

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Laboratorio di Amministrazione di Sistemi T

**Analisi e sviluppo di funzioni di rete virtuali
integrate in architetture SDN per applicazioni di
controllo remoto di macchine automatiche**

CANDIDATO:

DAVIDE DI DONATO

RELATORE:

MARCO PRANDINI

CORRELATORI:

FRANCO CALLEGATI, ANDREA MELIS

Anno Accademico 2015/2016

Sessione III

INDICE

ABSTRACT	1
INTRODUZIONE	2
SDN	4
Storia	4
Caratteristiche	5
Struttura.....	5
Implementazione del control plane.....	6
Flow forwarding	7
Applicazioni	7
Sicurezza.....	8
OPENFLOW	9
Storia	9
Struttura.....	9
Componenti del protocollo.....	10
State Machine.....	10
Configuration	12
Message Layer.....	13
System Interface.....	13
Tipi di messaggio notevoli	14
RYU	15
Storia e situazione corrente.....	15
Concetto e strutture comuni	15
Architettura di Ryu	16
Librerie di Ryu	16
OpenFlow Protocol e Controller	17
App e Ryu Manager.....	17
Northbound API	17
Ryu Applications	17
OPEN VSWITCH	19
Panoramica di Open vSwitch	19
Differenze tra Linux Bridge e Open vSwitch	19
Funzionamento di uno switch OpenFlow generico.....	19
Switch agent	20
Data plane	21
Ciclo di vita di un pacchetto.....	21
Struttura di Open vSwitch	22
MININET	23
Panoramica	23
Caratteristiche	23
Funzionamento.....	23
Performance	24
RASPBERRY	25
Panoramica	25
Caratteristiche	25
Hardware e software aggiuntivo utilizzato	26

TURNOUT CONTROLLER E WEB GUI	27
Obiettivi della tesi	27
Analisi dei requisiti.....	28
Casi d'uso	29
Scenari	30
Classi di analisi.....	32
Progettazione	33
Logica di base del controller: turnout.py	34
WSGI e OF_REST: turnout_rest.py	44
Realizzazione del server: my_fileserver.py	46
Web GUI: pagine HTML e file Javascript	48
Diagramma finale di progettazione.....	54
TEST SU RETE VIRTUALE	56
Topologia di test	56
Realizzazione della topologia di test	57
Esecuzione del test	59
TEST SU RETE REALE	64
Topologia di test	64
Realizzazione della topologia di test	65
Esecuzione del test	66
CONCLUSIONI	72
Test e risultati ottenuti	72
Sviluppi futuri	72
Riflessioni finali.....	73
BIBLIOGRAFIA	74

ABSTRACT

Con il concetto di SDN si intende un nuovo approccio al networking basato sulla coesistenza indipendente di due differenti “piani” in una rete: il control e il data plane. Attraverso il control plane, si riesce ad automatizzare, nel data plane, la configurazione dei dispositivi di rete, a prescindere dalle caratteristiche individuali delle apparecchiature e dalle funzionalità vendor-based offerte. Il protocollo che realizza l’astrazione che ci permette di comunicare allo stesso modo con differenti calcolatori è OpenFlow. In questo lavoro, si applica il concetto di SDN ad una rete sfruttando proprio OpenFlow, al fine di realizzare una controller application che permetta all’utente di attivare determinate funzionalità sul traffico. Inoltre, attraverso Open vSwitch, Mininet e i Raspberry, sono state poi realizzate due reti, una virtuale, l’altra reale, su cui implementare la controller application realizzata per testarne quindi il funzionamento e le performance, sia dell’algoritmo, sia dei Raspberry.

INTRODUZIONE

Questa tesi è basata sullo sviluppo di REST API per un controller SDN e sulla creazione di una Web GUI (Graphic User Interface) ad esse associata. Inoltre, è stato affrontato anche il problema dell'implementazione di questo controller su rete virtuale, realizzata attraverso Mininet, e su rete reale, realizzata invece attraverso vari calcolatori interconnessi per formare una LAN. Si vuole quindi, in definitiva, fornire una dimostrazione del funzionamento di una rete che metta in pratica il concetto di SDN [1] (Software-defined Networking) in relazione alle REST API sviluppate.

Con SDN, infatti, si intende un nuovo approccio alla gestione delle reti che non è più statico e dipendente dai vari calcolatori presenti, ma dinamico e totalmente svincolato dalla diversità delle apparecchiature affacciate sulla rete: ciò è realizzato attraverso un'astrazione delle funzionalità di più basso livello degli elaboratori stessi. Nello specifico, viene effettuato un disaccoppiamento tra il sistema che prende le decisioni su dove inviare il traffico (il cosiddetto control plane, ovvero il controller) e i sistemi sottostanti che effettivamente inoltrano il traffico alla destinazione scelta (il data plane).

Quindi, mentre questo data plane non presenta differenze rimarcabili rispetto a quello di una rete non-SDN, la novità vera, invece, ci è data dal control plane, rappresentato da un controller. Come controller, in questa tesi, si è scelto Ryu, che, a parte i numerosi pregi che verranno approfonditi più tardi, presenta inoltre un'architettura semplice e chiara (al contrario di altri controller), che si adatta particolarmente a lavori come questo. Il controller offrirà quindi alla rete la logica di base, il comportamento da realizzare in base alla REST API richiesta dalla Web GUI. Bisogna però che, oltre alla logica di base, ci sia il modo di comunicare con gli switch sotto il suo controllo le modifiche di basso livello da effettuare proprio da parte di questi ultimi, per garantire così che la richiesta dell'utente vada a buon fine.

È necessario, quindi, un protocollo, che gestisca e dia significato alle comunicazioni tra controller e switch presenti in rete, e, in particolare, riesca ad astrarre, a prescindere dal tipo di switch che riceverà la comunicazione, la serie di operazioni che quest'ultimo dovrà compiere per adeguarsi al comportamento richiesto dal controller. I più usati sono: OpenFlow [3] dell'ONF (Open Networking Foundation), Open Network Environment di Cisco e Network Virtualization Platform di Nicira. Per questa tesi però, si è scelto di utilizzare OpenFlow, dato l'enorme (e riconosciuto) lavoro di promozione delle reti SDN e standardizzazione del protocollo in oggetto da parte della ONF, che, tra l'altro, ad oggi conta più di 200 membri [4], tra cui ARM, Google, Facebook, Broadcom, Intel e Microsoft, giusto per citare i più famosi.

Scelti gli strumenti di base, l'obiettivo iniziale della tesi è stato quindi quello di realizzare delle REST API che permettessero di scegliere arbitrariamente il percorso che un traffico in ingresso da una certa porta dello switch avrebbe dovuto seguire per arrivare a destinazione, e, successivamente, una Web GUI che avrebbe garantito una corretta interazione con il controller. Poi, una volta codificata la logica di base del controller e ultimata la Web GUI, l'obiettivo finale è stato quello di testare il funzionamento di quanto preparato su una rete, prima virtuale, poi reale; in entrambe, sono stati simulati tentativi di comunicazione di vario tipo da un end-point all'altro della rete per verificare il corretto funzionamento del controller realizzato.

Per quanto riguarda la rete virtuale, si è scelto di usare Mininet [5], in quanto è un progetto open-source largamente supportato e utilizzato (soprattutto per progetti di ricerca sulle reti SDN, ad esempio per testare e verificare il funzionamento e le performance di controller SDN sperimentali [19]), che possiamo definire come un emulatore di reti rapido e intuitivo: permette di creare una rete appunto, con un numero arbitrario di host, switch e controller, e di impostare

i collegamenti tra tutti questi elementi in modo totalmente libero, consentendoci così di realizzare qualsivoglia topologia ci venga richiesta. In particolare, si è scelto di usare Open vSwitch [6], che è un noto switch multilayer virtuale, come tipo di switch presente nella rete sia virtuale che reale, per una serie di motivi: per la sua indubbia qualità (è stato usato in alcuni casi anche per prodotti commerciali), per il suo essere open-source, per le numerose feature offerte (che approfondiremo in seguito) e, infine, per la conclamata compatibilità con Ryu [2]. Per quanto riguarda invece la rete reale, si è scelto di usare come nodo intermedio un Raspberry [20] (noto ed economico single-board computer), che conterrà al suo interno sia uno switch sia il controller, modellato come descritto in precedenza, mentre agli estremi, avremo due calcolatori, ciascuno con all'interno uno switch e un host, che fungeranno da end-point; da notarsi, come vedremo meglio successivamente, che in realtà è presente nella rete un ulteriore Raspberry, sempre come nodo intermedio ma posto prima del Raspberry descritto sopra, che conterrà al suo interno un ulteriore switch e un ulteriore controller; il ruolo di quest'ultimo Raspberry verrà comunque analizzato approfonditamente più tardi. Da notarsi che tutti gli switch presenti in questa rete sono anch'essi Open vSwitch, per le stesse ragioni descritte prima, mentre i Raspberry sono stati scelti per testarne le performance in queste situazioni, e ricavarne così un termine di paragone per l'uso degli stessi in contesti più complessi e/o grandi, allo scopo, se possibile, di ridurre significativamente il costo totale delle apparecchiature necessarie ad effettuare il deploy della rete.

CAPITOLO 1

SDN

La “nuova” idea di base che imbracciamo quindi in questa tesi è l’SDN (Software-defined Networking) [1], che è fondamentalmente un approccio alla gestione delle reti che permette agli amministratori di gestire dei servizi di rete attraverso l’astrazione delle funzionalità dei livelli più bassi. Lo scopo è quello di sostituire la tradizionale architettura delle reti che non supporta l’elaborazione scalabile e dinamica e le necessità di storage di odierni environment come i data center. Il concetto principale è quello di dissociare il sistema che prende le decisioni su dove inviare il traffico (il control plane) dai sistemi sottostanti che rimandano il traffico alla destinazione selezionata (il data plane). È comunemente associato con il protocollo OpenFlow, anche se dal 2012 sono nate altre soluzioni proprietarie come Open Network Environment di Cisco e Network Virtualization Platform di Nicira.

1.1 Storia

Storicamente, i primi due progetti basati sul concetto di SDN sono stati GeoPlex e WebSprocket.

GeoPlex nacque dai laboratori AT&T e si basava sulle API di rete e sul linguaggio Java per creare un middleware di rete che mappava gli IP dove si eseguivano attività di interesse in uno o più servizi. GeoPlex era quindi, in generale, una piattaforma che gestiva l’associazione tra le reti e i servizi on-line: non si interessava quindi dei sistemi operativi presenti sugli switch o sui router, poiché AT&T voleva solo un "soft switch" che permetesse di riconfigurare gli switch fisici per adattarli a nuovi servizi presenti su un OSS (Operations Support System, sistema per il supporto alle operazioni). GeoPlex però non riusciva a cambiare radicalmente le configurazioni di questi apparati di rete in modo automatico e indipendente dai sistemi operativi, che divennero così un limite, se si considera l’odierno funzionamento di una rete basata su SDN.

WebSprocket, invece, nacque da un ingegnere di Sun, Mark Medovich, ed era composto da due entità: un NOS (Network Operating System) e un nuovo modello strutturato di runtime orientato agli oggetti (basato su Java) che poteva essere modificato in tempo reale da un compilatore e un class loader di rete. Con questo approccio, si permetteva la scrittura di applicazioni come thread Java che ereditavano le classi, il kernel e il network di WebSprocket e successivamente potevano essere modificate dal compilatore o dal class loader di rete per istanziare stack, interfacce e protocolli di rete propri. Da WebSprocket poi, derivò un tentativo di creazione di un soft switch commerciale, che rimase tuttavia incompiuto, ma per cui venne comunque realizzato, nel 2001, il primo reale test di una rete SDN.

Ulteriori sforzi, successivi al 2001, furono portati avanti da diversi ingegneri di varie società, ma solo con CableLabs, attraverso CableCard (una scheda per PC che consentiva agli utenti statunitensi di vedere e registrare canali televisivi via cavo) [25] nel 2007, si arrivò alla definizione di ciò che noi oggi intendiamo per SDN. Ulteriori passi avanti furono poi fatti dalla Berkley e dalla Stanford University intorno al 2008. L’Open Networking Foundation (ONF) venne infine fondata nel 2011 per promuovere le reti SDN e la standardizzazione di OpenFlow.

1.2 Caratteristiche

L'SDN è un'architettura pensata per essere dinamica, gestibile e adattabile. Il concetto fondamentale che permette di realizzare tutto ciò è il disaccoppiamento, tra l'effettivo controllo della rete da parte del controller, e le funzioni di forwarding usate dalle entità che realizzano l'instradamento del traffico, abilitando così la programmazione diretta del controllo della rete e l'astrazione dei livelli sottostanti attraverso le applicazioni e i servizi di rete.

Ne deriva che, nello specifico, l'architettura SDN deve essere:

- Direttamente programmabile;
- Dinamica: astrarre il controllo dal forwarding permette agli amministratori di modificare in tempo reale lo scorrere del traffico per soddisfare i cambiamenti richiesti;
- Gestita centralmente: l'intelligenza della rete è logicamente centralizzata in controller che mantengono una visione globale della rete, e appaiono alle applicazioni come singoli switch logici;
- Profondamente riconfigurabile: si possono configurare, gestire, rendere sicure e ottimizzare le risorse di rete molto rapidamente attraverso programmi SDN automatizzati e dinamici, che tra l'altro possono essere scritti senza problemi, poiché non dipendono su software proprietari;
- Basata su standard aperti e neutrale rispetto ai vendor.

L'SDN quindi, date le sue caratteristiche, permette, oltre ad una nuova gestione di una rete, di risolvere specificatamente alcuni problemi, che, con il passare del tempo, sono diventati sempre più comuni da riscontrare nelle reti, come ad esempio:

- La costante crescita della complessità delle connessioni, nel numero e nel tipo;
- La necessità di una protezione più accurata, strettamente legata al primo punto;
- La complessità nel caso dei servizi cloud;
- La gestione dei big data e dei mega datasets (dove abbiamo il problema della computazione parallela che richiede che ogni server sia connesso ad ogni altro che partecipa all'elaborazione, e tenendo conto della dimensione di queste reti, c'è la necessità di mantenere funzionanti numerose connessioni con un certo grado di affidabilità).

1.3 Struttura

I componenti dell'architettura sono diversi:

- SDN Application: programmi che, esplicitamente, direttamente e programmaticamente, comunicano i loro bisogni e quindi il comportamento di rete desiderato all'NBI Agent dell'SDN Controller attraverso l'SDN NBI (NorthBound Interface), usando l'NBI Driver. Sono costituiti dalla logica applicativa e da uno o più NBI Driver;
- SDN Controller: entità centralizzata che consiste in uno o più NBI Agent, nella logica di controllo SDN e nel driver CDPI (Control to Data-Plane Interface). L'NBI Agent gestisce la comunicazione con l'SDN Application, mentre il CDPI Driver la gestisce con l'SDN Datapath. In particolare, deve:
 - Tradurre le richieste dalla SDN Application agli SDN Datapath;
 - Fornire alla SDN Application informazioni di vario genere sulla rete (stato, eventi verificatisi, statistiche).
- SDN Datapath: dispositivo logico di rete che fornisce un incontestato controllo sull'istradamento e/o sull'elaborazione del traffico. Può includere tutte o un sottoinsieme delle risorse fisiche dell'elemento di rete; questo ci permette di definire più SDN Datapath all'interno dello stesso elemento di rete. Viceversa, è anche possibile che un

SDN Datapath contenga più elementi di rete: esiste quindi una certa libertà di definizione dello spazio logico assegnato ad un SDN Datapath. In generale, esso però comprende sempre un agent CDPI, che gestisce la comunicazione con l'SDN Controller attraverso l'SDN CDPI, e fornisce uno o più motori di trasferimento del traffico e zero o più funzioni di processamento del traffico;

- SDN Control to Data-Plane Interface (CDPI): interfaccia definita tra un SDN Controller e un SDN Datapath, che fornisce al primo:
 - Il controllo su tutte le operazioni di trasferimento dati;
 - La possibilità di negoziazione delle funzionalità attive per la comunicazione (Capabilities Advertisement);
 - La possibilità di visione di report statistici;
 - Notifiche di eventi verificatisi.
- SDN Northbound Interface (NBI): interfaccia tra SDN Application e SDN Controller che, tipicamente, attraverso quest'ultimo, fornisce una visione astratta della rete e permette l'espressione diretta del comportamento e delle richieste di rete alla prima.

Unendo insieme i componenti sopra descritti secondo le specifiche evidenziate, otteniamo lo schema riportato di seguito, che rappresenta la composizione tipica di una rete SDN.

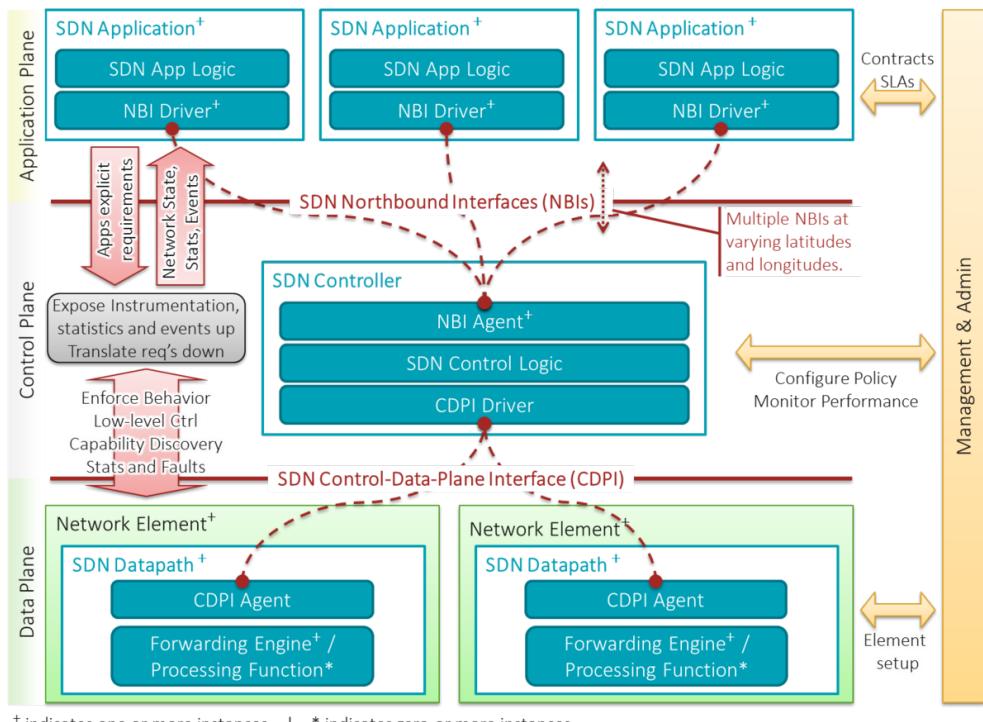


Fig. 1: schema di una rete SDN

1.4 Implementazione del control plane

Esistono tre fondamentali principi di design per l'implementazione del control plane in una rete: centralizzato, decentralizzato e gerarchico. La soluzione iniziale, che è anche la più semplice e diretta, è quella centralizzata, dove una singola entità di controllo ha una visione globale della rete; purtroppo, questa soluzione presenta evidenti limiti di scalabilità. Perciò, si sono adottati diversi approcci che si possono però ricondurre, approssimativamente, alle due categorie rimanenti sopra evidenziate. Nel caso di soluzioni gerarchiche, nella rete saranno presenti diversi controller decentralizzati, e ad ognuno verrà assegnata la responsabilità di una porzione

di rete, mentre le decisioni che richiedono una conoscenza globale della rete saranno prese dal controller centrale, posto ad un livello gerarchico più alto dei precedenti. I controller decentralizzati possono quindi prendere direttamente la loro decisione, oppure comunicare con gli altri controller presenti sullo stesso livello, al fine di conoscere la situazione in tempo reale delle altre porzioni di rete e prendere così, dinamicamente e adattandosi alla condizione del momento, la decisione migliore per il flow.

1.5 Flow forwarding

Nel protocollo comunemente più usato nelle reti SDN, ovvero OpenFlow, si usano tabelle TCAM (Ternary Content-Addressable Memory, un tipo speciale di memoria ad alta velocità ed efficienza) [26] per indirizzare sequenze di pacchetti (chiamate flows, o flussi). Questa sarà:

- Software: se si tratta di un soft switch, o di uno switch hardware facente parte di un control stack composto da più switch (ottenibile attraverso Open vSwitch, ad esempio); è anche possibile trovarla in alcuni switch hardware;
- ASIC (Application Specific Integrated Circuit, un circuito integrato personalizzato per un determinato uso) [27]: possibile se lo switch è hardware.

Se dei flussi arrivano ad uno switch, si effettua un lookup della tabella, e, se non si trova alcun riscontro, viene inviata una richiesta al controller chiedendo cosa fare del flusso. Si possono avere tre diverse modalità di funzionamento del controller:

- Modalità reattiva: il controller riceve le richieste e quindi crea e inserisce la regola per l'instradamento del flusso nella tabella secondo la logica di controllo;
- Modalità proattiva: il controller popola a priori tutte le tabelle di tutti gli switch della rete per tutti i possibili flussi;
- Modalità ibrida: il controller si comporta inizialmente in modalità proattiva, poi avvia la modalità reattiva, per gestire i possibili traffici inattesi.

Se invece il flusso ha una corrispondenza nella tabella, si applicano le azioni associate per quel particolare flusso.

1.6 Applicazioni

Le applicazioni sono numerose, ma è possibile isolare tre di maggiore interesse: SDMN, SD-WAN e SD-LAN.

L'SDMN (Software-defined Mobile Networking) è un'estensione del paradigma SDN per incorporare funzionalità specifiche di reti mobili, al fine di massimizzare l'uso di hardware e software generici appartenenti o alla rete di trasporto o alla rete di accesso radio.

L'SD-WAN è una Wide Area Network gestita usando i principi dell'SDN, al fine di minimizzare il costo della WAN usando CDN (Circuiti Diretti Numerici, ovvero linee dedicate che definiscono dei collegamenti uno ad uno o uno a molti attraverso tecniche digitali con velocità fino a 2,5 Gb/s) [28] meno costose, in alternativa alle linee MPLS (Multiprotocol Label Switching, tecnologia che permette di instradare flussi di traffico multiprotocollo tra due nodi tramite l'utilizzo di identificativi chiamati label) [29].

L'SD-LAN, infine, è una LAN gestita secondo i principi dell'SDN, di solito caratterizzata dall'uso di un sistema di gestione cloud e da una connettività wireless, senza l'utilizzo di un controller fisico.

1.7 Sicurezza

La sicurezza dell'architettura SDN in sé rimane una questione aperta, ma per quanto riguarda delle possibili applicazioni definibili per la sicurezza, create sfruttando l'architettura SDN, ci sono vari esempi: analizziamo in questa tesi una possibile protezione contro DDoS (Distributed Denial of Service) e di implementazione di un algoritmo MTD (Moving Target Defense, un algoritmo che si basa sul cambiamento o l'oscuramento dinamico di alcune proprietà chiave di una rete, con l'obiettivo di confondere l'attaccante durante la fase iniziale di studio da parte di quest'ultimo).

Per proteggersi dal DDoS, ad esempio, si può pensare di raccogliere periodicamente statistiche sul traffico nella rete in maniera standardizzata, per poi trarne delle conclusioni attraverso l'uso di algoritmi di classificazione (chiarendo se ci sono o meno anomalie) e, se viene rilevata un'anomalia, l'applicazione può istruire il controller su come riprogrammare il data plane, al fine di mitigare gli effetti dannosi del DDoS.

Mentre, per quanto riguarda l'implementazione di un algoritmo MTD su una rete SDN, è possibile subito notare che rispetto ad una comune rete, qui il compito ci è facilitato dal controller. Ad esempio, è possibile creare un'applicazione che assegna periodicamente IP virtuali agli host presenti nella rete, e lasciare che il controller gestisca il mapping tra IP virtuale e IP reale di un host al momento dell'arrivo di un traffico.

CAPITOLO 2

OPENFLOW

Discutendo di SDN, è ormai quasi impossibile non citare OpenFlow [3]. Quest'ultimo, infatti, come anticipato, è uno dei protocolli più usati quando si vuole creare una rete SDN: è fondamentale infatti per gestire e dare significato alle comunicazioni tra controller e switch presenti in rete. Anche questa tesi non farà eccezione, ed userà nello specifico OpenFlow 1.3, per garantire che il comportamento atteso sia quello richiesto attraverso le REST API. I motivi che ci hanno spinto a scegliere proprio la 1.3, a discapito di versioni più aggiornate, sono molteplici:

- La 1.3 è la versione di OpenFlow più aggiornata ad avere un Conformance Test Specification [8], che ci assicura quindi che, un hardware avente il certificato, si comporterà sicuramente secondo le linee guida dettate dal documento sopracitato, e non si riscontreranno quindi anomalie in fase di deploy;
- Nel caso della nostra tesi, essendo Open vSwitch come anticipato lo switch virtuale che andremo ad utilizzare, si è tenuto conto della tabella di compatibilità con OpenFlow [9], e si è notato che la versione più aggiornata completamente supportata di OpenFlow è la 1.3;
- È probabilmente la versione che verrà mantenuta più a lungo con ulteriori aggiornamenti, in quanto la stragrande maggioranza dei vendor senza particolare attenzione ad OpenFlow, è rimasta ferma per anni alla 1.0 e solo da un paio di anni circa hanno iniziato ad aggiornare alla 1.3, quindi è possibile affermare che quest'ultima sia un buon compromesso tra feature disponibili e supporto offerto dai vendor [10] [11].

2.1 Storia

Il concetto iniziale di OpenFlow fu descritto inizialmente alla Stanford University nel 2008, e poco dopo, nel dicembre del 2009, fu rilasciata la versione 1.0 del protocollo [12]. La successiva versione, la 1.1, fu poi pubblicata nel febbraio del 2011, e da lì in poi, lo sviluppo fu sotto la supervisione dell'ONF, che pubblicò altre 4 versioni del protocollo, arrivando quindi alla 1.5 (senza contare gli aggiornamenti rilasciati).

Contestualmente, dal 2011 in poi, numerose aziende e università hanno supportato attivamente la diffusione di questo protocollo, con progetti e ricerche, mirati a studiare il comportamento di quest'ultimo in reti SDN preesistenti e a sviluppare switch compatibili con OpenFlow.

2.2 Struttura

Arrivando ad analizzare più nel dettaglio OpenFlow: potremmo definirlo come un insieme di specifiche dettate dall'ONF [13].

La specifica principale è sicuramente quella che definisce una macchina astratta che effettua delle operazioni sui pacchetti in transito, ovvero lo switch. Quest'ultimo processa i pacchetti basandosi sia sul contenuto dei pacchetti che sulla sua configurazione attiva, ed è qui che entra in gioco il protocollo, che deve manipolare in tempo reale la configurazione dello switch ed essere in grado di ricevere e trasmettere eventi avvenuti sullo switch.

Invece, il controller, dal canto suo, deve saper “parlare” il protocollo per inviare agli switch i comandi astratti di manipolazione della configurazione e rispondere agli eventi che sono avvenuti.

Così definite le entità in gioco, possiamo osservare nell'immagine di seguito una schematizzazione astratta di una rete SDN, costruita appunto dalle specifiche definite dall'ONF e sopra riportate.

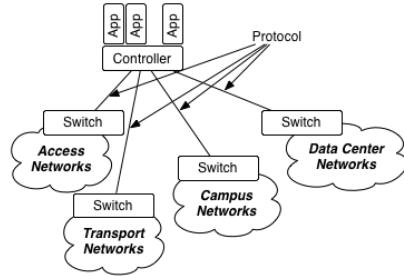


Fig. 2: schema astratto di una rete SDN

2.3 Componenti del protocollo

Separando poi il protocollo dalle altre due specifiche, ed esaminandolo singolarmente, non in merito quindi alle relazioni con le altre due entità, possiamo scomporlo in quattro componenti: il livello di messaggistica (Message Layer), la macchina a stati (State Machine), l'interfaccia di sistema (System Interface) e il livello di configurazione (Configuration).

Di seguito possiamo osservare lo schema dello stack del protocollo OpenFlow, delineato dai quattro componenti sopra descritti.

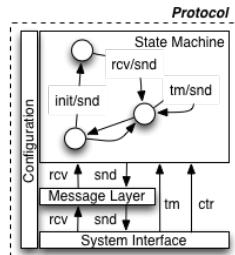


Fig. 3: scomposizione del protocollo OpenFlow

2.4 State Machine

Iniziamo con l'analizzare la macchina a stati finiti: essa definisce il comportamento del protocollo a basso livello. OpenFlow è piuttosto semplice in realtà, quasi ogni messaggio che viene scambiato non fa parte di una conversazione che lo racchiude, di cui abbiamo bisogno di conservare lo stato e le informazioni, l'unica eccezione è la fase di apertura della connessione, che prevede la negoziazione delle funzioni attive e della versione di OpenFlow da usare per lo scambio di messaggi, e che quindi dovrà essere gestita da una macchina a stati finiti.

