

ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Analisi e sviluppo di un'interfaccia web per gestire controller SDN

Relazione finale in

Programmazione di reti

Relatore

Prof. Franco Callegati

Presentata da

Nicholas Brasini

Sessione II

Anno Accademico 2016/17

INDICE

Capitolo 1 - Introduzione	4
Capitolo 2 - Software Defined Networking	5
2.1 Storia delle Software Defined Networking	6
2.2 Architettura delle Software Defined Networking	7
2.3 Modalità di implementazione del control plane	9
2.4 Modalità di implementazione del data plane	11
Capitolo 3 - Protocollo OpenFlow	12
3.1 Funzionamento del protocollo OpenFlow	12
Capitolo 4 - Gli strumenti utilizzati	17
4.1 Introduzione ad Open vSwitch	17
4.2 Struttura di Open vSwitch	18
4.3 Life cycle di un pacchetto all'interno di Open vSwitch	19
4.4 Introduzione a Mininet	20
4.5 Caratteristiche di Mininet	20
4.6 Configurazione di Mininet	21
4.7 Introduzione a Ryu	22
4.8 Storia dei controller e di Ryu	22
4.9 Struttura di Ryu	23
4.10 Configurazione di Ryu all'interno di Mininet	25
Capitolo 5 - Primi test con tutti gli strumenti	27
5.1 Progettazione della rete SDN	27
5.2 Verifica della connettività e analisi Wireshark	27
5.3 Chiamate REST tramite Postman	28
Capitolo 6 - Realizzazione del progetto	33
6.1 Strumenti utilizzati per la web app	33
6.2 File utilizzati e astrazione del controller	34
6.3 Casi d'uso	35
6.4 Home web app e relativi script	36
6.5 Settaggio degli script	38
6.6 Pagina switch.html e relativi script	40

6.7 Pagina flowSwitch.html e relativi script	46
6.8 Pagina addRule.html e relativi script	51
6.9 Pagina controller.html e relativi script	54
6.10 Analisi del file reset.js	58
6.11 Analisi del file restRyu.js	59
6.12 Analisi del file functionRyu.js	61
Capitolo 7 - Conclusioni	62
7.1 Sviluppi futuri	62

Capitolo 1 - Introduzione

Sono quasi giunto al termine di questa “avventura”, che mi ha portato a conoscere persone nuove e mi ha dato la possibilità di avventurarmi sempre più all’interno del mondo tecnologico. Ho passato tre anni intensi che mi hanno permesso di riflettere sul mio futuro e su quale campo mi piacerebbe approfondire nel corso della Laurea Magistrale. A partire da settembre scorso ho iniziato a studiare con convinzione e passione il settore relativo alle reti. Così, dopo aver svolto il tirocinio presso un’azienda che si occupava di sicurezza delle reti, ho deciso, insieme al professor Callegati, di focalizzare la mia Tesi della Laurea Triennale proprio in questo settore. Il professore mi ha parlato e proposto l’argomento relativo alle reti SDN (Software Defined Networking), un approccio relativamente nuovo nell’amministrazione delle reti, mirato ad una gestione più precisa, scalabile e dinamica delle stesse, attraverso livelli di astrazione di funzionalità di base [1]. L’argomento mi è parso subito interessante, soprattutto perché, oltre allo studio delle reti SDN nella loro vasta complessità, il professore mi ha proposto di realizzare una parte progettuale che mi permettesse di comprendere più a fondo l’argomento. Per questo, dopo una breve parentesi relativa ai concetti e agli strumenti utilizzati, approfondirò lo studio relativo al progetto, che consiste nella realizzazione di un’interfaccia web che permetta ad un utente che si affaccia per la prima volta nel mondo delle reti SDN di comprenderne il funzionamento attraverso una serie di funzionalità proposte. Inizialmente non ero a conoscenza della rete SDN e delle possibilità che garantiva una volta messa in atto. Mi sono reso conto però, studiando ma soprattutto mettendo in pratica quanto imparato, di come possa davvero essere rivoluzionaria nell’ambito delle reti per tutte le possibilità che offre. E’ per questo che, nel corso dello studio personale per realizzare questa Tesi di Laurea, mi sono soffermato, in maniera piuttosto interessata, a carpire e comprendere tutti i dettagli di questa tecnologia, che in un futuro sempre più vicino potrebbe sostituire le attrezzature odierne.

Capitolo 2 - Software Defined Networking

Prima di approfondire l'argomento relativo alle reti SDN, è doveroso mettere in luce come funziona l'attuale architettura di rete. Principalmente si può immaginare una rete come un insieme di router o switch in successione, ognuno dei quali è in grado, autonomamente, di poter gestire l'arrivo dei pacchetti in entrata e di poterli ridirezionare in uscita in base ad una serie di algoritmi di routing specifici. Ogni singolo dispositivo è programmato per funzionare in un certo modo imposto dalla casa costruttrice in base al firmware montato sullo specifico hardware. E' evidente però come questa tecnologia sia piuttosto limitata per diversi motivi. Principalmente infatti, in situazione di grande traffico e congestione sulla rete, questi dispositivi non riescono ad elaborare sempre in maniera ottimale tutti i pacchetti in ingresso e questo causa notevoli problemi all'interno della rete. Un altro problema piuttosto serio riguarda la flessibilità di queste reti. E' ovvio che, potendo essere router e switch diversi gli uni dagli altri, vadano eventualmente modificati e aggiornati secondo specifiche caratteristiche. In questo modo, qualora si dovessero gestire reti che cambiano dinamicamente, la perdita di flessibilità sarebbe importante e questo non consentirebbe di evitare problemi di sicurezza e velocità. Se si volesse ad esempio modificare un solo elemento, potrebbe essere necessario dover variare e modificare tutta la topologia della rete oppure riconfigurare tutti i dispositivi ad essa connessi. Considerati questi due macro settori, non si può non specificare poi come la manutenibilità della rete di conseguenza ne risenta: ogni dispositivo potrebbe essere configurato con uno specifico linguaggio a seconda dell'azienda produttrice e diventerebbe complesso gestire il tutto in maniera performante. Senza considerare poi che, ad oggi, l'architettura client-server sia considerata obsoleta per le richieste che pervengono ogni giorno alle reti. A causa soprattutto dei dispositivi mobile, la richiesta di risorse è continua e ci si aspetta che venga esaudita nel più breve tempo possibile ma soprattutto in maniera sicura. Per questo bisogna fare in modo che la rete sia flessibile dal punto di vista della grandezza, ovvero scalabile, ma soprattutto sicura e reattiva. Per risolvere questo problema e tutti quelli precedentemente elencati, è necessario capire quale sia la causa scatenante di tutto ciò. La risposta è che il *data plane*, ovvero la gestione della transizione dei pacchetti all'interno di router o switch, e il *control plane*, ovvero la scelta di come gestire suddetti pacchetti e come reindirizzarli grazie agli algoritmi di routing, vadano separati per poter permettere una gestione migliore della rete stessa.

2.1 Storia delle Software Defined Networking

La storia relativa alle Software Defined Networking può essere ricollegata all'avvento della tecnologia Java ad opera della Sun Microsystems, che nel 1995 rilasciò la prima versione del famosissimo linguaggio di programmazione. I primi progetti basati su SDN vennero messi a punto dalla compagnia texana AT&T, che nominò il proprio prodotto GeoPlex [2], e da un ingegnere della Sun, tal Medovich, il cui progetto prese il nome di WebSprocket [3].

Per quanto riguarda GeoPlex, si tratta di un prodotto basato sulle API di rete e sul linguaggio Java per creare un middleware di rete che fosse in grado di mappare gli IP dove venivano svolte attività di interesse in uno o più servizi. GeoPlex dunque era una raccolta di componenti e servizi aperti e standard che creavano una piattaforma sicura, scalabile, robusta e commerciale per applicazioni e servizi basati su IP. Integrava componenti che fornivano sicurezza e autenticazione, registrazione e gestione degli account. Il problema principale però fu relativo al fatto che GeoPlex non riuscì a stravolgere completamente la concezione della rete dell'epoca, pertanto non riuscì ad imporsi rimanendo nel limbo della tecnologia. Per quanto riguarda invece il secondo progetto, questo era composto principalmente da due entità: un NOS (Network Operating System) e un nuovo modello orientato agli oggetti (anche in questo caso basato su Java) che potesse essere modificato in real time da un compilatore. In questo modo si potevano creare applicazioni come thread Java che ereditassero classi, kernel e network di WebSprocket per poter essere riutilizzate successivamente per interfacce e protocolli propri. Nel 2001 venne realizzato il primo reale test di una rete SDN ma solamente nel 2007 si giunse al concetto odierno di SDN grazie al lavoro di un folto gruppo di società. In questo lasso di tempo però continuarono le ricerche e nel 2003 grazie al lavoro svolto da Burke e Carman vennero rilasciati i primi brevetti statunitensi relativi a questa tecnologia. Nel 2011 venne fondata l'Open Networking Foundation (ONF) per promuovere le reti SDN e la standardizzazione del protocollo OpenFlow.

2.2 Architettura delle Software Defined Networking

A questo punto entra in gioco il modello SDN, che ha come obiettivo di base quello di garantire una scissione totale tra il *data plane* e il *control plane*. L'idea è che ci sia un'entità che stia al di sopra dei dispositivi di rete come router e switch e che si prenda carico delle decisioni più importanti relative ai pacchetti in transito. L'entità descritta è rappresentata dal controller, che avrà dunque il compito di gestire i pacchetti e decidere cosa farne una volta giunti allo switch o al router di competenza. In questo modo i dispositivi di rete dovranno preoccuparsi unicamente di ricevere i pacchetti in entrata (nel caso di switch SDN puro), nel cosiddetto “*flow*”, flusso di pacchetti, e stabilire se ci sono delle azioni preimpostate per quello specifico flusso. Qualora non ci fosse alcuna azione specifica, il pacchetto verrà inviato al controller tramite uno specifico protocollo, che tratterò in seguito. Una volta analizzato dal controller, questo lo rimanderà allo switch e gli imporrà un certo tipo d'azione per tutti i pacchetti successivi simili a quello appena processato. A quel punto il device potrà indirizzarlo verso la destinazione prefissata. Ma come è composta l'architettura di una rete SDN? Principalmente è divisa in tre “layer”: *Infrastructure Layer*, *Control Layer* e *Application Layer*. [4]

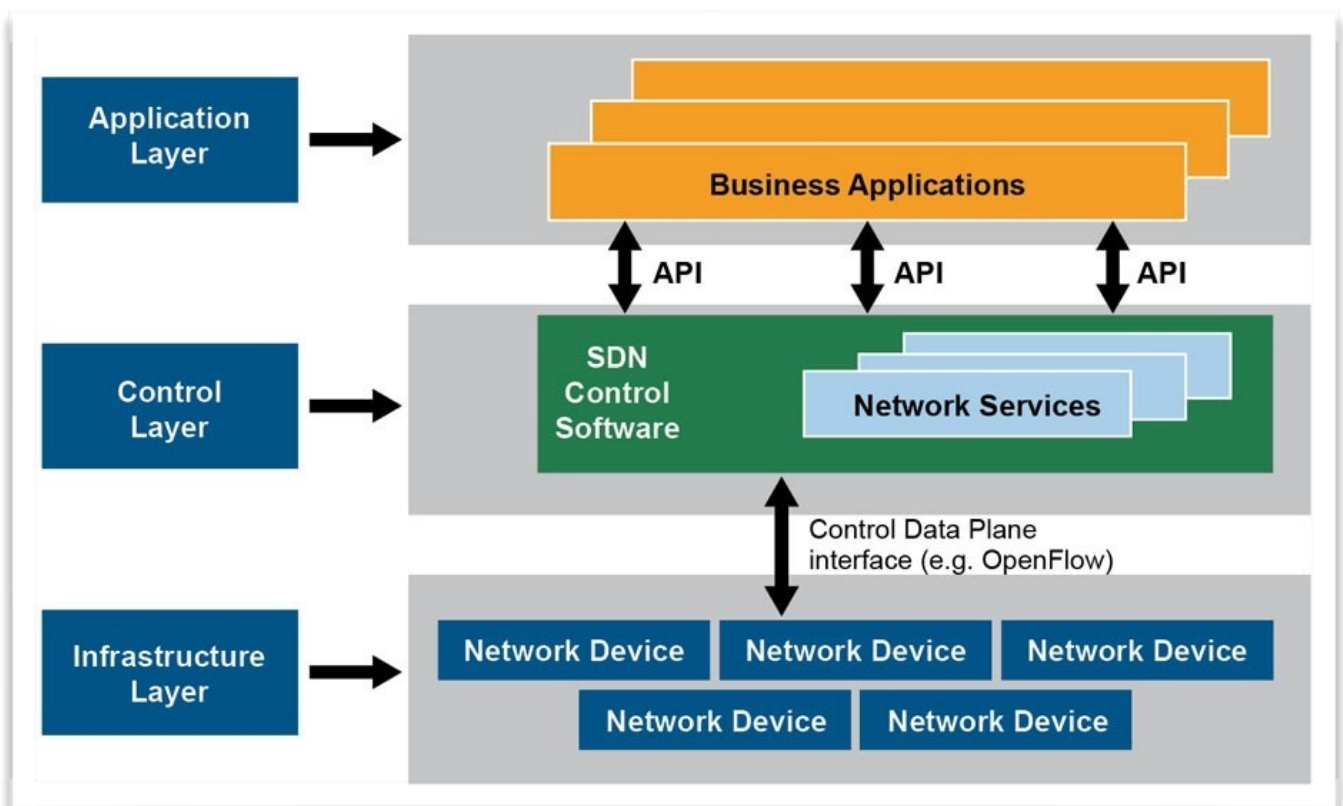


Illustrazione 1: architettura SDN

- **Infrastructure Layer.** Si tratta del livello più basso dell'architettura SDN. Fanno parte di questo livello tutti i dispositivi di rete come ad esempio switch e router che hanno come obiettivo la trasmissione dei pacchetti da un nodo all'altro. Si può notare come, giunti a questo punto, non sia più necessario che questi dispositivi siano "intelligenti", ovvero non dovranno implementare particolari algoritmi di routing per gestire il forwarding dei pacchetti. Dal momento che il *data plane* e il *control plane* sono separati, dovranno solamente essere in grado di inviare e ricevere flussi di pacchetti senza dover decidere cosa farne, dal momento che queste scelte saranno prese dal controller. Per poter comunicare con il controller dunque sarà sufficiente che all'interno di questi dispositivi siano presenti agent preposti a creare canali di collegamento sicuri con il layer successivo.
- **Control Layer.** E' il livello intermedio ed è qui che risiede il controller, l'entità fondamentale dell'architettura SDN e vero valore che differenzia una rete classica da quella delle Software Defined Networking. Il controller ha il compito di filtrare tutti i pacchetti che gli vengono recapitati dal livello inferiore in base ad una serie di strategie. Uno dei fattori decisivi riguarda la dinamicità: il controller non ha problemi per quanto riguarda la scalabilità, dal momento che, anche se si dovessero aggiungere dispositivi di rete al livello sottostante, sarebbe in grado di gestirne l'ingresso e di programmarli senza bisogno di interagire o modificare tutta la topologia di rete. Il controller potrà essere programmato in base alle esigenze e sarà fondamentale per il corretto funzionamento di tutta la rete. Per quanto riguarda il collegamento con l'*infrastructure layer*, questo sarà possibile grazie al protocollo OpenFlow, ovvero il protocollo più utilizzato per permettere lo scambio di messaggi tra i due livelli.
- **Application Layer.** Si riferisce alle applicazioni che sono direttamente fruibili dagli utenti finali. Rappresenta la parte più intelligente di una rete SDN e ha come scopo quello di gestire il reale funzionamento e controllo della rete tramite interfacce grafiche e strumenti dedicati all'utente finale.

2.3 Modalità di implementazione del control plane

Come detto in precedenza, lo scopo principale delle reti SDN è quello di creare un disaccoppiamento tra il *data plane* e il *control plane* per permettere una gestione dinamica della rete. Ma come vengono implementati nella pratica? Il *control plane* è relativo al secondo livello dell'architettura SDN e può essere gestito in tre diverse modalità: *centralizzata*, *decentralizzata* e *gerarchica*. La prima viene implementata tramite la realizzazione di un unico controller SDN che dovrà gestire tutta la rete. E' chiaro che rappresenti la modalità più diretta ma nello stesso tempo è evidente che non sia la migliore considerando l'enorme quantità di traffico, la dinamicità e la scalabilità che vengono richieste. La seconda soluzione prevede che ci siano più controller decentralizzati in grado di gestire la rete mentre la soluzione gerarchica è un ibrido delle due descritte in precedenza, con un controller centralizzato che gestirà i controller decentralizzati sotto le sue dipendenze. La soluzione migliore probabilmente è l'ultima: ogni controller decentralizzato avrà una porzione di rete che dovrà gestire ma potrà, nello stesso tempo, dialogare con altri controller. Le decisioni relative alla globalità della rete verranno invece prese dal controller centralizzato.

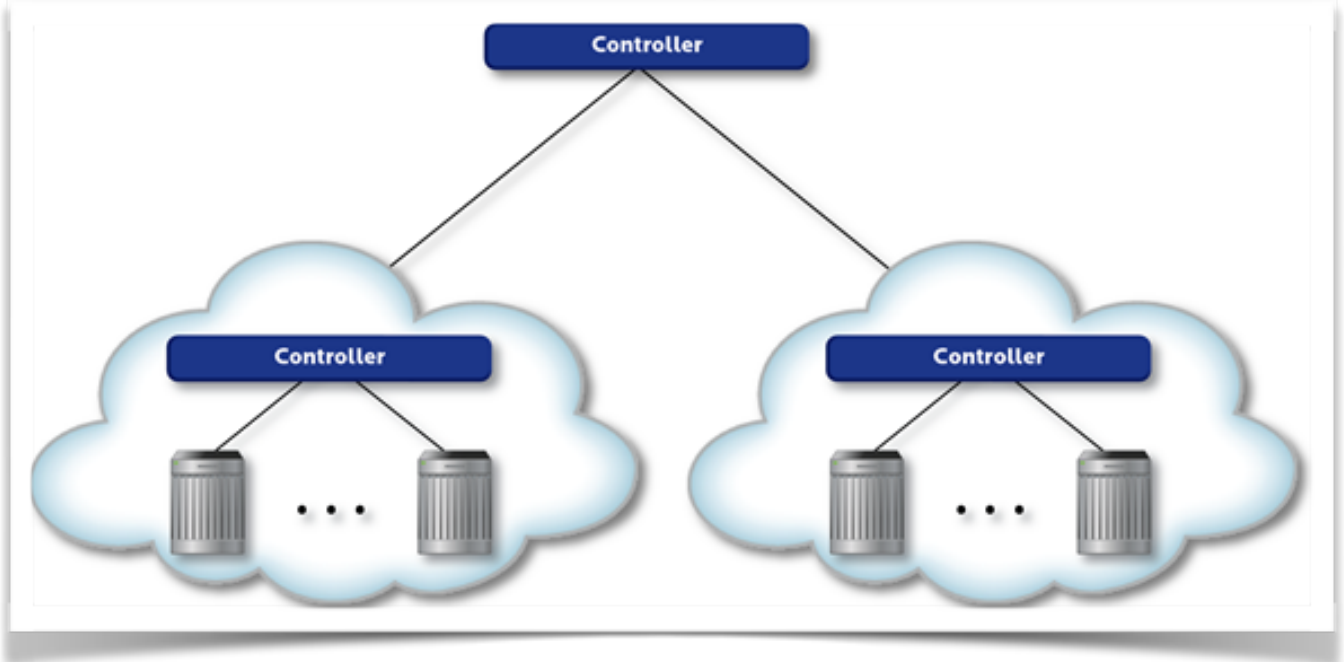


Illustrazione 2: control plane gerarchico

Per quanto riguarda invece il ruolo di ciascun controller nei confronti dello switch, ci sono tre possibilità: *equal*, *master* e *slave*. Bisogna sottolineare prima di tutto che ogni controller può ricoprire più cariche quando dialoga con switch diversi. Ad esempio il controller c0 potrebbe essere master per lo switch s1 ed equal per lo switch s2. Analizziamo ora i diversi ruoli che può ricoprire un controller:

- **SLAVE.** E' il ruolo con minor raggio d'azione. Un controller slave può accedere allo switch solo in modalità lettura, non può inviare comandi per modificare lo stato dello switch o per spedire pacchetti. Qualora un controller slave tentasse di inviare un comando a lui proibito, lo switch relativo gli risponderà con un messaggio d'errore. E' chiaro che se l'unico controller che fa riferimento alla rete SDN viene settato come slave, nessun pacchetto riuscirà a giungere a destinazione in quanto il controller slave non può installare alcun flusso sullo switch, pertanto non riuscirà mai a permettere il passaggio dei pacchetti tra host.
- **EQUAL.** E' il ruolo di default di un controller quando viene avviato lo switch. Un controller equal può accedere allo switch sia in lettura che in scrittura, può inviare comandi, modificarne lo stato e ricevere tutti i messaggi asincroni generati dagli switch.
- **MASTER.** E' il ruolo più importante all'interno della gerarchia dei controller. Non tanto per le caratteristiche e le azioni che può compiere, poiché possiede le stesse features di un controller equal. Però all'interno di una rete ogni switch può avere un solo controller master, mentre di equal e slave può averne anche di più contemporaneamente. Per poter gestire questo fatto, qualora lo switch volesse eleggere un controller a master, dovrà prima preoccuparsi di cambiare lo stato dell'attuale master in slave e poi eleggere il nuovo controller master tramite un messaggio.

