

Relatório de E.T.

Relatório nº3 - *Software-Defined Networking and OpenFlow*

Curso: METI

Turno: 3ª feira 08:00 > 09:30

Grupo: Bancada 8

Trabalho realizado por:

Luís Pereira, nº77984
Ruben Condesso, nº 81969

Índice

1 - Introdução	2
2 - Laboratório 4	3
2.1 - Ponto 3: <i>Testing VM</i>	3
2.2 - Ponto 4: Testing Mininet	3, 4
2.3 - Ponto 5: Test the network	5, 6, 7
3 - Laboratório 5	7
3.1 - Ponto 4: <i>Testing Mininet</i>	8
3.2 - Ponto 5: <i>Analysing hub behavior</i>	8, 9
3.3 - Ponto 7: <i>Pox of tutorial code for simple hub</i>	9, 10
3.4 - Ponto 8: <i>Implementing a learning switch</i>	10, 11

1. Introdução

Este relatório aborda o tema "*Software-Defined Networking and OpenFlow*", e é referente aos laboratórios nº4 e nº5, *Introduction to Mininet* e *Introduction to SDN controllers*, respetivamente. Desta forma, o relatório está dividido em duas partes: a primeira diz respeito ao laboratório nº4 e a segunda ao laboratório nº5.

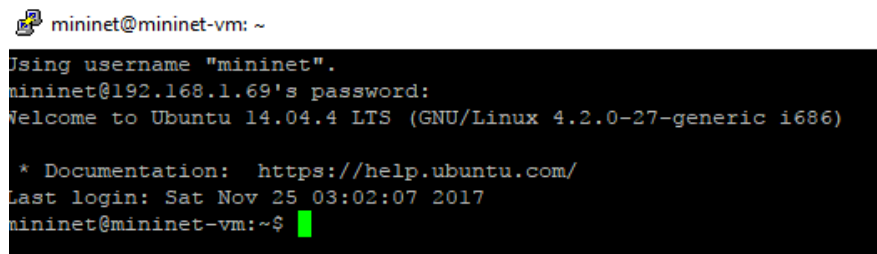
A junção dos dois guias tem como intuito criar um ambiente de teste para implementar e testar *SDN network*, juntamente com o protocolo *OpenFlow*. *SDN* é uma tecnologia que aborda a computação em *cloud*, facilitando a gestão da rede, e permite a configuração da rede com uma programação mais eficiente, melhorando a sua performance e monitorização. Um dos princípios da *SDN* é que a programação das tabelas de encaminhamento são feitas por um controlador (servidor centralizado), onde concretamente, neste relatório será demonstrado o uso e desenvolvimento de um simples módulo para um *POX controller*. O *OpenFlow* não é mais que um protocolo de comunicação emergente que permite a um servidor de *software* determinar o caminho do encaminhamento de pacotes, que deverá seguir uma rede de *switches*.

2. Laboratório nº4

Este laboratório tem como objetivo fazer uma introdução ao *mininet*. Este permite-nos criar uma rede virtual realística, onde é possível correr um código com um núcleo real, numa única máquina virtual, com um único comando, em poucos segundos, daí a sua grande utilidade. O ponto 3 e o ponto 4 do guia de laboratório têm apenas a finalidade de testar o funcionamento do sistema *mininet* e da VM, com o auxílio de alguns comandos.

Ponto 3 - *Testing VM*:

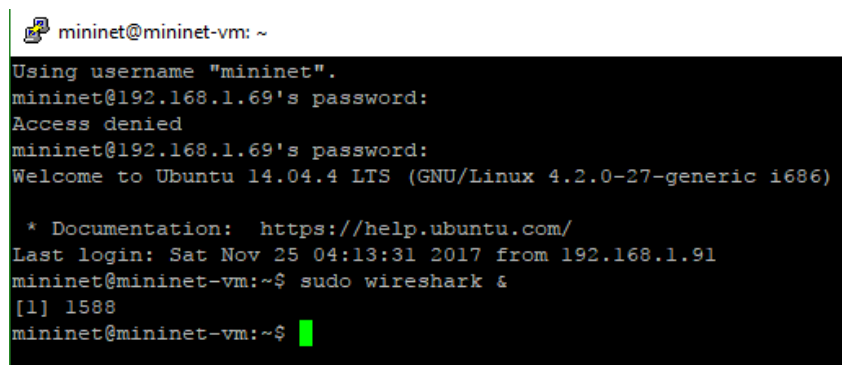
A partir do *print*, ilustrado em baixo, podemos ver que conseguimos fazer *login* no sistema *mininet*, pelo *X terminal*, a partir do *host system*:



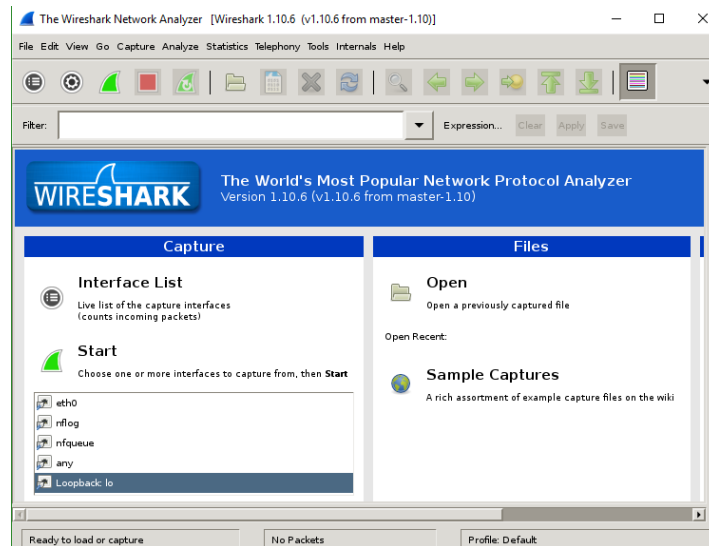
```
mininet@mininet-vm: ~  
Using username "mininet".  
mininet@192.168.1.69's password:  
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic i686)  
  
 * Documentation:  https://help.ubuntu.com/  
Last login: Sat Nov 25 03:02:07 2017  
mininet@mininet-vm:~$
```

Ponto 4 - *Testing mininet*:

No ponto 4.1 - *start wireshark* - o objetivo é abrir o *wireshark* a partir do *host system*, pelo *X terminal*:



```
mininet@mininet-vm: ~  
Using username "mininet".  
mininet@192.168.1.69's password:  
Access denied  
mininet@192.168.1.69's password:  
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic i686)  
  
 * Documentation:  https://help.ubuntu.com/  
Last login: Sat Nov 25 04:13:31 2017 from 192.168.1.91  
mininet@mininet-vm:~$ sudo wireshark &  
[1] 1588  
mininet@mininet-vm:~$
```



No ponto 4.2 - *interact with hosts and switches* - ao fazer o comando **sudo mn**, iremos criar uma topologia simples e entrar na CLI:

```
mininet@mininet-vm: ~
[1] 1588
mininet@mininet-vm:~$ sudo mn
*** Creating network
*** Adding controller
*** 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:
```

De seguida, ao fazer os comandos: **nodes**; **net**; **dump**; iremos visualizar os *nodes*, os *links* e a *dump information* de todos os nós, referentes à topologia em causa:

```
*** Starting CLI:
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=1731>
<Host h2: h2-eth0:10.0.0.2 pid=1733>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=1738>
<Controller c0: 127.0.0.1:6653 pid=1724>
mininet>
```

Por fim, no ponto 4.3 - *test conectivity between hosts* - fizemos um *test ping*, entre o *host0* e o *host1*, com sucesso:

```
mininet> h1 ping -c 1 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=8.36 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.369/8.369/8.369/0.000 ms
```

Ponto 5 - *test the network*:

No ponto 5.1 - *network instantiation* - é nos proposto criar uma rede, que inclui 3 *hosts* e um *switch* (comando - **sudo mn --topo single,3 --mac --switch ovsk -- controller remote**). Feito isto, adiante no ponto 5.3 - *ping test* - é nos dito para fazer um *ping test*, entre o *host1* e o *host2* (comando - **h1 ping -c3 h2**), onde não obtivemos qualquer resposta.

Isto tem uma razão de ser: no ponto 5.2 - *ovs-ofctl example usage* - um comando útil que nos é dado é o seguinte: **ovs-ofctl dump-flows s1**. Verificamos que a *flow table* está vazia, daí não obtemos resposta do *ping test*, além do mais, não existe nenhum controlador ligado ao *switch*, então este não sabe para onde enviar o tráfego recebido. Fica assim respondido à primeira pergunta do ponto 5.3, onde nos é perguntado a razão pela qual não obtivemos resposta no *ping*.

A seguir, executamos os seguintes comandos:

- **ovs-ofctl add-flow s1 in_port=1,actions=output:2 ;**
- **ovs-ofctl add-flow s1 in_port=2,actions=output:1;**

Tal, irá gerar *flows* que entram na porta1 e saem na porta2 e vice-versa.

