MEMORIA TRABAJO FINAL

Autores: Aitor Martín-Romo González y Patricia Ortuño Otero

CONTENIDO

1. INTRODUCCIÓN

El objetivo de esta práctica es desarrollar y profundizar en la orquestación de funciones de red virtualizadas añadiendo soporte QoS mediante SDN con Ryu y registros de las MACs de la red residencial. Para ello se va a trabajar con dos escenarios emulados en dos máquinas distintas. Este primero, sobre Kubernetes, permite emular las distintas redes y hosts del escenario y el clúster de Kubernetes de la central local. En este escenario, además, se trabajará con el entorno Open vSwitch, que permite emular la red de acceso AccesNet1, la red externa ExtNet1 que ataca directamente al router isp1, y sobre todo, permite su utilización en la emulación del brg1 como en las KNFs. Se trabajará también con VNX, herramienta la cual emulará los equipos de la red residencial, el router isp1 y el servidor s1. Dicho escenario será el anfitrión en todo momento y estará comunicándose con el entorno OSM durante todo el proceso.

Se recomienda la lectura de este documento en el orden propuesto ya que se irá explicando progresivamente en qué ficheros se han realizado modificaciones, los nuevos propuestos para la automatización de funcionalidades previas, y cómo van afectando al escenario global a medida que se van ejecutando.

Partiremos de un repositorio propio que deberá ser clonado en ~/shared, de nombre rdsv-scripts. El documento README.md de este se puede ver el código adjunto en esta entrega. Adicionalmente, la explicación de las imágenes Docker propias utilizadas se puede encontrar en el Anexo I: Contenedores Docker de este documento.

2. ESCENARIO K8S

2.1. PREPARAR ESCENARIO

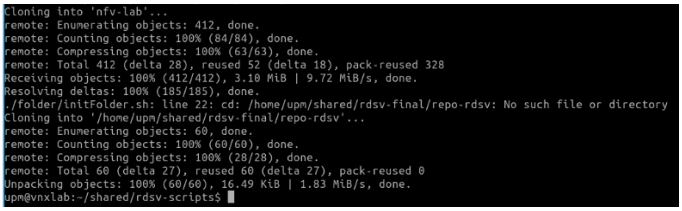
Para poder desplegar el escenario requerido con las redes residenciales esperadas, debemos comenzar clonando los repositorios necesarios para tener todas las funcionalidades y ficheros necesarios.

Para ello se provee de un fichero automatizado ejecutable como `./folder/initFolder.sh`

El contenido de initFolder.sh se puede ver en el código adjunto en esta entrega.

En él, se hace uso de la carpeta rdsv-final para almacenar los subdirectorios y ficheros de nfv-lab que serán útiles para el desarrollo de esta práctica. Como foco de interés para la máquina que alberga K8s, se sustituirán los ficheros con especificaciones XML para la instanciación de todos los equipos del escenario, puesto que se han incluido nuevas etiquetas (se explicarán posteriormente en este documento).

La salida de dicho comando se puede ver a continuación:



Se han creado dos nuevas carpetas ~/shared/nfv-lab y ~/shared/rdsv-final, con los ficheros mencionados.

2.2. ARRANQUE DE LAS MÁQUINAS

Una vez definida la distribución de directorios y ficheros para realizar esta práctica, procedemos al arranque inicial de los equipos de la red residencial, el router y el servidor.

Para ello se provee de un fichero automatizado ejecutable como `./kubernetes/initScenario.sh`

El contenido de initScenario.sh se puede ver en el código adjunto en esta entrega.

Dichos equipos instanciados con VNX han sido provistos con *tags* para la instalación de iperf3, la ejecución de dhclient y la obtención de sus IPs en la interfaz eth1. Se muestran a continuación dichas etiquetas, pero su uso no se verá hasta más adelante en este documento:

nfv3_home_lxc_ubuntu64.xml

```
<exec seq="dhclient_h11" type="verbatim">
  dhclient
</exec>
<exec seq="retrieve_ip_h11" type="verbatim">
  ifconfig | grep 192.168.255
</exec>
<exec seq="iperf3_h11" type="verbatim">
  sudo apt-get update
  sudo apt -y install iperf3 --fix-missing
</exec>
```

Cabe destacar que en la especificación xml para el servidor, también se ha añadido un *tag* que permita instalar iperf3 para la máquina s1.

