti di percorso sotto il controllo del layer 2 di SDN. Ciò semplificherebbe e non poco l'impostazione iniziale del problema descritto al quarto punto dei requisiti. Una volta autorizzato un pezzo di percorso, automaticamente il resto della tratta per arrivare a destinazione potrebbe essere trovato dal framework di ONOS.

Inoltre, proprio grazie agli intent, è possibile decidere con accuratezza quale cammino scegliere, quali switch da attraversare e da quali porte far fluire il traffico.

Si tratta di una potenzialità non da poco. Occorrerebbe solamente interrogarsi su come implementare questa funzionalità nel prototipo per avvicinarsi il più possibile al comportamento richiesto.

L'altra alternativa analizzata è quella offerta dal controller RYU.

Anch'esso costituito da un'architettura modulare, è principalmente costituito da script scritti in Python che ne regolano i comportamenti e le funzionalità. Offre numerose API agli sviluppatori e funziona essenzialmente ad eventi: scrivere un'applicazione Ryu significa fondamentalmente associare specifici comportamenti ad eventi causati dall'interazione con l'*Infrastructural Layer* e quindi con OpenFlow.

Dispone di una documentazione abbastanza ricca dove vengono spiegati nel dettaglio i moduli che costituiscono questo framework [16].

Più dettagliatamente, un'applicazione Ryu è costituita da moduli scritti in Python, tutti aventi una superclasse comune. Gli eventi generati da OpenFlow vengono intercettati da Ryu e possono essere gestiti attraverso degli handler appositi, individuati da decoratori.

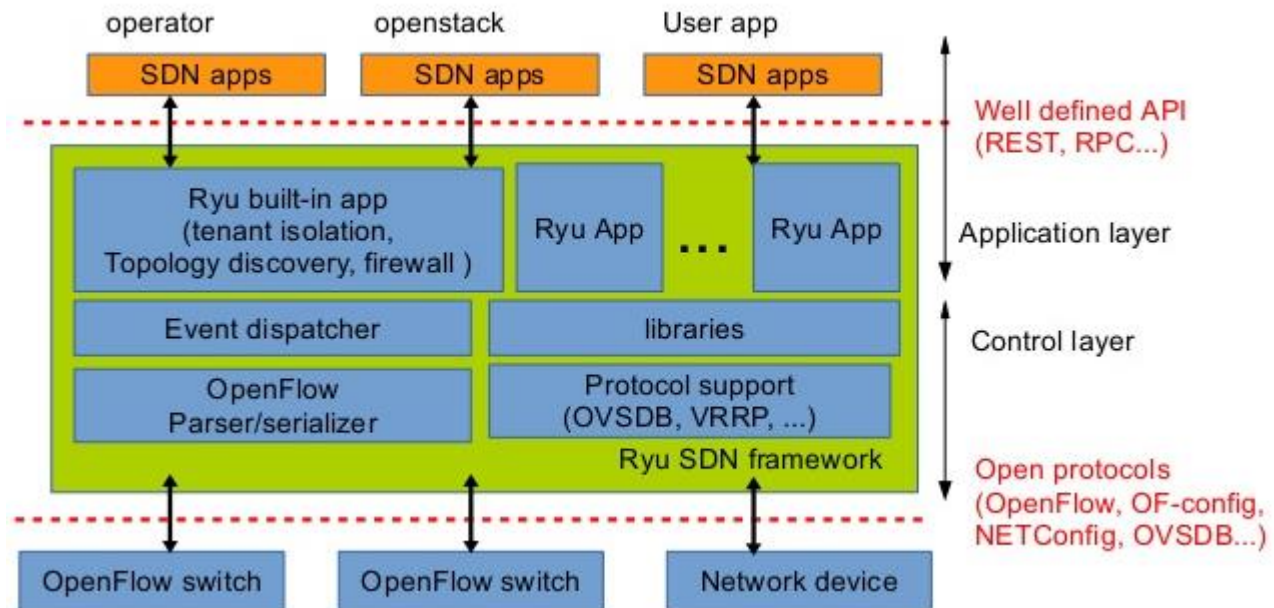


Illustrazione 10: Architettura Ryu

Offre inoltre servizi già disponibili all'integrazione in nuove applicazioni come l'importante Topology discovery, che permette di delineare precisamente tutta la topologia di rete.

Nello specifico caso della realizzazione del nostro prototipo, ciò risulta molto comodo.

Inoltre Ryu è anche molto interessante in quanto offre completo supporto alle REst API, addirittura offrendone tantissime di default. Tramite esse [16], è possibile accedere a numerose statistiche della rete: per esempio si possono osservare le flow tables di uno specifico switch e lo stato delle sue porte. Inoltre ci sono API di default che permettono anche di installare nuove flow solamente utilizzando una chiamata REst, e cioè un banale URL che digitato in un browser, tramite richieste HTTP attiverà dei metodi sottostanti interni all'architettura del controller stesso.

Dopo numerosi test di entrambi i controller per verificare il miglior componente che realizzasse al meglio le funzionalità richieste, si è scelto di utilizzare Ryu.

ONOS offre un framework veramente esteso e ricco di funzionalità, ma ciò dice poco nel nostro progetto, perché molte di esse non servono. Tale architettura infatti è un vero e proprio sistema operativo, con tutte le complessità tecniche che ne derivano.

Al fine di realizzare il prototipo voluto, non è necessario complicare così tanto il progetto. Realizzare un controller del genere con le funzionalità però richieste dal progetto, richiederebbe uno sforzo ulteriore nel comprendere le logiche interne di ONOS e come esse vengano effettivamente realizzate. È pur vero che tale sistema operativo di rete offre un'intuitività finale senza paragoni, ma è anche vero che essa nasconde molte scelte implementative realizzate dagli sviluppatori del framework.

Al contrario, il controller da realizzare deve essere il più chiaro possibile nelle scelte effettuate, senza lasciare nessun comportamento privo di alcuna spiegazione implementativa.

In questo senso, Ryu offre un framework dove praticamente sono assenti tutte le astrazioni comode di ONOS, ma permette allo sviluppatore di costruire la propria applicazione scegliendo cosa integrare come funzionalità di default offerte. Infatti per realizzare un controller Ryu occorre mettere mano a script scritti in Python e fortemente integrati con Openflow.

Tramite la ricezione di eventi dal layer sottostante, è possibile sviluppare comportamenti nuovi o scegliere di adottare servizi di default. In questo senso, risultano molto comode allo sviluppatore le numerose REst API offerte dal framework modulare.

Perciò la scelta del controller ha visto come vincitore Ryu, in quanto predispone di un framework molto più gestibile sotto tutti i punti di vista e privo di complessità e astrazioni invece presenti in controller più corposi come ONOS.

3.2 Open vSwitch

I componenti di rete utilizzati sono degli switch e perciò nasce l'esigenza di trovare un modo per integrarli nella rete. Non essendo possibile considerare degli switch fisici allo scopo di testare la rete, occorre perciò trovare dei modi per simularli, sia sulla rete virtuale che sulla rete fisica richiesta dai requisiti del progetto.

Open vSwitch è una implementazione open-source di uno switch multilivello.

Lo scopo principale di OVS è garantire la simulazione di questi componenti di rete all'interno di ambienti dove esiste dell'hardware virtualizzato, supportando una molteplicità di protocolli di rete. La cosa interessante è che uno switch OVS può essere anche utilizzato su hardware già esistente [18].

Tramite la documentazione online, si può notare che un OVS può essere gestito tramite vari comandi su Linux. Per esempio una volta diventati utenti root, sarà di estrema utilità visualizzare tutti gli switch creati e attivi sulla rete, e si potrà fare ciò attraverso il comando `ovs-vsctl show`.

Inoltre si può accedere direttamente ad uno switch su una determinata rete, visualizzandone informazioni complete tramite comandi come `ovs-ofctl dump-ports-desc [nome switch]` (per visualizzare la situazione completa delle porte presenti sullo switch) o come `ovs-ofctl dump-flows [nome switch]` (per visualizzare le flow entries impostate nello switch).

Molto importante inoltre risulta la possibilità di un OVS di realizzare automaticamente (attraverso dei comandi) alcuni protocolli per la gestione di percorsi di rete circolari, come lo spanning tree o il rapid spanning tree.

Resta perciò solamente un punto aperto: come realizzare in modo semplice delle topologie particolari di rete integrando switch virtuali gestiti da Open vSwitch e un numero arbitrario di host di prova, al fine di testare il comportamento del controller in varie situazioni? La risposta è: Mininet.

3.3 Mininet

Mininet è uno strumento molto utile nella simulazione di reti. In effetti esso permette la creazione di una rete virtuale configurata a seconde delle esigenze [19]. È possibile perciò specificare quanti switch ci sono sulla rete e come essi sono collegati tra di loro e tra gli host. Mininet inoltre prevede anche l'impostazione di un controller, che può essere specificato alla creazione della rete (altrimenti viene utilizzato quello di default).

Questo strumento è dotato di una propria interfaccia a riga di comando, che permette di monitorare le statistiche e i collegamenti della rete. Infatti, una volta configurata la rete, Mininet permette all'utente di interagire con i vari switch e gli host simulati.

In tal modo è possibile effettuare veri e propri collegamenti e testare protocolli di rete (ad esempio è possibile effettuare una semplice operazione di ping tra due host simulati nella rete).

La configurazione di rete può essere settata in due modi:

- 1) Tramite interfaccia propria di Mininet attraverso specifici comandi e opzioni;
- 2) Tramite creazione di script di Python poi caricati all'avvio di Mininet.

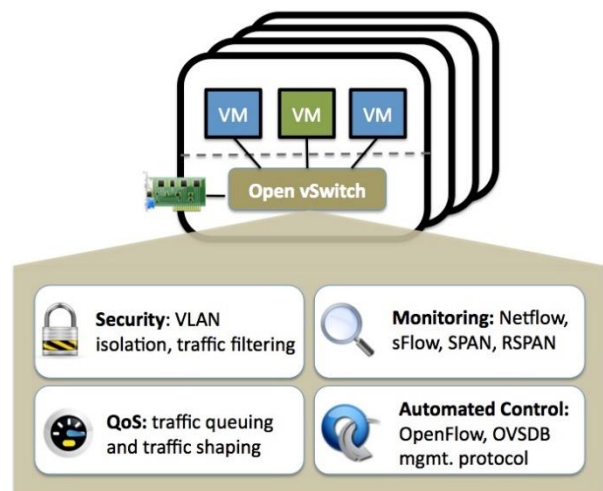


Illustrazione 11: Funzionalità di un Open vSwitch

Con semplici comandi questo strumento permette di creare delle topologie anche piuttosto complicate, al fine di testare al meglio i funzionamenti dei vari protocolli di rete tra cui anche OpenFlow (e la sua interazione con il *Control Layer*).

È importante tenere presente che Mininet automatizza il processo di creazione della rete, utilizzando dei comandi insiti nel kernel di Linux per la simulazione delle configurazioni volute.

Effettua ciò creando host come semplici namespace a livello kernel e switch attraverso l'utilizzo di tutte le funzionalità offerte da Open vSwitch.

È anche bene precisare che si può automatizzare tutto ciò che fa Mininet anche con dei semplici script ad hoc e anzi, in alcuni casi risulta più conveniente per configurare una certa impostazione di rete solamente avviando un certo file.

Quest'ultimo aspetto comunque sarà approfondito nel quinto capitolo.

3.4 Effettiva configurazione degli strumenti utilizzati

Per effettuare i test della rete, sono state utilizzate delle macchine virtuali e un Raspberry Pi 2 (Model B).

Per quanto riguarda le macchine virtuali, esse sono state configurate installando delle immagini OVA prese direttamente da sdnhub [20]. Questo portale molto utile per chi si approccia per la prima volta al mondo delle SDN, tempo fa ha reso disponibile delle immagini preconfezionate con tutto il necessario per il testing di reti SDN.

Virtualizzate tramite VMware, esse sono già complete di tutti gli strumenti necessari visti sopra.

Le versioni utilizzate per il testing delle reti sono le seguenti:

- 1) Mininet 2.2.1;
- 2) OVS 2.3.90;
- 3) Ryu 4.9.

Il Raspberry è utilizzato nella rete reale come contenitore del controller Ryu (nonché di un Open vSwitch). Esso è stato configurato installando come sistema operativo la versione Jessie di Raspbian. L'installazione di OVS e di Ryu non ha dato particolari problemi (sono stati installati semplicemente con i comandi `apt-get` da terminale). Al contrario Mininet ha avuto dei problemi di installazione.

In effetti dopo aver scaricato il package necessario all'installazione tramite `git clone git://github.com/mininet/mininet`, spostandoci nella cartella `util` di mininet, il file `install.sh` ha dato errori a terminale:

```
Detected Linux distribution: Raspbian 8.0 jessie armv6l
Install.sh currently only supports Ubuntu, Debian, Redhat and Fedora.
```

Aperto lo script di installazione [21], dopo un'analisi del codice risulta evidente che lo script memorizza nella variabile `DIST` la versione di Linux installata attraverso l'output del comando `sb_release -is`.

Confrontando questa variabile con i soli valori ammessi (Ubuntu, Debian, Fedora, RedHatEnterpriseServer, SUSE LINUX), lo script blocca l'installazione di Mininet perché la versione installata risulta essere Raspbian (voce non compresa in quelle ammesse).

Il problema può essere ovviato considerando che Raspbian Jessie è una versione specifica di Debian Jessie ottimizzata per l'utilizzo con i Raspberry. Perciò è possibile commentare le righe del check per far proseguire correttamente l'installazione del tool (commentare dalla riga 83 alla riga 86 del codice). In definitiva, le versioni degli strumenti visti in precedenza utilizzate per il testing su Raspberry sono le seguenti:

- 1) Mininet 2.3.0d1;
- 2) OVS 2.3.0;
- 3) Ryu 4.10.

Capitolo 4: La progettazione del controller

4.1 Struttura e aspetti dell'applicazione Ryu

Come già sottolineato, la struttura di un'applicazione Ryu è prevalentemente modulare. Tramite script di Python che estendono classi già esistenti nel framework, è possibile sviluppare un'applicazione Ryu senza dover scrivere tutto da zero.

Il controller Ryu è perfettamente integrato con OpenFlow: per esempio tale protocollo richiederebbe di definire e specificare certe operazioni come l'handshake iniziale tra switch e controller, ma siccome il framework di Ryu se ne occupa interamente, non è necessario che lo sviluppatore si preoccupi di certi comportamenti.

Il componente fondamentale è sicuramente l'app manager (`ryu.base.app_manager`) che ha il compito di caricare gli altri componenti e di inviare messaggi tra le varie applicazioni Ryu.

Inoltre, ne esistono molti altri tra cui quelli per la gestione dell'interazione con OpenFlow (encoder e decoder di tale protocollo e specifiche implementazioni a seconda della versione).

Sono anche comprese nel framework vere e proprie applicazioni che gestiscono automaticamente funzionalità usate normalmente nella scrittura di un controller Ryu, al fine di migliorare l'esperienza dello sviluppatore con parti di codice già realizzate.

In questo senso è stata fondamentale la conoscenza del funzionamento dell'applicazione `ExampleSwitch13`[22], che permette di gestire il funzionamento di uno switch sulla rete.

Infatti, tramite lo studio del codice è stato possibile venire a conoscenza di come funzionino gli eventi in Ryu e i rispettivi handler, nonché imparare la struttura di un'applicazione Ryu e le relative classi

da implementare o estendere nella scrittura del controller.

Per gestire le chiamate REst, Ryu ha bisogno di un web server da far partire all'avvio del controller. Fortunatamente il framework prevede già tale funzione realizzata tramite un'applicazione WSGI.

Il Web Server Gateway Interface (WSGI) è un protocollo di trasmissione che stabilisce e descrive comunicazioni ed interazioni tra server ed applicazioni web scritte nel linguaggio Python.

È quindi l'interfaccia standard del web service per la programmazione in Python.