Per quanto riguarda quindi l'apertura della connessione, possiamo dividerla in più fasi:

1. Creazione di una connessione TCP/TLS tra switch e controller;
2. Scambio simmetrico di un pacchetto di tipo OFPT_HELLO, per coordinare la versione da usare tra il controller e lo switch. Se questa fase fallisce, per un errore, un timeout o per versioni non compatibili, la macchina a stati finiti di entrambi i calcolatori viene riportata allo stato iniziale. Di seguito, possiamo trovare due schemi rappresentanti l'inizio della comunicazione tra switch e controller, mettendo in particolare evidenza il caso in cui lo scambio descritto sopra sia avvenuto con successo (nel primo schema) oppure no (nel caso del secondo e terzo schema);

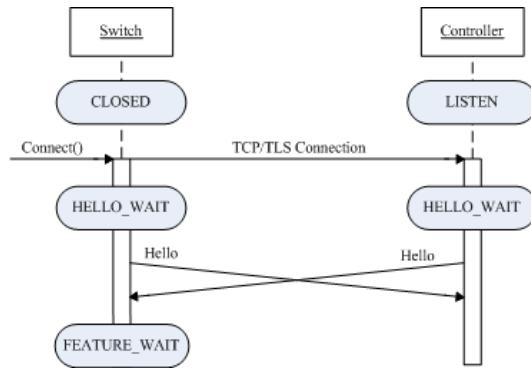


Fig. 4: scambio di Hello avvenuto con successo

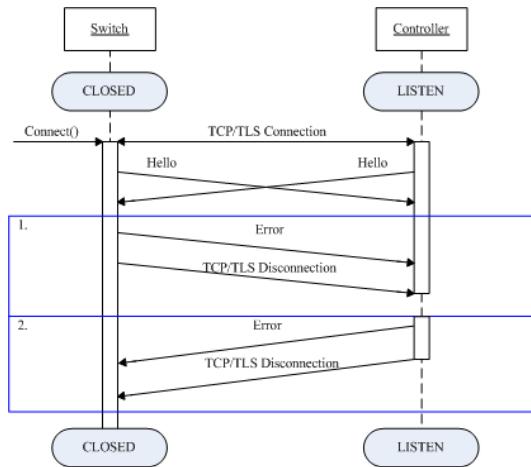


Fig. 5: scambio di Hello fallito per la mancanza di versione compatibile con entrambe le entità

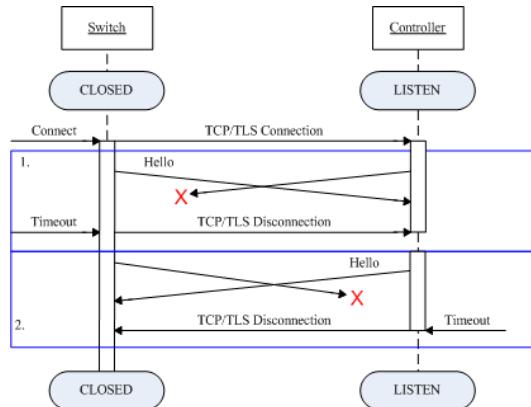


Fig. 6: scambio di Hello fallito per il mancato arrivo di Hello da una o dall'altra parte

3. Invio di un pacchetto di tipo OFPT_FEATURES_REQUEST dal controller allo switch, attraverso il quale il primo chiede al secondo quali funzioni possiede;
4. Invio di un pacchetto di tipo OFPT_FEATURES_REPLY dallo switch al controller, dove lo switch enumera tutte le funzioni da lui possedute: se il controller non riceve questo messaggio, si disconnette dallo switch. Di seguito, è possibile trovare due schemi che mostrano il primo uno scambio eseguito con successo, il secondo uno scambio fallito per mancato arrivo del messaggio da una o dall'altra parte.

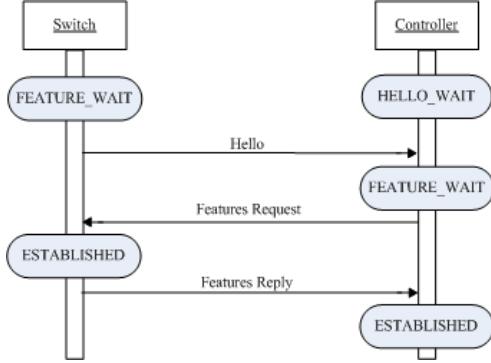


Fig. 7: scambio di features messages tra switch e controller avvenuto con successo

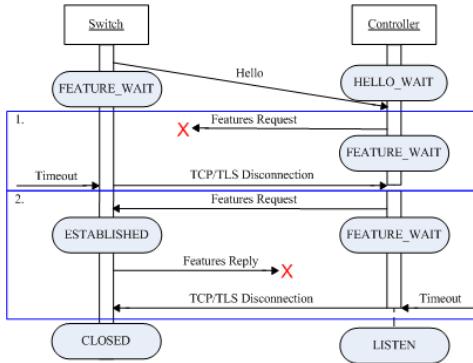


Fig. 8: scambio di features messages tra switch e controller fallito per la mancata ricezione del messaggio atteso dall'una o dall'altra parte

È inoltre presente, da OpenFlow 1.3, l'instaurazione di più connessioni ausiliarie [14], oltre alla principale, al fine di migliorare le performance dello switch sfruttando il parallelismo computazionale che molte apparecchiature di questo tipo offrono, anch'esse gestite dalla state machine.

2.5 Configuration

Passiamo ora al livello di configurazione. Questo, è il livello che si occupa di: fornire un linguaggio, che servirà a comunicare sia con il controller che con lo switch, e un utility, che si occuperà di controllare la grammatica di quanto scritto e genererà comandi che possono essere eseguiti per manipolare le impostazioni del controller o dello switch.

Il linguaggio è strutturato su due entità: le fasi e i tipi. I tipi sono predefiniti, e, nello specifico, sono composti da un insieme di costanti (che possono essere di IPv4, IPv6, interi, protocolli, stringhe e versioni di OpenFlow). I tipi predefiniti, poi, concorrono a formare le diverse fasi: esse devono essere eseguite dalla prima all'ultima, secondo l'ordine specificato, e descrivono in modo completo il comportamento che un controller o uno switch dovrebbe avere durante la creazione o l'accettazione di nuove connessioni attraverso i tipi presenti e il valore ad essi associato.

Andando ad ispezionare l'utility, invece, possiamo distinguere quattro diverse fasi di lavoro della stessa: il tokenizing, il parsing, l'elaborazione e la valutazione. La fase di parsing (che si può dire comprenda anche il tokenizing) si limiterà a controllare la validità dell'input rispetto ad una certa grammatica, la fase di elaborazione si occuperà di verificare la corrispondenza tra

tipi attesi e tipi riscontrati nell'input e la fase di valutazione, infine, leggerà l'input e lo trasformerà in comandi che possono essere eseguiti dallo switch o dal controller.

2.6 Message Layer

Analizziamo ora il livello di messaggistica. Possiamo dire che questo sia il livello chiave dello stack di OpenFlow, dato che definisce le strutture e le semantiche valide per ogni tipo di messaggio esistente. Si occupa in generale di costruire, comparare e manipolare messaggi. Ogni messaggio OpenFlow inizia sempre con la stessa struttura dell'header, a prescindere dalla versione utilizzata. In questo modo, possiamo sempre venire a conoscenza di tre importanti informazioni con cui identificare il messaggio, accedendo ai rispettivi campi: il campo della versione (Version Field), che ci indica la versione di OpenFlow a cui il messaggio appartiene, il campo della lunghezza (Length Field), che ci indica la lunghezza del messaggio per poter calcolare dove quest'ultimo terminerà nello stream di bytes, e lo xid (o identificatore di transazione), che contiene un valore usato per trovare corrispondenze con messaggi precedenti (ad esempio, riconoscere la risposta associata ad una particolare richiesta precedente). L'ultimo campo presente è il campo del tipo (Type Field) che ci indica che tipo di messaggio è presente e come interpretare quindi il payload, ma tutto ciò è ovviamente dipendente dalla versione di OpenFlow in uso. Di seguito, è possibile trovare una schematizzazione della struttura di un messaggio OpenFlow.

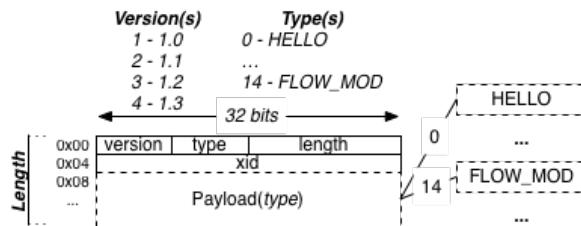


Fig. 9: schematizzazione del messaggio OpenFlow

2.7 System Interface

Infine, arriviamo all'interfaccia di sistema. Essa definisce come il protocollo interagisce con il “mondo esterno”, o, meglio, come interagisce con altri componenti che devono essere presenti sulla rete per il corretto funzionamento di quest'ultima con OpenFlow, ma che siano comunque esterni al protocollo stesso. In particolare, esistono quattro interfacce: l'interfaccia TCP/TLS, l'interfaccia switch agent, l'interfaccia controller application e l'interfaccia di configurazione (da non confondere con il livello di configurazione, a cui però è strettamente legata).

L'interfaccia TCP/TLS interagisce appunto con i protocolli di livello più basso e fornisce quindi un canale stream-oriented affidabile attraverso il quale controller e switch possono comunicare attraverso messaggi di OpenFlow.

L'interfaccia switch agent, invece, interagisce con il kernel di sistema di uno switch OpenFlow: si occupa di inoltrare i messaggi validi arrivati dal controller al kernel di sistema dello switch per processarli e di inoltrare messaggi asincroni generati nello switch alla controller application attraverso lo stack OpenFlow.

L'interfaccia controller application, dal canto suo, interagisce proprio con la controller application, posta in cima allo stack OpenFlow, e si occupa di gestire proprio lo scambio di messaggi da e verso la controller application stessa.

L'interfaccia di configurazione, infine, permette ad un operatore di sistema di configurare lo stack OpenFlow, cambiandone i parametri di configurazione prima o durante l'uso.

2.8 Tipi di messaggio notevoli

Oltre ai tipi di messaggio usati per il corretto stabilimento della connessione, ne esistono altri, che è importante citare: l'OFPT_PACKET_IN, l'OFPT_PACKET_OUT, l'OFPT_FLOW_MOD [14].

Il primo è un messaggio asincrono, inviato dallo switch al controller, che si emette se nessuna entry nella tabella dello switch corrisponde al flow in entrata o se è stata impostata come azione associata l'invio al controller. Può consistere nell'intero pacchetto arrivato allo switch, o solo di una parte e di un intero, chiamato Buffer Id, che indica la locazione del pacchetto nello switch (come sarà composto dipenderà sempre dalle azioni impostate per quel traffico).

Il secondo è un messaggio che spesso si trova in risposta al primo, viene inviato dal controller allo switch ed indica allo switch stesso di inviare un determinato pacchetto attraverso una certa porta: anche qui, il messaggio può contenere l'intero pacchetto o specificare solamente il Buffer Id; dipende da come si è ricevuto il pacchetto che si vuole emettere.

Il terzo e ultimo tipo di messaggio viene sempre inviato dal controller allo switch e contiene dei comandi che permetteranno di aggiungere, modificare o eliminare entry nella tabella dello switch stesso.

Sono tipi di messaggio che ovviamente si riscontrano molto di frequente nelle reti SDN, e, anche in questa tesi, saranno largamente sfruttati per realizzare la controller application richiesta.

CAPITOLO 3

RYU

Avendo quindi definito la struttura della rete SDN, e, come il protocollo scelto, ovvero OpenFlow, agisca per garantire il funzionamento della rete secondo le aspettative, analizziamo adesso il controller, ovvero l'entità che principalmente contiene la logica di controllo della rete SDN.

Per questa tesi, si è scelto di utilizzare Ryu [15]. Supportato da NTT e scritto interamente in Python, di sicuro presenta molte caratteristiche positive: è open-source, presenta API ben definite, è component-based e quindi permette il facile riutilizzo e modifica di moduli preesistenti e, infine, ne è stato testato il funzionamento su diversi switch, tra cui su uno virtuale (Open vSwitch, che useremo) e su diversi reali (tra cui apparecchiature di HP, NEC e IBM). La versione scelta per l'uso sulle macchine virtuali è la 4.9, mentre sui Raspberry la 4.10: in generale, si è cercato di utilizzare sempre la versione più recente disponibile (all'inizio del lavoro di tesi, la versione più recente di Ryu disponibile era la 4.9, rilasciata in data 05/12/16, mentre in data 13/01/17 è stata poi rilasciata la 4.10, che abbiamo adottato quindi per i nostri test su rete reale).

3.1 Storia e situazione corrente

Il primo controller SDN fu NOX, sviluppato inizialmente da Nicira Networks, e donato poi, nel 2008, alla comunità SDN, rendendo quindi il controller open-source. Successivamente, Nicira si dedicò allo sviluppo di un nuovo controller, chiamato ONIX, con Google e NTT; questo controller però, contrariamente ai piani originali, non divenne mai open-source.

Un nuovo famoso controller SDN che mano a mano rubò la scena fu Beacon, il cui sviluppo era iniziato nel 2010: era un controller Java-based che supportava OpenFlow. Da questo, nacque Floodlight, altro importante controller SDN che fornì la base per uno dei primi controller commerciali.

Successivamente, Cisco, HP, IBM e altri importanti vendor arrivarono a proporre i loro controller SDN: inizialmente questi ultimi erano basati su Beacon, ma ora si sono spostati verso OpenDaylight. Quest'ultimo, insieme a Ryu e a ONOS, costituisce un trittico di controller SDN open-source molto conosciuto, supportato e usato, al giorno d'oggi.

3.2 Concetto e strutture comuni

Un controller SDN, come già anticipato, è possibile vederlo come un'applicazione che agisce da punto di controllo strategico in una rete SDN: gestisce le configurazioni degli switch/router attraverso southbound API (che agiscono attraverso l'SDN CDPI), mentre le applicazioni e la logica di business attraverso northbound API (che agiscono attraverso l'SDN NBI). È possibile anche identificare un certo tipo di SDN Controller che agisca come una piattaforma: in questo caso, sono presenti già insieme al controller numerosi moduli indipendenti che è possibile usare per eseguire diverse operazioni di rete di base. Inoltre, è possibile sempre usare questi moduli come base di partenza per creare funzioni più complesse o semplicemente modificarli per migliorare o adattare secondo le proprie esigenze le funzioni offerte.

Ryu, ad esempio, ha proprio quest'ultimo specifico funzionamento, che lo rende, come anticipato nell'introduzione, facilmente modificabile e adattabile alle proprie necessità.

3.3 Architettura di Ryu

L'architettura di Ryu [16], seppur composta da molte entità, risulta, come detto, abbastanza semplice e intuitiva. Le elenchiamo qui sotto spiegandole rapidamente (alcune le riprenderemo e le analizzeremo meglio più avanti). Sono presenti:

- Northbound API: include il plug-in Openstack Neutron e il componente OF-Rest;
- Ryu Applications: applicazioni built-in o scritte da noi;
- Core Processes: componente che include molte funzionalità tra cui la gestione degli eventi, dei messaggi e dello stato del controller. Da notarsi che, anche se scritto con Python, il servizio di messaggistica supporta componenti sviluppati con altri linguaggi;
- App-Manager: componente fondamentale per tutte le applicazioni Ryu;
- Ryu-Manager: eseguibile principale;
- OpenFlow Protocol: è presente il supporto fino alla versione 1.5 (la più aggiornata);
- OpenFlow Controller: componente che si occupa di gestire gli switch OpenFlow.
- Librerie: Ryu contiene nativamente sia il supporto a vari protocolli di southbound sia diverse operazioni di processamento dei pacchetti.

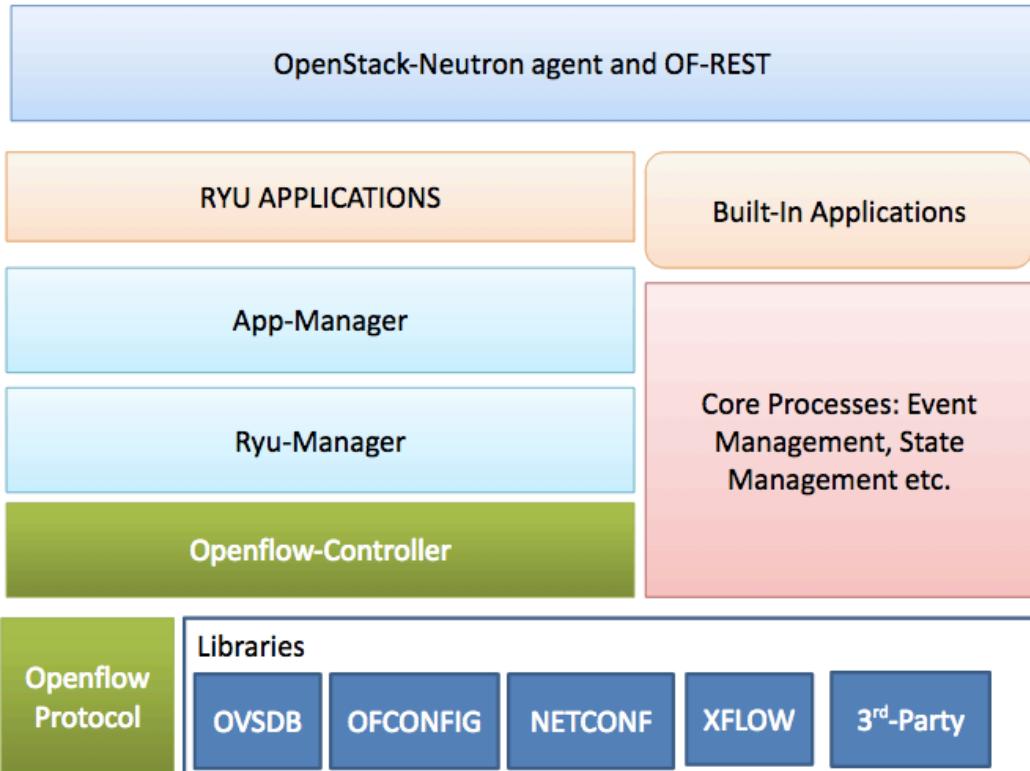


Fig. 10: schema dell'architettura di Ryu

3.4 Librerie di Ryu

In particolare, per quanto riguarda le librerie, possiamo vedere come, per quanto riguardi i protocolli di southbound, Ryu supporti OF-Config, Open vSwitch Database Management Protocol (OVSDDB), NETCONF e XFlow ed altri ancora di terze parti. Ad esempio, XFlow è principalmente usato per effettuare vari tipi di misurazioni riguardo al traffico passante in una rete. Inoltre, Ryu possiede una libreria per i pacchetti, che possiamo usare per effettuare il parsing e costruire dei pacchetti del protocollo desiderato (VLAN, MPLS e GRE, ad esempio).

3.5 OpenFlow Protocol e Controller

Per quanto riguarda il componente OpenFlow Protocol, esso semplicemente si limita a contenere due librerie: una di codifica e una di decodifica del protocollo.

L'OpenFlow Controller, invece, è più complesso: possiamo definirlo come la principale sorgente di eventi interni nell'architettura Ryu. La tabella sottostante associa le strutture e i messaggi OpenFlow alle corrispondenti API offerte da Ryu.

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

3.6 App e Ryu Manager

Come già detto, Ryu Manager è l'eseguibile principale: crea un processo che si pone in ascolto sull'IP specificato e sulla porta specificata (altrimenti, si usa 6633) e permette la connessione da parte degli switch.

L'App Manager invece, contiene una classe fondamentale, chiamata RyuApp, che tutte le applicazioni Ryu devono ereditare, secondo le specifiche dell'architettura.

3.7 Northbound API

Il plug-in di Openstack Neutron ci fornisce la possibilità di utilizzare GRE (Generic Routing Encapsulation, un protocollo di tunneling che permette di incapsulare numerosi protocolli di rete per collegamenti virtuali punto a punto) [30] e di creare configurazioni VLAN (Virtual LAN, un insieme di tecnologie che permettono di segmentare il dominio di broadcast in più reti logicamente non comunicanti tra loro, anche se in realtà condividono globalmente la stessa infrastruttura fisica) [31].

Invece, OF-Rest è un'interfaccia REST, che, in combinazione con WSGI (Web Server Gateway Interface) permette facilmente di introdurre nuove REST API in un'applicazione. Questi ultimi due strumenti saranno proprio quelli che useremo (come vedremo più tardi) per creare le REST API che realizzeranno il comportamento richiesto dal controller.

3.8 Ryu Applications

Le applicazioni Ryu sono entità single-thread, che implementano varie funzionalità.

Nello specifico, ogni applicazione Ryu gestisce una coda di ricezione per gli eventi, che è FIFO (First In First Out), per preservare l'ordine di arrivo degli eventi.

Il thread principale (padre) si limita ad eseguire in loop l'estrazione del primo evento in coda e successivamente la creazione di un nuovo thread figlio che esegua l'event handler appropriato per il tipo di evento estratto; se non ci sono eventi da servire, invece, si mette in attesa. Ammesso che si stia gestendo un evento, e che sia stato creato quindi il thread figlio, il thread padre si mette in attesa finché il figlio non termina, e solo allora il padre riprende l'attesa degli eventi da gestire (o serve il primo evento in attesa, se è presente).

Di seguito, riportiamo lo schema riassuntivo di funzionamento di un'applicazione Ryu.

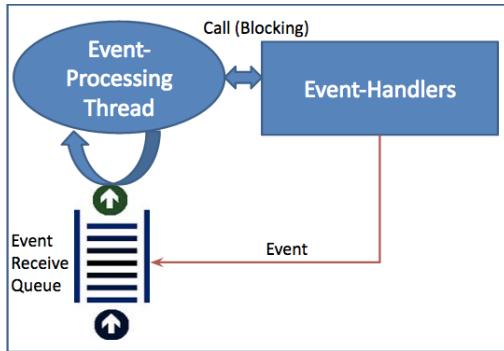


Fig. 11: schema di esecuzione dell'applicazione Ryu

CAPITOLO 4

OPEN VSWITCH

Iniziamo ora ad analizzare gli strumenti utilizzati per la fase di testing su rete virtuale e reale, e iniziamo proprio con il componente di base che verrà utilizzato per entrambe le reti: Open vSwitch [6]. Trattasi, come già detto, di uno switch multilayer virtuale open-source, di qualità riconosciuta: sarà quindi, nella nostra rete, l'entità preposta alla comunicazione con il controller, al fine di ricevere le indicazioni su come modificare il proprio comportamento per adattarlo a quello richiesto. In questa tesi, al contrario di Ryu, Open vSwitch non è mai stato aggiornato all'ultima versione disponibile: si è usata, per le macchine virtuali, la versione 2.5.0, mentre, per i Raspberry, la versione 2.3.0; nel primo caso, come vedremo meglio in seguito, è stata usata come macchina virtuale una già pronta all'uso con strumenti come Mininet, Open vSwitch e Ryu preinstallati, mentre, nel secondo, si è eseguita l'installazione da package (vedremo successivamente i dettagli). Non si è proceduto ad effettuare un aggiornamento poiché sia la versione 2.3 sia la versione 2.5 supportano già completamente OpenFlow 1.3 [9]; inoltre, non si sono riscontrati problemi nell'uso della sopracitata versione.

4.1 Panoramica di Open vSwitch

Open vSwitch è quindi uno switch che supporta interfacce di gestione standard e permette il controllo e l'estensione delle funzioni di forwarding in modo programmatico. Come risulta chiaro, Open vSwitch si adatta bene alla funzione di virtual switch in ambienti basati su VM (Virtual Machine), che è esattamente uno degli scopi per cui è stato utilizzato in questa tesi. È scritto quasi interamente in C, e offre diverse feature, tra cui il supporto alla VLAN, possibilità di configurazione della QoS (Quality of Service) e soprattutto, supporto ad OpenFlow e a diverse estensioni ad esso relative. Consente un forwarding altamente efficiente grazie all'uso di un modulo di Linux preesistente (uno dei principi chiave di Open vSwitch è stato proprio quello del riutilizzo, quando possibile, di moduli preesistenti).

4.2 Differenze tra Linux Bridge e Open vSwitch

Il primo quesito che sorge è sicuramente la motivazione che spinge all'uso di un ulteriore applicativo, in questo caso Open vSwitch, invece che il bridge Linux già installato e pronto all'uso (a patto di essere su SO Linux, ovviamente). In termini generali, è possibile dire che il primo si adatta meglio a scenari di frequenti cambiamenti [7]: feature come il mantenimento degli stati fisici e logici (appartenenti quindi ai livelli 2 e 3 dello stack) nella migrazione di una macchina virtuale da un host all'altro e la gestione in tempo reale di cambiamenti alla topologia della rete (attraverso OpenFlow), lo rende estremamente utile anche per ricerche e test di vario genere. Risulta quindi evidente come Open vSwitch sia stato in realtà ideato per il controllo di reti automatizzate e dinamiche, come appunto le reti SDN, e sia la scelta ideale per il lavoro di questa tesi.

4.3 Funzionamento di uno switch OpenFlow generico

Per chiarire il funzionamento di base di Open vSwitch, o di un qualsiasi altro switch che supporti OpenFlow, riconduciamoci a un caso generico, a una struttura che è, approssimativamente, sempre possibile individuare, a prescindere dal tipo di switch di cui siamo a disposizione; al contrario dei controller, per cui è molto difficile individuare una struttura di base, data la grande varietà esistente al giorno d'oggi (e già presentata nel capitolo precedente) e il diverso ruolo

che li svincola maggiormente, rispetto agli switch, da un inquadramento in un'unica struttura generale.

Incominciamo quindi l'analisi di uno switch OpenFlow generico [18]. Questo può essere diviso in due componenti: lo switch agent e il data plane interno.

Lo switch agent, genericamente, parla, attraverso il protocollo OpenFlow, a uno o più controller e al data plane. Infatti, lo switch agent tradurrà i comandi, provenienti dal controller, nelle istruzioni di basso livello necessarie e li manderà al data plane, e tradurrà le notifiche del data plane interno in messaggi OpenFlow e li invierà al controller.

Il data plane interno effettua invece manipolazioni e inoltro di tutti i pacchetti in transito per lo switch; a volte, a seconda delle configurazioni, manderà i pacchetti allo switch agent per ulteriori manipolazioni. Di seguito, troviamo lo schema astratto del generico switch OpenFlow.

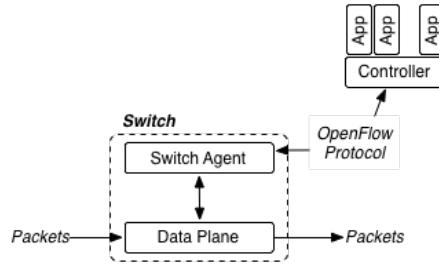


Fig. 12: schema astratto di uno switch OpenFlow

4.4 Switch agent

Lo switch agent si può scomporre in diverse entità: OpenFlow protocol (protocollo OpenFlow), core logic (logica di base), data plane protocol (protocollo del data plane), data plane offload (ulteriori funzionalità del protocollo).

- OpenFlow protocol: istanza del protocollo OpenFlow presente sullo switch;
- Core logic: componente che gestisce lo switch, in particolare esegue comandi sul data plane e sfrutta le funzioni offerte dal data plane offload;
- Data plane protocol: istanza del protocollo interno usata per configurare lo stato del data plane.

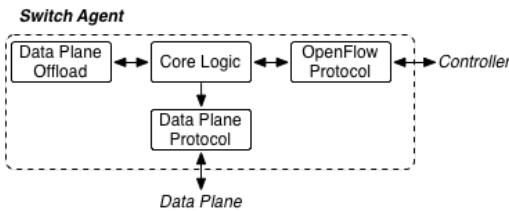


Fig. 13: schema interno dello switch agent

4.5 Data plane

Il data plane di uno switch OpenFlow è composto invece di porte, flows, tabelle di flow, classificatori e azioni. I pacchetti arrivano e lasciano il sistema attraverso le porte dello switch: attraverso i classificatori, riusciamo a capire se, per il traffico in ingresso, abbiamo già impostato il comportamento da eseguire, guardando nelle tabelle di flow. Se presente in una delle tabelle, ad ogni flow si associa un insieme di azioni da applicare a ogni pacchetto appartenente a quel flow. Di seguito, è presente lo schema interno del data plane.

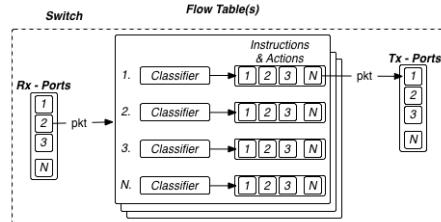


Fig. 14: schema interno del data plane

4.6 Ciclo di vita di un pacchetto

Se si esamina il ciclo di vita di un pacchetto che attraversa il data plane di uno switch, è possibile arrivare ad uno schema fisso, riportato di seguito.

