Introduzione a Open Defined Network, protocollo OpenFlow e tool mininet

Marco D'Agostino – M63000762 – 13 Giugno 2019

Mininet

Brevemente, si tratta di un tool per realizzare virtualmente topologie di reti per scopi di testing. Per l'utilizzo è necessario effettuare diverse azioni preliminari

Installazione

Bisogna scaricare una *VM mininet image* attraverso il comando

```
wget
https://github.com/mininet/mininet/releases/download/2.2.2/mininet-
2.2.2-170321-ubuntu-14.04.4-server-amd64.zip
```

e scompattare l'archivio con il comando `unzip`. Nella stessa cartella della VM image creare il file *start.sh* contenente il seguente codice

```
#!/bin/sh
sudo qemu-system-x86_64 -machine accel=kvm -m 2048 mininet-vm-
x86_64.vmdk -net nic,model=virtio -net
user,net=192.168.101.0/24,hostfwd=tcp::8022-:22
```

ed eseguire l'aggiornamento della macchina

```
sudo apt update && sudo apt upgrade && sudo apt-get clean && sudo
apt-get autoremove && sudo apt-get autoclean
```

poi riavviare.

Per collegarsi tramite *ssh* bisogna invocare

```
ssh -Y -p 8022 mininet@localhost
```

usando come username e password la stringa *mininet*.

Per far funzionare correttamente *wireshark* bisogna installare il plugin per filtrare correttamente i pacchetti openFlow. Eseguire

```
wget http://www.projectfloodlight.org/openflow.lua
/usrlib/x86*/wireshark/plugins/
```

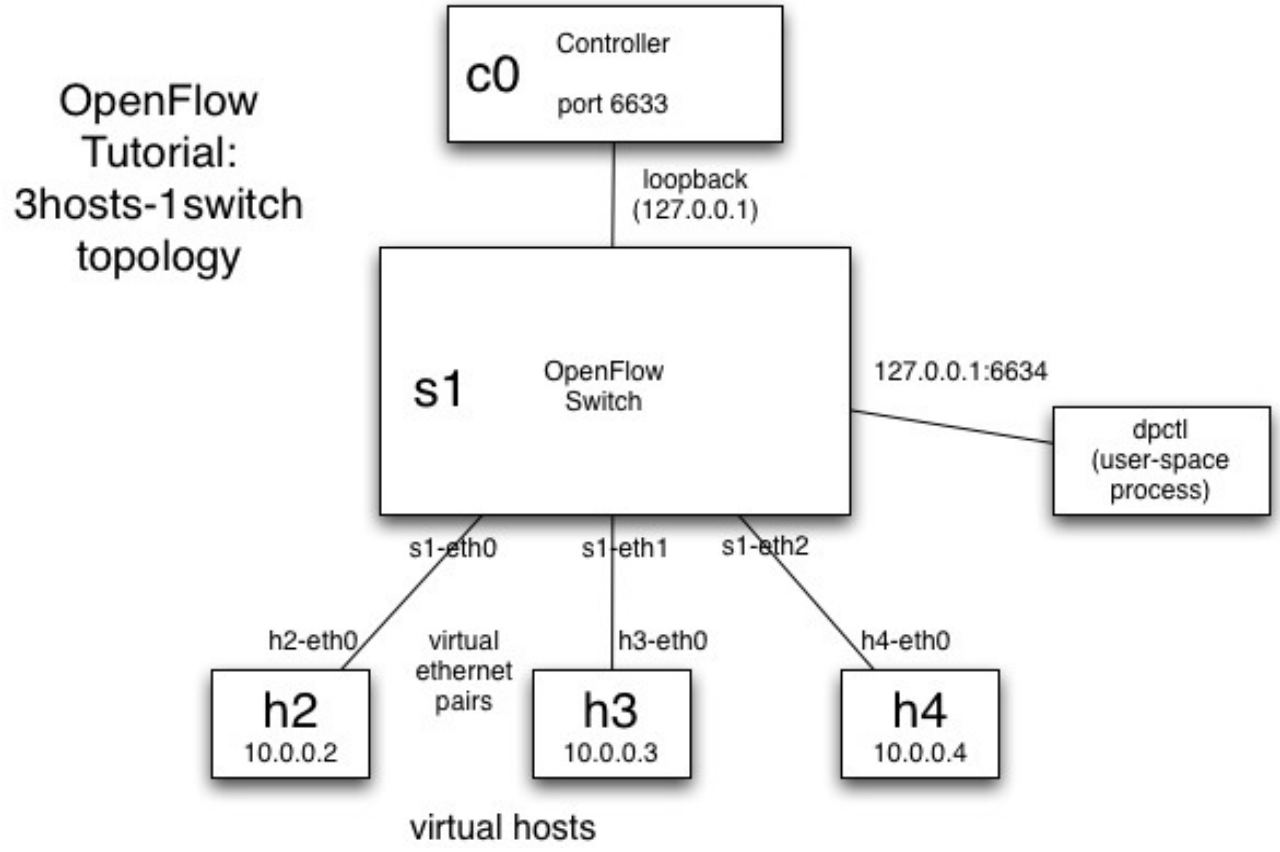
Wireshark tende a far saturare, dopo alcuni minuti di utilizzo, la memoria della VM. Se si incontrano degli errori basta svuotare la cartella */tmp/*.

