UNIVERSITY OF PISA

School of Engineering

Master of Science in Computer Engineering

Advanced Network Architectures and Wireless Systems

PROJECT DOCUMENTATION

SDN SUPPORT FOR MOBILITY

WORKGROUP:

Antonio Nunzio Pio Di Noia

Diego Casu

Leonardo Bernardi

ACADEMIC YEAR 2020/2021

Contents

1 Introduction **3**

2 Design **4**

2.1 REST interface . . . 4

2.1.1 Endpoint "users" . . . 4

2.1.2 Endpoint "service-address" . . . 4

2.1.3 Endpoint "servers" . . . 5

2.1.4 Endpoint "access-switches" . . . 5

2.2 Packet filtering . . . 6

2.3 ARP requests . . . 7

2.3.1 ARP requests from users . . . 7

2.3.2 ARP requests from servers . . . 8

2.4 Generic IP packet . . . 9

2.4.1 Requests to the service transiting through an access switch . . . 10

2.4.2 Responses from the service transiting through an access switch . . . 12

2.4.3 Packets traversing the core switches . . . 13

3 Testing **15**

3.1 Ping from a user towards the virtual address . . . 16

3.2 Ping from a user towards other users or generic addresses . . . 16

3.3 Ping from a server towards a user or another server . . . 17

3.4 Requests fulfilled by the closest server . . . 18

3.5 Load balancing with multiple equal cost paths . . . 19

3.6 Failure of a switch port . . . 19

3.7 Mobility scenario . . . 20

1. Introduction

The objective of this project is to design and develop a Floodlight module dedicated to the support of client mobility inside a network domain, which adopts the Ethernet and IPv4 standards to enable the communication.

The organization managing the domain offers to its clients the possibility to connect to multiple and spatially distributed WiFi access points, from which they can contact a service hosted at a virtual address. The service is implemented by a group of dedicated servers dislocated across the domain and interconnected by a set of SDN-enabled switches, coordinated in turn by a controller executing Floodlight.

The fundamental module requirements are the following:

1. it shall expose a REST interface through which the administrator can register a client to the list of users able to contact the service and configure basic aspects of the service itself, like the virtual address and the group of dedicated servers;
2. it shall enable registered clients to receive responses to their requests by the topologically closest server, i.e. the one that can be reached traversing the minimum number of switches, contacting only the virtual address. This requirement shall hold even in the cases of mobility between access points and sudden failure of links/switches/servers inside the network.

The document is organized as follows: Section 2 discusses the design of the module, with particular attention to the management of the REST interface, the packet filtering, the ARP requests and the generic IP packets; Section 3 reports the testing results obtained emulating the network with the support of Mininet-WiFi.

A reference topology, which will be used in the continuation of the document, is the one shown in Figure 1.

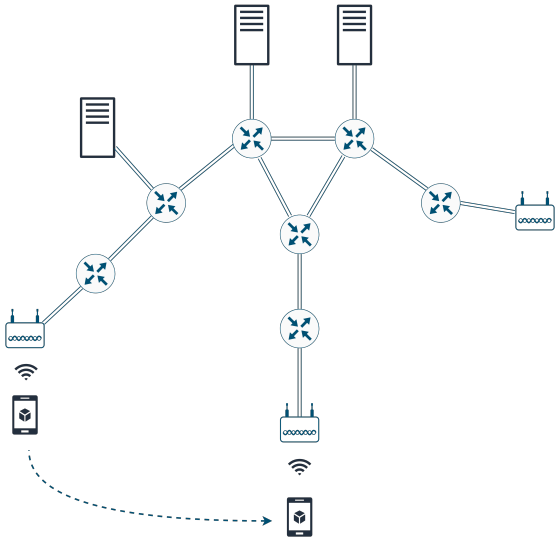

Figure 1: The reference topology, where the access points are non SDN-enabled devices.

2. Design

2.1 REST interface

In order to manage the network, a RESTful interface is offered to the administrator. The controller exposes a set of endpoints that can be used to retrieve network information, issue configuration changes for the service and trigger subscriptions; they are reachable at the following URLs:

- *http://CONTROLLER_IP:8080/ms/users/json*
- *http://CONTROLLER_IP:8080/ms/service-address/json*
- *http://CONTROLLER_IP:8080/ms/servers/json*
- *http://CONTROLLER_IP:8080/ms/access-switches/json*

2.1.1 Endpoint "users"

This endpoint permits to the administrator to manage the users requesting the service. A user is represented by its MAC address and username.

- **GET**
  Retrieves the list of users subscribed to the service. The list is given in the format *"macaddress":"username"*.
- **POST**
  Adds a client to the list of subscribed users. The user is submitted in JSON format specifying the parameters *"mac"* and *"username"*. The module checks if the user is already subscribed, i.e. if the MAC address is already present in the list, and if the username is available.
- **DELETE**
  Removes a client from the list of subscribed users. The user is submitted in JSON format specifying the parameter *"username"*. The module checks if the user is subscribed.

2.1.2 Endpoint "service-address"

This endpoint permits to the administrator to manage the virtual address of the service, i.e. its MAC and IPv4 addresses.

- **GET**
  Retrieves the virtual MAC and IPv4 addresses of the service.

- **POST**
  Sets the virtual MAC and IPv4 addresses of the service. The addresses are submitted in JSON format specifying the parameters *"mac"* and *"ipv4"*.

2.1.3 Endpoint "servers"

This endpoint permits to the administrator to manage the list of servers providing the service. A server is represented by its MAC and IPv4 addresses.

- **GET**
  Retrieves the list of servers providing the service. The list is given in the format *"macaddress":"ipv4"*.

- **POST**
  Adds a server to the list of servers providing the service. The server is submitted in JSON format specifying the parameters *"ipv4"* and *"mac"*. The module checks if the server is already registered.

- **DELETE**
  Removes a server from the list of servers providing the service. The server is submitted in JSON format specifying the parameter *"ipv4"*. The module checks if the server is present in the list.

2.1.4 Endpoint "access-switches"

This endpoint permits to the administrator to manage the list of access switches. An access switch is the first SDN-enabled device encountered by the packet of a user: it can be an access point, if the latter supports OpenFlow, or a switch. Access switches are fundamental to ensure a correct management of the packet filtering, the ARP requests and the virtualization of the service location, as discussed in the following sections.

- **GET**
  Retrieves the list of access switches. The list specifies the *"dpid"* for each switch.