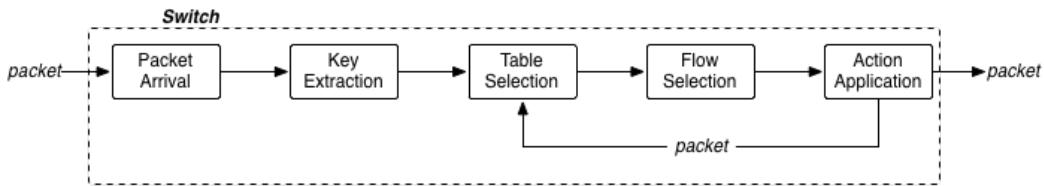


Fig. 15: ciclo di vita di un pacchetto

Le fasi sono diverse: arrivo del pacchetto, estrazione della chiave, selezione della tabella di flow, selezione del flow, applicazione dell'azione.

Prima di iniziare ad analizzarle una ad una, è importante quindi ricordare che, come emerge anche dallo schema, quando un pacchetto arriva ad uno switch, viene costruita una chiave formata da informazioni sia dal pacchetto arrivato, come ad esempio la versione del protocollo, sia riguardo a come e quando è arrivato, come ad esempio la porta di arrivo o il tempo di arrivo. La chiave è quindi poi usata per cercare una corrispondenza all'interno del flow table e applicare le azioni associate al flow (se la corrispondenza esiste) e applicarle al pacchetto. Le azioni applicabili sono diverse (ne parleremo comunque più avanti).

Passiamo a spiegare le fasi:

1. Arrivo del pacchetto: il pacchetto arriva ad una qualsiasi porta, fisica o virtuale, dello switch, e vengono memorizzate informazioni al riguardo;
2. Estrazione della chiave: il pacchetto viene analizzato e, unitamente a dei dati esterni al pacchetto, viene creata la chiave, che sarà l'unica entità che passerà alla prossima fase;
3. Selezione della tabella di flow: se esistono più tabelle di flow, bisogna selezionare la tabella in cui cercare. Se si è alla prima interazione con la fase di selezione della tabella di flow, si seleziona la prima tabella di default e mano a mano che si ripetono le interazioni viene cercata sulla tabella successiva;
4. Selezione del flow: il primo flow della tabella che possieda un classificatore che inglobi la key in esame, diventa il flow selezionato;

5. Applicazione dell'azione: al flow selezionato corrisponde un insieme di azioni, da applicare a tutti i pacchetti del suddetto flow; è da notarsi che, tra le azioni possibili, c'è anche quella dell'inoltro della key a un'altra tabella dello switch per l'applicazione di ulteriori azioni.

4.7 Struttura di Open vSwitch

Andando invece ad esaminare nel dettaglio la nostra scelta, Open vSwitch [17], possiamo distinguere diversi componenti. I principali sono:

- ovs-vswitchd: un demone che implementa lo switch, insieme a un modulo del kernel Linux per lo switching basato sui flow;
- ovsdb-server: un database leggero ospitato su un server, a cui ovs-vswitchd si può rivolgere per configurarsi;
- ovs-dpctl: un tool per configurare il modulo switch del kernel;
- ovs-vsctl: un utility per richiedere e ottenere la configurazione di ovs-vswitchd;
- ovs-appctl: un utility che invia comandi ai demoni Open vSwitch attivi.

Altri tool importanti, offerti da Open vSwitch, sono:

- ovs-ofctl: un utility per richiedere e controllare switch e controller OpenFlow;
- ovs-pki: un utility per creare e gestire un'infrastruttura a chiave pubblica per gli switch OpenFlow;
- ovs-testcontroller: un semplice controller OpenFlow.

Tra tutti questi, in particolare in questa tesi, useremo attivamente ovs-vsctl per controllare le configurazioni attive degli switch e assicurarci così che rispecchino esattamente quelle attese; un altro tool che useremo, per configurare inizialmente i nostri switch come richiesti, sarà ovs-ofctl, che ci permetterà di impostare con velocità e semplicità tutte le caratteristiche necessarie (come vedremo in seguito).

CAPITOLO 5

MININET

Passiamo ora ad esaminare il simulatore di rete utilizzato per costruire la rete virtuale richiesta: Mininet [5]. Trattasi, come anticipato, di un simulatore di reti rapido e intuitivo: ci permette di creare una rete con una qualsiasi topologia e costituita da un numero arbitrario di switch, host e controller. Anche qui, per motivazioni analoghe a quanto descritto all'inizio del capitolo 4, abbiamo usato diverse versioni: per quanto riguarda le macchine virtuali, si è usata la versione 2.2.1, mentre sui Raspberry la più aggiornata 2.3.0d1.

5.1 Panoramica

Mininet è, come ormai sappiamo, un emulatore di rete: crea una rete di host, switch, controller e collegamenti virtuali.

Gli host di Mininet hanno in esecuzione software di rete standard di Linux, mentre i suoi switch supportano OpenFlow per consentire un routing personalizzato altamente flessibile e la creazione di una rete SDN.

Viene largamente usato per scopi di ricerca, sviluppo, prototipazione e testing.

5.2 Caratteristiche

- Fornisce un semplice ed economico testbed di rete per lo sviluppo di applicazioni OpenFlow;
- Permette a più sviluppatori di lavorare indipendentemente in contemporanea sulla stessa topologia;
- Supporta regression tests (tipologia di collaudo del software che verifica se e come il comportamento del software precedentemente sviluppato cambia, se si modifica l'ambiente di interazione del software) a livello di sistema;
- Abilita il testing di complesse topologie di rete, senza la necessità di creare fisicamente la rete; include inoltre un set di base di topologie parametrizzate;
- Include una CLI (Command Line Interface) che conosce la topologia della rete e OpenFlow, è utilizzabile per il debugging e per condurre test di rete;
- Usabile out-of-the-box;
- Fornisce una API Python chiara ed estensibile per la creazione e sperimentazione di reti.

In breve, queste feature permettono di muoversi da Mininet a una rete reale con cambiamenti minimi (lo vedremo meglio in seguito).

5.3 Funzionamento

Quasi ogni sistema operativo utilizza l'astrazione del processo per virtualizzare le risorse computazionali. Invece, Mininet, usa una virtualizzazione process-based per creare (si è testato fino a 4096) host e switch su un singolo kernel di sistema operativo: difatti, Linux supporta network namespaces, cioè una feature che permette di creare virtualizzazioni leggere che associano a singoli processi (process-based virtualization) interfacce di rete differenti (insieme a diverse tabelle di routing e tabelle ARP). Linux mette poi a disposizione ulteriori feature, ma Mininet non le usa.

Ricapitolando, Mininet può creare:

- Switch OpenFlow a livello kernel o a livello user-space (dove user-space corrisponde a un insieme di locazioni dove i normali processi utenti vengono eseguiti, mentre kernel-space alla locazione dove è immagazzinato il codice del kernel, e da cui esegue; ovviamente quelli a livello kernel sono più performanti);
- Controller: per controllare gli switch;
- Host: che simulano comunicazioni attraverso la rete.

Mininet connette switch e host attraverso virtual ethernet (veth).

Per adesso, inoltre, dipende dal kernel Linux, ma ha l'intenzione di espandersi per utilizzare virtualizzazioni di altri sistemi operativi, come quelle di Solaris o di FreeBSD.

Infine, Mininet è scritto interamente in Python, eccetto per una piccola utility in C.

5.4 Performance

Mininet combina molte delle migliori feature degli emulatori, dei testbed hardware e dei simulatori; comparata con approcci di virtualizzazione di sistemi completi, Mininet è molto più performante in termini di boot time, scalabilità, facilità di installazione e banda (tipicamente 2 Gbps di banda totale anche su hardware modesto). Inoltre, è gratuito e sempre disponibile, velocemente riconfigurabile e riavviabile. Infine, esegue codice vero e originale (codice delle applicazioni, del kernel e del control plane), si connette facilmente a reti reali ed è possibile manipolare la performance dei componenti della rete in tempo reale.

CAPITOLO 6

RASPBERRY

Come ultimo degli strumenti utilizzati per questo lavoro di tesi, c'è il Raspberry [20]. È un famoso calcolatore dalle medio-basse prestazioni e dal costo contenuto, concepito inizialmente con l'obiettivo di stimolare l'insegnamento di base dell'informatica nelle scuole, ma che ultimamente sta espandendo il suo raggio d'utilizzo e sta trovando spazio nei progetti più disparati. In questa tesi, si è scelto di usare il Raspberry Pi 2 Model B, che, al contrario del modello Pi 1, fornisce performance ragionevoli, e, al contrario del modello Pi 3, offre solo caratteristiche strettamente necessarie per questo lavoro (non c'era bisogno in questo caso di Bluetooth o Wi-Fi, offerti invece dal Pi 3).

6.1 Panoramica

Il Raspberry Pi, come già detto, è un single-board computer, ovvero è un calcolatore implementato su una sola scheda elettronica, sviluppato nel Regno Unito dalla Raspberry Pi Foundation. Il suo lancio al pubblico è avvenuto il 29 febbraio 2012: da allora, molti prodotti si sono avvicendati, sino ad arrivare all'attuale Raspberry Pi 3 Model B; si è però mantenuta la struttura di base.

Il progetto infatti, ruota attorno a un System-on-a-chip (SoC) Broadcom, che incorpora un processore ARM, una GPU VideoCore IV e una RAM che varia da 256 MB a 1 GB in base al modello considerato. Infine, il progetto non prevede né hard disk né unità a stato solido, ma si affida ad una scheda SD per il boot e per la memoria non volatile. È un calcolatore pensato per ospitare sistemi operativi basati sul kernel Linux o RISC OS.

6.2 Caratteristiche

Di seguito, troviamo un'immagine dall'alto del Raspberry Pi 2 Model B, dove possiamo iniziare a distinguere vari componenti che andremo a citare subito sotto.

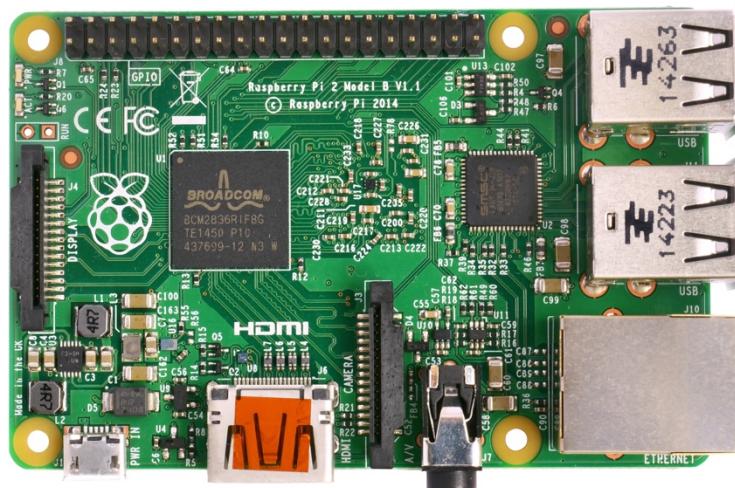


Fig. 16: Raspberry Pi 2 Model B visto dall'alto

Andando ora ad analizzarlo nello specifico, elenchiamo le numerose caratteristiche [21]:

- SoC: Broadcom BCM2836;
- CPU: Quad-core ARM Cortex-A7 900 MHz;
- GPU: Dual Core VideoCore IV® Multimedia Co-Processor, supporta OpenGL ES 2.0, OpenVG, e risoluzione massima di 1080p30;
- RAM: 1GB LPDDR2;
- Sistemi operativi supportati: basati su kernel Linux o su RISC OS, avviati dalla Micro SD;
- Dimensioni: 85 x 56 x 17 mm;
- Alimentazione: Micro USB da 5V e 2A;
- Connattività: 1 porta Ethernet 10/100 BaseT;
- Output Video: HDMI (rev 1.3 & 1.4);
- Output Audio: jack da 3.5mm, HDMI;
- USB: 4 porte USB 2.0;
- Connitori GPIO: 40-pin 2.54 mm. Trattasi di un'interfaccia che permette di collegare il Raspberry a uno o più dispositivi esterni, di input o di output;
- Connettore per camera: Camera Serial Interface (CSI-2);
- Connettore per display: Display Serial Interface (DSI);
- Slot per Memory Card: Micro SDIO.

6.3 Hardware e software aggiuntivo utilizzato

Per quanto riguarda il sistema operativo, abbiamo installato Raspbian Jessie With Pixel (la versione di gennaio 2017) [22], ovvero il sistema operativo ufficialmente supportato dalla Raspberry Pi Foundation. Abbiamo poi installato manualmente i software necessari già presentati: Mininet, Open vSwitch e Ryu.

Come anticipato, il Raspberry conterrà al suo interno sia il controller sia lo switch. Lo switch, cercando di replicare il più possibile il funzionamento di una rete reale, opererà su interfacce Ethernet reali. Avendo il Raspberry solo una porta Ethernet (che, come vedremo a breve non basta per questo lavoro), abbiamo dovuto usare degli adattatori USB-Ethernet aggiuntivi, tre, per la precisione (più tardi sarà spiegato il motivo). Si è scelto di usare gli adattatori forniti da Apple, raffigurati qui di seguito, in quanto si è riscontrata una compatibilità maggiore, con il Raspberry scelto e con Raspbian, rispetto ad altri adattatori presenti sul mercato.



Fig. 17: adattatore USB-Ethernet di Apple

CAPITOLO 7

TURNOUT CONTROLLER E WEB GUI

Iniziamo ora ad analizzare il lavoro svolto. Per prima cosa, esaminiamo il controller richiesto, e come è stato realizzato attraverso il framework di Ryu. È stato chiamato Turnout Controller in quanto si occupa, come anticipato, di ridirigere il traffico in ingresso a seconda della volontà dell'utente, espressa attraverso la Web GUI. Andiamo ora a studiare il comportamento richiesto con maggiore attenzione.

7.1 Obiettivi della tesi

Si vuole creare un controller SDN in grado di ridirezionare il traffico in ingresso da determinate porte dello switch (le chiameremo da qui in poi porte monitorate), e lasciar invece fluire, senza alcuna possibilità di ridirezione, il traffico in ingresso dalle rimanenti porte.

Per quanto riguarda il traffico che deve essere ridirezionato, si osservi la figura sottostante.

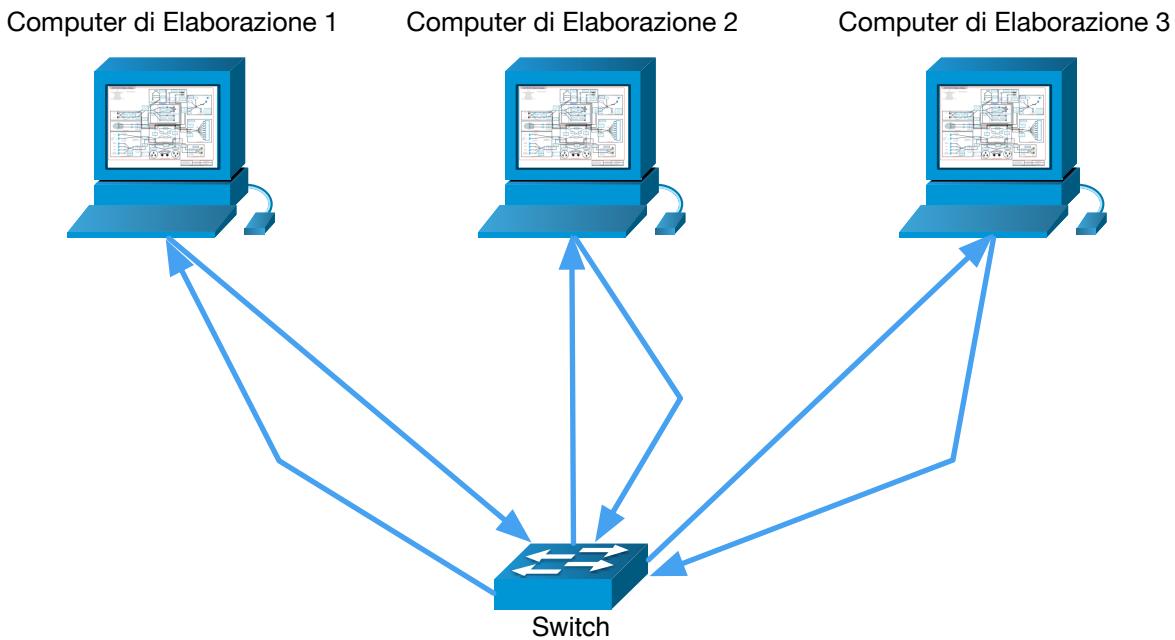


Fig. 18: schema delle connessioni tra switch e computer di elaborazione

Lo switch in figura, che è proprio lo switch che risponde al Turnout Controller, quando entra un traffico da una porta monitorata, deve poter chiedere all'utente se si vuole effettuare una ridirezione e, in quel caso, che tipo di ridirezione effettuare.

Se l'utente sceglie di non effettuare ridirezioni, il traffico segue il percorso normale, ovvero esce dalla porta dello switch a cui si affaccia il calcolatore destinatario.

Se invece l'utente sceglie di ridirezionarlo, le possibilità di ridirezione sono associate al numero di computer di elaborazione connessi allo switch: l'utente può scegliere infatti di ridirezionare il traffico verso uno o più di questi computer di elaborazione. Il traffico entrerà ed uscirà dai computer di elaborazione selezionati, uno alla volta, senza subire manipolazioni, e una volta uscito dall'ultimo computer di elaborazione selezionato, il traffico dovrà essere infine inoltrato alla sua destinazione originale.

I computer di elaborazione altro non sono che calcolatori che effettuano non meglio precise funzioni di analisi sul traffico, senza però manipolarlo: non verranno approfonditi in questa tesi. L'unico aspetto importante da notare è la presenza, su ogni computer di elaborazione, di due porte Ethernet, una da cui si riceverà il traffico dallo switch, l'altra da cui si inoltrerà il traffico ricevuto di nuovo verso lo switch (da qui la presenza di due frecce unidirezionali per ogni computer di elaborazione invece di una sola bidirezionale).

Infine, bisogna mantenere uno storico delle ridirezioni scelte dall'utente, sia per ragioni di monitoraggio delle ridirezioni attive sulla rete, sia per concedere, all'utente, la possibilità di cambiare le decisioni prese precedentemente.

7.2 Analisi dei requisiti

È possibile isolare due requisiti fondamentali:

1. Determinato traffico deve passare senza fornire la possibilità di essere ridirezionato;
2. Per quanto riguarda il restante invece, l'utente stesso dovrà decidere se e come ridirezionarlo.

La discriminante, come abbiamo visto, che ci fa distinguere tra traffico da analizzare e traffico da far passare, è la porta: se la porta fa parte di un insieme di porte che si è scelto di monitorare, il traffico farà parte del primo requisito, altrimenti del secondo. Possiamo facilmente ricavare la porta di ingresso analizzando il messaggio in transito.

Una volta conosciuta la porta di entrata del traffico, e avendo quindi discriminato che tipo di traffico è, andiamo ad analizzare le azioni che dovremo intraprendere in ogni caso.

Nel caso del primo requisito, il comportamento da ottenere è semplice: bisogna garantire, con un messaggio di tipo OFPT_FLOW_MOD, il fluire senza interruzioni del traffico, dalla porta di ingresso alla porta di uscita. Ovviamente, nel nostro caso, non conosciamo a priori i tipi di traffico che transiteranno nella nostra rete, quindi il controller dovrà lavorare in modalità reattiva, gestendo quindi in tempo reale le richieste che arriveranno.

Questo varrà anche nel caso del secondo requisito, dove però ovviamente non verranno inviati automaticamente messaggi di tipo OFPT_FLOW_MOD, ma questi ultimi dovranno attendere prima l'input da parte dell'utente, per poi essere generati rispettando quanto richiesto, e infine inviati: nello specifico, l'utente dovrà selezionare verso quali computer di elaborazione vuole ridirigere il traffico. Proprio in questo caso, si realizza inizialmente il bisogno di una REST API, che permetta all'utente, proprio attraverso una richiesta HTTP, di passare, come argomento all'API che realizzeremo, i parametri per la realizzazione di adeguati messaggi di OFPT_FLOW_MOD.

Risolti, almeno semplicisticamente, i due requisiti di base, andiamo adesso ad analizzare in dettaglio il secondo, dato che il primo sembra risolto senza il bisogno di ulteriori elucubrazioni. Difatti, nel secondo, sebbene sia stato chiarito il concetto generale di funzionamento, emerge un ulteriore dubbio: come notificare l'utente dell'ingresso nello switch di traffico da gestire? C'è bisogno di utilizzare un ulteriore componente, non citato precedentemente, in grado di aggiungere logica client-side a una pagina Web: Javascript; in particolare, nella nostra tesi, si farà largo uso di JQuery. Si può pensare, sfruttando proprio JQuery, di far apparire un pop-up all'utente, quando nuovo traffico si presenterà in ingresso allo switch: attraverso poi dei pulsanti del pop-up, l'utente sarà in grado di esprimere le proprie volontà riguardo quel traffico. Tutto questo sarà anch'esso gestito attraverso un'ulteriore REST API.

Infine, c'è il bisogno di mantenere uno storico visualizzabile e modificabile delle ridirezioni impostate (da qui in poi le chiameremo routes, rotte). Anch'esso potrà probabilmente essere realizzato attraverso JQuery e una REST API; per quanto riguarda la necessità di modificare

le rotte impostate, è possibile utilizzare un pop-up simile a quello che si utilizzerà per il secondo requisito.

Un'ultima nota riguarda i computer di elaborazione: come detto precedentemente, trattasi di calcolatori con due porte Ethernet che, dopo aver analizzato il traffico in modi ignoti, inoltra di nuovo lo stesso traffico analizzato (senza modifiche) verso lo switch. Essi rappresentano, in ultimo stadio, le destinazioni verso cui possiamo ridirezionare il traffico.

7.3 Casi d'uso

Di seguito, una serie di casi d'uso, che descrivono le funzioni che dovrà offrire il sistema che andremo a creare.

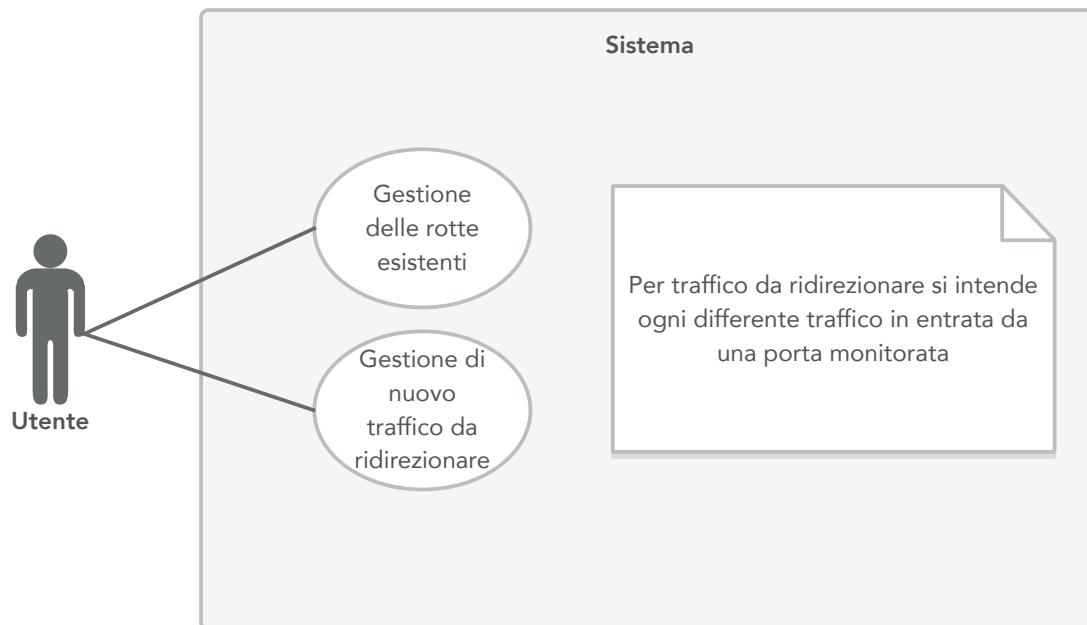


Fig. 19: funzioni offerte dal sistema all'utente

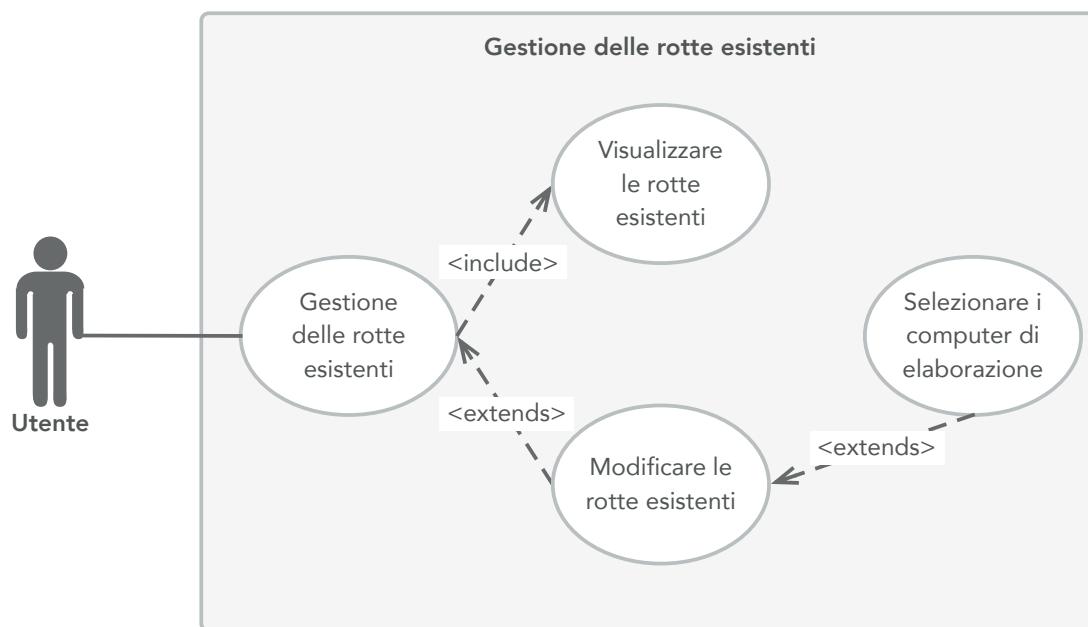


Fig. 20: casi d'uso di "Gestione delle rotte esistenti"

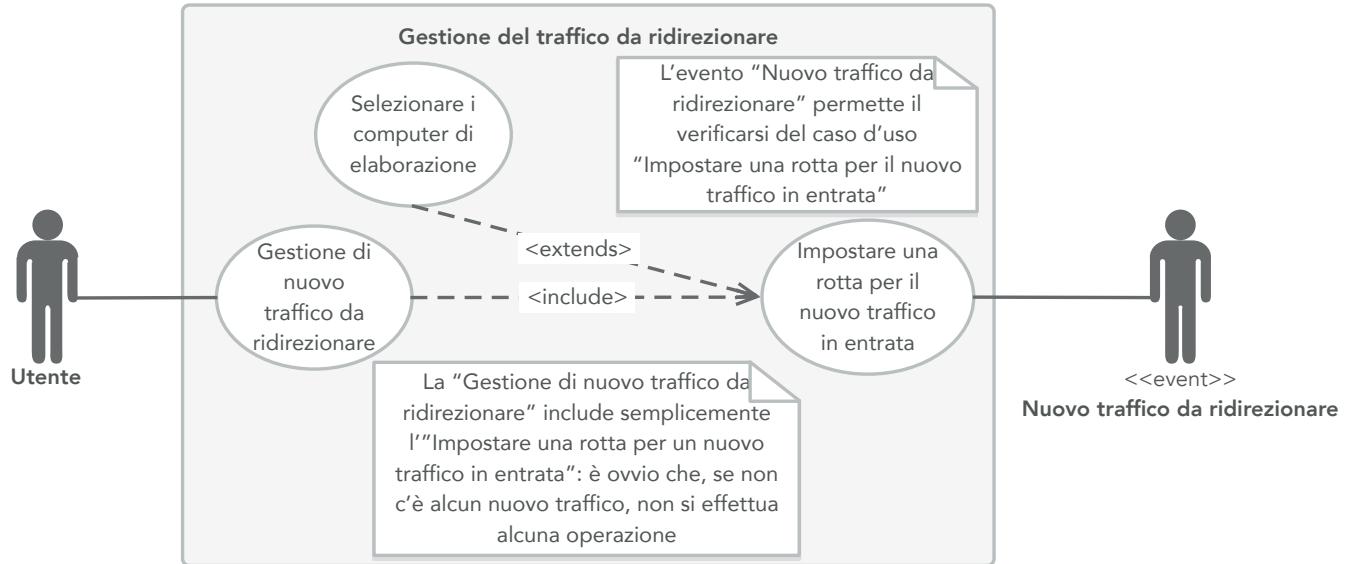


Fig. 21: casi d'uso di "Gestione di nuovo traffico da ridirezionare"

7.4 Scenari

Adesso, con gli scenari, analizziamo nel dettaglio le funzioni viste prima.

