

**PRÁCTICA 4**  
**Integración de API REST de Controlador Ryu con página Web e**  
**Integración de Balanceador de carga con página Web**



**Universidad  
del Cauca**

Presentado por:

***FEDERICO FRANCISCO ALBERTO BENITEZ GONZALEZ***  
***JUAN JOSE PAREDES ROSERO***  
***SANTIAGO FELIPE YEPES CHAMORRO***

Presentado a:

***Ing. Oscar Mauricio Caicedo***

**UNIVERSIDAD DEL CAUCA**  
**INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES**  
**LABORATORIO 2 DE SISTEMAS DE TELECOMUNICACIONES**  
**POPAYÁN, COLOMBIA**  
**JULIO-2019**

**Parte1: Conexión de API REST de Controlador Ryu con página Web**

El primer punto de esta práctica consiste en la creación de una página web que sirva de interfaz para que un usuario pueda crear y eliminar reglas de control de tráfico que bloqueen la comunicación entre dos host dentro de una red, especificando sus direcciones MAC y las horas del día dentro de las cuales funcionará dicha condición.

### Introducción:

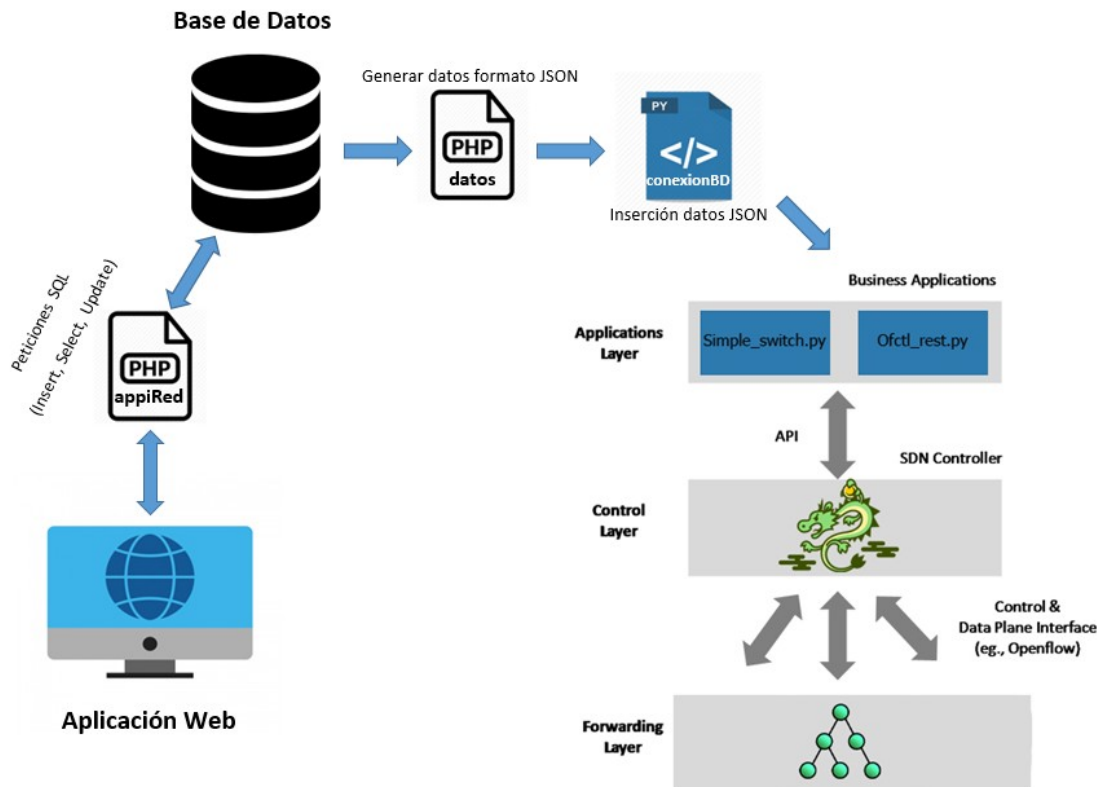


Figura 1: Arquitectura para el desarrollo de la actividad.

Como se muestra en la figura 1, para cumplir con el requerimiento planteado se realiza una arquitectura que permita la comunicación entre la aplicación web y el plano de datos, este procedimiento se efectúa paso a paso por diferentes puntos que explicaremos a continuación:

- **Aplicación web:** es la encargada de crear y mostrar al cliente, las reglas que permiten el control del tráfico. El cliente tiene la posibilidad de añadir por medio de un formulario la hora de inicio, hora de fin, mac de origen y mac de destino.
- **appiRed.php:** es el código PHP que permite generar las consultas SQL entre la aplicación web y la base de datos, ejecutando 3 consultas: `SELECT` para obtener las reglas que ya se encuentran en la base de datos, `INSERT` para añadir las nuevas reglas dadas en el formulario y `UPDATE` para realizar las veces de eliminar (esto será explicado en el procedimiento).

- **Base de datos:** almacena en una tabla las diferentes datos que necesitamos para la ejecución de nuestro proyecto.
- **datos.php:** genera un array tipo JSON con la información de todos los datos en la tabla, este array puede ser leído si se accede al link donde se encuentra este archivo.
- **conexionBD:** se considera a este script uno de los más importantes en la arquitectura, debido a que el se encarga de leer el JSON generado en el archivo anterior para luego realizar la lógica necesaria para administrar la red. El envía a la API\_REST de Ryu los flujos que el cliente añadió en la aplicación web, esto lo hace por medio de una solicitud HTTP tipo POST que contiene los flujos para lograr la nueva regla. Una vez enviados ella seguirá analizando los cambios en la base de datos, esto se hace en un intervalo de un minuto.
- **Arquitectura SDN:** la arquitectura SDN tiene 3 planos en los que se ejecutan las siguientes cosas:
  - Plano de Aplicación: cuando se establece el controlador Ryu el iniciara con dos APIS cruciales, `simple_switch` y `ofctl_rest`.
  - Plano de Control: en este plano se ejecuta el controlador remoto Ryu.
  - Plano de Datos: en este plano se ejecuta una topología tipo árbol con una profundidad igual a 3 y un fanout igual a 3.

### Procedimiento:

Inicialmente se ejecutó un servidor Web junto con un motor de base de datos MySQL, utilizando la herramienta XAMPP, debido a que esta herramienta facilita tanto la instalación como el uso de este tipo de software ya que incluye un servidor Apache y una base de datos MariaDB.

Una vez corriendo el servidor Web Apache es posible acceder a la página web creada a través del link <http://localhost/enfasis3/appRed.php> con se muestra en la figura 2. Esta página contiene unos campos de ingreso que le permiten a un usuario especificar direcciones MAC de dos hosts, de destino y de origen, entre los cuales se desea que no exista comunicación, bien sea de forma permanente, es decir hasta que se elimine la regla, o de forma temporal para lo cual tiene que especificar un intervalo de horas. De igual forma permite al usuario mirar todas las reglas activas existentes en la base de datos a través de una tabla en su parte izquierda.