In parole povere, il protocollo specifica

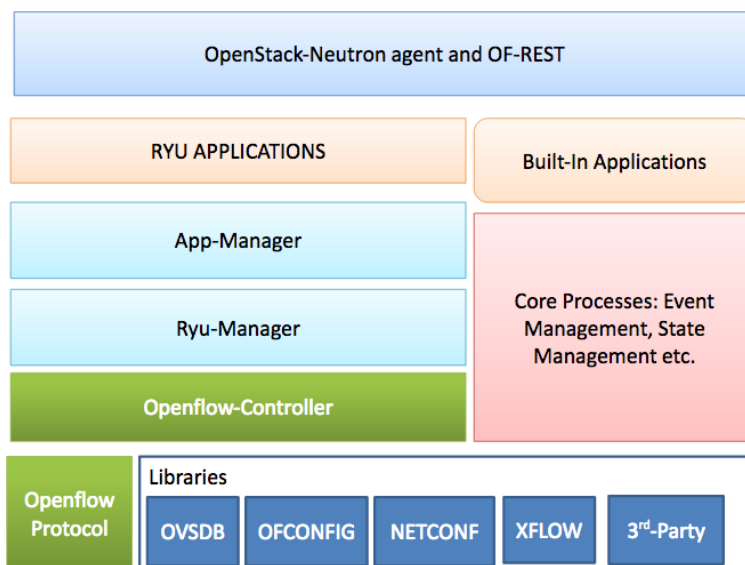


Illustrazione 12: Architettura Ryu

come i server si facciano carico delle richieste provenienti dai browser/client ed inoltrino le informazioni richieste alle relative applicazioni, oltre a come utilizzare le informazioni di cui si sono fatti carico e a come rispondere [23].

In Ryu, integrando il file `wsgi.py` nella propria applicazione, è possibile usufruire di componenti già disponibili come `ControllerBase` e `WSGIApplication`.

Il primo è un semplice modello di controller compatibile con le REst API e WSGI, mentre il secondo offre i servizi di un'applicazione WSGI. Nello scrivere il proprio controller, è sufficiente includere quindi anche componenti provenienti dall'applicazione `wsgi.py`.

In definitiva i file che sono stati scritti per la realizzazione di tutti i comportamenti previsti dai requisiti sono i seguenti:

- 1) `my_fileserver.py`: per la gestione del caricamento sei servizi web quali pagine HTML e script in Javascript;
- 2) `tap.py`: per l'aggiunta di regole di flow all'interno dei vari switch della rete;
- 3) `live.py`: per la gestione dell'intero comportamento descritto nel quarto punto dei requisiti ovvero la gestione in tempo reale delle regole di flow;

Oltre a questi, sono stati scritti anche i rispettivi file per la gestione delle REst API, tutti i file per la parte web del progetto (pagine HTML e script Javascript) e lo script di avvio `run_web_gui.sh`.

4.2 Caricamento dei moduli web del controller

È noto dai requisiti che il progetto del controller richiede l'avvio di un'applicazione WSGI per la creazione e l'utilizzo di REst API. Il file `my_fileserver.py` realizza principalmente questo comportamento.

Il modulo è diviso in due classi: `WebController` e `WebRestApi`. La prima si occupa dell'avvio effettivo del core principale del controller web, mentre la seconda gestisce le REst API, al fine di mappare istruzioni precise a determinati URL.

Nell'init di `WebController` si definisce un campo interno alla classe: la cartella in cui andare a cercare tutte le risorse web.

```
def __init__(self, req, link, data, **config):
    super(WebController, self).__init__(req, link, data, **config)
    self.directory =
os.path.join(os.path.dirname(os.path.abspath(__file__)), 'web')
```

Python memorizza nella variabile `__file__` il percorso del modulo attualmente considerato (in questo caso `my_fileserver.py`) e attraverso essa è possibile determinare la cartella dove sono caricate tutte le risorse web.

Infatti, il metodo `join` costruisce un percorso prendendo quello del modulo considerato e concatenandolo con la parola “web”.

Ora è quindi noto che il controller andrà a cercare nella cartella “web” tutte le risorse richieste per il caricamento, sapendo che essa è posta nello stesso percorso del file `my_fileserver.py`.

La classe definisce poi tre metodi essenziali nello sviluppo del controller e della sua parte di applicazione web:

- 1) `make_response`: serve a creare l'effettiva risposta alla richiesta di caricamento di una risorsa web e quindi di un certo tipo di file. Per creare una risposta POST, occorre inizialmente essere a conoscenza del tipo di file (MIME type). Se viene trovato attraverso l'utilizzo di un metodo appropriato (`mimetypes.guess_type(filename)`), tale risultato viene memorizzato in una variabile apposita, altrimenti si procede ad indicare come tipo di file quello più generico possibile, ovvero uno stream di ottetti (file binario).
Si procede quindi a riempire il body della risposta leggendo il file considerato. Viene restituita la risposta pronta (che sarà tradotta in una POST HTTP).
- 2) `get_file`: serve a realizzare la vera e propria apertura del file richiesto dal controller. Semplicemente scatena il metodo descritto precedentemente in caso di file attualmente esistente nel percorso considerato. Nel caso tale file non sia presente, verrà creata una risposta HTTP di errore (precisamente di status 400, ovvero richiesta non valida).
- 3) `get_root`: si collega al metodo `get_file`. Infatti quest'ultimo dà in output la pagina `index.html`, ovvero la homepage, se come nome di file viene specificato un campo nullo.

La seconda classe invece è incaricata di mappare i metodi sopra descritti a URL ben definiti, in modo da richiamare quei comportamenti attraverso degli indirizzi web.

Solo l'init è definito: appena la classe viene istanziata, si procede con il mapping.

Il mapper è preso direttamente dal modulo WSGI di Ryu e attraverso il metodo connect di cui dispone, è possibile esprimere quale controller gestirà le richieste e quale azione dovrà associare a quale URL (si utilizzano ovviamente i metodi definiti nella classe WebController).

```
mapper.connect('web', '/web/{filename:.*)',
               controller=WebController, action='get_file',
               conditions=dict(method=['GET']))

mapper.connect('web', '/',
               controller=WebController, action='get_root',
               conditions=dict(method=['GET']))
```

In definitiva, l'applicazione considerata avvierà il core del controller web e i metodi utili all'interfacciamento con il server WSGI: quando l'utente digiterà nella barra degli indirizzi l'URL corrispondente alla homepage del controller (di default localhost:8080), scatterà infatti la chiamata REst associata al metodo get_root.

Allo stesso modo, ogni volta che interagirà con elementi delle varie pagine web di cui è costituito il controller, egli scatterà metodi GET e i successivi metodi POST per il caricamento dei contenuti di una pagina (collegati al metodo get_file).

Si ricorda che tutto ciò è possibile grazie all'ambiente delle REst API fornito dal framework di Ryu. Come si è visto, è possibile crearne di nuove, ma è anche utile utilizzarne alcune già definite (come si farà successivamente).

```
instantiating app /home/ubuntu/ryu/ryu/app/WebGUI/my_fileserver of WebRestApi
instantiating app ryu.app.rest_topology of TopologyAPI
instantiating app ryu.app.ofctl_rest.py of RestStatsApi
instantiating app /home/ubuntu/ryu/ryu/app/WebGUI/tap_rest of TapRestApi
instantiating app ryu.topology.switches of Switches
instantiating app /home/ubuntu/ryu/ryu/app/WebGUI/live_rest of LiveRestApi
(23670) wsgi starting up on http://0.0.0.0:8080/
(23670) accepted ('127.0.0.1', 55424)
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/index.html HTTP/1.1" 200 5140 0.228222
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/css/pure-min-0.5.0.css HTTP/1.1" 200 19086 0.001040
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/css/font-awesome.min.css HTTP/1.1" 200 20858 0.000808
(23670) accepted ('127.0.0.1', 55425)
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/css/pure-custom.css HTTP/1.1" 200 19106 0.000863
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/css/joint.all.min.css HTTP/1.1" 200 23869 0.000720
(23670) accepted ('127.0.0.1', 55426)
127.0.0.1 - - [19/Feb/2017 09:33:18] "GET /web/js/jquery.min.js HTTP/1.1" 200 96522 0.001329
(23670) accepted ('127.0.0.1', 55427)
(23670) accepted ('127.0.0.1', 55428)
(23670) accepted ('127.0.0.1', 55429)
127.0.0.1 - - [19/Feb/2017 09:33:19] "GET /web/js/tap.js HTTP/1.1" 200 10495 0.002368
127.0.0.1 - - [19/Feb/2017 09:33:19] "GET /web/js/utills.js HTTP/1.1" 200 1234 0.003519
127.0.0.1 - - [19/Feb/2017 09:33:19] "GET /web/img/controller.png HTTP/1.1" 200 1408 0.002385
127.0.0.1 - - [19/Feb/2017 09:33:19] "GET /v1.0/topology/switches HTTP/1.1" 200 148 0.001020
```

Illustrazione 13: Output del controller all'avvio

Come si nota dall'illustrazione precedente, appena il controller viene avviato, anche il web server (WSGI) parte. Una volta istanziata l'applicazione my_fileserver, è possibile richiedere il caricamento delle risorse web. Appena l'utente si reca nella homepage, una nuova richiesta viene accettata e l'applicazione sopra descritta comincia a caricare gli elementi necessari alla pagina richiesta.

Come si nota dall'output verboso del controller, attraverso le chiamate REst vengono scatenate delle GET che a loro volta garantiscono all'utente la corretta visualizzazione della pagina richiesta, ovviamente con tutti gli elementi di cui fa parte (file Javascript, css, ecc...).

4.3 Aggiunta manuale di regole di flow

La pagina index.html, ovvero la homepage del server web per la gestione del controller, nonché pagina utilizzata dall'utente per aggiungere flow entries, si presenta in questo modo:

Illustrazione 14: Homepage del controller: index.html

Attraverso questa pagina si può realizzare il comportamento descritto dal secondo punto dei requisiti: permettere l'inserimento da parte dell'utente di regole di traffico attraverso la modifica delle flow table degli switch.

Inoltre viene realizzato qua anche il comportamento descritto dal primo punto dei requisiti, ovvero il fatto che il controller debba riconoscere precisamente tutte le entità presenti sulla rete.

Per gestire tutti i comportamenti dinamici e gli eventi generati lato client dall'utente al click su determinati elementi, la pagina è gestita da script Javascript con alcuni elementi di JQuery.

L'azione fondamentale da realizzare in primis è sicuramente riempire dinamicamente la lista di switch tra cui l'utente può scegliere per inserire regole.

Ciò infatti deve essere fatto considerando l'attuale topologia della rete: la lista va riempita in base agli switch attualmente esistenti e collegati sulla rete.

Per realizzare ciò, appena la pagina viene caricata grazie all'applicazione my_fileserver.py, vengono caricati anche i file Javascript da cui essa dipende.

Tra questi sono presenti jquery.min.js, utils.js e tap.js. Il primo è semplicemente una libreria di JQuery, il secondo presenta pochi metodi di utilità ricorrenti e il terzo è il vero core della pagina index.html.

Appena la pagina viene caricata, scatta l'evento dinamico updateSwitchList:

```
function updateSwitchList() {
    var switchSelect = document.getElementById("switch");
    $.getJSON(url.concat("/v1.0/topology/switches"), function(switches) {
        $.each(switches, function(index, value) {
            var el = document.createElement("option");
            el.textContent = value.dpid;
            el.value = value.dpid;
        });
    });
}
```

```

        switchSelect.appendChild(el);
        portMap[value.dpid] = value.ports;
    });
    }).then(updatePorts);}

```

Inizialmente, dopo aver considerato la select degli switch presente nel DOM della pagina, attraverso la funzione `getJSON` ci si interfaccia con una chiamata REst per la selezione dinamica degli switch sulla rete. Con questo metodo infatti, prendendo un URL specifico (`/v1.0/topology/switches`) si attiva una chiamata REst di default descritta nel file `ofctl_rest.py`.

Questo modulo fa parte di un'applicazione Ryu già disponibile allo sviluppatore che restituisce statistiche su tutti i componenti attualmente presenti sulla rete. Si parla dell'app `TopologyDiscovery`. Tramite il metodo `getJSON`, compreso nella libreria JQuery, si crea perciò una lista di switch da inserire nella select adibita, nonché un dizionario per associare tutte le porte ad un determinato switch.

Se il metodo ha successo, occorre riempire gli elementi del DOM strettamente dipendenti dalla select contenente gli switch: infatti quando l'utente clicca su un determinato switch della select, occorre riempire la lista di porte di entrata e di uscita su cui registrare la regola di flow.

Per questo interviene immediatamente `UpdatePorts` che permette di visualizzare le porte associate allo switch correntemente selezionato tramite il recupero delle informazioni necessarie attraverso la struttura dati creata da `UpdateSwitchList`.

La regola da inserire deve perciò essere impostata scegliendo obbligatoriamente su quale switch installarla, da quale porta e verso quale porta sia autorizzato a passare il traffico. Se una di queste impostazioni non viene selezionata dall'utente, si procede a dare errore.

Sono facoltativi invece i campi più specifici con cui filtrare il traffico: si possono infatti determinare precisamente gli host di andata e di ritorno, filtrandoli tramite indirizzo IP o MAC. Infine l'utente può scegliere se far passare solo una certa tipologia di traffico, a seconda del protocollo.

Per comunicare al controller la volontà di inserire un nuovo comportamento in un certo switch, e quindi per scatenare una flow mod a livello di OpenFlow, si utilizza ancora una volta l'ambiente delle REst API. Nel file `tap_rest.py` sono infatti definiti dei comportamenti collegati alla ricezione di una POST HTTP, per la creazione e l'eliminazione di una regola decisa dall'utente.

Perciò, una volta impostati i parametri della regola di flow da far inserire al controller nello switch scelto, occorre creare una POST HTTP a cui il controller sarà collegato tramite REst.

Una volta premuti i bottoni Set o Clear, verranno scatenati eventi legati all'aggiunta o alla rimozione della regola e perciò vengono definiti nel file Javascript due metodi per realizzare questi comportamenti: `setTap` e `clearTap`.

Il corpo della richiesta POST contenente tutti i parametri di modifica viene realizzato in entrambi i casi grazie alla funzione `makePostData`. Si decide di impostare il corpo della richiesta POST con la struttura dati seguente (definita in pseudocodice):

```

var tapInfo =
{
  'fields': {
    'dl_src': indirizzo sorgente ethernet (stringa),
    'dl_dst': indirizzo destinazione ethernet (stringa),
    'dl_type': tipo di traffico (intero),
    'dl_vlan': VLAN ID (intero),
    'nw_src': indirizzo sorgente ip (stringa),
    'nw_dst': indirizzo destinazione ip (stringa),
    'nw_proto': protocollo (intero),
    'tp_src': porta sorgente (intero),
    'tp_dst': porta destinazione (intero)},
  'sources': lista di {'dpid': datapath id (intero), 'port_no': numero
    porta (intero)},
  'sinks': lista di {'dpid': datapath id (intero), 'port_no': numero
    porta (intero)},
}

```

Si tratta di una definizione dei campi da modificare molto prossima alla sintassi con cui tali modifiche vengono impostate effettivamente attraverso OpenFlow.

La `makePostData` perciò raccoglie le info dal DOM della pagina al fine di creare una struttura dati come quella descritta sopra. Qua vengono anche convertite le semplici stringhe di scelta del tipo di traffico (come per esempio ARP, TCP, UDP, ecc..) in definizioni precise di OpenFlow, sfruttando alcune combinazioni dei campi descritti nella struttura dati `tapInfo` [24]:

```
if (trafficType == 'ARP') {
    tapInfo.fields['dl_type'] = 0x806;
}
// Set prerequisite of IPv4 for all other types
else if (trafficType == 'ICMP') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['nw_proto'] = 1;

} else if (trafficType == 'TCP') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['nw_proto'] = 6;
}
else if (trafficType == 'HTTP') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['nw_proto'] = 6;
    tapInfo.fields['tp_port'] = 80;
}
else if (trafficType == 'HTTPS') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['tp_port'] = 443;
    tapInfo.fields['nw_proto'] = 6;
}
else if (trafficType == 'UDP') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['nw_proto'] = 0x11;
}
else if (trafficType == 'DNS') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['tp_port'] = 53;
    tapInfo.fields['nw_proto'] = 0x11;
} else if (trafficType == 'DHCP') {
    tapInfo.fields['dl_type'] = 0x800;
    tapInfo.fields['tp_port'] = 67;
    tapInfo.fields['nw_proto'] = 0x11;
}
```

Una volta impostata correttamente la struttura dati, è possibile avviare la chiamata REst appropriata per l'aggiunta o la rimozione della regola specificata attraverso proprio l'utilizzo di `tapInfo`.