Titolo	Visualizzare le rotte esistenti
Descrizione	Visualizzare tutte le rotte esistenti attraverso il caso d'uso "Impostare una rotta per il nuovo traffico in entrata"
Relazione	
Attori	Utente
Precondizioni	
Postcondizioni	
Scenario principale	<ol style="list-style-type: none"> 1. L'utente chiede di visualizzare le rotte 2. Il sistema restituisce le rotte esistenti, visualizzandole nella pagina
Scenari alternativi	
Requisiti non funzionali	
Punti aperti	

Titolo	Modificare le rotte esistenti
Descrizione	Modificare una delle rotte esistenti, visualizzate nella pagina aperta dal caso d'uso "Gestione delle rotte esistenti"
Relazione	<pre> graph TD A((Gestione delle rotte esistenti)) -->[<extends>] B((Modificare le rotte esistenti)) A -->[<extends>] C((Selezionare i computer di elaborazione)) </pre>
Attori	Utente
Precondizioni	Almeno una rotta deve essere stata impostata nel caso d'uso "Impostare una rotta per il nuovo traffico in entrata"
Postcondizioni	<ol style="list-style-type: none"> I dettagli della rotta, presenti nella lista, sono stati modificati secondo le richieste dell'utente È stato cambiato il comportamento dello switch in modo appropriato, al fine di conformarsi alle nuove richieste dell'utente
Scenario principale	<ol style="list-style-type: none"> L'utente chiede di modificare una rotta, tra quelle visualizzate L'utente modifica la rotta, selezionando i computer di elaborazione voluti e deselectando quelli non voluti L'utente conferma le modifiche Il sistema salva le modifiche
Scenari alternativi	<ol style="list-style-type: none"> L'utente chiede di modificare una rotta, tra quelle visualizzate L'utente modifica a piacimento i dettagli della rotta L'utente scarta le modifiche Nessuna modifica viene salvata dal sistema
Requisiti non funzionali	
Punti aperti	

Titolo	Impostare una rotta per il nuovo traffico in entrata
Descrizione	Impostare una rotta per un traffico in entrata nello switch attraverso una porta monitorata
Relazione	<pre> graph LR A((Selezionare i computer di elaborazione)) -- "<extends>" --> B((Gestione di nuovo traffico da ridirezionare)) B -- "<include>" --> C((Impostare una rotta per il nuovo traffico in entrata)) </pre>
Attori	Utente
Precondizioni	Deve essersi verificato l'evento "Nuovo traffico da ridirezionare"
Postcondizioni	<ol style="list-style-type: none"> La nuova rotta è stata aggiunta alla lista delle rotte impostate È stato cambiato il comportamento dello switch in modo appropriato, al fine di conformarsi alle nuove richieste dell'utente
Scenario principale	<ol style="list-style-type: none"> L'utente specifica la rotta, selezionando i computer di elaborazione voluti L'utente chiede di salvare la rotta Il sistema salva la rotta
Scenari alternativi	
Requisiti non funzionali	
Punti aperti	

7.5 Classi di analisi

Di seguito, viene presentato il diagramma delle classi di analisi, che punta, più che altro, ad identificare, in maniera ancora abbastanza astratta, le entità in gioco e le relazioni esistenti tra di loro. Nello specifico, sono state individuate tre entità: ControllerApplication, Route e ElaborationComputer, che corrispondono al sistema nella sua totalità, alle rotte e ai computer di elaborazione.

Una scelta è sicuramente da evidenziare: la creazione dell'entità ControllerApplication, che non compare nel documento dei requisiti o nei casi d'uso.

Si è scelto infatti, invece di creare due entità, una che gestisca le rotte esistenti e una che gestisca l'arrivo di nuovo traffico, di crearne una sola, ControllerApplication, che gestisca entrambi i casi d'uso "Gestione...". Si è proceduto in questa maniera poiché le responsabilità delle due entità da gestire sarebbero state molto simili e strettamente legate: mantenendo una sola entità invece di due, si riesce a concentrare in un unico componente tutta la logica che il controller deve offrire, semplificando così il diagramma, e, probabilmente, la gestione delle funzioni da offrire. Si noti infatti, che in questo modo, l'entità che aggiunge una nuova rotta non avrà bisogno di usare l'istanza della classe che gestisce lo storico delle rotte, perché sarà direttamente lei a gestirlo.

Alcune note finali: per Datapath si intende il Datapath SDN (di cui non si sono presentati dettagli nel diagramma, sia per semplicità, sia per intuitività), mentre per Traffic sono stati presentati solamente gli attributi di sicuro utilizzo in fase di progettazione (senza chiudere la porta ad ulteriori future aggiunte, ovviamente).

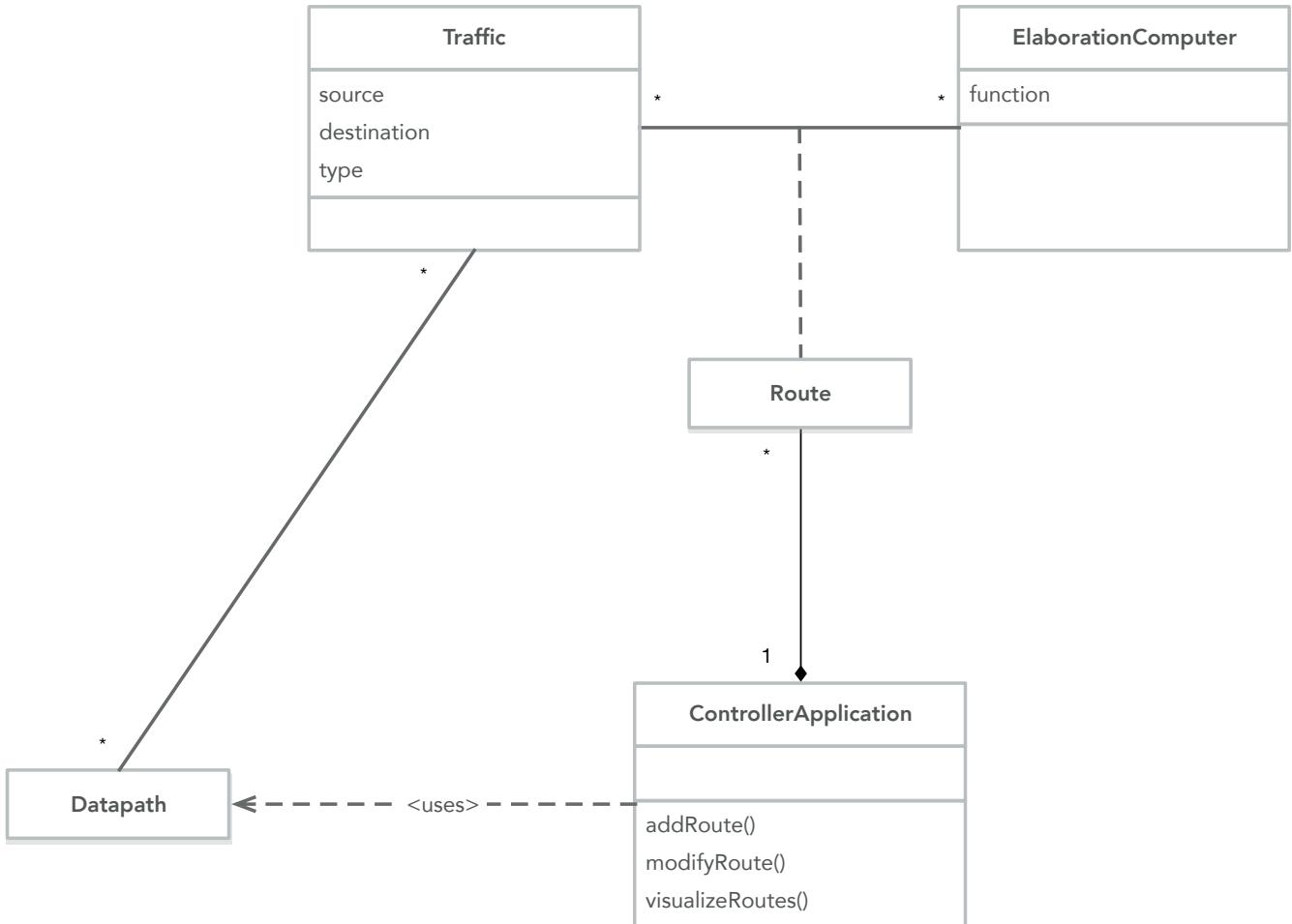


Fig. 22: diagramma delle classi di analisi

7.6 Progettazione

Ora andiamo ad introdurre la fase di progettazione. Verranno elencate alcune “traduzioni” effettuate per passare dal diagramma di analisi a quello di progettazione e alcune scelte progettuali, indipendenti dal diagramma di analisi e da quello di progettazione finale.

Discuteremo della logica di base del controller, codificata nel file turnout.py, che quindi esporrà le funzioni descritte precedentemente in analisi.

Poi, dovremo sfruttare WSGI e l’interfaccia REST di Ryu, al fine di collegare le API presentate in turnout.py alla Web GUI: verrà quindi creato il file turnout_rest.py, che realizzerà appunto il collegamento suddetto.

La Web GUI, infine, è ottenuta da due file HTML, index.html e live.html, che realizzano effettivamente la GUI dell’applicazione. Esse però sarebbero statiche, senza i file di Javascript associati, routes.js e live.js, che implementano la dinamica client-side: permettono l’invio delle richieste alla controller application e le modifiche in tempo reale alla pagina Web, quando necessario (come vedremo meglio successivamente).

Infine, per semplicità, si è proceduto ad effettuare l’implementazione della controller application nel caso in cui il controller comandi al massimo uno switch: vedremo più avanti delle considerazioni riguardo a questa scelta.

7.7 Logica di base del controller: turnout.py

Per prima cosa, codifichiamo la logica di base che dovrà essere offerta dal controller. Si ricorda che il controller scelto, Ryu, è interamente scritto in Python, e anche i suoi moduli dovranno esserlo. Il file che conterrà la logica di base sarà appunto chiamato turnout.py.

Tralasciando gli import necessari al funzionamento della logica (facilmente individuabili), andiamo subito ad analizzare la definizione di classe:

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

    def __init__(self, *args, **kwargs):
        super(Turnout, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.MONITORED_PORTS = (1,)
        self.DICTIONARY = {"QoSCheck": (3, 4)}
        self.communications = ""
        self.routes = ""
        self.my_datapath = None
```

Come è facilmente possibile individuare, spicca l'ereditarietà con la classe RyuApp presente nella componente App Manager (secondo l'architettura di Ryu precedentemente mostrata), che è necessaria, sempre secondo le specifiche per l'architettura, per creare applicazioni per il controller Ryu. OFP_VERSIONS, prima variabile che si incontra, definisce le versioni di OpenFlow che è possibile usare per il funzionamento di quest'applicazione: nel nostro caso, solo la 1.3.

Iniziamo ora ad osservare le variabili di classe dichiarate.

La prima, mac_to_port, ci servirà per implementare un'attitudine del controller che chiameremo di learning, poiché questa variabile servirà per mantenere in memoria l'associazione tra MAC e porta su cui si trova un determinato calcolatore in relazione allo switch dove transita il traffico, quando il calcolatore prova per la prima volta a comunicare, al fine di evitare flood non necessari e creare un controller che si adatti dinamicamente allo switch da supervisionare (modalità reattiva, appunto).

La seconda, MONITORED_PORTS, è una tupla che contiene tutti i numeri che OpenFlow ha associato alle porte da cui entra traffico che deve essere monitorato.

La terza, DICTIONARY, è un dizionario appunto, che associa ad una stringa, che indica la funzione di un computer di elaborazione, i numeri che OpenFlow ha associato alle porte Ethernet dello switch alle quali si affaccia il computer di elaborazione che esegue la funzione indicata dalla stringa. Si è scelto, infatti, per semplicità, visto che non si approfondiranno i computer di elaborazione, di sostituire l'entità ElaborationComputer con una stringa e con due numeri, in modo da semplificare lo sviluppo del comportamento del controller. Un'ultima nota, riguardante la stringa: ovviamente, la chiave per cercare nel dizionario sarà data dall'utente, che la selezionerà, in fase di input, tra quelle disponibili (vedremo successivamente come).

La quarta, communications, è una stringa che lista una sola volta tutti i nuovi traffici visti in ingresso allo switch controllato: tentativi di ritrasmissione di un messaggio o ulteriori messaggi dello stesso traffico non vengono aggiunti alla stringa; questo perché communications è la stringa incaricata di fornire all'utente la lista dei traffici per cui effettuare una ridirezione: se aggiungessimo a communications due volte lo stesso traffico solo perché è arrivato un altro

messaggio di quel traffico, l'utente poi (secondo il sistema che vedremo dopo) sarebbe costretto a impostare due volte la ridirezione voluta per lo stesso traffico (assolutamente errato). La penultima, routes, è la stringa incaricata di mantenere lo storico di tutte le rotte impostate dall'inizio dell'attività del controller: come vedremo poi, quando sarà impostata o modificata una rotta, questa sarà la variabile che andremo a manipolare.

L'ultima, my_datapath, è la variabile forse più importante: mantiene un riferimento fisso allo switch che è sotto la giurisdizione del controller, al fine di poter modificare le rotte impostate successivamente (vedremo poi perché è necessario mantenere questo riferimento). Ovviamente, qui è impostata a None; alla prima occasione utile, sarà impostata correttamente. Passiamo ora ad analizzare la gestione dell'evento associato alla fase di scambio dei messaggi di Feature.

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

    self.my_datapath = ev.msg.datapath

    datapath = self.my_datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

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

    datapath.send_msg(mod)

    match = parser.OFPMatch()

    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]

    inst =
[parser.OFPIInstructionActions(ofproto.OFPI_APPLY_ACTIONS, actions)]

    mod = parser.OFPFlowMod(datapath=datapath, priority=0,
match=match, instructions=inst)

    datapath.send_msg(mod)
```

Come già anticipato nel capitolo riguardante Ryu, l'API che ci permette di dichiarare un event handler per uno scambio simmetrico o asimmetrico di messaggi tra controller e switch è set_ev_cls [23]. In particolare, esso, è un decoratore, che decora appunto il metodo che dovrà gestire l'evento citato in set_ev_cls. Quest'ultima infatti, prende come argomenti: il tipo di evento gestito (in questo caso, EventOFPSwitchFeatures, che indica l'arrivo del messaggio di Feature Reply) e lo stato della fase di negoziazione al momento del verificarsi dell'evento (in questo caso, CONFIG_DISPATCHER, che indica la fase in cui il controller è in attesa del messaggio di Feature Reply). Se questi due argomenti si verificano nello stesso tempo, l'evento viene gestito, altrimenti, viene ignorato. È inoltre da notarsi che, se un certo evento non viene

gestito, Ryu provvede a gestirlo automaticamente: non bisogna implementare tutti i possibili eventi esistenti quando si crea un'applicazione.

Ora, entriamo nel codice dell'event handler: per semplicità, dividiamolo in due parti: la riga che invia il primo messaggio (il primo `datapath.send_msg(mod)`) è la riga separatrice; iniziamo quindi ad analizzare la prima parte.

Riprendendo la nostra variabile di classe `my_datapath`, assegniamole il valore dell'unico switch che andremo a controllare (secondo la semplificazione fatta di base), per poterla usare più tardi. Da notarsi che esiste l'istanza `ev`, che identifica l'evento verificatosi: da esso possiamo ricavare tutte le informazioni necessarie all'esecuzione dell'event handler.

Successivamente, creo tre variabili: `datapath`, a cui assegno l'istanza del datapath per cui si è verificato l'evento, `ofproto`, a cui assegno un'altra istanza che contiene le componenti di base di OpenFlow, `parser`, a cui assegno un'ultima istanza che contiene il decoder e l'encoder di OpenFlow (ci si riferisce sempre alla versione 1.3).

Attraverso questi tre strumenti, procedo a creare un messaggio di tipo `FlowMod` (attraverso il costruttore `parser.OFPFlowMod`) che assegno alla variabile `mod`: basta assegnare come `datapath` il datapath considerato, come `command` `ofproto.OFPC_DELETE` (che indica la cancellazione di tutte le regole presenti sul datapath, dato che non è fornito un filtraggio, come invece vedremo in seguito), come porte di uscita bersaglio (`out_port`) `OFPP_ANY` e come gruppi bersaglio (`out_group`) `OFPP_ANY`; infatti, quando viene inviato un messaggio di tipo `FlowMod` con comando `DELETE` o `DELETE_STRICT`, essa bersagliera solo le flow entry con gli `out_group` e `out_port` specificati, quindi bisogna impostare `OFPP_ANY` per essere sicuri di bersagliare tutti.

Infine, attraverso un metodo del datapath, si invia il messaggio.

Questa prima sezione serve per assicurarsi che, qualunque siano le flow entry attualmente presenti sugli switch, quando il controller si riavvia, anch'essi in qualche modo fanno lo stesso, tornando alla situazione di partenza per quanto riguarda le flow entry presenti: è molto utile in contesti di sperimentazione come il nostro, dove molti test sono stati eseguiti per raffinare il comportamento del controller.

Andiamo ora alla seconda sezione.

Per prima cosa, vediamo che viene assegnata alla variabile `match` un'istanza di `OFPMatch` costruita senza parametri. Questo ci garantisce che il messaggio di `FlowMod` che stiamo per inviare avrà effetto su tutti i prossimi traffici che passeranno per lo switch: se avessimo voluto filtrare, avremmo dovuto passare degli argomenti durante la creazione dell'istanza.

Poi, assegniamo alla variabile `actions`, attraverso il costruttore `OFFActionOutput` di `parser`, un'istanza che indica che tutti i successivi pacchetti che transiteranno dovranno essere mandati al controller (`ofproto.OFPP_CONTROLLER`) interamente (`ofproto.OFPCML_NO_BUFFER`), senza occupare spazio per i pacchetti sullo switch e dover rintracciare poi i messaggi attraverso il `buffer_id` (che è un identificativo di una zona di memoria dello switch che può contenere parte del pacchetto transitato).

Infine, assegniamo alla variabile `inst`, un'istanza di `OFPIInstructionActions`, che comunichi al datapath di applicare immediatamente (`ofproto.OFPI_APPLY_ACTIONS`) le azioni indicate (`actions`).

Si costruisce quindi il messaggio, che viene assegnato poi a `mod`, in modo simile a quanto avvenuto precedentemente: qui però viene settata la priorità, 0, il `match`, con la variabile `match`, e le istruzioni, con la variabile `inst`. In questo modo, si bersagliano tutti i pacchetti di qualunque traffico in ingresso (poiché l'istanza di `match` è vuota) che non abbiano flow entry di più alta priorità con cui fanno match.

Infine, viene inviato anche questo messaggio al datapath suddetto. Ci si è assicurati, attraverso quest'ultimo messaggio, che tutti i pacchetti per cui non esista una flow entry associata vengano mandati, nella loro interezza, al controller.

Ora, analizziamo il secondo e ultimo evento gestito.

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

    msg = ev.msg
    in_port = msg.match['in_port']
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    dpid = datapath.id
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    src = eth.src
    dst = eth.dst
    protocol = self.getProtocol(pkt)

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

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD
    actions = [parser.OFPActionOutput(out_port)]

    key = "%s %s %s" % (src, dst, protocol)

    if in_port not in self.MONITORED_PORTS:
        if self.filtered_ip(dst, eth):
            self.accept(msg)
        else:
            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)
    else:
        if self.communications.find(key) < 0 and
self.filtered_ip(dst, eth):

            self.communications += str(src)
            self.communications += ' '
```

```

        self.communications += str(dst)
        self.communications += ' '
        self.communications += str(protocol)
        self.communications += '\n'

    if not self.filtered_ip(dst, eth):
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

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

```

In questo caso, l'evento che si vuole gestire è quello di `Packet_In` (`EventOFPPacketIn`) che avviene quando la connessione tra controller e switch è stata stabilita con successo (`MAIN_DISPATCHER`).

Anche in questo caso, ricaviamo il messaggio dall'evento, e dal messaggio ricavato otteniamo una serie di istanze che ci saranno utili a breve; ne presentiamo alcune che non erano presenti nel precedente event handler.

La variabile `dpid`, ad esempio, conterrà l'id del datapath per un uso successivo (lo vedremo a breve).

La variabile `pkt`, invece, conterrà l'istanza del pacchetto ricavato dai dati del messaggio attraverso il costruttore `packet.Packet(msg.data)`. Procediamo analizzando il pacchetto, passando al metodo `get_protocol` la classe `Ethernet`, ottenendo così l'header `Ethernet` del pacchetto stesso. Da questo, otteniamo la sorgente e la destinazione del pacchetto, e infine, attraverso la funzione di classe `getProtocol`, a cui passiamo `pkt`, troviamo il protocollo specifico del pacchetto.

Trovate tutte queste informazioni, addentriamoci ora nel cuore dell'event handler.

Attraverso il metodo `set_default`, inizializziamo come valore associato alla chiave `dpid` un dizionario vuoto (a meno che un valore non sia già stato impostato) e poi associamo alla chiave `src` il valore di `in_port`: tutto questo al fine di realizzare il comportamento di learning del controller, che in questo modo tiene in memoria le associazioni, per ogni datapath, di MAC e porta, allo scopo di evitare, se possibile, di fare flood per raggiungere la destinazione richiesta. Cosa che può avvenire proprio subito dopo, quando cerchiamo il destinatario del pacchetto in `mac_to_port`: se lo si trova, si assegna ad `out_port` la porta associata, altrimenti, si assegna il flood (`OFPP_FLOOD`). A questo punto, similmente a come fatto precedentemente, si assegna alla variabile `actions`, attraverso il costruttore `OFPActionOutput` di `parser`, un'istanza che indica che l'azione da effettuare (non per tutti i traffici; per quale specifico traffico lo vedremo dopo) è far fuoriuscire il traffico dalla `out_port` impostata precedentemente.

A questo punto, creiamo una stringa `key` per un uso prossimo.

Adesso, si arriva alla logica centrale di questo event handler.

Inizialmente, si guarda se la porta da cui entra il traffico è una di quelle da monitorare: da qui distinguiamo due casi.

Se la porta non è da monitorare, abbiamo due possibilità. Se il traffico in ingresso viene filtrato con successo (`self.filtered_ip(dst, eth)` risulta vero), e quindi non è uno di quelli che potremmo chiamare di Discovery (LLDP, ARP o simili), allora accettiamo direttamente il traffico, inviando un messaggio di `FlowMod` in modo simile a quanto fatto precedentemente attraverso la

funzione di classe accept, a cui passiamo come argomento il messaggio da cui ricavare le informazioni necessarie. Se invece il traffico è uno di quelli di Discovery, allora prima si imposta la variabile data con il contenuto del messaggio, se è stato effettivamente inviato al controller, ovvero se `msg.buffer_id == ofproto.OFP_NO_BUFFER` è vero, altrimenti si lascia data impostata a None; a quel punto, si costruisce un'istanza di `OFPPacketOut` con le informazioni raccolte e la si invia al datapath, per istruirlo di emettere quel messaggio dalla porta di uscita indicata in actions.

Anche se la porta è quella da monitorare, abbiamo sempre due possibilità. Se il traffico in ingresso viene filtrato con successo, e, inoltre, quel traffico non è già stato visto precedentemente (`self.communications.find(key) < 0` risulta vero), allora si aggiunge una riga nella stringa `communications` per questo traffico. Invece, se semplicemente il traffico è uno di quelli di Discovery, si applica lo stesso algoritmo visto precedentemente per la stessa condizione.

Una nota finale, prima di passare agli ultimi metodi importanti della classe, riguarda la decisione di gestire i traffici di Discovery attraverso dei `PacketOut`, invece che magari con dei messaggi di `FlowMod`. Tale scelta ci è data dalla natura intrinseca dei traffici di Discovery, che solitamente si verificano prima dell'inizio del passaggio di traffici legati a livelli applicativi dello stack di rete, e quindi prima del passaggio del traffico che si mira realmente a far circolare in rete: date le premesse, si è scelto di non impostare con messaggi di `FlowMod` la circolazione di questi messaggi, ma con dei `PacketOut`. Utilizziamo quindi i traffici di Discovery per far apprendere in questo modo al controller tutte le associazioni dello switch tra MAC e porta, dato che ad ogni messaggio di questo tipo si attiverà l'event handler, visto che non c'è una flow entry che gestisca quel traffico; invece di impostare un messaggio di `FlowMod` per permettere il passaggio di questi traffici, che porterebbe probabilmente, per qualunque nuovo messaggio di alcuni di quei traffici, a fare flood indistintamente. Contestualmente, in questo modo si è quasi sicuri del fatto che ogni traffico temporalmente successivo abbia come destinazione un calcolatore affacciato ad una porta nota, evitando quindi anche in questo caso flood.

Ora, passiamo ai principali metodi offerti, le REST API di cui abbiamo parlato precedentemente. Vediamo le prime due insieme, dato che sono molto semplici e corte.

```
def list_communications(self):
    actual = self.communications[:self.communications.find('\n')]
    self.communications =
self.communications[self.communications.find('\n') + 1:]

    return actual

def list_routes(self):
    return self.routes
```

La prima, `list_communications`, restituisce una stringa contenente tutti i nuovi traffici per cui l'utente deve ancora impostare una rotta. Prima però, si occupa di togliere dalla stringa `communications` la prima riga, poiché, data che c'è stata la richiesta di visualizzare i traffici per cui ancora non c'è una rotta, questo implica che la prima riga (presente in `actual` che viene

restituito), rappresenta un traffico per cui verrà subito impostata la rotta (come vedremo meglio in seguito).

La seconda, `list_routes`, semplicemente si limita a restituire una stringa, `routes`, contenente tutte le rotte impostate.

Andiamo ora con il terzo e ultimo metodo. È molto lungo e macchinoso, quindi lo divideremo in pezzi.

```
def set_route(self, serialized_form):

    original_serialized_form = serialized_form

    requestedFunctions = []

    src = serialized_form[:serialized_form.find(" ")]

    serialized_form = serialized_form[(serialized_form.find(" ") + 1):]

    dst = serialized_form[:serialized_form.find(" ")]

    serialized_form = serialized_form[(serialized_form.find(" ") + 1):]

    proto = serialized_form[:serialized_form.find(" ")]

    if serialized_form[serialized_form.find(proto) + len(proto)] != '\n':
        serialized_form = serialized_form[(serialized_form.find(" ") + 1):]

    while serialized_form.find("&") >= 0:

        requestedFunctions.append(serialized_form[serialized_form.find("function=") + 9: serialized_form.find("&")])

        serialized_form = serialized_form[serialized_form.find("&") + 1:]

        if serialized_form:
            requestedFunctions.append(serialized_form[serialized_form.find("function=") + 9:])

    key = src + " " + dst + " " + proto + " "
```

Per prima cosa, possiamo notare che abbiamo un argomento, oltre a `self`, che sarà l'input dell'utente: questo input è formattato sempre in un determinato modo. Questo ci permette di effettuare agevolmente il parsing della stringa e ottenere la sorgente (`src`), la destinazione (`dst`) e il protocollo del traffico (`proto`), e poi, attraverso ulteriori controlli e manipolazioni, trovare anche le funzioni richieste dall'utente (`requestedFunctions`, quelle effettuate dai computer di elaborazione). Infine, creiamo una stringa `key`, per agevolare alcune funzioni successive.

```
if self.isModified(original_serialized_form):
```

```

        activatedFunctions = []
        routeFunctions = self.routes[self.routes.find(key) +
len(key):self.routes.find("\n", self.routes.find(key))]

        while routeFunctions.find(",") >= 0:

activatedFunctions.append(routeFunctions[:routeFunctions.find(",")])
        routeFunctions =
routeFunctions[routeFunctions.find(",") + 2:]
        if routeFunctions:
            activatedFunctions.append(routeFunctions)

differences = False

for fun in activatedFunctions:
    if fun not in requestedFunctions:
        differences = True
        break

if not differences:
    for fun in requestedFunctions:
        if fun not in activatedFunctions:
            differences = True
            break

if differences:

    self.modifyFunctions(1, src, dst, proto,
activatedFunctions)
    self.modifyFunctions(0, src, dst, proto,
requestedFunctions)

    firstroutes = self.routes[:self.routes.find(src + " "
+ dst + " " + proto)]
    lastroutes = self.routes[self.routes.find("\n",
self.routes.find(src + " " + dst + " " + proto)):]
    self.routes = firstroutes + lastroutes

    self.routes += src + " " + dst + " " + proto + " "
    for fun in requestedFunctions:
        self.routes += fun + ", "
    if len(requestedFunctions) > 0:
        self.routes = self.routes[:-2] + "\n"
    else:
        self.routes = self.routes[:-1] + " " + '\n'

```

Viene utilizzato ora il metodo di classe `isModified` (con argomento l'input dell'utente), per venire a conoscenza se una rotta per quel traffico è stata già impostata.
Analizziamo ora il caso in cui la condizione sia vera.

Anche qui, ci sono numerosi controlli e manipolazioni, al fine di ottenere le funzioni già attivate per questo traffico (`activatedFunctions`).

A questo punto, si controlla se ogni funzione attivata è anche stata richiesta e viceversa, per riuscire a determinare se ci sono delle differenze, rispetto alla rotta già impostata (un utente potrebbe aver inviato per sbaglio in input la richiesta di modifica di una rotta richiedendo però solo funzioni già attive, ad esempio). Non appena si trova una differenza, si interrompe il controllo e si entra nella condizione (`if differences`): qui sostanzialmente avvengono tre operazioni.

1. Tutte le funzioni attive vengono disabilitate, attraverso il metodo di classe `modifyFunctions`, che prende in ingresso la modalità di funzionamento, `mode` (un intero, 0 o 1, che indica se le funzioni che vengono passate come argomento devono essere attivate o disabilitate) = 1, la sorgente, la destinazione, il protocollo e le funzioni;
2. Tutte le funzioni richieste vengono abilitate, sempre attraverso il metodo di classe `modifyFunctions` con `mode` = 0;
3. Attraverso alcune manipolazioni su `routes`, viene tolta la riga che indicava la rotta da modificare, e viene aggiunta in coda la rotta modificata.

Più tardi analizzeremo `modifyFunctions`, ora analizziamo l'ultima sezione di codice.