3. CONECTIVIDAD PARA LAS DOS REDES RESIDENCIALES

3.1. PREPARAR ESCENARIO

Al igual que en K8s, el primer paso a seguir es clonar los repositorios necesarios para tener todas las funcionalidades y ficheros necesarios para desplegar las instancias renes. Se hará uso del mismo fichero automatizado.

En este caso, los comandos de interés son los de clonación de los ficheros que contienen las IPs para la instanciación de ambas redes residenciales: **osm_renes1.sh** y **osm_renes2.sh**. Para la creación de los servicios de red, también es de utilidad clonar (o actualizar) el repositorio público repo-rdsv perteneciente a los autores de esta práctica.

Para más información consultar el apartado “Preparar escenario” de la sección “Escenario K8s”.

3.2. CREAR INSTANCIAS RENES

Para otorgar conectividad a las redes residenciales, debemos comenzar por instanciar dos NS, uno para una de ellas: renes1 y renes2.

Para ello, se ha creado un fichero automatizado ejecutable como `source ./osm/createRenes.sh`

El contenido de createRenes.sh se puede ver en el código adjunto en esta entrega.

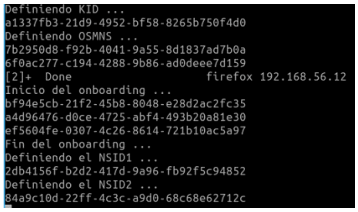
Cabe destacar que se usa el identificador *source* previo a dicho comando ya que, por defecto, un *script* en *bash* se ejecuta en otra terminal, por lo que, al finalizar su ejecución, perderíamos el contenido de variables tales como KID u OSMNS. Ya que estamos interesados en conservarlas, se hace uso de dicho modificador.

Tras la definición del identificador asociado al clúster Kubernetes donde instanciaremos los pods *access* y *cpe*, ejecutamos en segundo plano la interfaz gráfica de OSM, para poder observar la correcta creación de nuestros paquetes e instancias, y poder tener trazabilidad sobre las mismas.

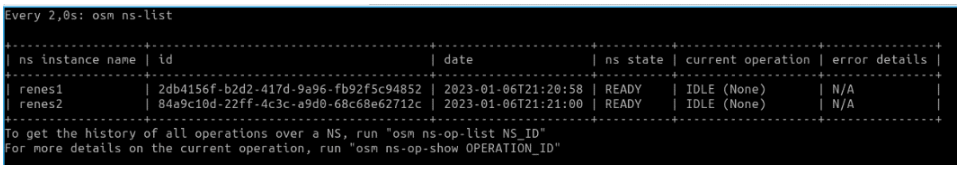
De forma automatizada, se hace uso del repositorio repo-rdsv propio para instanciarlo como base del clúster que está siendo configurado. Gracias a ello, las imágenes para iniciar los pods serán las que hemos subido a Dockerhub.

A continuación, se crean los paquetes VNF para *access* y *cpe*, y el paquete de servicio de red renes. Con ello ya instanciado, procedemos a la creación de los dos servicios de red necesarios: renes1 y renes2. Finalmente, la ejecución de este *script* muestra una pantalla con ambos servicios siendo desplegados, para que el usuario pueda visualizar su progreso, o reiniciar dicho proceso en caso de fallo.

La ejecución de dicho comando muestra los siguientes mensajes por pantalla:



Finaliza mostrando una vista con ambos servicios de red desplegados y listos para utilizar:



Si se accede a la interfaz gráfica mencionada, se puede ver como el *onboarding* de los paquetes y servicios de red ha sido satisfactorio. Se adjuntan imágenes de dicha interfaz gráfica en el Anexo II de este documento.

3.3. DAR CONECTIVIDAD A LAS MÁQUINAS

Una vez desplegadas las instancias de red renes1 y renes2, debemos proceder a la creación de túneles virtuales que permitan a los equipos de la red residencial comunicarse con el exterior y tener, por tanto, acceso a Internet.

Para ello, se ha creado un fichero automatizado ejecutable como `./osm/runRenes.sh`

El contenido de runRenes.sh se puede ver en el código adjunto en esta entrega.

En la preparación del escenario, se clonaron los ficheros **osm_renes1.sh** y **osm_renes2.sh**, correspondientes a la entrega del trabajo de la práctica 4, con las IPs necesarias para definir los *gateways* para otorgar conectividad a ambas redes residenciales.