- **POST**
  Adds a switch to the list of access switches. The access switch is submitted in JSON format specifying the parameter *"dpid"*. The module checks if the access switch is already present in the list and if it is connected to the network.

- **DELETE**
  Removes a switch from the list of access switches. The access switch is submitted in JSON format specifying the parameter *"dpid"*. The module checks if the access switch is present in the list.

2.2 Packet filtering

The first operation to perform on packets entering the network is to filter out those that are considered not permitted. A packet is considered not permitted if it is not involved in a communication flow with the service provided by the network domain: out of this scope, the user should have no reason to perform an action.

The filtering must be performed as soon as possible, avoiding that forbidden packets enter the network at all. For this reason, all the filtering actions are managed by the controller whenever a packet, coming from a server or from a user, arrives at the first SDN-enabled switch of the network: the filtering consists in simply discarding or not packets depending on their properties and no rules in the flow tables of the switches are installed. This is achieved stopping the execution of the chain of modules, in particular by avoiding that the packet-in is handled by the *Forwarding* module.

The filtering is based on the following ordered considerations, where packets not marked as accepted or discarded at one level of inspection must undergone the subsequent one:

1. if the packet comes from a server providing the service, it is always accepted. The servers are managed by the network administrator herself, so there is no reason to discard their packets;
2. if the packet comes from a not subscribed user, it is discarded;
3. if the packet comes from an access switch, it must undergone a deeper inspection. If the packet comes from a core switch, it already passed a previous filtering and so it is accepted;
4. if the packet contains an ARP request, it is accepted in the case the IP address to resolve is the virtual IP address of the service, otherwise it is discarded. ARP requests to resolve real IP addresses of other users or servers must be blocked if originating at the user's side;
5. if the packet carries an IPv4 one, it is accepted only if the destination IP and MAC addresses are the virtual ones of the service;
6. if a packet arrives at this level of inspection, it is discarded.

It should be noted that, as explained in the following sections, the connectivity between endpoints is guaranteed by the installation of appropriate flow mods in the switches. The flow mods are not persistent and expire after a certain timeout, therefore there is still the possibility that a user can continue, even if for a short fraction of time, to contact the service after she has been removed from the list of subscribed user. This is unavoidable, but it can be mitigated by tuning carefully the hard timeout of the rules or implementing an authentication method at the application layer.

2.3 ARP requests

After performing packet filtering, only allowed packets are able to transit through the network. Therefore, the first operation to take into account is the handling of ARP requests made by the users and by the servers.

2.3.1 ARP requests from users

Accepted ARP requests made by subscribed users are only those that are issued to resolve the virtual IP address of the service. Given that there is no host with this virtual IP address, the only possibility to reply to such requests is to handcraft the corresponding ARP reply into the controller and to forward it directly to the issuer.

This mechanism is completely transparent from the point of view of the user and it is implemented at the access switches: whenever a subscribed user sends out an ARP request to resolve the virtual IP address of the service into the corresponding virtual MAC address, the access switch will send a packet-in to the controller. The latter generates an ARP reply with the following characteristics:

- Sender IP address: the virtual IP address of the service.
- Sender MAC address: the virtual MAC address of the service.
- Target IP address: the IP address of the user that issued the ARP request.
- Target MAC address: the MAC address of the user that issued the ARP request.

The ARP reply is encapsulated into an Ethernet frame, whose source MAC address is the virtual MAC address of the service, and sent within a packet-out; no flow mods are sent to the switch.

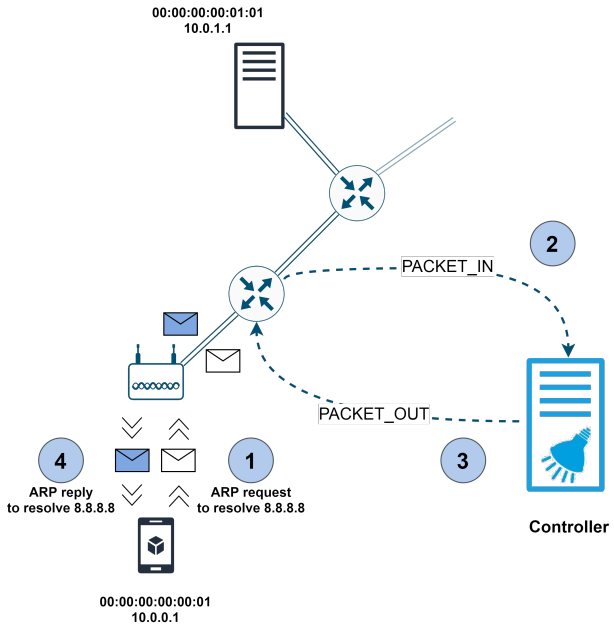

Figure 2: Management of ARP requests to discover the mapping *virtual IP - virtual MAC* of the service.

2.3.2 ARP requests from servers

The second class of allowed ARP requests are those issued by the servers to resolve the IP address of a user or of another server. Even in this case, the choice is to let the controller reply to these requests, leveraging the *DeviceManager* module of Floodlight, which keeps track of the devices in the network. This approach reduces the amount of operations that would be needed in the network; in particular, it avoids:

1. to forward the ARP request to the access switch of the user;
2. to handle the ARP request at the access switch, replacing the IP and MAC addresses of the server with the virtual ones of the service, and to forward the modified ARP request;
3. to handle the ARP reply at the access switch, replacing the virtual IP and MAC addresses of the service with the real ones of the server that issued the ARP request, and to forward the modified ARP reply.

Whenever an ARP request is sent out by a server, the first server-side switch will send a packet-in to the controller, which will craft the corresponding ARP reply, inserting the following values of the fields:

- Sender IP address: the IP address of the user or server object of the request.
- Sender MAC address: the MAC address of the user or server object of the request.
- Target IP address: the IP address of the server that issued the ARP request.
- Target MAC address: the MAC address of the server that issued the ARP request.

The ARP reply is encapsulated into an Ethernet frame, whose source MAC address is the MAC address of the target user or server, and sent within a packet-out; no flow mods are sent to the switch.

