UNIVERSITÀ DI PISA

Computer Engineering

Advanced Networks Architecture and Wireless Systems

Asynchronous Message Delivery

Group Project Report

TEAM MEMBERS:
Biagio Cornacchia
Matteo Abaterusso

Academic Year: 2022/2023

Contents

1 Introduction

This project consists of developing an **SDN-based network** that implements an **Asynchronous Message Delivery** (AMD) System. An AMD is a communication model based on the store-and-forward technique that allows a client to send messages to a server that can be temporarily offline. This results in decoupling the client and the server.

So, in the system can be identified two different entities: the **client** which sends the messages and the **server** that receives and consumes them. To this aim, the server needs to subscribe itself to the AMD system that will assign it a **virtual IP** and **MAC**. These addresses can be used by the client to contact it. During the lifetime of the server in the system, it can change its status from *online* to *offline*. In this case the system has to buffer the incoming messages and resend them to it as soon as it is back *online*. Instead, if the server is *online*, all client requests are delivered synchronously.

An example of this system can be the following:

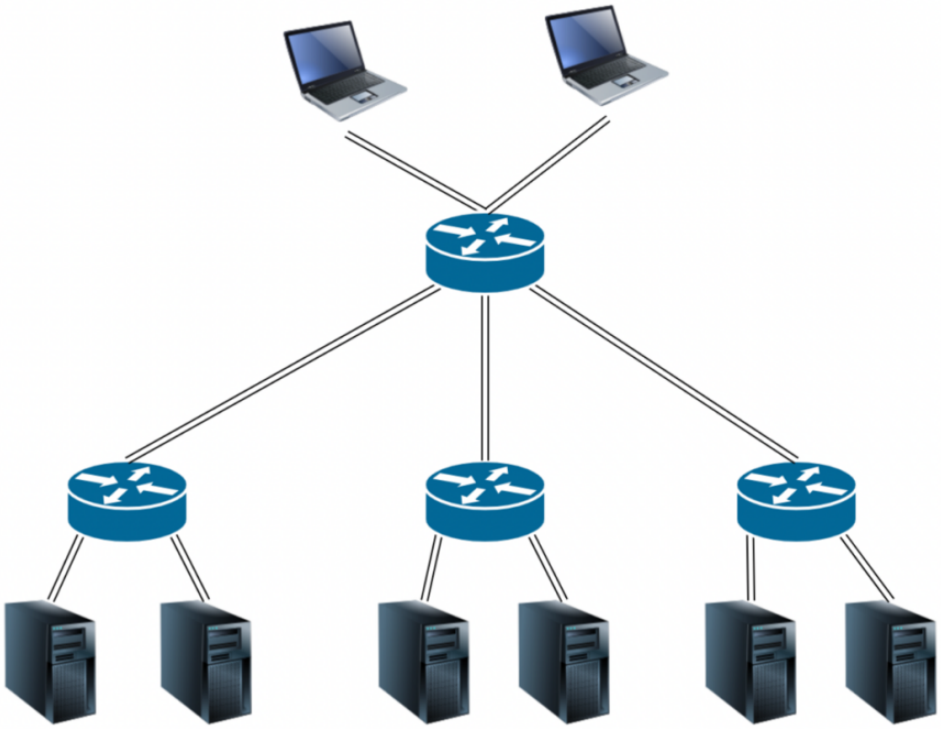

Figure 1: System overview.

The server uses the **REST APIs** exposed by the system to *subscribe*/*unsubscribe* and to update its *status*. Moreover, in a real case multiple servers are subscribed so even the client can use the **REST APIs** to retrieve information about them.

The AMD system has been simulated using **Mininet** that allows to create a realistic virtual network, running real kernel, switch and application code. The network is composed by **Open vSwitch** (OVS) instances that are open-source implementation of a distributed virtual multi-layer switches, supporting the **OpenFlow protocol** to communicate with the SDN controller. The latter is provided by a Java framework called **Floodlight** that is an implementation of an OpenFlow controller.

2 Design

In this section will be shown all the design choices that have been taken. A key point of the design phase is the use of **virtual addresses** to communicate with the servers. This allows the controller to act in a **transparent manner** during the messages exchange, translating the virtual addresses into physical ones if the server is *online* or storing them in a buffer if the server is *offline*.

2.1 REST APIs

In order to allow a server to subscribe, update its status and unsubscribe, the controller provides two possible resources: **subscription** and **status**. In addition, the controller provides the **info** resource that allows the clients to retrieve information about all the subscribed servers.

2.1.1 Subscription

The endpoint related to this resource is:

- http://<CONTROLLER_IP>:8080/amd/server/subscription/json

The methods allowed for this endpoint are the following:

- **POST**: it is used to subscribe a server. This request requires the physical IP and MAC address of the server, and a description of the provided services. If the operation is successful, it returns the virtual IP address assigned to the server. The controller has to check if the server is already subscribed.
- **DELETE**: it is used to unsubscribe a server. This request requires the virtual IP assigned to the server during the subscription. It returns the result of the operation. The controller has to check if the server is subscribed.

2.1.2 Status

The endpoint related to this resource is:

- http://<CONTROLLER_IP>:8080/amd/server/status/json

The method allowed for this endpoint is the following:

- **PUT**: it is used to update the status of the server. If the server wants to go *online*, it has to specify the virtual IP assigned during the subscription and its physical IP and MAC. Instead, if the servers wants to go *offline*, it has to specify only the virtual IP. In both cases the request returns the result of the operation. The controller has to verify that the server is subscribed and that the new requested status is different from the current one.

Notice that, when the server wants to come back *online*, it can change its previous physical addresses in a **transparent manner** (from the client point of view).

2.1.3 Info

The endpoint related to this resource is:

- http://<CONTROLLER_IP>:8080/amd/server/info/json

The method allowed for this endpoint is the following:

- **GET**: it is used to retrieve a list of all subscribed servers and the services they offer.

2.2 ARP requests

The virtual IPs assigned to the servers don't belong to the same **broadcast domain** of the network, so the controller has to be capable of intercept and reply to all the *ARP requests* related to these addresses. In particular, when the controller receives one of them, it creates an *ARP reply* with the following fields:

- **Sender Protocol Address**: virtual IP of the *ARP request*
- **Sender Hardware Address**: virtual MAC related to the requested virtual IP
- **Target Protocol Address**: IP address of the client that made the *ARP request*
- **Target Hardware Address**: MAC address of the client that made the *ARP request*

Then the *ARP reply* is encapsulated into an Ethernet frame and sent back to the source of the request.