Lo switch può gestire il cambio dei ruoli di tutti i controller e per poterlo fare deve ricevere dal singolo controller un messaggio di tipo Role-Request, che dovrà contenere il ruolo e il `generation_id`. Quest'ultimo è un contatore incrementale che viene confrontato, nel momento dell'arrivo, con il `generation_id` presente all'interno dello switch. Qualora il primo fosse inferiore al secondo, la richiesta non verrebbe considerata recente per cui scartata, altrimenti il controller assumerà il ruolo richiesto.

2.4 Modalità di implementazione del data plane

Dopo aver analizzato il *control plane*, bisogna capire come è strutturato il *data plane* in una rete SDN. Fondamentalmente, ogni dispositivo di rete che utilizza il protocollo OpenFlow (di cui parlerò nel paragrafo successivo) deve mantenere al proprio interno una o più *flow table* (tabelle di flusso) al cui interno saranno inseriti i flussi relativi a certe tipologie di pacchetti. Queste tabelle sono di tipo TCAM (Ternary Content-Addressable Memory, memoria ad alta velocità [5]) e vengono utilizzate per indirizzare i flow di pacchetti. Nel momento in cui un certo flusso perviene ad un dispositivo di rete, viene effettuato il lookup delle tabelle: se non si trova alcuna corrispondenza, allora il pacchetto verrà spedito al controller che deciderà cosa farne (eliminarlo oppure creare una regola per quello e i flussi successivi); se invece una regola è già stata inserita all'interno del dispositivo, verranno applicate le azioni per quel particolare tipo di flusso. Il controller può gestire queste azioni secondo tre modalità [6]:

- **Modalità reattiva.** Una volta che il controller ha ricevuto il pacchetto dal dispositivo di rete, crea una regola per quello e i successivi flussi e la inserisce nel dispositivo secondo la logica di controllo.
- **Modalità proattiva.** Il controller popola fin da subito tutte le tabelle di tutti i dispositivi della rete per tutti i possibili flussi.
- **Modalità ibrida.** Il controller inizialmente gestisce il traffico tramite la modalità proattiva, salvo poi attivare quella reattiva e gestire eventuali richieste indesiderate.

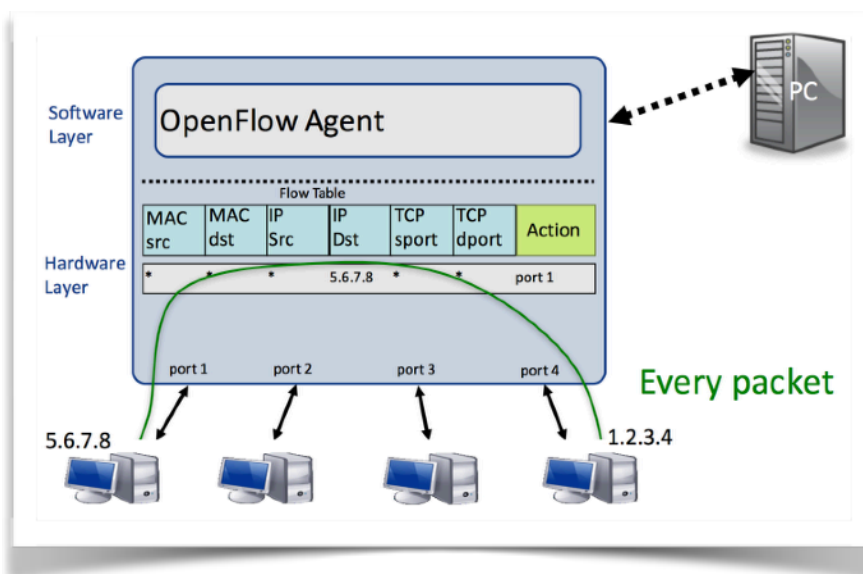


Illustrazione 3: modalità proattiva del controller

Capitolo 3 - Protocollo OpenFlow

Il protocollo OpenFlow nasce per poter permettere la comunicazione tra i dispositivi di rete ed il controller che ha come compito quello di gestire il *control plane* dell'architettura di rete. La ONF, l'organizzazione che gestisce lo standard OpenFlow, lo ha definito come la prima interfaccia di comunicazione tra il *control plane* e il *data plane* di un'architettura SDN [7]. Questo tipo di protocollo risulta essere assolutamente fondamentale all'interno delle reti SDN, ed è per questo motivo che, ad oggi, è anche il protocollo più utilizzato in questo settore. E' necessario per spostare il controllo della rete fuori da dispositivi di rete proprietari e portarlo in un software di controllo esterno. Venne sviluppato a partire dal 2007 e nel 2011 venne rilasciata la versione 1.1, mentre qualche mese più tardi la ONF approvò la versione 1.2 di OpenFlow e la pubblicò nel febbraio 2012. Ad oggi la versione più recente è la 1.5.

3.1 Funzionamento del protocollo OpenFlow

Come sottolineato nei paragrafi precedenti, il protocollo OpenFlow è il protocollo più utilizzato per la comunicazione tra dispositivi di rete e controller (altri protocolli conosciuti sono l'OVSDB, YANG e NetConf). Il protocollo dunque riesce a mettere in comunicazione due dei tre livelli dell'architettura SDN, ovvero l'*infrastructure layer* e il *control layer*. OpenFlow viene chiamato in causa non appena uno switch viene inserito all'interno della topologia SDN. Il primo passo da effettuare sarà quello della creazione di un canale di comunicazione sicuro (TCP/TLS) tra switch e controller. Una volta fatto ciò, sarà lo switch ad avviare la conversazione inviando un pacchetto di tipo OF_HELLO al controller, all'interno del quale specificherà quale versione sarà in grado di supportare (campo Version). Di conseguenza il controller risponderà a sua volta con un pacchetto OF_HELLO in cui stabilirà la versione definitiva del protocollo da utilizzare. Qualora non ci fosse una versione comune, lo switch non potrà entrare a far parte della rete e verrà notificato dal controller tramite un pacchetto di tipo OF_HELLO_FAILED_ERROR_MSG (vedasi *Illustrazione 4*). Dopo questo scambio iniziale di messaggi, il controller invierà un pacchetto di tipo OF_FEATURES_REQUEST che conterrà solamente un header. Lo switch risponderà con un pacchetto OF_FEATURES_REPLY all'interno del quale specificherà una serie di dati relativi al dispositivo di rete come ad esempio la capacità, la dimensione e un datapathID che lo identifichi univocamente. Successivamente per conoscere alcune informazioni come ad

esempio lo stato del componente, il numero di porte, i flussi del dispositivo ecc. si utilizzano pacchetti di tipo OF_MULTIPART_REQUEST e OF_MULTIPART_REPLY, anche se in alcune versioni del programma per “sniffare” il traffico Wireshark questi vengono identificati come OF_PORT_DESC_STATS_REQUEST e OF_PORT_DESC_STATS_REPLY (vedasi *Illustrazione 5*).

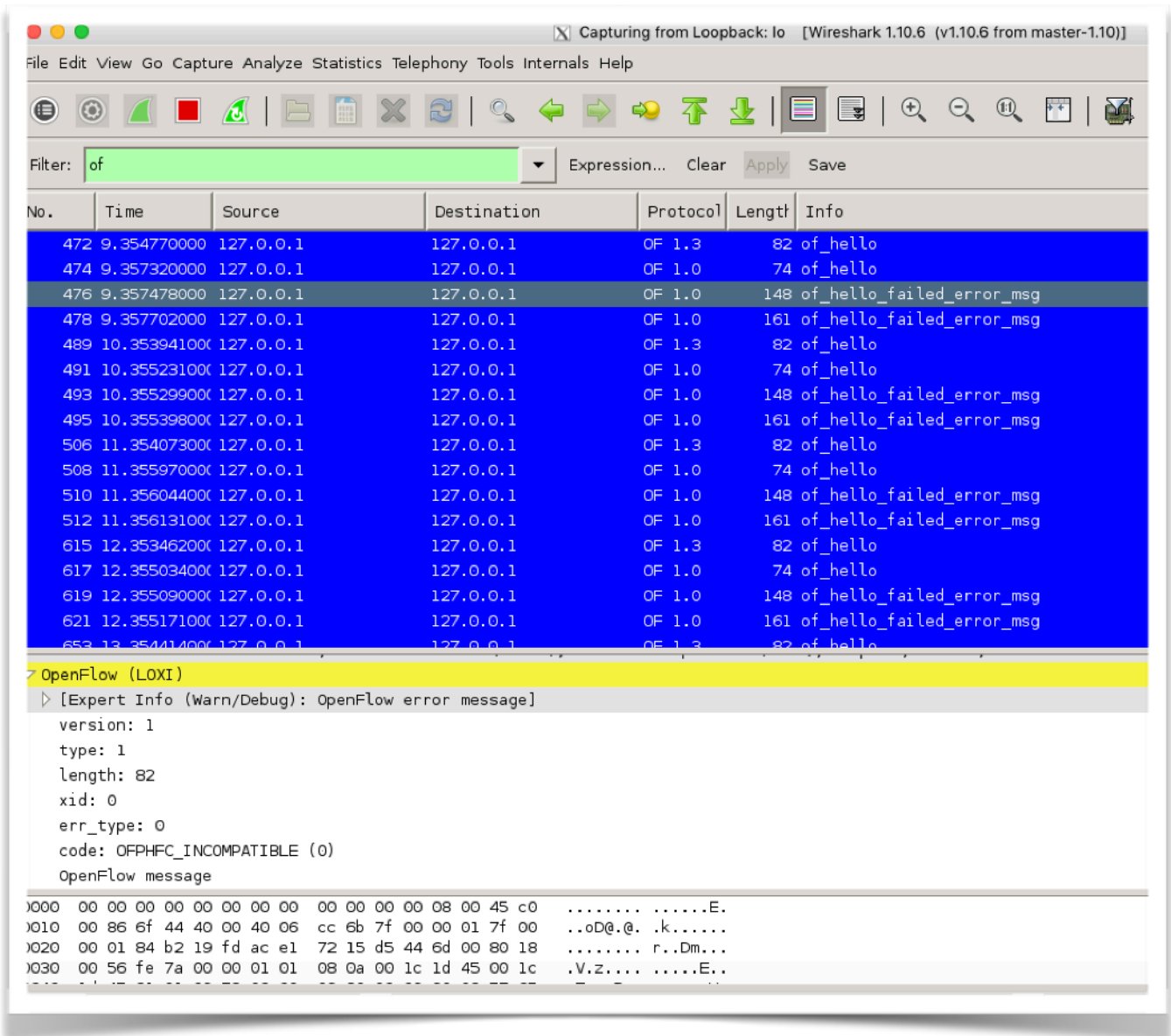


Illustrazione 4: versioni del protocollo incompatibili tra controller e switch

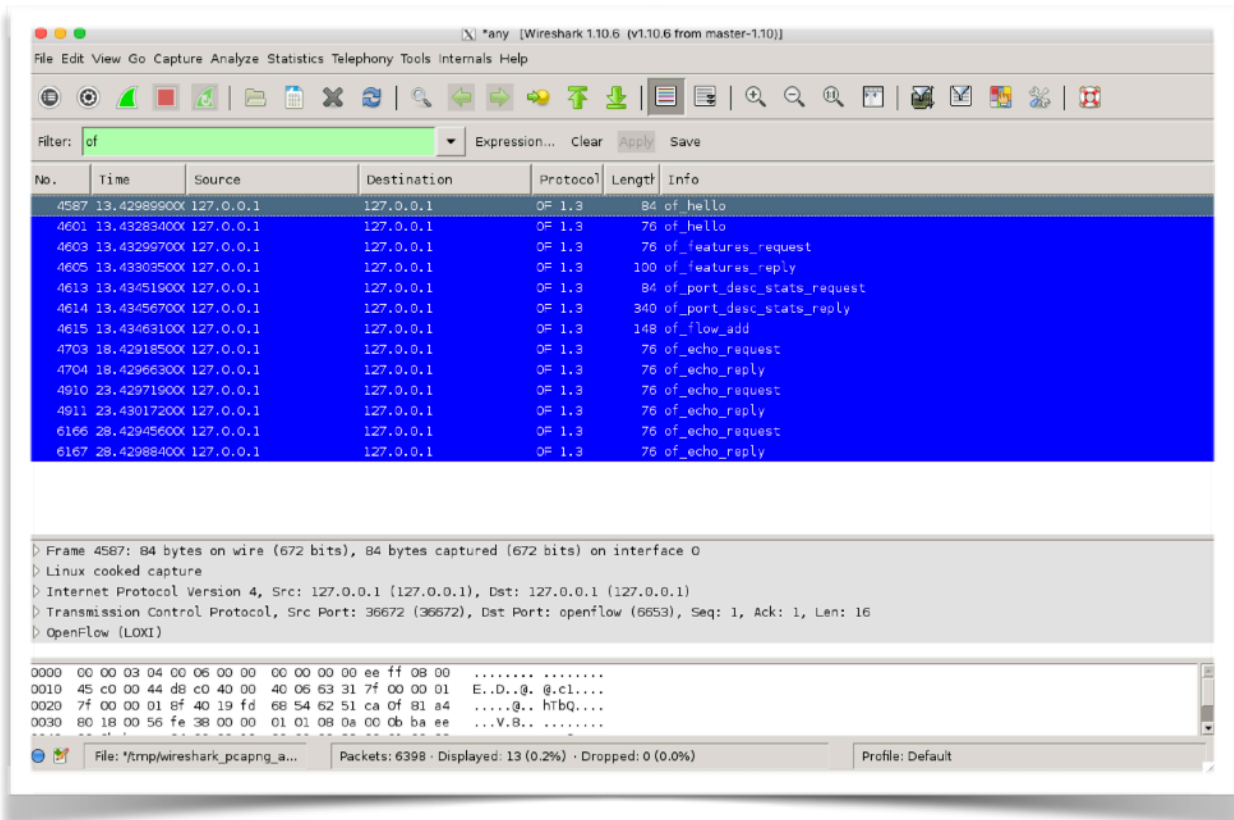


Illustrazione 5: configurazione iniziale tra switch e controller

Dopo aver effettuato correttamente la configurazione iniziale, il dispositivo di rete è pronto per ricevere i flussi di pacchetti in entrata. A questo punto bisognerà stabilire le modalità d'azione del controller (vedasi il paragrafo 2.5): nei test che ho effettuato ho utilizzato la versione 1.3 del protocollo OpenFlow e il controller ha agito in maniera reattiva. Non appena lo switch riceverà in ingresso un particolare flusso, controllerà immediatamente le proprie flow table alla ricerca di una corrispondenza con il pacchetto

arrivato: se non troverà azioni relative a quel pacchetto, allora di default invierà al controller un pacchetto di tipo OF_PACKET_IN. All'interno del packetIn ci saranno un header, un buffer_id che rappresenterà in maniera univoca l'id del pacchetto arrivato allo switch e altri campi (total_len, in_port, ecc).

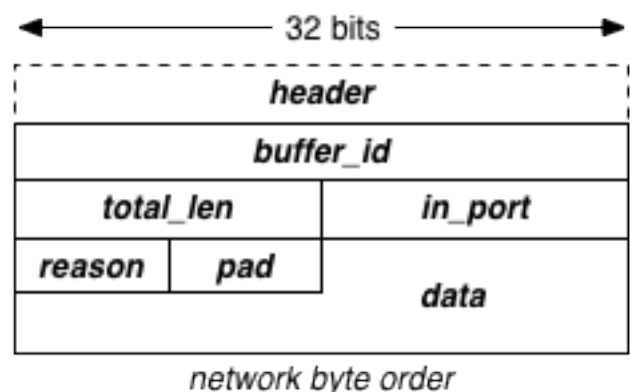


Illustrazione 6: struttura del packet_in

Una volta che il controller avrà ricevuto il packetIn, stabilirà quale azione registrare all'interno dello switch e gliela comunicherà tramite un pacchetto di tipo OF_FLOW_ADD. In questo modo lo switch potrà registrare nelle sue flow table il tipo d'azione da eseguire in futuro ed eventualmente, tramite sollecitazione del controller, potrà mandare in uscita il pacchetto tramite un messaggio di tipo OF_PACKET_OUT. Da quel momento in poi il percorso tra l'host mittente e l'host destinatario sarà registrato e non ci sarà più bisogno di coinvolgere il controller.