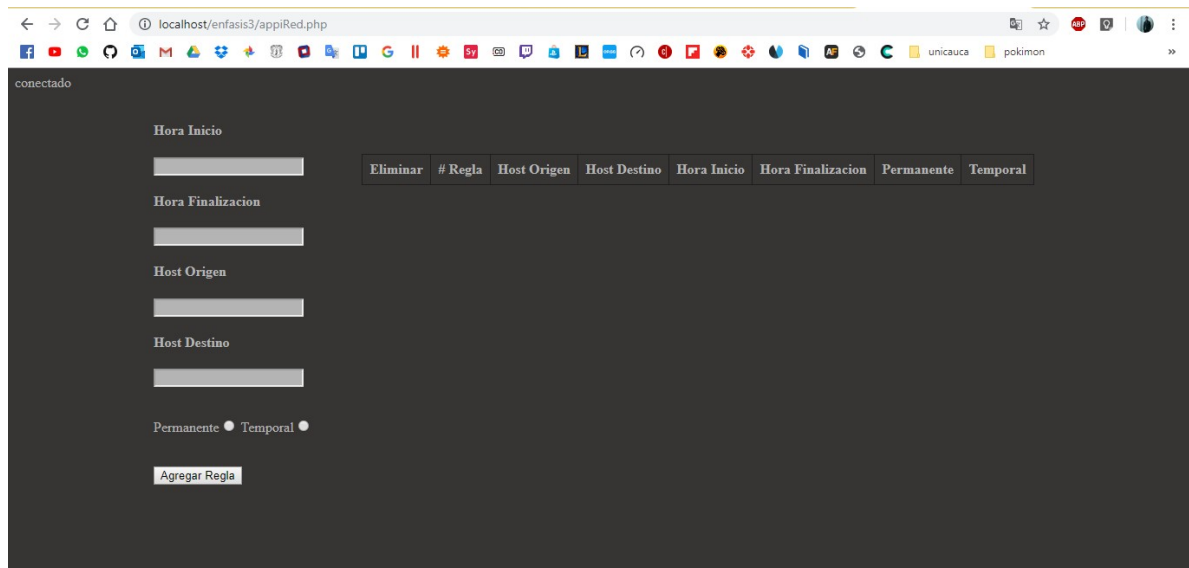


Figura 2: Página Web para creación y verificación de reglas por parte del usuario.

Los datos ingresados por el usuario y mostrados a él son almacenados en una base de datos MariaDB conectada al motor de gestión de base de datos MySQL. Esta base de datos cuenta con una tabla llamada **reglas**, la cual contiene las siguientes columnas: **id**, autoincremental, **horalnicio**, **horaFin**, **hostOrigen**, **hostDestino**, **permanente**, **temporal**; estos dos últimos utilizados para definir si la regla se aplicada durante determinado tiempo o permanentemente y por último la columna **activación**, usada para diferenciar reglas que están activas de reglas inactivas o que no deberían tenerse en cuenta. La estructura de esta base de datos se puede evidenciar en el figura 3.

	id	horalnicio	horaFin	hostOrigen	hostDestino	permanente	temporal	activacion
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	77	11:00:00	12:00:00	00:00:00:00:00:03	00:00:00:00:00:0a	0	1	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	78	00:00:00	00:00:00	00:00:00:00:00:02	00:00:00:00:00:14	1	0	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	79	23:00:00	23:55:00	00:00:00:00:00:01	00:00:00:00:00:1b	0	1	1

Figura 3: Estructura de la tabla "reglas" de la base de datos.

En la figura 3 se pueden ver registros existentes guardados en dicha tabla, realizados con fines de ensayo desde la página web y a su vez utilizados para comprobar que, efectivamente se puedan observar la reglas creadas en su sección correspondiente, como se puede ver en la figura 4. Cada regla tiene asociado un botón "**Eliminar**", el cual envía una solicitud hacia los archivos php del servidor, quienes procesan la solicitud modificando el

campo **activación** de la respectiva regla, asignándole un valor de cero, sin eliminarla, lo cual hace que sea deshabilitada impidiendo su uso por el controlador y evitando que se muestre en la página web.

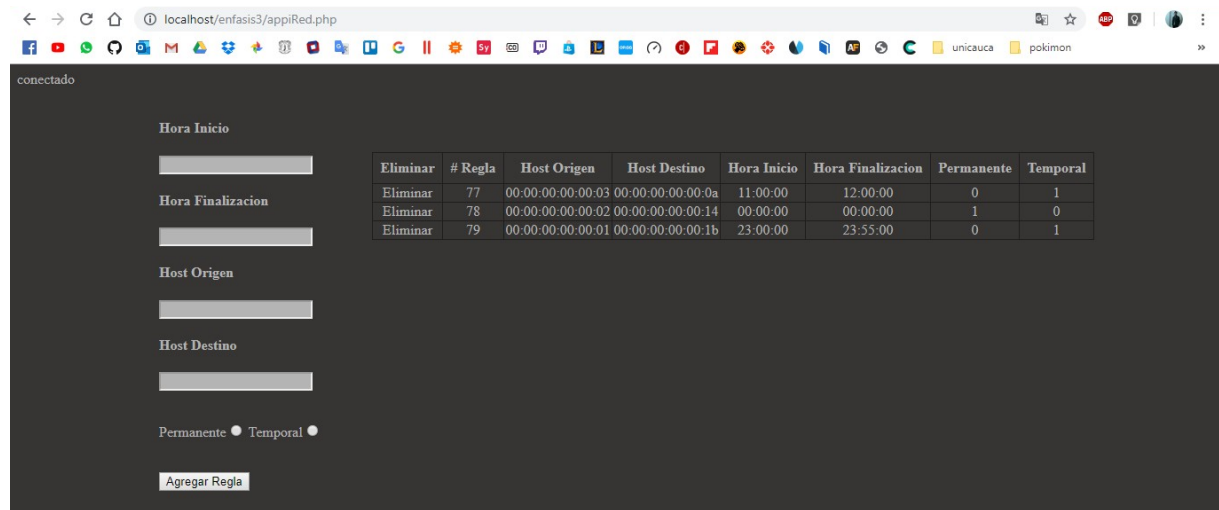


Figura 4: Página web con registros cargados desde la base de datos.