Topologia utilizzata

Attraverso il comando

```
sudo mn --topo single,3 --mac --switch ovsk --controller default
```

realizziamo una topologia a stella con un singolo switch e tre host, come in figura



- *--topo* permette di specificare la topologia di rete
  - *single* per un singolo switch
  - *3* per indicare il numero di host
- *--mac* serve per fare in modo che gli indirizzi MAC delle schede di rete siano sempre gli stessi e facilmente leggibili, invece che casuali e di difficile memorizzazione.
- *--switch ovsk* indica che si tratta di un Open vSwitch in *kernel-mode*, che offre prestazioni nettamente superiori ad uno eseguito in *user-mode*.
- *--controller default* per utilizzare un controller non personalizzatoProtocollo di comunicazione OpenFlow

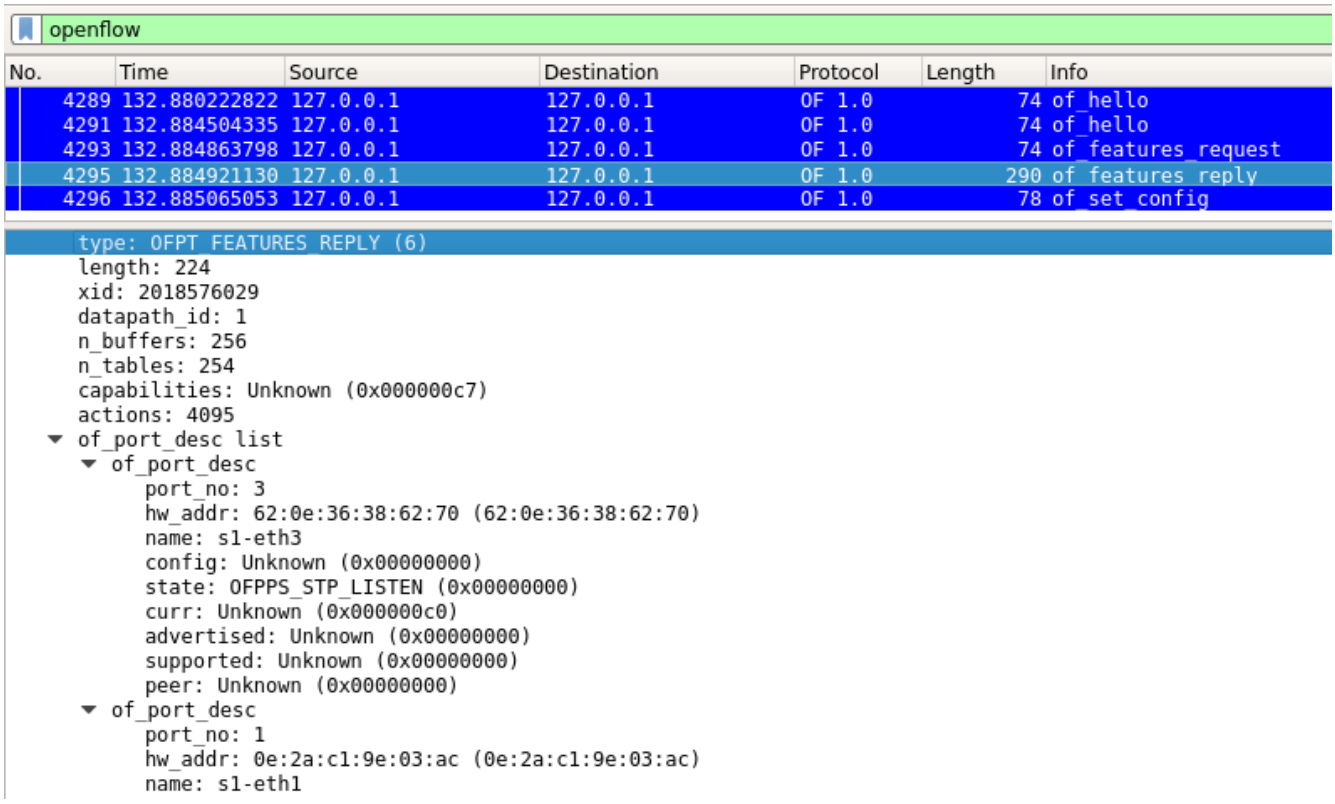
Prime interazioni tra controller e OpenFlow Switch