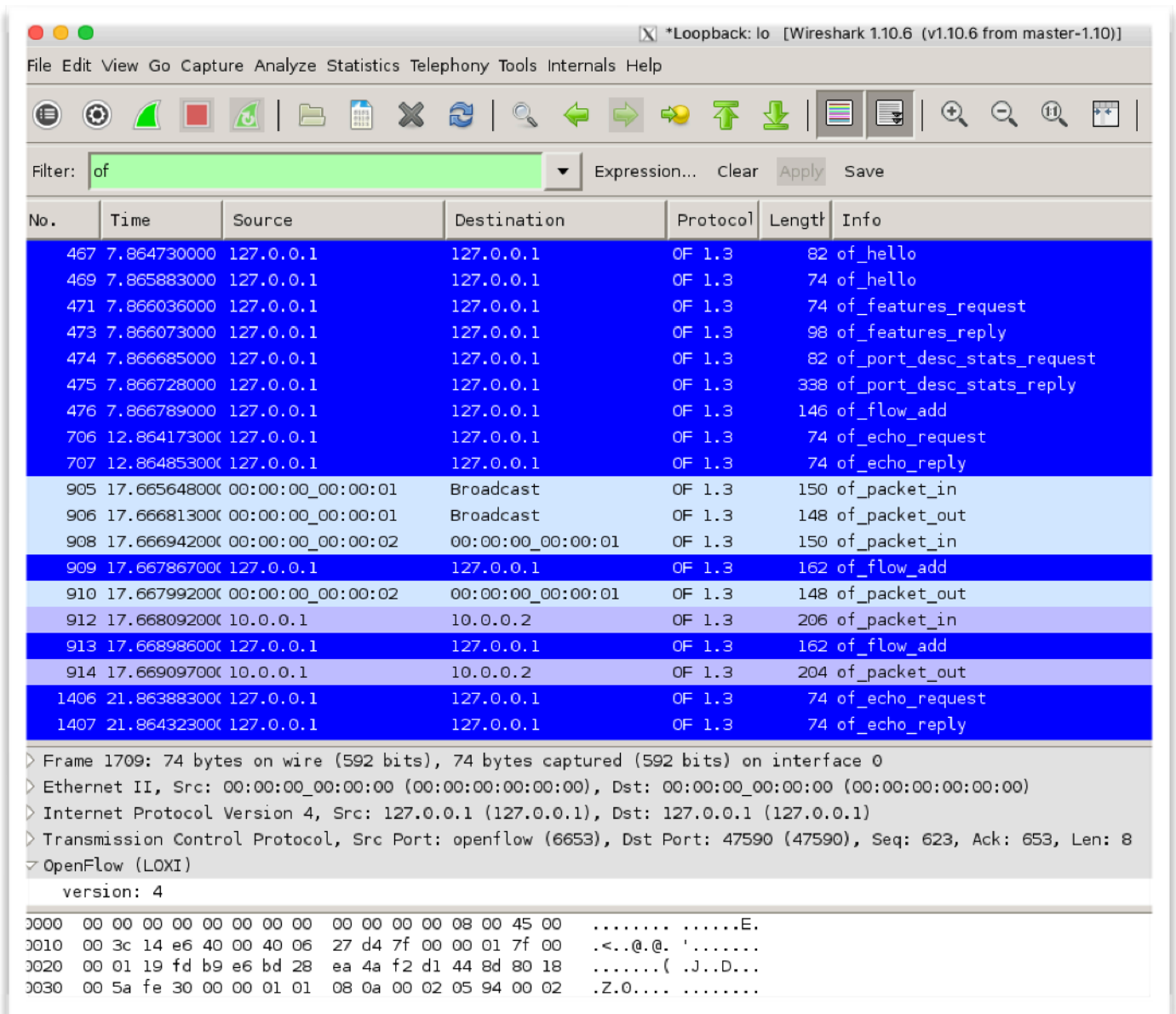
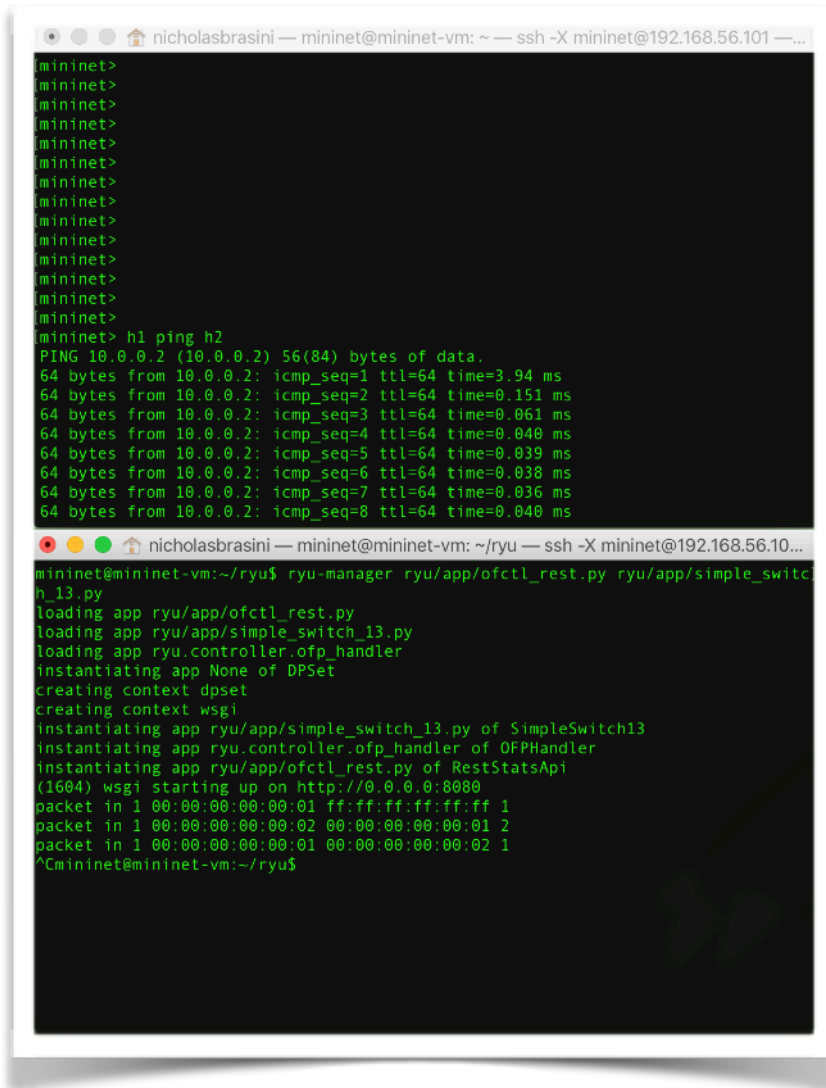


Illustrazione 7: scambio di packet_in (n° 912), packet_out (n° 914) e flow_add (n° 913)



```
nicholasbrasini — mininet@mininet-vm: ~ — ssh -X mininet@192.168.56.101 — ...
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet>
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=3.94 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.151 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.061 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.040 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.039 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.038 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.040 ms

nicholasbrasini — mininet@mininet-vm: ~/ryu — ssh -X mininet@192.168.56.10...
mininet@mininet-vm:~/ryu$ ryu-manager ryu/app/ofctl_rest.py ryu/app/simple_switc
h_13.py
loading app ryu/app/ofctl_rest.py
loading app ryu/app/simple_switch_13.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/ofctl_rest.py of RestStatsApi
(1604) wsgi starting up on http://0.0.0.0:8080
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
^Cmininet@mininet-vm:~/ryu$
```

Illustrazione 8: scambio di messaggi di tipo Ping tra due host virtuali

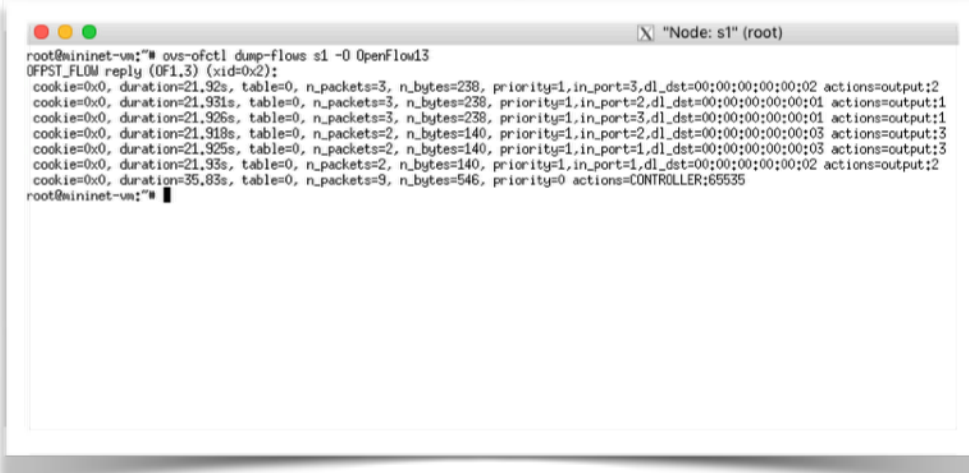
Nell'illustrazione 8 è mostrato uno scambio di pacchetti tra due host virtuali creati all'interno della rete Mininet (che descriverò nei paragrafi successivi) con a capo un controller Ryu (anche di questo ne parlerò in seguito). Dopo aver lanciato il comando `h1 ping h2`, con il quale si vuole verificare la disponibilità di connessione tra i due host virtuali h1 e h2, è evidente che il primo pacchetto giunga a destinazione impiegando molto più tempo rispetto agli altri (3,94 ms contro 0,45 ms di media dei successivi). Questo succede perché, essendo il primo pacchetto che viene scambiato tra i due host e che transita all'interno dello switch, quest'ultimo lo recapiterà in un primo momento al controller. Una volta che il controller avrà stabilito la regola di transito per i due host, non ci sarà più bisogno di passare da lui ma lo switch riuscirà ad instradarli direttamente anche nel caso in cui si dovesse lanciare il comando speculare, perché la rotta tra h2 e h1 sarebbe accettata nello stesso modo.

Capitolo 4 - Gli strumenti utilizzati

Dopo aver approfondito e studiato una rete SDN, comprensiva del protocollo di comunicazione OpenFlow, bisogna ora analizzare gli strumenti utilizzati nel corso dei test effettuati su Virtual Machine. Nel corso dei paragrafi successivi parlerò di Open vSwitch, Mininet e Ryu, che sono i principali strumenti di cui mi sono servito nel corso di questi mesi. Grazie ad ognuno di questi componenti mi è stato possibile realizzare una rete SDN e di carpirne i concetti principali.

4.1 Introduzione ad Open vSwitch

Il primo strumento che andrò a trattare è Open vSwitch [8], il componente di base che mi permetterà di mettere in piedi una rete SDN. Si tratta di uno switch multilayer virtuale open-source piuttosto utilizzato nel mondo dell'informatica e delle reti virtuali. Nei test che ho effettuato ho utilizzato la versione di Open vSwitch preinstallata nella macchina virtuale che conteneva Mininet [9]. Open vSwitch è uno switch che supporta interfacce di gestione standard e permette il controllo e l'estensione delle funzioni di forwarding in modo programmatico. Lo switch è scritto quasi interamente in linguaggio C e offre diverse funzionalità, tra cui il supporto alla VLAN ma soprattutto il supporto ad OpenFlow. Questo consente una gestione dei pacchetti efficiente grazie all'utilizzo di un modulo Linux preesistente. Tramite una serie di comandi Linux sarà possibile, una volta ottenuti i permessi di amministratore, visualizzare diverse informazioni come: visualizzazione di tutti gli switch attivi sulla rete tramite `ovs-vsctl show` ; visualizzazione delle porte presenti su uno specifico switch tramite `ovs-ofctl dump-ports-desc [nome switch]` ; visualizzazione delle flow entries impostate nello switch tramite `ovs-ofctl dump-flow [nome switch]`.



```
root@mininet-vm:~# ovs-ofctl dump-flows s1 -O OpenFlow13
OFFST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=21.92s, table=0, n_packets=3, n_bytes=238, priority=1,in_port=3,dl_dst=00:00:00:00:00:02 actions=output:2
cookie=0x0, duration=21.93s, table=0, n_packets=3, n_bytes=238, priority=1,in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
cookie=0x0, duration=21.926s, table=0, n_packets=3, n_bytes=238, priority=1,in_port=3,dl_dst=00:00:00:00:00:01 actions=output:1
cookie=0x0, duration=21.918s, table=0, n_packets=2, n_bytes=140, priority=1,in_port=2,dl_dst=00:00:00:00:00:03 actions=output:3
cookie=0x0, duration=21.925s, table=0, n_packets=2, n_bytes=140, priority=1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:3
cookie=0x0, duration=21.93s, table=0, n_packets=2, n_bytes=140, priority=1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:2
cookie=0x0, duration=35.83s, table=0, n_packets=9, n_bytes=546, priority=0 actions=CONTROLLER:65535
root@mininet-vm:~#
```

Illustrazione 9: visualizzazione delle flow entries in un Open vSwitch

4.2 Struttura di Open vSwitch

Un Open vSwitch generico è fondamentalmente suddiviso in due parti: lo *switch agent* e la parte relativa al *data plane*. Lo switch agent è quel componente che si occupa di dialogare con il/i controller e con il data plane attraverso il protocollo OpenFlow. Lo switch agent ha il compito di tradurre i comandi, che giungono dal controller, nelle istruzioni di basso livello necessarie e le inoltrerà poi al data plane, traducendo le notifiche in messaggi OpenFlow che verranno inviati al controller. Lo switch agent si suddivide a sua volta in quattro principali componenti: *Data Plane Offload*, che è il protocollo del data plane; *Core Logic*, che è il gestore dello switch ed esegue comandi sul data plane sfruttando le funzioni del componente precedente; l'*OpenFlow Protocol*, che è il protocollo utilizzato per comunicare con il controller ed infine il *Data Plane Protocol*, protocollo utilizzato per configurare lo stato del data plane.

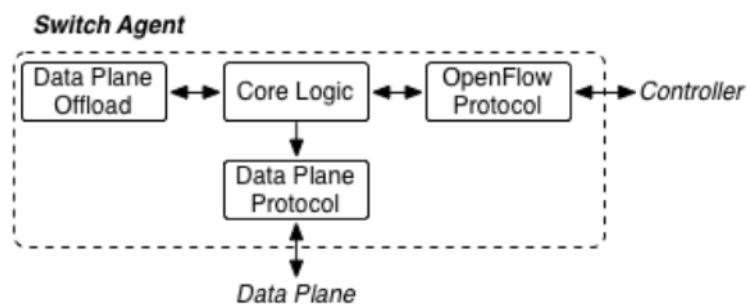


Illustrazione 10: schema dello switch agent

Il *data plane* invece è composto di flow, flow table, porte, classificatori ed azioni (output). I pacchetti che giungono in entrata o che escono in uscita possono farlo attraverso le porte dell'Open vSwitch; grazie ai classificatori è invece possibile capire se i pacchetti che transitano all'interno dello switch hanno già regole da seguire oppure se devono essere inviati al controller prima di essere inviati fuori dallo switch.

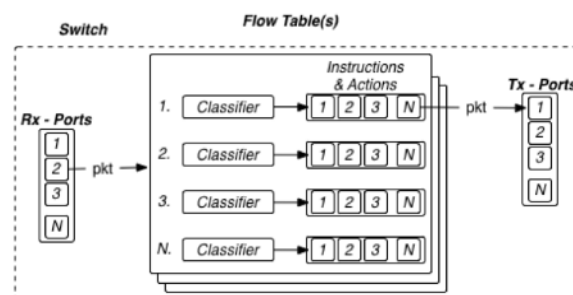


Illustrazione 11: schema del data plane

4.3 Life cycle di un pacchetto all'interno di Open vSwitch

Prima di tutto bisogna sottolineare che appena un pacchetto giunge all'interno di un Open vSwitch viene generata una sorta di chiave identificativa per il singolo pacchetto: per costruire questa chiave verranno utilizzate diverse informazioni, come ad esempio quando il pacchetto è arrivato, la versione del protocollo utilizzato, la porta e il tempo di arrivo. Una volta che il pacchetto è arrivato dunque, dopo aver configurato la sua chiave, vengono eseguite le seguenti azioni, nel cosiddetto *life cycle* del pacchetto all'interno dell'Open vSwitch.

- **Packet Arrival.** E' la prima fase e il pacchetto arriva all'interno dello switch attraverso una delle porte che ha a disposizione. Vengono memorizzate informazioni relative al pacchetto in ingresso.
- **Key Extraction.** La fine della prima fase coincide con l'inizio della seconda, ovvero l'analisi del pacchetto per poter creare una chiave identificativa.
- **Table Selection.** Ovviamente all'interno di uno switch potrebbero esserci più flow table contenenti le regole segnalate dal controller. In questa fase si analizzano, una ad una, le table flow alla ricerca della corrispondenza con la chiave creata in precedenza.
- **Flow Selection.** Una volta entrati all'interno della tabella, si analizzano i diversi flow alla ricerca di un'eventuale coincidenza. Se si trova la coincidenza, questo diventa il flow selezionato che contiene l'azione da dover eseguire per quello specifico pacchetto.
- **Action Application.** E' l'ultima fase del ciclo di vita di un pacchetto all'interno dello switch. Una volta che è stato selezionato il flow relativo alla chiave, si analizzano le azioni da dover mettere in atto per quello e per i pacchetti dello stesso flow. A quel punto potrebbe essere necessario inoltrare la chiave del pacchetto ad un'altra tabella dello switch per permettere l'esecuzione di altre azioni.

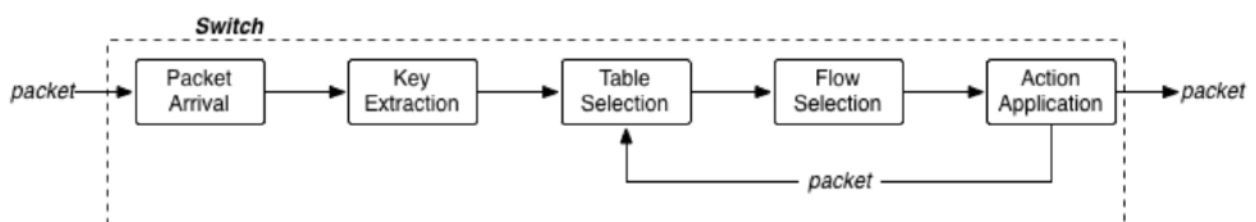


Illustrazione 12: life cycle di un pacchetto

4.4 Introduzione a Mininet

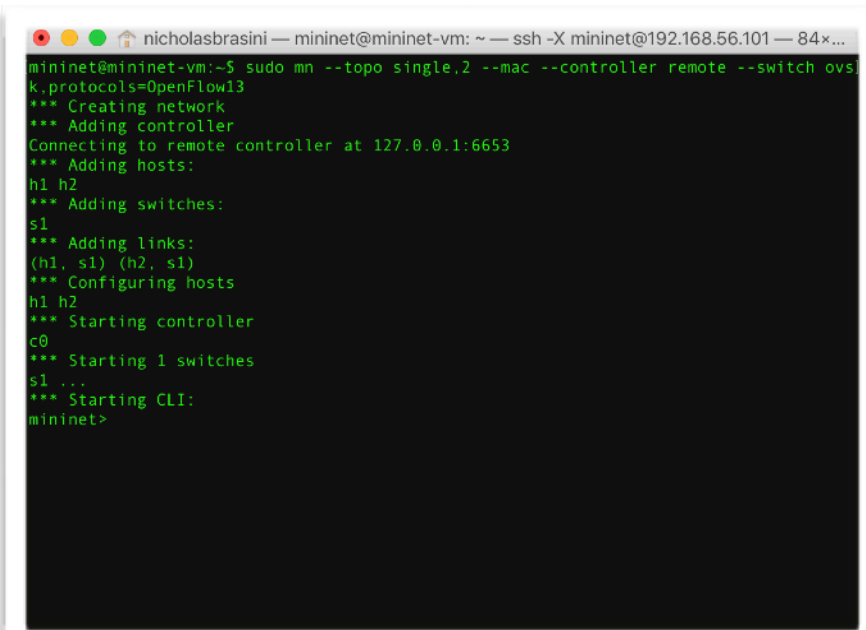
Prima di iniziare a lavorare su questa tesi non avevo mai sentito parlare di questo fantastico programma. Dopo averci lavorato diverso tempo, posso infatti affermare che risulta essere fondamentale per tutte quelle persone che vogliono provare a cimentarsi nel mondo delle reti non avendo magari a disposizione troppi strumenti utili per metterne in piedi una. Mininet infatti è uno strumento molto utile per la simulazione delle reti [10], in grado di realizzare topologie anche piuttosto complesse per sperimentare e verificare il corretto funzionamento di alcuni protocolli, primo tra tutti quello OpenFlow.

4.5 Caratteristiche di Mininet

Come detto in precedenza, Mininet permette di creare topologie di rete a seconda delle proprie necessità. Grazie a Mininet infatti è possibile configurare tutta una serie di host virtuali, che saranno creati come semplici namespace a livello kernel e anche di switch attraverso le funzionalità offerte da Open vSwitch. Oltre a questi due fondamentali elementi è anche possibile utilizzare il controller di default offerto dal sistema oppure utilizzarne uno proprio. Per i miei test, come spiegherò in seguito, ho utilizzato il controller Ryu preferendolo a quello proposto da Mininet per tutta una serie di motivi. Mininet permette dunque di poter realizzare una topologia di rete a seconda delle proprie esigenze: questo è possibile sia tramite interfaccia di Mininet con specifici comandi (attraverso la CLI, *Command Line Interface*) oppure tramite la creazione di script di Python caricati nel momento in cui Mininet viene avviato. Mininet è stato realizzato in modo da automatizzare il processo della creazione di rete, utilizzando comandi del kernel di Linux per la simulazione delle configurazioni volute. E' possibile creare un numero arbitrario di host, switch e controller, cosicché l'utente possa testare anche le topologie di rete più disparate. Mininet connette switch e host attraverso virtual ethernet (veth) e permette dunque di poter effettuare test su reti anche complesse senza avere la necessità di realizzarle nel mondo fisico. E' uno strumento molto potente che ben si adattava ai test che ho realizzato per la mia tesi e, a conti fatti, si è rivelato un ottimo strumento per apprendere a fondo i dettagli del mondo delle SDN.