Para verificar a *flow table*, fizemos o seguinte comando: **sudo ovs-ofctl dump-flows s1**; originando o seguinte *output*:

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=16.079s, table=0, n_packets=0, n_bytes=0, idle_age=16, in
  port=1 actions=output:2
  cookie=0x0, duration=6.21s, table=0, n_packets=0, n_bytes=0, idle_age=6, in_po
  rt=2 actions=output:1
```

Com o auxílio do *print* ilustrado acima, podemos ver a tabela completa do *flow*. Fazendo novamente o *ping*, entre o *host1* e o *host2*, podemos verificar que já temos conectividade entre ambos, pelo que o *ping* ocorre com sucesso como podemos concluir pela imagem abaixo:

```
mininet> h1 ping -c3 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=0.496 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.053 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.055 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.053/0.201/0.496/0.208 ms
```

Existe um aumento de 3 pacotes no *n_packets*, mas por outro lado o *n_bytes* não corresponde aos 192bytes supostamente enviados. Fazendo *test pings*, entre os outros *hosts*, verificamos que só temos conexão entre o *host1* e *host2*, sendo que nenhum dos

outros *hosts* se encontra *reachable*. Isto deve-se à falta dos dois comandos ilustrados acima entre os *hosts* em causa, que causou depois a conectividade entre eles.

Fazendo o comando `sudo ovs-ofctl del-flows s1`, verificamos que deixamos de ter conectividade entre o *host1* e *host2*, visto que terminamos o *flow* em *s1*.

No ponto 5.4 - *start controller and view startup messages in Wireshark* - fizemos o comando `controller tcp:6653`, que irá começar um simples *controller*, no porto 6653, que irá agir como um *learning switch* sem que tenha que ter qualquer entrada de *flows*. As mensagens que obtivemos foram as seguintes:

1687	146.74590100	127.0.0.1	127.0.0.1	OF 1.0	74 of_hello
1695	146.89516000	127.0.0.1	127.0.0.1	OF 1.0	74 of_hello
1697	146.89692000	127.0.0.1	127.0.0.1	OF 1.0	74 of_hello
1699	146.89697500	127.0.0.1	127.0.0.1	OF 1.0	74 of_features_request
1701	146.89698200	127.0.0.1	127.0.0.1	OF 1.0	78 of_set_config
1703	146.90004400	127.0.0.1	127.0.0.1	OF 1.0	290 of_features_reply
2180	150.96495200	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_request
2182	150.96515100	127.0.0.1	127.0.0.1	OF 1.0	74 of_echo_reply

Podemos visualizar diferentes tipos de mensagens: primeiro temos a troca de mensagens *hello* entre o *controller* e o *switch*, sendo que a primeira é do tipo *controller-switch* e representa o envio do número da versão por parte do *controller* para o *switch*; e de forma análoga, a segunda mensagem é do tipo *switch-controller*, e representa a resposta do *switch* para o *controller*, contendo o número da sua versão.

Segue-se a mensagem *features request*, que é do tipo *controller-switch*, onde é feita a pergunta, por parte do *controller*, que portos se encontram disponíveis. Depois vem a mensagem *features reply*, do tipo *switch-controller*, onde o *switch* responde com a lista de portos, a velocidade de cada porto, e as tabelas e ações que suporta. Por fim, vem as normais mensagens *echo_request* e *echo_reply* trocadas entre os dois, que tem como finalidade manter a conexão entre ambos "viva".

O ponto 5.5 - *view Openflow messages for Ping* - tem como objetivo ver as mensagens geradas, nas respostas dos pacotes. Usando o seguinte filtro no *wireshark*: `of and not (of10.echo_request.type or of10.echo_reply.type)`, seguido de um *ping* entre o *host1* e o *host2*, obtivemos as seguintes mensagens:

442	2.979613000	00:00:00_00:00:01	Broadcast	OF 1.0	128 of_packet_in
443	2.979771000	127.0.0.1	127.0.0.1	OF 1.0	92 of_packet_out
449	2.979963000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.0	128 of_packet_in
450	2.980032000	127.0.0.1	127.0.0.1	OF 1.0	148 of_flow_add
453	2.980345000	10.0.0.1	10.0.0.2	OF 1.0	184 of_packet_in
454	2.980439000	127.0.0.1	127.0.0.1	OF 1.0	148 of_flow_add
457	2.980680000	10.0.0.2	10.0.0.1	OF 1.0	184 of_packet_in
458	2.980754000	127.0.0.1	127.0.0.1	OF 1.0	148 of_flow_add
1167	7.994112000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.0	128 of_packet_in
1168	7.994246000	127.0.0.1	127.0.0.1	OF 1.0	148 of_flow_add
1172	7.994559000	00:00:00_00:00:01	00:00:00_00:00:02	OF 1.0	128 of_packet_in
1173	7.994650000	127.0.0.1	127.0.0.1	OF 1.0	148 of_flow_add

Podemos verificar as mensagens *packet_in*, do tipo *switch-controller*, que significam que foi recebido um pacote que não constava ainda na *flow table* do *switch*, razão pela qual foi enviado para o *controller* (tem como origem *host1* e como destino o *host2*, ou vice-versa sendo que há uma resposta dos dois lados no *ping*). Pode-se ver também a mensagem *packet_out* (tendo como origem o *host1*), do tipo *controller-switch*, que representa o envio de pacote para um ou mais portos. Finalmente, podemos enviar mensagens *flow_add*, do tipo *switch-controller*, que significa que foi adicionado um *flow* específico à *flow table* do *switch*.

No ponto 5.6 - *benchmark controller w/iperf* - é utilizado o comando **iperf** que tem como finalidade medir a velocidade entre computadores, neste caso será feito a medição da velocidade entre os *hosts*. Entre o *host1* e o *host2*, obtivemos a seguinte velocidade:

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['29.6 Gbits/sec', '29.6 Gbits/sec']
mininet>
```