Nel caso di una aggiunta, il codice è il seguente (molto simile per la rimozione):

```
$.post(url.concat("/v1.0/tap/create"), JSON.stringify(tapInfo),
function() {
    }, "json")
    .done(function() {
        originalMain = $('#main').clone();
        $('#post-status').html('');
```

```
$('#main').html('<h2>Tap created</h2><p>Successfully created tap.
Check the <a href="/web/stats.html#flow">flow statistics</a> to verify
that the rules have been created.</p><button class="pure-button pure-
button-primary" onclick="restoreMain()">Create another tap</button>');
})
.fail(function() {
    $('#post-status').html('<p style="color:red;
background:silver;">Error: Tap creation failed. Please verify your
input.</p>');
});
```

Come descritto dal codice, in caso di fallimento verrà restituito un errore, mentre in caso di successo la pagina HTML cambierà totalmente e in modo dinamico restituendo questa visualizzazione:
Discorso analogo per l'eliminazione di una regola:

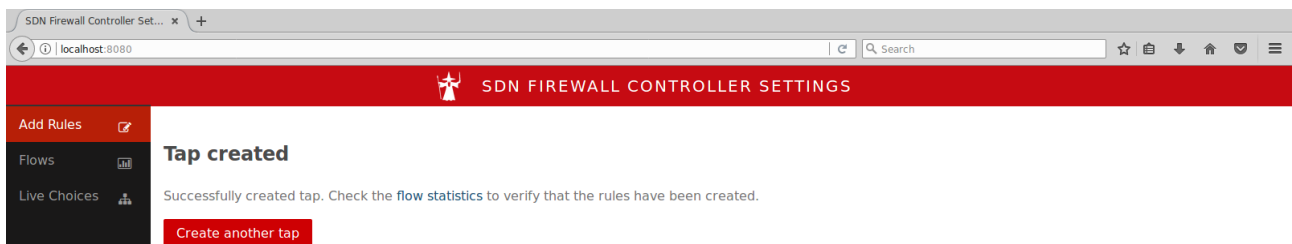


Illustrazione 15: Regola aggiunta correttamente

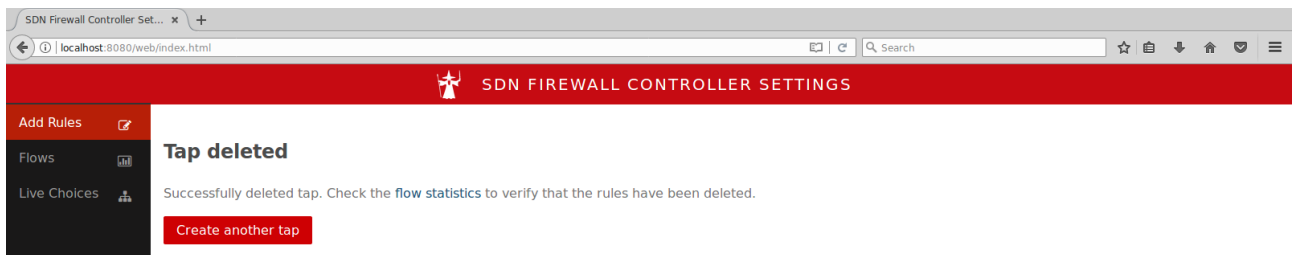


Illustrazione 16: Regola rimossa correttamente

Quindi, quando una regola viene rimossa o aggiunta correttamente, significa che il controller avrà eseguito un certo metodo raccogliendo tutti i dati dalla richiesta POST inviata tramite chiamata REst. Perciò ora occorre analizzare il modulo del controller adibito all'aggiunta di regole OpenFlow sugli switch scelti: tap.py.

Inizialmente viene definita una matrice necessaria a memorizzare le informazioni degli host a seconda della loro tipologia:

```
self.broadened_field = {'dl_host': ['dl_src', 'dl_dst'],
                        'nw_host': ['nw_src', 'nw_dst'],
                        'tp_port': ['tp_src', 'tp_dst']}
```

Tale matrice associa ad un host ethernet l'indirizzo sorgente e destinazione ethernet, ad un host IP l'indirizzo sorgente e destinazione IP e ad un host TCP l'indirizzo sorgente e destinazione TCP.

Il resto del codice è principalmente composto dai metodi agganciati alle REst API di cui si discuteva in precedenza.

A seconda della richiesta, il metodo create_tap o il metodo delete_tap, vengono eseguiti.

Queste funzioni, nella prima parte hanno un comportamento analogo: nella struttura dati broadened_field si tiene traccia di tutte le info sugli host da includere nell'inserimento o nell'eliminazione delle regole.

Quindi inizialmente si modificano tutte le chiavi generiche giunte tramite POST HTTP con le chiavi specifiche presenti in broadened_fields.

Fatto questo, i due metodi hanno comportamenti diversi in modo da aggiungere o eliminare regole. Per quanto riguarda `create_tap`, per ogni porta sorgente e per ogni porta destinazione di uno specifico switch specificato nel form della pagina html, si procede ad inviare un messaggio di OpenFlow Mod dal controller allo switch considerato:

```
ofproto = datapath.ofproto
ofproto_parser = datapath.ofproto_parser
in_port = source['port_no']
out_port = sink['port_no']
```

La struttura dati seguente è quella specificata dall'utente tramite form:

```
filter_fields = filter_data['fields'].copy()
```

Viene creata la lista di azioni:

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

Viene creato il vero e proprio match da inserire:

```
if in_port != 'all':
    filter_fields['in_port'] = in_port
match = ofctl_v1_3.to_match(datapath, filter_fields)
```

Viene generato un cookie, ovvero un identificatore per il messaggio di flow mod:

```
cookie = random.randint(0, 0xffffffffffffffff)
inst = [ofproto_parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
```

Viene installata poi la regola nello switch. Notare i campi `idle_timeout` e `hard_timeout`: il primo imposta il tempo idle di attesa prima di scartare il messaggio e il secondo imposta il tempo massimo di attesa prima di scartare il messaggio.

```
mod = ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,
instructions=inst, cookie=cookie)
```

Infine, il messaggio di modifica viene inviato:

```
datapath.send_msg(mod)
```

Analogo è il discorso per la rimozione di una regola: cambia solamente l'azione all'interno del messaggio di FlowMod, che ora sarà quello di delete:

```
mod = ofproto_parser.OFPFlowMod(datapath=datapath,
command=ofproto.OFPFC_DELETE, match=match, out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY)
```

L'ultima cosa da notare nel codice di questo modulo è la gestione degli errori. OpenFlow prevede che gli errori che si verificano, possano essere gestiti tramite appositi handler. Questo si concretizza in Ryu con la sua logica ad eventi: quando un errore si verifica, esso viene gestito da un'apposita sezione di codice delimitata da un certo decoratore che descrive l'evento atteso.

In questo caso, per esempio, si attende l'evento di un qualsiasi errore che si può verificare in qualsiasi step della comunicazione tra controller e switch tramite OpenFlow. L'handler seguente notifica semplicemente l'errore:

```
@set_ev_cls(ofp_event.EventOFPErrormsg, [HANDSHAKE_DISPATCHER,
CONFIG_DISPATCHER, MAIN_DISPATCHER])
def error_msg_handler(self, ev):
    msg = ev.msg
    LOG.info('OFPErrormsg received: type=0x%02x code=0x%02x
message=%s', msg.type, msg.code, ryu.utils.hex_array(msg.data))
```

4.4 Visualizzazione delle statistiche della rete

L'applicazione descritta in precedenza si occupa quindi di aggiungere regole di flow inserite manualmente dall'utente. Normalmente per visualizzare lo stato della rete, inteso come informazioni su tutti gli switch connessi, è necessario andare ad interrogare ogni singolo switch.

Tramite REst API questo problema può essere ovviato ed è possibile ottenere una visione centralizzata di tutte le statistiche della rete.

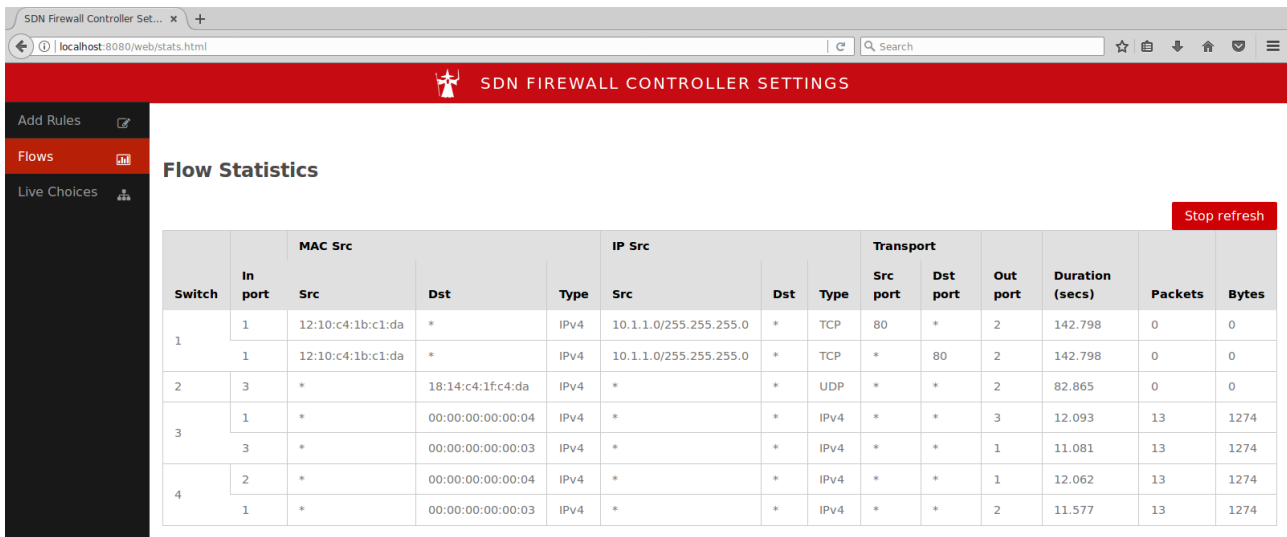
Utili allo scopo sono due API di default: quella che manda una GET all'URL `/stats/switches` e quella che manda una GET all'URL `/stats/flow/`. La prima restituisce la lista di tutti gli switch connessi alla rete e la seconda le flow entries installate su uno specifico componente.

Questi due comportamenti, una volta concatenati, possono darci tutte le informazioni di tutti gli switch connessi all'attuale topologia di rete.

Perciò non occorre definire nessun nuovo modulo del controller in quanto ne esiste già uno di default che definisce queste due API: `ofctl_rest.py`.

Tramite queste, è possibile realizzare il comportamento richiesto dal terzo punto dei requisiti: visualizzare tutte le statistiche per le flow degli switch presenti sulla rete.

La pagina `stats.html`, incaricata di eseguire questi comportamenti, si presenta in questo modo:



Switch	In port	MAC Src			IP Src			Transport			Duration (secs)	Packets	Bytes
		Src	Dst	Type	Src	Dst	Type	Src port	Dst port	Out port			
1	1	12:10:c4:1b:c1:da	*	IPv4	10.1.1.0/255.255.255.0	*	TCP	80	*	2	142.798	0	0
	1	12:10:c4:1b:c1:da	*	IPv4	10.1.1.0/255.255.255.0	*	TCP	*	80	2	142.798	0	0
2	3	*	18:14:c4:1f:c4:da	IPv4	*	*	UDP	*	*	2	82.865	0	0
3	1	*	00:00:00:00:00:04	IPv4	*	*	IPv4	*	*	3	12.093	13	1274
	3	*	00:00:00:00:00:03	IPv4	*	*	IPv4	*	*	1	11.081	13	1274
4	2	*	00:00:00:00:00:04	IPv4	*	*	IPv4	*	*	1	12.062	13	1274
	1	*	00:00:00:00:00:03	IPv4	*	*	IPv4	*	*	2	11.577	13	1274

Illustrazione 17: Visualizzazione delle flow entries installate sugli switch: stats.html

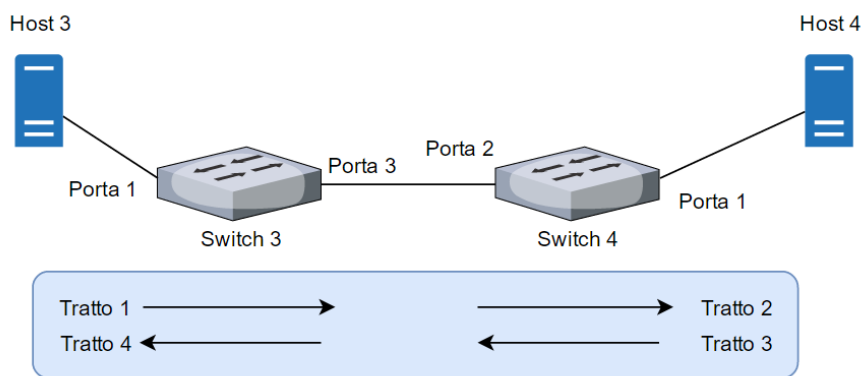
Questa pagina ha il compito di riassumere visivamente le regole installate tramite la pagina `index.html` e tramite la pagina `live.html` (descritta in seguito).

Per esempio, nell'immagine appena sopra è possibile visualizzare alcune regole aggiunte in precedenza. Sul primo switch sono installate due regole per la gestione del traffico HTTP (notare il tipo TCP e le porte 80 sia di destinazione che sorgente che rappresentano questo protocollo) diretto a qualunque host, entrante sulla porta 1 e uscente dalla porta 2 dello switch.

Inoltre l'host sorgente è anche descritto da un indirizzo IP (oltre che ad un indirizzo MAC).

Similmente la seconda regola, dove però si filtra traffico UDP entrante sulla porta 3 e diretto alla porta 2 dello switch.

Infine le quattro regole successive (due per ogni switch) rappresentano le regole che occorre



impostare nella rete per far fluire traffico da un eventuale host connesso allo switch 3 ad un eventuale host connesso allo switch 4. Sono presenti quattro regole perché i tratti di percorso sono effettivamente quattro (come è possibile notare dall'illustrazione a lato).

Illustrazione 18: Tratti di percorso tra due host collegati a due switch differenti

Inoltre sono presenti anche statistiche sui secondi trascorsi dall'inserimento di una specifica regola nonché sui pacchetti e sui byte scambiati tramite quella flow entry.

Affinché la pagina mostri delle statistiche veramente reali, essa viene aggiornata automaticamente ogni 5 secondi, in modo osservare se nel mentre ci sono state modifiche alle flow tables degli switch. È anche presente un pulsante per bloccare questo comportamento (e cioè fermare il refresh automatico).

Come detto sopra, non occorre realizzare nessun comportamento nel controller Ryu in sé, poichè la pagina è totalmente gestita solamente da uno script Javascript.

Tale script (flow-stats.js), esegue ogni cinque secondi la funzione `updateFlowStats`.

Essa, attraverso le REST API specificate sopra, per ogni switch trovato nella rete popola la tabella principale della pagina con tutte le informazioni lette nelle varie flow entries di ciascuno switch.

Se l'utente preme il pulsante per fermare il refresh automatico, semplicemente il metodo `stopFlowStatsTableRefresh` bloccherà il comportamento che realizza il refresh della pagina ogni cinque secondi (attraverso `clearInterval` [25]).

4.5 Inserimento di regole in tempo reale

I primi tre comportamenti descritti dal documento dei requisiti sono stati realizzati. Ora manca al controller una pagina da cui l'utente possa accettare in tempo reale i pacchetti che chiedono di transitare nella rete verso un host destinatario.