Adicionalmente, se sustituye el fichero **renes_start.sh** original por otro que contiene las siguientes modificaciones:

```
$ACC_EXEC ip link add vxlanacc type vxlan id 0 remote $HOMETUNIP dstport 4789 dev net1
# En la siguiente línea se ha corregido el dispositivo, que debe ser eth0
$ACC_EXEC ip link add vxlanint type vxlan id 1 remote $IPCPE dstport 8742 dev eth0
$ACC_EXEC ovs-vsctl add-port brint vxlanacc
$ACC_EXEC ovs-vsctl add-port brint vxlanint
$ACC_EXEC ifconfig vxlanacc up
$ACC_EXEC ifconfig vxlanint up
$CPE_EXEC ovs-vsctl add-port brint vxlanint -- set interface vxlanint type=vxlan options:remote_ip=$IPACCESS options:key=1 options:dst_port=8742
```

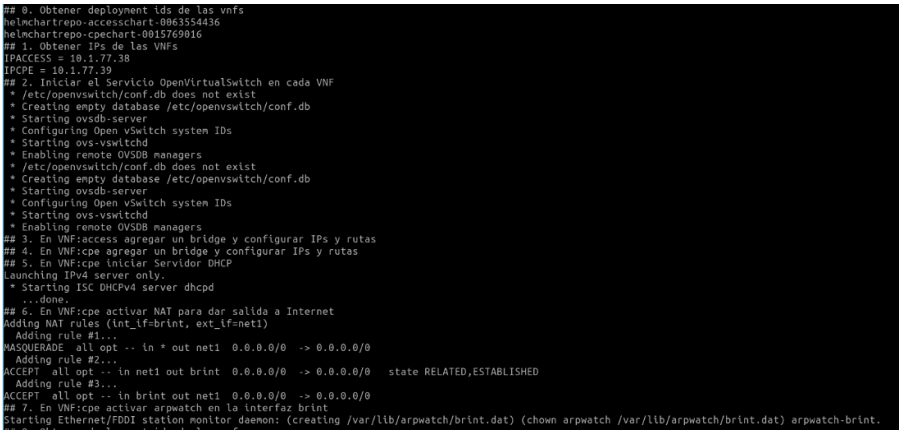
De este modo, los túneles de comunicación se crean a través de comandos de Linux, no de Open vSwitch, para un mejor funcionamiento con QoS. Como se puede ver, tenemos un bridge virtual denominado brint, con dos puertos: vxlanacc y vxlanint. El primero permite la comunicación con la red residencial, mientras que el segundo es para la comunicación interna del clúster de Kubernetes.

Asimismo, definimos que vxlanint tiene un identificador de valor 1 en KNF:access, por lo que, a la hora de desplegar dicha interfaz en KNF:cpe, determinamos una key de valor 1, permitiendo la comunicación en la red interna del clúster.

Adicionalmente, se ha incluido un séptimo paso en renes_start.sh para la ejecución del servicio arpwatch en los pods KNF *cpe*, escuchando en la interfaz brint y almacenando la salida de sus escuchas en el fichero brint.dat:

```
## 7. En VNF:cpe activar arpwatch en la interfaz brint
echo "## 7. En VNF:cpe activar arpwatch en la interfaz brint"
$CPE_EXEC arpwatch -i brint -f brint.dat
$CPE_EXEC etc/init.d/arpwatch start
```

Por tanto, la ejecución de dicho comando tiene la siguiente salida:

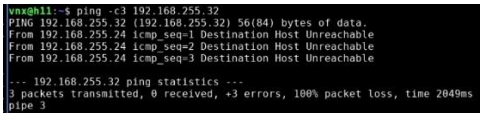


Como se puede ver, adicionalmente a la funcionalidad esperada de creación de túneles VXLAN y definición de reglas NAT, aparece un séptimo paso, el cual no era parte de la práctica anterior. Este nuevo paso lo hemos definido para iniciar arpwatch una vez instanciado el bridge virtual brint. De este modo, se ha iniciado la captura de tráfico ARP en los pods KNF *cpe*, lo cual veremos en la sección Arpwatch de este documento.

4. ARPWATCH

Arpwatch es un programa de software informático de código abierto que ayuda a monitorear la actividad del tráfico de ethernet en una red y mantiene una base de datos de los emparejamientos de direcciones ethernet/ip. Produce un registro de emparejamientos con la información acerca de las direcciones IP y MAC junto con marcas de tiempo, para que se pueda monitorear cuando apareció dicho emparejamiento.