Per osservare le interazioni tra controller e switch apriamo wireshark in background

```
sudo wireshark &
```

inseriamo il filtro *of and not (of10.echo_request.type or of10.echo_reply.type)* e selezioniamo l'interfaccia di *loopback*. Poi avviamo mininet con il controller di default. I messaggi OpenFlow (of) scambiati sono

1	OFPT_HELLO	Controller → Switch	Il controller informa lo switch della versione di of utilizzata
2	OFPT_HELLO	Switch → Controller	Lo switch risponde con la versione supportata
3	OFPT_FEATURES_REQUEST	Controller → Switch	Il controller chiede le porte disponibili
4	OFPT_FEATURES_REPLY	Switch → Controller	Lo switch invia lo stato delle sue porte, vedi immagine
5	OFPC_FRAG_NORMAL (of_set_config)	Controller → Switch	Il controller chiede l'attributo *expiration* delle *flow entry*



Inserimento di flow entry nella tabella

Vogliamo che il nostro ofSwitch si comporti come un learning switch. Per fare ciò il controller deve inserire correttamente entry nella sua *flow table*. Fortunatamente questo è il comportamento standard del nostro controller.

Per osservare questa fase eseguiamo, dopo aver avviato la topologia, il comando

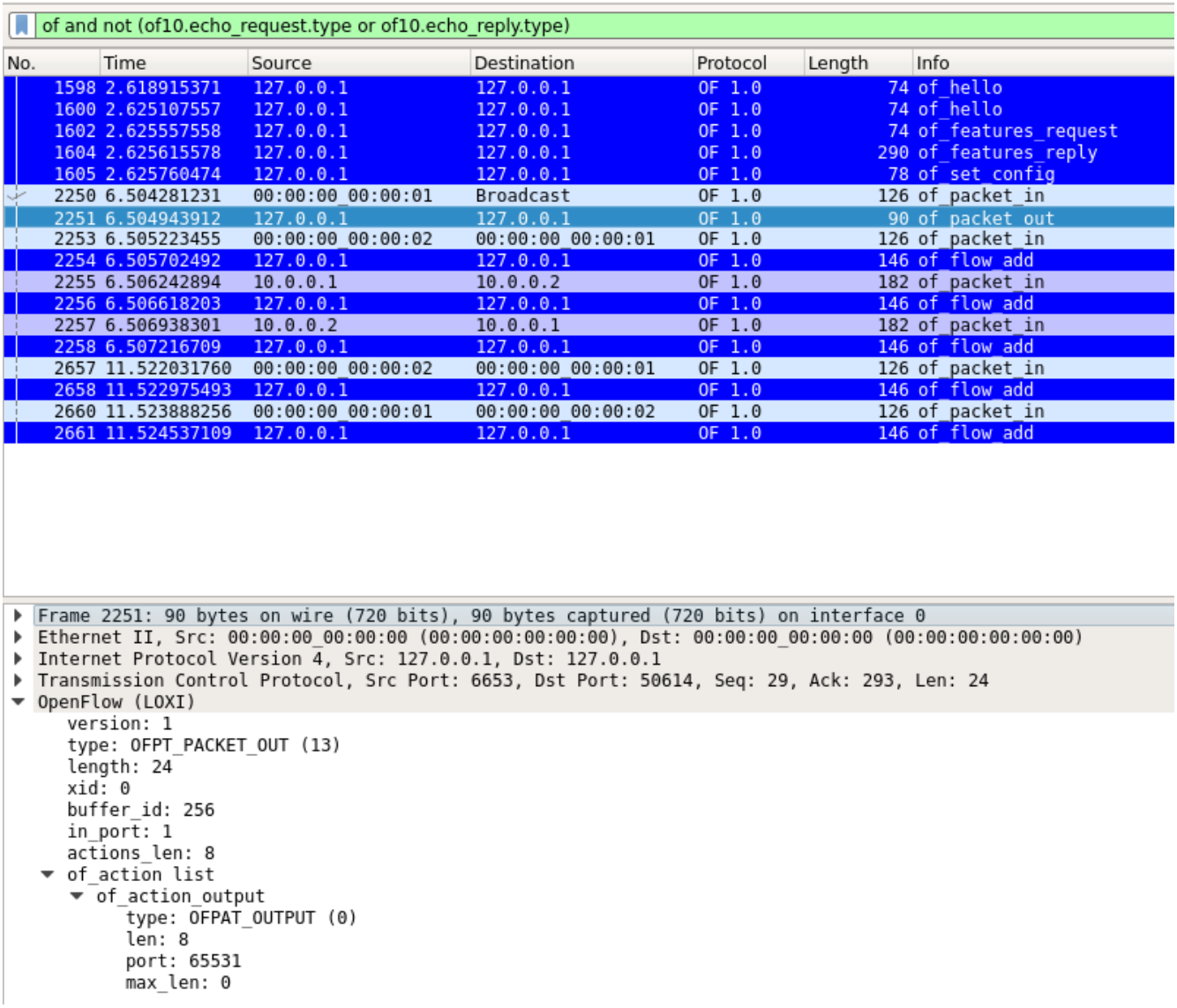
```
h1 ping -c1 h2
```

e facciamo qualche considerazione su ciò che avviene. Prima di tutto si utilizza prima il protocollo ARP per individuare il mac address di h2, poi avviene l'interazione con il protocollo ICMP, e poi di nuovo ARP per scoprire l'indirizzo di h1. Lo switch è di tipo L2, quindi le *arp request* devono avere un effetto su di esso (non saranno solo queste ultime ad avere effetto).

no.	**Descrizione**	**Protocollo**
1	h1 invia una *ARP request* allo switch con indirizzo di broadcast	ARP request