Le difficoltà nel realizzare questo comportamento sono principalmente due:

- 1) Creare un'applicazione Ryu in grado di gestire pacchetti in tempo reale facendo scegliere all'utente se autorizzare o meno il passaggio. Questo comporta il dover trovare un modo di raccogliere e gestire gli eventi di `packetIn` (come spiegato nel sotto capitolo 2.3);
- 2) Trovare il modo di creare in Ryu dei comportamenti simili agli intent di ONOS. Per esempio, prendendo come riferimento l'illustrazione 18, il controller dovrà essere in grado, una volta autorizzato dall'utente il tratto 1, di autorizzare automaticamente il resto delle tratte (tratta 2, 3 e 4) senza chiederne il consenso all'utente (avrebbe poco senso). Questo deve essere generalizzato anche per un percorso finale che attraversa n switch e diverse topologie di rete (lineari, triangolari, ecc...).

La pagina `live.html`, incaricata di eseguire questi comportamenti, si presenta in questo modo:

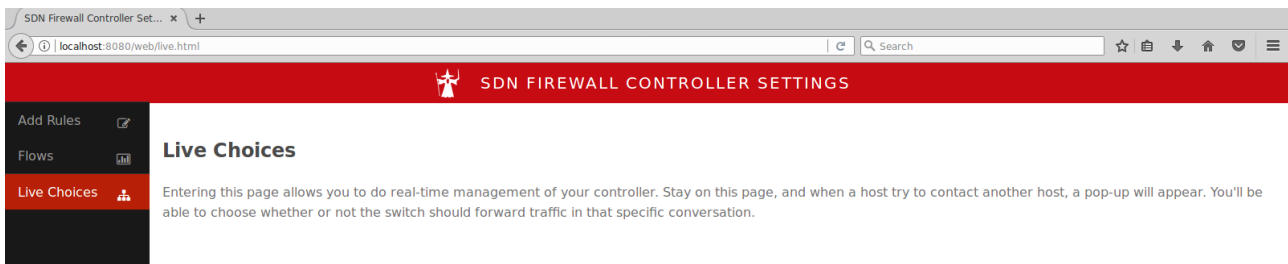


Illustrazione 19: Autorizzazione del traffico in tempo reale: `live.html`

La strategia adottata prevede che questa pagina si aggiorni automaticamente ogni mezzo secondo per notificare l'eventuale presenza di nuovi pacchetti da autorizzare che di default vengono mandati tramite OpenFlow al controller (evento di `packetIn`).

Attraverso uno script di Javascript, ci si interfaccia con il controller: in questo caso il file `live.js` prevede svariati metodi con cui interagire con Ryu. Come al solito si sceglie di farlo utilizzando l'ambiente delle REST API.

Infatti, nel file `live_rest.py` vengono definite tre nuove chiamate REST con lo stesso pattern visto per il file `my_fileserver.py`: prima si scrive una classe dove vengono definiti dei metodi che poi verranno implementati nel file `live.py` (vero core di questa parte di applicazione), e poi se ne scrive un'altra strettamente legata all'istanza del server WSGI di Ryu, al fine di mappare i metodi con URL ben definiti.

A seguire, un estratto di `live_rest.py` che rende chiaro il comportamento sopra descritto (viene riportato solo il metodo `accept` e il suo mapping con le REST API):

```
class LiveController(ControllerBase):
    [...]
    def accept(self, req, **kwargs):
        self.live_app.accept()
    [...]

class LiveRestApi(live.Live):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION,
                    ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        'wsgi': WSGIApplication,
    }

    def __init__(self, *args, **kwargs):
        super(LiveRestApi, self).__init__(*args, **kwargs)
        wsgi = kwargs['wsgi']
        wsgi.register(LiveController,
                      {live_instance_name: self})
        mapper = wsgi.mapper

        mapper.connect('live', '/rest/accept',
                      controller=LiveController, action='accept',
                      conditions=dict(method=['GET']))
    [...]
```

In definitiva, l'interazione tra utente e controller è gestita tramite il file Javascript `live.js` che si interfaccia ad esso tramite chiamate REST.

L'idea alla base della progettazione di questa parte di codice è che tramite una chiamata REst apposita (quella che risponde all'URL `rest/communications`) si possano raccogliere tutte le richieste di `packetIn` registrate dal controller, in modo da farle autorizzare o meno dall'utente tramite altre due REst (collegate agli URL `/rest/deny` e `/rest/accept`).

Il primo passo quindi è quello di intercettare i messaggi di `packetIn` sul controller Ryu.

Nel file `live.py` è definito un handler per la gestione di eventi di questo genere, segnalato da un apposito decoratore:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
```

Ogni qualvolta si verificano questi eventi, vengono inizialmente raccolte tutte le informazioni sul pacchetto entrante nel controller tramite i metodi di accesso forniti da Ryu.

Tale parte di codice è qua riportata perché è ricorrente in questo modulo:

```
in_port = msg.match['in_port'] #su quale porta è arrivato il pacchetto?
datapath = msg.datapath #informazioni sul datapath dello switch
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
dpid = datapath.id #quale switch? restituisce l'id (es: 1, 2 ecc)
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
src = eth.src #indirizzo ethernet sorgente (= mac address)
dst = eth.dst #indirizzo ethernet destinazione (= mac address)
```

Successivamente, si memorizzano in una struttura dati apposita del controller (`mac_to_port`) informazioni su indirizzi degli switch e porte associate. Questo si fa per evitare flood successivi: infatti nel caso in cui occorresse inviare un pacchetto verso una certa destinazione, lo switch semplicemente invierebbe tale pacchetto su tutte le porte di uscita al fine di trovarne la destinazione. Con la struttura dati qui considerata si evita questo problema perché i dispositivi vengono associati alle porte.

Per cui la prima volta si farà flood, ma dalla seconda in poi sarà nota la porta su cui il pacchetto dovrà uscire per raggiungere la destinazione e perciò si eviterà di inviare i pacchetti su tutte le porte.

Inizialmente quindi occorre aggiungere lo switch nella struttura dati assieme al riferimento alla sua porta di ingresso:

```
self.mac_to_port.setdefault(dpid, {})
self.mac_to_port[dpid][src] = in_port
```

Successivamente si deve trovare nella tabella l'indirizzo MAC di destinazione: se associato allo switch esiste un campo destinazione, si estrae l'informazione sulla porta da cui i pacchetti devono uscire, altrimenti per forza la porta di uscita sarà un flood e perciò il pacchetto verrà inviato su tutte le porte:

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

Dopo aver analizzato il pacchetto e dopo aver memorizzato il legame tra switch e porte, occorre procedere trovando un'altra informazione: il protocollo con cui il pacchetto vorrebbe transitare sulla rete.

Per fare ciò si utilizza un metodo di ausilio che analizza le informazioni del pacchetto in maniera simile a quanto fatto nel file Javascript tap.js (descritto nel sotto capitolo 4.3): ogni combinazione di porta e numero di protocollo rappresenta infatti un certo tipo di traffico [26].

Si sceglie di descrivere per semplicità solamente pacchetti ipv4:

```
def getProtocol(self, pkt):
    pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
    tp = pkt.get_protocol(tcp.tcp)
    port = 0
    if tp:
        port = tp.dst_port
    ud = pkt.get_protocol(udp.udp)
    if ud:
        port = ud.dst_port
    #print "PORTA: %s" % port
    if pkt_ipv4:
        protocol = pkt_ipv4.proto
        if protocol==1:
            return "ICMP"
        if protocol==6:
            if port==80:
                return "HTTP"
            if port==443:
                return "HTTPS"
            return "TCP"
        if protocol==17:
            if port==53:
                return "DNS"
            if port==67:
                return "DHCP"
            return "UDP"
    return "Unknown. If you confirm, you will add a general traffic rule
(= every type of traffic) between src and dst"
```

L'algoritmo utilizzato prevede che le informazioni sui nuovi pacchetti (intese come coppia di indirizzi sorgente e destinazione e tipo di protocollo) vengano inserite in una lista (communications) solamente nei casi in cui:

- 1) Non abbiano legami con la storia passata dei pacchetti. Infatti, tramite una variabile (story) si tiene traccia dei collegamenti che occorre fare tra i vari host. Una volta che questi collegamenti vengono accettati o rifiutati dall'utente, la loro entry viene eliminata dalla struttura dati. È una variabile utile al fine di non accumulare richieste di packetIn identiche. Infatti, se arrivassero 100 packetIn e non esistesse questa struttura dati, tutti e 100 i collegamenti verrebbero scritti in communications e perciò l'utente finale dovrebbe autorizzare 100 volte il pacchetto (e non ce ne sarebbe bisogno perché una volta autorizzato il primo pacchetto, non si verificherebbero più eventi di packetIn simili, in quanto i pacchetti passerebbero normalmente tra gli switch grazie alla regola impostata precedentemente);
- 2) Non riguardino particolari tipologie di traffico: per semplicità infatti si richiede che alcune tipologie di traffico non vengano autorizzate manualmente dall'utente.

In effetti alcuni protocolli generano molto traffico sulla rete verso svariati indirizzi MAC a loro adibiti, e perciò ha poco senso che l'utente finale se ne occupi. Esiste per esempio il protocollo ARP [27]: esso esegue la mappatura tra indirizzi IP e indirizzi MAC dei vari componenti connessi, generando parecchio traffico all'avvio della rete.

Si può anche considerare il Link Layer Discovery Protocol che è utilizzato da ogni dispositivo di rete per segnalare la propria identità, per trovare i dispositivi vicini e per informare delle proprie funzionalità nell'ambito di una LAN [28]. Questi ed altri protocolli largamente utilizzati al fine di gestire correttamente utilità e configurazioni iniziali della rete, sono filtrati attraverso il metodo `filtered_ip`: con esso si autorizza traffico inerente a tutti i mac address adibiti al LLDP, al multicast ipv4 [29] e al multicast ipv6 [30] nonché richieste ARP e di broadcast. Non si inseriscono regole negli switch, ma semplicemente tutti i pacchetti che seguono questi protocolli vengono mandati a destinazione tramite `packetOut`. Quindi, se si vuole aggiungere o eliminare protocolli automaticamente autorizzati, occorre semplicemente modificare `filtered_ip`.

Nel caso le due condizioni appena descritte siano verificate, le informazioni su indirizzo sorgente, indirizzo di destinazione e protocollo utilizzato vengono aggiunte nella stringa `communications`. Se sulla rete invece sta passando traffico inerente a protocolli filtrati con `filtered_ip`, essi vengono subito spediti a destinazione tramite messaggi di `packetOut` previsti da `OpenFlow`:

```
if self.filtered_ip(dst, eth) == False:
    data = None

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

Nel codice appena sopra, si nota la creazione di un messaggio di `packetOut` dal controller allo switch destinatario, con tutti i parametri invariati rispetto a quelli inerenti al `packetIn` giunto precedentemente al controller.

Perciò ora avremo una lista di collegamenti da effettuare intercettati dal controller e salvati in una variabile (`communications`). È necessario quindi notificare l'utente di queste richieste e vedere se accettarle o meno a seconda delle sue scelte.

Per fare ciò, il file Javascript `live.js` prevede il metodo `getFirstCommunication`. Tramite esso si interroga una REst API collegata al metodo `listCommunications` di `live.py` (tale binding è effettuato nel file `live_rest.py` come specificato precedentemente) che permette di accedere ai collegamenti in coda nel controller:

```
function getFirstCommunication() {
    $.get(url.concat("rest/communications"))
```

Questa REst, tramite POST, restituirà in modo ordinato e una alla volta, delle stringhe contenenti le informazioni principali del pacchetto: indirizzo sorgente, indirizzo destinazione e protocollo (esempio di stringa: 00:00:00:00:00:01 00:00:00:00:00:02 ICMP).

Tramite questa informazione, occorre comporre il messaggio da inviare all'utente:

```
.done(function(data) {
    if (data)
    {
        if(data!="done")
        {
            src = data.split('\n')[0].split(' ')[0];
            dst = data.split('\n')[0].split(' ')[1];
```

```
proto = data.split('\n')[0].split(' ')[2];
if(proto=="Unknown.")
    proto = data.split('\n')[0].substring(36)
```

Notare un caso particolare: se il protocollo trovato dal controller è sconosciuto (perché non previsto) la parola Unknown verrà registrata nella variabile proto e l'utente visualizzerà il seguente messaggio: *Unknown. If you confirm, you will add a general traffic rule (= every type of traffic) between src and dst*

In qualsiasi caso comunque verrà avviato il metodo choose function:

```
        chooseFunction();
    }
    else
    {
        console.log("Requested page is empty");
    }
})
.fail(function(data) {
    console.log("Failed sending request");
});
```

Tale metodo scatena una delle due REst API per accettare o rifiutare il passaggio di traffico, a seconda delle scelte dell'utente (allowCommunication e denyCommunication semplicemente si agganciano agli URL delle REst API per far scattare i metodi associati definiti nel controller):

```
function chooseFunction() {
    if (confirm(createConfirmText(src, dst,proto)) == true)
    {
        allowCommunication(src, dst);
    }
    else
    {
        denyCommunication(src, dst);
    }
}
```

Rimane perciò da considerare come effettivamente il controller accetti o rifiuti le richieste e soprattutto come fare in modo che abbia un comportamento algoritmicamente intelligente.

Come si diceva precedentemente (riferimento all'illustrazione 18), è fondamentale che se il traffico per arrivare a destinazione deve attraversare 100 host, non vengano notificati all'utente 100 nuovi eventi di packetIn.

Si vuole un comportamento simile all'intent di ONOS: una volta che un tratto di percorso verso destinazione viene autorizzato, i tratti restanti dovranno essere automaticamente autorizzati, senza nessuna decisione da parte dell'utente (ha già autorizzato una volta quel tipo particolare di traffico, non ha senso che lo faccia altre volte).

Con riferimento alla topologia presentata nell'illustrazione 18, occorre analizzare con precisione il comportamento che deve gestire il controller.