```
else:  
    self.modifyFunctions(0, src, dst, proto,  
requestedFunctions)  
  
    self.routes += src + " " + dst + " " + proto + " "  
    for fun in requestedFunctions:  
        self.routes += fun + ", "  
    if len(requestedFunctions) > 0:  
        self.routes = self.routes[:-2] + "\n"  
    else:  
        self.routes = self.routes[:-1] + " " + "\n"
```

Ci eravamo lasciati alla verifica della presenza o meno, attraverso il metodo di classe `isModified`, di una rotta per il traffico per cui ne stiamo appunto impostando una.

Prima, abbiamo analizzato il caso in cui la condizione fosse verificata, ora, osserviamo quando invece la condizione è falsa.

In questo caso, attraverso il metodo di classe `modifyFunctions`, come sopra, attiviamo le funzioni richieste. Poi, viene aggiunta in coda la rotta appena impostata.

Ovviamente, la disabilitazione delle funzioni precedenti e la rimozione della riga in `routes` non è necessaria, perché non è presente alcuna rotta precedente per il traffico in questione.

Terminiamo ora l'analisi di `turnout.py` osservando la funzione principale del controller, che realizza il percorso di ridirezione come richiesto: `modifyFunctions`.

```
def modifyFunctions(self, mode, src, dst, proto, functions):  
  
    dpid = self.my_datapath.id  
    ofproto = self.my_datapath.ofproto  
    parser = self.my_datapath.ofproto_parser  
    in_ports = []  
    out_ports = []
```

```

in_ports.append(self.mac_to_port[dpid][src])

for fun in functions:
    out_ports.append(self.DICTIONARY[fun][0])
    in_ports.append(self.DICTIONARY[fun][1])

if dst in self.mac_to_port[dpid]:
    out_ports.append(self.mac_to_port[dpid][dst])
else:
    out_ports.append(ofproto.OFPP_FLOOD)

for index in range(len(in_ports)):

    match = self.getMatchString(proto, parser,
in_ports[index], src, dst)

    if mode == 0:

        actions = [parser.OFPActionOutput(out_ports[index])]

        inst =
[parsed.OFPIInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                actions)]
        mod = parsed.OFPFlowMod(datapath=self.my_datapath,
priority=1, match=match, instructions=inst)

        if mode == 1:
            mod = parsed.OFPFlowMod(datapath=self.my_datapath,
command=ofproto.OFPFC_DELETE, match=match, out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY)

        self.my_datapath.send_msg(mod)

```

Incominciamo memorizzando le informazioni che ci serviranno poi: l'id del datapath (in dpid), ofproto e parser (gli stessi visti precedentemente).

Poi, inizializziamo le porte di uscita e di entrata (out_ports e in_ports), e aggiungiamo ad in_ports, come porta iniziale di entrata, quella a cui si affaccia il calcolatore da cui parte il traffico (ovviamente). Poi, aggiungiamo, per ogni funzione, la porta di uscita dello switch, a cui si affaccia il calcolatore che esegue quella funzione, ad out_ports, e la porta di entrata dello switch, sempre del calcolatore suddetto, ad in_ports. Infine, aggiungiamo, ad out_ports, la porta a cui si affaccia il calcolatore destinatario della comunicazione (tutto questo procedimento diverrà chiaro a breve).

Ora, per ogni funzione richiesta, creiamo un'istanza di OFPMatch attraverso il metodo di classe getMatchString, che ci permette di creare un match ad hoc per il traffico passante; poi, se mode = 0 (quindi bisogna attivare le funzioni), creiamo istanze di OFPActionOutput, OFPIInstructionActions e OFPFlowMod come abbiamo potuto vedere precedentemente (stando attenti ad impostare la priority a 1, maggiore di quella della flow entry che manda tutto il traffico al controller), se invece mode = 1, semplicemente creiamo un'istanza di OFPFlowMod con

command OFPFC_DELETE similmente a quanto visto precedentemente (ricordarsi qui di aggiungere match, per assicurarsi di bersagliare solo il traffico voluto).

Infine, prima di ripartire col ciclo (se non si è giunti al termine), inviare il messaggio di FlowMod. In questo modo, si è ridirezionato quello specifico traffico in ingresso, tante volte quante sono le funzioni attivate; infatti, il traffico che inizialmente doveva andare verso la porta dello switch a cui si affaccia il calcolatore destinatario, viene ridirezionato verso la porta di ingresso del calcolatore che esegue quella funzione. Poi, visto che ci si aspetta che quel traffico torni, identico, attraverso la porta di uscita di quello stesso calcolatore, si ridireziona quel traffico in entrata dalla porta dello switch che corrisponde a quella di uscita di quel calcolatore, verso il successivo calcolatore, in modo simile a quanto visto prima, oppure verso la destinazione finale, se tutte le funzioni sono state attivate.

7.8 WSGI e OF_REST: turnout_rest.py

Ora, dobbiamo sfruttare WSGI e l'interfaccia REST di Ryu (OF_REST) come anticipato, al fine di collegare la Web GUI alle API della controller application, e realizzare così delle REST API. Andiamo quindi, subito ad analizzare il codice del file turnout_rest.py.

Questo file contiene due classi: TurnoutController e TurnoutRestApi. Per prima, analizziamo TurnoutRestApi, dividendola in sezioni.

```
class TurnoutRestApi(turnout.Turnout):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    _CONTEXTS = {
        'wsgi': WSGIApplication,
    }
```

Come si può vedere dalla dichiarazione della classe, questa eredita la classe Turnout definita precedentemente, del modulo turnout (che corrisponde al nome del file che la contiene senza l'estensione), per un uso che vedremo a breve.

Poi, similmente a quanto visto per turnout.py, definisce OFP_VERSIONS, che indica le versioni di OpenFlow che è possibile usare per il funzionamento di questa applicazione.

A questo punto, viene definita la variable _CONTEXT, per specificare la classe del Web Server di Ryu compatibile con WSGI. Attraverso quest'assegnamento, potremo successivamente ricavare l'istanza del Web Server da una chiave chiamata wsgi (lo vedremo a breve).

```
def __init__(self, *args, **kwargs):
    super(TurnoutRestApi, self).__init__(*args, **kwargs)
    wsgi = kwargs['wsgi']
    wsgi.register(TurnoutController,
                  {turnout_instance_name: self})
    mapper = wsgi.mapper

    mapper.connect('routes', '/rest/routes',
controller=TurnoutController, action='list_routes',
conditions=dict(method=['GET']))
```

```

    mapper.connect('routes', '/rest/communications',
controller=TurnoutController, action='list_communications',
conditions=dict(method=['GET']))

    mapper.connect('routes', '/rest/set_route',
controller=TurnoutController, action='set_route',
conditions=dict(method=['POST']))

```

Troviamo ora il metodo di inizializzazione classico, su cui non ci dilungheremo, e il corpo del metodo.

La prima operazione che troviamo riguarda proprio quanto anticipato poco sopra: l'ottenimento dell'istanza di WSGIApplication attraverso la key chiamata wsgi, e l'assegnamento della prima alla variabile wsgi. Attraverso wsgi poi, procediamo a registrare la classe controller TurnoutController (che vedremo poi), associandole attraverso un dizionario, l'istanza di TurnoutRestApi alla chiave turnout_instance_name, che è una variabile globale che riporta il nome (può essere qualunque) dell'istanza di TurnoutRestApi.

A questo punto, ottenuto il mapper, attraverso le connect possiamo mappare a degli URL prestabiliti i servizi da offrire. Analizziamo la prima connect, tutte le altre funzionano allo stesso modo.

Il primo argomento, la stringa routes, indica un legame che si viene a creare alla fine dell'esecuzione del metodo, per cui url("routes") restituirà /rest/routes o /rest/communications o /rest/set_route indistintamente dal tipo di richiesta fatta (difatti, essendo inutilizzato, è lasciato sempre route).

Il secondo argomento, appunto, indica l'URL a cui deve essere fatta la richiesta per far partire la gestione della stessa da parte del TurnoutController, indicato come terzo argomento.

Il quarto argomento, ci indica il nome esatto del metodo di TurnoutController da attivare per la gestione della richiesta.

Il quinto infine, pone una condizione per l'attivazione per la richiesta, ad esempio, nel primo caso, è necessario che la richiesta sia GET per far partire la gestione dell'evento.

Allo stesso modo, funzionano le altre connect.

Passiamo ora alla presentazione dell'altra classe: TurnoutController.

```

class TurnoutController(ControllerBase):
    def __init__(self, req, link, data, **config):
        super(TurnoutController, self).__init__(req, link, data,
**config)
        self.turnout_app = data[turnout_instance_name]

    def list_communications(self, req, **_kwargs):
        body = self.turnout_app.list_communications()
        return Response(content_type='text/html', body=body)

    def list_routes(self, req, **_kwargs):
        body = self.turnout_app.list_routes()
        return Response(content_type='text/html', body=body)

    def set_route(self, req, **_kwargs):
        self.turnout_app.set_route(req.body)
        return Response(status=200)

```

La classe, come si vede, eredita ControllerBase (bisogna farlo, per classi come TurnoutController): questo implica la presenza nel metodo di inizializzazione delle variabili req, link, data e **config (utilizzeremo in realtà solo data, come è possibile vedere).

Assegniamo alla variabile di classe turnout_app l'istanza di TurnoutRestApi ricavandola da data attraverso la chiave turnout_instance_name: possiamo farlo grazie al metodo register della classe precedente.

A questo punto, analizziamo i metodi offerti, che terminano il nostro processo di creazione delle REST API: ogni metodo presenta gli stessi argomenti. Anche qui, utilizzeremo solo req, e, tra l'altro, solo in set_route: è una variabile che rappresenta (nelle richieste POST) il body della richiesta dell'utente. A parte le prime due funzioni, che si limitano a richiamare i metodi presenti nella classe Turnout, tramite il riferimento suddetto, e a restituire una pagina Web di risposta il cui body è dato dal risultato del metodo appena chiamato, l'ultimo metodo, set_route, chiama anch'esso un metodo attraverso il body della richiesta inviato dall'utente, e infine, risponde con una pagina Web con status 200, per indicare che la richiesta è stata eseguita con successo.

7.9 Realizzazione del server: my_fileserver.py

A questo punto, manca solo da realizzare il server. Si può pensare di usare una strategia simile a quanto appena visto con turnout_rest.py: difatti, questa sarà l'idea di base di my_fileserver.py. Avremo quindi anche qui due classi: WebRestApi e WebController. Partiamo dalla prima.

```
class WebRestApi(app_maager.RyuApp):
    _CONTEXTS = {'wsgi': WSGIApplication}

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

        wsgi = kwargs['wsgi']

        mapper = wsgi.mapper

        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']))
```

La realizzazione è estremamente simile a quella di TurnoutRestApi, quindi non ci dilungheremo in spiegazioni ridondanti. Da notarsi, però, il secondo argomento della prima connect, che è una forma di scrittura che ci permette di isolare il nome del file richiesto e usarlo successivamente (vedremo poi come).

```
class WebController(ControllerBase):
    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')

    def make_response(self, filename):
        filetype, encoding = mimetypes.guess_type(filename)

        if filetype is None:
            filetype = 'application/octet-stream'

        res = Response(content_type=filetype)

        res.body = open(filename, 'rb').read()
        return res

    def get_root(self, req, **kwargs):
        return self.get_file(req, None)

    def get_file(self, req, filename, **kwargs):
        if (filename == "" or filename is None):
            filename = "index.html"
        try:
            filename = os.path.join(self.directory, filename)
            return self.make_response(filename)
        except IOError:
            return Response(status=400)

```

Anche qui, inizialmente, si ha la stessa realizzazione di TurnoutController.

Partendo dai metodi enunciati nelle connect precedenti, analizziamo get_root e get_file.

La prima, get_root, semplicemente invoca il metodo di classe get_file con argomenti req e None.

La seconda, get_file, contiene infatti come argomento il filename, come dicevamo precedentemente. Se il filename è vuoto o è None, cosa comune quando si accede a un sito Web poiché significa che desideriamo accedere alla home page di quel sito, si imposta come filename proprio index.html. A questo punto, si entra in un blocco controllato, dove si trova il percorso completo del file e lo si passa al metodo di classe make_response; se c'è un errore, si restituisce una pagina con status 400 (per indicare l'errore). Da notarsi che il percorso completo del file lo si trova usando il metodo join di os.path, eseguito con una variabile di classe, directory, e filename, che conosciamo; directory è semplicemente il percorso assoluto di sistema che conduce alla cartella web, che si trova nella cartella in cui si trova my_fileserver.py, in questo caso. Risulta chiaro, quindi, che il nostro server lavora con file presi da locale.

In make_response, si trova il tipo di file e l'encoding (o se non si trova, viene assegnato il tipo di file più generico possibile), a quel punto, si crea una risposta (res) che ha come tipo di contenuto il tipo di file appena trovato. Poi, si apre filename (che si ricorda, contiene il percorso completo al file) e lo si legge, riversandolo sul body della risposta. A quel punto, viene restituito res.

7.10 Web GUI: pagine HTML e file Javascript

Ora, rimane solamente da analizzare la GUI realizzata per il nostro server, ovvero l'interfaccia offerta all'utente e che `my_fileserver.py` caricherà dinamicamente in base alle richieste dell'utente nel browser. Non ci dilungheremo sulla struttura delle pagine HTML, ma più tardi verranno illustrate delle immagini per mostrare la GUI finale. Ora concentriamoci sulla logica client-side realizzata dai due file js, `routes.js` e `live.js`. Partiamo da `live.js`, in quanto più semplice.

```
setInterval(getFirstCommunication, 2000);
```

Innanzitutto, è da dire che `live.js` definisce la logica client-side della pagina `live.html`, che vedremo dopo.

Il main di `live.js` è composto semplicemente dalla riga sopra, in cui si definisce l'esecuzione di `getFirstCommunication` ogni 2 secondi (che esamineremo immediatamente sotto).

```
function chooseFunctions() {
    $dialog.html(createDialogText(src, dst, proto));
    $dialog.dialog('open');
}

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

        if (data)
        {
            src = data.split('\n')[0].split(' ')[0];
            dst = data.split('\n')[0].split(' ')[1];
            proto = data.split('\n')[0].split(' ')[2];
            chooseFunctions();
        }
        else
        {
            console.log("Requested page is empty");
        }
    }).fail(function(data) {
        console.log("Failed sending request");
    });
}
```

La funzione `getFirstCommunication` si occupa di leggere, dall'url dove `turnout_rest.py` scrive i traffici per cui impostare la rotta (come visto precedentemente), il primo traffico da servire, realizzando così una logica FIFO. Se la richiesta ha successo (ovvero la pagina esiste, e questo vuol dire che il controller è attivo e in ascolto) e la pagina non è vuota, si legge l'intero body della pagina richiesta, e, attraverso varie manipolazioni, si trova la sorgente (src), la destinazione (dst) e il protocollo (proto) del traffico alla prima riga del body. A questo punto si avvia la funzione `chooseFunctions`.

La funzione chooseFunctions si occupa di creare dinamicamente il testo, da immettere nel dialog che apriremo per l'utente, contenente i dettagli del traffico; lo fa attraverso la funzione createDialogText. Una volta immesso il testo, viene aperto il dialog.

L'url di base (la variabile url), ovvero quello da dove è in ascolto la WSGI Application, viene settato nel file utils.js (inclusa in entrambe le pagine HTML), e corrisponde a <http://localhost:8080/>, poiché il controller viene eseguito in locale (e quindi anche la WSGI Application, di default) e la porta di default per le WSGI Application è la 8080.

Il dialog è così formato:

```
var $dialog = $('

</div>)')
  .html('')
  .dialog({
    closeOnEscape: false,
    open: function(event, ui) { $(".ui-dialog-titlebar-
close").hide(); },
    autoOpen: false,
    title: 'Traffic Alert',
    buttons: {
      "Ok": function() {
        var request_body = src + " " + dst + " " + proto + " "
        $('#myform').serialize();
        $.post(url.concat("rest/set_route"), request_body);
        $dialog.dialog('close');
      }
    }
});


```

Come è possibile immediatamente notare, al suo interno è vuoto (html('')), verrà impostato successivamente con createDialogText, non è possibile chiuderlo in alcun modo (eccetto cambiando pagina; è stata fatta questa scelta per assicurarsi che l'utente, quando spunta il dialog, sia costretto a impostare la rotta) e, quando viene cliccato il pulsante Ok, invii, attraverso richiesta POST alla pagina rest/set_route, un body di solo testo dove compaiono i dettagli del traffico e il form serializzato, così che la controller application abbia tutte le informazioni necessarie all'impostazione della rotta. Infine, viene chiuso il dialog.

È necessaria ora una digressione, senza mostrare altro codice (di scarso interesse concettuale), per spiegare, come viene realizzata la scalabilità del codice client-side per uno switch con n computer di elaborazione. Mentre lato controller application abbiamo le due variabili MONITORED_PORTS, che è una tupla indicante le porte dello switch attraverso cui entra traffico per cui si può voler impostare una rotta, e DICTIONARY, che associa ad ogni funzione (stringa) le due porte dello switch a cui si affaccia il relativo computer di elaborazione, e possiamo quindi facilmente adattare il codice allo switch che andremo a controllare, client-side è un po' più complicato.

Prima di tutto, si ha il dialog, che deve essere creato non nella pagina HTML, ma nel codice js, come si è visto, e modificare l'HTML interno del dialog dinamicamente in base al traffico da servire (come visto prima). Per adattare facilmente il codice JS a uno switch di n funzioni, viene conservata, in utils.js, una lista di stringhe indicanti le funzioni offerte dallo switch, che ovviamente devono essere le stesse presenti su turnout.py (a meno dello spazio, incluso client-side per formattare adeguatamente la stringa per l'utente, poi comunque alla controller application saranno inviate senza spazio). Tornando poi a live.js invece, troveremo, subito dopo

la definizione del dialog, l'esecuzione di una funzione (createCheckboxes), che si occuperà di creare dinamicamente, in base alle stringhe immesse nella lista suddetta, il corpo del form con cui l'utente interagirà all'apertura del dialog, e che sarà inserito, ovviamente, attraverso createDialogText, per formare di volta in volta un dialog che si adatti al traffico passante. Questa struttura, che quindi permette l'adattabilità del codice, in modo rapido e intuitivo, a uno switch connesso a n computer di elaborazione, si basa tutto sulla presenza, client-side e server-side, di una variabile che definisca (con lo spazio, se necessario, client-side, e senza spazio, invece, server-side) le funzioni (come stringhe) attivabili dallo switch che andremo a controllare. Passiamo ora a routes.js.

```
$( document ).ready(function() {
    getRoutes();
});
```

Anche qui, abbiamo il caricamento delle rotte impostate attraverso una funzione (getRoutes), che si verifica però solo quando avviene l'evento ready (non a intervalli regolari).

```
function getRoutes() {

    var routesTableBody = document.getElementById('routes');

    while (routesTableBody.firstChild) {
        routesTableBody.removeChild(routesTableBody.firstChild);
    }

    $.get(url.concat("rest/routes"))
        .done(function(data) {

            if (data)
            {
                var routes = data.split('\n');
                for(x in routes)
                {
                    if(routes[x])
                    {
                        addEntry(routes[x]);
                    }
                }
            }
            else
            {
                console.log("Requested page is empty");
            }
        }).fail(function(data) {
            console.log("Failed sending request");
        });
}
```

La funzione getRoutes funziona in maniera molto simile a quanto visto precedentemente con getFirstCommunication, quindi non ci dilungheremo in spiegazioni ripetitive (le rotte vengono lette da rest/routes, come indicato in turnout_rest.py). L'unica cosa che vale la pena notare (e che vedremo meglio in seguito) è la presenza, nel documento HTML, di una tabella, dove ogni entry identifica una rotta impostata: all'inizio, prima della lettura delle rotte correnti, per semplicità, la tabella viene completamente svuotata per essere ripopolata successivamente, riga per riga, attraverso la funzione addEntry, che prende in ingresso proprio una riga (che non sia vuota), rappresentante una rotta impostata. La funzione addEntry si basa tutta su manipolazioni e creazione di nuovi elementi per il DOM, quindi non la analizzeremo nella sua interezza, ma andremo a esaminare solo alcuni passi salienti per la modifica delle rotte (successivamente).

```

btn.onclick = function(e) {
    var detail = details[e.target.id];
    $dialog.html(createDialogText(detail.src, detail.dst,
detail.proto, detail.activated));
    sorgente = detail.src;
    destinazione = detail.dst;
    protocollo = detail.proto;
    $dialog.dialog('open');
};