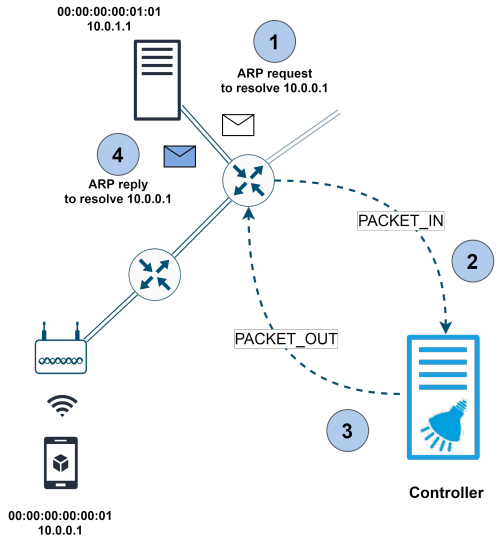

Figure 3: Management of ARP requests made by a server to discover a mapping *IP address - MAC address*.

2.4 Generic IP packet

The IP communication inside the domain is based on the idea that the virtualization of the service location is managed at the access level of the network: the translation of the virtual source/destination address during a packet exchange is delegated to the first SDN-enabled device that receives the packet of the user, namely an access switch. This approach is the most efficient, since:

- it leads to a classical IP-based core network management, which can be addressed using the already available and efficient modules of Floodlight, like the *Forwarding* module;
- it masks completely the virtualization mechanism to any device inside the core network, meaning that servers can exchange packets between themselves or an administrator using their real IP addresses, the same protocols and without the need to change the filtering rules, as long as the communication does not traverse the access level;
- it allows to reorganize the topology of the core network transparently and without the reconfiguration of the controller, since the only relevant information for the virtualization is the list of access switches, which is expected to be stable across time or at most increase in size.

Nevertheless, this is also the only applicable design, given the filtering rules that Floodlight enforces on the frames causing a packet-in event: if the source MAC address belongs to a device that is unknown to the controller, it is automatically discarded and no flow mod or packet-out is sent to the origin switch. A design with virtual address translation on both the user and server sides, involving the destination address when a request to the service is made and the source one when a response from the service is issued, would fail, because any response packet would be discarded when traversing the switches in-between the two access sides. In fact, this kind of packets would be encapsulated in frames having the virtual MAC address of the service as source MAC address, which cannot be discovered and recognized by the controller.

Given these considerations, three possible classes of packets can be identified: the ones representing requests to the service transiting through an access switch, the ones representing responses from the service transiting through an access switch and the ones traversing the core switches, irrespectively of being requests or responses. These classes are treated separately in the following subsections.

2.4.1 Requests to the service transiting through an access switch

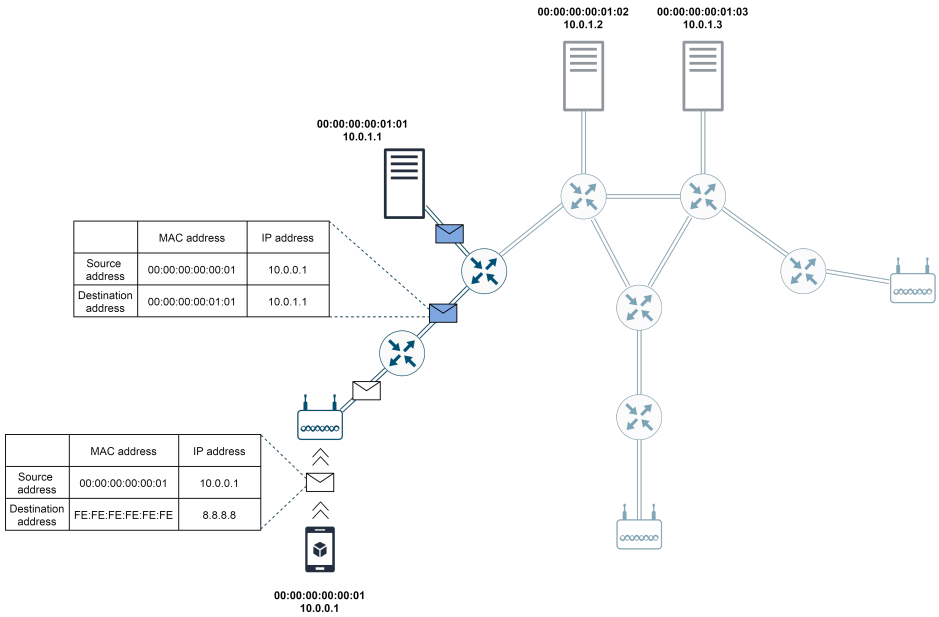

Figure 4: Translation of the virtual address of the service at the access switch during a request.

This class of packets is directed to the virtual address of the service, hence it needs a translation of the destination address, which must be replaced with the one of the topologically closest available server.

To identify the closest host, the *DeviceManager* and *RoutingManager* services provided by Floodlight are exploited. In particular, the entire list of registered servers is consulted and, for each one, the list of attachment points to the network, in the form of switch-port couples, is retrieved. Then, all the possible paths between the current access switch and the end switches attached to the servers are evaluated in terms of number of hops, selecting the shortest one: if multiple servers can be reached with the minimum number of hops, a load balancing mechanism chooses the server whose address has been translated less times.

If a server fails or it does not exist a traversable path to reach it, the Floodlight services will detect the state of the network and provide consistent results, ensuring a transparent adaptation of the translation mechanism; the same happens in case of user mobility between the different access points of the network. There is still the possibility that the services are contacted during a transient phase, when the known topology is not up to date: in this case, packet loss is inevitable and its management is delegated to the transport or application layer.

Once the server has been chosen, the controller sends a flow mod to the access switch, specifying the instructions on how to perform the translation, and a packet out to be forwarded towards the previously identified shortest path. From this point on, the request to the service will traverse the core network having destination MAC and IP addresses equal to the real ones of a server.