Una vez se tiene la página web funcionando correctamente al igual que la parte BackEnd en el servidor encargada de la interacción con la base de datos, entonces se puede proceder a la creación del script python *conexionBD.py*, el cual se puede observar completo en la figura 6. Este archivo junto con *datos.php* son los mediadores de la comunicación entre API REST de Ryu controller. El funcionamiento de este script se detalla a continuación:

Inicialmente se realiza una solicitud HTTP tipo GET hacia el servidor web, específicamente al archivo *datos.php* (figura 5), el cual devolverá un respuesta con un archivo JSON que contiene las reglas almacenadas en la base de datos. Este archivo se almacena como un Diccionario en la variable llamada *todos*. (Líneas 5 y 6).

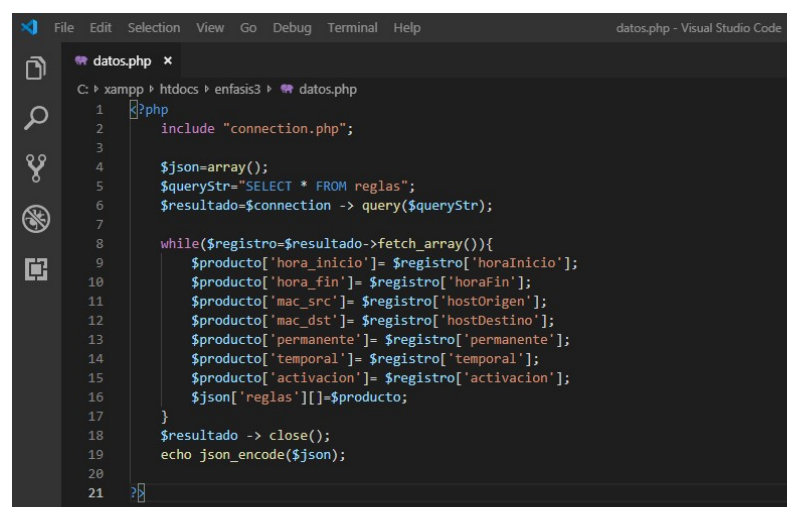


Figura 5: Script php para devolver un archivo JSON.

```

1 import json
2 import requests
3 from datetime import datetime as timeN
4 from datetime import date, time, timedelta
5 response = requests.get("http://192.168.0.12/enfasis3/datos.php")
6 todos = json.loads(response.text)
7
8
9 #nuevojson = open("/home/usuario/Escritorio/eljson.txt", "w")
10
11 for dat in todos["reglas"]:
12     if str(dat["activacion"]) == "1":
13         #nuevojson.write(str(dat["mac_src"])+ " "+str(dat["mac_dst"])+ " "+str(dat["hora_inicio"])+ " "+str(dat["hora_fin"])+ "\n")
14
15     fuente = str(dat["mac_src"]);
16     #00:00:00:00:00:00
17     fuente = int(fuente[15:],16)
18
19     if(fuente<4):
20         switch=3
21     elif(fuente<7):
22         switch=4
23     elif(fuente<10):
24         switch=5
25     elif(fuente<13):
26         switch=7
27     elif(fuente<16):
28         switch=8
29     elif(fuente<19):
30         switch=9
31     elif(fuente<22):
32         switch=11
33     elif(fuente<25):
34         switch=12
35     else:
36         switch=13
37
38     if int(dat["temporal"]) == 1:
39         #---
40         horaIni = dat["hora_inicio"];
41         horaIni = int(horaIni[0:2]);
42         horaFin = dat["hora_fin"];
43         horaFin = int(horaFin[0:2]);
44         #---
45         horaIni = time(int(dat["hora_inicio"][0:2]),int(dat["hora_inicio"][3:5]),0)
46         horaFin = time(int(dat["hora_fin"][0:2]),int(dat["hora_fin"][3:5]),0)
47         horaAct = time(timeN.now().hour,timeN.now().minute,0)
48
49         #if horaIni<time.now().hour and horaFin<time.now().hour:
50         if horaIni<horaAct and horaFin>horaAct:
51             headers = {"GET HTTP/1.1 "
52                         "Content-Type": "application/json",
53                         "cache-control": "no-cache"
54                     }
55             data = {'dpid':switch,
56                     'cookie':42,
57                     'priority':50000,
58                     'match':{
59                         'dl_dst':dat["mac_dst"],
60                         'dl_src':dat["mac_src"]
61                     },
62                     'actions':[]
63                 }
64             data_json = json.dumps(data)
65             url = "http://192.168.0.59:8080/stats/flowentry/add"
66             response = requests.post(url, headers = headers, data = data_json)
67
68             print("el switch es: "+str(switch)+"\nmac origen: "+dat["mac_src"]+
69                   "\nla hora inicio es: "+dat["hora_inicio"]+
70                   "\nla hora fin es: "+str(horaIni)+
71                   "\nla hora del sistema es : "+str(timeN.now().hour))
72         else:
73             headers = {"GET HTTP/1.1 "
74                         "Content-Type": "application/json",
75                         "cache-control": "no-cache"
76                     }
77             data = {'dpid':switch,
78                     'cookie':42,
79                     'priority':50000,
80                     'match':{
81                         'dl_dst':dat["mac_dst"],
82                         'dl_src':dat["mac_src"]
83                     },
84                     'actions':[]
85                 }
86             data_json = json.dumps(data)
87             url = "http://192.168.0.59:8080/stats/flowentry/delete_strict"
88             response = requests.post(url, headers = headers, data = data_json)
89             print("Regla eliminada")
90         else:
91             headers = {"GET HTTP/1.1 "
92                         "Content-Type": "application/json",
93                         "cache-control": "no-cache"
94                     }
95             data = {'dpid':switch,
96                     'cookie':42,
97                     'priority':50000,
98                     'match':{
99                         'dl_dst':dat["mac_dst"],
100                         'dl_src':dat["mac_src"]
101                     },
102                     'actions':[]
103                 }
104             data_json = json.dumps(data)
105             url = "http://192.168.0.59:8080/stats/flowentry/add"
106             response = requests.post(url, headers = headers, data = data_json)
107             print("Regla permanente agregada")
108         else:
109             fuente = str(dat["mac_src"]);
110             #00:00:00:00:00:00
111             fuente = int(fuente[15:],16)
112
113             if(fuente<4):
114                 switch=3
115             elif(fuente<7):
116                 switch=4
117             elif(fuente<10):
118                 switch=5
119             elif(fuente<13):
120                 switch=7
121             elif(fuente<16):
122                 switch=8
123             elif(fuente<19):
124                 switch=9
125             elif(fuente<22):
126                 switch=11
127             elif(fuente<25):
128                 switch=12
129             else:
130                 switch=13
131
132             headers = {"GET HTTP/1.1 "
133                         "Content-Type": "application/json",
134                         "cache-control": "no-cache"
135                     }
136             data = {'dpid':switch,
137                     'cookie':42,
138                     'priority':50000,
139                     'match':{
140                         'dl_dst':dat["mac_dst"],
141                         'dl_src':dat["mac_src"]
142                     },
143                     'actions':[]
144                 }
145             data_json = json.dumps(data)
146             url = "http://192.168.0.59:8080/stats/flowentry/delete_strict"
147             response = requests.post(url, headers = headers, data = data_json)
148             print("Regla eliminada")
149         #nuevojson.close();

```

Figura 6: Script python conexionBD.py

Posteriormente, con ayuda de un ciclo se extrae cada una de las reglas contenidas en el vector "**reglas**" del JSON obtenido (línea 11). Para cada una de estas reglas se realiza una comprobación del campo **activación**, si la regla está habilitada(1) debe ser enviada a la API REST del controlador para que sea agregada a los switches correspondientes. El switch al cual será añadida la regla se determina verificando el host de origen (Líneas 20 a 37).