An example of the ARP management is shown below:

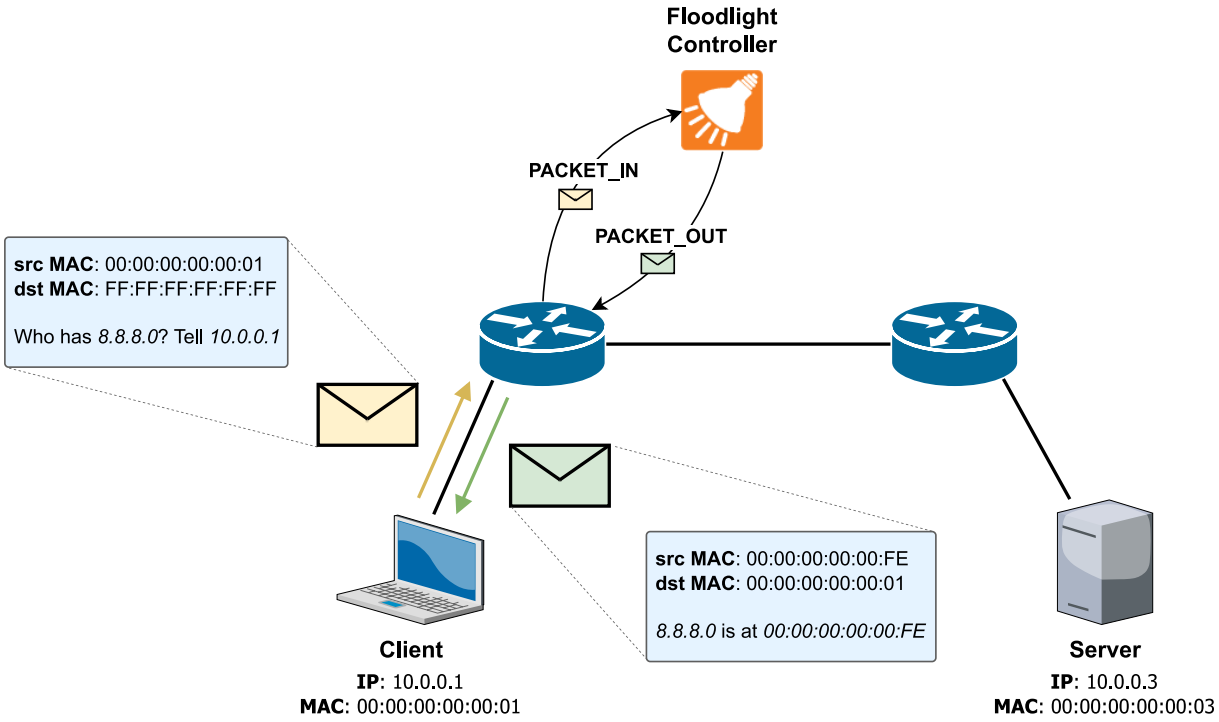

Figure 2: ARP request and ARP reply.

2.3 IPv4 requests

The management of **IPv4 packets** in the network depends on the **status** of the destination server.

If the server is *online*, it is necessary to consider two cases:

1. the packet goes from the client to the server
2. the packet goes from the server to the client

The first case happens when a packet produced by a client reaches the first switch. The **match rules** that identify this case are the following:

- The **destination MAC** must be a virtual MAC associated to a subscribed server
- The **destination IP** must be the virtual IP associated to the same subscribed server

The **actions** performed on these packets are the following:

- The **destination MAC** is translated to the physical MAC associated to the subscribed server
- The **destination** IP is translated to the physical IP associated to the subscribed server
- The packet is forwarded applying the *Shortest Path algorithm*

An example of client-to-server *IPv4 packet* management is shown below:

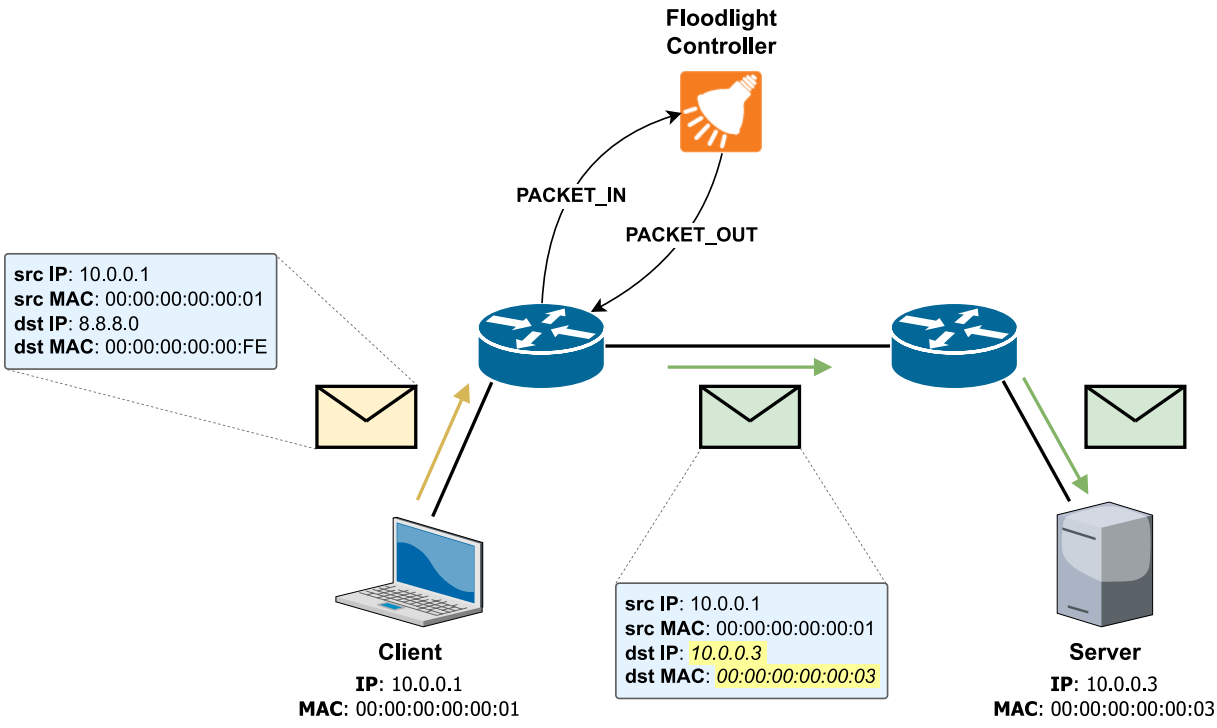

Figure 3: Client-to-server IPv4 packet.