var temp = new route(src, dst, proto, activated);
details[btn.getAttribute('id')] = temp;

```

La tabella, dove vanno le entry rappresentanti le rotte, hanno, come ultimo campo, una colonna che accoglie, per ogni entry, un pulsante recante la scritta Modify, che permette di modificare appunto la rotta a cui è associato. Nella funzione addEntry, infatti, dopo aver trovato tutte le altre informazioni della rotta e averle aggiunte alla tabella, si associa al pulsante di quella rotta un event handler, che gestisce appunto il click dell'utente. Difatti, una volta cliccato, nell'event handler si recuperano i dettagli della rotta attraverso una lista mantenuta in memoria (details), indicizzata attraverso l'id del pulsante che ha scatenato l'evento, si imposta l'HTML interno di un dialog (caratterizzato allo stesso modo della pagina precedente) e lo si apre (anche questo dialog, in caso si scelga di confermare le modifiche, invia la richiesta a rest/set_route). Le ultime due righe del codice postato, invece, realizzano proprio il salvataggio in details di un oggetto route, che è caratterizzato proprio da una sorgente, una destinazione, un protocollo, e dalle relative funzioni attivate.

Rimane, ora, solo da osservare la realizzazione delle pagine HTML, riassunta qui di seguito da una serie di screenshot.

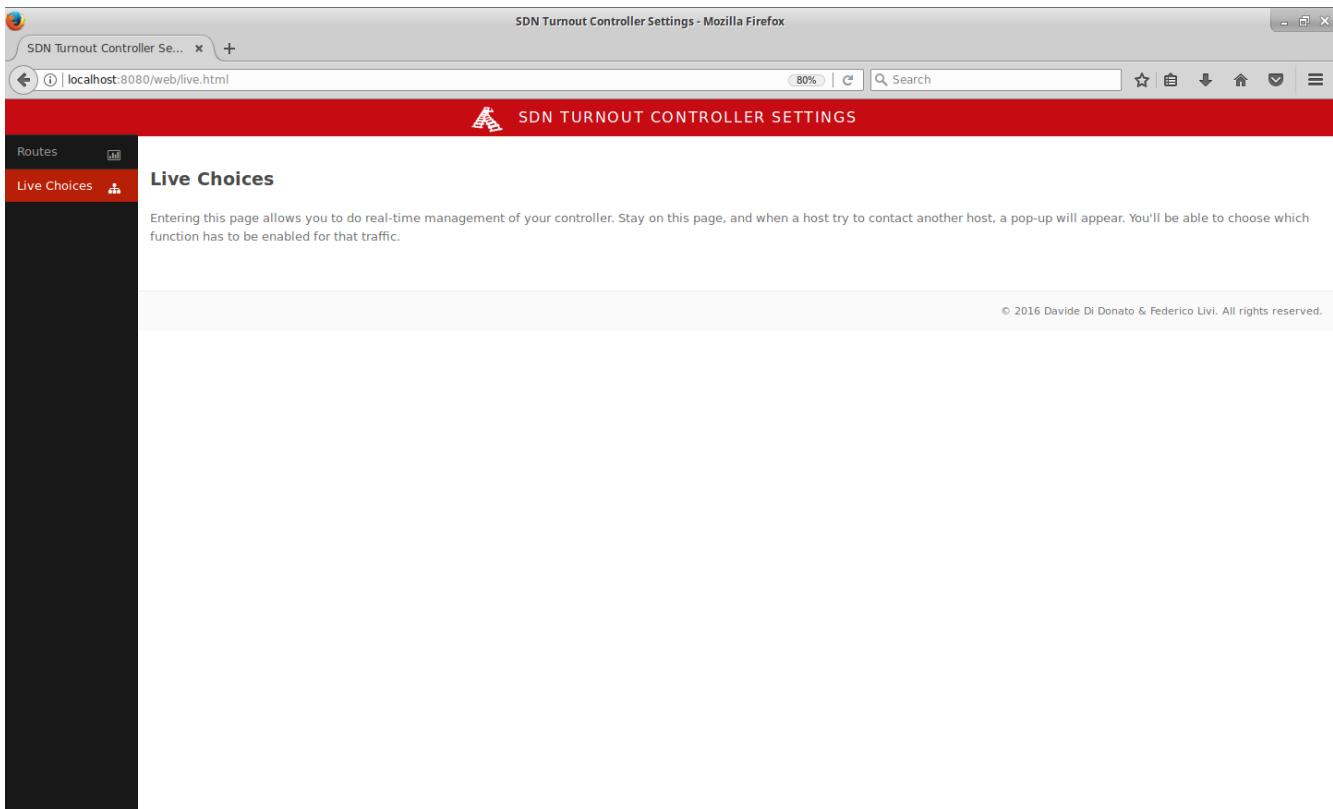


Fig. 23: Live Choices è la sezione della Web GUI che si occupa di gestire il nuovo traffico in ingresso

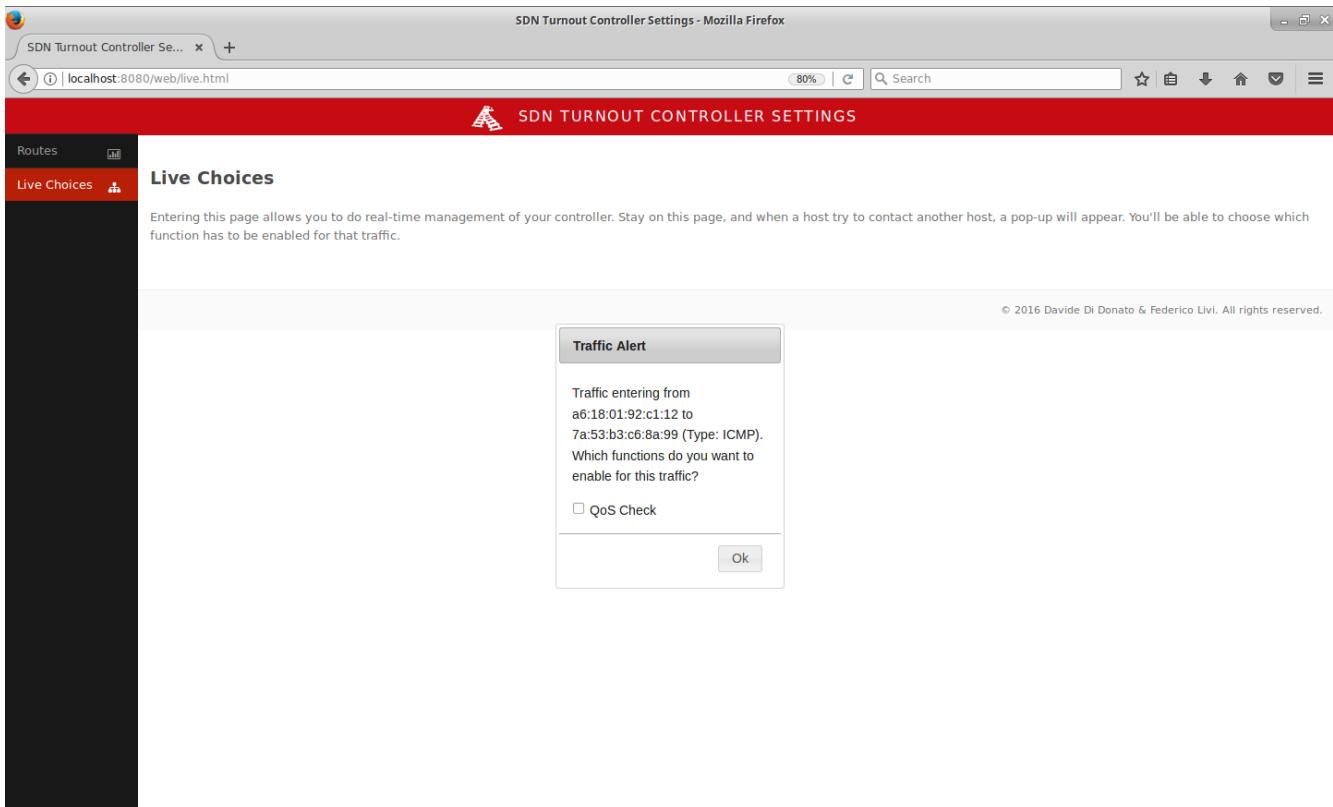


Fig. 24: Live Choices quando viene rilevato un nuovo traffico in ingresso

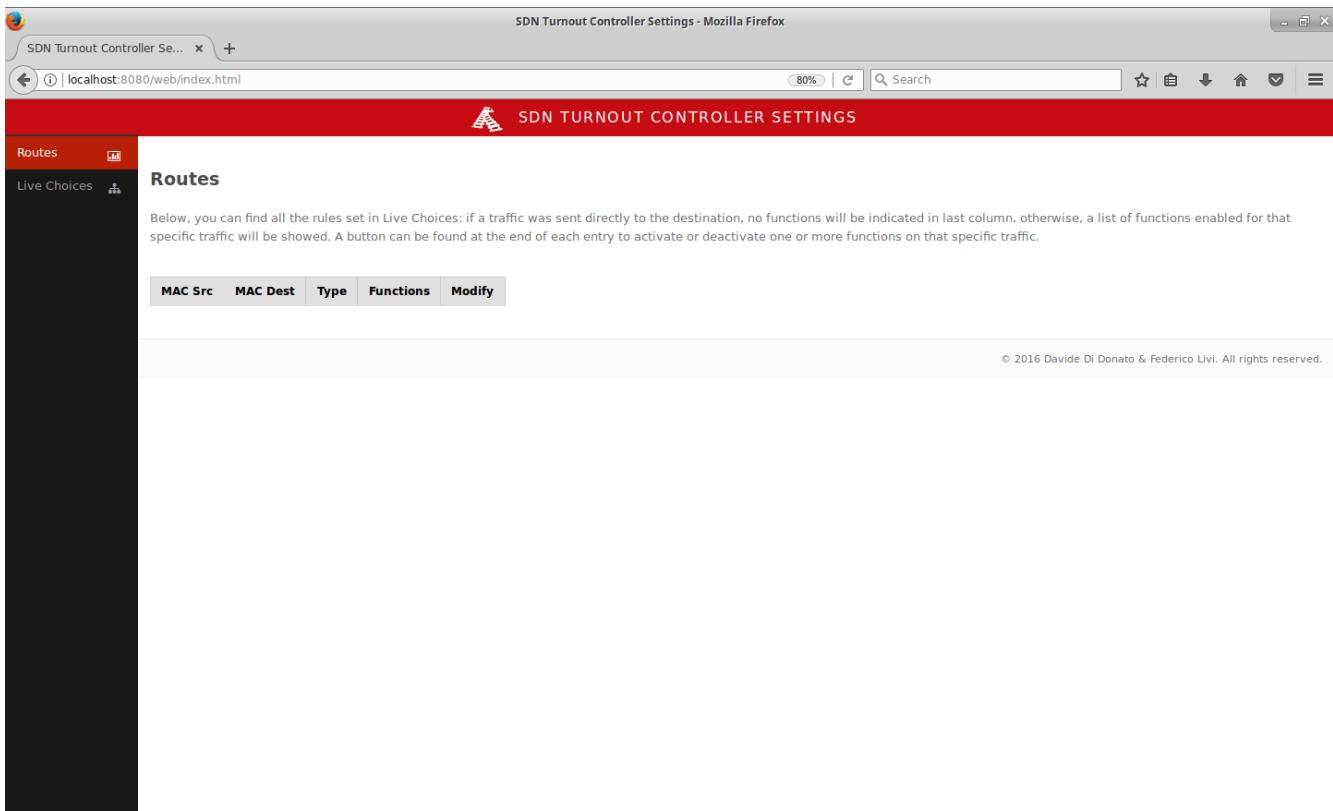


Fig. 25: Routes è la sezione della Web GUI che si occupa della gestione delle rotte esistenti

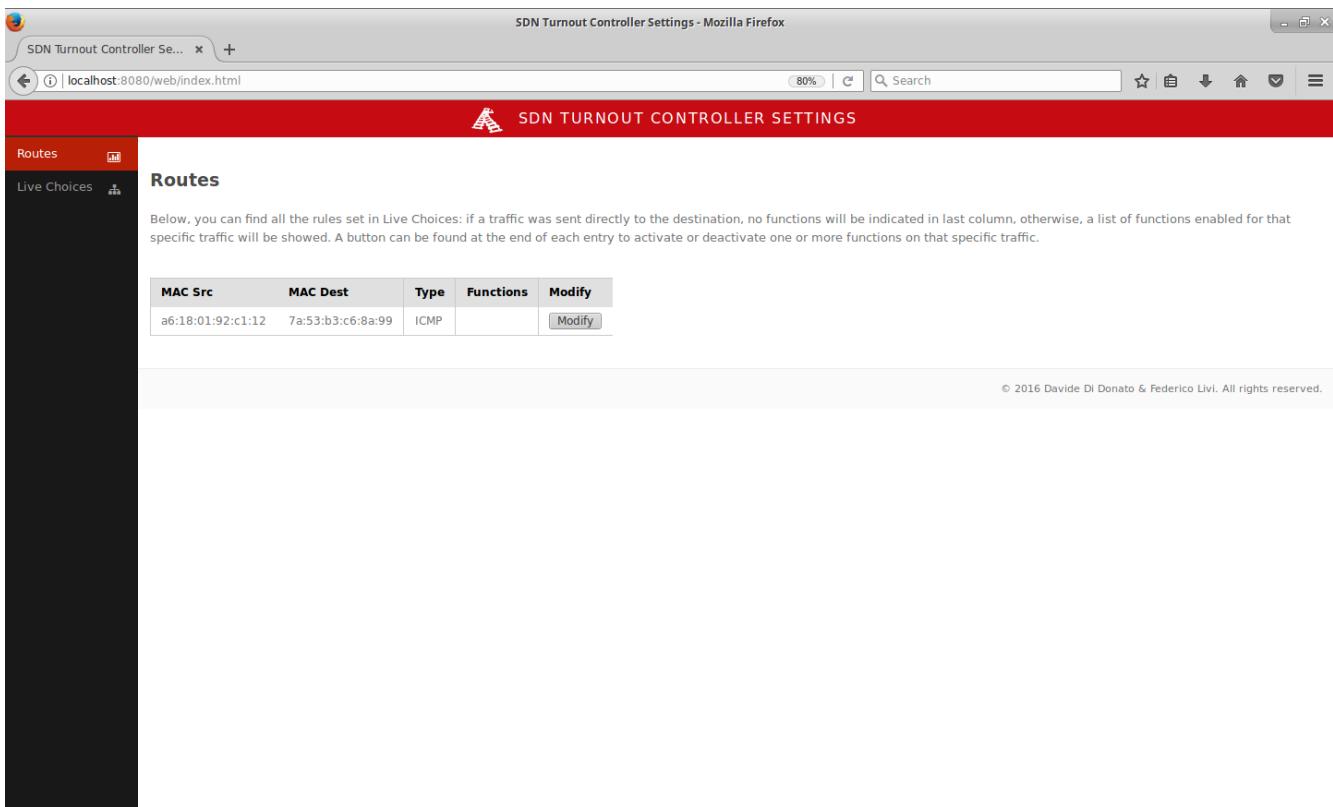


Fig. 26: Routes quando viene impostata una rota in Live Choices

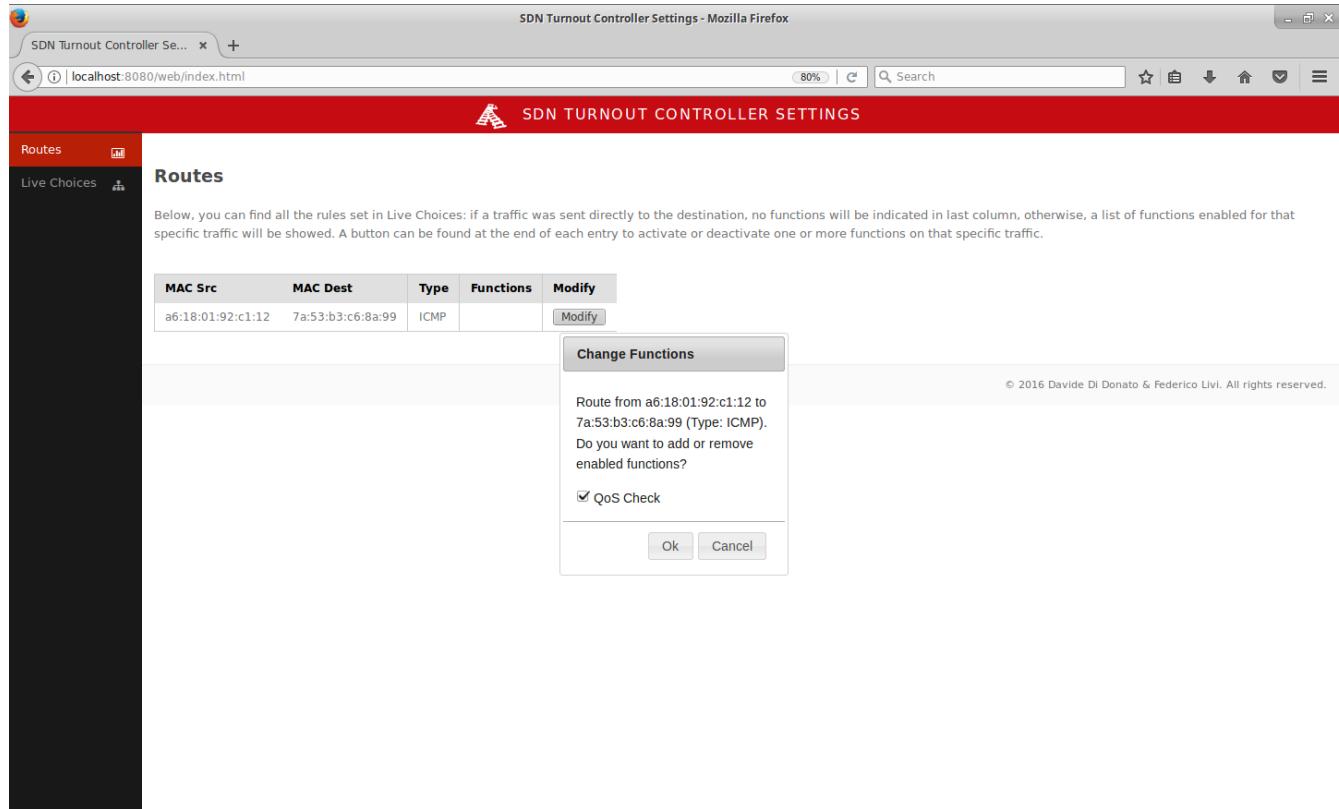


Fig. 27: Routes quando si clicca sul pulsante Modify di una qualsiasi rottta esistente

7.11 Diagramma finale di progettazione

Chiudiamo, infine, questo capitolo mostrando, di seguito, il diagramma finale di progettazione risultante dalle classi realizzate e descritte precedentemente. Vengono incluse, in questo diagramma, anche alcune classi non di progetto, ma di cui si è discusso precedentemente e risultano vitali per il funzionamento del sistema, anche al fine di rendere chiari i collegamenti che ci sono tra l'architettura Ryu (e non solo) e i moduli da noi sviluppati.

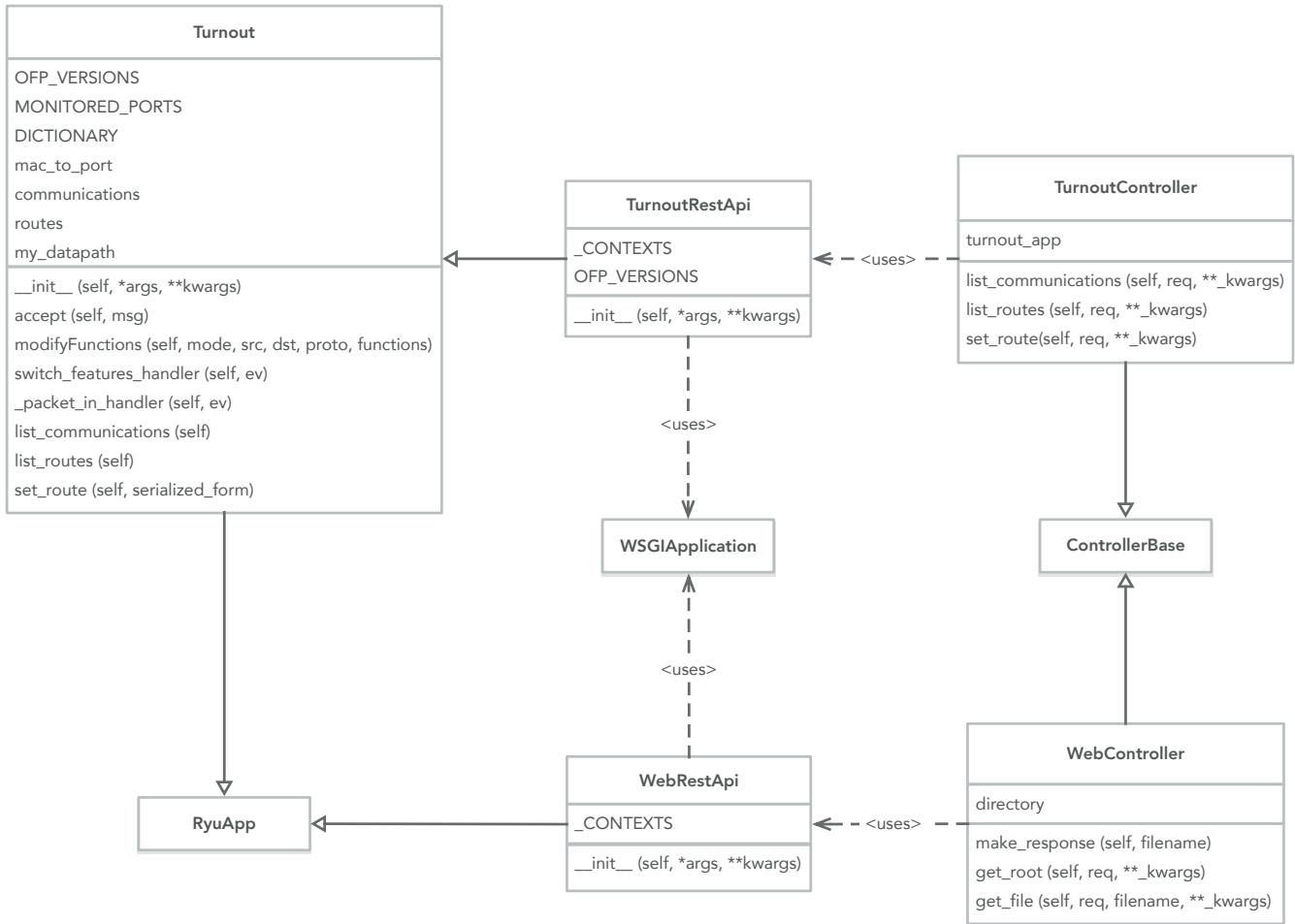


Fig. 28: classi di progettazione

CAPITOLO 8

TEST SU RETE VIRTUALE

Completato ora lo sviluppo del nostro controller SDN, procediamo col testarlo, inizialmente, su una rete virtuale, prima di passare alla rete reale che è il nostro obiettivo ultimo.

8.1 Topologia di test

Per cominciare, presentiamo la topologia sulla quale si effettuerà questo test. Sarà composta, in linea generale, da due end-point di comunicazione, che vogliono parlare attraverso uno switch, che sarà l'unico datapath presente nella topologia di prova. Quest'ultimo, sarà ovviamente guidato dal nostro controller SDN, che mira quindi a fornire all'utente un controllo completo sul traffico passante attraverso l'impostazione di una rotta per quest'ultimo in base alle scelte dell'utente stesso in relazione ai computer di elaborazione connessi.

Date queste le premesse, bisogna che lo switch presenti quindi una caratteristica fondamentale: la presenza di n computer di elaborazione, per testare il comportamento della controller application in situazioni complesse. Nella nostra topologia, si collegheranno quattro computer di elaborazione, corrispondenti quindi ad altrettante funzioni, e, in ultimo stadio, a otto differenti porte Ethernet. Dato che però noi non ci occuperemo di realizzare nel concreto i computer di elaborazione, e sapendo che essi si occuperanno solamente di analizzare il traffico senza manipolarlo, possiamo procedere nel nostro test cortocircuitando le interfacce dove sarebbero collegati i computer di elaborazione, e far passare così, lo stesso identico traffico da un'interfaccia all'altra senza aggiungere complessità aggiuntiva.

Di seguito, viene riportata un'immagine esplicativa della topologia che vogliamo andare a realizzare.

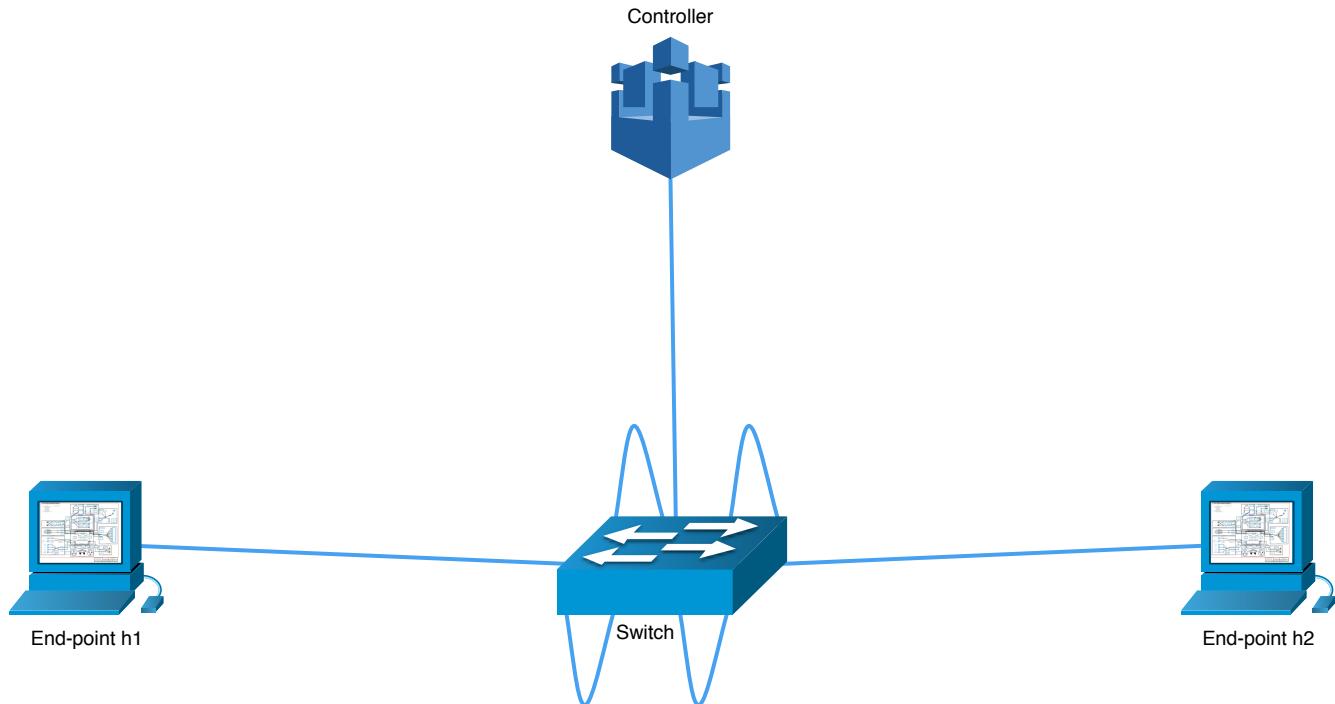


Fig. 29: topologia di riferimento da realizzare; da notarsi in particolare i cortocircuiti, che rappresentano astrattamente i computer di elaborazione che accolgono e riemettono lo stesso traffico senza manipolarlo

8.2 Realizzazione della topologia di test

Ora, al fine di virtualizzare la topologia richiesta, useremo una macchina virtuale, disponibile qui [24]. È un file OVA, quindi si riesce ad importare e a far partire agevolmente in VMWare. Contiene già tutti gli strumenti presentati precedentemente, e con le versioni specificate. A questo punto, è possibile, eseguendo uno script che usa funzionalità di Mininet, Open vSwitch e di Linux stesso, effettuare il deploy della rete sulla macchina virtuale. Andiamo ad analizzarlo.

```
#!/bin/bash
if test `whoami` != root; then
    echo "Permesso negato. Avviare lo script come root!"
    exit 1
fi
```

Innanzitutto, andremo a controllare come sia stato avviato questo script, perché è necessario aviarlo con privilegi di root: se non è questo il caso, si esce.

```
if test "$1" == 'c'; then
    echo "Pulizia iniziata: "
    mn -c
    echo -n "Eliminazione dei namespace: "
    for i in $(seq 1 2)
    do
        ip netns del h"$i"
    done
    echo "fatto!"
    exit 2
fi
```

Nel caso sia stato avviato con privilegi di root, si effettua un test per verificare se lo script sia stato avviato con la lettera c come primo argomento: se è questo il caso, si utilizza un comando di Mininet per effettuare una pulizia generale rapida (mn -c), poi, per ogni end-point presente nella rete, si cancella il network namespace associato (ip netns del h"\$i"), e infine si esce. Addentriamoci ora nel caso in cui lo script sia stato quindi avviato con root e senza la lettera c come primo argomento.

```
ip netns add h1
ip netns add h2
echo "Host disponibili: "
ip netns list
```

Come risulta chiaro, vengono creati due network namespace, uno per ogni end-point (o host) della rete (h1 e h2). Poi, per debug, vengono listati i network namespace esistenti.

```
echo "Creazione switch s1.."
ovs-vsctl add-br s1
echo "Creazione collegamento tra switch e h1"
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
```

```

echo "Creazione collegamento tra switch e h2"
ip link add h2-eth0 type veth peer name s1-eth2
ip link set h2-eth0 netns h2
ovs-vsctl add-port s1 s1-eth2
echo "Creazione collegamento tra s1-eth7 e s1-eth8"
ip link add s1-eth7 type veth peer name s1-eth8
ovs-vsctl add-port s1 s1-eth8
ovs-vsctl add-port s1 s1-eth7
echo "Creazione collegamento tra s1-eth5 e s1-eth6"
ip link add s1-eth5 type veth peer name s1-eth6
ovs-vsctl add-port s1 s1-eth5
ovs-vsctl add-port s1 s1-eth6
echo "Creazione collegamento tra s1-eth0 e s1-eth9"
ip link add s1-eth0 type veth peer name s1-eth9
ovs-vsctl add-port s1 s1-eth0
ovs-vsctl add-port s1 s1-eth9
echo "Creazione collegamento tra s1-eth3 e s1-eth4"
ip link add s1-eth3 type veth peer name s1-eth4
ovs-vsctl add-port s1 s1-eth3
ovs-vsctl add-port s1 s1-eth4

```

Poi, viene creato un bridge, s1, e vengono creati mano a mano tutti i collegamenti che esso ha. Per quanto riguarda il primo collegamento, quello con h1, viene innanzitutto detto, con il primo comando, che l'interfaccia h1-eth0 di h1 è collegata all'interfaccia s1-eth1 di s1. Poi, viene tolta l'interfaccia h1-eth0 dal network namespace globale per ricondurla al namespace di h1; infine, viene aggiunta la porta s1-eth1 a s1. Stesso lavoro verrà poi effettuato per h2.

I restanti collegamenti creati rappresentano i cortocircuiti, analizziamo per esempio il primo collegamento: viene creato un collegamento tra la porta s1-eth7 e s1-eth8, e con i restanti due comandi, entrambe le porte vengono aggiunte allo switch s1. Passiamo ora agli ultimi comandi per terminare la configurazione della nostra rete virtuale.

```

echo "Stoppo i flood dalle interfacce cortocircuitate.."
for i in $(seq 3 10)
do
    ovs-ofctl mod-port s1 "$i" no-flood
done

```

Il ciclo soprastante è molto importante: venendosi a creare dei loop, dati i cortocircuiti tra le porte dello switch, si creano dei problemi, in quanto ad esempio, nei primi traffici che si verificano nello switch, il controller non conosce, per ogni calcolatore, la relativa porta dello switch a cui si affaccia, e quindi si fanno dei flood, considerando la nostra topologia, dalla porta da cui si affaccia h1 o dalla porta a cui si affaccia h2. Però, se si permette il flood a h1 o a h2, i pacchetti di un qualsiasi traffico entreranno quindi anche attraverso tutte le interfacce cortocircuitate a due a due, a cui però non è connesso alcun calcolatore; inoltre, i pacchetti di quel traffico entrato percorrerà il cortocircuito verso l'altra porta e uscirà di nuovo, provocando probabilmente un nuovo flood, e così via, fino a saturare la banda disponibile per i canali di trasmissione e lo switch stesso. Attraverso quell'unico comando, invece, si impedisce la possibilità, per le interfacce cortocircuitate, di fare flood, risolvendo il problema senza la necessità di applicare STP o RSTP (Spanning Tree Protocol o Rapid Spanning Tree Protocol).

```

for i in $(seq 1 2)
do
    echo "Aggiunta indirizzo a h$i.."
    ip netns exec h"$i" ifconfig h"$i"-eth0 10.0.0."$i"
done

echo "Configurazione interfacce.."
for i in $(seq 0 9)
do
    ifconfig s1-eth"$i" up
    ifconfig s1-eth"$i" 0
done
ifconfig s1 up

```

A questo punto, si applicano comuni comandi di Linux per: configurare correttamente le interfacce degli host (settando gli IP per le interfacce) nel primo ciclo e azzerare invece gli IP delle interfacce dello switch e avviare nel secondo ciclo. Infine, si avvia anche lo switch.

```

echo "Aggiunta controller"
ovs-vsctl set-controller s1 tcp:127.0.0.1:6633
echo "Situazione openvswitch:"
ovs-vsctl show

```

In quest'ultima parte dello script, si setta l'indirizzo di rete del controller (che tipicamente ascolta alla porta 6633) a cui deve fare riferimento lo switch s1 (localhost, praticamente), e infine si procede a visualizzare la situazione di Open vSwitch, per debug, attraverso l'ultimo comando.

8.3 Esecuzione del test

Per far partire il controller, abbiamo realizzato uno script (`run_web_gui.sh`) che ci permette, tramite la sua esecuzione, di eseguire il RyuManager, componente fondamentale dell'architettura Ryu, che provvederà poi a caricare i moduli a cui siamo interessati (`turnout_rest.py` e `my_fileserver.py`). Di seguito, la riga di codice di cui parlavamo.