Inizialmente la rete si presenta come mostrato nell'illustrazione successiva. Esistono quattro tratti di percorso per andare dall'Host 3 all'Host 4, due per ogni direzione all'interno di un singolo switch. Nella fase iniziale essi sono tutti non autorizzati dal controller:

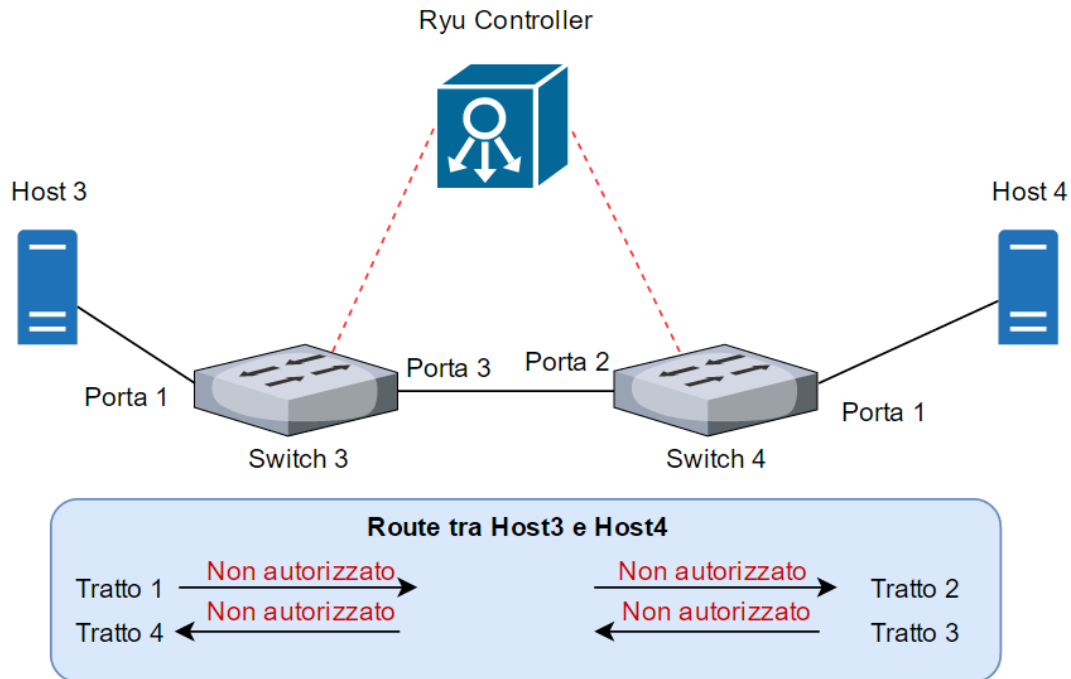


Illustrazione 20: Situazione iniziale

Se Host 3 comincia a spedire dei pacchetti diretti a Host 4 (tramite per esempio un semplice ping), il controller deve prevedere che siccome sugli switch non ci sono regole su come gestire e dove mandare tali pacchetti, essi debbano essere mandati interamente al controller tramite messaggi di packetIn di OpenFlow. Tale traffico deve essere autorizzato dall'utente, perché si tratta di una comunicazione

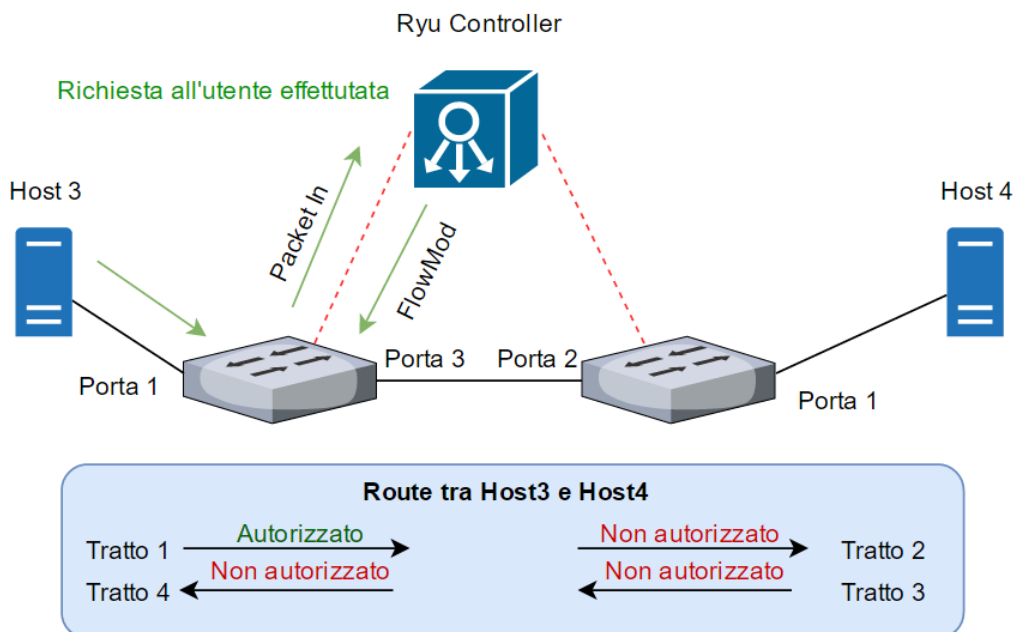


Illustrazione 21: Host 3 inizia ad inviare pacchetti diretti a Host 4

nuova tra i due host, con un certo protocollo e mai registrata prima d'ora. Si ammette che l'utente accetti il traffico. Una volta autorizzato, verrà impostata una regola di passaggio per il primo tratto:

Nel mentre si scateneranno altri messaggi di packetIn diretti al controller per quanto riguarda il secondo tratto. Però, siccome il primo tratto del percorso è già stato autorizzato, nessuna richiesta di autorizzazione viene mandata all'utente e il secondo tratto viene autorizzato automaticamente dal controller:

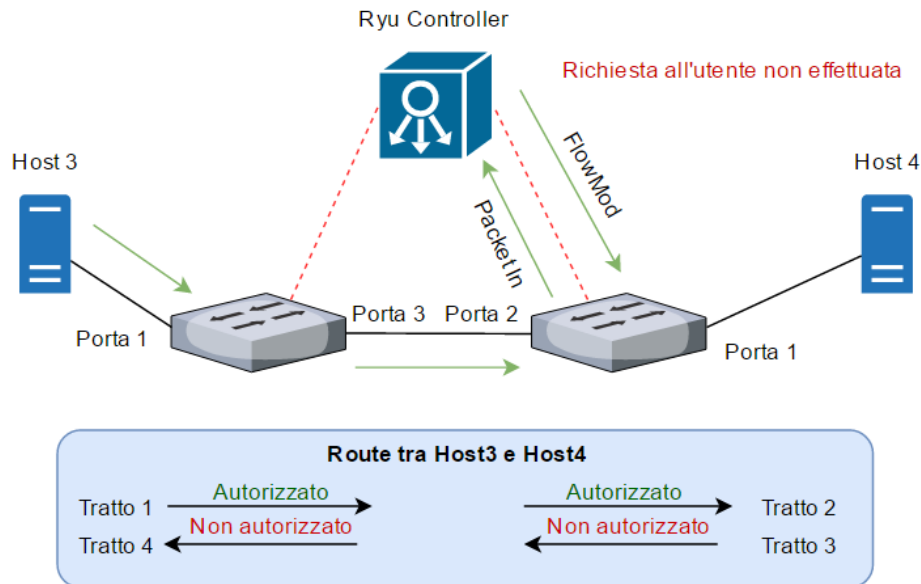


Illustrazione 22: Traffico osservato anche nella seconda parte del percorso

Similmente si comporta il traffico di ritorno. Infatti esistono già tratti autorizzati di quel percorso nel verso contrario, perciò nessuna richiesta di autorizzazione è posta all'utente:

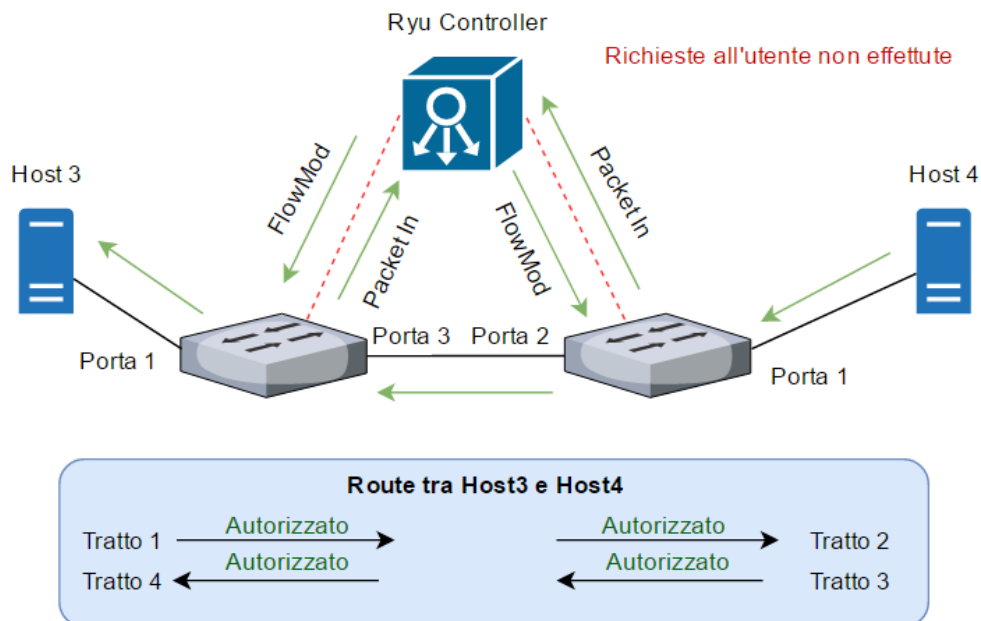


Illustrazione 23: Traffico di ritorno

Infine nell'illustrazione successiva si descrive il normale passaggio dei pacchetti una volta autorizzati tutti i tratti (solo il primo direttamente dall'utente).

Nessuna richiesta viene fatta al controller perché ormai tutte le regole di flow sono già state installate negli switch di rete:

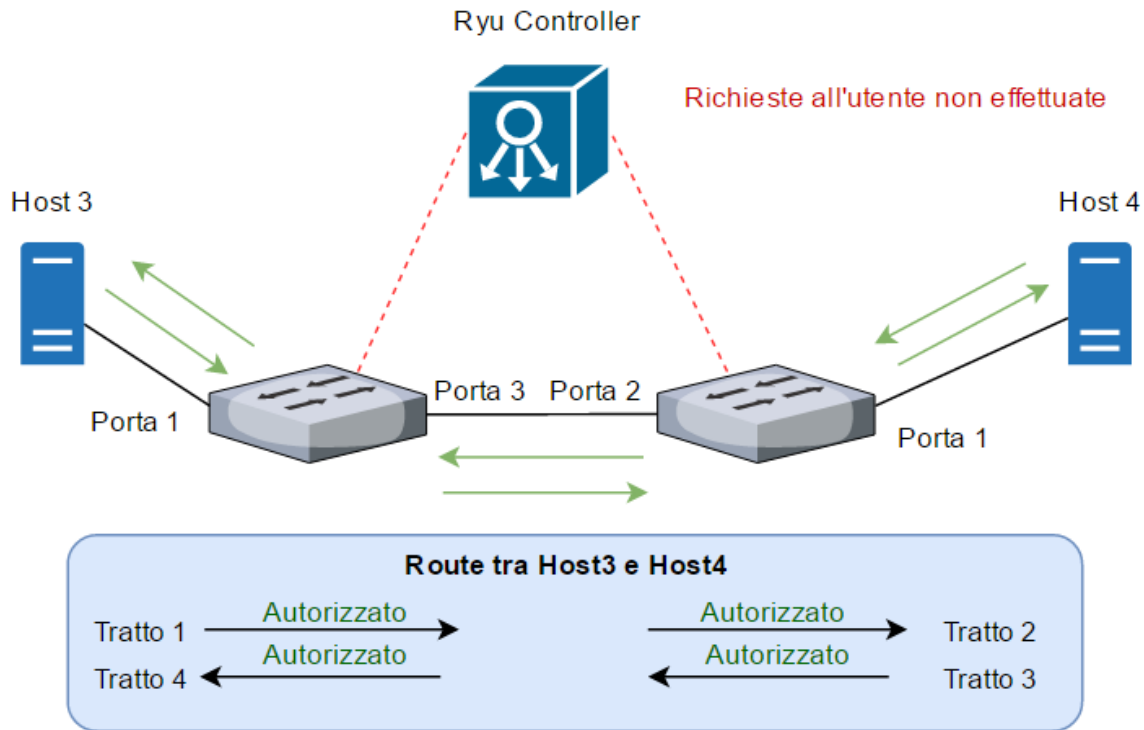


Illustrazione 24: Situazione della rete una volta installate tutte le regole

Considerando ora il codice vero e proprio, si affida al metodo `list_communications`, definito nel file `live.py` del controller, l'incarico di rispondere alle richieste inviate dal metodo `getFirstCommunication`.

L'algoritmo del metodo verifica che la generica stringa costituita da indirizzo sorgente, indirizzo di destinazione e protocollo, sia comparsa effettivamente per la prima volta.

Questa stringa, utilizzando un paragone con il mondo di SQL, rappresenta una chiave. Infatti nell'applicazione non possono esistere due o più stringhe (definite come sopra) che hanno indirizzo sorgente identico, indirizzo di destinazione identico e protocollo identico.

Il funzionamento è molto semplice: se all'avvio il controller riceve una certa richiesta di `packetIn` da uno switch, tale richiesta viene inserita nella lista `currentRoutes`. Questa variabile è una lista di tutti i tratti di percorso attualmente già gestiti dal controller (o che comunque stanno per essere gestiti). Ogni volta che arriva un `packetIn` al controller, esso viene confrontato con questa lista: se una entry esiste già, significa che l'utente ha già autorizzato tale traffico.

Si sta perciò considerando un tratto intermedio di percorso della rete, che deve essere autorizzato senza interazione con l'utente (che ha già autorizzato il tratto precedente). In caso contrario, si procede con la normale richiesta all'utente.

Inoltre `currentRoutes` tiene traccia di entrambi i versi di percorrenza.

Per esempio nel caso di un semplice ping da host 1 a host 2, il traffico dovrà transitare in un verso (diretto a destinazione), ma anche nell'altro (traffico di risposta). Perciò `currentRoutes` prevede che una chiave come ad esempio `00:00:00:00:01 00:00:00:00:02 ICMP` sia perfettamente identica a `00:00:00:00:02 00:00:00:00:01 ICMP` (perché si tratta del traffico di ritorno). Questo comportamento è sviluppato grazie al metodo `check`.

Di seguito è riportato il codice dell'algoritmo principale del modulo e cioè quello contenuto in `list_communications`:

```
def list_communications(self):
    actual = self.communications
    self.communications
    self.communications[self.communications.find('\n') + 1:]
    if(actual!=''):
        if self.check(actual[:actual.find('\n')]) == True:
            return "done";
        else:
            self.currentroutes.append(actual[:actual.find('\n')])
            return actual
```

Innanzitutto si registra la comunicazione da considerare attualmente attraverso la variabile `actual`, eliminandola dalla lista delle comunicazioni (poiché verrà trattata ora).

Se esiste un tratto di percorso che richiede l'autorizzazione, si procede a verificare che non sia appartenente ad una route già impostata precedentemente e lo si fa attraverso il metodo `check`.

Se questo comportamento è verificato, si restituisce la stringa `done` al file Javascript in modo che non richieda all'utente l'autorizzazione per quel pezzo di percorso (il metodo `getFirstCommunication` contenuto in `live.js` semplicemente non farà apparire il dialog e la richiesta sarà accettata in modo trasparente dal controller).

In caso contrario, si procede a restituire la entry, in modo che lo script `live.js` chieda l'autorizzazione all'utente per il passaggio del traffico tra gli host specificati e con il protocollo indicato (come descritto precedentemente, queste informazioni sono estrapolate direttamente dai pacchetti in arrivo).

Di seguito è riportato il metodo (`check`) con il quale verificare se il pacchetto in entrata è nuovo o riferito a connessioni già registrate:

```
def check(self, to_find):
    add = to_find.split( )
    case1 = "%s %s %s" % (add[0], add[1], add[2])
    case2 = "%s %s %s" % (add[1], add[0], add[2])
    return (case1 in self.currentroutes or case2 in self.currentroutes)
```

Con le operazioni riportate si creano due stringhe con il primo e il secondo campo scambiati rispettivamente: in questo modo si può dire che per esempio `00:00:00:00:00:02 00:00:00:00:00:01 ICMP` e `00:00:00:00:00:01 00:00:00:00:00:02 ICMP` sono perfettamente identici e se almeno uno dei due viene trovato nella lista `currentRoutes`, allora non occorre richiedere l'interazione con l'utente.

Una volta inserito correttamente la route in `currentRoutes`, si richiede perciò all'utente tramite `getFirstCommunication`, metodo contenuto in `live.js`, se accettare o meno tale percorso, e si realizza ciò attraverso una normalissima finestra di dialogo di Javascript. La pagina web perciò si presenterà così:

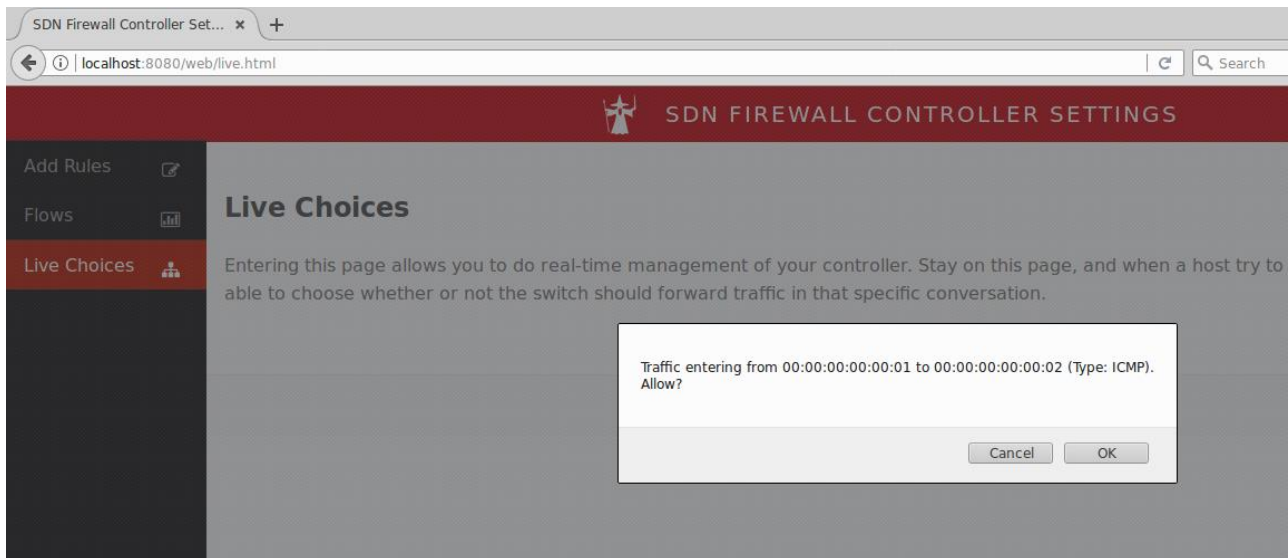


Illustrazione 25: Finestra di dialogo per autorizzare o meno il traffico

L'utente ora può effettuare due scelte: se decide di autorizzare il traffico, tramite chiamata REst verrà scatenato il metodo `accept` del controller, in caso contrario `deny` servirà la richiesta.