4.6 Configurazione di Mininet

Detto ciò, bisogna spiegare come sono riuscito ad installare e ad utilizzare Mininet. Dopo aver scaricato l'ultima versione (la 5.1) della VM VirtualBox [11], mi sono recato sul sito di Mininet e ho scaricato l'immagine VM di Mininet che utilizza la versione più aggiornata del programma, ovvero la 2.2.2. Una volta fatto ciò, ho avviato la macchina virtuale con l'immagine di Mininet. Dopo aver effettuato l'accesso tramite le credenziali riportate sul sito, mi si è presentata un'interfaccia grafica piuttosto spartana: la schermata infatti non poteva essere ingrandita e il puntatore del mouse non era riconosciuto. A quel punto ho deciso di collegarmi tramite SSH dalla mia macchina, ma per farlo ho dovuto aggiungere una *'Rete solo host'* configurata con un indirizzo IP a piacere. In questo modo ho potuto utilizzare il mio terminale per muovermi all'interno della VM. La configurazione della prima topologia di rete era piuttosto semplice: tramite il comando `sudo mn` è possibile realizzare una rete che abbia due host (h1 e h2), uno switch (s1) e un controller (c0). Dopo aver imparato come gestire alcuni comandi di cui non ero a conoscenza, per tutti i test successivi ho utilizzato il comando `sudo mn -- topo single,2 --mac -- controller remote --switch ovsk,protocols=OpenFlow13`, che mi permette di realizzare una rete con due host (h1 e h2), uno switch Open vSwitch che accetti pacchetti OpenFlow 1.3 e che utilizzi un controller remoto e non quello offerto di default. Mininet poi mette a disposizione una serie interessante di comandi come ad esempio *iperf*, che testa la connessione tra i due host, *net*, che permette di visualizzare lo stato della rete, e *link*, che visualizza in output tutti i collegamenti della rete appena creata.



```
nicholasbrasini — mininet@mininet-vm: ~ — ssh -X mininet@192.168.56.101 — 84x...
mininet@mininet-vm:~$ sudo mn --topo single,2 --mac --controller remote --switch ovsk,protocols=OpenFlow13
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Illustrazione 13:
creazione della topologia
di rete con due host, uno
switch e un controller
remoto

4.7 Introduzione a Ryu

Giunti a questo punto, non si può non parlare del controller Ryu (che in giapponese significa flusso), il vero motore della nostra rete SDN. Ryu infatti mi è sembrato fin da subito uno dei controller più intuitivi e ho deciso di eseguire i test proprio con questo controller rispetto a quello offerto di default da Mininet (utilizzando l'ultima versione, la 4.15). Come detto in precedenza, per potersi avvalere dei servizi di Ryu, basterà specificare all'atto della creazione della topologia di rete di voler utilizzare un controller remoto, che potrà essere istanziato anche successivamente. Sia che venga eseguito prima della creazione della rete, sia che venga fatto partire dopo, Ryu verrà riconosciuto come il controller da utilizzare e verrà "agganciato" alla nostra rete SDN.

4.8 Storia dei controller e di Ryu

Ryu però non è l'unico controller disponibile sul mercato [12]. Nel corso degli ultimi anni infatti sono stati molteplici i controller messi a disposizione degli utenti. Ma prima di tutto bisogna chiarire meglio il concetto di controller. Un controller infatti può essere visto come il cervello della rete. È l'applicazione che funge da punto di controllo strategico nella rete SDN: gestisce il controllo di flusso verso i dispositivi di rete tramite le Southbound API e le applicazioni utilizzando le Northbound API. Il primissimo controller SDN fu progettato da Nicira Networks e venne presentato con il nome NOX. Nel 2008 NOX è stato donato alla comunità SDN in maniera tale da poter diventare uno strumento di studio per eventuali evoluzioni. Successivamente vennero realizzati numerosi controller di tipo opensource, cosicché tutta la comunità potesse lavorarci. I più noti sono Beacon, basato su Java, Trema e lo stesso Ryu. È evidente che con la nascita degli switch SDN, numerosi produttori stanno facendo a gara per poter inserire sul mercato i propri controller SDN. Fornitori come Cisco, HP e IBM da qualche anno hanno iniziato a lavorare su questi prodotti basandosi prevalentemente su Beacon e su OpenDaylight.



Illustrazione 14: logo del controller Ryu

4.9 Struttura di Ryu

Ryu è formato da numerose entità ma non per questo risulta di difficile comprensione. Di seguito verranno elencati i componenti principali con annesse le relative funzionalità.

- **Northbound API.** Sono le API che Ryu mette a disposizione dell'utente per poter collegare il controller al mondo esterno [13]. I plug-in che Ryu ci mette a disposizione sono Openstack Neutron (permette di creare configurazioni VLAN) e OF-REST, un'interfaccia REST che, in combinazione con WSGI (*Web Server Gateway Interface*, un framework per associare applicazioni web a server web in Python [14]) permette di introdurre nuove REST API in un'applicazione. Il campo del Northbound API è sicuramente uno dei più esplorati dagli utilizzatori di Ryu in quanto permette di interagire con le applicazioni esterne. Può essere considerato un mondo 'ad alto livello' rispetto alle Southbound API, che lavorano a basso livello. Uno dei vantaggi principali è che adesso non bisogna più programmare secondo la logica dei diversi dispositivi di rete ma basterà conoscere i linguaggi base (come Java, Python, ecc) per poter gestire facilmente la propria rete.
- **Ryu Applications.** Rappresenta l'entità che riguarda le applicazioni built-in o quelle scritte personalmente da noi. Sono entità single-thread che implementano diverse funzionalità. Ogni applicazione Ryu gestisce una coda di ricezione per gli eventi in entrata. La coda è di tipo FIFO (First In First Out), per cui il primo evento che verrà catturato sarà anche il primo ad essere servito. Il thread principale in maniera ciclica (attraverso un loop continuo) estrae dalla coda il primo evento che deve essere servito e poi crea un nuovo thread che esegua l'event handler relativo al tipo di evento estratto. Qualora non ci fossero eventi da servire si metterà in attesa. Una volta invece che ha preso in carico un evento, e dopo aver generato il thread relativo a quel tipo di evento, sarà costretto ad attendere la sua fine prima di poter servire altri elementi della coda.
- **App-Manager.** Rappresenta il componente fondamentale che si deve trovare all'interno di tutte le applicazioni Ryu. Esso contiene una classe fondamentale, detta RyuApp, che tutte le applicazioni Ryu dovranno ereditare per funzionare correttamente.
- **Ryu-Manager.** E' l'eseguibile principale: crea un processo che si pone in ascolto sull'IP e sulla porta specificati (la porta di default è la 6633) e permette la connessione da parte dei dispositivi di rete.

- **Core Processes.** E' il componente che al suo interno racchiude molte funzionalità, dalla gestione degli eventi e dei messaggi fino alla gestione dello stato del controller.
- **OpenFlow controller.** Componente che di base si occupa di gestire la comunicazione con gli switch OpenFlow.
- **OpenFlow Protocol.** E' il protocollo relativo allo scambio di messaggi con i dispositivi di rete. Attualmente Ryu supporta tutte le versioni di OpenFlow, ovvero fino alla 1.5. Al suo interno contiene due librerie, una di codifica e una di decodifica del protocollo.
- **Libraries.** Nell'illustrazione 15 è possibile notare come Ryu supporti una grande vastità di librerie, come ad esempio OVSDB, OFCONFIG, XFLOW e altre librerie di terze parti.

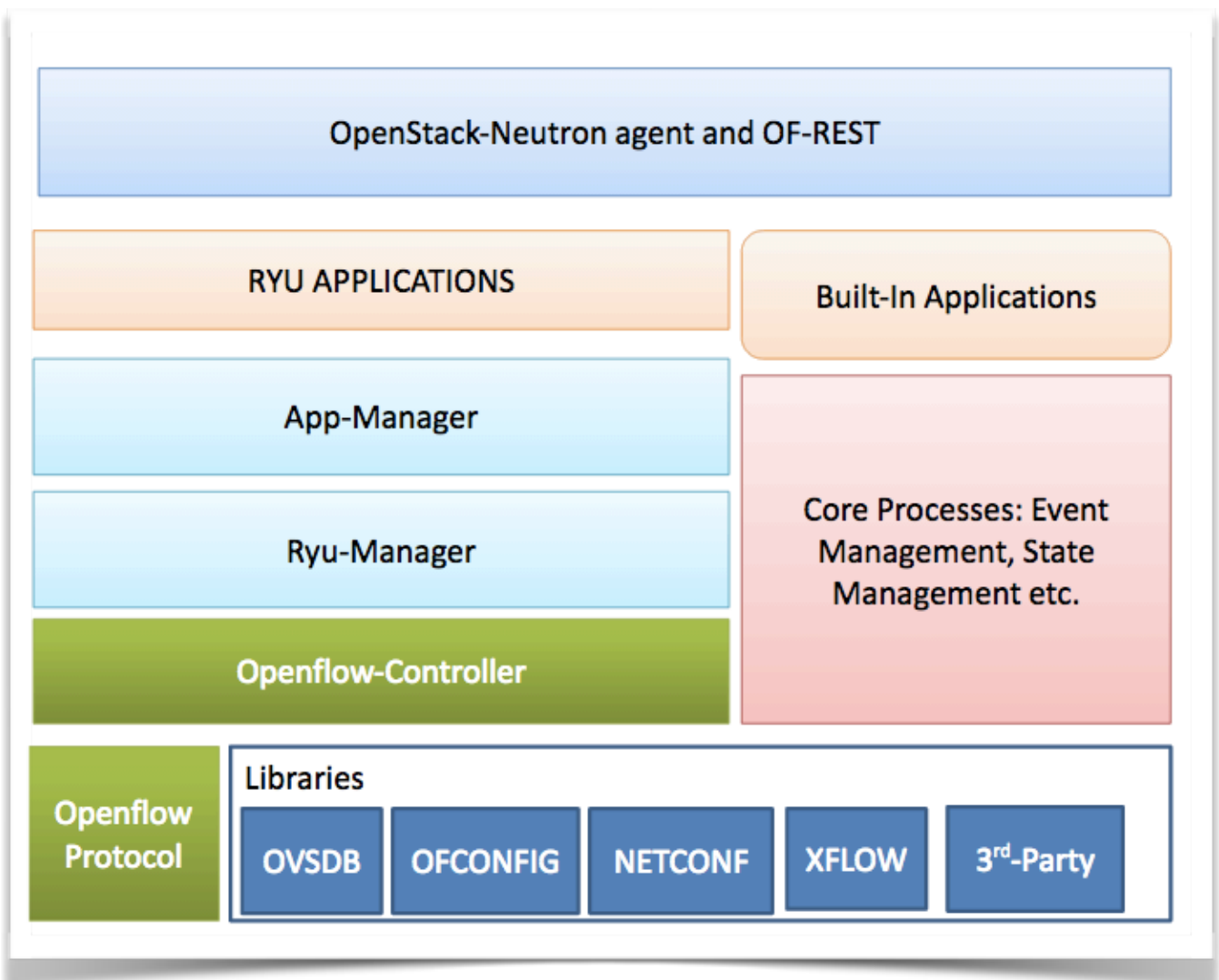
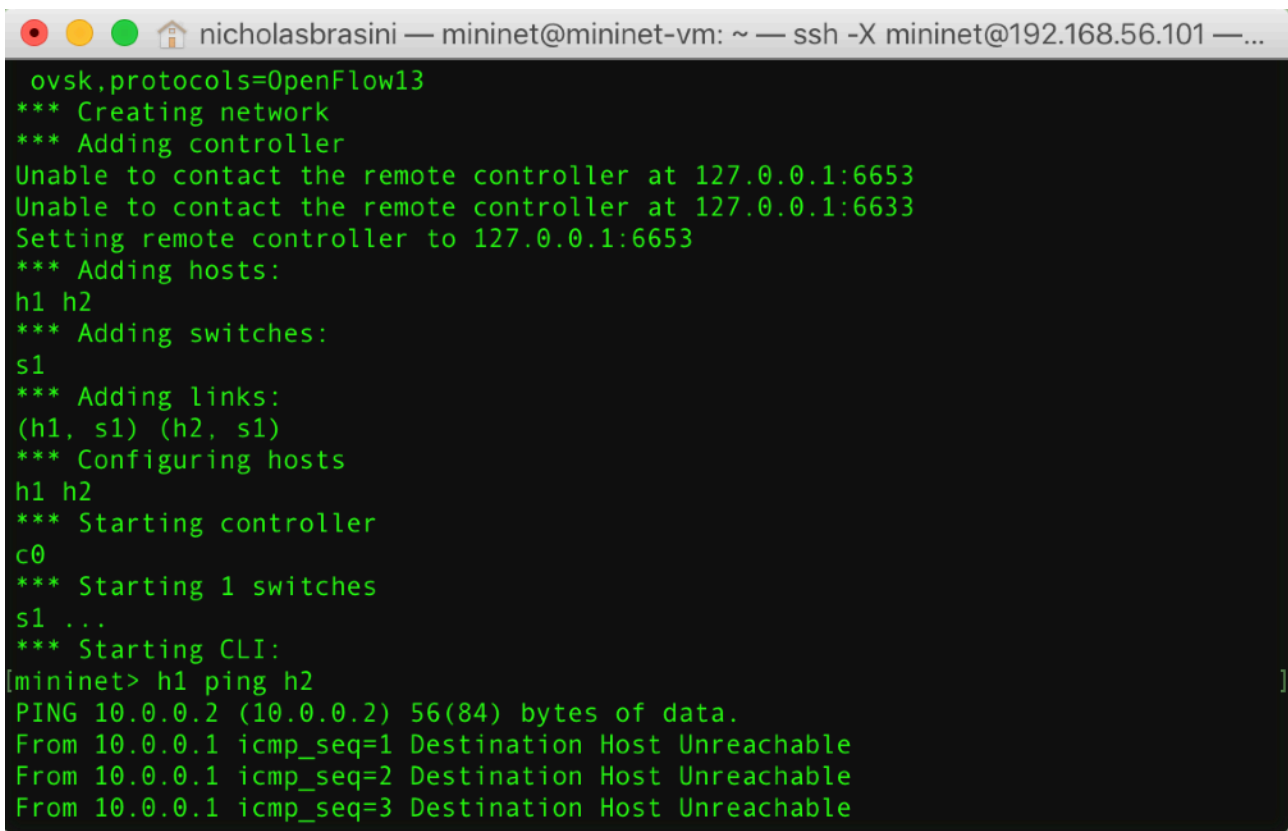


Illustrazione 15: la struttura interna del controller Ryu

4.10 Configurazione di Ryu all'interno di Mininet

Dopo aver studiato nei dettagli la struttura e il funzionamento di Ryu, sono passato ai test pratici. In precedenza ero riuscito a configurare una rete SDN tramite il programma di simulazione delle reti Mininet, ma mi ero fermato alla mera configurazione senza aver gestito la questione del controller. L'illustrazione 13 rappresenta infatti la creazione della rete con due host, uno switch e un controller remoto che però non è ancora stato configurato. Se a quel punto avessi deciso di verificare la connettività tra i due host lanciando il comando `h1 ping h2`, il risultato che avrei ottenuto sarebbe stato il seguente:



```
nicholasbrasini — mininet@mininet-vm: ~ — ssh -X mininet@192.168.56.101 —...
ovsk,protocols=OpenFlow13
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
[mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

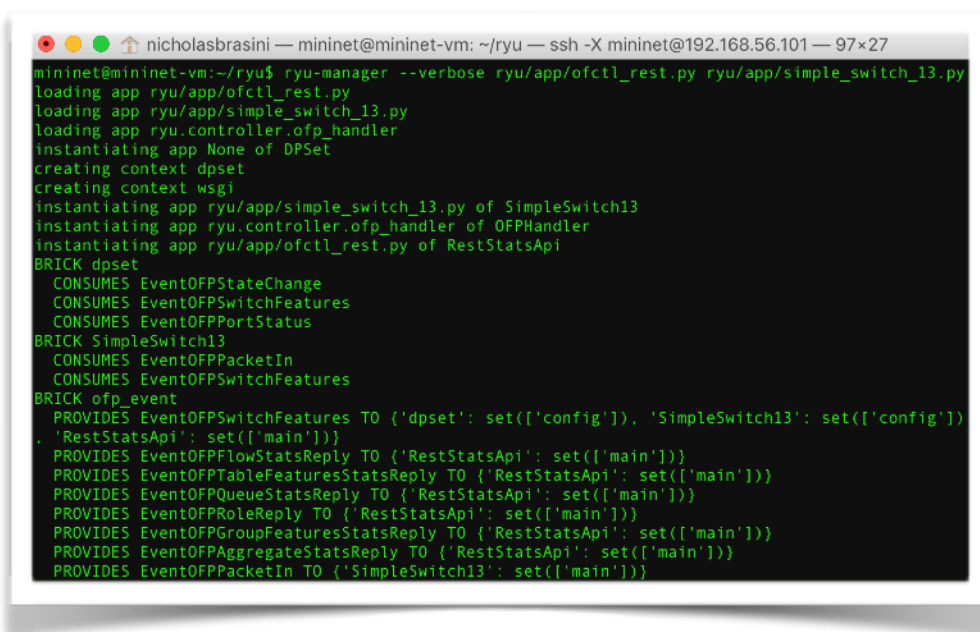
Illustrazione 16: risultato del Ping tra gli host `h1` e `h2`

E' facilmente intuibile il motivo del risultato *Destination Host Unreachable*. Senza aver impostato alcun controller infatti, i pacchetti giungevano allo switch OpenFlow che però non sapeva assolutamente cosa poterne fare, dal momento che il controller era assente. Nella logica delle reti SDN infatti i pacchetti che giungono ai dispositivi di rete devono essere girati al controller qualora nelle loro flow table non siano presenti regole relative ai pacchetti in entrata. Ma a quel punto lo switch si trovava in difficoltà non potendo recapitare il pacchetto

al controller, per cui il flusso era destinato a fermarsi e a non giungere a destinazione.

A quel punto ho iniziato a studiare le funzionalità di base che il controller Ryu mette a disposizione degli utenti e mi sono imbattuto principalmente in due file: *ofctl_REST.py* e *simple_switch_13.py*. Grazie a questi due file ho potuto procedere nei test relativi alla rete SDN creata in precedenza. Il primo file è quello che mi permette di utilizzare le REST API come ad esempio *GET /stats/switches* che mi consente di visualizzare in output il numero di switch utilizzati dalla rete, oppure *GET /stats/flow/<dpid>* che mi permette di visualizzare le regole dei flussi contenute all'interno di un certo switch. Quindi questo file mi permette di comunicare e di venire a conoscenza di tutta una serie di informazioni relative alla rete SDN tramite REST API. E' chiaro però che, stando così le cose, le REST API possano essere invocate dopo aver fatto partire il server WSGI di cui ho discusso in precedenza. Questa classe si occupa inoltre di gestire la creazione di un context wsgi per poter gestire la questione delle chiamate per conoscere lo stato della rete.

Il secondo file invece permette di poter stabilire una connessione tra i diversi host e dunque, ad esempio, di poter verificare con il comando *h1 ping h2* che i due host siano raggiungibili e riescano a scambiarsi messaggi. Studiando nel dettaglio le caratteristiche di questo file, ho notato come funzionino le azioni e gli handler che noi osserviamo ad alto livello ogni volta che dei pacchetti giungono ai dispositivi di rete. Nello specifico ho notato come venga gestita la ricezione dei packetIn da parte del controller ogni volta che uno switch ne inoltra uno dopo aver ricevuto un certo flusso di pacchetti in entrata.



```
nicholasbrasini — mininet@mininet-vm: ~/ryu — ssh -X mininet@192.168.56.101 — 97x27
mininet@mininet-vm:~/ryu$ ryu-manager --verbose ryu/app/ofctl_rest.py ryu/app/simple_switch_13.py
loading app ryu/app/ofctl_rest.py
loading app ryu/app/simple_switch_13.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/ofctl_rest.py of RestStatsApi
BRICK dpset
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPPortStatus
BRICK SimpleSwitch13
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPSwitchFeatures TO {'dpset': set(['config']), 'SimpleSwitch13': set(['config']), 'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPacketIn TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPortStatus TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPortQueueStatsReply TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPortRoleReply TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPortGroupFeaturesStatsReply TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPortAggregateStatsReply TO {'RestStatsApi': set(['main'])}
  PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
```

Illustrazione 17: configurazione e lancio del controller Ryu

Capitolo 5 - Primi test con tutti gli strumenti

Terminata la spiegazione di tutti i componenti messi in campo, è ora di cominciare a spiegare i primi test effettuati con gli strumenti descritti nei paragrafi precedenti. Per poter realizzare la rete SDN, ho dunque utilizzato Mininet e Ryu in aggiunta al programma Postman [15], un'applicazione che permette di costruire, testare e documentare API in maniera molto agile. Tramite Postman è infatti possibile effettuare delle chiamate API senza dover mettere mano al codice dell'applicazione, consentendo di effettuare chiamate tramite un'interfaccia grafica molto intuitiva.