Tras la inicialización de arpwatch junto con el propio KNF:cpe, que es el pod que alberga esta funcionalidad, procedemos a comprobar que, efectivamente, rellene el fichero asociado a la interfaz brint con dichos pares ethernet/ip. Para ello, ejecutaremos un ping desde el equipo h11 a una dirección desconocida para el mismo:



Como se puede ver, el destino es inalcanzable. El DHCP asignaba IPs en el rango 192.168.255.20 – 30, por lo que esa IP sabemos con certeza que iba a ser desconocida para h11. El protocolo ARP especifica que se hará broadcast para localizar al destinatario, llegando así la petición al KNF:cpe, y pudiendo ser detectada por el arpwatch.

A continuación podemos ver el resultado de la ejecución del fichero automatizado `./captureArp.sh` cuyo código es visible en el código adjunto en esta entrega. Este script permite ver los procesos en ejecución para Linux, relacionados con arpwatch. Como era de esperar, aparece la interfaz brint en escucha y el correo electrónico de uno de los participantes de este trabajo, tal como fue definido en el Dockerfile del KNF:cpe. Posteriormente, se puede observar el contenido de la tabla ARP actual, y de *brint.dat*, con las dos IPs de los equipos de la red residencial asociada a ese KNF:cpe, y su propia IP en esa subred.

5. QOS

En esta sección se explicará el procedimiento seguido para lograr las prestaciones de QoS solicitadas para un escenario que despliegue equipos en una red residencial con IPs asignadas de forma dinámica. Adicionalmente, se mostrarán las pruebas realizadas con la herramienta iperf3 para demostrar su correcto funcionamiento.

5.1. OBTENCIÓN DE IPS

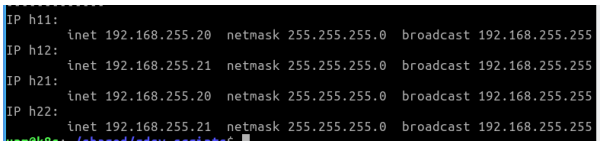
Comenzaremos obteniendo las IPs asignadas a los equipos de las redes residenciales. Ya que se utiliza DHCP para IPv4, sabemos que, en cada instanciación, se asignarán de forma dinámica, es decir, no tendrán una IP fija.

Es por ello por lo que comenzamos ejecutando un fichero automatizado, desde la máquina K8s, que primero permita asegurarse de que dichos equipos tienen una IP asignada en esa interfaz, y, después, que permita obtenerla. Para ello, se ha creado un fichero automatizado ejecutable como `./kubernetes/getIps.sh`

El contenido de getIps.sh se puede ver en el código adjunto en esta entrega.

Se ha hecho uso de los *tags* definidos en el fichero xml para la creación de los equipos de la red residencial, vistos en apartados anteriores de este documento. Adicionalmente, a pesar de la asignación dinámica de IPs, sabemos que estas siempre serán de clase C y que iniciarán con la terminación 192.168.255, lo cual nos permite hacer un mejor filtrado de su valor de cara a mostrar la IP visualmente por pantalla.

La ejcución de dicho comando tiene la siguiente salida:



Como era de esperar, solo varía el último dígito de cada IP de dicha interfaz, lo cual coincide con la máscara de red mostrada en pantalla. Las IPs obtenidas se deben anotar para su posterior uso desde la máquina de OSM.

5.2. ARRANQUE DE OPENFLOW EN KNF ACCESS

Desde OSM, podemos acceder a los pods KNF *access*, e iniciar el conmutador OpenFlow haciendo uso de un script que tome como parámetros de entrada las 4 IPs obtenidas para los equipos de la red residencial.

Para ello, se ha creado un fichero automatizado ejecutable como `./osm/configureOpenFlow.sh -a 192.168.255.20 -b 192.168.255.21 -c 192.168.255.20 -d 192.168.255.21`

El contenido de configureOpenFlow.sh se puede ver en el código adjunto en esta entrega.

Como se puede ver, el *script* automatizado espera 4 *flags*, cada una para una de las IPs de los equipos de la red residencial. Tras una comprobación de que dichos parámetros hayan sido correctamente definidos y, siguiendo la misma lógica vista en los ficheros de instanciación de renes para dar conectividad a la red residencial, ejecutamos una serie de comandos en los pods KNF *access*.

Para el pod KNF *access* asociado a renes1, se enviarán h11 y h12 por parámetro, mientras que, al asociado a renes2, se le enviarán h21 y h22.