Questi due metodi perciò sono gli unici (oltre al `packetOut` per i tipi di traffico consentiti di default) a modificare i comportamenti degli switch sulla rete. Infatti manderanno messaggi di `flow mod` a livello di OpenFlow per modificare entries nelle flow tables degli switch.

Il metodo `accept`, dopo aver raccolto tutte le informazioni sul pacchetto di cui autorizzare il traffico (raccolto nella variabile `messages`), provvede a trovare quale sia la porta di uscita dello switch su cui mandare il traffico. Viene utilizzato a questo scopo la struttura dati `mac_to_port`: in modo simile a quanto già affermato precedentemente, se la destinazione è nota viene trovato anche il riferimento alla porta dello switch, altrimenti la porta verrà impostata a `ofproto.OFPP_FLOOD` (il pacchetto perciò verrà mandato su tutte le porte):

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

Il match da effettuare (e cioè le informazioni sulla regola da inserire) è creato dal metodo `getMatch`. Esso, tramite le informazioni carpite dal pacchetto, imposta una regola appropriata tra i due host a seconda del tipo di traffico e quindi del protocollo con cui essi si scambiano dati.

Questo metodo è molto simile al già considerato `getProtocol`: a seconda del numero di protocollo e della porta di un pacchetto ipv4, si restituisce un match appropriato. Di seguito un esempio di match che regola traffico HTTP:

```
if protocol==6:
    if port==80:
        return parser.OFPMatch(in_port=in_port,
                                eth_dst=dst, eth_type=0x0800, ip_proto=6, tcp_dst=80)
```

L'unica azione che deve eseguire il controller Ryu, è quella di mandare il pacchetto sulla porta di uscita trovata al passo precedente con l'istruzione di applicarla immediatamente:

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

Successivamente si effettua per completezza una distinzione: un pacchetto gestito da un controller che utilizza OpenFlow, può essere mandato interamente al controller oppure bufferizzato sullo switch dove è entrato. Nel secondo caso perciò il messaggio di mod dovrà prevedere anche un riferimento a quale sia effettivamente tale buffer. A livello OpenFlow è utilizzata a questo scopo la variabile `buffer_id`:

```
if self.messages[0].buffer_id:
    mod = parser.OFPFlowMod(datapath=datapath,
                            buffer_id=self.messages[0].buffer_id,
                            priority=1, match=match,
                            instructions=inst)
else:
    mod = parser.OFPFlowMod(datapath=datapath, priority=1,
                            match=match, instructions=inst)
```

Infine il messaggio di flow mod può essere finalmente inviato. Inoltre, deve essere rimossa la entry in story che tiene traccia dei collegamenti attuali da effettuare e anche il pacchetto viene eliminato dalla lista dei messaggi arrivati al controller:

```
datapath.send_msg(mod)
if key in self.story:
    self.story.remove(key)
self.messages.pop(0)
```

L'operazione di deny è l'esatto contrario della accept, ma con alcune particolarità. Innanzitutto il match creato darà informazioni solo sulla porta di ingresso e l'host destinazione:

```
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
```

Inoltre la mod da inserire avrà tutti i campi esattamente identici a quelli della mod di accept a parte il campo delle istruzioni (che in questo caso non viene inserito):

```
mod = parser.OFPFlowMod(datapath=datapath,
                        buffer_id=self.messages[0].buffer_id, priority=1, match=match)
```

Infatti, non specificare alcuna istruzione viene tradotto nella logica di OpenFlow come una regola di drop: tutto ciò che è diretto verso l'host con un certo indirizzo MAC e che è entrato su una certa porta dello switch verrà quindi eliminato.

Per cui verranno ignorate anche richieste successive dallo stesso host, con la stessa destinazione e con lo stesso protocollo, in quanto verrà considerata la regola di drop già impostata.

L'unico modo per variare questo comportamento è quello di inserire manualmente la regola appropriata tramite l'utilizzo della pagina `index.html`.

Rimane perciò da considerare solamente il comportamento a default del controller e cioè cosa succede inizialmente quando uno switch riceve un nuovo pacchetto che deve essere inviato a destinazione ma per il quale non ha regole installate.

Questo comportamento è descritto dal metodo `switch_features_handler` che come `packet_in_handler` gestisce un certo evento: qua si considera la prima fase del protocollo OpenFlow (vedere illustrazione 6):

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
```

Inizialmente si elimina qualsiasi flow entry dallo switch considerato. È un comportamento utile al fine di resettare completamente il controller e gli switch di rete ad esso connessi:

```
mod = parser.OFPFlowMod(datapath=datapath, command=ofproto.OFPFC_DELETE,
    out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
    datapath.send_msg(mod)
```

Infine si installa la cosiddetta table-miss flow entry. Si tratta di una entry particolare che viene scritta negli switch al fine di gestire i comportamenti iniziali di tali componente.

Quando uno switch nuovo viene installato sulla rete, esso non presenta nessuna regola di default per gestire i pacchetti (non sa proprio come mandarli a destinazione). Tramite la table-miss flow entry, si decide che tutti i pacchetti di cui non è previsto un percorso da nessuna regola vengano mandati al controller interamente (non si prevedono buffer all'interno degli switch come specificato dalla keyword `OFPCML_NO_BUFFER`). Il codice che regola questo comportamento è il seguente:

```
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
    ofproto.OFPCML_NO_BUFFER)]
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
    actions)]
mod = parser.OFPFlowMod(datapath=datapath, priority=0, match=match,
    instructions=inst)
datapath.send_msg(mod)
```

Per concludere si può affermare che un modulo così costruito, riesce a gestire perfettamente il traffico in tempo reale. Gli eventi di `packetIn` vengono infatti raccolti dal controller e segnalati all'utente immediatamente (più precisamente ogni mezzo secondo). In seguito, l'utente ha la facoltà di scegliere se accettare o meno il traffico, e lo farà solo una volta per lo stesso tipo di collegamento e protocollo (le regole dei tratti successivi vengono impostate automaticamente).

In base alle sue scelte, il controller regolerà gli switch tramite OpenFlow al fine di installare regole di passaggio di traffico o di eliminazione di tutti i pacchetti futuri identici a quello non autorizzato dall'utente.

È opportuno infine notare che il browser deve rimanere aperto sulla pagina `web.live.html` in modo da gestire il traffico in tempo reale attraverso i metodi considerati in questo sotto capitolo.

4.6 Script per l'effettivo avvio del controller

Una volta sviluppati tutti i moduli di cui si compone il controller al fine di realizzare le specifiche previste da requisiti, è necessario scrivere uno script per avviare l'applicazione completa.

Il file `run_web_gui.sh` realizza questa funzione:

```
~/ryu/bin/ryu-manager --observe-links ~/ryu/ryu/app/WebGUI/my_fileserver
~/ryu/ryu/app/WebGUI/tap_rest ~/ryu/ryu/app/WebGUI/live_rest
ryu.app.rest_topology ryu.app.ofctl_rest.py
```

Il manager di Ryu, che come detto sopra ha il compito di caricare i moduli di un'applicazione, provvede ad avviare tutto:

- 1) `my_fileserver`: modulo (considerato nel sotto capitolo 4.2) che provvede a caricare i moduli web del server WSGI;
- 2) `tap_rest.py`: modulo (considerato nel sotto capitolo 4.3) per l'aggiunta manuale di regole. Si tratta del file che gestisce le REst API (e che ha anche accesso diretto all'implementazione reale di esse nel file `tap.py`);
- 3) `live_rest.py`: modulo (considerato nel sotto capitolo 4.4) per l'inserimento in tempo reale di regole di traffico. Si tratta del file che gestisce le REst API (e che ha anche accesso diretto all'implementazione reale di esse nel file `live.py`);
- 4) `ofctl_rest.py`: modulo compreso di default nel framework di Ryu utile per REst API già definite e utilizzate dal prototipo nella pagina `stats.html` al fine di visualizzare statistiche di rete.
- 5) `rest_topology`: modulo compreso di default nel framework di Ryu utile per REst API già definite e utilizzate nel prototipo dalla pagina `index.html` per tracciare la topologia di rete.

Per avviare il controller perciò è sufficiente avviare tale script.

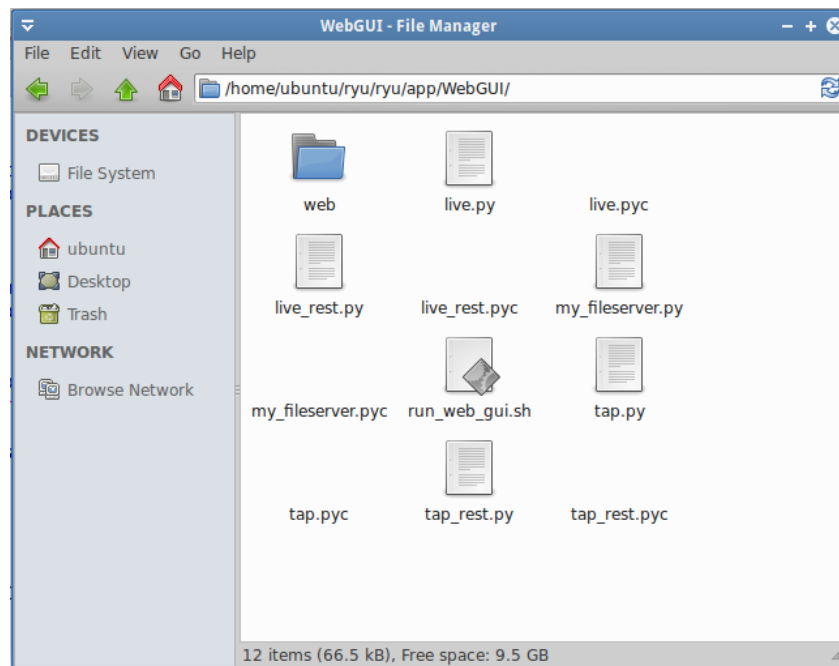


Illustrazione 26: Cartella con i file utilizzati dal controller Ryu

Capitolo 5: Testing del controller

5.1 L'ambiente di testing

Una volta realizzato il controller, è chiaro che occorre testarlo per verificarne il corretto funzionamento. È necessario sperimentare su diverse tipologie di rete per osservare come il controller si adatti ad esse e verificare che non ci siano casi particolari in cui tale prototipo possa incontrare difficoltà nella gestione corretta del traffico.

Per realizzare ciò, si sceglie di effettuare il testing in due diversi ambienti: una rete simulata e una rete reale.

La prima verrà sviluppata grazie agli strumenti descritti nel terzo capitolo, mentre la seconda verrà creata attraverso l'utilizzo di due macchine e un Raspberry Pi 2 Model B. Perciò è doveroso introdurre quest'ultimo.

Il Raspberry Pi 2 Model B [31] è un single-board computer (ovvero un calcolatore implementato su una scheda elettronica) sviluppato nel regno unito dalla Raspberry Pi Foundation e rilasciato in commercio nel febbraio 2012.



Illustrazione 27: Raspberry Pi 2 Model B

Questo è stato concepito come uno strumento economico (quindi alla portata di tutti) utile per stimolare l'insegnamento dell'informatica e delle basi di programmazione soprattutto nelle scuole. In effetti, fino ad ora ne sono state prodotte sette versioni e i loro prezzi sono veramente contenuti (da 5 a 35 dollari statunitensi).

La scheda è stata progettata per ospitare sistemi operativi basati sul kernel Linux o RISC OS.

Il dispositivo utilizzato è la

seconda board realizzata dall'azienda che prevede miglioramenti sotto il punto di vista hardware rispetto al primo modello rilasciato. La scheda integra un System-on-a-chip (SoC) Broadcom BCM2836 che è composto da un processore ARM Cortex-A7 32 bit con una frequenza di 900MHz, una GPU VideoCore IV e 1 Gigabyte di memoria SDRAM condivisa con la scheda video.

Per completezza se ne riporta la scheda tecnica:

- SoC – Broadcom BCM2836 quad core Cortex A7 processor @ 900MHz with VideoCore IV GPU
- System Memory – 1GB SDRAM (PoP)
- Storage – micro SD card slot (push release type)
- Video & Audio Output – HDMI and AV via 3.5mm jack.
- Connectivity – 10/100M Ethernet
- USB – 4x USB 2.0 ports, 1x micro USB for power
- Expansion: 2×20 pin header for GPIOs / Camera header / Display header
- Power – 5V via micro USB port.

Si tratta perciò di un calcolatore a tutti gli effetti e si presta benissimo al testing nella rete reale del prototipo realizzato.

5.2 La rete virtuale

Per creare una rete virtuale si utilizza principalmente Mininet. Questo strumento infatti realizza topologie di rete più o meno complesse (integrando Open vSwitch) con pochi e semplici comandi.

In seguito si considerano due reti virtualizzate attraverso Mininet.

La prima è una rete molto piccola costituita solamente da due switch in cascata connessi rispettivamente a due host:

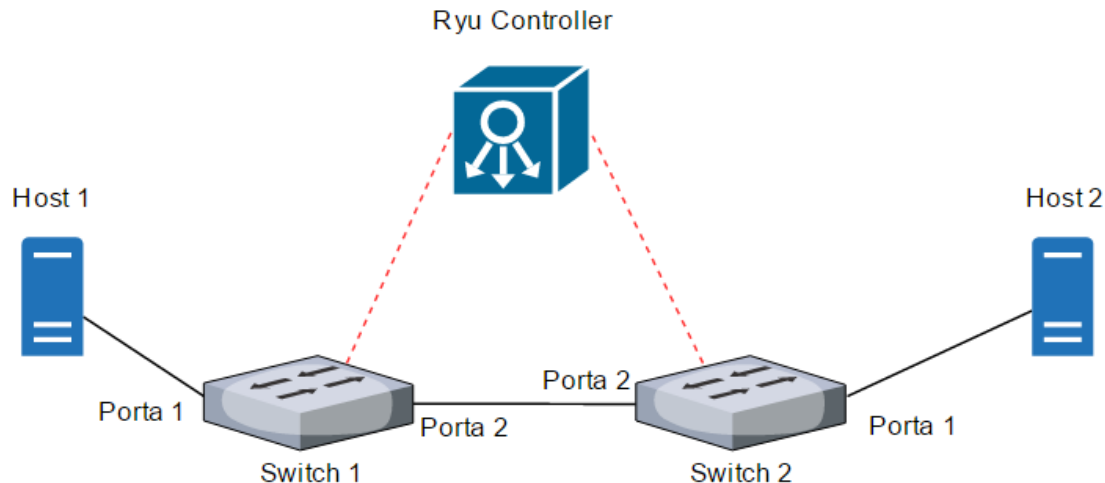


Illustrazione 28: Prima rete virtuale di test

È possibile simulare tali switch, tali host e i rispettivi collegamenti solamente utilizzando il comando seguente digitato a terminale [32]:

```
sudo mn --topo linear,2 --mac --controller remote --switch ovsk
```

Con esso si comunica a Mininet la volontà di creare una topologia lineare con due switch in cascata (--topo linear,2), con indirizzi mac semplici (--mac attribuisce id unici di facile lettura), con un controller remoto (--controller remote indica di non utilizzare il controller di default di Mininet ma uno remoto in ascolto sulla porta 6633 di localhost) e utilizzando switch simulati con Open vSwitch (--switch ovsk). Una volta avviato anche il controller su un terminale differente attraverso lo script visto in precedenza (nel sotto capitolo 4.6), è possibile avviare il test vero e proprio: un semplice ping tra i due host simulati per testare il passaggio di traffico ICMP.