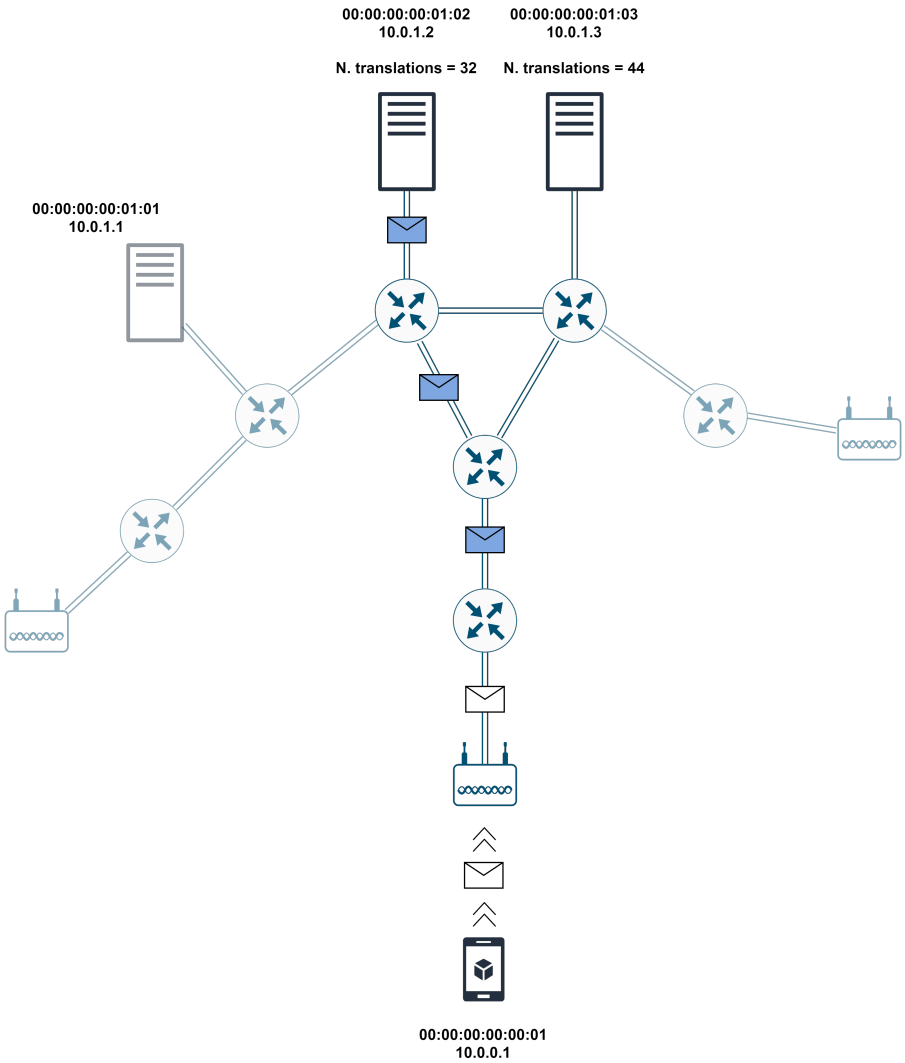

Figure 5: Load balancing in the translation of the virtual address of the service during a request.

2.4.2 Responses from the service transiting through an access switch

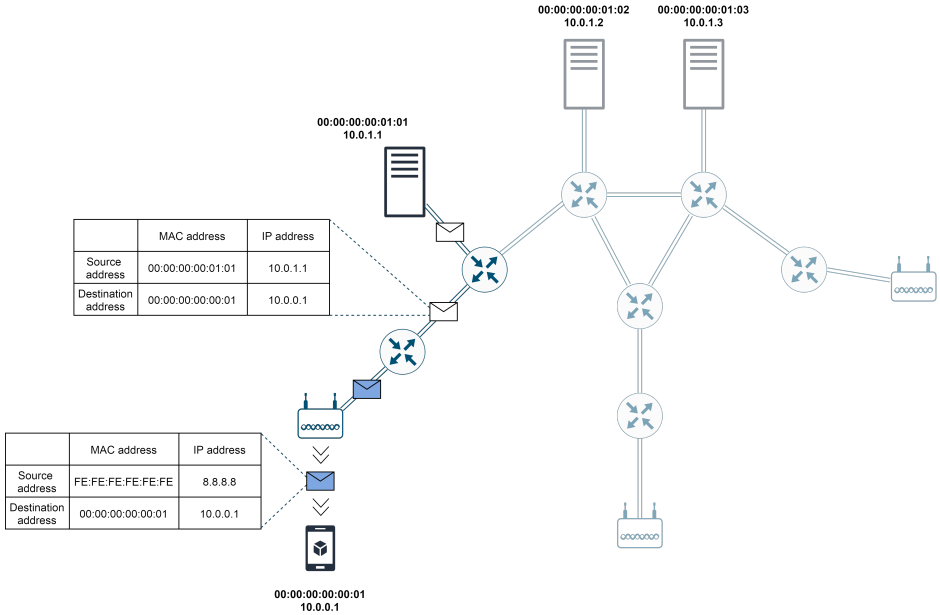

Figure 6: Translation of the virtual address of the service at the access switch during a response.

Suppose that the forwarding of a request towards a server and of a response towards a user is correctly implemented in the core network, where both the source and destination addresses are always relative to real devices. Then, the virtualization of a response coming from a server, namely the translation of the source address into the virtual address of the service, must be done in the access switch, being the last SDN-enabled network device through which the response transits before arriving to the user.

This time, the task of the controller is easier: it must query the *DeviceManager* to find the attachment points of the user and, among them, select only the ones that identify the access switch that sent the packet-in; then, a packet-out and a flow mod, specifying the translation of the source and the forwarding through a port, are sent.

In a mobility scenario, it could happen that the access switch receiving the response is not anymore an attachment point for the specific user, if the user moved through the network and previously installed flow mods did not expire: in this case, the packet is simply discarded, delegating again the management of the loss to the transport or application layer.

2.4.3 Packets traversing the core switches

Since the virtualization of the service location is done in the access switches, the packets transiting through the core network have addresses relative to the real devices: they don't need special forwarding strategies, except for the constraint of being forwarded through the shortest path.

These operations are already offered by Floodlight by means of the *Forwarding* module, which exploits the Dijkstra's algorithm and whose flow mod timeouts can be configured, thus respecting the needed requirements; moreover, it is also efficient, because it installs rules proactively in the switches along the path that the packet will follow. The proactive behaviour works well when it concerns core switches, but it leads to an unwanted management of the translation, if it involves the access switches: when the server sends a response to the user and this packet is received by the first server-side switch, the *Forwarding* module will install flow mods in all the switches in the path towards the user, hence also in her access switch. When the latter will receive the packet, it will not contact the controller and it will not learn how to translate the source address into the virtual address of the service, but it will immediately forward the packet to the user, since a rule matching the packet is installed in its flow table. Therefore, the packet will arrive to the user with a plain source address, breaking the virtualization mechanism.