En la sección sobre contenedores Docker de este documento se menciona como la imagen para este tipo de pods, vnf-img, incluía un fichero initOpenFlow.sh, que se explicaría más adelante. Por tanto, se va a explicar a continuación. Se va a ir mostrando por extractos de código debido a su extensión y relevancia:

initOpenFlow.sh

```
#!/bin/bash

while getopts a:b: flag
do
    case "${flag}" in
        a) hx1=${OPTARG};;
        b) hx2=${OPTARG};;
    esac
done

if [[ $hx1 == "" ]] || [[ $hx2 == "" ]]
then
    echo "Se deben definir las IPs de la red residencial"
else
    # Instalar dependencias necesarias
    echo "Instalando dependencias ..."
    cd ryu/
    pip install .
    cd ..

    # Para activar OpenFlow version 1.3 en el bridge brint, porque sino se pone por defecto la v1.0:
    echo "Definiendo version OpenFlow 1.3 ..."
    ovs-vsctl set Bridge brint protocols=OpenFlow13
```

Comenzamos comprobando que los *flags* asociados a las variables hx1 y hx2 hayan sido definidos.

```
# Definir el puerto del manager de OpenFlow:
echo "Definiendo propiedades del controller ..."
ovs-vsctl set-manager ptcp:6632
ovs-vsctl set-controller brint tcp:127.0.0.1:6633
ovs-vsctl set bridge brint other-config:datapath-id=0000000000000001

# Crear qos_simple_switch_13.py
echo "Creando qos_simple_switch_13.py ..."
sed '/OFPFlowMod(/,/)/s/)/, table_id=1)/' ryu/ryu/app/simple_switch_13.py > ryu/ryu/app/qos_simple_switch_13.py
```

Determinamos que el conmutador OpenFlow utiliza el puerto 6632 en TCP para comunicarse con el controlador ryu-manager, para el cual definimos una conexión TCP en el puerto 6633 y una configuración para el bridge virtual de conexión con el switch de identificador 00…01.

A continuación, siguiendo las especificaciones del documento sobre Qos de Ryu [1], creamos un archivo como modificación del conmutador básico, pero que incluya calidad de servicio por flujo, registrando las entradas en la tabla de identificador 1 de OpenFlow.

```
# Instalar dependencias
echo "Instalando dependencias ..."
cd ryu/; python3 ./setup.py install
cd ..

# Para ejecutar la aplicacion Ryu:
echo "Ejecutando aplicacion Ryu qos_simple_switch_13.py ..."
ryu-manager ryu/ryu/app/rest_qos.py ryu/ryu/app/qos_simple_switch_13.py ryu/ryu/app/rest_conf_switch.py & > ryulogs.log

# Terminates the program (like Ctrl+C)
PID=$!
sleep 5
kill -INT $PID
```

Instalamos las dependencias necesarias del directorio /ryu (mencionado en la sección de contenedores Docker) y ejecutamos el controlador ryu-manager, guardando los logs en un fichero.

En este, hacemos uso de 3 ficheros. El primero de ellos proporciona una API para poder gestionar las colas de los conmutadores, sus reglas, y sus VLANes de ser así necesario. El segundo es el fichero que acabamos de crear que incluye toda la funcionalidad de un conmutador OpenFlow pero añadiéndole calidad de servicio. El tercero de ellos proporciona una API para configurar el propio conmutador basándonos en un *dpid*.

Esperamos 5 segundos a que se termine de configurar y forzamos un Ctrl+C por pantalla para que no bloquee la ejecución de la terminal.

```
# KNF access -> brgX : 12 Mbps bajada
# brgX -> KNF access : 6 Mbps subida
# Definir la ruta del manager
curl -X PUT -d '"tcp:127.0.0.1:6632"' http://localhost:8080/v1.0/conf/switches/0000000000000001/ovsdb_addr
sleep 2

# Crear una cola con QoS
# 12 Mbps como máximo para el enlace
# Cola 0 - hx1 : mínimo 8 Mbps
# Cola 1 - hx2 : máximo 4 Mbps
curl -X POST -d '{"port_name": "vxlanacc", "type": "linux-htb", "max_rate": "12000000", "queues": [{"min_rate": "8000000"}, {"max_rate": "4000000"}]}'
http://localhost:8080/qos/queue/0000000000000001
```

Haciendo uso de las APIs proporcionadas por los ficheros instanciados con ryu-manager, procedemos a determinar los requerimientos específicos de nuestro escenario y de la QoS deseada sobre el conmutador definido.