Instead, the second case occurs when a packet (usually a response) produced by a server reaches the first switch. The **match rules** that identify this case are the following:

- The **source MAC** must be a physical MAC associated to a subscribed server
- The **source IP** must be the physical IP associated to the same subscribed server

The **actions** performed on these packets are the following:

- The **source MAC** is translated to the virtual MAC associated to the subscribed server
- The **source IP** is translated to the virtual IP associated to the subscribed server
- The packet is forwarded applying the *Shortest Path algorithm*

An example of server-to-client *IPv4 packet* management is shown below:

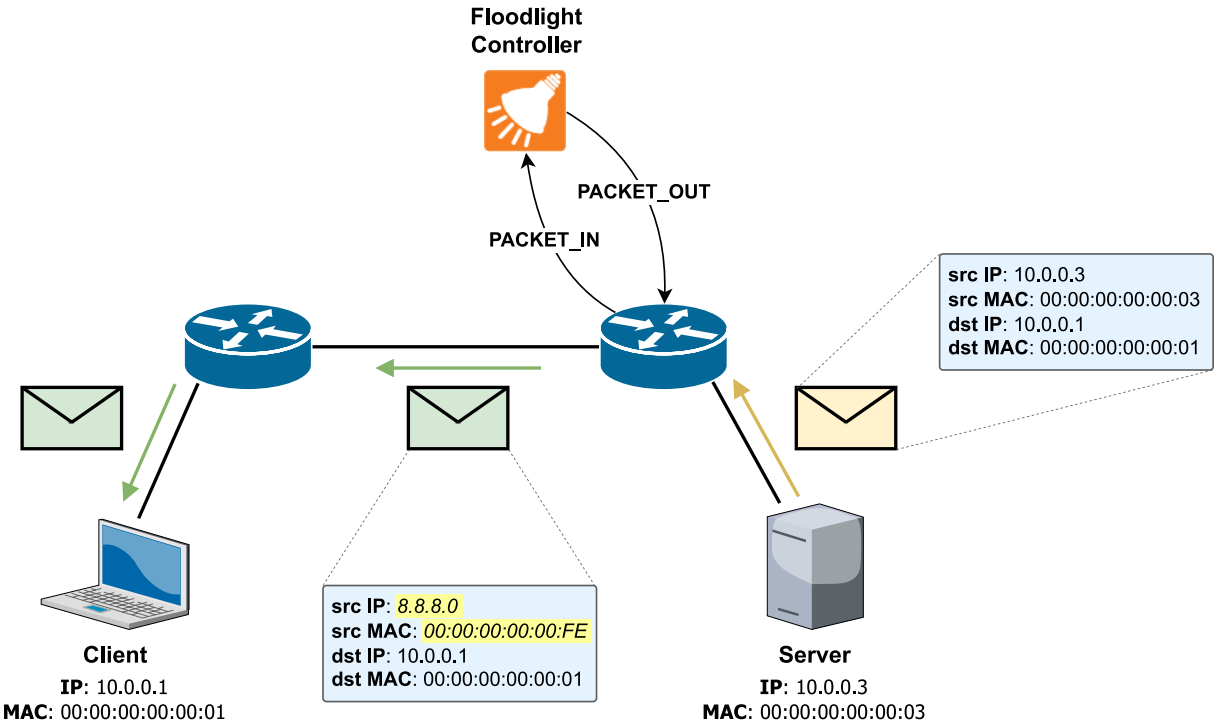

Figure 4: Server-to-client IPv4 packet.

As said before, both of these cases occur on the **access switches**, i.e. on the switches to which clients and servers are directly connected. All other switches will be in charge of **forwarding** the translated packet to the destination.

If the server is *offline*, the first switch that receives packets intended for a subscribed server, sends them to the controller which **stores** them in a **buffer**. When the server comes back *online*, the controller sends the packets in the buffer to the **switch closest** to it in order to minimize the network traffic. Obviously, before sending the packets to the switch, the controller will perform translation from virtual addresses to physical addresses. This is done at this stage and not before as the server may come back *online* with different physical addresses.

An example of packets buffering is shown below:

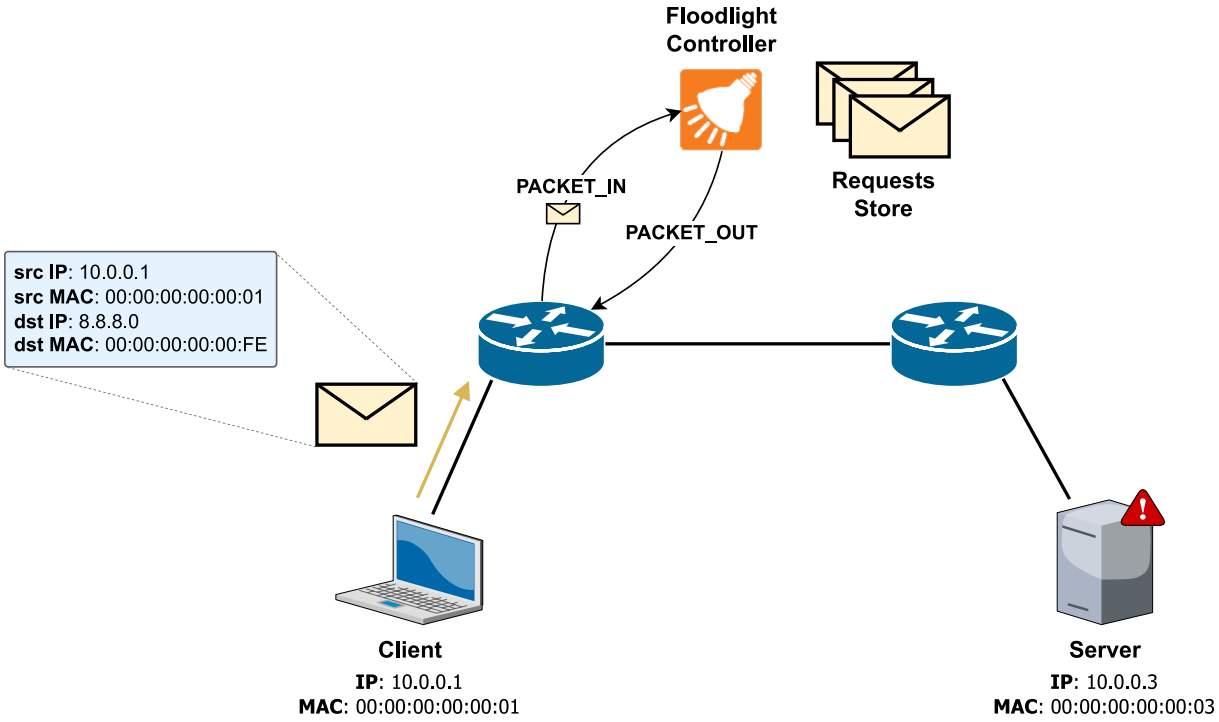

Figure 5: Packets buffering when the server is offline.

All these procedures are carried out assuming that the servers **do not communicate with each other**.

2.3.1 Flow Entries

An important design choice is if it would be worth to use flow entries so that the switch does not contact the controller for every packet received. The main parameters that should be analyzed are the *soft timeout* and *hard timeout* which establish the lifetime of the flow entries on a switch.

The main problem that can arise is that if a server changes is status from *online* to *offline* but the address translation entries are not yet expired, the switch will continue to wrongly process and send packets to the destination. This will cause a **transition time** during which packets will be lost. So, the trade-off is to choose a timeout large enough to avoid too many interactions with the controller but small enough to limit packets loss.

The chosen approach is to actively intervene when the server status changes (from *online* to *offline*) by **removing** the flow entries related to that server. The **advantage** is the possibility to set a fairly high timeout without losing a lot of packets. On the other hand, the **disadvantage** is the fact that the controller has to contact different switches to complete the server status update. Obviously, if the server status changes frequently, this solution may not be suitable. Moreover, the fact that translation flow entries are installed only on **access switches** is exploited to interact with a reasonable number of switches (refer to Section 3.1.3 for more details about optimization).

3 Implementation

As said before, the AMD has been developed using the Java framework called **Floodlight**, that implements an **OpenFlow** controller. The following figure shows the package structure of the AMD module:

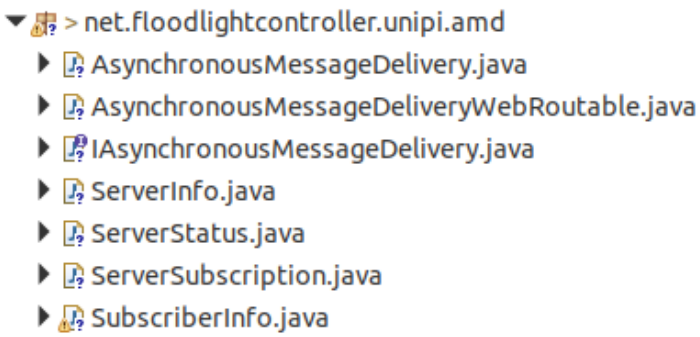

Figure 6: AMD Package

All the classes will be explained in detail.

3.1 Implementation Issues

Before analyzing the classes, it is necessary to delve into some issues related to the framework used.

3.1.1 Forwarding Module Flooding

The default behaviour of the *Forwarding module* is that if the destination address of the packet is **unknown** then a **flooding** action is applied. In the *AMD system*, the **client-to-server** packets have a virtual IP as destination so it is always unknown. Since the *Forwarding module* is located before the *AMD module* in the pipeline, firstly the packet is flooded as it is and then translated and forwarded accordingly to the *Shortest Path*. This results in having **duplicated packets** in the network.

A similar problem occurs for the **server-to-client** reply packets which have the physical IP of the server as source address. In this case, the packets are firstly forwarded by the *Forwarding module*, and then translated and forwarded by the *AMD module*. This results again in having **duplicated packets** in the network.

An example of duplicated packet is shown in Section 4.6.

To solve this problem, the *Forwarding module* has been modified in order to **ignore** packets causing duplicates, that will be handled only by the *AMD module*.

The change applied to the *Forwarding module* (in the function *processPacketInMessage* is shown below:

```
if (eth.getPayload() instanceof IPv4) {
    IPv4 pkt = (IPv4)eth.getPayload();

    IAsynchronousMessageDelivery amd =
```

```
            this.context.getServiceImpl(IAsynchronousMessageDelivery.class);

    if (amd.isHandledByAMD(pkt.getSourceAddress().toString(),
                           pkt.getDestinationAddress().toString())) {
        return Command.CONTINUE;
    }
}
```

The *isHandledByAMD* function is explained in the Section 3.2.

3.1.2 OVS Security Policy

By default, the **Open vSwitch** implements a security policy that **drops** packets with an **external source MAC address**, unless there is a flow mod that matches the packet. In the *AMD system*, each server has an associated virtual MAC that will be considered always as external. This results in dropping the **server-to-client** packets when it reaches a switch that has only to forward it (because it has been already translated).

To solve this problem, it is necessary to install a **flow mod** on all the switches to avoid their default behaviour. To grant security, this flow mod is installed when a server subscribes to the system and is removed when it unsubscribes.

The parameters specified in the flow mod are the following:

- **priority** = 1: a low priority has been chosen in order to not intervene if others rules match the packet
- **dl_src** = VIRTUAL_MAC: it represents the match rule
- **action** = normal: it forces the switch to use the "normal pipeline" to process the packet

3.1.3 Flush Flow Mods Optimization

As introduced in Section 2.3.1, the translation **flow mods** are installed only on the **access switches**, so only the latter should be contacted to remove the flow mods. In order to retrieve a list of only access switches, it is exploited the *Devices Service module* to get all the hosts connected to the network. Through them it is possible to get the access switch to which they are connected.