2	Lo switch non sa cosa fare perchè la richiesta non corrisponde a nessuna entry nella sua f*low table* e quindi chiede al controller. Intanto associa il *mac address* di h1 alla porta no. 1!	OFPT_PACKET_IN (OFPR_NO_MATCH)
3	Il controller invia una risposta contentente l'azione che lo switch deve eseguire (ovvero OFPAT_OUTPUT, inoltra in broadcast)	OFPT_PACKET_OUT
4	Lo switch inoltra il frame a tutti	ARP request
5	h2 risponde e quindi invia un frame ad h1	ARP reply
6	Quindi lo switch si vede arrivare un frame nella porta no. 2 dall'host h2, quindi inserisce una nuova associazione [porta no. - mac address] e chiede al controller cosa fare perchè la sua flow table è ancora vuota.	OFPT_PACKET_IN (OFPR_NO_MATCH)
7	Il controller dice allo switch di inserire una nuova entry nella sua flow table	OFPT_FLOW_MOD
8	Lo switch inserisce una nuova flow entry e poi inoltra il pacchetto all'host corretto secondo la nuova regola.	ARP reply
9	L'host h1 invia ah h2 un messaggio di ping	ICMP echo request
10	Lo switch non sa cosa fare di questo frame perchè non riguarda una arp request come quello di prima. Viene chiesto cosa fare al controller	OFPT_PACKET_IN (OFPR_NO_MATCH)
11	Il controller fa aggiungere una nuoova entry nella tabella dello switch	OFPT_FLOW_MOD
12	Lo switch aggiunge la regola e inoltra il frame contenente la richiesta ICMP	ICMP echo request
13	L'host h2 risponde secondo il protocollo ICMP	ICMP echo reply
14	Lo switch si vede arrivare un frame contentente la risposta ICMP che non corrisponde a nessuna entry della sua tabella. Quindi invia un messaggio al controller	OFPT_PACKET_IN (OFPR_NO_MATCH)
15	Ancora una volta il controller fa aggiungere una nuova entry alla tabella, questa volta corrispondente alla risposta ICMP.	OFPT_FLOW_MOD
16	Quindi lo switch aggiunge la rule e inoltra il frame	ICMP echo reply
*	L'host h2 non conosce il mac address di h1, quindi inizia un	