De seguida, testamos entre o *host1* e o *host3*, obtivemos a seguinte velocidade:

```
mininet> iperf h1 h3
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['30.7 Gbits/sec', '30.7 Gbits/sec']
mininet>
```

E por fim, testamos entre o *host2* e o *host3*:

```
mininet> iperf h2 h3
*** Iperf: testing TCP bandwidth between h2 and h3
*** Results: ['30.4 Gbits/sec', '30.4 Gbits/sec']
mininet>
```

Podemos verificar que as três velocidades se aproximam bastante umas das outras, isto deve-se ao facto de estarmos a testar numa topologia bastante simples, o que leva a estas aproximações de resultados.

3. Laboratório 5

Este laboratório tem como finalidade complementar o anterior, com uma introdução aos *SDN controllers*, continuando a usar o *mininet*. Vai ser desenvolvido ao longo do laboratório, um simples módulo para o controlador POX, onde este oferece uma *framework* para comunicar com *SDN switches*, usando o *OpenFlow*.

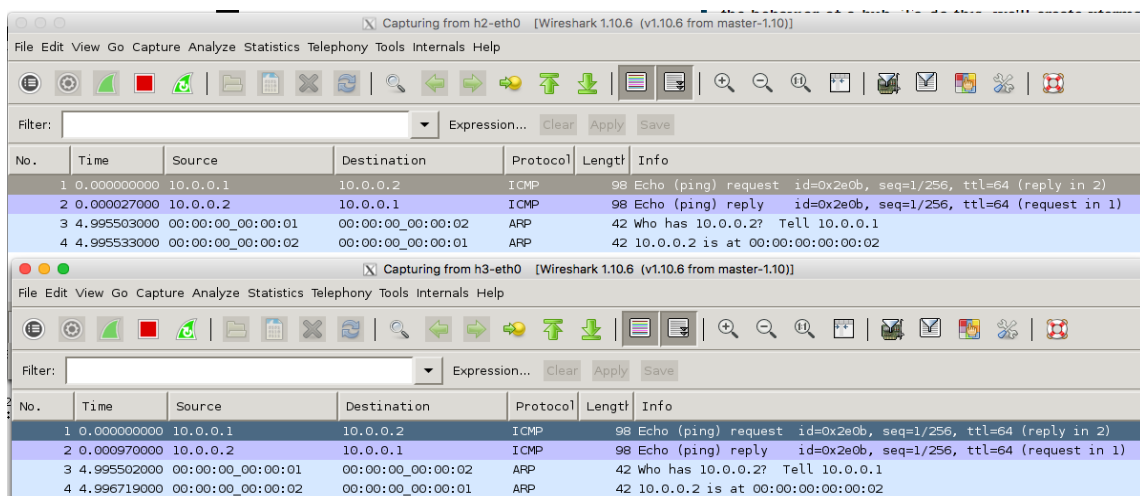
Ponto 4 - *starting mininet* - criamos a mesma topologia usada no laboratório anterior. De seguida, lançamos o exemplo de um *hub* simples, a partir do diretório do POX, com o comando: `./pox.py log.level --DEBUG misc.of_tutorial`. Obtivemos as seguintes respostas do POX:

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_tutorial
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:32:47)
DEBUG:core:Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 2]
```

Ponto 5 - *Analysing Hub Behavior*:

Vamos verificar que cada *host* consegue fazer *ping* para os restantes, e cada um deles irá visualizar o mesmo tráfego (o comportamento do *hub*).

Realizado o comando `ping -c1 10.0.0.2`, podemos verificar, que no *host2* e no *host3*, são recebidas exatamente as mesmas mensagens, como podemos verificar na figura abaixo:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x2e0b, seq=1/256, ttl=64 (reply in 2)
2	0.000027000	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2e0b, seq=1/256, ttl=64 (request in 1)
3	4.995503000	00:00:00_00:00:01	00:00:00_00:00:02	ARP	42	who has 10.0.0.2? Tell 10.0.0.1
4	4.995533000	00:00:00_00:00:02	00:00:00_00:00:01	ARP	42	10.0.0.2 is at 00:00:00:00:00:02

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x2e0b, seq=1/256, ttl=64 (reply in 2)
2	0.000970000	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2e0b, seq=1/256, ttl=64 (request in 1)
3	4.995502000	00:00:00_00:00:01	00:00:00_00:00:02	ARP	42	who has 10.0.0.2? Tell 10.0.0.1
4	4.996719000	00:00:00_00:00:02	00:00:00_00:00:01	ARP	42	10.0.0.2 is at 00:00:00:00:00:02

Da mesma forma, podemos verificar que acontece o mesmo fazendo *ping* para um *host* que não existe como é o caso do 10.0.0.5:

The image shows two Wireshark captures. The top capture is from h2-eth0 and the bottom is from h3-eth0. Both show a sequence of network events: a ping request to 10.0.0.2, a ping reply from 10.0.0.2, and then three ARP requests for 10.0.0.2, 10.0.0.5, and 10.0.0.1. The ARP requests are for the purpose of 'Who has 10.0.0.2?' and 'Who has 10.0.0.5?'. The ping request is from 10.0.0.1 to 10.0.0.2, and the reply is from 10.0.0.2 to 10.0.0.1.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x2e0b, seq=1/256, ttl=64 (reply in 2)
2	0.000027000	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2e0b, seq=1/256, ttl=64 (request in 1)
3	4.995503000	00:00:00_00:00:01	00:00:00_00:00:02	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
4	4.995533000	00:00:00_00:00:02	00:00:00_00:00:01	ARP	42	10.0.0.2 is at 00:00:00:00:00:02
5	1263.6588640	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
6	1264.6452120	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
7	1265.6224320	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1

Enviando 10 pacotes do *host1* para o *host3* apresentamos os resultados:

- Max RTT: 44.542 ms;
- Min RTT: 2.888 ms;
- Avg RTT: 21.187 ms

Enviando 10 pacotes do *host1* para o *host2* apresentamos os resultados:

- Max RTT: 45.115 ms;
- Min RTT: 4.366ms;
- Avg RTT: 22.527 ms

Os resultados são praticamente os mesmos uma vez que o *hub* envia sempre para todos os *hosts* a perguntar quem é o destinatário do pacote e só depois o entrega.

Ponto 7 - *POX of tutorial code for simple hub*:

Iremos explicar o código sobre o método *act_like_hub()* descrito no ficheiro *of_tutorial.py* que se localiza na pasta *pox/pox/misc*.

```
def act_like_hub(self, packet, packet_in):