A continuación, se compara el campo **temporal** de la regla, si este tiene un valor de 1 (Línea 39) entonces es necesario extraer la Hora de inicio (**horalnicio**) y la hora de final (**horaFinal**), para verificar si la hora actual del sistema se encuentra en ese rango (Línea 52). Si se cumple esta condición, se construye la regla que bloquea el tráfico entre los respectivos hosts y se la envía a la API REST del controlador para que la aplique al switch correspondiente, en caso contrario (Línea 77) se envía una solicitud a la API REST del Ryu para que elimine la misma regla, permitiendo así el tráfico en las horas que estén fuera del rango establecido.

En caso de que el campo **temporal** tenga un valor de 0, lo cual indica que será permanente (Línea 100), por lo tanto únicamente se enviará la solicitud a la API REST del controlador para que agregue la regla a la tabla de flujo del switch respectivo. Esta regla permanecerá operando en el switch hasta que el usuario la deshabilite desde la página web, es decir hasta que su campo **activación** tome un valor de 0, para este caso (Línea 122) se enviará una solicitud a la API REST del controlador para que elimine dicha red.

Los procesos de **añadir o de eliminar una regla** de flujo se realizan enviando archivos JSON hacia la API REST del controlador contenidos en solicitudes HTTP POST, cuyos valores son definidos dependiendo del caso. Cuando se desea añadir una regla, inicialmente se definen los headers de la solicitud como se muestra en la figura 7:

```
headers = {"GET HTTP/1.1 "
           "Content-Type": "application/json",
           "cache-control": "no-cache"
          }
```

Figura 7: Definición de headers de solicitud POST.

De igual forma se define el archivo json correspondiente al cuerpo de la solicitud como se muestra en la siguiente figura 8:

```
data = {'dpid':switch,
        'cookie':42,
        'priority':50000,
        'match':{
            'dl_dst':dat["mac_dst"],
            'dl_src':dat["mac_src"]
        },
        'actions':[]
       }

data_json = json.dumps(data)
```

Figura 8: Definición de parámetros de configuración del flujo y parse a JSON.

Los parámetros del diccionario **data** son definidos de acuerdo a la regla que se desee añadir y siguiendo la documentación del controlador RYU. El atributo **dpid** corresponde al id o



número del switch que se desea configurar, **priority** es un valor necesario para este caso, el cual le otorga una prioridad más alta a la regla que se está añadiendo haciendo que el switch la elija esta por encima de la demás; el vector **match** define todos los campos que se desea que coincidan, en este caso solo se usan las direcciones MAC de origen y destino. Por último se encuentra el vector **actions** el cual especifica las acciones que debe ejecutar el switch cuando los headers de un paquete coincidan con los valores especificados en el vector **match**.

Una vez definidas todas las partes de la solicitud POST se procede a enviarla, dependiendo del proceso que se desee realizar: si se desea añadir la solicitud se envía al link mostrado en la figura 9:

```
url = "http://192.168.0.59:8080/stats/flowentry/add"  
  
response = requests.post(url, headers = headers, data = data_json)
```

*Figura 9: Envío de solicitud POST para añadir regla.*

En caso que se desee eliminar la solicitud en el vector data se deben especificar los mismos campos, esto con el fin de ser más específicos y facilitar la comparación de las reglas dentro del switch, de modo que se evite la eliminación de una regla equivocada. El proceso se muestra en la figura 10:

```
url = "http://192.168.0.59:8080/stats/flowentry/delete_strict"  
  
response = requests.post(url, headers = headers, data = data_json)
```

*Figura 10: Envío de solicitud POST para eliminar regla.*

A pesar de que el anterior script funciona correctamente, se tenía la necesidad de implementar un mecanismo que permita **automatizar** la ejecución del mismo, de modo que se esté monitorizando los cambios sobre la base de datos que el usuario realice desde la página web como la creación o inhabilitación de alguna regla.

Como solución al problema anteriormente mencionado se optó por el uso del demonio **Crontab** el cual es un archivo que posee cada usuario de un sistema Linux, el cual contiene una lista de comandos que se ejecutan en tiempos establecidos por el usuario, con los cuales también se pueden correr scripts como el mencionado anteriormente.

Para hacer uso de este método, inicialmente se debe acceder al archivo **crontab** del usuario para editar su contenido. Esto se realiza como se muestra en la figura 11:



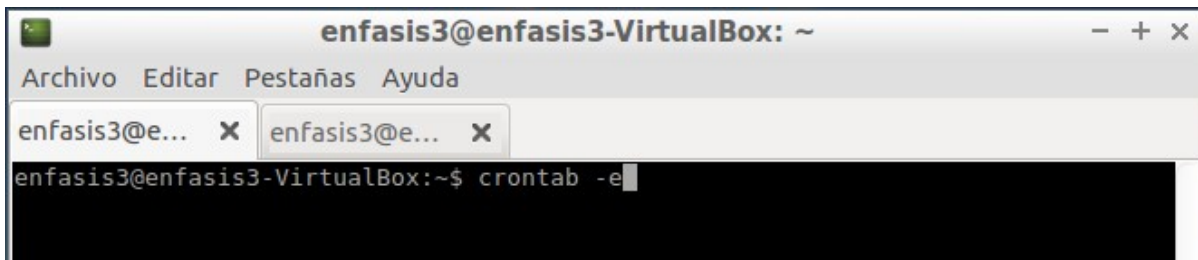


Figura 11: Acceso al archivo Crontab.

Una vez se ejecuta este comando, el sistema preguntará por el editor de preferencia para modificar el archivo. Con la ayuda de ese editor se podrán agregar al final de dicho archivo todos los comandos que se deseen ejecutar automáticamente, especificando al comienzo la cantidad de minutos, horas, días, meses y año que dicen al sistema la periodicidad con la que se desea ejecutar un determinado comando. Para nuestro caso se consideró adecuado realizar la ejecución del script cada minuto, lo anterior se logra escribiendo el comando como se muestra en la figura 12. El funcionamiento del comando se complementa agregando al final de la línea parámetros que permiten crear un archivo llamado *fichero.log*, el cual contendrá todos los errores que ocurran al momento de ejecutarse.