The flow mods that the *AMD system* installs are of two types: the **client-to-server** translation and the **server-to-client** translation. The latter can be easily removed since the access switch connected to the server can be **directly identified** in the list of switches. Instead, on all the other access switches found is made an **attempt to remove** the client-to-server translation (because it not possible to find the hosts that have communicated with the server).

3.2 Asynchronous Message Delivery

This class implements the functionalities of the AMD system. The main attributes of the class are the following:

- `protected Queue<String> availableAddresses;`

This structure contains the **available virtual addresses** that can be assigned to a new subscriber. A *Queue* is chosen as data structure so that a virtual address that has just been released can't be right after reassigned, since it may still be present in the ARP tables of hosts.

- `protected Map<String, SubscriberInfo> subscribedServers;`

  This structure contains all the **subscribers info**. The *SubscriberInfo* class is composed by the physical addresses, the status and the services offered by a server. To retrieve the information about a subscriber must be used its **virtual IP address**.

- `protected Map<String, Queue<Ethernet>> requestsStore;`

  This structure contains a **buffer** of Ethernet frames for each subscriber, used when the subscriber goes *offline*. To retrieve a specific buffer, the **virtual IP** of the subscriber must be used.

- ```
  protected final String[][] virtualAddresses = {
      {"00:00:00:00:00:FE", "8.8.8.0"}, {"00:00:00:00:01:FE", "8.8.8.1"},
      {"00:00:00:00:02:FE", "8.8.8.2"}, {"00:00:00:00:03:FE", "8.8.8.3"},
      {"00:00:00:00:04:FE", "8.8.8.4"}, {"00:00:00:00:05:FE", "8.8.8.5"}
  };
  ```

  This structure represents the **pool** of **virtual addresses** that can be assigned to the subscribers.

The main methods of the class are the following:

- `private String getVirtualMACAddress(String virtualIp)`

  It returns the virtual MAC associated to the specified virtual IP.

- `private Command handleARPRequest(IOFSwitch sw, Ethernet eth, OFPacketIn pi)`

  This function is called when an *ARP request* is received. It creates and sends the *ARP reply*.

- `private Command handleIPPacket(IOFSwitch sw, OFPacketIn pi, Ethernet eth)`

  This function is called when an *IPv4 packet* is called. It creates the **flow mod** if the server is *online* or it **buffers** the packet.

- ```
  private OFPort getOuputPort(DatapathId srcSwitchDPID,
                              MacAddress dstHostMAC,
                              OFPort srcPort)
  ```

  This function exploits the *Shortest Path* calculation provided by the *Routing Service*, in order to find the **output port** to use in the **flow mod**.

- `private boolean sendBufferedPackets(String virtualIp, String ip, String mac)`

  This function is called when a server comes back **online**. It is in charge of retransmit the buffered packets to the server.

- `private void flushFlowRules(IPv4Address serverVirtualIP)`

  This function is called when a server goes *offline* or *unsubscribes* itself. It **removes** all the **flow rules** associated to that server from all the interested switches.

- `private void enableSecurityPolicy(MacAddress mac)`

  It is called when there is a **new subscription**. It installs a **flow mod** on all the switches in order to allow the usage of the **virtual MAC** associated with the subscribed server.

- `private void disableSecurityPolicy(MacAddress mac)`

  It is called when a server **unsubscribes** itself. It removes the **flow mods** installed during the subscription with the *enableSecurityPolicy*.

The following methods represent the implementation of the interface *IAsynchronousMessageDelivery*, used by other modules and classes:

- `public boolean isHandledByAMD(String srcIP, String dstIP)`

  This function is used by the *Forwarder* module in order to check if a packet should be handled by itself or by the AMD system.

- `public String subscribeServer(String ip, String mac, List<String[]> services)`

  This function is used by *ServerSubscription* class to **subscribe** a server.

- `public boolean unsubscribeServer(String virtualIp)`

  This function is used by *ServerSubscription* class to **unsubscribe** a server.

- `public int setServerOnline(String virtualIp, String ip, String mac)`

  This function is used by *ServerStatus* class to change the **status** of a server from *offline* to *online*.

- `public int setServerOffline(String virtualIp)`

  This function is used by *ServerStatus* class to change the **status** of a server from *online* to *offline*.

- `public List<List<String>> getServersInfo()`

  This function is used by *ServerInfo* class to retrieve a list of **all subscribed servers** and the **services** they offer.

3.3 Asynchronous Message Delivery Web Routable

This class enables the REST APIs service. The main methods are the following:

- `public Restlet getRestlet(Context context)`

  This function associates for each **endpoint** the class that manages the **specified resource**.

- `public String basePath()`

  It returns the **root path** for the **resources**.

3.4 Server Subscription

This class represents the **subscription** resource. The main methods of the class are the following:

- `public Map<String, Object> subscribe(String postBody)`

  This function is called when a *POST request* to **subscribe** a new server is received. It uses the *IAsynchronousMessageDelivery* to perform the operation.

  Below is shown an example of request and response.

```
{
  "ip": "10.0.0.3",
  "mac": "00:00:00:00:00:03",
  "services": [
    {
      "port": "5000",
      "description": "UDP server"
    }
  ]
}
```

```
{
  "status": "Success",
  "virtualIp": "8.8.8.0"
}
```

- `public Map<String, Object> unsubscribe(String deleteBody)`

  This function is called when a *DELETE request* to **unsubscribe** a server is received. It uses the *IAsynchronousMessageDelivery* to perform the operation.

  Below is shown an example of request and response.

```
{
  "virtualIp": "8.8.8.0"
}
```

```
{
  "status": "Success"
}
```

3.5 Server Status

This class represents the **status** resource. The method of the class is the following:

- `public Map<String, Object> changeStatus(String putBody)`

  This function is called when a *PUT request* to update the **status** of a subscriber is received. It uses the *IAsynchronousMessageDelivery* to perform the operation.