    self.resend_packet(packet_in, of.OFPP_ALL);
```

Ao usarmos esta função, estaremos a reenviar (chama a função *resend_packet*) os pacotes para todas as portas do *hub* (*of.OFPP_ALL*), excepto a de *input* (de onde vem o pacote),

uma vez que não sabemos quem irá ser o recetor. Iremos criar então uma *action*, para cada porto específico passado como argumento na função, para que estes pacotes sejam enviados para as respetivas portas. É também emitida uma mensagem para o *switch*, em como fez esse reenvio.

Ponto 8 - *implementing a learning switch*:

No ponto 8.1 - *implementing a learning switch without installing table flows entries* - tivemos que editar o ficheiro *of_tutorial*, mais concretamente tivemos que completar a função *act_like_switch()*, que se encontrava nesse ficheiro de modo a respeitar as seguintes regras: para cada pacote que chegasse ao controlador se o endereço de origem do *mac* não fosse conhecido antes, teria de se registado a associação entre esse endereço e o *input port*, no dicionário; e para a mesma situação, teríamos de enviar o pacote para todos os *output ports*; caso o destino do *mac* estivesse já associada a um porto específico, teríamos de enviar apenas para o porto indicado. Posto isto, alteramos o ficheiro *of_tutorial* de modo a seguir as regras impostas, dando origem ao ficheiro: *of_tutorial_8.1*.

Começamos por comentar a chamada da função *act_like_hub()*, na função *handlepacketIn()* e retiramos os comentários na mesma função na chamada de *act_like_hub()*, para quando chegasse um pacote fosse esta a função a ser chamada e não a anterior.

Depois criamos dois *if*'s, um para caso o *destination mac* tivesse já associado a um porto conhecido e outro para caso o *source mac* fosse conhecido. Se entrasse no primeiro *if*: iríamos chamar a função *resend_packet()*, para reenviar o pacote em causa mas apenas para o porto apropriado (*self.mac_to_port[packet.dst]*). Caso entrasse no segundo *if*: iríamos primeiro associar o *source mac* ao *input port* no dicionário (*mac_to_port*), e depois iríamos reenviar o pacote para todos os *output ports* (*self.resend_packet(packet.in, of.OFPP_ALL)*), tal como tínhamos visto na função *act_like_hub()*. Se não entrasse em nenhum dos dois *if*'s, iria para um *else* onde iria reenviar o pacote para todo o lado, excepto o *input port*.

Para verificar o funcionamento da função fizemos os requeridos testes que se encontram no enunciado.

Fazendo um *ping* entre o *host1* e o *host2*, e entre o *host1* e o *host3*, obtivemos os seguintes resultados no POX:

```

mininet@mininet-vm: ~/pox
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:32:47)
DEBUG:core:Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 2]
00:00:00:00:00:01 -> Nao conheco! A adicionar!
ff:ff:ff:ff:ff:ff not known, resend to everybody
00:00:00:00:00:02 -> Nao conheco! A adicionar!
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:02
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:02
ff:ff:ff:ff:ff:ff not known, resend to everybody
00:00:00:00:00:03 -> Nao conheco! A adicionar!
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:03
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:01
Conhecido!Reenvio -> 00:00:00:00:00:03

```

Pode se verificar os *prints* colocados na função *act_like_switch*, primeiro quando recebe o *ping*, de um destino que não conhece e depois adiciona ao dicionário, e posteriormente quando o *destination mac* já está associado a um porto específico e é reenviado para o porto em causa.

No *wireshark*, também foi possível visualizar a troca de mensagens aquando o processo de ligação do POX e *mininet*, e posteriormente quando foi feito um *ping* entre o *host1* o *host2*:

*Loopback: lo [Wireshark 1.10.6 (v1.10.6 from master-1.10)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
591	9.282295000	127.0.0.1	127.0.0.1	OF 1.0	74	of_hello
598	9.595888000	127.0.0.1	127.0.0.1	OF 1.0	74	of_hello
600	9.603307000	127.0.0.1	127.0.0.1	OF 1.0	74	of_hello
602	9.603504000	127.0.0.1	127.0.0.1	OF 1.0	74	of_features_request
604	9.603928000	127.0.0.1	127.0.0.1	OF 1.0	290	of_features_reply
605	9.611197000	127.0.0.1	127.0.0.1	OF 1.0	78	of_set_config
609	9.647966000	127.0.0.1	127.0.0.1	OF 1.0	74	of_barrier_reply
629	12.491577000	00:00:00:00:00:01	Broadcast	OF 1.0	126	of_packet_in
631	12.510854000	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
632	12.511122000	00:00:00:00:00:02	00:00:00:00:00:01	OF 1.0	126	of_packet_in
633	12.512144000	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
634	12.512347000	10.0.0.1	10.0.0.2	OF 1.0	182	of_packet_in
635	12.513003000	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
636	12.513188000	10.0.0.2	10.0.0.1	OF 1.0	182	of_packet_in
637	12.513796000	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
607	9.647835000	127.0.0.1	127.0.0.1	OF 1.0 +	146	of flow delete + of barrier request

Podemos ver as mensagens trocadas entre o *controller* e *switch*, quando foi iniciada a rede. Ou seja, podemos visualizar as mensagens *hello*, *features_request*, *features_reply*, tal como foram descritas no ponto 5.4, do laboratório 4.

E podemos visualizar, igualmente, as mensagens, aquando foi feito o *ping*, concretamente as mensagens *packet_in* e *packet_out*, que também foram referidas no laboratório 4, mas agora no ponto 5.5. Podemos assim verificar as mensagens enviadas para o *controller*,

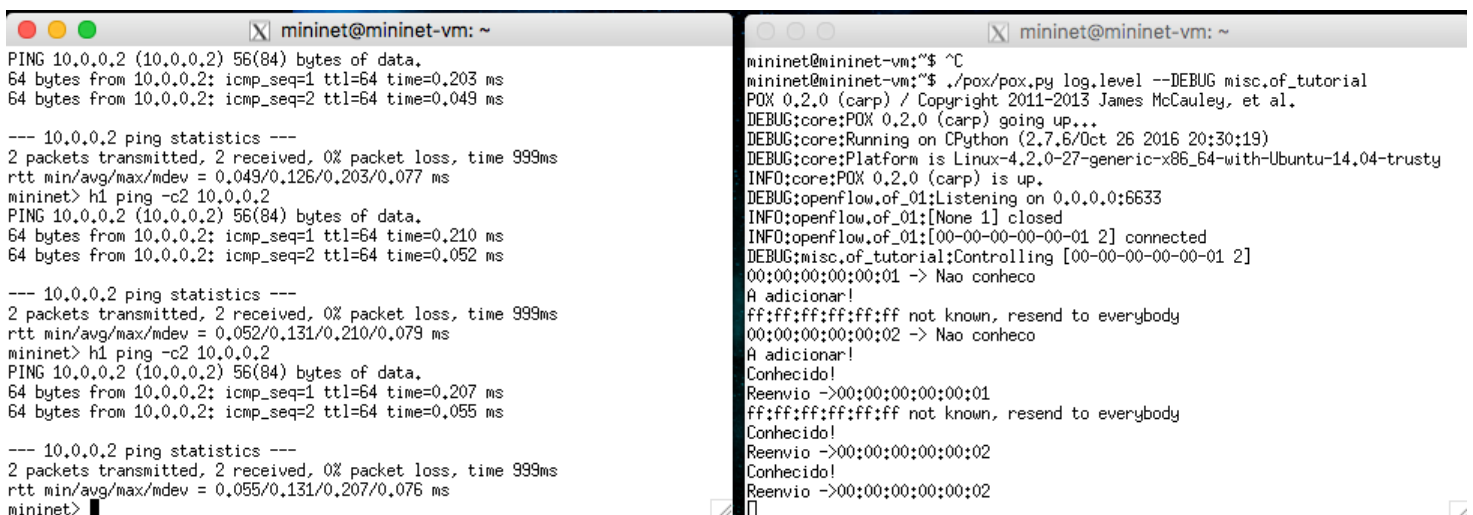
quando é feito o *ping*, e não são visualizadas mensagens *flow_add* porque neste exercício não há a adição de *flows* à *flow table*.

O ponto 8.2 - *implementing a learning switch with table flows entries* - tem como diferenças do exercício anterior, que desta vez não iremos enviar os pacotes todos para o controlador. Iremos criar *action rules*, cada vez que um novo mapeamento é aprendido.

Mantivemos a estrutura de código, na função, relativamente ao exercício anterior, com a diferença no primeiro *if*. Se entrasse nesse *if*, iríamos criar uma *action*, tal como na função *resend_packet()*, só passaríamos o porto apropriado, dando origem à linha de código: *action = of.ofp_action_output(port = self.mac_to_port[packet.dst]);* e posteriormente iremos associar essa *action* a uma mensagem, e enviar a mesma ao *switch*. O ficheiro relativo a este exercício está titulado como *of_tutorial_8.2*.

Mais uma vez, seguimos os testes referidos no enunciado, de modo a verificar o funcionamento da função.

Fazendo um *ping*, o *host1* e o *host2*, obtivemos o seguinte resultado:



```
mininet@mininet-vm: ~
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.203 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.049 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.049/0.126/0.203/0.077 ms
mininet> h1 ping -c2 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.210 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.052 ms