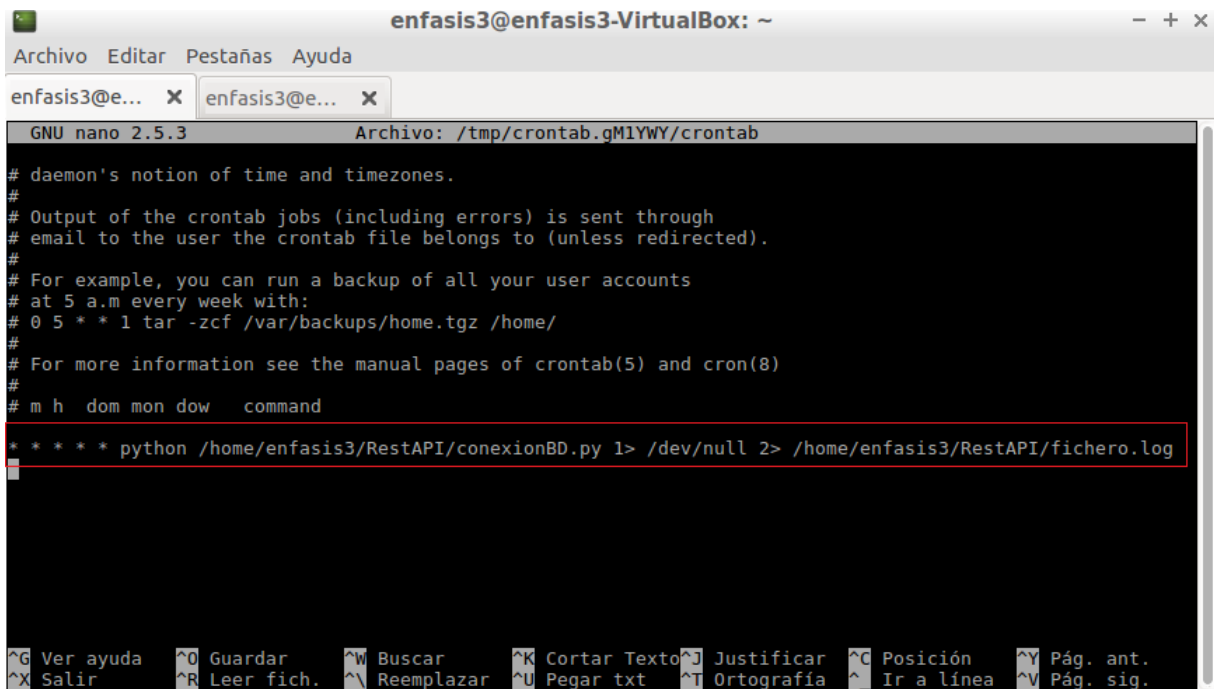


Figura 12: Comando de automatización para ejecución del script.

Con el archivo Crontab editado correctamente se procede a presionar las teclas Ctrl + O para guardar y Ctrl + X para salir del editor. Al realizar este proceso se muestra un mensaje en consola indicando que se inicializó dicho archivo, como se muestra en la figura 13.

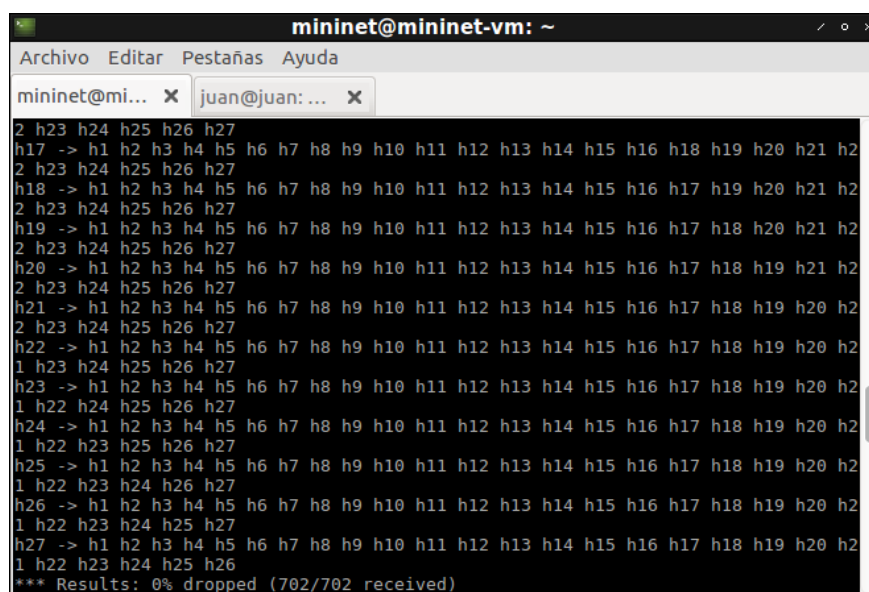
```
enfasis3@enfasis3-VirtualBox:~$ crontab -e
crontab: installing new crontab
enfasis3@enfasis3-VirtualBox:~$
```

Figura 13: Mensaje de confirmación de ejecución de archivo Crontab.

Una vez el script `conexionDB.py` se esté ejecutando por medio del CRON de linux las actualizaciones que se hagan en la base de datos se comprobarán cada minuto, por lo que pasamos a comprobar que las reglas definidas desde la aplicación se estén ejecutando y se estén cumpliendo.

Ejecutaremos el comando que inicia la topología fat tree en mininet especificando el controlador remoto con su respectiva IP, en este momento la red no podrá realizar ningún ping, esto se debe a que el controlador Ryu no se está ejecutando, para ello debemos lanzar el comando: `ryu-manager ryu.app.simple_switch ryu.app.ofctl_rest`, en este comando se ejecutarán dos aplicaciones.

Una es el `simple_switch` que me define las tablas de flujo de los host que realicen ping, por lo tanto si se ejecuta el comando: `pingall` el controlador definirá en cada switch de la topología la tabla de flujo correspondiente, esto se puede observar en la figura 14 donde se ejecuto el comando `pingall` y todos los pings fueron realizados con éxito.