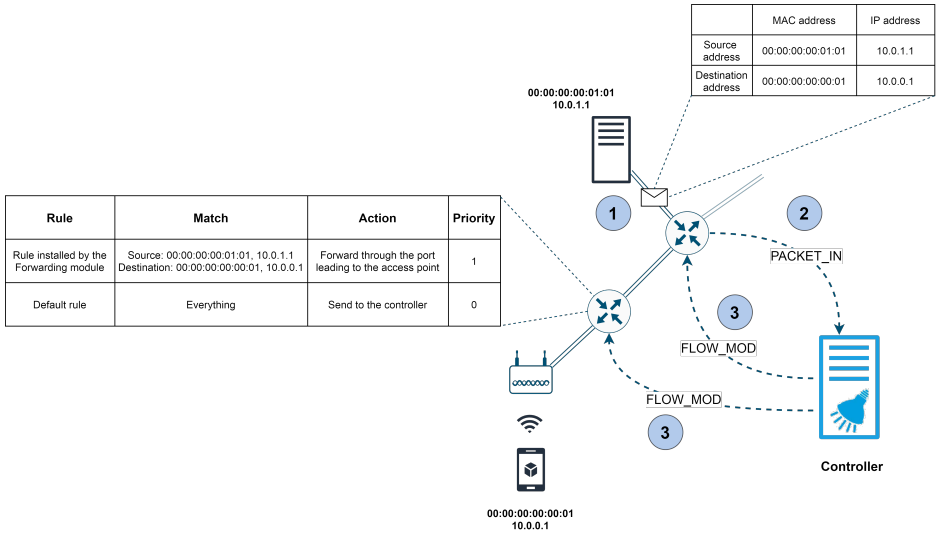

Figure 7: Proactive installation of rules done by the *Forwarding* module.

This unwanted behaviour can be prevented managing the priority of the rules installed in the access switches: if the default rule, specifying that the packet should be sent to the controller when no match is found, is set with a priority higher than the one used by the *Forwarding* module, the proactive rules will be ignored and never applied. To reach this goal, it is sufficient to increase the priority of the default rule to a value higher than 1, when a switch is registered as an access switch via the REST interface; conversely, the priority is set back to 0 when the switch is removed from the list of access switches.

An additional detail must be taken into consideration to make the solution fault tolerant: when a switch

connects to the network, its flow table is flushed and, if OpenFlow 1.3+ is used, a default rule with priority 0 is installed during the handshake with the controller. Suppose a previously registered access switch fails: after its reconnection, it will never receive a flow mod to change the priority of the default rule to 1, unless additional actions are taken. To avoid this behaviour, there are at least three possible fixes, depending on the needs of the organization:

- manually remove the switch from the list of access switches when it fails and register it again when it is reconnected, using the REST interface. In the scenario of this project, it is assumed that the failure of a switch is detected and appropriately managed by the administrator, meaning that, when the switch fails, it stays in the disconnected status until further actions are taken; moreover, it is supposed that a working switch does not cyclically move between the connected and disconnected status on its own. If the switch allows external programs to run under the control of its OS or detection programs probing the status of the switches are used, the procedure could be automatized;
- modify the Floodlight module managing the handshake between the controller and the switches, so that it knows about the distinction between core and access ones, for example synchronizing its data structures with a database storing the information. This is the most robust solution, since it eliminates the root of the problem and works even with switches changing status repeatedly, but also the most complex. Easier solutions relying on the notifications of modules implementing the *IOFSwitchListener* interface are not reliable, because there is no guarantee that the actions following the notification of a new switch connection will be executed after the handshake protocol has finished: if the custom actions are executed before, the priority would be overwritten and set to 0;
- set up an external script or a timer inside the controller to refresh the default rule of the access switches periodically, where the period could be tuned based on parameters and characteristics of the switches provided by the manufacturer. This solution could be a good compromise between the first two approaches.

It should be noted that installing both the translating rules for the request and the response at the same time is not a robust solution, because it works only assuming that the user sends a request and receives a response through the same access switch, and that a response arrives always before the timeout of the associated rule expires. In a congestion scenario involving switches, servers and the controller, or in a mobility scenario, these assumptions are likely to not be met.

3. Testing

The module is implemented using Floodlight 1.1 and tested emulating a network topology with the support of Mininet-WiFi; the chosen topology is the one of Figure 8, where OpenFlow 1.3 is used as southbound protocol.

The controller configuration, injected using the REST interface, is such that:

- the service is reachable at the default virtual address FE:FE:FE:FE:FE:FE, 8.8.8.8;
- $sta1$, $sta2$ and $sta3$ are subscribed, while $sta4$ is not subscribed;
- the registered access switches are $s1$, $s3$ and $s5$;
- the servers providing the service are $h1$, $h2$ and $h3$.

The access points are non SDN-enabled and instantiated so that client isolation is enforced: users connected to the same access point cannot communicate directly without transiting through an access switch. All the tests are made after the involved devices have been discovered by the controller.

In the following, only the most important functional tests are reported and detailed.