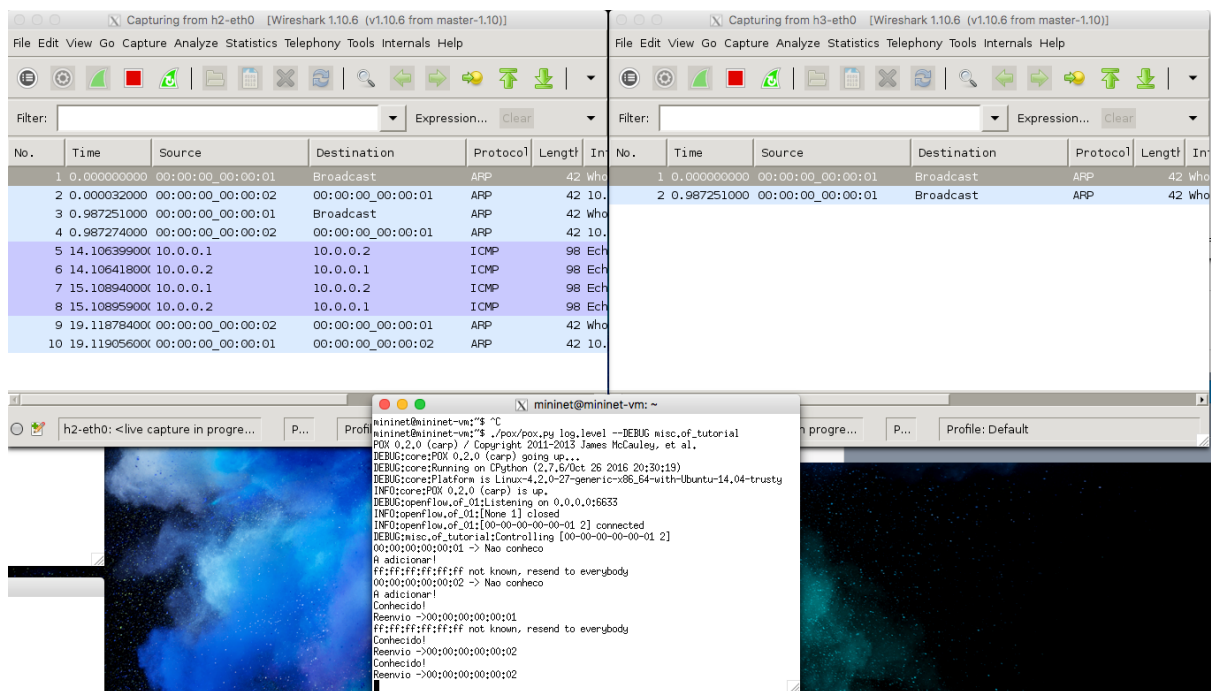
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.052/0.131/0.210/0.079 ms
mininet> h1 ping -c2 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.207 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.055 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.055/0.131/0.207/0.076 ms
mininet>
```

```
mininet@mininet-vm: ~$ ^C
mininet@mininet-vm: ~$ ./pox/pox.py log.level --DEBUG misc.of_tutorial
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:30:19)
DEBUG:core:Platform is Linux-4.2.0-27-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 2]
00:00:00:00:00:01 -> Nao conheco
A adicionar!
ff:ff:ff:ff:ff:ff not known, resend to everybody
00:00:00:00:00:02 -> Nao conheco
A adicionar!
Conhecido!
Reenvio ->00:00:00:00:00:01
ff:ff:ff:ff:ff:ff not known, resend to everybody
Conhecido!
Reenvio ->00:00:00:00:00:02
Conhecido!
Reenvio ->00:00:00:00:00:02
```

Pode-se verificar pela figura do lado esquerdo que foi feito dois *pings*, entre o *host1* e o *host2*, sendo que paralelamente, pela figura do lado direito, só houve um pedido ao *controller*. Desta vez, os pacotes não foram todos enviados para o *controller*, como acontecia no exercício anterior, uma vez que da segunda vez que fazemos o ping ele tem uma *action* que determina qual é a porta para onde é encaminhado o tráfego, não sendo necessário o envio em broadcast para todas as portas.

Com o auxílio do *wireshark*, fizemos a mesma ação: um *ping* com dois pacotes, do *host1* para o *host2*, obtendo assim o seguinte resultado:



Como se pode verificar pela figura acima, ao fazer o *ping*, entre os dois *hosts*, houve o envio do primeiro pacote seguido do envio do segundo pacote, onde desta vez o *controller* já sabia para onde enviar (dado o adicionamento à *flow table* na primeira vez), e desta forma enviou logo o pacote para o porto correto.

Para verificar as *flow tables* executamos o seguinte:

```
mininet@mininet-vm:~$ sudo ovs-ofctl del-flows s1
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=20.681s, table=0, n_packets=4, n_bytes=336, idle_age=3, in
 _port=1 actions=output:2
 cookie=0x0, duration=19.696s, table=0, n_packets=3, n_bytes=238, idle_age=2, in
 _port=2 actions=output:1
mininet@mininet-vm:~$
```

Após executar o comando `sudo ovs-ofctl del-flows s1` percebemos que não temos conectividade entre *host1-host2*, *host1-host3* e *host2-host3*. De seguida ao fazermos os *pings*, tal como pedido no exercício, percebemos que a *flow table* fica com *actions* novas criadas pelo código do exercício 8.2. Assim, ao fazermos um *ping* entre o *host1* e o *host2*, iremos inserir *flows* na tabela, e dessa forma o controlador da segunda vez já não irá receber os pacotes.