5.1 Progettazione della rete SDN

Il primo passo da compiere riguarda la progettazione della rete SDN. Dal momento che questi test serviranno semplicemente per verificare quanto appreso finora, non utilizzerò tutti gli strumenti formali che invece mi verranno utili quando parlerò, più avanti, del progetto vero e proprio di questa Tesi. Questi test possono essere visti come “prologo” di quello che sarà il lavoro finale. Prima di tutto, ho deciso di creare una rete SDN proprio come discusso nel capitolo relativo a Mininet. Tramite l'apposito comando ho dunque realizzato una rete composta da due host (h1 e h2), uno switch OpenFlow (s1) e un controller Ryu remoto (c0). Ho sfruttato la connessione SSH per potermi connettere alla VM e ho utilizzato due terminali: nel primo ho creato la topologia della rete descritta in precedenza, mentre nel secondo ho fatto partire il controller Ryu grazie ai file *ofctl_REST.py* e *simple_switch_13.py*.

5.2 Verifica della connettività e analisi Wireshark

Una volta realizzata la rete, ho effettuato la prova della connessione lanciando il comando *h1 ping h2*. Avendo connesso la rete al controller Ryu, i pacchetti sono tranquillamente giunti a destinazione. Dopodiché ho effettuato qualche prova con Wireshark per visualizzare il traffico OpenFlow analizzando i diversi tipi di pacchetti inviati da s1 a c0 e viceversa. Ad eccezione di qualche problema iniziale, la configurazione è andata a buon fine e l'analisi del traffico tramite il suddetto programma mi ha confermato che la rete era pronta per essere analizzata tramite le REST API di Ryu.

5.3 Chiamate REST tramite Postman

Dopo una fase iniziale in cui ho testato le chiamate REST da linea di comando tramite il comando *curl*, ho deciso di utilizzare l'applicazione Postman descritta ad inizio capitolo. In questo modo ho potuto agire in maniera più veloce grazie anche alla comoda interfaccia che Postman mette a disposizione dell'utente. Le prime prove sono state relative al numero di switch all'interno della rete, per verificare che il risultato fosse 1. Successivamente ho provato ad interagire con la mia rete attraverso chiamate più complicate, come ad esempio quella che appare nell'Illustrazione 18. In quel caso ho chiamato la *GET /stats/flow/1*, che mi permette di avere in output i flussi relativi allo switch 1 dopo aver eseguito *h1 ping h2*.

```
"1": [
  {
    "actions": [
      "OUTPUT:1"
    ],
    "idle_timeout": 0,
    "cookie": 0,
    "packet_count": 5,
    "hard_timeout": 0,
    "byte_count": 434,
    "duration_sec": 6,
    "duration_nsec": 488000000,
    "priority": 1,
    "length": 96,
    "flags": 0,
    "table_id": 0,
    "match": {
      "dl_dst": "00:00:00:00:00:01",
      "in_port": 2
    }
  },
  {
    "actions": [
      "OUTPUT:2"
    ],
    "idle_timeout": 0,
    "cookie": 0,
    "packet_count": 4,
    "hard_timeout": 0,
    "byte_count": 336,
    "duration_sec": 6,
    "duration_nsec": 486000000,
    "priority": 1,
    "length": 96,
    "flags": 0,
    "table_id": 0,
    "match": {
      "dl_dst": "00:00:00:00:00:02",
      "in_port": 1
    }
  },
],
```

Illustrazione 18: risultato della chiamata GET /stats/flow/1

Quello che è possibile notare dall'Illustrazione 18 è il risultato della chiamata appena effettuata. In questo modo possiamo analizzare nel dettaglio le regole installate sopra allo switch s1 dopo aver lanciato il comando *h1 ping h2*. Quello che ci aspettiamo di trovare sono proprio due azioni (relative allo switch), come effettivamente succede. Avendo lanciato il ping tra i due host si immagina che il controller c0 abbia installato nello switch due regole, ovvero quella che permetta ai pacchetti che escono da h1 di raggiungere h2 e viceversa. Il controller Ryu infatti effettua questa operazione simultaneamente, mentre ad esempio il controller di default di Mininet deve autorizzare entrambe le destinazioni. In sintesi Ryu installa due regole anche con il solo comando *h1 ping h2*, mentre il controller di default di Mininet ha bisogno anche del duale (in questo caso *h2 ping h1*). Analizzando la prima regola, è possibile carpire le seguenti informazioni fondamentali:

- **actions**. Rappresenta il tipo di regola installata sopra allo switch. In questo caso abbiamo OUTPUT:1 che specifica come la destinazione sia h1.
- **idle_timeout**. E' un valore che varia da controller a controller ma può essere personalizzato all'atto della chiamata. Indica per quanto tempo la regola debba essere mantenuta all'interno dello switch prima che esso riceva un altro pacchetto diretto alla destinazione indicata. Se ad esempio il valore fosse 60, se per 60 secondi non arrivasse alcun pacchetto da h2 diretto ad h1 allora la regola sarebbe eliminata. Lavora a stretto contatto con l'**hard_timeout**. Se il valore di quest'ultimo fosse 0, allora il funzionamento dell'**idle_timeout** rimarrebbe come descritto sopra; se invece fosse diverso da 0, le regole verrebbero cancellate dopo il tempo dell'**idle_timeout** se non si ricevono pacchetti o dopo il tempo dell'**hard_timeout** [16].
- **cookie**. Funziona grossomodo come i cookie del web. Ad ogni pacchetto potrebbe essere assegnato un particolare valore per eventuali modifiche future.
- **packet_count**. Indica il numero di pacchetti transitati nello switch da una certa sorgente ad una certa destinazione.
- **hard_timeout**. Come detto in precedenza, è il valore che indica dopo quanto la regola nello switch debba essere eliminata, indipendentemente da quello che succede. Lavora a stretto contatto con l'**idle_timeout**. Se entrambi i valori sono posti uguale a 0, allora la regola non sarà eliminata a meno di chiamate specifiche.
- **byte_count**. Indica i byte transitati da una certa sorgente ad una certa destinazione all'interno di uno switch.
- **duration_sec / duration_nanosec**. Indica, secondo l'unità di misura, quanto tempo hanno impiegato i pacchetti a transitare.

- **priority**. Rappresenta la priorità di una certa regola all'interno dello stesso switch rispetto ad altre regole che hanno come soggetti gli stessi flussi.
 - **match**. Indica la porta sorgente e l'host destinazione della specifica regola.
- Successivamente, affascinato come specificato in apertura, dall'argomento della sicurezza informatica, ho provato ad inscenare una sorta di attacco da parte di cracker. In una rete SDN c'è la possibilità che si verifichino alcuni exploit dovuti a vulnerabilità non ancora sconfitte, come ad esempio attacchi di tipo DDoS. Cosa succederebbe se uno dei due host fosse considerato compromesso e fosse meglio escluderlo dalla rete? Lo stesso si può dire se l'host indicato si è scoperto essere un host malevolo. Cosa potrebbe fare il controller? Tramite la chiamata *POST /stats/flowentry/add* è possibile inserire una propria regola all'interno dello switch. In questo modo ho inserito il codice presente all'interno dell'Illustrazione 19 su Postman e ho effettuato la chiamata.

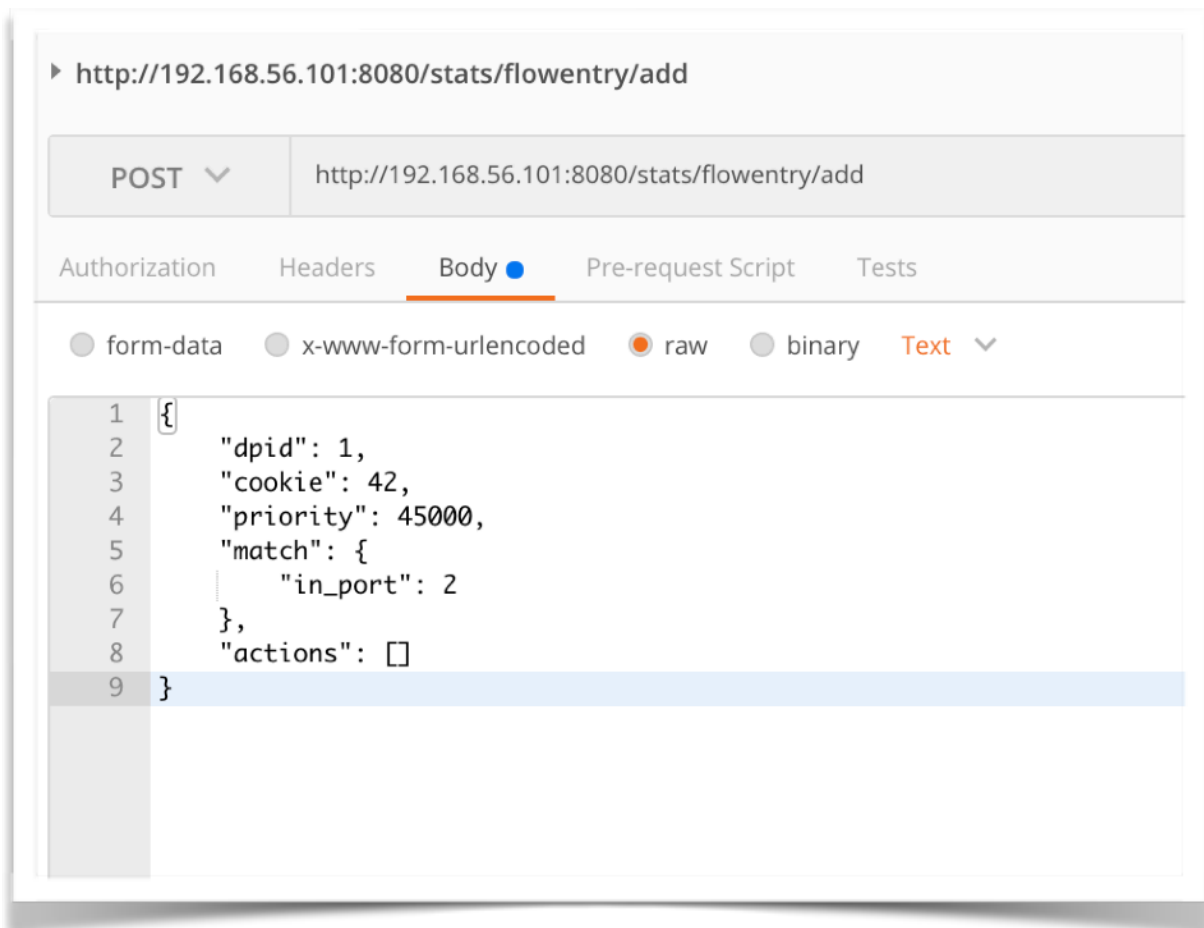


Illustrazione 19: chiamata di tipo POST per estromettere l'host 2

Con questa chiamata ho impostato il campo *dpid*, che mi identifica lo switch, a 1. Inoltre ho impostato una priorità più elevata rispetto a quella presente all'interno della regola che consente il passaggio di pacchetti da h1 ad h2. Il campo *actions* è volutamente vuoto per far capire che l'host 2 non deve riuscire a far nulla. A questo punto, per verificare che tutto sia andato correttamente, basta lanciare il comando *h1 ping h2* e vedere se le informazioni giungono in porto oppure no. Inoltre si potrebbe richiamare la *POST /stats/flowentry/add* e verificare come, oltre alle due regole dell'Illustrazione 18, se ne è aggiunta una terza, che è quella inserita in precedenza per neutralizzare h2.

```

nicholasbrasini — mininet@mininet-vm: ~ — ssh -X mininet@192.168.56.101 —...
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.037 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.037/1.495/5.640/2.394 ms
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=9 Destination Host Unreachable
From 10.0.0.1 icmp_seq=10 Destination Host Unreachable
From 10.0.0.1 icmp_seq=11 Destination Host Unreachable
From 10.0.0.1 icmp_seq=12 Destination Host Unreachable
From 10.0.0.1 icmp_seq=13 Destination Host Unreachable
From 10.0.0.1 icmp_seq=14 Destination Host Unreachable
From 10.0.0.1 icmp_seq=15 Destination Host Unreachable
From 10.0.0.1 icmp_seq=16 Destination Host Unreachable
From 10.0.0.1 icmp_seq=17 Destination Host Unreachable
From 10.0.0.1 icmp_seq=18 Destination Host Unreachable
From 10.0.0.1 icmp_seq=19 Destination Host Unreachable
From 10.0.0.1 icmp_seq=20 Destination Host Unreachable
From 10.0.0.1 icmp_seq=21 Destination Host Unreachable
From 10.0.0.1 icmp_seq=22 Destination Host Unreachable
From 10.0.0.1 icmp_seq=23 Destination Host Unreachable
From 10.0.0.1 icmp_seq=24 Destination Host Unreachable

nicholasbrasini — mininet@mininet-vm: ~/ryu — ssh -X mininet@192.168.56.10...
EVENT ofp_event->dpset EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath object at 0x7fc10e2
26a90>
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
(4716) accepted ('192.168.56.1', 58503)
Sending message with xid(f85597a6) to datapath(0000000000000001): version=None,m
sg_type=None,msg_len=None,xid=0xf85597a6,OFPFlowStatsRequest(cookie=0,cookie_mas
k=0,flags=0,match=OFPMatch(oxm_fields={}),out_group=4294967295,out_port=42949672
95,table_id=255,type=1)
EVENT ofp_event->RestStatsApi EventOFPFlowStatsReply
192.168.56.1 - - [19/Jul/2017 08:55:14] "GET /stats/flow/1 HTTP/1.1" 200 932 0.0
11645
Sending message with xid(f85597a7) to datapath(0000000000000001): version=None,m
sg_type=None,msg_len=None,xid=0xf85597a7,OFPFlowMod(buffer_id=4294967295,command
=0,cookie=42,cookie_mask=0,flags=0,hard_timeout=0,idle_timeout=0,instructions=[]
,match=OFPMatch(oxm_fields={'in_port': 2}),out_group=4294967295,out_port=4294967
295,priority=45000,table_id=0)
192.168.56.1 - - [19/Jul/2017 09:29:39] "POST /stats/flowentry/add HTTP/1.1" 200
139 0.001291

```

Illustrazione 20: i pacchetti non arrivano più a destinazione

```
{
  "actions": [],
  "idle_timeout": 0,
  "cookie": 42,
  "packet_count": 29,
  "hard_timeout": 0,
  "byte_count": 1666,
  "duration_sec": 197,
  "duration_nsec": 679000000,
  "priority": 45000,
  "length": 64,
  "flags": 0,
  "table_id": 0,
  "match": {
    "in_port": 2
  }
}
```

Illustrazione 21: la nuova regola aggiunta allo switch che neutralizza h2

Come si può vedere dall'Illustrazione 21, la regola che abbiamo inviato prima tramite comando POST al server è andata a buon fine, e può essere visualizzata all'interno delle regole di s1. In questo modo quando si tenta di comunicare con h2, i messaggi non possono essergli recapitati in virtù della priorità data a questa regola, che è più elevata della priorità relativa alla regola che invece permetterebbe di poter comunicare.

Capitolo 6 - Realizzazione del progetto

Giunti a questo punto, dopo aver effettuato tutte le prove del caso, bisogna analizzare il vero obiettivo di questa tesi triennale, ovvero la realizzazione di un'interfaccia web per poter gestire la rete SDN studiata in precedenza. Lo scopo principale è quello di permettere ad un utente che si sta affacciando per la prima volta nel mondo delle reti SDN di poterne comprendere il funzionamento ad alto livello, senza bisogno di sapere come gestire a basso livello una rete di questo tipo. Per poter fare ciò sono state messe in campo diverse tecnologie, oltre alla conoscenza della rete SDN e del protocollo OpenFlow, come ad esempio l'HTML, il CSS e il JavaScript.

6.1 Strumenti utilizzati per la web app

Oltre agli strumenti indicati nei paragrafi precedenti, come ad esempio Mininet e il controller Ryu, mi sono avvalso dell'applicazione MAMP-PRO [17] per realizzare un web server locale dove poter utilizzare la mia web app. Per poter interagire con la rete SDN ho sfruttato le chiamate REST (REpresentational State Transfer) messe a disposizione dai due file .py che ho utilizzato, ovvero *ofctl_REST.py* e *simple_switch_13.py* [18]. Si tratta di un approccio molto spesso utilizzato nell'ambito dei servizi web ed è solitamente preferito al più complesso Simple Object Access Protocol (SOAP) perché utilizza meno larghezza di banda e quindi meno risorse. Attraverso le chiamate REST è possibile dunque ricevere, modificare, creare o eliminare risorse e tutto ciò viene fatto in maniera stateless, ovvero senza lasciare traccia di interazioni passate. All'interno del mio progetto ho utilizzato le richieste HTTP di tipo GET (per visualizzare statistiche relative agli switch) e POST (per inserire un flow all'interno di uno specifico switch). In sintesi dunque il mio percorso è stato il seguente: ho dapprima studiato le caratteristiche di una rete SDN poiché era un argomento a me ancora ignoto. Successivamente ho realizzato la rete SDN attraverso Mininet e l'ho collegata ad un controller Ryu remoto. A quel punto ho prima familiarizzato con le chiamate REST attraverso il programma Postman e poi inserito il tutto all'interno di una web GUI che mi permettesse di rendere più comprensibile le risposte ricevute dal controller Ryu.

6.2 File utilizzati e astrazione del controller

Per rendere più chiari i paragrafi successivi, ho pensato di elencare i file che ho utilizzato per realizzare il progetto. Ho creato 5 file .html e 2 file .css per poter gestire la parte grafica della web app e 8 file .js (uno per ogni file .html contenente le relative funzioni, uno per gestire il reset dei form, uno per il salvataggio nel sessionStorage del browser di IP e controller scelto e uno, il file *functionRyu.js*, che rappresenta il “cervello” poiché al suo interno sono raccolte tutte le chiamate alle funzioni utilizzate contenute negli altri file .js) per gestire le chiamate REST e la presentazione dell’output ottenuto. Ho cercato, per quanto possibile, di rendere il codice pulito per evitare ripetizioni e ridondanze, ma soprattutto ho realizzato il progetto in un’ottica futura. In questo caso mi sono concentrato sul controller Ryu e relative chiamate REST, ma in un futuro prossimo un altro ragazzo potrebbe decidere di approfondire il mio progetto, magari utilizzando un altro tipo di controller che non sia Ryu. A quel punto sarebbe troppo oneroso riprogettare tutto da capo se la parte JavaScript (e quindi inerente alle chiamate di Ryu) non fosse totalmente disaccoppiata dal resto del codice. Per questo ho deciso che nel file home.html, che è il primo file che viene utilizzato nell’interfaccia web, ci sia la possibilità di scegliere quale controller utilizzare. Ho inserito chiaramente Ryu, che è la nostra prima scelta, e altri due (OpenDaylight e POX) a scopo dimostrativo. Una volta che l’utente ha scelto quale controller utilizzare, la scelta verrà salvata nel sessionStorage del browser, per cui rimarrà la stessa fino a quando il browser non verrà chiuso. L’alternativa era salvarla nel localStorage ma a quel punto la scelta sarebbe rimasta la stessa anche dopo l’eventuale chiusura del browser. L’utente poi si recherà in una delle altre quattro pagine .html che contengono le varie funzionalità della web app. L’unico file .js incluso di base nelle pagine .html è il file *passaggioIP.js*, che contiene le funzioni per gestire l’IP inserito e il controller scelto e si può considerare dunque un file universale, perché lavora sul file .html indipendentemente dal controller scelto. Per il resto sarà tutto importato in base alle scelte fatte dall’utente, per poter mantenere una certa indipendenza del codice .html rispetto ai file .js.

6.3 Casi d'uso

Sintetizzato brevemente l'obiettivo del progetto, bisogna ora analizzare le funzionalità offerte dalla web app. Ho deciso di schematizzare il tutto attraverso un diagramma dei casi d'uso, in maniera tale da rendere più chiare, una volta che analizzerò il codice, tutte le funzionalità presenti.