```
#!/bin/sh
```

```

ryu/bin/ryu-manager --observe-links ryu/ryu/app/WebGUI/my_fileserver
ryu/ryu/app/WebGUI/turnout_rest

```

Ora, osserviamo, attraverso le immagini, il deploy e un esempio del funzionamento della rete. Da considerare che per questo test, le associazioni tra funzioni offerte e porte dello switch saranno le seguenti:

1. QoS Check: 3 e 4;
2. DDoS Prevention: 5 e 6;
3. Load Balancer: 7 e 8;
4. Intrusion Detection: 9 e 10.

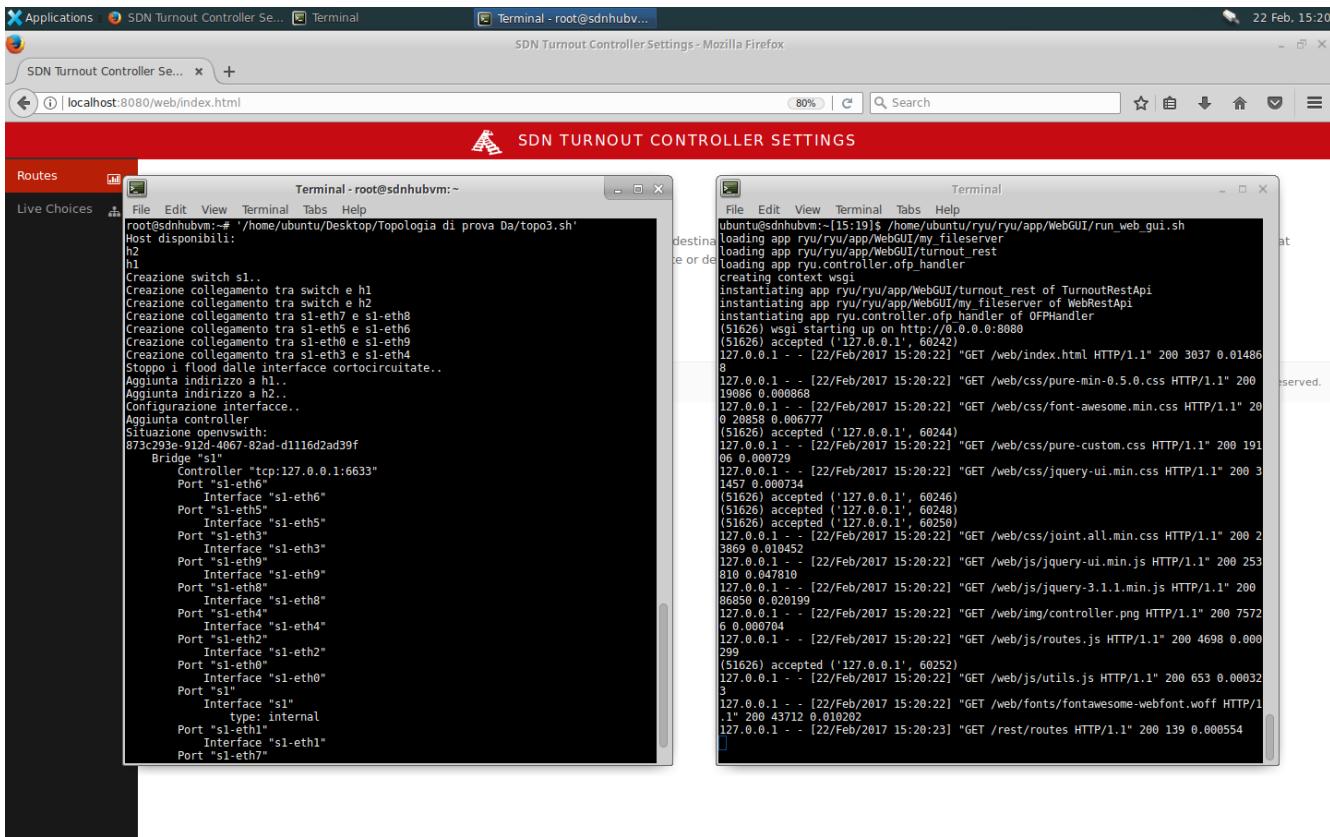


Fig. 30: avviamo la rete virtuale e il controller

```
Terminal - root@sdnhubvbm:~-
File Edit View Terminal Tabs Help
Creazione collegamento tra switch e h1
Creazione collegamento tra switch e h2
Creazione collegamento tra s1-eth7 e s1-eth8
Creazione collegamento tra s1-eth5 e s1-eth6
Creazione collegamento tra s1-eth0 e s1-eth9
Creazione collegamento tra s1-eth3 e s1-eth4
Stoppo i flood dalle interfacce cortocircuitate..
Aggiunta indirizzo a h1..
Aggiunta indirizzo a h2..
Configurazione interfacce..
Aggiunta controller
Situazione openvswitch:
873c293e-912d-4067-82ad-d1116d2ad39f
    Bridge "s1"
        Controller "tcp:127.0.0.1:6633"
        Port "s1-eth0"
            Interface "s1-eth0"
        Port "s1-eth9"
            Interface "s1-eth9"
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1-eth4"
            Interface "s1-eth4"
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth8"
            Interface "s1-eth8"
        Port "s1-eth2"
            Interface "s1-eth2"
        Port "s1-eth6"
            Interface "s1-eth6"
        Port "s1-eth3"
            Interface "s1-eth3"
        Port "s1-eth7"
            Interface "s1-eth7"
        Port "s1-eth5"
            Interface "s1-eth5"
        ovs_version: "2.5.0"
TUTTO FATTO!
root@sdnhubvbm:~# ip netns exec h1 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```
--- 10.0.0.2 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14010ms
```

Fig. 31: proviamo ad effettuare un tentativo di ping da h1 a h2; come possiamo notare, non passa

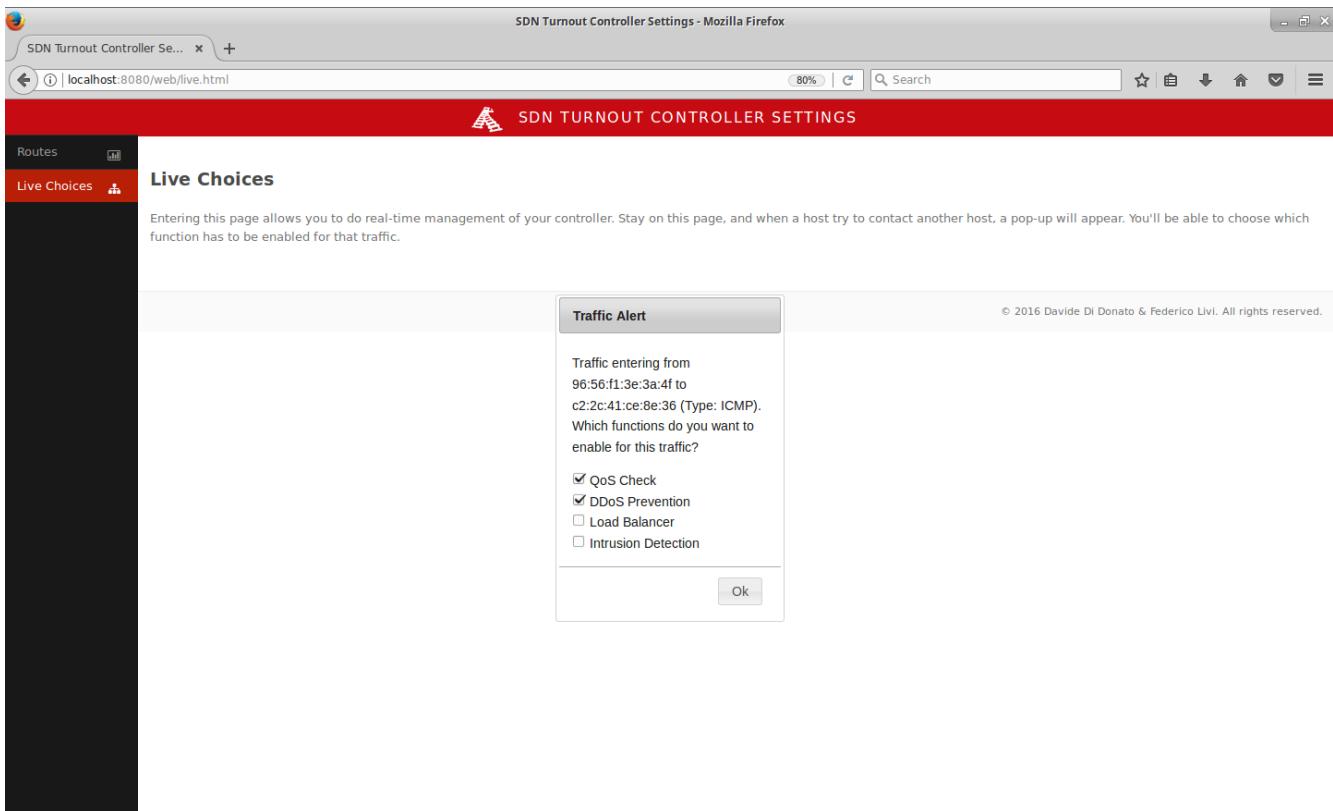


Fig. 32: se andiamo però alla sezione Live Choices, troveremo un pop-up che ci avvisa del traffico passato; selezioniamo le funzioni che vogliamo attivare sul traffico, e clicchiamo ok

```
Terminal - root@sdnhubvm:~
File Edit View Terminal Tabs Help
Controller "tcp:127.0.0.1:6633"
Port "s1-eth0"
 Interface "s1-eth0"
Port "s1-eth9"
 Interface "s1-eth9"
Port "s1-eth1"
 Interface "s1-eth1"
Port "s1-eth4"
 Interface "s1-eth4"
Port "s1"
 Interface "s1"
 type: internal
Port "s1-eth8"
 Interface "s1-eth8"
Port "s1-eth2"
 Interface "s1-eth2"
Port "s1-eth6"
 Interface "s1-eth6"
Port "s1-eth3"
 Interface "s1-eth3"
Port "s1-eth7"
 Interface "s1-eth7"
Port "s1-eth5"
 Interface "s1-eth5"
ovs version: "2.5.0"
TUTTO FATTO!
root@sdnhubvm:~# ip netns exec h1 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14010ms
root@sdnhubvm:~# ip netns exec h1 ping 10.0.0.2
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=2.88 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.329 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.121 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.142 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.100 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.088 ms
```

Fig. 33: come è possibile notare, ora il ping passa; andiamo però a verificare le flow entry presenti sullo switch nell'immagine seguente

```

Terminal - root@sdnhubvm:~#
File Edit View Terminal Tabs Help
^C
--- 10.0.0.2 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14010ms
root@sdnhubvm:~# ip netns exec h1 ping 10.0.0.2
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=2.88 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.329 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.121 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.142 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.100 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=0.089 ms
64 bytes from 10.0.0.2: icmp_seq=16 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=17 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=18 ttl=64 time=0.089 ms
64 bytes from 10.0.0.2: icmp_seq=19 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=20 ttl=64 time=0.088 ms
^C
--- 10.0.0.2 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 19003ms
rtt min/avg/max/mdev = 0.059/0.243/2.887/0.609 ms
root@sdnhubvm:~# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=1,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:3
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=4,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:5
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=6,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:2
 cookie=0x0, duration=63.180s, table=0, n_packets=20, n_bytes=1960, idle_age=44, priority=1,icmp,in_port=2,dl_dst=96:56:f1:3e:3a:4f actions=output:1
 cookie=0x0, duration=362.472s, table=0, n_packets=79, n_bytes=7642, idle_age=58, priority=0 actions=CONTROLLER:65535
root@sdnhubvm:~#
```

Fig. 34: come è possibile notare, vengono installate sullo switch le regole che ci aspettiamo; dalla porta 1 (h1) il traffico viene ridirezionato sulla 3, poi, quello che entrerà dalla porta 4 (lo stesso che è uscito dalla porta 3), verrà ridirezionato sul successivo computer di elaborazione allo stesso modo, e infine verrà ridirezionato (nella terzultima regola), verso il calcolatore destinario

The screenshot shows the 'SDN Turnout Controller Settings' interface in Mozilla Firefox. The main page has a red header bar with the title 'SDN TURNOUT CONTROLLER SETTINGS'. On the left, there's a sidebar with tabs for 'Routes' (which is selected) and 'Live Choices'. The main content area displays a table of routes:

| MAC Src           | MAC Dest          | Type | Functions       | Modify                                                                      |
|-------------------|-------------------|------|-----------------|-----------------------------------------------------------------------------|
| 96:56:f1:3e:3a:4f | c2:2c:41:ce:8e:36 | ICMP | QoS Check, DDoS | <span style="border: 1px solid #ccc; padding: 2px;">Change Functions</span> |

A modal dialog box is open over the table, titled 'Change Functions'. It contains the following text and checkboxes:

Route from 96:56:f1:3e:3a:4f to c2:2c:41:ce:8e:36 (Type: ICMP).  
Do you want to add or remove enabled functions?

QoS Check  
 DDoS Prevention  
 Load Balancer  
 Intrusion Detection

At the bottom of the modal are 'Ok' and 'Cancel' buttons.

Fig. 35: andando alla sezione Routes, ovviamente la rotta compare; cliccando il pulsante Modify viene fuori il form dove possiamo impostare le funzioni volute; settiamole e clicchiamo ok

```

File Edit View Terminal Tabs Help
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.100 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=0.089 ms
64 bytes from 10.0.0.2: icmp_seq=16 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=17 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=18 ttl=64 time=0.089 ms
64 bytes from 10.0.0.2: icmp_seq=19 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=20 ttl=64 time=0.088 ms
^C
-- 10.0.0.2 ping statistics --
20 packets transmitted, 20 received, 0% packet loss, time 19003ms
rtt min/avg/max/mdev = 0.059/0.243/2.887/0.609 ms
root@sdnhubvbm:~# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=1,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:3
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=4,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:5
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=6,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:2
 cookie=0x0, duration=63.180s, table=0, n_packets=20, n_bytes=1960, idle_age=44, priority=1,icmp,in_port=2,dl_dst=96:56:f1:3e:3a:4f actions=output:1
 cookie=0x0, duration=362.472s, table=0, n_packets=79, n_bytes=7642, idle_age=58, priority=0 actions=CONTROLLER:65535
root@sdnhubvbm:~# ip netns exec h1 ping 10.0.0.2
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=0.567 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.181 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.091 ms
^C

```

Fig. 36: come è possibile notare, il ping passa anche stavolta...

```

File Edit View Terminal Tabs Help
y=1,icmp,in_port=4,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:5
 cookie=0x0, duration=16.772s, table=0, n_packets=0, n_bytes=0, idle_age=16, priority=1,icmp,in_port=6,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:2
 cookie=0x0, duration=63.180s, table=0, n_packets=20, n_bytes=1960, idle_age=44, priority=1,icmp,in_port=2,dl_dst=96:56:f1:3e:3a:4f actions=output:1
 cookie=0x0, duration=362.472s, table=0, n_packets=79, n_bytes=7642, idle_age=58, priority=0 actions=CONTROLLER:65535
root@sdnhubvbm:~# ip netns exec h1 ping 10.0.0.2
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=0.567 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.181 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.088 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.091 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.089 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.096 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=0.134 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=0.090 ms
^C
-- 10.0.0.2 ping statistics --
15 packets transmitted, 15 received, 0% packet loss, time 14002ms
rtt min/avg/max/mdev = 0.082/0.130/0.567/0.119 ms
root@sdnhubvbm:~# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=23.553s, table=0, n_packets=15, n_bytes=1470, idle_age=4, priority=1,icmp,in_port=1,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:5
 cookie=0x0, duration=23.553s, table=0, n_packets=15, n_bytes=1470, idle_age=4, priority=1,icmp,in_port=6,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:9
 cookie=0x0, duration=23.553s, table=0, n_packets=15, n_bytes=1470, idle_age=4, priority=1,icmp,in_port=10,dl_src=96:56:f1:3e:3a:4f,dl_dst=c2:2c:41:ce:8e:36 actions=output:2
 cookie=0x0, duration=133.268s, table=0, n_packets=35, n_bytes=3430, idle_age=4, priority=1,icmp,in_port=2,dl_dst=96:56:f1:3e:3a:4f actions=output:1
 cookie=0x0, duration=432.560s, table=0, n_packets=81, n_bytes=7726, idle_age=13, priority=0 actions=CONTROLLER:65535
root@sdnhubvbm:~#

```

Fig. 37: ... e andando a controllare le flow entry, abbiamo proprio quello che ci aspettiamo; per concludere possiamo anche notare come il traffico da h2 a h1 sia stato accettato direttamente, e come la flow entry a priorità più bassa sia proprio quella che indica allo switch di passare i pacchetti dei traffici per cui ancora non c'è una flow entry corrispondente, al controller, che sceglierà poi come gestirli

# CAPITOLO 9

## TEST SU RETE REALE

Per concludere questo lavoro di tesi, si è scelto infine, di testare il comportamento del controller SDN sviluppato su una rete reale, dove, peraltro, è già presente un altro controller SDN, sviluppato con finalità diverse (le approfondiremo subito sotto).

### 8.1 Topologia di test

Per cominciare, presentiamo la topologia sulla quale si effettuerà questo test. Sarà composta, in linea generale, anche qui, da due end-point di comunicazione, che saranno rappresentati da due calcolatori aventi come macchina virtuale la stessa che abbiamo citato al capitolo precedente. Nella macchina virtuale di questi calcolatori, verranno virtualizzati (ancora) un host e uno switch. Come nodi intermedi, avremo invece i Raspberry, che citavamo inizialmente, sui quali sarà virtualizzato uno switch e sarà messo in esecuzione un controller: un Raspberry conterrà il controller realizzato in questa tesi (controller 2), l'altro invece, il controller già realizzato di cui si parlava prima (controller 1). Di seguito, troviamo lo schema della rete che andremo a realizzare, che servirà a chiarire anche (e soprattutto) i collegamenti esistenti tra i diversi calcolatori.

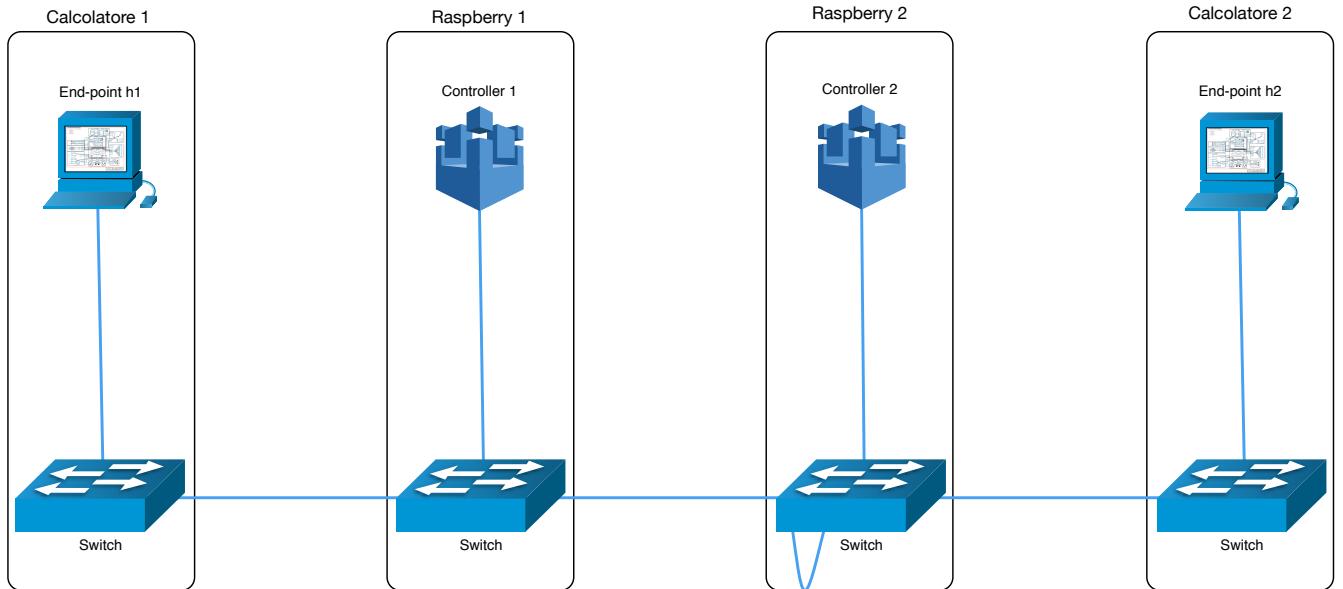


Fig. 38: topologia di riferimento da realizzare; da notarsi che in questo caso, è disponibile in rete una sola funzione, un solo calcolatore di elaborazione, anche qui rappresentato come un unico cortocircuito; avendo infatti il Raspberry solo 4 porte USB, considerando che la porta già presente è utilizzata per la connessione con lo switch del Raspberry 1 e una delle porte USB-Ethernet è usata per la connessione allo switch del calcolatore 2, rimangono 3 porte USB, per un massimo quindi di 1 funzione di elaborazione del traffico, dato che essa impegnava 2 porte. Ogni collegamento che unisce due diversi spazi rettangolari è realizzato attraverso porte Ethernet reali, il cortocircuito del Raspberry 2 è l'unica eccezione di collegamento reale all'interno di uno stesso spazio.

Mentre il calcolatore 1 e il calcolatore 2 sono ambienti da virtualizzare che non presentano particolari ambiguità o difficoltà, avendo solo uno switch in più a cui sono collegati (anche qui lo scopo è far comunicare i due end-point nel modo richiesto dall'utente), possiamo andare ad analizzare direttamente (in modo astratto) il lavoro svolto dal controller 1.

Il controller 1, si potrebbe dire, agisce come un firewall, con una politica impostata di default deny: è possibile, attraverso una REST API, consentire in tempo reale il passaggio del traffico,

allo stesso modo di come il controller 2 permette l'impostazione di una rotta per il traffico in transito. Quindi, quando si verificherà un nuovo traffico in transito in uno switch gestito dal controller 1, un pop-up apparirà all'utente, e gli sarà richiesto, così, di consentire o meno il traffico in transito. Se si accetta, o si rifiuta, una flow entry viene impostata sullo switch che ha generato il packet\_in, e mano a mano che il pacchetto attraversa altri switch e si verificano altri packet\_in, non verrà più chiesto all'utente di autorizzare quel traffico, perché la decisione è stata già presa per un altro switch, quindi si procederà a installare automaticamente una flow entry in accordo alla decisione precedente anche sul nuovo switch. Vale lo stesso se c'è un traffico di risposta al traffico iniziale: esso viene automaticamente autorizzato.

## 8.2 Realizzazione della topologia di test

Per ogni calcolatore e Raspberry, configurati come detto precedentemente, abbiamo preparato degli script, come quello appena visto, per creare ogni sottorete necessaria al funzionamento della rete totale. Non commenteremo i file anche qui, dato che i comandi visti prima vengono riutilizzati allo stesso modo e per le stesse finalità anche qui, ma guarderemo solo alcuni nuovi comandi e parleremo della struttura di ogni sottorete.

Per cominciare, possiamo partire dagli script da avviare sui calcolatori 1 e 2, dato che sono molto simili. Ognuno di loro crea un host associato a un network namespace (h1 per il calcolatore 1 e h2 per il calcolatore 2) e uno switch (s1 per il primo, s4 per il secondo): a quel punto, si collega l'interfaccia dell'host all'interfaccia dello switch creato (h1-eth0 con s1-eth1, h2-eth0 con s4-eth1; entrambe virtuali) e si aggiunge quest'ultima allo switch. Si associa alle interfacce degli host un indirizzo IP (10.0.0.1 a h1 e 10.0.0.2 a h2), mentre l'indirizzo IP dell'interfaccia dello switch è messo a 0 e viene avviata, e poi, viene avviato anche lo switch. Allo switch di ognuna, inoltre, è stata aggiunta anche la porta eth0, fisica, propria del calcolatore su cui viene avviato lo script, con l'IP azzerato. Infine, vengono collegati gli switch al controller 1 col comando visto precedentemente (nella rete reale, il controller 1 si trova a 10.2.2.1:6633). Questi, sono passi che abbiamo visto anche nel test della nostra rete virtuale. Ora, rimangono da analizzare i comandi finali presenti invece in questi script della rete reale.