	nuovo botta e risposta seguendo il protocollo ARP dove vengono aggiunte due nuove entry della flow table (ARP request da h2 → h1 e ARP reply da h1 → h2)	

In totale, un semplice ping tra due host produce 5 f*low entry* (la prima ARP request è in broadcast e non innesca nessuna aggiunta alla *flow table*).



Con wireshark si può osservare che un messaggio di tipo **of_packet_in** porta con se non solo una sorta di header (contenente ad esempio t*ype: OFPT_PACKET_IN (10)* e *in_port: 1*) ma anche tutto l'ethernet packet che lo ha causato! Questa cosa sarà molto importante quando si dovrà programmare il controller.

Un messaggio di tipo **of_packet_out** viene inviato dal controller per dire allo switch l'operazione che deve eseguire. Si può osservare che il messaggio è costituito da una lista di azioni.

Alterntivamente, il controller può trasmettere un messaggio **of_flow_add** per aggiungere una flow entry. Esso è costituito da una lista di azioni identica al caso precedente e una sezione of_match che descrive la nuova *flow entry*.

Possiamo osservare le nuove *flow entry* eseguendo, sempre nella VM ma al di fuori della shell di mininet, il comando

```
sudo ovs-ofctl dump-flows s1
```

Utilizzo di un controller remoto

Il controller remoto ci serve per poi utilizzarne uno programmato da noi.

Questa volta bisogna avviare la topologia con il comando

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

ma solo dopo aver avviato il controller, altrimenti non funziona.

In un'altra finestra di terminale, dalla home directory spostiamoci nella cartella pox avviamo il controller POX attraverso il comando

```
./pox.py log.level --DEBUG misc.of_tutorial
```

Carichiamo la topologia ed eseguiamo un test con iperf. Osserveremo che le prestazioni sono peggiori del controller pecedente. Questo perchè, aprendo il file *./pox/misc/of_tutorial.py,* il controller è stato programmato per funzionare come un hub. Per utilizzarlo come uno switch basterebbe commentare la linea *`self.act_like_hub()`* e decommentare *`self.act_like_hub()`*, ma è necessario implementare manualmente quest'ultima funzione.

Controller implementato come hub con POX

Per scrivere il codice del *learning switch* è meglio partire dalla comprensione di un codice più semplice.

```
def act_like_hub (self, packet, packet_in):
    """
    Implement hub-like behavior -- send all packets to all ports besides
    the input port.
    """

    # We want to output to all ports -- we do that using the special
    # OFPP_ALL port as the output port.
    self.resend_packet(packet_in, of.OFPP_ALL)
```

Il parametro packet dovrebbe essere l'intero messaggio del protocollo OpenFlow mentre il parametro *packet_in* è la parte che potremmo chiamare "header" del messaggio. Ad occhio, viene chiesto allo switch di inoltrare il frame ricevuto su tutte le porte, in modo da assumere il comportamento di uno switch.

```
def resend_packet (self, packet_in, out_port):
    """
    Instructs the switch to resend a packet that it had sent to us.
    "packet_in" is the ofp_packet_in object the switch had sent to the
    controller due to a table-miss.
    """
    msg = of.ofp_packet_out()
    msg.data = packet_in

    # Add an action to send to the specified port
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)

    # Send message to switch
    self.connection.send(msg)
```

Per rispondere alla richiesta di uno switch indicandogli l'azione da compiere, il metodo standard è quello di preparare un messaggio con `ofp_packet_out()`, si assegna all'attributo data il valore di *packet_in*. Infine si indica l'azione, che è un inoltro su tutte le porte, quindi `ofp_action_output()`. I nomi delle funzioni sono identici alle label viste su wireshark!