```
ubuntu@sdnhubvm:~[08:05]$ sudo mn --topo linear,2 --mac --controller remote --switch ovsk
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s2) (s2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=5167 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.274 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.065 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.105 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=0.483 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=0.077 ms
```

Digitando h1 ping h2 nella CLI di Mininet si simula questo comportamento.

Inizialmente il ping non arriva a destinazione perché il comportamento del controller previsto dai requisiti è quello di bloccare tutto il traffico.

Tale richiesta di ping verrà notificata all'utente tramite la pagina live.html nella stessa maniera vista precedentemente (vedere illustrazione 25).

Illustrazione 29: Prova di ping con Mininet

Una volta autorizzata la regola, i pacchetti arrivano a destinazione e quindi il ping ha effettivamente successo. Come si può notare dall'illustrazione 29, il primo ping ha un tempo molto più elevato che dipende sostanzialmente da quanto velocemente l'utente accetta la regola dalla corrispondente pagina web. Si può ora procedere ad effettuare un ping al contrario: h2 ping h1.

Come ci si aspetta, il traffico passa senza nessuna richiesta di autorizzazione del traffico da parte dell'utente: infatti, il ping precedente ha impostato quattro regole per ciascuno dei tratti che compongono il percorso tra i due host (di andata e di ritorno).

Perciò il ping ora considerato farà passare dei pacchetti attraverso gli switch senza che essi chiedano niente al controller, limitandosi solamente a seguire le regole impostate nelle loro flow tables.

Le regole installate negli switch possono essere osservate nella pagina stats.html:

Switch	In port	MAC Src		Type	IP Src			Transport		Out port	Duration (secs)	Packets	Bytes
		Src	Dst		Src	Dst	Type	Src port	Dst port				
1	1	*	00:00:00:00:00:02	IPv4	*	*	IPv4	*	*	2	1178	1179	115542
	2	*	00:00:00:00:00:01	IPv4	*	*	IPv4	*	*	1	1176.983	1178	115444
2	2	*	00:00:00:00:00:02	IPv4	*	*	IPv4	*	*	1	1177.991	1179	115542
	1	*	00:00:00:00:00:01	IPv4	*	*	IPv4	*	*	2	1177.5	1179	115542

Illustrazione 30: Statistiche delle regole impostate sulla prima rete virtuale di test

Inoltre possono essere aggiunte manualmente regole di flow all'interno delle flow tables degli switch grazie alla pagina index.html, che grazie alle REst API di default per il ritrovamento della topologia utilizzata dalla rete, riesce a rilevare tutti gli switch presenti e le rispettive porte:

Add Rules

Following form allows sending FlowMod messages to a specific switch. You can send a FlowMod message with command Add using button Set, when you'll finish compiling this form. If you want to delete a previous inserted rule, then you can rewrite that rule's parameters in this form and click Clear. Otherwise, you can delete a bunch of rules previously inserted all at once by specifying only source and sink ports and clicking Clear: this will cause all previous rules inserted regarding those ports to be deleted.

Switch DPID:

Source port(s):

Sink port(s):

MAC address:

IP address:

Common traffic types:

Illustrazione 31: Inserimento manuale di regole nella prima rete virtuale di test

La seconda topologia prevede un numero più elevato di switch e una configurazione con dei loop:

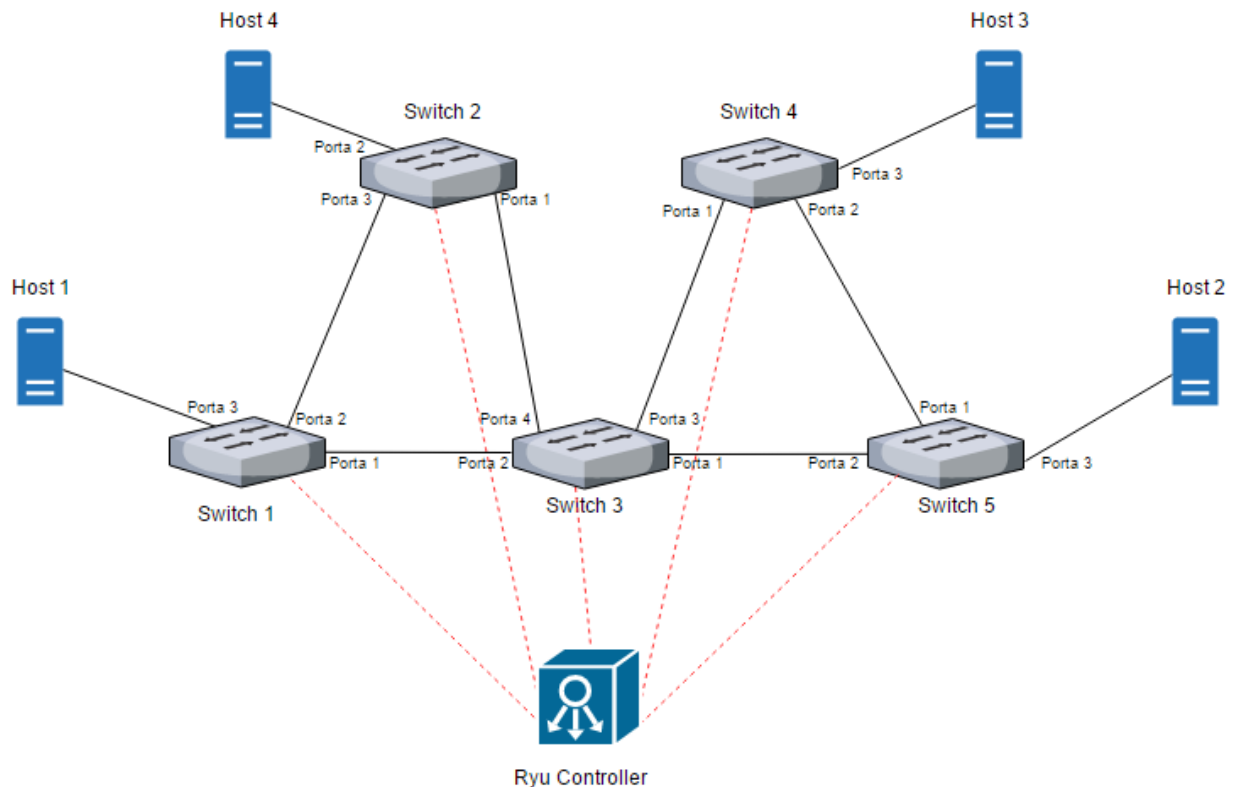


Illustrazione 32: Seconda rete virtuale di test

Più precisamente questa topologia è formata da cinque switch collegati a quattro host e sono presenti due sottoreti triangolari che formano dei loop. Per comunicare a Mininet la volontà di creare una rete di test così fatta occorre creare uno script apposito.

Il file `double_triangle.py` contiene essenzialmente la definizione di tutti i componenti voluti e i collegamenti da effettuare tra di essi. Mininet offre dei metodi molto intuitivi al fine di realizzare ciò:

```
h1 = self.addHost( 'h1' )
h2 = self.addHost( 'h2' )
h3 = self.addHost( 'h3' )
[...]
self.addLink( s1, s2)
self.addLink( s1, s3)
self.addLink( s2, s3)
self.addLink( s3, s4)
[...]
```

Successivamente occorre quindi avviare tale file tramite Mininet con il seguente comando (molto simile a quello della prima rete a parte lo switch `--custom`):

```
sudo mn --custom ~/mininet/custom/double_triangle.py --topo=mytopo --mac
--controller remote --switch ovsk
```

Si avvia perciò anche il controller, si fa partire un ping di prova tra due host qualunque della rete e si nota un comportamento particolare: la destinazione è irraggiungibile.

```

ubuntu@sdnhubvm:~[09:59]$ sudo mn --custom ~/mininet/custom/double_triangle.py --topo=mytopo --mac --controller remote --switch ovsk
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(s1, h1) (s1, s2) (s1, s3) (s2, h4) (s2, s3) (s3, s4) (s3, s5) (s4, h3) (s4, s5) (s5, h2)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable

```

Illustrazione 33: Destinazione irraggiungibile

Questo succede a causa della topologia della rete: infatti i pacchetti vengono mandati da un host all'altro e poi cominciano a vagare all'interno dei loop. Questo problema perciò non è causato da un comportamento non corretto del controller, ma dalla topologia stessa della rete. Si può porre rimedio utilizzando un algoritmo di spanning tree in modo da eliminare i loop della rete.

Lo spanning tree è un protocollo di comunicazione standard utilizzato per realizzare reti complesse (a Livello fisico) con percorsi ridondanti utilizzando tecnologie di Livello datalink (il livello 2 del modello OSI) come IEEE 802.2 o IEEE 802.11. Lo spanning tree viene eseguito dai bridge e dagli switch, e mantiene inattive alcune interfacce in modo da garantire che la rete rimanga connessa ma priva di loop [33]. Open vSwitch offre in modo semplice la possibilità di impostare tra gli switch degli algoritmi di spanning tree: alcune porte verranno chiuse in modo che il traffico sicuramente non potrà passarci attraverso e quindi così facendo, si eviteranno percorsi ridondanti.

Si ferma perciò il controller e da un altro terminale si digita il comando seguente per ogni switch presente sulla rete:

```
ovs-vsctl set Bridge [NOME SWITCH] stp_enable=true
```

Questi comandi modificheranno la topologia della rete nel modo seguente:

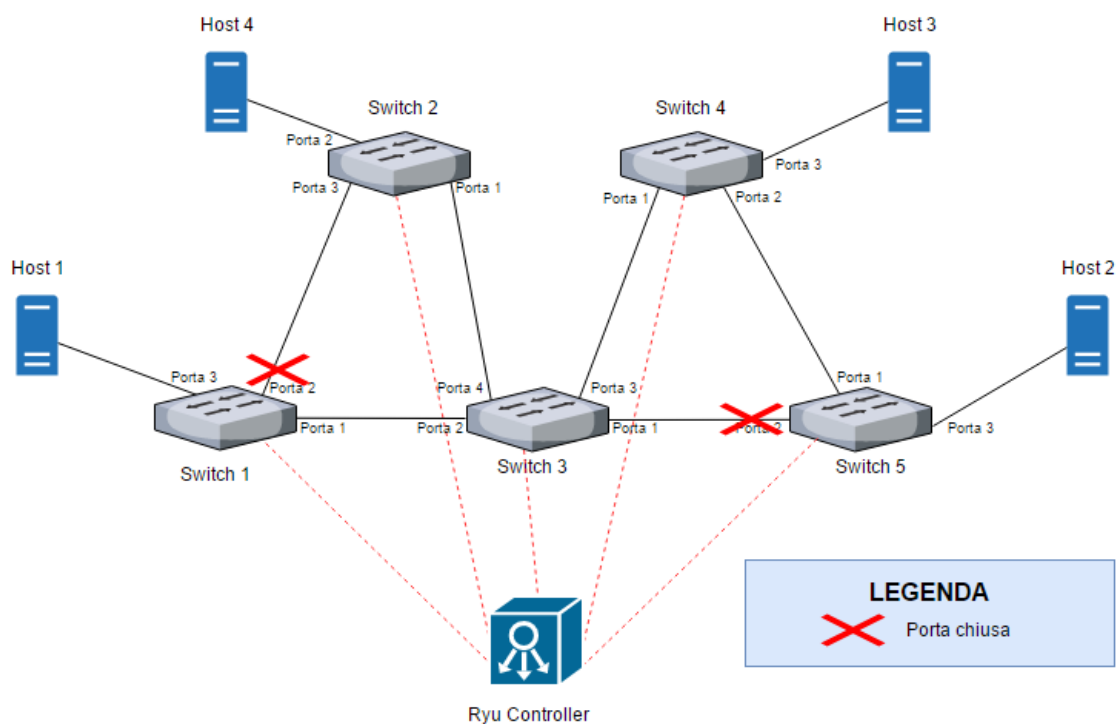


Illustrazione 34: Modifica alla seconda rete virtuale di test per evitare loop

Chiudendo le porte indicate nell'illustrazione, tutti i loop vengono eliminati perché alcuni tratti di percorso non sono più percorribili. Se si prova a riavviare il controller e da Mininet si riesegue il comando di ping, si nota infatti che ora i pacchetti transitano sulla rete e arrivano correttamente a destinazione.

Infine, similmente a quanto visto per la prima rete virtuale di test, è possibile aggiungere regole in modo manuale attraverso la pagina `index.html` e visualizzarle attraverso la pagina `stats.html` (vedere illustrazioni 30 e 31).

5.3 La rete reale

Per sviluppare la rete reale effettiva si utilizzano due computer portatili e un Raspberry Pi 2 (Model B). I calcolatori fungono da endpoint della rete, sui quali vengono simulati gli host, mentre il Raspberry è il contenitore effettivo del controller. Per quanto riguarda invece gli switch, essi vengono comunque simulati su ognuno dei tre componenti utilizzati. La topologia di test sarà perciò la seguente (i componenti sono configurati come descritto nel sotto capitolo 3.4):

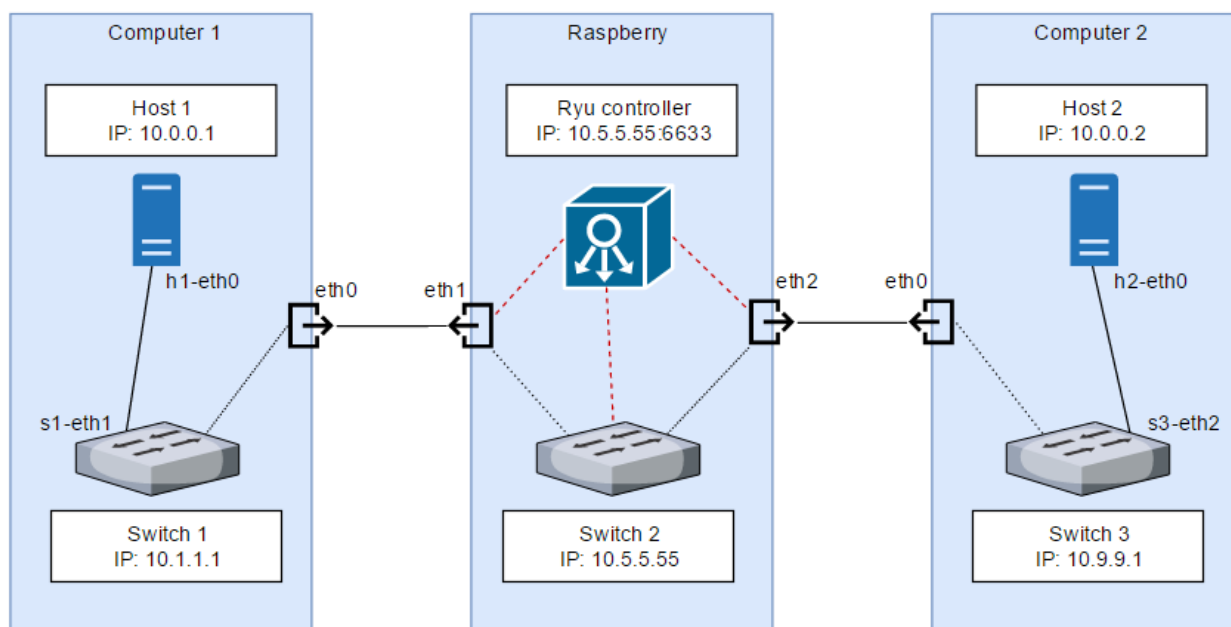


Illustrazione 35: Rete reale di test

Come si nota dall'illustrazione, il controller Ryu deve sapere riconoscere tutti e tre gli switch, anche se essi si trovano su macchine differenti. Inoltre, nella topologia vengono utilizzate sia interfacce reali che virtuali: gli switch Open vSwitch utilizzeranno infatti un'interfaccia virtuale per comunicare con gli host sulla stessa macchina e un'interfaccia reale per comunicare con i componenti presenti sulle altre macchine.

Per realizzare una topologia del genere, si sceglie di non utilizzare Mininet, ma di realizzare tre script per regolare i comportamenti voluti. Infatti, è più sensato che attraverso il semplice avvio di uno script da terminale si monti automaticamente la configurazione voluta per quella determinata macchina. Perciò si realizzano tre script, ciascuno eseguibile su una macchina differente.

Questi script non sono altro che un insieme di comandi che internamente utilizza anche Mininet. Rendendoli eseguibili tramite script, si ottiene un controllo più completo e veloce della configurazione della rete.

Si analizzano ora gli script di avvio `s1.sh` (da eseguire sul computer 1) e `s2.sh` (da eseguire sul Raspberry). Non serve analizzare lo script `s3.sh` (da eseguire sul computer 2) in quanto è praticamente speculare allo script `s1.sh`.

In quest'ultimo, dopo aver verificato che l'utente che sta eseguendo lo script sia root (infatti tutti i comandi devono essere eseguiti con questa identità) si procede a definire uno campo apposito

(occorre digitare `./s1.sh c` per avviare questa modalità) per la pulizia completa di tutte le modifiche effettuate dallo script:

```
if test "$1" == 'c'; then
    echo "Pulizia iniziata: "
    mn -c
    echo -n "Eliminazione namespace: "
    ip netns del h1
    exit 2
fi
```

Si tratta di un riferimento ad un comando di Mininet (`mn -c`) che elimina completamente le modifiche al sistema per simulare una certa tipologia. Inoltre, si procede ad eliminare il namespace `h1` (simulazione dell'host `h1`). Questa sezione di comandi è comune a tutti e tre gli script.

Successivamente si creano le impostazioni vere e proprie della rete.

Inizialmente si crea l'host vero e proprio, simulato con un semplice namespace all'interno della macchina e l'istanza dello switch Open vSwitch (tramite comando specifico di tale strumento):

```
ip netns add h1
ovs-vsctl add-br s1
```

Occorre poi collegarlo da una parte all'host e dall'altra all'interfaccia `eth0`.

Per la prima configurazione occorre creare una virtual interface attraverso il comando `ip link`. Essa sarà costituito da due interfacce endpoint che devono essere rispettivamente associate da una parte all'host e dall'altra allo switch:

```
ip link add h1-eth0 type veth peer name s1-eth1
ip link set h1-eth0 netns h1
ovs-vsctl add-port s1 s1-eth1
```

Occorre poi inglobare l'interfaccia `eth0` all'interno dello switch: in tal modo essa sarà adibita esclusivamente ad esso. Si fa ciò settando a 0 l'interfaccia `eth0` e utilizzando il comando Open vSwitch seguente:

```
ifconfig eth0 0.0.0.0
ovs-vsctl add-port s1 eth0
```

Infine si configurano gli indirizzi IP e le regole di routing interne al kernel di Linux. L'host `h1` avrà come indirizzo IP `10.0.0.1` e lo switch Open vSwitch avrà indirizzo IP `10.1.1.1` e farà capo ad un controller posto all'indirizzo IP `10.5.5.55:6633`. Perciò si impostano i percorsi di routing appropriati e si attivano le interfacce:

```
ovs-vsctl set-controller s1 tcp:10.5.5.55:6633
ifconfig s1-eth1 up
ifconfig s1-eth1 0
ifconfig s1 up
ip addr add 10.1.1.1/24 dev s1
ip route add 10.5.5.0/24 dev s1
```

Come si diceva, lo script `s3.sh` è speculare mentre `s2.sh` è comunque molto simile a `s1.sh`.

Infatti `s2.sh` avrà tutti i comandi molto simili a quelli considerati sopra: saranno presenti quelli per la configurazione delle interfacce `eth1` e `eth2` adibite esclusivamente allo switch e quelli per l'aggiunta dei vari indirizzi IP e dei vari percorsi di routing per raggiungere gli altri componenti della rete.

Si nota l'assenza però della configurazione di un host simulato e dell'interfaccia virtuale ad esso associata.

In seguito sono riassunti i comandi principali dello script adibiti ad effettuare queste configurazioni:

```
ovs-vsctl add-br s2
ifconfig eth1 0.0.0.0
ovs-vsctl add-port s2 eth1
ifconfig eth2 0.0.0.0
ovs-vsctl add-port s2 eth2
ovs-vsctl set-controller s2 tcp:10.5.5.55:6633
ip addr add 10.5.5.55/24 dev s2
sleep 1
ip route add 10.1.1.0/24 dev s2
ip route add 10.9.9.0/24 dev s2
```

Una volta avviati gli script rispettivamente sui tre calcolatori, il funzionamento è praticamente identico alla prima rete virtuale di test, con le uniche differenze che ora la topologia è lineare a tre switch invece che due e che ora la rete è composta da tre componenti reali.

Il ping effettuato dall'host 1 verso l'host 2 (da computer 1 tramite comando `ip netns exec h1 ping 10.0.0.2`), viene bloccato dal controller attivo sul Raspberry in attesa dell'interazione con l'utente. Se quest'ultimo decide di autorizzare il traffico, verranno inserite in cascata sei regole nei tre switch, due regole per ognuno (di andata e di ritorno).

Infine in qualsiasi momento l'utente può visualizzare statistiche sul traffico della rete attraverso la pagina `stats.html` e aggiungere manualmente regole di traffico attraverso la pagina `index.html`.

Capitolo 6: Considerazioni finali

6.1 Conclusioni

Come si spiegava precedentemente, il modello del Software Defined Networking è un grandissimo passo avanti nel campo della gestione precisa e sicura di una rete. Attraverso un elemento come il controller, è possibile infatti amministrare e monitorare efficacemente una rete.

In definitiva, il prototipo presentato realizza tutti i comportamenti richiesti dai requisiti: permette l'aggiunta di regole di traffico sia manualmente che in tempo reale attraverso l'interazione con l'utente. Inoltre, in qualsiasi momento, è possibile monitorare tutte le regole precedentemente impostate, in maniera tale da avere un controllo efficace su tutto il traffico e nel caso decidere di interrompere qualche collegamento.

Uno dei vantaggi delle SDN infatti è che tutto è centralizzato: non occorre più andare a controllare i singoli componenti della rete per avere informazioni sulle rotte di traffico impostate.

Il controller realizzato amministra tutto alla perfezione e ha il grande vantaggio di adattarsi alle diverse topologie di rete che deve monitorare. Attraverso i test effettuati infatti si è visto che il comportamento è quello voluto anche se il numero degli switch e i loro collegamenti cambiano. È perfino possibile aggiungere o eliminare switch appartenenti alla rete mentre il controller è attivo: quest'ultimo automaticamente rileva le modifiche apportate nel sistema attraverso l'ambiente delle REst API offerto dal framework. Ryu si è rivelato il tipo di controller adatto alle esigenze del prototipo di controller realizzato: il suo ambiente di sviluppo provvede a garantire un controllo dettagliato su tutto il sistema nonché alla gestione precisa e mirata di tutti i suoi comportamenti, grazie ad un'architettura prevalentemente modulare.

Inoltre l'applicazione presenta un livello di astrazione non da poco. Infatti basti pensare che l'utente deve solo cliccare e inserire dei dati su una certa pagina, ma nel mentre vengono registrate delle modifiche sia dal controller (nel *Control Layer*) che da OpenFlow stesso (nell'*Infrastructure Layer*).

Il testing su Raspberry inoltre suggerisce l'idea che questi componenti di basso costo possano essere effettivamente utilizzati nella gestione di una rete: infatti, solamente il controller potrebbe essere confinato all'interno di uno di questi calcolatori senza che esso ne risenta.

Ovviamente occorrerebbe però valutare quanto sia effettivamente grande la rete da gestire, in modo da verificare che il Raspberry, calcolatore limitato nell'hardware, riesca a gestire una grande quantità di pacchetti che entrerebbero nel sistema ogni secondo.

Con il prototipo realizzato però si può constatare che questo computer potrebbe gestire in modo egregio reti di piccola dimensione (per esempio quelle di piccole aziende).

Questa tipologia di controller si appresta a molteplici utilizzi: per esempio potrebbe controllare efficacemente reti aziendali alle quali collegati ci sono non solo componenti di rete ma anche macchine automatiche o host di qualsiasi genere. Infatti da un'unica postazione (e quindi da remoto), è possibile avere un quadro generale di tutta la situazione complessiva della rete e dei percorsi effettivamente autorizzati per il passaggio del traffico.

In questo modo si aumenta e non di poco il fattore sicurezza della rete stessa: un host che vuole scambiare del traffico verso un altro host deve per forza essere autorizzato dalla rete. Finché non c'è autorizzazione il traffico non passa: in questo modo è possibile limitare preventivamente possibili attacchi alla rete.

L'unico problema riscontrato nel testing di questo controller è stato il suo comportamento nel caso di reti che presentano dei percorsi ciclici e quindi dei loop: in questo caso i packetIn registrati dal controller sono stati veramente tanti e i pacchetti hanno cominciato a perdersi nei percorsi ciclici. Come evidenziato precedentemente, il problema però può essere facilmente risolvibile non a livello di controller ma a livello di rete: gli switch, una volta implementati degli algoritmi di spanning tree, riuscirebbero a chiudere tempestivamente i percorsi ciclici e quindi risolverebbero ogni sorta di problema anche in queste reti particolari. Questa configurazione è molto facile da impostare tramite switch Open vSwitch (come testato precedentemente).

Concludendo, si può perciò affermare che il controller realizzato è completo: realizza infatti tutte le funzionalità richieste, fondamentali nella gestione corretta di una rete.

Inoltre, in quanto rete SDN, provvede a realizzare un notevole aumento della sicurezza, fattore di notevole importanza nell'ambito di una rete.

6.2 Sviluppi futuri

L'architettura modulare del controller si appresta in maniera egregia a possibili sviluppi successivi: se si vuole realizzare un nuovo comportamento si può infatti scrivere una nuova pagina web e gestirne le funzioni tramite script di Javascript collegati direttamente al controller per mezzo di REst API.

Inizialmente si potrebbe pensare di risolvere i problemi legati alle topologie circolari non dalla rete, ma dal controller stesso: esso potrebbe constatare che la rete presenta dei cicli e quindi tempestivamente potrebbe inviare messaggi di flow mod agli switch per correggere tali loop. Oppure si potrebbe definire il controller in modo che non permetta il passaggio di traffico su certi percorsi della rete, al fine di isolarli completamente.

Ma a parte questi possibili sviluppi che comunque rimarrebbero dei perfezionamenti dell'architettura già esistente, si potrebbe pensare di integrare questo controller sulla rete assieme ad altri controller che realizzino altre funzionalità sul traffico. Se la rete da gestire infatti fosse di grosse dimensioni, sarebbe molto efficace dividerla in varie sottoreti, ciascuna controllata da un controller differente.

Infine, come si diceva nelle conclusioni, si è certi dell'effettivo aumento della sicurezza che un controller SDN può offrire. Questo però non implica che la rete sia effettivamente sicura sotto ogni punto di vista: infatti il controller è ora un elemento fondamentale da proteggere all'interno della rete. Se con un attacco si riesce in qualche modo ad arrivare ad esso, si possono fare dei danni.

Uno sviluppo futuro del controller potrebbe essere perciò quello di creare dei controlli sul traffico che entra nel controller, attraverso la realizzazione di particolari applicazioni adibite allo scopo. Sicuramente poi avere più controller identici sulla rete potrebbe aumentare l'affidabilità della rete stessa: se il controller primario infatti si guastasse improvvisamente, gli switch non saprebbero a chi mandare i pacchetti in arrivo. Occorre perciò prevedere che un secondo controller, se per qualche motivo il traffico non arrivasse al controller principale, diventi automaticamente destinatario di tutti i packetIn. Un single point of failure non è mai una scelta saggia nello sviluppo di una rete e la ridondanza potrebbe essere una soluzione al problema.

Perciò gli sviluppi futuri sono molteplici: possono essere applicati sia per perfezionare il controller già esistente, sia per aggiungerne funzionalità nuove. Rimane comunque sicuro che possibilmente il principale sviluppo da effettuare in primis è quello di verificare e gestire nel modo appropriato l'interoperabilità del controller con altri componenti simili o identici.

Bibliografia

- [1] https://en.wikipedia.org/wiki/Software-defined_networking
- [2] Fei Hu, Qi Hao, and Ke Bao, *A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation*
- [3] <http://plvision.eu/expertise/software-defined-networking/>
- [4] <https://www.techopedia.com/definition/28935/openflow>
- [5] <https://www.quora.com/Why-dose-the-Openflow-protocol-exist-the-group-table-And-what-the-relationship-between-pipeline-and-group-table>
- [6] <http://www.crn.com/slide-shows/virtualization/300083256/the-10-coolest-software-defined-networking-technologies-of-2016.htm>
- [7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, *OpenFlow: Enabling Innovation in Campus Networks*
- [8] <http://www.datacenterjournal.com/data-center-enterprise-use-sdn-market-80-percent-year-ago/>
- [9] <http://searchcloudstorage.techtarget.com/definition/RESTful-API>
- [10] <http://flowgrammable.org/sdn/openflow/message-layer/>
- [11] https://en.wikipedia.org/wiki/List_of_SDN_controller_software
- [12] https://en.wikipedia.org/wiki/OpenDaylight_Project
- [13] <http://www.projectfloodlight.org/floodlight/>
- [14] <http://onosproject.org/>
- [15] <https://wiki.onosproject.org/display/ONOS/Intent+Framework>
- [16] <http://ryu.readthedocs.io/en/latest/index.html>
- [17] http://ryu.readthedocs.io/en/latest/app/ofctl_rest.html
- [18] https://en.wikipedia.org/wiki/Open_vSwitch
- [19] <http://mininet.org/>
- [20] <http://sdnhub.org/tutorials/sdn-tutorial-vm/>
- [21] <https://github.com/mininet/mininet/blob/master/util/install.sh>
- [22] https://osrg.github.io/ryu-book/en/html/switching_hub.html#ch-switching-hub
- [23] https://it.wikipedia.org/wiki/Web_Server_Gateway_Interface
- [24] <http://flowgrammable.org/sdn/openflow/classifiers/>
- [25] https://www.w3schools.com/jsref/met_win_clearinterval.asp
- [26] https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers
- [27] https://it.wikipedia.org/wiki/Address_Resolution_Protocol
- [28] https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol
- [29] <https://technet.microsoft.com/en-us/library/cc957928.aspx>
- [30] <http://www.iana.org/assignments/ethernet-numbers/ethernet-numbers.xhtml>
- [31] https://it.wikipedia.org/wiki/Raspberry_Pi
- [32] <http://mininet.org/walkthrough/>
- [33] [https://it.wikipedia.org/wiki/Spanning_tree_\(networking\)](https://it.wikipedia.org/wiki/Spanning_tree_(networking))

Ringraziamenti

Ringrazio il Prof. Marco Prandini e il Prof. Franco Callegati per avermi dato la possibilità di sperimentare queste nuove tecnologie attraverso un lavoro di tesi triennale.

Ringrazio il Dott. Andrea Melis per la sua pazienza e per essersi sempre dimostrato prontamente disponibile a chiarire ogni dubbio riscontrato nello sviluppo del prototipo.

Ringrazio la mia famiglia per avermi seguito durante questo cammino e per aver fatto tanti sacrifici per farmi raggiungere questo traguardo.

Ringrazio la mia fidanzata Chiara per avermi seguito, supportato (e sopportato) durante tutto il percorso della triennale, dandomi tanta fiducia e spronandomi sempre a dare il meglio di me.

Per ultimi ma non ultimi ringrazio tutti gli amici, vecchi e nuovi, che mi sono stati vicini in questi anni e hanno condiviso bei momenti con me. Un ringraziamento particolare a Davide che è stato sempre molto disponibile nei miei confronti, soprattutto in questo ultimo periodo.

Grazie.