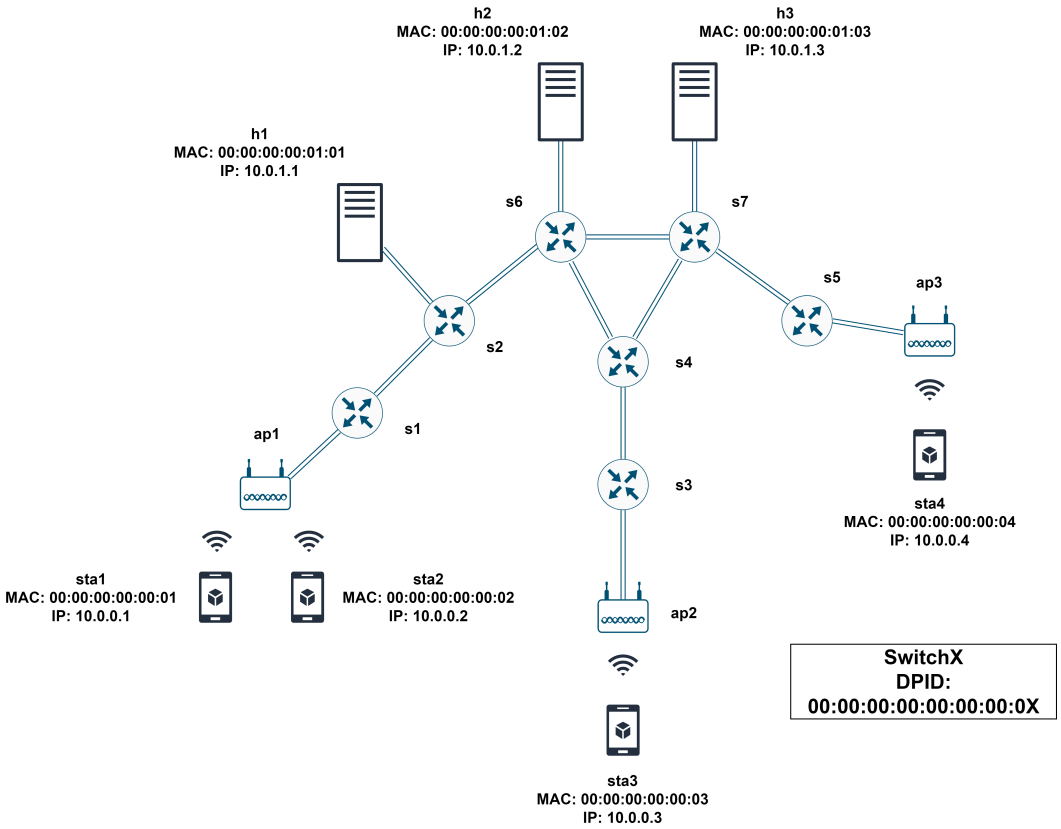

Figure 8: Reference topology used for testing.

3.1 Ping from a user towards the virtual address

The subscribed user $sta1$ and the not subscribed user $sta4$ are considered. A ping to the virtual address of the service is attempted: the one issued by $sta1$ succeeds, while the one issued by $sta4$ fails, since the access switch discards the ARP requests and the IP address is not resolved.

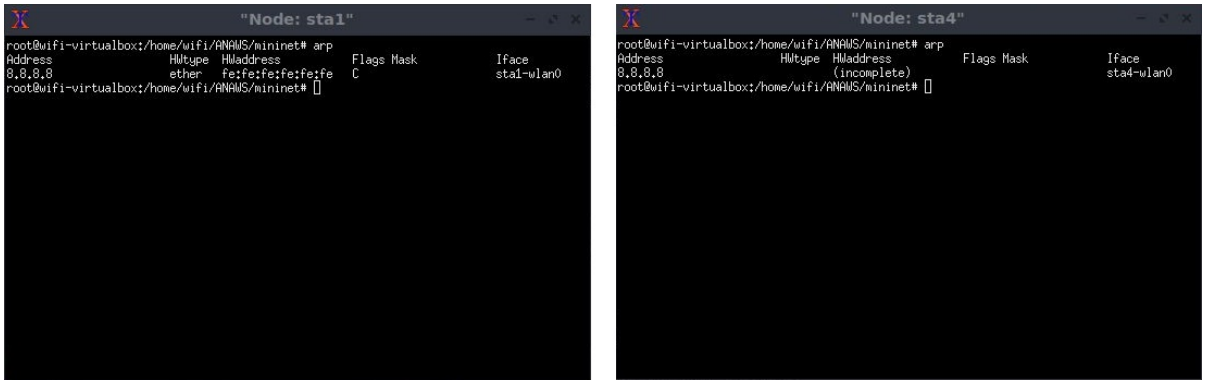

Figure 9: ARP tables of $sta1$ and $sta4$ after the test.

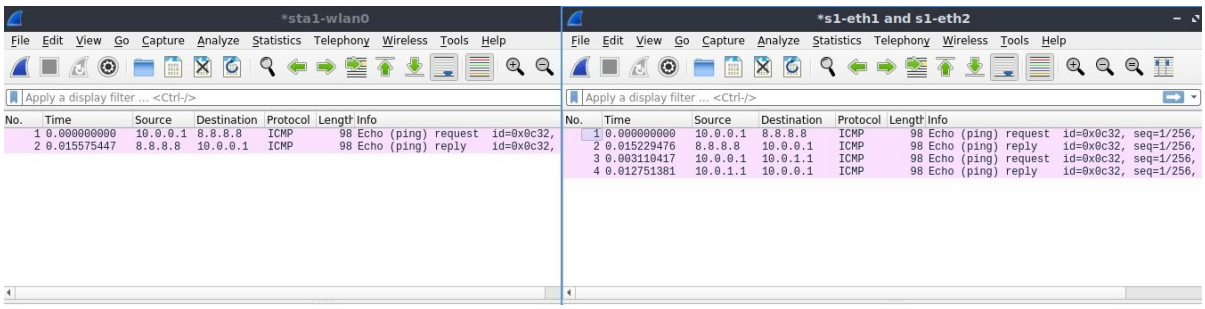

Figure 10: Translation of the virtual address during the ping made by $sta1$.

The test is repeated after removing $sta1$ from the list of subscribed users, but before the expiration of the entries in the ARP table. Even if the user knows how to resolve the virtual IP address, the packet is discarded at the access switch.

3.2 Ping from a user towards other users or generic addresses

The subscribed user $sta1$ and the not subscribed user $sta4$ are considered. Pings to the subscribed user $sta2$, to the server $h3$ and to the address 9.9.9.9 are attempted: the procedures fail at the ARP level, since the access switch discards the ARP requests and the IP addresses are not resolved.

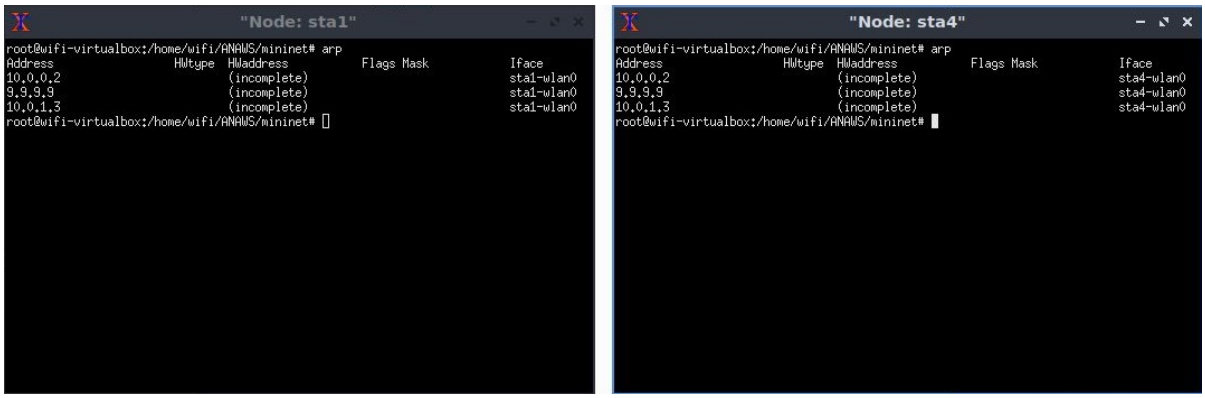

Figure 11: ARP tables of $sta1$ and $sta4$ after the test.