  Below is shown an example of request and response for the *offline* to *online* case.

```
{
  "newStatus": "online",
  "virtualIp": "8.8.8.0",
  "ip": "10.0.0.3",
  "mac": "00:00:00:00:00:03"
}
```

```
{
  "status": "Success"
}
```

Below is shown an example of request and response for the *online* to *offline* case.

```
{
  "newStatus": "offline",
  "virtualIp": "8.8.8.0"
}
```

```
{
  "status": "Success"
}
```

3.6 Server Info

This class represents the **info** resource. The method of the class is the following:

- `public Map<String, Object> GetInfo()`

  This function is called when a *GET request* is received. It uses the *IAsynchronousMessageDelivery* to reply with the information about the subscribers.

  Below is shown an example of response.

```
{
  "availableServices": [
    {
      "virtualIp": "8.8.8.0",
      "services": [
        "5000: UDP server"
      ]
    },
    {
      "virtualIp": "8.8.8.1",
      "services": [
        "9090: UDP server",
        "6000: Web server"
      ]
    }
  ]
}
```

4 Testing

In order to verify the correctness of the *AMD system*, the network represented in Figure 7 was created using **Mininet**. It also shows the names of the hosts and interfaces used during the testing phase.

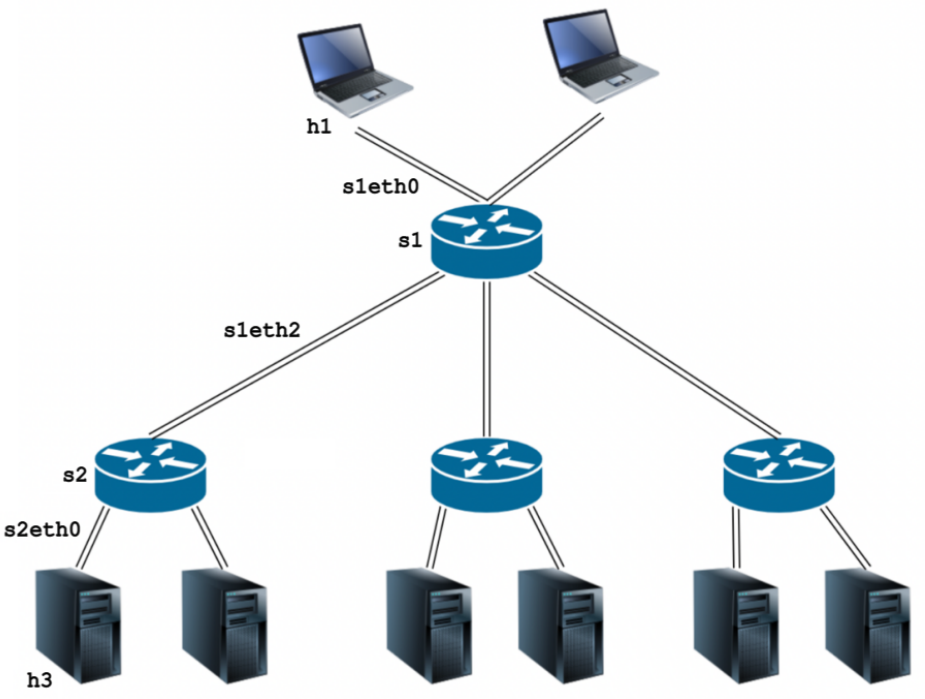

Figure 7: Test network.

The **test workflow** will begin with the subscription of a new server to the system. Then a *UDP packet* will be sent from the client to the server and the traffic will be analyzed using *tcpdump*. The same test will be performed when the server is *offline* and after it is back *online*. After that, the traffic generated by a *ping* command is analyzed to test the server-to-client communication. Finally, the server will be unsubscribed.

4.1 Server Subscription

To test the subscription request, the **Boomerang SOAP and REST Client** browser extension is used. The server under consideration is the Mininet host *h3* with physical IP and MAC address shown in Figure 8. It also shows that the controller assigned to the server the **virtual IP** *8.8.8.0*.

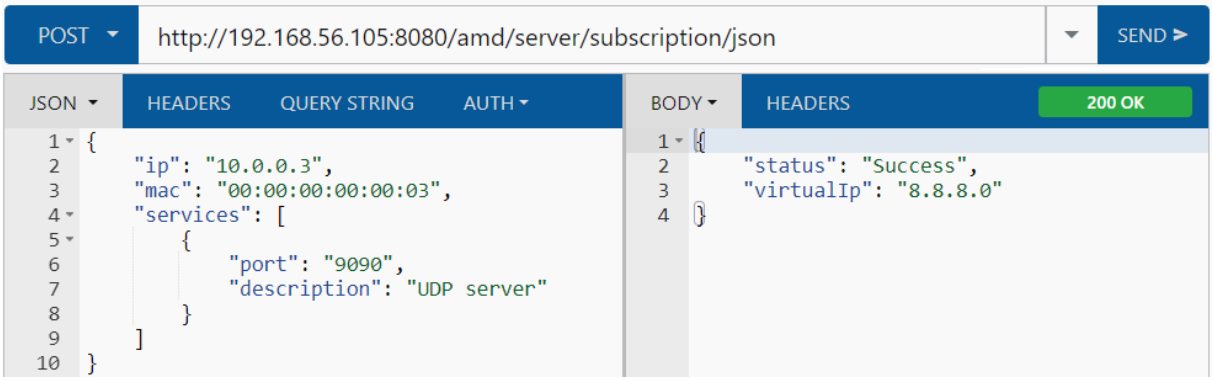

Figure 8: Example of server subscription request.

After the subscription, a *GET request* is issued to check if the server is correctly subscribed. As expected, Figure 9 shows that the controller responds with a list containing only one server, having **virtual IP** *8.8.8.0* and the **services** specified in the subscription.

Figure 9: Example of request to retrieve the list of subscribers.

4.2 UDP Packet Test

Starting from a server subscribed and online, an *UDP packet* is sent from the Mininet host *h1* (considered as client) to the server *h3*. As shown in Figure 10, the tool used is *netcat*.

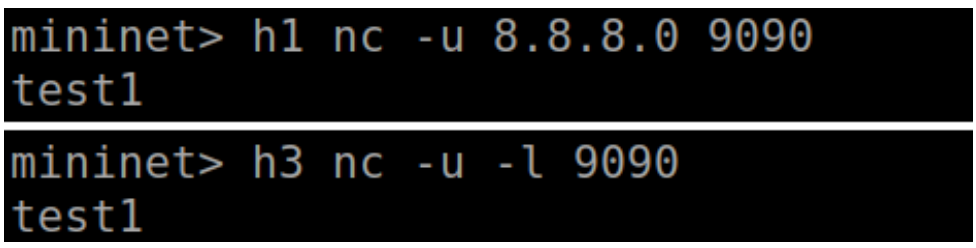

Figure 10: *Netcat* client and server example.