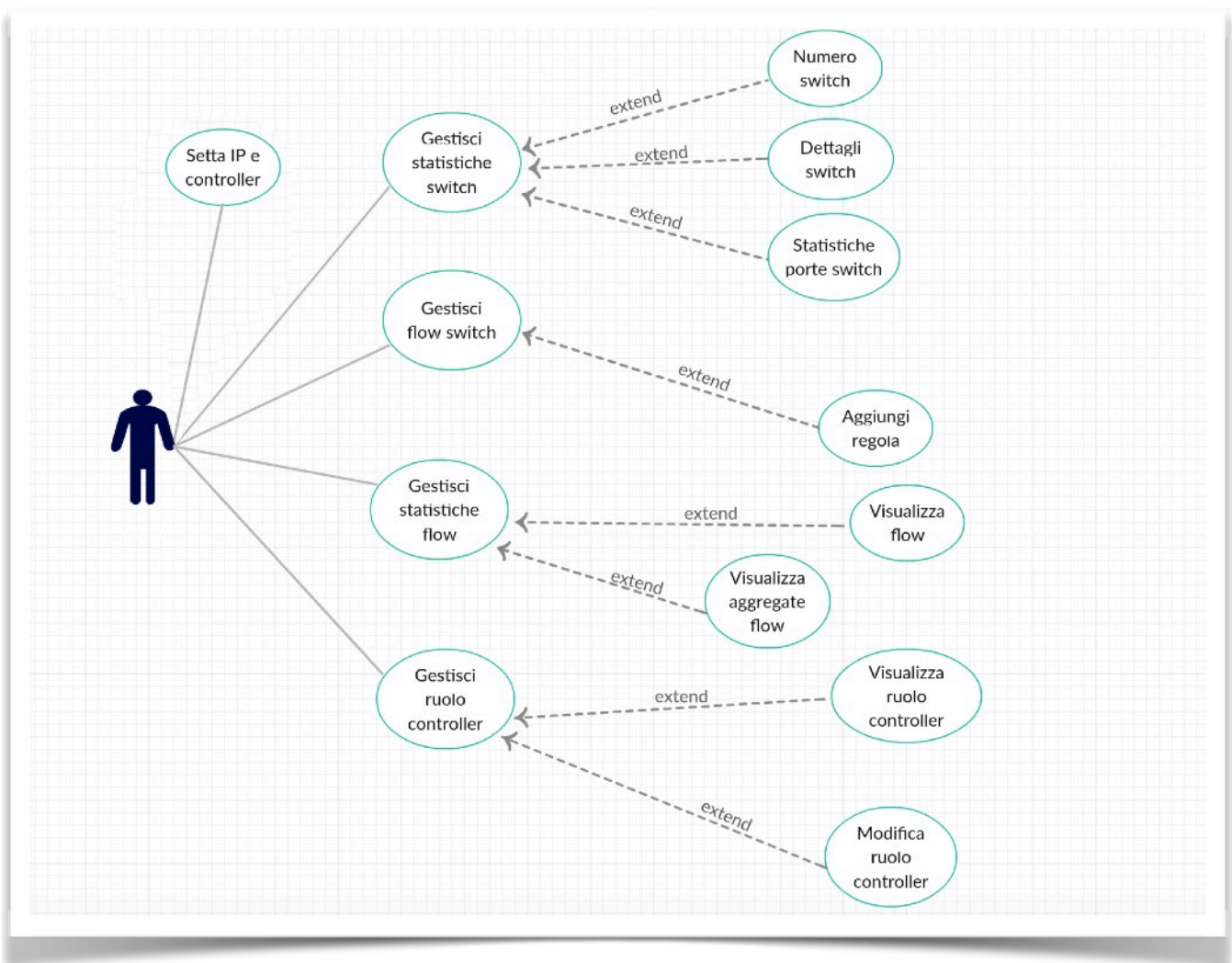


Illustrazione 22: diagramma dei casi d'uso della web app

6.4 Home web app e relativi script

Iniziamo l'analisi dal primo file che viene utilizzato all'interno della web app, ovvero *home.html*. Non analizzerò la parte di codice html, che prevede essenzialmente un menù laterale con il quale potersi muovere all'interno della web app, ma spiegherò come funziona attraverso l'interfaccia che viene presentata all'utente.



Illustrazione 23: prima parte della home



Illustrazione 24: seconda parte della home

Come è possibile notare dalla Illustrazione 23 e dalla Illustrazione 24, la home del sito è caratterizzata da un primo contenitore al cui interno è possibile inserire l'indirizzo IP del controller della rete SDN ed un combobox che permette di scegliere il tipo di controller da utilizzare. L'inserimento dell'IP non è obbligatorio in quanto può essere aggiunto manualmente anche nelle pagine successive, ma qualora si decidesse di aggiungerlo, lo si ritroverebbe anche nelle altre pagine senza bisogno di doverlo scrivere ogni volta. Dopo aver premuto il bottone Save, verrà richiamata la funzione *setIP()* presente all'interno del file esterno *passaggioIP.js*.

```
function setIP() {
  /* salvo globalmente il controller scelto, mi servira' per capire quali file richiamare */
  var controller = document.getElementById("controllerScelto");
  var ctrUser = controller.options[controller.selectedIndex].text;
  sessionStorage.setItem("controllerScelto", ctrUser);

  /* salvo globalmente l'IP inserito dall'utente per riproporlo in tutte le altre schermate */
  var el = document.getElementById("ip").value;
  sessionStorage.setItem("ipScelto", el);

  /* se tutto e' andato a buon fine, stampo l'ok */
  var ok = document.getElementById("ref").value;
  ref.innerHTML = "Dati salvati correttamente".bold();
}
```

Illustrazione 25: funzione setIP() chiamata dopo aver premuto Save

La funzione *setIP()* si preoccupa principalmente di prelevare dal combobox il controller scelto dall'utente e di salvarlo nel *sessionStorage* tramite il metodo *.setItem()*, che prende in ingresso due parametri: il primo è il nome con il quale vogliamo salvare la stringa, il secondo invece è la stringa stessa. Successivamente la funzione memorizzerà nel *sessionStorage* l'indirizzo IP inserito dall'utente. Infine, per poter permettere all'utente di capire che il salvataggio è andato a buon fine, verrà stampata sotto il bottone la stringa "Dati salvati correttamente".

La seconda parte della home invece è rappresentata nell'Illustrazione 24 e svolge lo stesso ruolo del menù laterale. Cliccando su uno dei quattro pulsantoni si verrà infatti reindirizzati alla pagina .html di riferimento per poter sfruttare le funzionalità della web app.

6.5 Settaggio degli script

Come detto in precedenza, non appena l'utente cliccherà su uno dei quattro pulsantoni, oppure selezionerà una voce dal menù laterale, verrà fatto partire il caricamento della pagina scelta. Prima che la pagina venga caricata però, tramite il metodo *onLoad()*, verranno richiamate due funzioni: *setScript()* e *getIPBase()*, contenute nel file *passaggioIP.js*.

```
function setScript() {
    var contrScelto = sessionStorage.getItem("controllerScelto");

    if (contrScelto == "Ryu") {
        $('head').append('<script src="../js/reset.js"></script>')
        .append('<script src="../js/restRyu.js"></script>')
        .append('<script src="../js/switchFunction.js"></script>')
        .append('<script src="../js/manageFlowFunction.js"></script>')
        .append('<script src="../js/controllerFunction.js"></script>')
        .append('<script src="../js/flowFunction.js"></script>')
    } else {
        /* altri controller da inserire in futuro */
    }
}
```

Illustrazione 26: funzione per importare gli script in base al controller scelto

Tramite questa funzione sarà possibile caricare nella pagina .html che si sta per aprire tutti gli script necessari per poter gestire la rete SDN. Come è evidente dall'Illustrazione 26, se il controller scelto è Ryu, come nel nostro caso, allora verranno inseriti gli script di Ryu che ho realizzato e che analizzerò nei paragrafi successivi. Per uno sviluppo futuro invece potranno essere inseriti altri script per controller diversi da Ryu senza dover cambiare assolutamente nulla all'interno del codice .html. Bisognerà avere solamente l'accortezza di realizzare un file esterno che ricalchi la struttura del file *functionRyu.js* (che analizzerò nei paragrafi successivi), al cui interno sono richiamate tutte le funzioni dei file esterni in maniera tale che la parte html non debba più essere modificata. In questo modo se inizialmente nel codice html relativo alla pagina degli switch era chiamata la funzione *numberSwitch()*, contenuta nel file *switchFunction.js*, adesso è sostituita con la *getFunctionNumberSwitch()*, che si trova nel file *functionRyu.js*, e che al suo interno richiama la *numberSwitch()*. In questo modo, qualora si volesse modificare o sostituire la funzione da chiamare

(in questo caso *numberSwitch()*), il codice html rimarrebbe lo stesso, mentre la funzione da modificare sarebbe la *getFunctionNumberSwitch()*, della quale andrebbe semplicemente modificato il corpo. Così facendo è possibile mantenere il codice html lontano da modifiche qualora si decidesse di inserire un nuovo controller.

```
function getIPBase() {  
  if (sessionStorage.getItem("ipScelto") != null) {  
    document.getElementById("ip").defaultValue = sessionStorage.getItem("ipScelto");  
  }  
}
```

Illustrazione 27: funzione chiamata per riempire il form dell'IP controller

Oltre alla funzione *setScript()*, ogni volta che viene caricata una pagina, viene anche chiamata la funzione *getIPBase()* contenuta sempre nel file *passaggioIP.js*. Questa funzione andrà a cercare nel *sessionStorage* se l'utente ha inserito un IP. Se lo ha fatto, allora si preoccuperà di riempire il form dell'IP controller della pagina in caricamento. In questo modo sarà possibile evitare all'utente di dover reinserire ogni volta l'IP del controller. Bisogna notare però che questo passaggio non è obbligatorio ma soprattutto reversibile: se l'utente decidesse di inserire un certo indirizzo IP per il controller, potrebbe poi comunque modificarlo sia dalla home che all'interno delle altre pagine. Se lo dovesse modificare dalla home, allora la scelta si ripercuoterebbe anche sulle altre pagine; se invece lo modificasse solo nel form di una pagina specifica che non sia la home, l'IP cambierebbe solamente in quella pagina e non nelle altre. Così facendo l'utente ha la possibilità di analizzare più controller in pagine diverse senza essere vincolato alla scelta fatta inizialmente. Nella funzione *getIPBase()* ho inserito il controllo per verificare che l'utente abbia inserito effettivamente un indirizzo IP, altrimenti in tutti i form verrebbe stampata la dicitura *null*.

6.6 Pagina switch.html e relativi script

Supponiamo ora che l'utente decida di cliccare sul pulsante "SWITCH STATS". Una volta cliccato, si verrà reindirizzati alla pagina *switch.html* e nel frattempo verrà richiamata la funzione *setScript()* contenuta nel file *passaggioIP.js* analizzata nei paragrafi precedenti.



Illustrazione 28: prima parte della schermata del file *switch.html*

Dopo aver parlato delle funzionalità che vengono richiamate nel momento del caricamento della pagina (si noti che le due funzioni, *setScript()* e *getIPBase()* vengono chiamate per tutti i file *.html* che vengono aperti ad eccezione di *home.html*), passiamo all'analisi della schermata relativa al file *switch.html*.

L'Illustrazione 28 mette in mostra la prima parte di ciò che l'utente si troverà di fronte se, come supposto, cliccherà il bottone relativo allo switch. La prima funzionalità offerta dalla web app riguarda il numero di switch connessi al controller. Cliccando sul bottone **NUMERO SWITCH** infatti, verrà generata una chiamata REST di tipo GET che permetterà di ottenere in output il numero effettivo degli switch connessi.


```

6  function numeroSwitch() {
7      var c1 = document.getElementById('ip').value;
8      var xmlhttp = new XMLHttpRequest();
9      var el = document.getElementById("switchPresenti");
10     /* con la try catch posso verificare se l'indirizzo IP del controller inserito è corretto */
11     try{
12         xmlhttp.open("GET", getProtocol() + c1 + getLocalhost() + getNumberSwitches(), false);
13         xmlhttp.onreadystatechange = checkReadyState;
14         function checkReadyState() {
15             if (xmlhttp.readyState === 4) {
16                 if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
17                     var data = JSON.parse(xmlhttp.responseText);
18                     var numSw = JSON.stringify(data);
19                     var parseSw = numSw.slice(1,numSw.length - 1).bold();
20                     el.innerHTML = parseSw;
21                 } else {
22                     el.innerHTML = "Nessuno switch presente nella rete SDN".bold();
23                 }
24                 return;
25             }
26         }
27         xmlhttp.send(null);
28     } catch (error){
29         el.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
30     }
31 }
32

```

Illustrazione 29: funzione per conoscere il numero di switch connessi al controller

La funzione *numeroSwitch()*, contenuta all'interno del file *switchFunction.js* (file al cui interno saranno contenute tutte le funzioni JavaScript relative a questa pagina .html) verrà richiamata non appena l'utente cliccherà il bottone per conoscere il numero degli switch. Prima di tutto verrà preso il riferimento all'IP inserito dall'utente (riga 7), poi verrà creato un oggetto di tipo XMLHttpRequest che permetterà di effettuare la chiamata REST. Dopodiché tramite il metodo *.open()* (riga 12) si potrà generare la chiamata. Il primo argomento rappresenta il tipo di chiamata, in questo caso GET. Da qui in poi comincia l'astrazione che ho messo in pratica per disaccoppiare il più possibile la parte .html da quella .js, ma soprattutto in previsione di modifiche future. La funzione *getProtocol()* è contenuta all'interno del file *restRyu.js* e mi restituisce il protocollo utilizzato (qui la stringa *http://*). La funzione *getLocalhost()* è anch'essa contenuta nel file *restRyu.js* e mi restituisce la porta del server web (qui la stringa *:8080/*) mentre la funzione *getNumberSwitches()* mi restituisce la chiamata REST da effettuare (qui *stats/switches*). Questo è il procedimento che ho adottato in tutte le funzioni .js per poter rendere più pulito il codice ma soprattutto modificabile in futuro. Se ad esempio dovesse cambiare la sintassi di una chiamata REST, basterà recarsi nel file *restRyu.js* dove sono contenute tutte le chiamate REST di Ryu e modificarla una volta sola. Successivamente,

quando cambia lo stato della mia richiesta, verifico che la risposta della chiamata sia la 200, che rappresenta l'esito positivo della chiamata stessa e catturo l'output nella variabile *data* dopo averne "parsato" il risultato. A questo punto ne serializzo il contenuto con la funzione *JSON.stringify()*, elaboro l'output ottenuto e lo stampo a video (riga 20). Qualora la connessione non fosse riuscita perché l'IP inserito non è stato trovato, verrebbe stampato un messaggio per invitare l'utente a controllare meglio l'IP che è stato inserito.

ANALISI SWITCH

Inserire il datapath dello switch da analizzare:

Cliccare il bottone per conoscere le caratteristiche dello switch: **STATS**

Switch: 1
Versione software : 2.0.2
Versione hardware : Open vSwitch
Serial number : None
Costruttore : Nicira, Inc.

ANALISI PORTE SWITCH

Inserire il datapath dello switch da analizzare:

Cliccare il bottone per conoscere le statistiche delle porte dello switch: **STATS**

Lo switch inserito non è presente

Illustrazione 30: seconda parte della schermata del file switch.html

Nell'illustrazione 30 vengono presentate le altre due funzioni relative a questa pagina. Si tratta di **ANALISI SWITCH**, che restituirà in output le caratteristiche dello switch analizzato come ad esempio versione software e versione hardware, e **ANALISI PORTE SWITCH**, dove invece verranno sottolineate le caratteristiche delle porte di un certo switch (pacchetti transitati, byte totali, ecc). Per entrambe le funzioni è richiesto l'inserimento del datapath dello

switch, che lo identifica in maniera univoca rispetto agli altri eventuali switch presenti nella rete.

```
35 function detailsSwitch(){
36     var el = document.getElementById("statsSwitch");
37     el.innerHTML = "";
38     var c1 = document.getElementById('ip').value;
39     var c2 = document.getElementById('datapathAnalisi').value;
40     var xmlhttp = new XMLHttpRequest();
41
42     try {
43         xmlhttp.open("GET", getProtocol() + c1 + getLocalhost() + getStatsSwitch() + c2, false);
44         xmlhttp.onreadystatechange = checkReadyState;
45         function checkReadyState() {
46             if (xmlhttp.readyState === 4) {
47                 if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
48                     var data = JSON.parse(xmlhttp.responseText);
49                     var bella = JSON.stringify(data);
50                     // creo due array splittati. Il primo diviso in base alle virgole, il secondo in base ai due punti
51                     var prova = bella.split(',');
52                     var def = bella.split(':');
53                     var el = document.getElementById("statsSwitch");
54
55                     // numero switch
56                     var switch1 = "Switch: ".bold() + " " + def[0].slice(2, def[0].length - 1);
57                     el.innerHTML += switch1 + "<br/>";
58
59                     // versione sw dello switch
60                     var versSw = "Versione software : ".bold() + " " + prova[1].slice(11, prova[1].length - 1);
61                     el.innerHTML += versSw + "<br/>";
62
63                     // versione hw dello switch
64                     var versHw = "Versione hardware : ".bold() + " " + prova[2].slice(11, prova[2].length - 1);
65                     el.innerHTML += versHw + "<br/>";
66
67                     // serial number dello switch
68                     var serialN = "Serial number : ".bold() + " " + prova[3].slice(14, prova[3].length - 1);
69                     el.innerHTML += serialN + "<br/>";
70
71                     // costruttore dello switch
72                     var costruttore = "Costruttore : ".bold() + " " + def[6].slice(1, def[6].length - 3);
73                     el.innerHTML += costruttore + "<br/>";
74                 } else {
75                     var el = document.getElementById("statsSwitch");
76                     el.innerHTML = "Lo switch inserito non è presente".bold();
77                     return;
78                 }
79             }
80         }
81         xmlhttp.send(null);
82     } catch (error){
83         el.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
84     }
85 }
```

Illustrazione 31: funzione che viene chiamata per le caratteristiche di uno switch

La funzione *detailsSwitch()* viene chiamata non appena l'utente clicca il bottone STATS relativo alla parte di analisi dello switch. Anche questa funzione, come la precedente, si trova nel file esterno *switchFunction.js*. Anche in questo caso, recupero l'indirizzo IP del controller (riga 38) e in più il datapath dello switch che si vuole analizzare (riga 39). Successivamente chiamo la *.open()* per la chiamata REST passando come parametro finale la *getStatsSwitch()*, che mi

restituisce la stringa relativa alla chiamata da fare (in questo caso */stats/desc/*). Dopo aver ricevuto la risposta e salvato l'output, ho creato due array partendo da quello originale con la funzione *split()*. Nel primo caso ho diviso l'output dopo ogni ',', mentre nel secondo dopo ogni ':'. In questo modo ho potuto affrontare in maniera più semplice l'elaborazione dell'output fornitomi in precedenza. Nell'Illustrazione 30 c'è un esempio del risultato finale: dopo aver inserito il datapath dell'unico switch presente nella rete, l'elaborazione è stata mostrata a video. Nel form relativo al datapath non c'è più nulla poiché ogni volta che viene chiamata una funzione che ha bisogno di un datapath, di seguito viene invocata la relativa funzione contenuta nel file esterno *reset.js* che permette di "svuotare" il form per un successivo utilizzo.

```
for (b=0; b <= prova.length; ++b) {
  if (z == 0) {
    // se sono al primo ciclo, ovvero alla prima riga della risposta, estrapolo il n° dello switch, altrimenti no
    if (b == 0) {
      var numeroSwitch = def[0].slice(2, def[0].length -1);
      el.innerHTML += "Switch: ".bold() + numeroSwitch + "<br/><br/>";
      var packetTDropped = prova[b].slice(20, prova[b].length );
      el.innerHTML += "Pacchetti trasmessi scartati: ".bold() + packetTDropped + "<br/>";
    } else {
      var packetTDropped = prova[b].slice(14, prova[b].length);
      el.innerHTML += "Pacchetti trasmessi scartati: ".bold() + packetTDropped + "<br/>";
    }
  } else if (z == 1) {
    var packetReceived = prova[b].slice(13, prova[b].length);
    el.innerHTML += "Pacchetti ricevuti: ".bold() + packetReceived + "<br/>";
  } else if (z == 3) {
    var byteTransmitted = prova[b].slice(11, prova[b].length);
    el.innerHTML += "Byte trasmessi: ".bold() + byteTransmitted + "<br/>";
  } else if (z == 4) {
    var packetRDropped = prova[b].slice(13, prova[b].length);
    el.innerHTML += "Pacchetti ricevuti scartati: ".bold() + packetRDropped + "<br/>";
  } else if (z == 5) {
    var port = prova[b].slice(10, prova[b].length);
    el.innerHTML += "Numero di porta analizzato: ".bold() + port + "<br/>";
  } else if (z == 8) {
    var byteReceived = prova[b].slice(11, prova[b].length);
    el.innerHTML += "Byte ricevuti: ".bold() + byteReceived + "<br/>";
  } else if (z == 14) {
    if (b != prova.length - 1) {
      var packetTransmitted = prova[b].slice(13, prova[b].length - 1);
      el.innerHTML += "Pacchetti trasmessi: ".bold() + packetTransmitted + "<br/><br/>";
      z = - 1;
    } else {
      var packetTransmitted = prova[b].slice(13, prova[b].length - 3);
      el.innerHTML += "Pacchetti trasmessi: ".bold() + packetTransmitted + "<br/><br/>";
      z = - 1;
    }
  }
  z++;
}
} else {
  var el = document.getElementById("statsPortsSwitch");
  el.innerHTML = "Lo switch inserito non è presente".bold();
  return;
}
}
```