3.3 Ping from a server towards a user or another server

The server $h2$ is considered. Pings to the subscribed users $sta1$ and $sta3$, to the not subscribed user $sta4$ and to the server $h3$ are attempted.

For what concerns the address resolution via ARP, the MAC addresses of all the involved nodes are correctly learnt, even in the case of a not subscribed user.

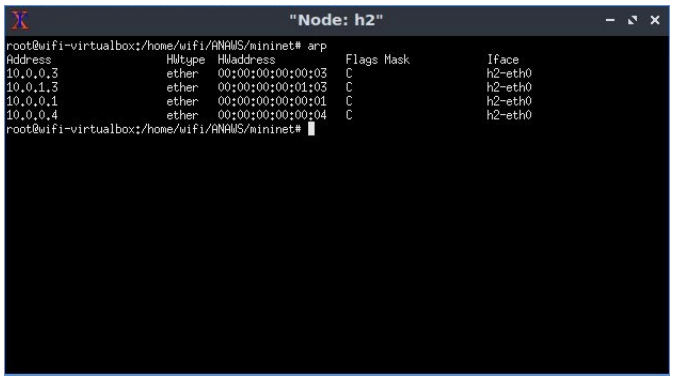

Figure 12: ARP table of $h2$ after the test.

The ICMP echo requests sent to $sta1$ and $sta3$ arrive at the users, but with source address equal to the virtual address of the service, due to the translation done at the access switches. This means that the users are able to reply, but the response is sent to the closest server and not necessarily to $h2$. In particular, the ping to $sta1$ fails, because the ICMP echo reply is forwarded to $h1$, while the ping to $sta3$ can succeed or fail depending on the load balancing ($h2$ and $h3$ can be reached with equal cost). If $h2$ and $h3$ have the same number of translations and the flow of ping requests is continuous, the command returns a mix of successful and failed attempts.

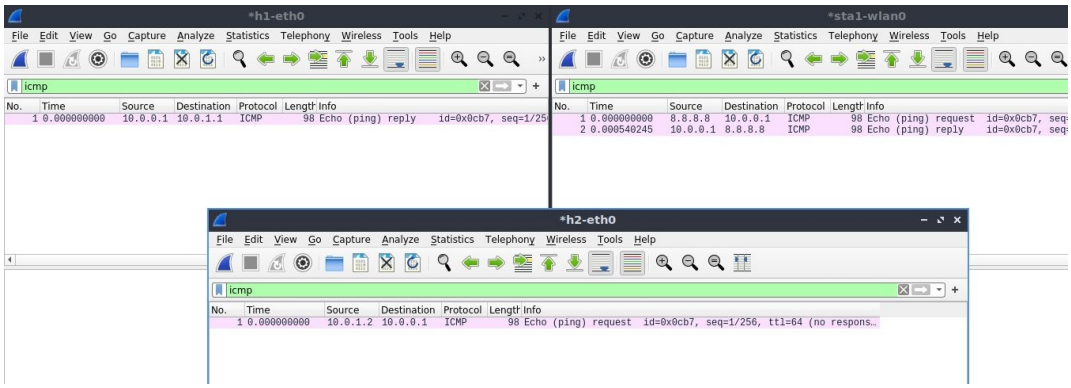

Figure 13: The flow of packets during the ping sent by $h2$ to $sta1$: the ICMP echo reply is redirected to $h1$.

The ping sent to $sta4$ is not successful, since the latter is not subscribed and cannot reply, while a ping sent to $h3$ is always successful, because packets between servers flow normally.

3.4 Requests fulfilled by the closest server

The subscribed user $sta1$ and a client-server application simulating a video streaming over HTTP, with frames requested individually using the GET method, are considered. The server application is hosted at $h1$, $h2$ and $h3$, and returns, together with the frames, the real IP address of the host. The test consists in the following steps:

1. at the beginning, $h1$ and $h3$ are up and waiting for GET requests, while $h2$ is disconnected from the network;
2. $sta1$ runs the client application, specifying the virtual address of the service as the destination address. It starts receiving frames from the closest server $h1$;
3. $h1$ is disconnected from the network. After a transient state, the flow of frames is provided by $h3$;
4. $h2$ is connected to the network. After a transient state, the flow of frames is provided by $h2$;
5. $h1$ is connected again to the network. After a transient state, the flow of frames is provided by $h1$.

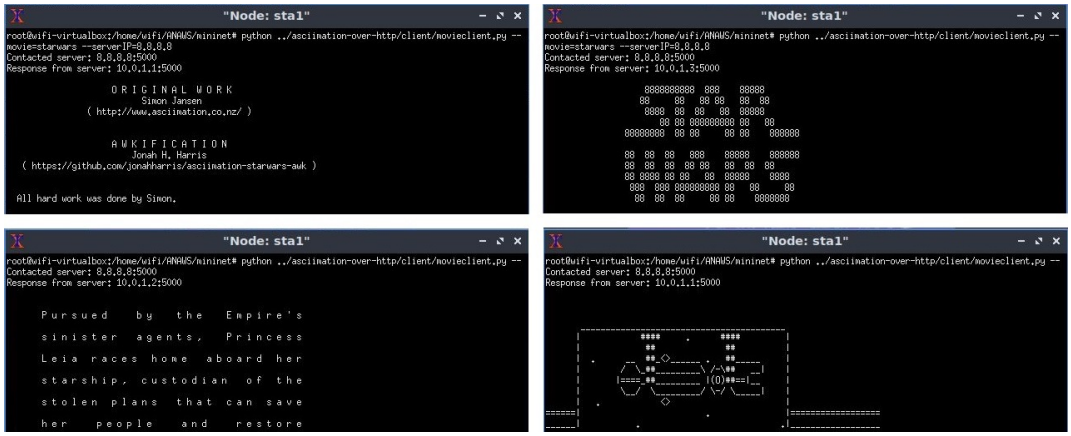

Figure 14: The output of the video streaming application during the test.

3.5 Load balancing with multiple equal cost paths

The subscribed user $sta3$ and the client-server application simulating a video streaming are considered. The application is hosted by the nodes $h2$ and $h3$, while $sta3$ runs the client application specifying the virtual address of the service as the destination address. The frames are provided alternatively by the two servers, where the change of the node is triggered by the expiration of the flow mods.