The Figure 11 shows the **network traffic** analyzed using *tcpdump*. It is possible to see that the packet going from host *h1* to switch *s1* has destination IP *8.8.8.0*, while the

packet going from switch *s1* to switch *s2* has destination IP *10.0.0.3*. As intended, the switch *s1*, that is the **access switch** for *h1*, performed the **addresses translation**.

```
student@osboxes:~/Downloads$ sudo tcpdump udp -i s1eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:47:08.893659 IP 10.0.0.1.55521 > 8.8.8.0.9090: UDP, length 6

student@osboxes:~/Downloads$ sudo tcpdump udp -i s1eth2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth2, link-type EN10MB (Ethernet), capture size 262144 bytes
10:47:08.916889 IP 10.0.0.1.55521 > 10.0.0.3.9090: UDP, length 6

student@osboxes:~/Downloads$ sudo tcpdump udp -i s2eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s2eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:47:08.919649 IP 10.0.0.1.55521 > 10.0.0.3.9090: UDP, length 6
```

Figure 11: Example of *tcpdump* output.

4.3 UDP Packet Test with Offline Server

This test aims to test the **buffering functionality** needed when a server goes *offline*. Through Boomerang extension, an *update status request* is simulated for the server *h3* with virtual IP *8.8.8.0*.

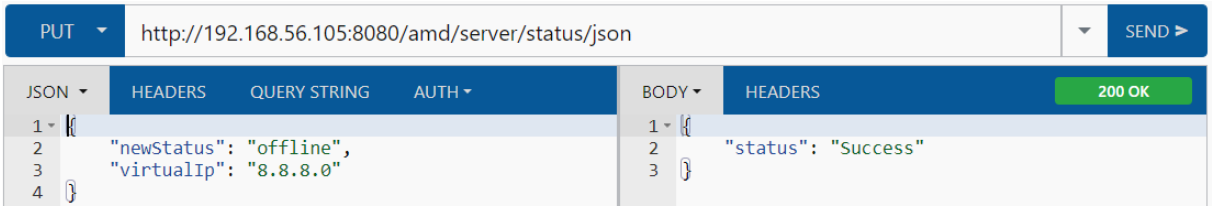

Figure 12: Example of status update to *offline*.

Also in this case, an *UDP packet* is sent from host *h1* to server *h3*. As shown in Figure 13, the packet with destination IP *8.8.8.0* reaches the switch *s1* and it is not forwarded.

```
student@osboxes:~/Downloads$ sudo tcpdump udp -i s1eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:56:59.944695 IP 10.0.0.1.46457 > 8.8.8.0.9090: UDP, length 6

student@osboxes:~/Downloads$ sudo tcpdump udp -i s1eth2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth2, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Figure 13: Example of *tcpdump* output when the server is *offline*.

In order to verify that the packet sent is buffered in the controller, the server is simulated to come back *online* as shown in Figure 14.

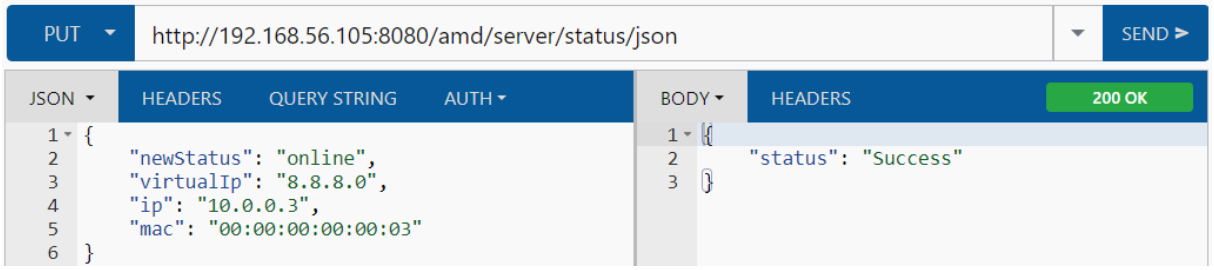

Figure 14: Example of status update to *online*.

Tcpdump shows that the switch *s2* does not receive any packet from the switch *s1*, but it sends to server *h3* the **previous packet** with the destination IP **already translated**. This means that the controller sends the packet directly to the **switch closest** to the server.

```
student@osboxes:~/Downloads$ sudo tcpdump udp -i s1eth2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth2, link-type EN10MB (Ethernet), capture size 262144 bytes

student@osboxes:~/Downloads$ sudo tcpdump udp -i s2eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s2eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:01:35.245062 IP 10.0.0.1.46457 > 10.0.0.3.9090: UDP, length 6
```

Figure 15: Example of *tcpdump* output when the server is *online*.

4.4 Ping Test

So far, only **client-to-server** packets have been analyzed. To test the **server-to-client** path, the result of the *ping* command has been analyzed.

```
mininet> h1 ping -c 3 8.8.8.0
PING 8.8.8.0 (8.8.8.0) 56(84) bytes of data.
64 bytes from 8.8.8.0: icmp_seq=1 ttl=64 time=16.8 ms
64 bytes from 8.8.8.0: icmp_seq=2 ttl=64 time=3.21 ms
64 bytes from 8.8.8.0: icmp_seq=3 ttl=64 time=2.36 ms

--- 8.8.8.0 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 2.367/7.481/16.864/6.644 ms
```

Figure 16: Example of *ping* from client to server.

As shown in Figure 17, the *h1* access switch (*s1*) performs the addresses translation for the **client-to-server** packets whereas the *h3* access switch (*s2*) performs the addresses translation for the **server-to-client** packets.