```
ip addr add 10.1.1.1/24 dev s1
sleep 1
ip route add 10.2.2.0/24 dev s1
ip route add 10.3.3.0/24 dev s1
ip route add 10.4.4.0/24 dev s1
```

Il primo comando associa un indirizzo IP allo switch s1 (per s4 sarà 10.4.4.1/24), e, contestualmente, vengono eseguiti tre comandi che istruiscono il sistema su quale sia l'interfaccia associata ad ogni destinazione che si può voler raggiungere. Da notarsi che il comando di sleep è necessario in quanto a volte la modifica data da ip addr non è subito vista da ip route, quindi si sceglie di attendere per garantire la corretta esecuzione dei comandi successivi. I comandi qui riportati saranno presenti, con le dovute modifiche, anche sugli altri script.

Ora passiamo ai due Raspberry.

Per quanto riguarda il Raspberry contenente il controller 1, viene creato uno switch (s2) con due porte (eth0 e eth1), entrambe con IP azzerati. Viene impostato come IP dello switch 10.2.2.1/24 e vengono aggiunte le indicazioni per raggiungere le altre sottoreti come sopra. Ovviamente, viene impostato per lo switch il controller associato, che sarà il controller presente in locale (10.2.2.1:6633).

Passando invece al Raspberry contenente il controller 2, viene creato uno switch (s3) con 4 porte (eth0, eth1, eth2, eth3), tutte con IP azzerati. Viene poi collegato al controller 2 presente in locale (10.3.3.1:6633), e viene impostato anche per esso un indirizzo IP (10.3.3.1/24) e le indicazioni per raggiungere le altre sottoreti. Infine, si identificano le porte che effettuano il cortocircuito e si nega loro la possibilità di fare flood con lo stesso comando visto precedentemente.

È importante notare che, al contrario dei calcolatori, dove si è virtualizzato l'host incaricato della comunicazione e le interfacce associate ad esso, nei Raspberry, ogni interfaccia citata è reale, realizzata o attraverso la porta Ethernet già presente, o attraverso gli adattatori Apple USB-Ethernet citati precedentemente. Inoltre, in questo caso, se si va ad analizzare lo script del secondo Raspberry, si potrà notare che manca appunto il comando di collegamento tra due interfacce che realizzi il cortocircuito: essendo questo un test su rete reale, abbiamo utilizzato un normale cavo Ethernet per collegare arbitrariamente una coppia di porte tra quelle disponibili (l'importante è ricordarsi poi di adattare il codice in base ai collegamenti effettuati).

Per ultimo, è necessario notare come i due controller si siano divisi le responsabilità della rete totale: nello specifico, il controller 1 guida sia lo switch del calcolatore 1, sia quello del calcolatore 2, sia quello che trova in rete locale, mentre il controller 2 guida solo quello che trova in rete locale. Questo perché, potendo il traffico partire anche dal calcolatore 2 per primo, è necessario che il controller 1 accetti il transito di quello stesso traffico sulla rete: il controller 2 infatti non si deve curare di ciò che arriva da una porta non monitorata, procede ad accettarlo e basta. Se così non fosse, e ad esempio il controller 2 applicasse lo stesso comportamento del primo per le porte non monitorate, ci sarebbero sovrapposizioni di responsabilità e una conseguente complicazione della logica del controller 2 non banale.

Prima di eseguire gli script, abbiamo effettuato i collegamenti fisici attraverso cavi Ethernet di diversa tipologia, per replicare, in modo identico, la topologia sopra presentata; i collegamenti virtuali saranno eseguiti attraverso comandi degli script ovviamente, come abbiamo visto al capitolo precedente.

### 8.3 Esecuzione del test

Facciamo partire ora il test, dopo aver effettuato tutti i collegamenti, eseguendo per ogni calcolatore e Raspberry lo script corrispondente. Avviamo inoltre, allo stesso modo di come abbiamo visto nel capitolo precedente, i controller sui due Raspberry. Di seguito, troviamo una serie di immagini esplicative del test che si è condotto. Ricordiamo che c'è solo una funzione presente, dato il limite dato dalle porte USB presenti, che sarà QoSCheck, mappato alle porte dello switch 2 e 3.

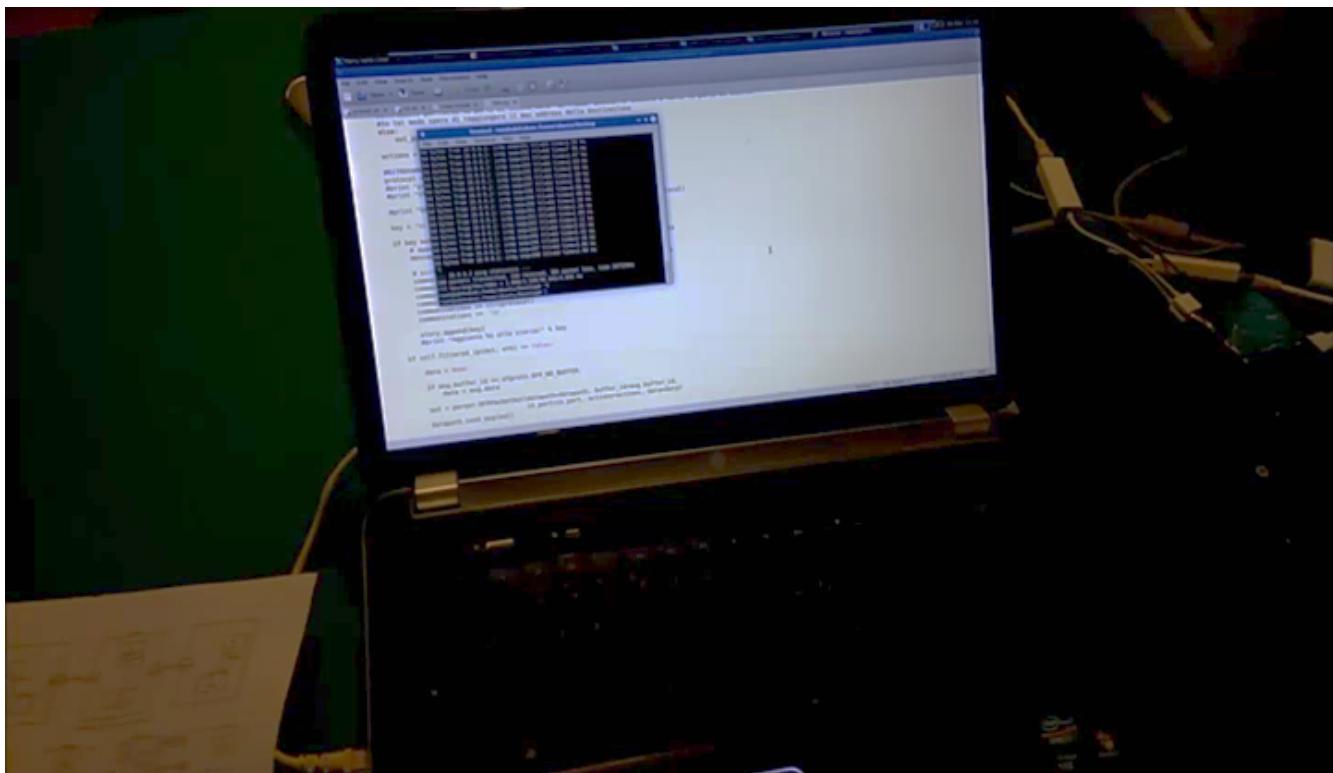


Fig. 39: calcolatore I

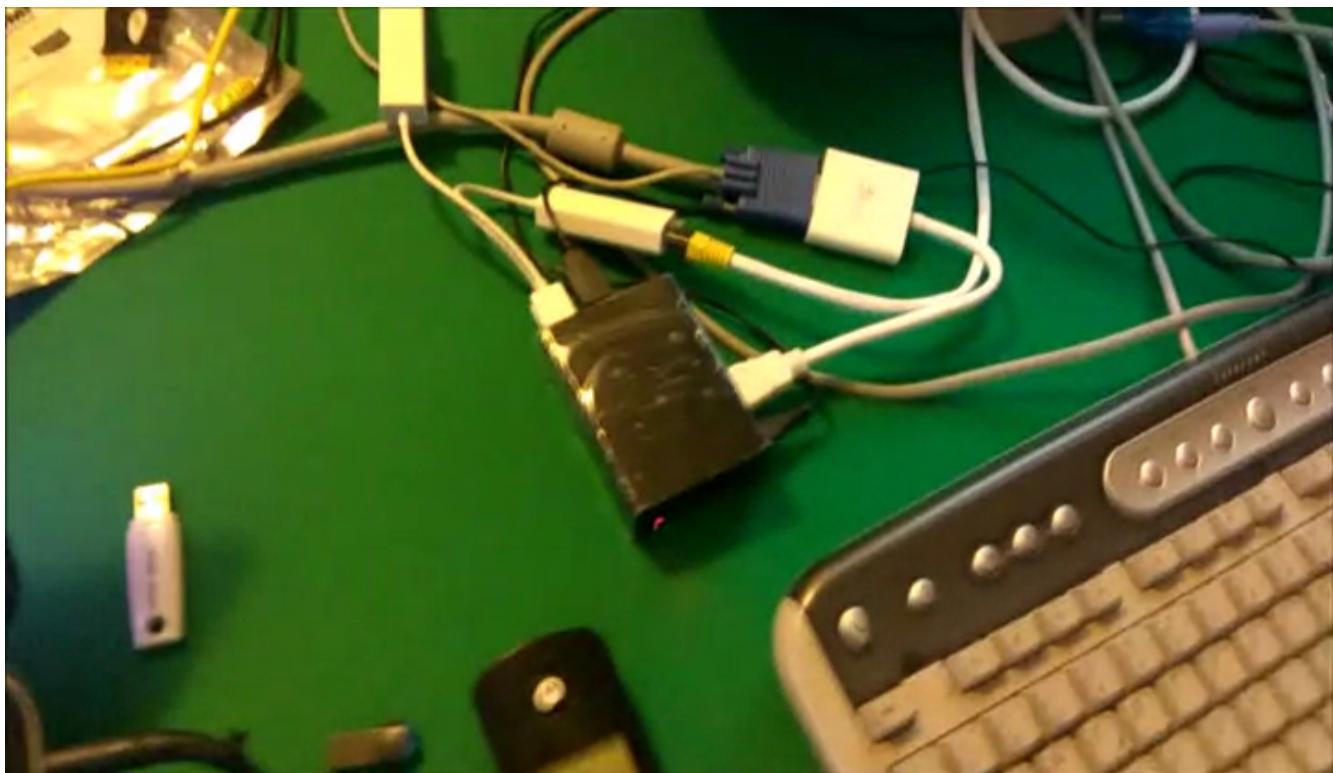


Fig. 40: Raspberry I

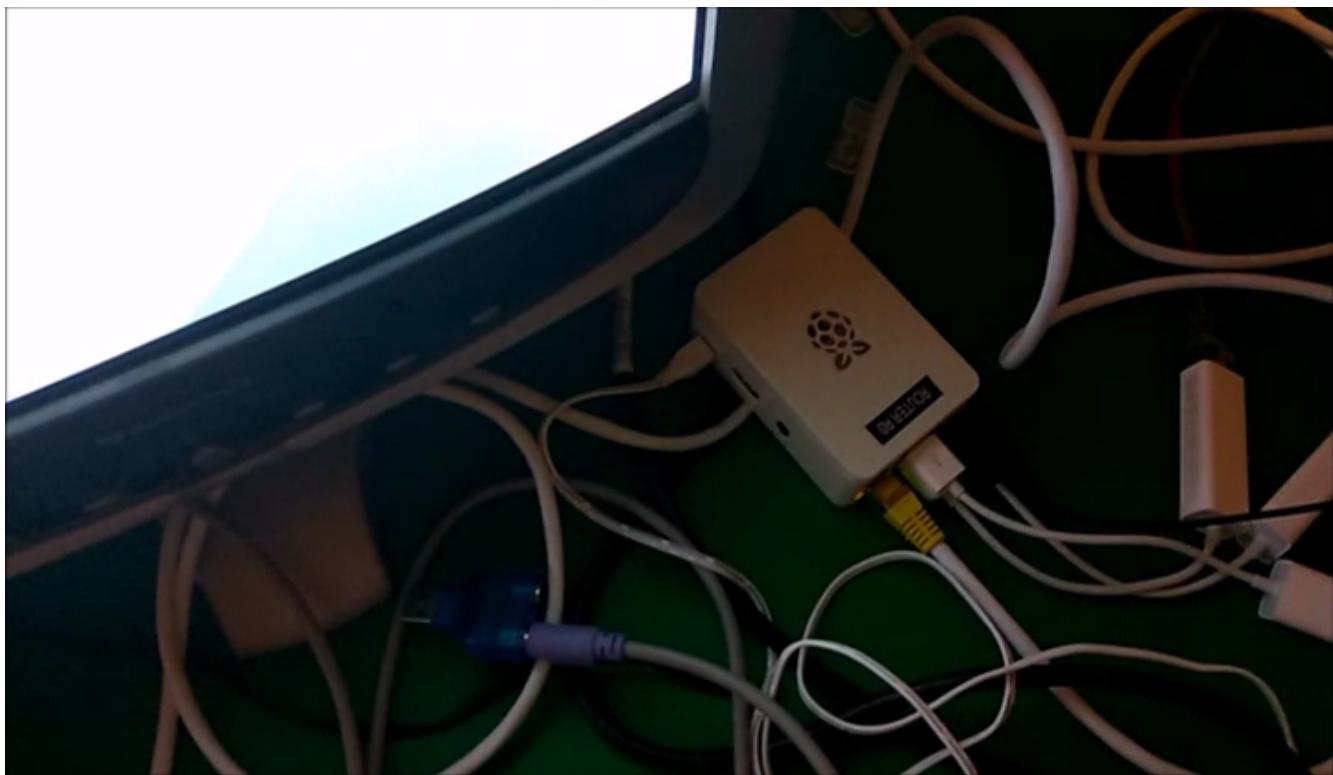


Fig. 41: Raspberry 2



Fig. 42: calcolatore 2

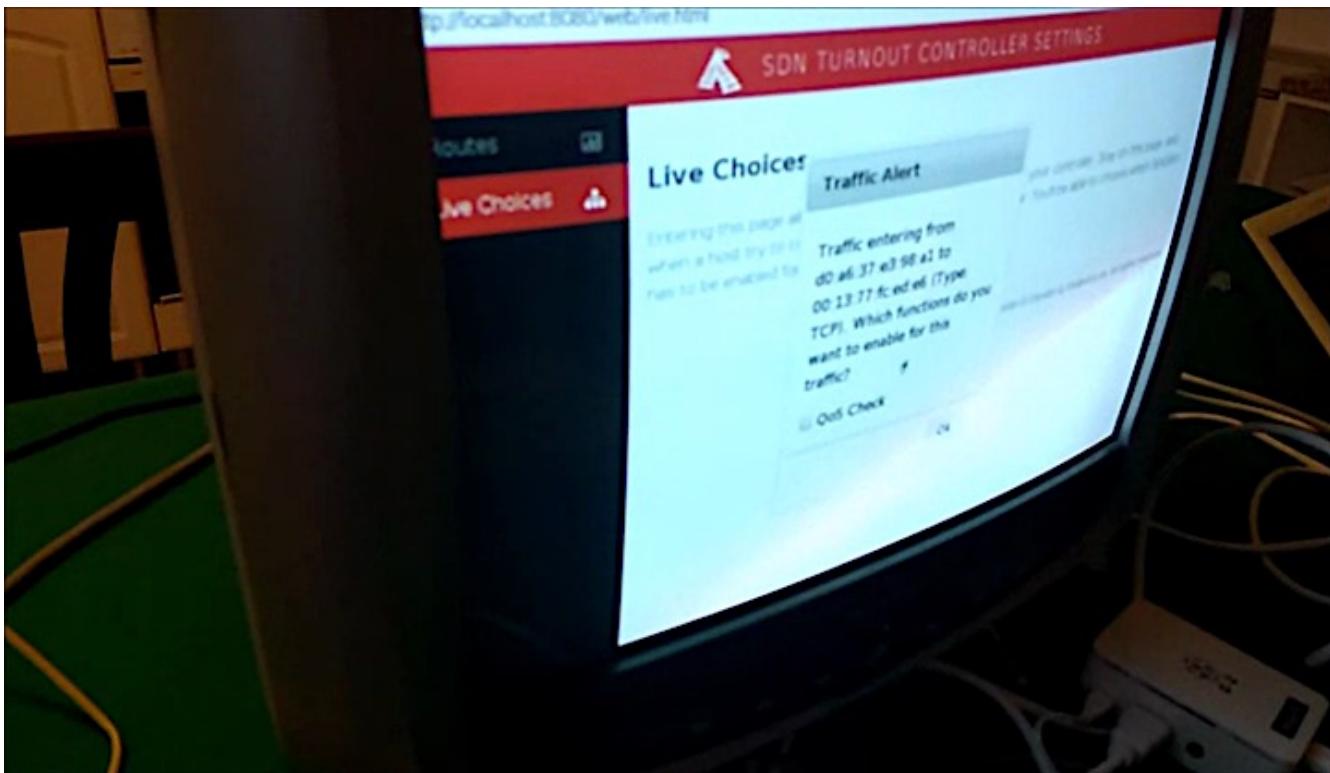


Fig. 43: primo traffico in entrata sul controller 2; il controller 1 tenta di comunicare con il calcolatore 2, dato che lo switch all'interno di quel calcolatore viene gestito da lui, ma non ci riesce, perché il traffico è ovviamente bloccato dal Raspberry 2: per poterli far comunicare, dobbiamo accettare il traffico

Fig. 44: viene fatto partire il ping da h1 nel calcolatore 1 verso h2 nel calcolatore 2

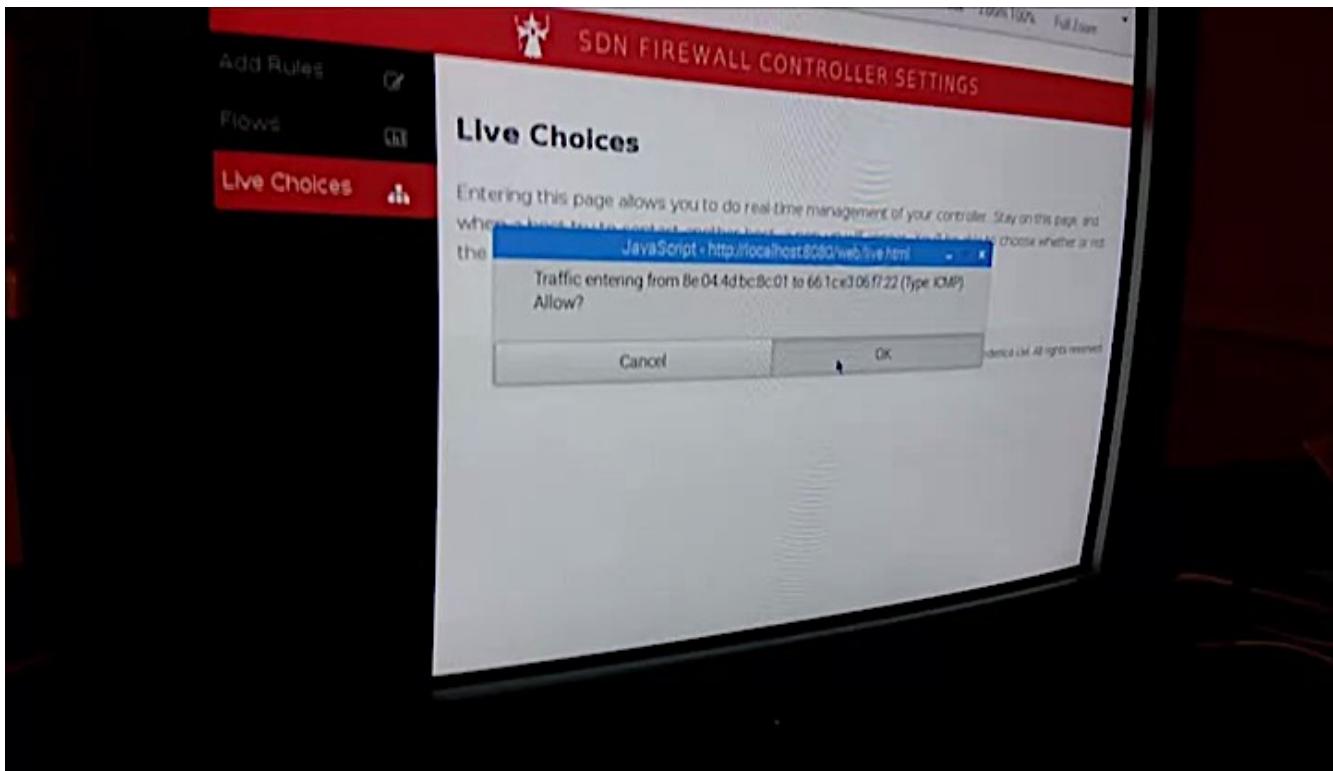


Fig. 45: viene accettato il traffico di ping proveniente da h1 dal controller 1

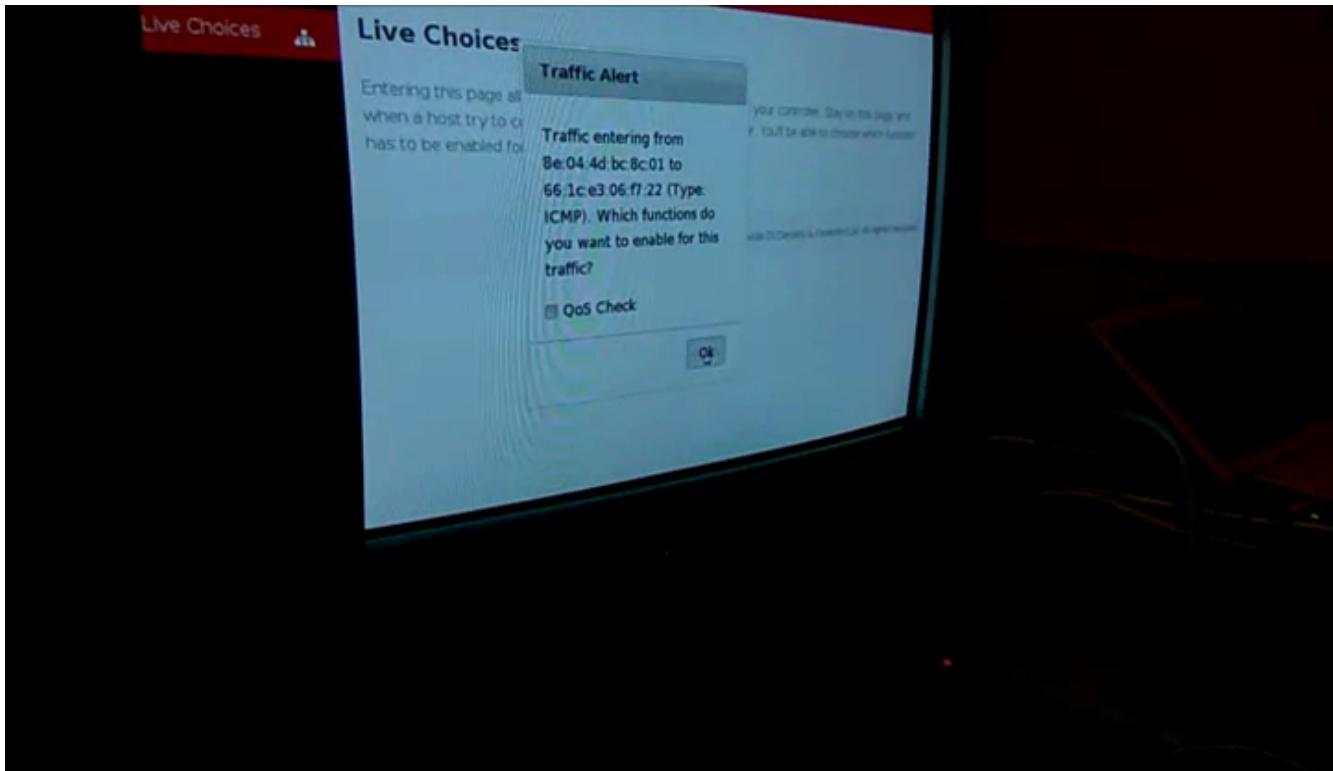
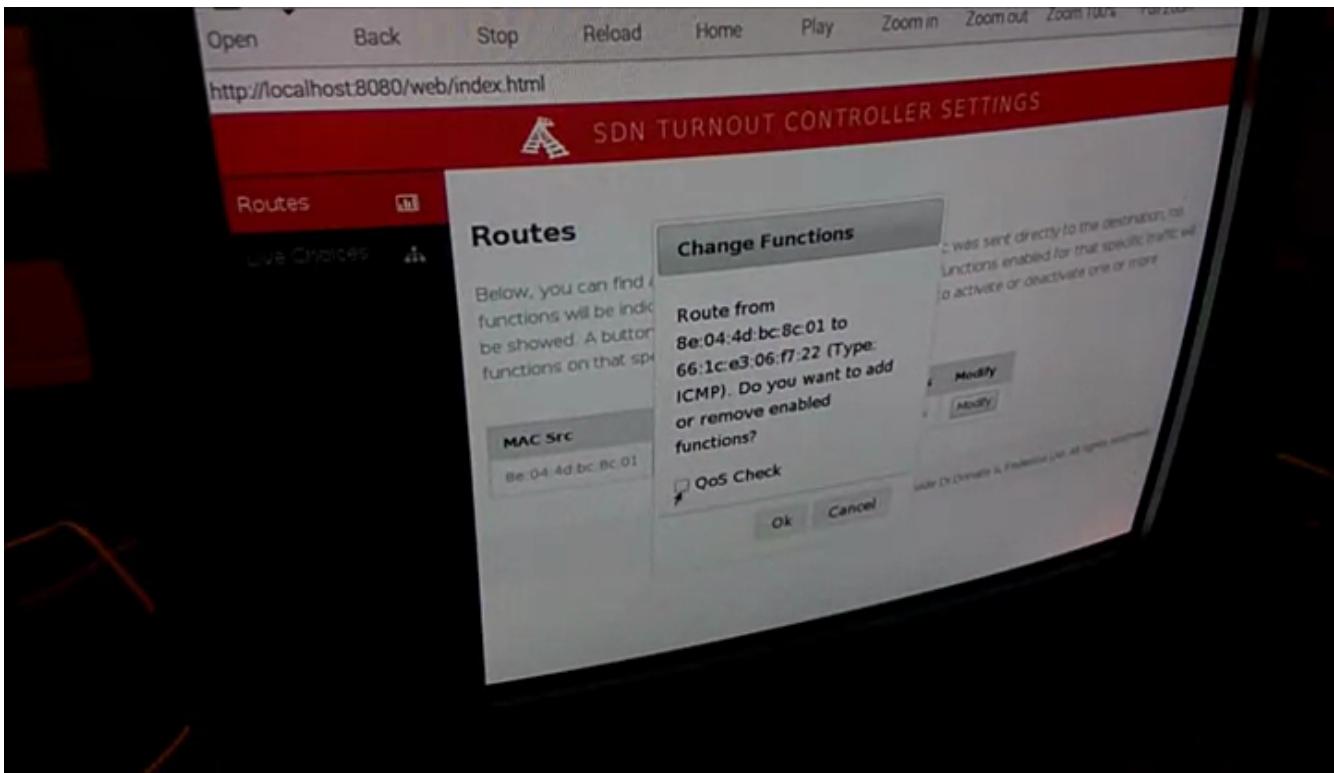


Fig. 46: viene impostata la rotta per il traffico di ping, appena accettato dal controller 1 e ora arrivato al controller 2, partito da h1, attivando l'unica funzionalità disponibile QoS Check

*Fig. 47: ora il ping, controllando su h1, passa*



*Fig. 48: andiamo a disabilitare ora la funzione attivata precedentemente...*

*Fig. 49: ...e possiamo vedere, su h1, che il ping passa ancora*

# CAPITOLO 9

## CONCLUSIONI

Terminati ora i test, passiamo a dare le ultime considerazioni e pensieri riguardo al lavoro fatto e come è possibile, magari, migliorarlo.

### 9.1 Test e risultati ottenuti

Si è riuscito ad ottenere nella rete reale, come si è visto, il funzionamento che ci si aspettava. In particolare, abbiamo riscontrato un picco di lavoro per i Raspberry che si assesta intorno al 30%: è da considerare che si è usato un browser leggero per effettuare le operazioni (kweb), e, nel frattempo, erano in esecuzione un'emulazione di rete (per simulare la presenza di uno switch) e l'applicazione RyuManager, che si è occupato di caricare i due moduli della nostra controller application (my\_fileserver.py e turnout\_rest.py). Tutto sommato, è da dire che è un buon indice: di sicuro, il Raspberry sarà in grado di supportare controller application formate da molti più moduli e molto più complesse.

Per quanto riguarda i problemi derivanti dai cortocircuiti (dato il loop), siamo riusciti, attraverso i comandi di no-flood, a garantire il funzionamento della rete e a scremare gran parte del traffico presente all'inizio che rendeva le comunicazioni impossibili tra i due end-point.

### 9.2 Sviluppi futuri

Di sicuro, un'estensione possibile al progetto che salta immediatamente all'occhio è quella di creare una controller application che sia in grado di gestire (con un controller centralizzato quindi) più switch, non solo uno.

Inoltre, è da studiare attentamente la gestione corrente del traffico di Discovery (ARP, LLDP, e così via), poiché viene consentito indistintamente: un attaccante, inseritosi in un qualsiasi punto nella rete, sarebbe in grado di usare proprio questo traffico per scoprire facilmente la topologia della rete e ricavare informazioni sull'hardware presente in rete. È possibile pensare di realizzare un computer di elaborazione, ad esempio, che si agganci allo switch gestito dal controller di questa tesi, che funzioni da sonda, in modo da individuare agilmente anomalie del traffico passante, e comunicare, in un qualche modo, lo stato anomalo al controller, che provvederebbe così a manovre protettive al fine di oscurare la struttura della rete ad un attaccante; ad esempio, implementando un algoritmo MTD, citato proprio nel primo capitolo di questa tesi.

Più in generale, è possibile pensare quindi di creare computer di elaborazione che svolgono una qualsivoglia funzione sul traffico: grazie all'automazione introdotta dal concetto di SDN e OpenFlow, le possibilità sono davvero molte, e alcune configurazioni di rete (come implementare un algoritmo MTD), che prima comportavano un lavoro troppo complesso e oneroso, ora vengono semplificate in maniera decisiva.

Infine, un'ultima nota è di sicuro da riservare al traffico che giunge dal controller 1, che punta a stabilire la connessione TCP/TLS con h2 presente all'interno del calcolatore 2: sarebbe sicuramente utile nascondere il dettaglio all'utente, al fine di abilitare automaticamente, senza intercessione dell'utente stesso, il traffico passante, poiché è una connessione che di sicuro si vuole stabilire, e, molto probabilmente, non si ha necessità di attivare funzioni di analisi proprio su questo traffico (anche se, non è del tutto da escludere).

### **9.3 Riflessioni finali**

In questo lavoro di tesi abbiamo applicato quindi il concetto di SDN su due reti diverse, una virtuale, l'altra reale. Attraverso l'architettura di Ryu, pensata per la creazione di una controller application attraverso l'aggiunta e la rimozione di moduli che la vanno a comporre, abbiamo sviluppato diverse REST API, al fine di consentire all'utente di impostare e modificare rotte per dei traffici in ingresso, oltre che a visualizzarle.

## BIBLIOGRAFIA

- [1] SDN, Wikipedia: [https://en.wikipedia.org/wiki/Software-defined\\_networking](https://en.wikipedia.org/wiki/Software-defined_networking)
- [2] Che cos'è il controller Ryu, SDX Central:  
<https://www.sdxcentral.com/sdn/definitions/sdn-controllers/open-source-sdn-controllers/what-is-ryu-controller/>
- [3] OpenFlow, Wikipedia: <https://en.wikipedia.org/wiki/OpenFlow>
- [4] FAQ, ONF: <https://www.opennetworking.org/about/faqs>
- [5] Overview, Mininet: <http://mininet.org/overview/>
- [6] Home Page, Open vSwitch: <http://openvswitch.org>
- [7] Perché Open vSwitch, Open vSwitch: <http://docs.openvswitch.org/en/latest/intro/why-ovs/>
- [8] Conformance Test Specification per OpenFlow 1.3.4, ONF:  
<https://www.opennetworking.org/images/stories/downloads/working-groups/OpenFlow1.3.4TestSpecification-Basic.pdf>
- [9] FAQ riguardanti OpenFlow, Open vSwitch:  
<http://docs.openvswitch.org/en/latest/faq/openflow/>
- [10] SDN e OpenFlow, Cisco: <http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-59/161-sdn.html>
- [11] SDN e OpenFlow, Extreme Networks:  
<http://learn.extremenetworks.com/rs/extreme/images/Open-Flow-and-SDNs-wp.pdf>
- [12] Cos'è OpenFlow, SDX Central: <https://www.sdxcentral.com/sdn/definitions/what-is-openflow/>
- [13] OpenFlow, Flowgrammable: <http://flowgrammable.org/sdn/openflow/>
- [14] OpenFlow 1.3 Specifications, ONF:  
<https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>
- [15] Controller SDN, SDX Central: <https://www.sdxcentral.com/sdn/definitions/sdn-controllers/>
- [16] Ryu, The New Stack: <https://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/>

[17] Cos'è Open vSwitch, Open vSwitch: <http://docs.openvswitch.org/en/latest/intro/what-is-ovs/>

[18] Switch compatibili con OpenFlow, Flowgrammable:  
[http://flowgrammable.org/sdn/openflow/#tab\\_switch](http://flowgrammable.org/sdn/openflow/#tab_switch)

[19] Emulatori di rete open-source, Brian Link Letter: <http://www.brianlinkletter.com/open-source-network-simulators/>

[20] Raspberry Pi, Wikipedia: [https://it.wikipedia.org/wiki/Raspberry\\_Pi](https://it.wikipedia.org/wiki/Raspberry_Pi)

[21] Raspberry Pi 2 Model B, Adafruit: <https://cdn-shop.adafruit.com/pdfs/raspberrypi2modelb.pdf>

[22] Raspbian, RaspberryPi: <https://www.raspberrypi.org/downloads/raspbian/>

[23] Ryu documentation, Ryu: <http://ryu.readthedocs.io/en/latest/>

[24] Macchina virtuale usata, SDNHub: <http://sdnhub.org/tutorials/sdn-tutorial-vm/>

[25] CableCard, Wikipedia, <https://en.wikipedia.org/wiki/CableCARD>

[26] Content-addressable memory, Wikipedia: [https://en.wikipedia.org/wiki/Content-addressable\\_memory#Ternary\\_CAMs](https://en.wikipedia.org/wiki/Content-addressable_memory#Ternary_CAMs)

[27] ASIC, Wikipedia: [https://en.wikipedia.org/wiki/Application-specific\\_integrated\\_circuit](https://en.wikipedia.org/wiki/Application-specific_integrated_circuit)

[28] Circuito diretto numerico, Wikipedia:  
[https://it.wikipedia.org/wiki/Circuito\\_diretto\\_numerico](https://it.wikipedia.org/wiki/Circuito_diretto_numerico)

[29] Multi Protocol Label Switching, Wikipedia:  
[https://it.wikipedia.org/wiki/Multi\\_Protocol\\_Label\\_Switching](https://it.wikipedia.org/wiki/Multi_Protocol_Label_Switching)

[30] Generic Routing Encapsulation, Wikipedia:  
[https://en.wikipedia.org/wiki/Generic\\_Routing\\_Encapsulation](https://en.wikipedia.org/wiki/Generic_Routing_Encapsulation)

[31] VLAN, Wikipedia: <https://it.wikipedia.org/wiki/VLAN>