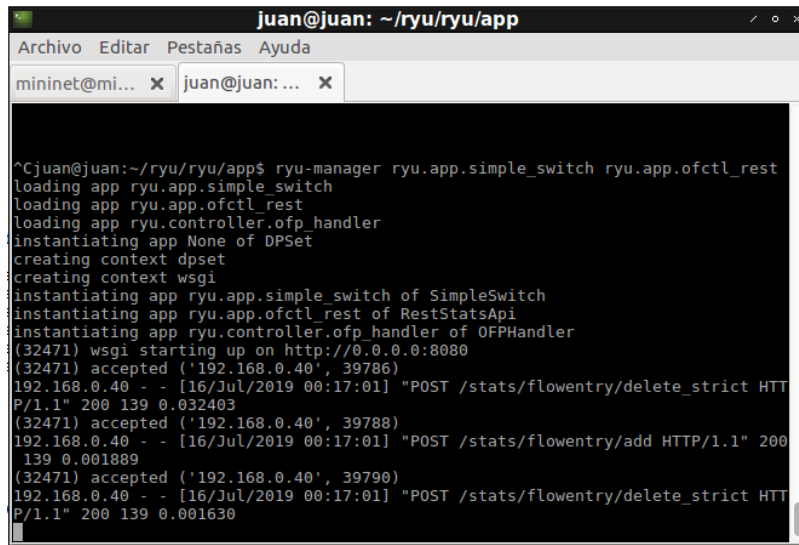


```
mininet@mininet-vm: ~
Archivo Editar Pestañas Ayuda
mininet@mi... x juan@juan: ... x
2 h23 h24 h25 h26 h27
h17 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h18 h19 h20 h21 h2
2 h23 h24 h25 h26 h27
h18 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h19 h20 h21 h2
2 h23 h24 h25 h26 h27
h19 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h20 h21 h2
2 h23 h24 h25 h26 h27
h20 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h21 h2
2 h23 h24 h25 h26 h27
h21 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
2 h23 h24 h25 h26 h27
h22 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h23 h24 h25 h26 h27
h23 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h24 h25 h26 h27
h24 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h25 h26 h27
h25 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h24 h26 h27
h26 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h24 h25 h27
h27 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h24 h25 h26
*** Results: 0% dropped (702/702 received)
```

Figura 14: Ejecución Pingall con su resultado.

La otra aplicación es `ofctl_rest.py`, esta proporciona una funcionalidad REST que permite en cualquier momento añadir o quitar una tabla de flujo en los switch de la topología. Recordemos que `ofctl_rest.py` tiene un contacto directo con el script `conexionBD.py`, el cual enviará, si se cumplen sus condiciones, nuevos flujos a esta API que se encargará de notificar al controlador, y este se encargará de notificar a los switch que se encuentra una nueva regla.

Por lo tanto se procede a ejecutar comando mientras el archivo `conexionBD.py` siga ejecutandose por medio del CRON. (figura 15).

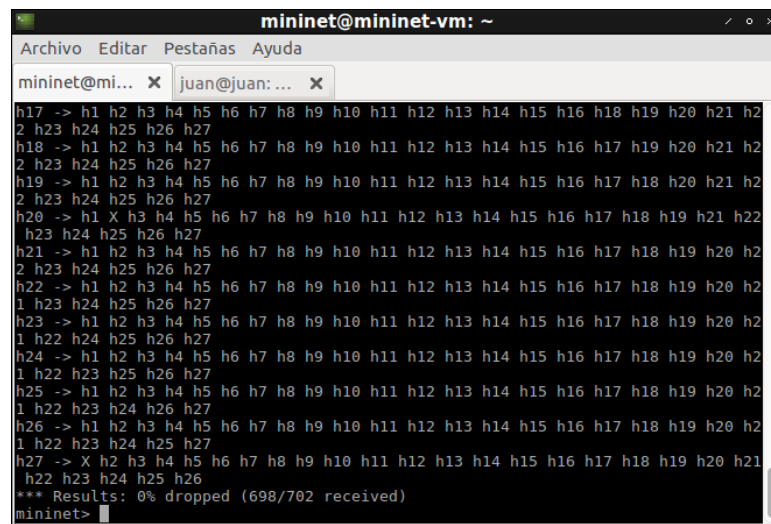


```
^Cjuan@juan:~/ryu/ryu/app$ ryu-manager ryu.app.simple_switch ryu.app.ofctl_rest
loading app ryu.app.simple_switch
loading app ryu.app.ofctl_rest
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.simple_switch of SimpleSwitch
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.controller.ofp_handler of OFPHandler
(32471) wsgi starting up on http://0.0.0.0:8080
(32471) accepted ('192.168.0.40', 39786)
192.168.0.40 - - [16/Jul/2019 00:17:01] "POST /stats/flowentry/delete_strict HTTP/1.1" 200 139 0.032403
(32471) accepted ('192.168.0.40', 39788)
192.168.0.40 - - [16/Jul/2019 00:17:01] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.001889
(32471) accepted ('192.168.0.40', 39790)
192.168.0.40 - - [16/Jul/2019 00:17:01] "POST /stats/flowentry/delete_strict HTTP/1.1" 200 139 0.001630
```

Figura 15: Ejecución controlador con las dos aplicaciones para dar soporte a la red.

Como se puede notar en la figura 15 el comando lanzará el controlador que creará un mini servidor en la máquina donde se realizó la ejecución. La ip de esta máquina está definida en *conexionBD.py*, esta ip es utilizada para realizar el envío de los flujos. Por lo que podemos ver en la figura el controlador recibe 3 solicitudes HTTP del tipo POST desde la IP 192.168.0.40 la cual contiene dos reglas, la primera es *flowentry/add* y la segunda *flowentry/delete\_strict*; que añaden un flujo o lo eliminan respectivamente.

Solo nos queda comprobar que estos nuevos flujos estén actuando en la red, por lo volvemos a ejecutar el comando: *pingall*, y comprobaremos que si este realizando las restricciones. (ver figura 16, se observa que solo se recibieron 698 de 702 pings).



```
mininet@mininet-vm: ~
mininet@mi... X juan@juan: ... X
h17 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h18 h19 h20 h21 h2
2 h23 h24 h25 h26 h27
h18 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h19 h20 h21 h2
2 h23 h24 h25 h26 h27
h19 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h20 h21 h2
2 h23 h24 h25 h26 h27
h20 -> h1 X h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h21 h22
h23 h24 h25 h26 h27
h21 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
2 h23 h24 h25 h26 h27
h22 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h23 h24 h25 h26 h27
h23 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h24 h25 h26 h27
h24 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h25 h26 h27
h25 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h24 h26 h27
h26 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h2
1 h22 h23 h24 h25 h27
h27 -> X h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21
h22 h23 h24 h25 h26
*** Results: 0% dropped (698/702 received)
mininet>
```

Figura 16: Comando pingall ejecutando las reglas.