```
student@osboxes:~/Downloads$ sudo tcpdump icmp -i s1eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:12:42.440767 IP 10.0.0.1 > 8.8.8.0: ICMP echo request, id 20445, seq 1, length 64
11:12:42.457614 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 1, length 64
11:12:43.441675 IP 10.0.0.1 > 8.8.8.0: ICMP echo request, id 20445, seq 2, length 64
11:12:43.444796 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 2, length 64
11:12:44.446272 IP 10.0.0.1 > 8.8.8.0: ICMP echo request, id 20445, seq 3, length 64
11:12:44.448534 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 3, length 64

student@osboxes:~/Downloads$ sudo tcpdump icmp -i s1eth2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1eth2, link-type EN10MB (Ethernet), capture size 262144 bytes
11:12:42.451152 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 1, length 64
11:12:42.457121 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 1, length 64
11:12:43.442129 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 2, length 64
11:12:43.444527 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 2, length 64
11:12:44.446358 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 3, length 64
11:12:44.448526 IP 8.8.8.0 > 10.0.0.1: ICMP echo reply, id 20445, seq 3, length 64

student@osboxes:~/Downloads$ sudo tcpdump icmp -i s2eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s2eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:12:42.454219 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 1, length 64
11:12:42.454557 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 20445, seq 1, length 64
11:12:43.444157 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 2, length 64
11:12:43.444255 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 20445, seq 2, length 64
11:12:44.448361 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 20445, seq 3, length 64
11:12:44.448520 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 20445, seq 3, length 64
```

Figure 17: Example of *tcpdump* output when the *ping* is performed.

Figure 18 shows the **flow mods** installed on the two switches. The first flow mod on switch *s1* uses as match rules the server virtual addresses as destination and the related action is to translate them into physical addresses. This is the addresses translation flow mod for **client-to-server** packets. Instead, the first flow mod on switch *s2* uses as match rules the server physical addresses as source and the related action is to translate them into virtual addresses. This is the addresses translation flow mod for **server-to-client** packets. Finally, the second flow mod on both the switches is the one that avoids dropping the packet having an **external source MAC** (as discussed in Section 3.1.2).

```
student@osboxes:~/Downloads$ sudo ovs-ofctl dump-flows s1 | grep 'cookie'
 cookie=0x0, duration=44.833s, table=0, n_packets=2, n_bytes=196, idle_timeout=120, hard_timeout=240, idle_age=42, ip,dl_dst=00:00:00:00:00:
fe,nw_dst=8.8.8.0 actions=mod_dl_dst:00:00:00:00:00:03,mod_nw_dst:10.0.0.3,output:3
 cookie=0x0, duration=2335.922s, table=0, n_packets=23, n_bytes=1366, idle_age=39, priority=1,dl_src=00:00:00:00:00:fe actions=NORMAL
 cookie=0x0, duration=3830.018s, table=0, n_packets=961, n_bytes=74103, idle_age=4, priority=0 actions=CONTROLLER:65535
student@osboxes:~/Downloads$ sudo ovs-ofctl dump-flows s2 | grep 'cookie'
 cookie=0x0, duration=49.117s, table=0, n_packets=2, n_bytes=196, idle_timeout=120, hard_timeout=240, idle_age=47, ip,dl_src=00:00:00:00:00:
03,nw_src=10.0.0.3 actions=mod_dl_src:00:00:00:00:00:fe,mod_nw_src:8.8.8.0,output:3
 cookie=0x0, duration=2340.220s, table=0, n_packets=10, n_bytes=420, idle_age=43, priority=1,dl_src=00:00:00:00:00:fe actions=NORMAL
 cookie=0x0, duration=3834.322s, table=0, n_packets=443, n_bytes=35279, idle_age=9, priority=0 actions=CONTROLLER:65535
```

Figure 18: Example of *flow tables* content.

4.5 Server Unsubscription

Finally, the **Boomerang** extension is used to simulate the unsubscription request for the server *h3*.

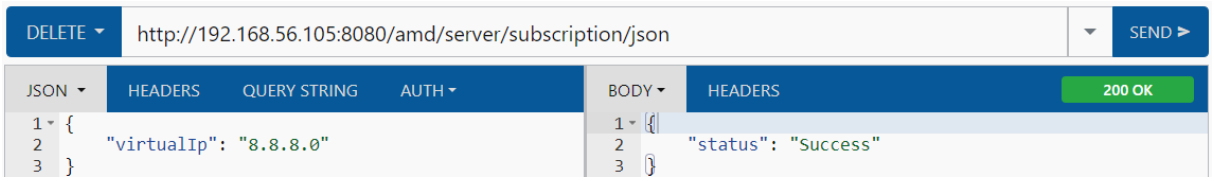

Figure 19: Example of server unsubscription request.

Figure 20 shows that after the unsubscription the *ping* properly fails.

```
mininet> h1 ping -c 3 8.8.8.0
PING 8.8.8.0 (8.8.8.0) 56(84) bytes of data.

--- 8.8.8.0 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2041ms
```

Figure 20: Example of *ping* output when the server is not subscribed anymore.

Figure 21 shows that the **flow mods** have been **removed** from the switches.

```
student@osboxes:~/Downloads$ sudo ovs-ofctl dump-flows s1 | grep 'cookie'
 cookie=0x0, duration=7198.820s, table=0, n_packets=1658, n_bytes=126489, idle_age=0, priority=0 actions=CONTROLLER:65535
student@osboxes:~/Downloads$ sudo ovs-ofctl dump-flows s2 | grep 'cookie'
 cookie=0x0, duration=7205.082s, table=0, n_packets=686, n_bytes=53416, idle_age=4, priority=0 actions=CONTROLLER:65535
```

Figure 21: Example of *flow tables* content after server unsubscription.

4.6 Duplicated Packet Issue

One last test has been made to show the problem of duplicated packets. In Figure 22 is shown what happens **disabling the IP packet control** in the *Forwarding* module (problem discussed in Section 3.1.1). *Tcpdump* displays that the server *h3* receives packet translated by switch *s1*, the packet translated by switch *s2* and the original packet sent by host *h1* that is **flooded** by the Forwarding module on both switches.

```
student@osboxes:~/Downloads$ sudo tcpdump udp -i s2eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s2eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
06:38:26.396833 IP 10.0.0.1.44689 > 8.8.8.0.9090: UDP, length 5
06:38:26.397031 IP 10.0.0.1.44689 > 10.0.0.3.9090: UDP, length 5
06:38:26.407161 IP 10.0.0.1.44689 > 10.0.0.3.9090: UDP, length 5
```

Figure 22: Example of duplicated packets received by the server.