Illustrazione 32: parte del codice relativo alla funzione per l'analisi delle porte

L'Illustrazione 32 rappresenta una parte del codice della funzione chiamata quando invece l'utente clicca sul bottone STATS relativo all'analisi delle porte di uno switch. La prima parte, che è quella che ho ommesso, è la stessa delle altre funzioni, dove prelevo l'indirizzo IP e il datapath inserito dall'utente. La parte che ho messo in evidenza è forse la più interessante da analizzare della funzione *portsSwitch()*. Il ciclo *for* rappresenta il cuore della funzione: è lì dentro infatti che l'output viene elaborato per poter essere reso presentabile e fruibile agli occhi dell'utente. Inizialmente mi ero chiesto come poter fare per elaborare delle informazioni di cui non conosco il numero esatto. In precedenza infatti sapevo che la risposta sull'analisi delle caratteristiche dello switch mi avrebbe restituito solamente un "blocco" di informazioni formato da un numero ben specifico di righe. Qui invece dipende da quante porte ha ciascuno switch. Per ovviare a questo problema, ho inserito una variabile *z* che inizialmente è posta a 0. Ogni ciclo invece è caratterizzato dalla variabile *b*, che viene incrementata ogni volta che termina un'istruzione. Il ciclo ovviamente terminerà una volta analizzati tutti i dati dell'array. Dal momento che, in questo caso, ogni blocco di informazioni è formato da 14 righe (dopo aver *splittato* e realizzato l'array), la variabile *z* può arrivare a valere al massimo 14, per poi essere posta a -1 prima dell'incremento che la porterà ad essere pari a 0, e di conseguenza a ricominciare il giro. L'unica incongruenza che mi ha creato qualche piccola complicazione è stata data dal fatto che l'ultimo blocco informativo terminava sempre con due '}' che nei blocchi precedenti non c'erano. Per cui ogni volta che si arrivava al "passo 14" ($z = 14$) bisognava anche verificare che *b* fosse diversa dalla dimensione dell'array - 1: in quel caso continuavo normalmente, altrimenti significava che eravamo giunti al termine dell'array e allora andavano anche eliminati gli ultimi due caratteri in eccesso. Terminato il ciclo, verranno stampati i dettagli relativi all'analisi delle porte dello switch indicato, come ad esempio il numero di pacchetti trasmessi, i pacchetti inviati, quelli rifiutati e la quantità di byte transitata in entrata e in uscita. Nell'Illustrazione 30 ho inserito un datapath relativo ad uno switch che non era connesso al controller e il messaggio in output è stato "*Lo switch inserito non è presente*" perché ovviamente quello switch non faceva parte della rete SDN capitanata dal controller inserito.

6.7 Pagina flowSwitch.html e relativi script

Supponiamo invece che adesso l'utente decida di cliccare sul pulsante "FLOW STATS" o di spostarsi in questa pagina tramite il menù laterale, accessibile da qualunque pagina. Una volta cliccato, si verrà reindirizzati alla pagina *flowSwitch.html* e nel frattempo verrà richiamata la funzione *setScript()* contenuta nel file *passaggioIP.js* analizzata nei paragrafi precedenti.

The screenshot displays a web interface with two main sections: "IP CONTROLLER" and "FLOW STATS".

IP CONTROLLER
Inserire l'indirizzo IP del Controller:

FLOW STATS
Inserire il datapath dello switch da analizzare:

Cliccare il bottone per conoscere i flow nello switch:

Switch: 1

Azione: OUTPUT:1
Idle timeout: 0
Cookie: 0
Pacchetti transitati: 3
Priorita': 1
Table ID: 0
Porta d'ingresso: 2

Azione: OUTPUT:2
Idle timeout: 0
Cookie: 0
Pacchetti transitati: 2
Priorita': 1
Table ID: 0
Porta d'ingresso: 1

Azione: OUTPUT:CONTROLLER
Idle timeout: 0
Cookie: 0
Pacchetti transitati: 3
Priorita': 0
Table ID: 0

Illustrazione 33: prima parte della schermata del file *flowSwitch.html*

Nell'illustrazione 33 è possibile notare la prima funzionalità relativa a questa pagina, ovvero FLOW STATS. Questa funzione rappresenta forse la più importante di tutta l'applicazione, poiché consente all'utente di poter capire il tipo di traffico interno ad un certo switch e relative statistiche. Non appena l'utente avrà inserito nel form il datapath dello switch che vuole analizzare e avrà cliccato il bottone FLOW, ciò che comparirà (se il datapath inserito è corretto) sarà molto simile a ciò che vediamo nell'illustrazione 33. Le informazioni che possono essere estrapolate sono il tipo d'azione di un certo flusso, l'idle_timeout, il cookie, i pacchetti transitati, la priorità, la table ID e la porta d'ingresso. Qualora non ci fosse stato alcuno scambio di pacchetti, ciò che vedremmo sarebbe solamente l'azione relativa al Controller. In questo caso invece avevo lanciato il comando *pingall* per cui sono apparsi anche altri due flussi, relativi ai due terminali in contatto.

```

27     for (b=0; b <= prova.length; ++b) {
28         if (z == 0) {
29             if (prova.length == 13) {
30                 var azione = "Azione:".bold() + " " + prova[b].slice(19, prova[b].length - 2);
31                 el.innerHTML += azione + "<br/>";
32             } else if (b == 0) {
33                 if (prova.length < 13) {
34                     el.innerHTML = "Lo switch inserito non ha flow".bold();
35                 } else {
36                     var numeroSwitch = first[b].slice(2, first[b].length - 1);
37                     el.innerHTML += "Switch: ".bold() + numeroSwitch + "<br/><br/>";
38                     var azione = "Azione:".bold() + " " + second[5];
39                     el.innerHTML += azione + "<br/>";
40                 }
41             } else {
42                 var azione = "Azione:".bold() + " " + prova[b].slice(13, prova[b].length - 2);
43                 el.innerHTML += azione + "<br/>";
44             }
45         } else if (z == 1) {
46             var azione = "Idle timeout:".bold() + " " + prova[b].slice(15, prova[b].length);
47             el.innerHTML += azione + "<br />";
48         } else if (z == 2) {
49             var azione = "Cookie:".bold() + " " + prova[b].slice(9, prova[b].length);
50             el.innerHTML += azione + "<br />";
51         } else if (z == 3) {
52             var azione = "Pacchetti transitati:".bold() + " " + prova[b].slice(15, prova[b].length);
53             el.innerHTML += azione + "<br />";
54         } else if (z == 8) {
55             var azione = "Priorita:".bold() + " " + prova[b].slice(11, prova[b].length);
56             el.innerHTML += azione + "<br />";
57         } else if (z == 11) {
58             var azione = "Table ID:".bold() + " " + prova[b].slice(11, prova[b].length);
59             el.innerHTML += azione + "<br />";
60         }
61         else if (z == 12) {
62             if (b == prova.length - 1) {
63                 if (prova[b].slice(8, 10) == '{}') {
64                     /* significa che sono nel controller e non c'è bisogno della porta d'ingresso */
65                 } else {
66                     var azione = "Porta d'ingresso:".bold() + " " + prova[b].slice(19, prova[b].length - 4);
67                     el.innerHTML += azione + "<br/><br/>";
68                     z = -1;
69                 }
70             } else if (prova[b].slice(10, 11) == 'd'){
71             } else {
72                 var azione = "Porta d'ingresso:".bold() + " " + prova[b].slice(19, prova[b].length - 2);
73                 el.innerHTML += azione + "<br/><br/>";
74                 z = -1;
75             }
76         } else if (z == 13) {
77             if (b == prova.length - 1) {
78                 var azione = "Porta d'ingresso:".bold() + " " + prova[b].slice(10,prova[b].length - 4);
79                 el.innerHTML += azione + "<br/><br/>";
80                 z = -1;
81             } else {
82                 var azione = "Porta d'ingresso:".bold() + " " + prova[b].slice(10,prova[b].length - 2);
83                 el.innerHTML += azione + "<br/><br/>";
84                 z = -1;

```

Illustrazione 34: parte centrale della funzione flow()

L'Illustrazione 34 rappresenta la parte centrale della funzione *flow()* che si trova nel file *flowFunction.js* ed è la responsabile dell'elaborazione dell'output ricevuto in seguito alla chiamata per visualizzare tutti i flow installati sopra ad uno specifico switch. Anche in questo caso ho evitato di mostrare la parte relativa alla chiamata iniziale, che prevede come per le altre funzioni la cattura dell'IP e del datapath dello switch che si vuole analizzare. Una volta catturato l'output, bisognerà renderlo in maniera più chiara rispetto a come ci viene presentato inizialmente. Come nel caso dell'analisi delle porte dello switch, anche qui non so per certo il numero di informazioni che la chiamata mi restituirà, perché ogni volta potrei avere un numero diverso di regole all'interno dello switch. Quindi anche qui mi sono avvalso dell'utilizzo del ciclo for, con le variabili *b* e *z* analizzate in precedenza. Per questa funzione ho dovuto addirittura creare tre array diversi (prova, first e second) dopo aver splittato l'output iniziale sulla base di ' , ', ' : ' e ' “ ‘ per poter gestire in maniera più semplificata i passi successivi. La prima difficoltà era rappresentata dal fatto che ci fosse un unico flow installato, quello relativo al controller. In quel caso (riga 29) ho verificato che la lunghezza totale dell'array principale fosse 13 (la dimensione del blocco di ogni informazione) perché alla voce "Azione" l'output del controller è più lungo rispetto agli altri casi (ad esempio OUTPUT: 1 e OUTPUT:CONTROLLER). Se invece *b* = 0 significa che è la prima iterazione che compio e devo verificare che sia almeno un flusso: qualora la dimensione dell'array fosse < 13, stamperei a video la dicitura "Lo switch inserito non ha flow". Dopodiché tutto procede in maniera regolare fino al "passo 12" (*z* = 12). Ogni volta devo verificare se mi trovo in un flusso relativo al controller oppure no, perché nel primo caso non ci sarebbe bisogno di stampare la porta d'ingresso poiché nell'output del controller ciò non è presente. Per fare ciò analizzo due caratteri dell'array ad una certa iterazione: se sono uguali a '{ }' significa che sono nel controller per cui evito di effettuare stampe, altrimenti procedo e stampo la porta d'ingresso del flusso attualmente analizzato. Un'incongruenza del controller Ryu riguarda invece una diversa interpretazione della chiamata per visualizzare le statistiche del flusso. Se infatti faccio la richiesta per visualizzarla (GET), in output mi verrà restituito il campo *dl_dest* che rappresenta l'host destinatario, mentre nella chiamata per aggiungere un flow manualmente (POST, lo analizzerò successivamente) il campo *dl_dest* non è richiesto. Per evitare incongruenze ho dunque preferito evitarlo anche nella visualizzazione del flusso, e ciò si evince alla riga 70: se un certo carattere è uguale a 'd' (primo carattere di *dl_dest*) allora evito quel campo e non lo mostro, altrimenti significa che sto analizzando un flusso inserito dall'utente manualmente e allora quel campo non appare proprio non essendo richiesto.



Illustrazione 35: seconda parte della schermata del file flowSwitch.html

L'illustrazione 35 rappresenta la seconda parte della schermata della pagina relativa alle statistiche dei flussi. La funzione **AGGREGATE FLOW STATS** permette di ricevere in output il datapath dello switch analizzato, il numero di pacchetti transitati attraverso lo switch scelto, il numero di byte complessivi e il numero totale di flow attualmente installati sopra lo switch. E' una funzione utile perché rispetto alla precedente permette di avere una visione d'insieme dei flussi relativi ad uno specifico switch. In questo caso, prima di premere il bottone **FLOW** che richiama la funzione `getFunctionAggregateFlow()` contenuta nel file `functionRyu.js`, la quale a sua volta richiama la `flowAggregate()`, ho inserito come datapath l'unico presente all'interno della rete e l'output presentato è la risposta giunta dal controller. In questo caso i pacchetti totali transitati all'interno dello switch sono 8 (la somma di quelli all'interno dell'illustrazione 33), i byte 560 e ci sono attualmente 3 flussi installati, che sono quelli relativi al controller e ai due terminali.

```

105 function flowAggregate(){
106     var el = document.getElementById("flowAggrPresenti");
107     el.innerHTML = "";
108     var c1 = document.getElementById('ip').value;
109     var c2 = document.getElementById('datapath3').value;
110     var xmlhttp = new XMLHttpRequest();
111     try{
112         xmlhttp.open("GET", getProtocol() + c1 + getLocalhost() + getStatsFlowAggregate() + c2, false);
113         xmlhttp.onreadystatechange = checkReadyState;
114         function checkReadyState() {
115             if (xmlhttp.readyState === 4) {
116                 if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
117                     var data = JSON.parse(xmlhttp.responseText);
118                     var bella = JSON.stringify(data);
119                     var prova = bella.split(',');
120                     var el = document.getElementById("flowAggrPresenti");
121
122                     var switchAnalizz = "Switch analizzato: ".bold() + c2;
123                     el.innerHTML += switchAnalizz + "<br/>";
124
125                     var packetCount = "Packet count: ".bold() + prova[0].slice(22, prova[0].length);
126                     el.innerHTML += packetCount + "<br/>";
127
128                     var byteCount = "Byte count: ".bold() + prova[1].slice(13, prova[1].length);
129                     el.innerHTML += byteCount + "<br/>";
130
131                     var flowCount = "Flow count: ".bold() + prova[2].slice(13, prova[2].length - 3);
132                     el.innerHTML += flowCount + "<br/>";
133                 } else {
134                     var el = document.getElementById("flowAggrPresenti");
135                     el.innerHTML = "Lo switch inserito non è presente".bold();
136                     return;
137                 }
138             }
139         }
140         xmlhttp.send(null);
141     } catch (error){
142         el.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
143     }
144 }

```

Illustrazione 36: codice relativo alla funzione flowAggregate()

La funzione viene richiamata non appena si clicca sul bottone FLOW. E' molto semplice in quanto siamo certi di ottenere uno specifico numero di informazioni, per cui non ci sarà bisogno di utilizzare un ciclo for. Alla riga 112 è possibile notare come si sia lavorato in un'ottica futura: utilizzando le funzioni getProtocol(), getLocalhost() e getStatsFlowAggregate() basterà modificare solamente il loro contenuto all'interno del file restRyu.js per avere tutte le chiamate aggiornate. Se invece avessi inserito manualmente http:// oppure :8080/, nel caso in cui il localhost ad esempio fosse diventato :8888/, avrei dovuto cambiare manualmente tutte le diciture in tutte le funzioni in cui ne facevo uso. Il processo di parsing rimane uguale a quelli usati in precedenza: creo un array splittato dopo aver serializzato l'output e poi su quello costruisco l'output finale che verrà presentato all'utente.

6.8 Pagina addRule.html e relativi script

Dopo aver analizzato la pagina relativa ai flussi di uno switch, passiamo ora alla penultima pagina .html che riguarda l'aggiunta di un flusso all'interno di uno specifico switch. Come nei casi precedenti, l'utente potrà raggiungere questa pagina tramite l'apposito pulsantone presente nella home oppure tramite il menù laterale presente all'interno di tutte le pagine .html.

Manage flow

IP CONTROLLER

Inserire l'indirizzo IP del controller:

192.168.56.101

ADD FLOW

Switch:

Idle timeout:

Porta d'ingresso:

Porta d'uscita:

Priorità:

Cliccare il bottone per aggiungere la regola: **ADD FLOW**

Illustrazione 37: schermata relativa alla pagina addRule.html

L'unica funzionalità di questa pagina è quella che appare nell'Illustrazione 37, ovvero la possibilità di aggiungere un flusso all'interno di uno switch. Per poter mettere in pratica ciò sarà necessario che l'utente inserisca il datapath dello switch sul quale vuole aggiungere il flusso, l'idle_timeout per dire dopo quanti secondi la regola deve essere eliminata se non giungono pacchetti, la porta d'ingresso, la porta d'uscita e la priorità che deve avere il flow. Una volta inserito tutto ciò basterà cliccare sul bottone ADD FLOW e verrà richiamata la funzione *getFunctionRule()* che a sua volta richiamerà la funzione *rule()* contenuta nel file *manageFlowFunction.js*. Se si dovesse inserire un datapath non presente all'interno della rete verrebbe stampato il messaggio "La regola non è stata aggiunta". Se invece si dovesse inserire un flusso che ha come porte d'ingresso e d'uscita le stesse di un flusso già esistente, verrebbe data la precedenza a quello dei due che ha una priorità maggiore.

```
function rule() {
  var c1 = document.getElementById('ip').value;
  var switchAdd = document.getElementById('switchAdd').value;
  var inAdd = document.getElementById('inAdd').value;
  var outAdd = document.getElementById('outAdd').value;
  var priorityAdd = document.getElementById('priorityAdd').value;
  var idleTimeout = document.getElementById('idle').value;
  var e1 = document.getElementById("ruleOK");
  var xmlhttp = new XMLHttpRequest();
  try {
    xmlhttp.open("POST", getProtocol() + c1 + getLocalhost() + getAddRule(), false);
    xmlhttp.onreadystatechange = checkReadyState;
    function checkReadyState() {
      if (xmlhttp.readyState == 4) {
        if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
          e1.innerHTML = "La regola è stata aggiunta".bold();
        } else {
          e1.innerHTML = "La regola non è stata aggiunta".bold();
          return;
        }
      }
    }
    /* dopo aver messo nelle variabili ciò che ha scritto l'utente, faccio la richiesta POST per aggiungere a regola */
    var request = "{\"dpid\":\" + switchAdd + "\",\"idle_timeout\":\" + idleTimeout + "\",\"priority\":\" + priorityAdd + "\",\"match\":{\"\"in_port\":\" + inAdd + "\",\"actions\":{\"\"type\":\"OUTPUT\",\"port\":\" - outAdd + \"}}}\";
    xmlhttp.send(request);
  } catch (error) {
    var e1 = document.getElementById("ruleOK");
    e1.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
  }
}
```

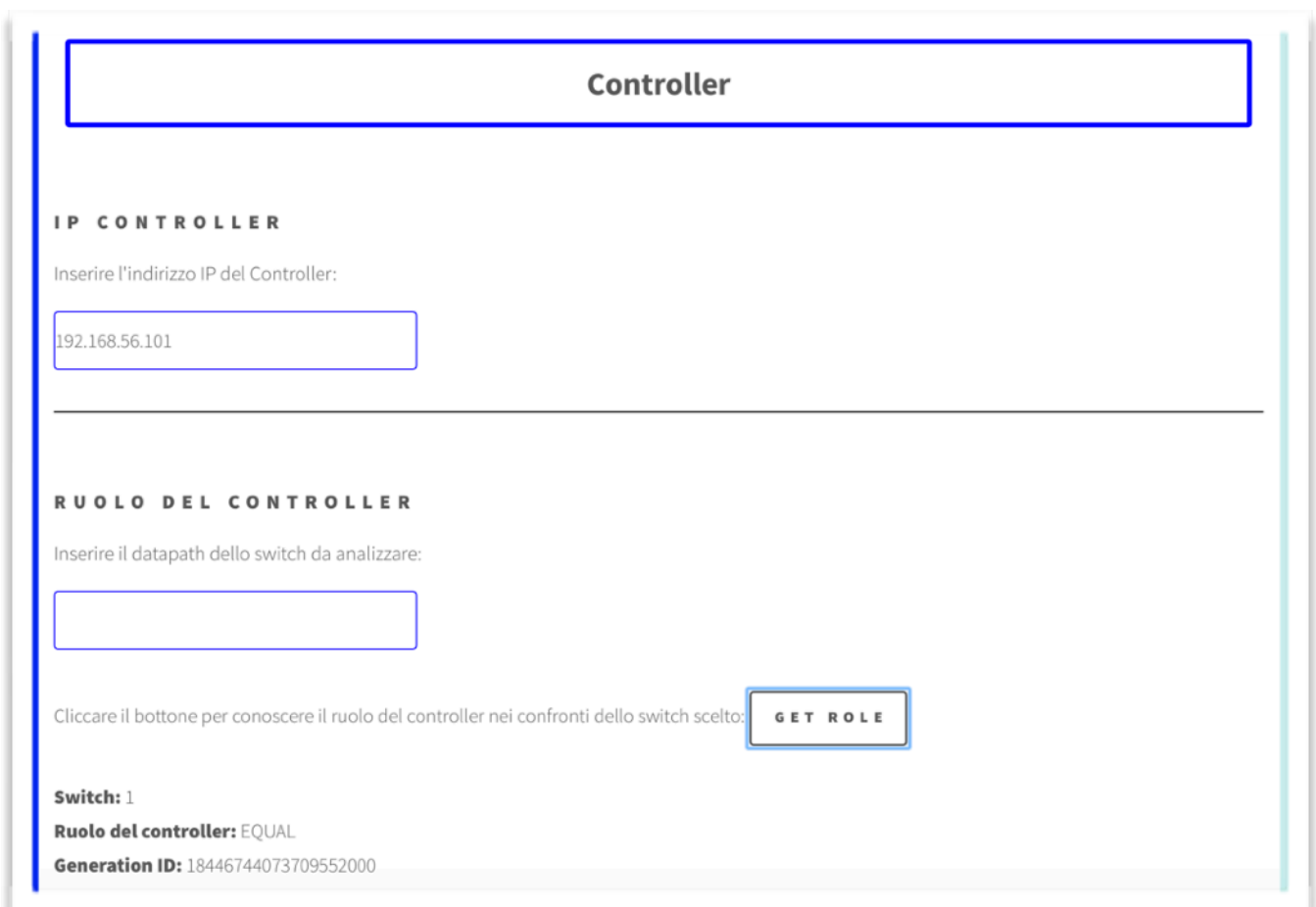
Illustrazione 38: funzione rule() richiamata alla pressione del bottone ADD FLOW

La funzione dell'Illustrazione 38 viene richiamata non appena l'utente clicca il bottone per aggiungere la regola da lui inserita all'interno dello switch indicato. E' la prima chiamata di tipo POST che utilizziamo all'interno dell'applicazione. Tutto procede come al solito: vengono recuperate tutte le informazioni inserite dall'utente nei form della pagina e viene creato un oggetto di tipo XMLHttpRequest. A questo punto si inoltra la chiamata tramite il metodo *.open()*, al quale verranno passati come argomenti il tipo di chiamata (POST), il protocollo, l'IP, il localhost e la chiamata REST che vogliamo effettuare (in questo caso chiamo la funzione *getAddRule()* che mi restituirà la stringa *stats/flowentry/add*). In questo caso è stato abbastanza complesso realizzare la stringa *request* (riga 28) a causa di tutti i caratteri speciali che la

chiamata Ryu richiedeva. Un'altra differenza rispetto alle chiamate GET è che nel momento in cui si inoltra la richiesta tramite il metodo *.send()* bisogna passare una variabile contenente la richiesta, mentre nella GET non bisognava inserire nulla.