## Parte 2: Conexión de Balanceador de carga con página Web

Para el siguiente punto se requiere que por medio de una página web, se genere el balanceo de carga de una red, la cual se le va a establecer dos servidores web y una ip

virtual, encargada de recibir solicitudes y distribuirlas de forma equitativa, y así generar un balance, guiados con el artículo “Dynamic Load Balancing using Software Defined Networks” y con la herramienta *LoadBalancer.py* que es proporcionada en la página:

<https://github.com/exploitthesystem/Ryu-SDN-Load-Balancer/blob/master/LoadBalancer.py>, se permite lanzar como un manejador ryu, dentro de el, posee ya las especificaciones tanto de la ip virtual como de los servidores, como se puede apreciar a continuación:

```
2
3 class LoadBalancer(app_manager.RyuApp):
4     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION] # Specify the use of OpenFlow v13
5     virtual_ip = "10.0.0.10" # The virtual server IP
6     ## Hosts 5 and 6 are servers.
7     H5_mac = "00:00:00:00:00:05" # Host 5's mac
8     H5_ip = "10.0.0.5" # Host 5's IP
9     H6_mac = "00:00:00:00:00:06" # Host 6's mac
10    H6_ip = "10.0.0.6" # Host 6's IP
11    next_server = "" # Stores the IP of the next server to use in round robin manner
12    current_server = "" # Stores the current server's IP
13    ip_to_port = {H5_ip: 5, H6_ip: 6}
14    ip_to_mac = {"10.0.0.1": "00:00:00:00:00:01",
15                "10.0.0.2": "00:00:00:00:00:02",
16                "10.0.0.3": "00:00:00:00:00:03",
17                "10.0.0.4": "00:00:00:00:00:04"}
```

Figura 17: Variables definidas en el LoadBalancer.py

El código presentado, instancia de manera global las variables que regirán el funcionamiento del balanceador, dando aquí, la limitación que se requiere superar por medio de la práctica, para este caso, basándose en el funcionamiento de la parte uno, se utiliza un despliegue similar, pero se resalta una característica clave que dará la oportunidad de dinamizar la red sin requerir de una herramienta externa, como lo es crontab. Cuando se realiza una solicitud a la ip virtual, desde cualquiera de los host, que no se encuentran definidos como servidores, estos van requerir de un primer manejo por el controlador(*LoadBalancer.py*), y este punto se puede aprovechar al surgir del evento *packet\_In*, dando la instancia a donde ser enrutado y así guardar, en el switch, el flujo concerniente, ya que cada uno va a estar definido particularmente para cada ip virtual que sea definida; simplificando el problema aún más, solo se hará en función de una única regla que variará para cada caso, ahorrando procesamiento a cambio de limitar la variedad.

La página web queda de la siguiente manera:



Figura 18: Página web para el balanceador de carga

Como se aprecia solo se tiene tres características establecidas, ya que con esto se va a generar el manejo de los demás factores que definen el enrutamiento en la red, al generar un nuevo servicio (definase servicio como aquella aplicabilidad que van a prestar los servidores web, y que requiere en este caso de una ip virtual y dos servidores), pero solamente se va modificar el ya establecido.

Ahora, por parte del funcionamiento del Load balancer se le agrega el siguiente código:

```
def packet_in_handler(self, ev):

    response = requests.get("http://192.168.0.12/enfasis3/crut/datos.php")
    todos = json.loads(response.text)

    serv=todos["servicio"][0]
    self.virtual_ip = str(serv["ipVirtual"])

    #print(serv["ipServidorUno"][7:8]+"\n\n")

    self.H5_ip = str(serv["ipServidorUno"])
    self.H5_mac = "00:00:00:00:00:0"+str(serv["ipServidorUno"][7:8])

    self.H6_ip = serv["ipServidorDos"]
    self.H6_mac = "00:00:00:00:00:0"+str(serv["ipServidorDos"][7:8])

    ip_to_port = {self.H5_ip: int(serv["ipServidorUno"][7:8]), self.H6_ip: int(serv["ipServidorDos"][7:8])}

    self.ip_to_port.setdefault(self.H5_ip,int(serv["ipServidorUno"][7:8]))
    self.ip_to_port.setdefault(self.H6_ip,int(serv["ipServidorDos"][7:8]))

    if self.cambio == 0:
        self.next_server = self.H5_ip
        self.current_server = self.H5_ip
        self.cambio=1
    else:
        self.next_server = self.H6_ip
        self.current_server = self.H6_ip
        self.cambio=0

    for i in (1,2,3,4,5,6):
        if (int(serv["ipServidorUno"][7:8])!=i and int(serv["ipServidorDos"][7:8])!=i):
            self.ip_to_mac.setdefault("10.0.0."+str(i),"00:00:00:00:00:0"+str(i))
```

Figura 19: Código dinamizador del LoadBlancer.py

Para esta parte, al no requerir de un elemento como el contrab, se a realizado la solicitud de forma directa en el controlador, por lo que nos da la ventaja que no va a ver necesidad de una actualización periódica, ya que cada vez que se envíe un paquete a una ip virtual nueva está va a ser la misma ocurrencia de entrada a la actualización del funcionamiento deseado

en la red; por lo que por medio de un parseo básico, se obtienen los datos obtenidos en el json como se mostró en la primera parte, y luego cada uno es asignado a las variables globales (no se les cambió el nombre a dichas variables para evitar corromper el código y alterar funcionamiento general que si se desea mantener).

Por otro lado, en consideración a que cada ocurrencia de un evento va a realizar la solicitud a la base de datos requiriendo que se le entregue el enrutamiento inicial, pero ya que este debe cambiar por cada solicitud, se tiene el problema que no existe un cambio de servidor para la primera solicitud, por lo que se solucionó con un cambio de host por medio de una bandera, la cual analiza un primer estado y luego intercala los hosts de los servidores para realizar el balanceo, como se muestra a continuación:

```
if self.cambio == 0:
    self.next_server = self.H5_ip
    self.current_server = self.H5_ip
    self.cambio=1
else:
    self.next_server = self.H6_ip
    self.current_server = self.H6_ip
    self.cambio=0
```

Figura 20: Condicional para intercalar los paquetes.

La red utilizada es:

```
sudo mn --topo=single,6 --controller=remote,ip=x.x.x.x --switch ovs,protocols=OpenFlow13 --mac --arp
```

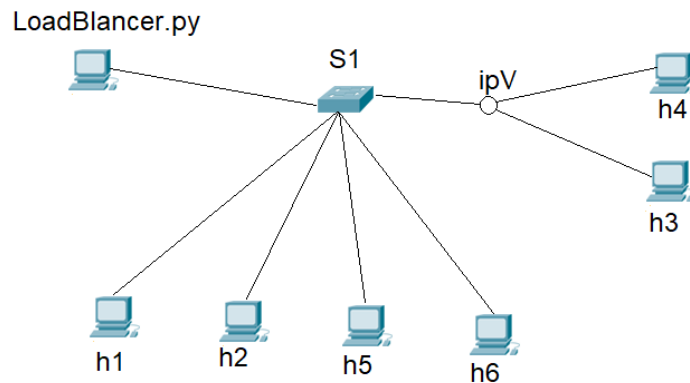


Figura 21: Red de pruebas del LoadBalancer.py

Se definen los servidores entrando al xterm de cada host, para el ejemplo el h4 y h3, como se ve a continuación se corre el comando **python -m SimpleHTTPServer 80** en ambos para esperar las solicitudes; para el caso se realizan **curl** a la ip 10.0.0.10, y obtener una página web en formato HTML