Ora che il messaggio è pronto, esso può essere inviato allo switch attraverso una chiamata alla funzione `self.connection(send)`.

Controller implementato come banale learning switch

La chiave per realizzare questo passo è quella di mantenere un dizionario che associa ad un indirizzo mac una porta. La struttura adibita a ciò ha, nel codice, l'identificativo *mac_to_port* e viene inizializzata nella funzione `__init__` (costruttore del controller). Di seguito il codice che implementa lo switch

```
  def act_like_switch (self, packet, packet_in):
    """
    Implement switch-like behavior.
    """

    # Learn the port for the source MAC
    switchInputPort = packet_in.in_port

    #self.mac_to_port ... <add or update entry>
    self.mac_to_port[packet.src] = switchInputPort

    if packet.dst in self.mac_to_port:
      # Send packet out the associated port
      self.resend_packet(packet_in, self.mac_to_port[packet.dst])
```

```
    else:
      # Flood the packet out everything but the input port
      self.resend_packet(packet_in, of.OFPP_ALL)
```

Ogni volta che arriva un frame su una porta, viene aggiornata la coppia [*chiave, valore]* = *[indirizzo mac, numero porta]* e se la destinazione è nota (`if packet.dst in self.mac_to_port`) si invia allo switch il comando di inoltrare il pacchetto sulla porta nota, altrimenti si inoltra il frame su tutte le porte.

Questo comportamento genera tanti messaggi *of_packet_out* quanto tanti messaggi *of_packet_in*. Nessun messaggio *of_flow_add*/*of_flow_mod* viene generato e quindi lo switch rimane sempre “stupido” perché aspetta sempre la decisione del controller.

Controller implementato come un vero learning switch

Il controller non deve essere sempre chiamato in causa e deve inserire flow entry all’interno della tabella dello switch in modo tale che quest’ultimo possa successivamente ricordre le scelte fatte dal primo. Il codice da implementare necessita, questa volta, di un po’ di attenzione

```
def act_like_switch (self, packet, packet_in):
    """
    Implement switch-like behavior.
    """

    # Learn the port for the source MAC
    switchInputPort = packet_in.in_port

    self.mac_to_port[packet.src] = switchInputPort

    #if the port associated with the destination MAC of the packet is known:
    if packet.dst in self.mac_to_port:
      log.debug("Installing flow...")
      log.debug( \
          str(packet_in.in_port)  \
          +"/"+str(packet.src)    \
          +"/"+str(packet.dst)    \
          +" => " \
          +str(self.mac_to_port[packet.dst])+ \
          "\n")

      msg = of.ofp_flow_mod()

      # Set fields to match received packet
      msg.match = of.ofp_match.from_packet(packet, packet_in.in_port)

      #< Set other fields of flow_mod (timeouts? buffer_id?) >
      msg.buffer_id = packet_in.buffer_id

      #< Add an output action, and send -- similar to resend_packet() >
      msg.actions.append(of.ofp_action_output( \
          port = self.mac_to_port[packet.dst]))
      self.connection.send(msg)
```

```
    else:
      # Flood the packet out everything but the input port
      self.resend_packet(packet_in, of.OFPP_ALL)
```

Il messaggio di tipo *of_flow_add* viene preparato con una chiamata alla funzione *`ofp_flow_mod()`*. Come visto in wireshark, un messaggio del genere include il campo of_match contenente i dati della flow entry. Questi sono derivati dal “payload” del flow message attraverso la funzione `ofp_match.from_packet()`. È molto importante anche far corrispondere il *buffer_id* dei due messaggi. Infine si aggiunge anche l’azione di inoltro del frame verso la porta giusta (stiamo sempre nel ramo if). Se non si conosce la porta di destinazione associata al mac address destinazione si inoltra su tutte le porte.

Il comportamento di questo learning switch è pressocchè lo stesso di quello osservato nella precedente sessione di wireshark.