6.9 Pagina controller.html e relativi script

Siamo giunti all'analisi dell'ultima pagina .html relativa all'interfaccia web per gestire e visualizzare statistiche relative ad una rete SDN. Come spiegato nella parte di teoria, ogni switch può essere collegato a più controller, anche di tipologie diverse. Ogni controller riveste un ruolo (equal, master o slave) per ogni specifico switch: per lo switch 1 ad esempio potrebbe essere equal, mentre per lo switch 2 potrebbe essere master. All'interno della pagina controller.html, raggiungibile come al solito o dalla home principale o dal menù laterale, l'utente avrà la possibilità di richiedere il ruolo di un controller per uno specifico switch o di modificarlo in base alle proprie esigenze.



The screenshot shows a web form titled "Controller". It is divided into two main sections: "IP CONTROLLER" and "RUOLO DEL CONTROLLER".

IP CONTROLLER
Inserire l'indirizzo IP del Controller:

RUOLO DEL CONTROLLER
Inserire il datapath dello switch da analizzare:

Cliccare il bottone per conoscere il ruolo del controller nei confronti dello switch scelto:

Switch: 1
Ruolo del controller: EQUAL
Generation ID: 18446744073709552000

Illustrazione 39: prima parte della schermata relativa al file controller.html

Come si può notare dall'Illustrazione 39, la prima funzionalità messa a disposizione dalla pagina riguarda la visualizzazione del ruolo del controller per un certo switch e il suo generation ID. In questo caso ho inserito come datapath 1, che rappresenta l'unico switch disponibile presente all'interno della nostra

attuale rete. Il risultato segnala appunto lo switch scelto, il ruolo che ha il controller nei confronti dello switch indicato e il suo generation ID.

```
5 function getRoleController(){
6   var c1 = document.getElementById('ip').value;
7   var c2 = document.getElementById('datapath').value;
8   var xmlhttp = new XMLHttpRequest();
9   var el = document.getElementById("roleController");
10  el.innerHTML = "";
11  /* con la try catch posso verificare se l'indirizzo IP del controller inserito è corretto */
12  try{
13    xmlhttp.open("GET", getProtocol() + c1 + getLocalhost() + getStatsRole() + c2, false);
14    xmlhttp.onreadystatechange = checkReadyState;
15    function checkReadyState() {
16      if (xmlhttp.readyState === 4) {
17        if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
18          var data = JSON.parse(xmlhttp.responseText);
19          var roleC = JSON.stringify(data);
20          var roleSplit = roleC.split(',');
21          var roleSplitSwitch = roleC.split(':');
22          var switchN = roleSplitSwitch[0].slice(2, roleSplitSwitch[0].length - 1);
23          el.innerHTML += "<br/>Switch: ".bold() + switchN + "<br/>";
24          var role = roleSplit[0].slice(15, roleSplit[0].length - 1);
25          el.innerHTML += "Ruolo del controller: ".bold() + role + "<br/>";
26          var genID = roleSplit[1].slice(16, roleSplit[1].length - 3);
27          el.innerHTML += "Generation ID: ".bold() + genID + "<br/>";
28        } else {
29          el.innerHTML = "Lo switch richiesto non è presente nella rete SDN".bold();
30          return;
31        }
32      }
33    }
34    xmlhttp.send(null);
35  } catch (error){
36    el.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
37  }
38 }
39
```

Illustrazione 40: funzione `getRoleController()` chiamata quando l'utente clicca sul bottone

Nell'illustrazione 40 è rappresentata la funzione `getRoleController()`, che viene richiamata non appena l'utente clicca sul bottone GET ROLE. Dopo aver catturato l'indirizzo IP del controller e il datapath relativo allo switch scelto, viene effettuata la richiesta al controller tramite la chiamata GET. Come per le altre circostanze, vengono passati al metodo `.open()` i parametri relativi al protocollo, all'indirizzo IP, al localhost, alla chiamata REST da effettuare (in questo caso la funzione `getStatsRole()` restituirà la stringa `stats/role/`) e allo switch scelto. Il parsing è molto semplice perché riceveremo sempre in output, dopo aver applicato la funzione `.split()`, un blocco informativo composto da tre righe.

MODIFICA RUOLO DEL CONTROLLER

Inserire il datapath dello switch di cui si vuol modificare il ruolo del controller:

Scegliere quale ruolo assegnare al controller:

Cliccare il bottone per modificare il ruolo del controller nei confronti dello switch scelto:

Il ruolo del controller è stato modificato

Illustrazione 41: seconda parte della schermata relativa al file controller.html

L'illustrazione 41 rappresenta la seconda parte della schermata relativa al controller. Mentre nella precedente c'era la possibilità di visualizzare le statistiche relative ad un certo controller nei confronti di un certo switch, in questo caso viene data la possibilità all'utente di modificare il ruolo del controller nei confronti di un certo switch. L'utente dovrà inserire nel form apposito il datapath dello switch e nel combobox dovrà selezionare uno dei tre ruoli possibili da assegnare al controller. Dopo aver fatto ciò, alla pressione del bottone `EDIT ROLE` verrà richiamata la funzione `getFunctionEditRoleController()`, che a sua volta chiamerà la `editRoleControllerSwitch()` presente all'interno del file `controllerFunction.js`. In questo caso era stato inserito l'unico datapath disponibile all'interno della rete attuale, scelto il ruolo `Slave` e cliccato il bottone. Se l'operazione è andata a buon fine, verrà stampato il messaggio *"Il ruolo del controller è stato modificato"*, per segnalare all'utente che l'operazione è stata portata correttamente a termine.


```

43 function editRoleControllerSwitch(){
44     var c1 = document.getElementById('ip').value;
45     var switchDatapath = document.getElementById("datapath1").value;
46     var el = document.getElementById("editRoleController");
47     var resultRole = document.getElementById("roleContr");
48     var strUser = resultRole.options[resultRole.selectedIndex].text.toUpperCase();
49     var xmlhttp = new XMLHttpRequest();
50     try {
51         xmlhttp.open("POST", getProtocol() + c1 + getLocalhost() + getStatsEditRole(), false);
52         xmlhttp.onreadystatechange = checkReadyState;
53         function checkReadyState() {
54             if (xmlhttp.readyState === 4) {
55                 if ((xmlhttp.status == 200) || (xmlhttp.status == 0)) {
56                     el.innerHTML = "Il ruolo del controller è stato modificato".bold();
57                 } else {
58                     el.innerHTML = "Il ruolo del controller non è stato modificato".bold();
59                 }
60             }
61         }
62     }
63     /* dopo aver messo nelle variabili ciò che ha scritto l'utente, faccio la richiesta POST per aggiungere a regola */
64     var request = "{\"dpid\":\"" + switchDatapath + "\",\"role\":\"" + strUser + "\"}";
65     xmlhttp.send(request);
66 } catch (error){
67     var el = document.getElementById("ruleOK");
68     el.innerHTML = "Verificare la correttezza dell'indirizzo IP del Controller inserito".bold();
69 }
70 }

```

Illustrazione 42: codice della funzione editRoleControllerSwitch()

L'illustrazione 42 mostra il codice dell'ultima funzionalità messa a disposizione dalla web app, ovvero quella che darà la possibilità all'utente di modificare il ruolo di un controller nei confronti di un certo switch. Dopo aver recuperato l'indirizzo IP, il datapath dello switch e il ruolo che si vuole assegnare al controller (convertito poi in caratteri maiuscoli grazie al metodo *toUpperCase()* visto che la sintassi della chiamata REST prevede che il ruolo sia totalmente scritto in maiuscolo) viene effettuata la chiamata tramite metodo POST. Come nel caso dell'aggiunta di un flow, dunque, sarà necessario creare una variabile al cui interno inserire la stringa per poter modificare il ruolo del controller (riga 64). Nel momento in cui la richiesta verrà presa in carico, verrà stampato il messaggio del buon esito dell'operazione (riga 56) o viceversa (riga 58).

6.10 Analisi del file reset.js

```
5 /***** FILE SWITCH.HTML *****/
6
7 /* Prima funzione: ANALISI SWITCH. Funzione per pulire il campo in cui l'utente inserisce il datapath per analizzare le caratteristiche dello switch */
8 function resetFormSwitch(){
9     var datapathAnalisi = document.getElementById("datapathAnalisi");
10    datapathAnalisi.value= "";
11 }
12
13 /* Seconda funzione: ANALISI PORTE SWITCH. Funzione per pulire il campo in cui l'utente inserisce il datapath per analizzare le porte */
14 function resetDatapathPort(){
15     var datapathPort = document.getElementById("datapath1");
16     datapathPort.value= "";
17 }
18
19
20
21
22
23
24 /***** FILE FLOW SWITCH.HTML *****/
25
26 /* Prima funzione: FLOW STATS. Funzione per pulire il campo in cui l'utente inserisce il datapath per un certo flow */
27 function resetFormFlow(){
28     var datapathFlow = document.getElementById("datapath");
29     datapathFlow.value= "";
30 }
31
32 /* Seconda funzione: FLOW AGGREGATE STATS. Funzione per pulire il campo in cui l'utente inserisce il datapath per i flow aggregati */
33 function resetFormFlowAggr(){
34     var datapathAggregate = document.getElementById("datapath3");
35     datapathAggregate.value= "";
36 }
```

Illustrazione 43: funzioni per poter resettare i form in cui va inserito il datapath

Dopo aver analizzato i file relativi alle pagine .html e alle loro specifiche funzioni, rimane da valutare l'utilizzo di tre file .js. Il primo di questi è il file *reset.js*. Come si intuisce dal nome, questo file contiene al suo interno funzioni JavaScript che, una volta chiamate, permettono di “pulire” i form al cui interno l'utente ha inserito il datapath dello switch da selezionare. Ogni form ha il suo ID che viene utilizzato in queste funzioni per prendere il riferimento al form e riuscire a collegarsi a lui. A questo punto basterà settare il valore di quel form con una stringa vuota. Queste funzioni vengono richiamate nel momento in cui l'utente preme il bottone per ottenere certi risultati all'interno delle pagine .html. Ho realizzato queste semplici funzioni per comodità e per mantenere un certo ordine all'interno della web app, perché altrimenti l'utente avrebbe dovuto ogni volta pulire il form da ciò che aveva deciso di inserire.

6.11 Analisi del file restRyu.js

Il secondo dei file JavaScript da analizzare brevemente è *restRyu.js*. E' un file al cui interno sono state inserite tutte le stringhe delle chiamate REST utilizzate all'interno della web app con relativa funzione getter per poter restituire la stringa adatta alla situazione. Ho deciso di realizzare questo file per rendere il codice più pulito e adatto a cambiamenti futuri. Se infatti il controller Ryu dovesse subire delle modifiche in futuro tali per cui anche le chiamate qui utilizzate andassero modificate, sarebbe troppo oneroso ricercare all'interno del codice tutte le ricorrenze e modificarle. In questo modo invece basterà modificarle una volta all'interno di questo file per poter avere la modifica riversata su tutta l'applicazione. Grazie a questo file poi sarà possibile aggiungere in futuro ulteriori chiamate REST e relativi getter per avere più funzionalità disponibili.

```
1
2  /* Chiamate rest generali */
3  var protocol = "http://";
4  var localhost = ":8080/";
5
6  /* Chiamate rest per il primo file switch.html */
7  var numberSwitches = "stats/switches";
8  var statsSwitch = "stats/desc/";
9  var statsPortsSwitch = "stats/port/";
10
11 /* Chiamate rest per il secondo file flowSwitch.html */
12 var statsFlow = "stats/flow/";
13 var statsFlowAggregate = "stats/aggregateflow/";
14
15 /* Chiamate rest per il terzo file addRule.html */
16 var addRuleFlow = "stats/flowentry/add";
17
18 /* Chiamate rest per il quarto file controller.html */
19 var statsRole = "stats/role/";
20 var statsEditRole = "stats/role";
21
22
23 function getProtocol() {
24     return protocol;
25 }
26
27 function getLocalhost() {
28     return localhost;
29 }
30
31 function getNumberSwitches() {
32     return numberSwitches;
33 }
34
35 function getStatsSwitch() {
36     return statsSwitch;
37 }
```

Illustrazione 44: chiamate REST salvate all'interno di variabili e rese disponibili con getter

La realizzazione di questo file è stata pensata anche per poter permettere la diffusione di più controller all'interno della stessa web app. Se l'utente volesse aggiungere ad esempio un controller POX, potrebbe realizzare un file del tutto simile a quello appena analizzato utilizzando però la propria sintassi. L'Illustrazione 44 mostra poi che, oltre alle chiamate tipiche del controller Ryu, sono state inserite anche quelle relative al protocollo da utilizzare e alla porta del localhost, perché in un futuro potrebbero variare e anche in questo caso sarebbe oneroso andare alla ricerca di tutte le occorrenze nel codice per sostituirle.

6.12 Analisi del file functionRyu.js

L'ultimo file che rimane da analizzare è quello relativo alle funzioni che vengono utilizzate con il controller Ryu. All'interno delle pagine .html, nel momento in cui un bottone viene premuto, bisogna richiamare certe funzioni. Ho deciso che non sarebbe stato troppo elegante lasciare le chiamate a funzione dirette a quelle di Ryu, perché nel caso si utilizzasse un controller diverso da quello da me scelto bisognerebbe andare a modificare manualmente tutte le chiamate. Ho optato dunque per inserire all'interno della funzione che viene chiamata alla pressione del bottone, un'ulteriore funzione che è quella che mi rappresenta il raggiungimento del risultato voluto dall'utente. Così facendo basterà in futuro realizzare un file simile a *functionRyu.js*, che utilizzi gli stessi nomi delle funzioni, ma variare il contenuto di ogni singola funzione inserendo un riferimento a quelle del proprio controller.

```
6  /* Funzioni per il file switch.html */
7
8  function getFunctionNumberSwitch() {
9      numeroSwitch();
10 }
11
12 function getFunctionDetailsSwitch() {
13     detailsSwitch();
14 }
15
16 function getFunctionResetFormSwitch() {
17     resetFormSwitch();
18 }
19
20 function getFunctionPortSwitch() {
21     portsSwitch();
22 }
23
24 function getFunctionResetDatapathPort() {
25     resetDatapathPort();
26 }
27
28
29 /* Funzioni per il file flowSwitch.html */
30
31 function getFunctionFlow() {
32     flow();
33 }
34
35 function getFunctionResetFormFlow() {
36     resetFormFlow();
37 }
38
39 function getFunctionAggregateFlow() {
40     flowAggregate();
41 }
42
43 function getFunctionResetFormAggregateFlow() {
44     resetFormFlowAggr();
45 }
```

Illustrazione 45: funzioni che vengono chiamate alla pressione di un bottone

Capitolo 7 - Conclusioni

Siamo dunque arrivati alla conclusione del progetto relativo all'interfaccia web che permette ad un utente non ancora esperto conoscitore delle reti SDN di poter iniziare a familiarizzare con questa tecnologia, che nei prossimi anni dovrebbe prendere sempre più piede fino a soppiantare le attuali architetture di rete. Partendo da una rete virtualizzata, sono riuscito a realizzare una web application che sfrutta le chiamate REST del controller Ryu per poter in qualche modo gestire e visualizzare dati relativi alla rete realizzata.

7.1 Sviluppi futuri

Come ampiamente discusso nel corso di questa Tesi, uno degli obiettivi principali era quello di poter realizzare una web app che fosse, in un futuro prossimo, utilizzabile e modificabile da altri ragazzi per poter implementare via via nuove funzionalità. Io ho scelto il controller Ryu e implementato diverse chiamate REST che il controller metteva a disposizione, però sarebbe utile vedere come lavorano in parallelo due o più controller che gestiscono simultaneamente la stessa rete SDN e gli stessi switch. Per questo motivo ho cercato, dove possibile, di disaccoppiare il codice JavaScript dalle pagine html, per non far risultare il lavoro troppo oneroso qualora si volessero inserire nuove funzionalità o addirittura controller diversi da Ryu. Le funzionalità implementate in questo progetto sono quasi tutte relative alle statistiche degli switch e dei controller, pertanto si potrebbe ampliare la scelta e le operazioni concesse all'utente come ad esempio modificare certi flussi oppure far scegliere all'utente se permettere la trasmissione di un flusso oppure no.

BIBLIOGRAFIA

- [1] https://en.wikipedia.org/wiki/Software-defined_networking
- [2] https://www.cerias.purdue.edu/news_and_events/events/security_seminar/
- [3] <http://www.networxsecurity.org>
- [4] https://www.citrix.com/content/dam/citrix/en_us/documents/oth/sdn-101-an-introduction-to-software-defined-networking-it.pdf
- [5] <https://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/>
- [6] <http://home.deib.polimi.it/cesana/teaching/FIR2016-2017/lezioni/SDN.pdf>
- [7] <http://archive.openflow.org>
- [8] <http://openvswitch.org>
- [9] <http://mininet.org/download/>
- [10] <http://mininet.org>
- [11] <https://www.virtualbox.org>
- [12] <https://www.sdxcentral.com/sdn/definitions/sdn-controllers>
- [13] <http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network>
- [14] https://it.wikipedia.org/wiki/Web_Server_Gateway_Interface
- [15] <https://www.getpostman.com>
- [16] <http://www.brocade.com/content/html/en/configuration-guide/fastiron-08040-sdnguide/GUID-067D5654-2300-41CD-BFF8-16712AC071C2.html>
- [17] <https://www.mamp.info/en/>
- [18] http://ryu.readthedocs.io/en/latest/app/ofctl_rest.html