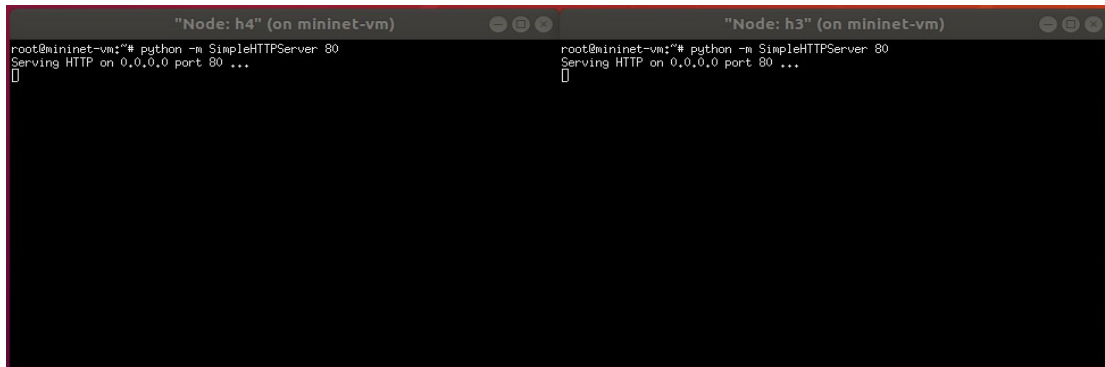


Figura 22: Host h4 y h3 definidos como servidores

una vez ya se han definido los servidores se procede a realizar la petición, y como se ve a continuación, cada host que realiza una solicitud va a obtenerla de un servidor distinto:

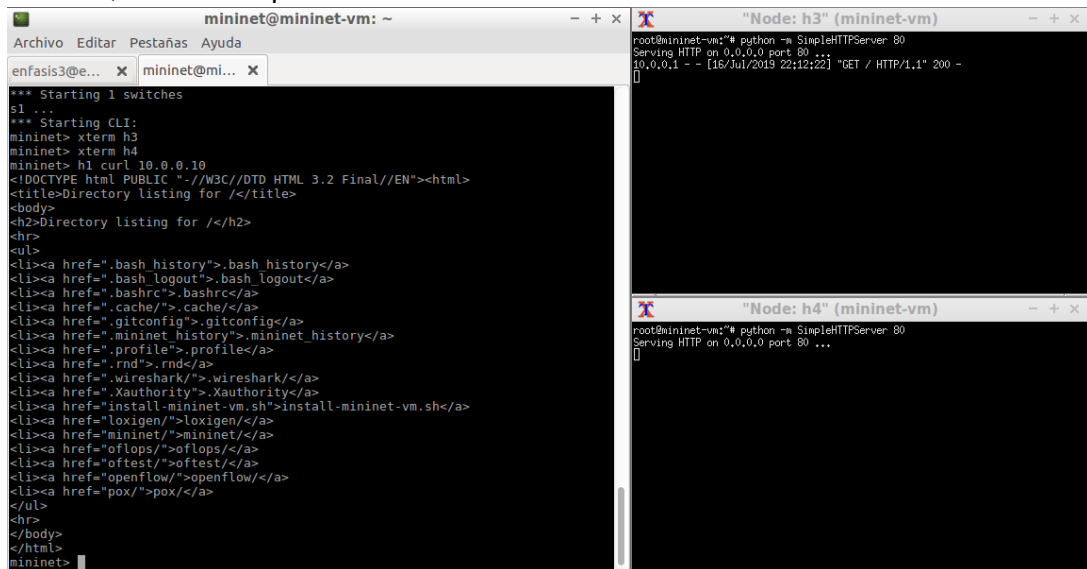


Figura 23: h1 curl 10.0.0.10

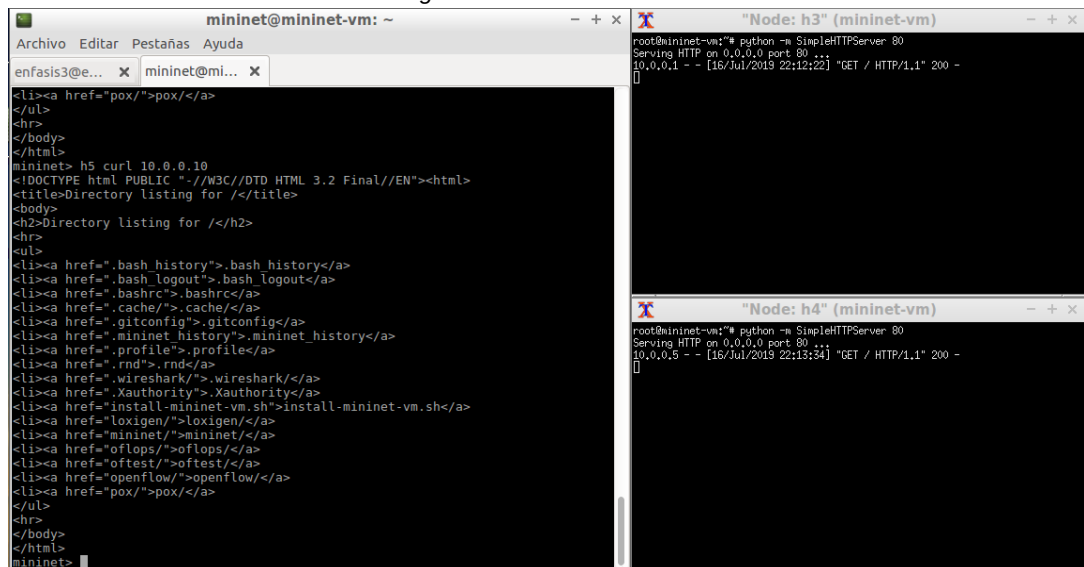


Figura 23: h5 curl 10.0.0.10



## **Conclusiones:**

En esta práctica se ha podido ver la gran utilidad y dinamicidad que permite aplicar la implementación de SDN en una red, ya que gracias a su funcionamiento, conlleva a romper las típicas barreras existentes a la hora de gestionar una red, y permite que dicha gestión, sea más eficiente, y directa respecto a los cambios que se deseen hacer como administrador de red.

La oportunidad de involucrar múltiples tecnologías, como lo son bases de datos, páginas web, lenguaje de servidor (php), python y mininet, demuestra la gran adaptabilidad que obtiene en un manejo estándar de una red SDN, ya bajo condiciones de regímenes antiguos dependería de una constante adecuación directa para un ptimo funcionamiento, mientras que en este caso su implementación entra en una nueva brecha que la define quien decida regir las políticas y condiciones de la red, y cómo van a interactuar estas con otras funciones adicionales que faciliten su uso.