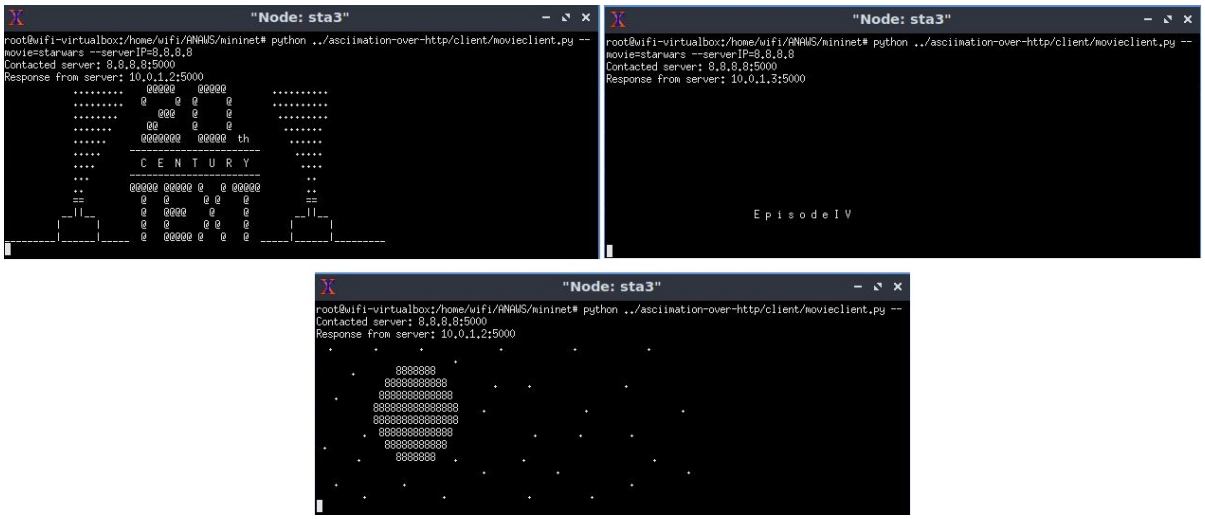

Figure 15: The output of the video streaming application during the load balancing test.

3.6 Failure of a switch port

The subscribed user $sta3$ and the client-server application simulating a video streaming are considered. The test consists in the following steps:

1. at the beginning, $h1$, $h2$ and $h3$ are up and waiting for GET requests;
2. $sta3$ runs the client application, specifying the virtual address of the service as the destination address. It starts receiving frames alternatively from the closest servers $h2$ and $h3$;
3. the interface $s4 - eth2$, through which $s4$ can reach $s7$, is shut down. After a transient state, the flow of frames is provided by $h2$ only, which becomes the closest server;
4. $h2$ is disconnected from the network. After a transient state, the flow of frames is provided alternatively by $h1$ and $h3$, which are now the closest servers. It should be noted that, until the number of translations of $h1$ does not reach the number of translations of $h3$, the frames are provided by $h1$ only.

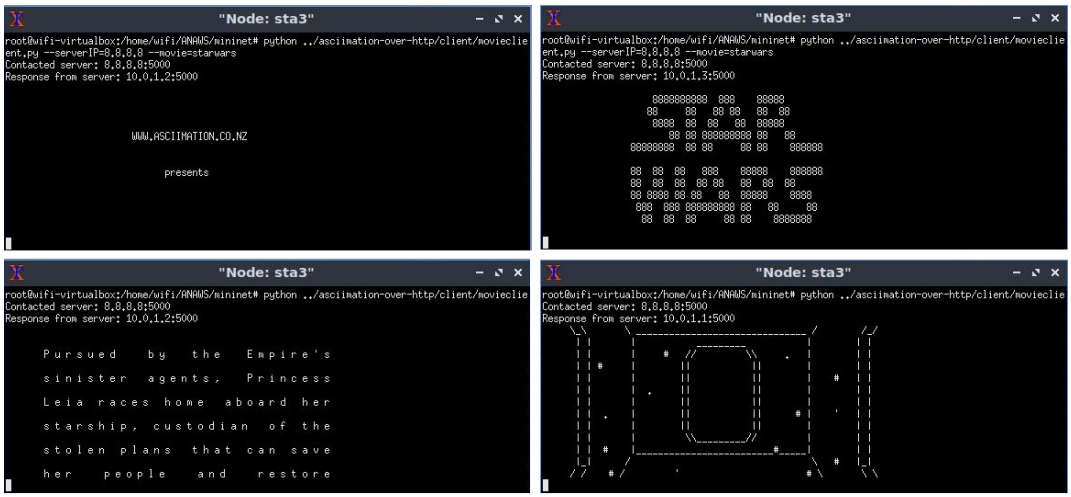

Figure 16: The output of the video streaming application during the test.

3.7 Mobility scenario

The subscribed user $sta1$ and the client-server application simulating a video streaming are considered. $sta1$ runs the client application specifying the virtual address of the service as the destination address, while the server application is hosted by the nodes $h1$ and $h2$. At the beginning of the emulation, $sta1$ moves from an initial point near $ap1$ towards an end point near $ap2$, connecting to the access point offering the strongest signal.

The result is that the flow of frames, initially provided by $h1$, begins to be provided by $h2$ as soon as the switch of devices is completed and the controller learns the new position of the user.

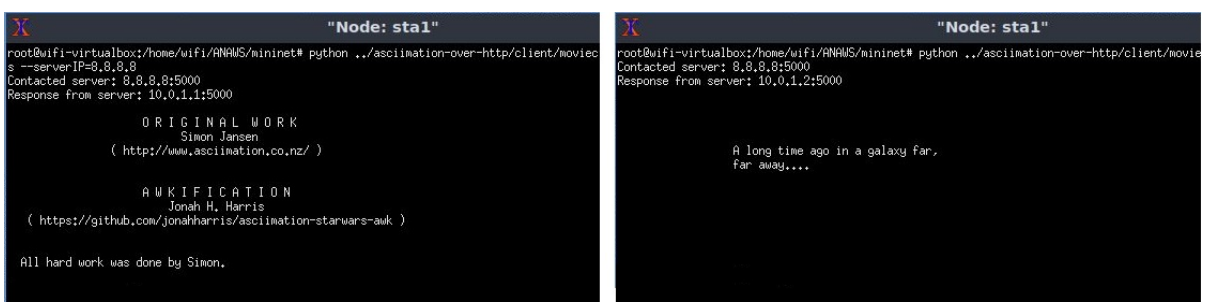

Figure 17: The output of the video streaming application during the mobility test.