Desde KNF *access* limitamos el tráfico de bajada, por lo que hay que especificar un caudal máximo de 12Mbps para el túnel, y de salida por el puerto vxlanacc de brint. Asimismo, definimos dos colas, una con caudal mínimo de 8Mbps y otra con 4Mbps de máximo.

```
# Definir a que cola pertenece cada tráfico
curl -X POST -d '{"match": {"nw_dst": "'$hx1'"}, "actions":{"queue": "0"}}' http://localhost:8080/qos/rules/0000000000000001
curl -X POST -d '{"match": {"nw_dst": "'$hx2'"}, "actions":{"queue": "1"}}' http://localhost:8080/qos/rules/0000000000000001
```

Gracias a la obtención de las IPs a través de los *flags* del fichero, asignamos el tráfico hacia hx1 a la cola 0 y el tráfico hacia hx2 a la cola 1, cumpliendo así las especificaciones requeridas.

Adicionalmente, nos gustaría destacar que, para llegar a las conclusiones de qué identificador de switch utilizar y qué puertos y bridges virtuales estaban disponibles, hicimos uso de comandos propios de Open VSwitch. A continuación, podemos ver como el bridge virtual definido es brint y sus puertos son vxlanacc y vxlanint para su conexión con máquinas exteriores:

```
root@helmchartrepo-accesschart-0039529782-fdb66ccff-xh564:/# ovs-vsctl show
d1eed3ef-2e58-4aa9-a88a-f0fff15cc8c8
    Bridge brint
        Port vxlanacc
            Interface vxlanacc
        Port brint
            Interface brint
                type: internal
        Port vxlanint
            Interface vxlanint
    ovs_version: "2.13.8"
```

Asimismo, utilizamos la API proporcionada gracias a la instanciación de ryu-manager utilizando dichos ficheros para obtener el identificador del conmutador a utilizar, y la clave de configuración ovsdb_addr:

```
root@helmchartrepo-accesschart-0039529782-fdb66ccff-xh564:/# curl -X GET http://localhost:8080/v1.0/conf/switches
(603) accepted ('127.0.0.1', 53482)
127.0.0.1 - - [03/Jan/2023 21:36:11] "GET /v1.0/conf/switches HTTP/1.1" 200 128 0.001489
["0000000000000001"]root@helmchartrepo-accesschart-0039529782-fdb66ccff-xh564:/#
```

```
root@helmchartrepo-accesschart-0039529782-fdb66ccff-xh564:/# curl -X GET http://localhost:8080/v1.0/conf/switches/0000000000000001
["ovsdb_addr"]root@helmchartrepo-accesschart-0039529782-fdb66ccff-xh564:/#
```

Una vez explicado todo el procedimiento seguido para determinar la conectividad entre el conmutador OpenFlow y el controlador, y la instanciación de las reglas para el mismo junto con la asignación de los flujos a diferentes colas según los requerimientos solicitados en el enunciado de esta práctica, procedemos a mostrar la salida del comando por pantalla:

```
(183) wsgi starting up on http://0.0.0.0:8080
[QoS][INFO] dpid=0000000000000001: Join qos switch.
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0(183) accepted ('127.0.0.1', 47366)
100    20    0     0  100    20      0   6666 --:--:-- --:--:-- --:--:--  6666
127.0.0.1 - - [13/Jan/2023 21:56:41] "PUT /v1.0/conf/switches/0000000000000001/ovsdb_addr HTTP/1.1" 201 120 0.001885
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0(183) accepted ('127.0.0.1', 47382)
100   305  100   173  100   132   4119   3142 --:--:-- --:--:-- --:--:--  7261
[{"switch_id": "0000000000000001", "command_result": {"result": "success", "details": {"0": {"config": {"min-rate": "8000000"}}, "1": {"config": {"max-rate": "4000000"}}}}}]127.0.0.1 - - [1
3/Jan/2023 21:56:43] "POST /qos/queue/0000000000000001 HTTP/1.1" 200 282 0.041100
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0(183) accepted ('127.0.0.1', 47390)
100    63    0     0  100    63      0  63000 --:--:-- --:--:-- --:--:-- 63000
127.0.0.1 - - [13/Jan/2023 21:56:43] "POST /qos/rules/0000000000000001 HTTP/1.1" 400 124 0.000271
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0(183) accepted ('127.0.0.1', 47394)
100    63    0     0  100    63      0  63000 --:--:-- --:--:-- --:--:-- 63000
127.0.0.1 - - [13/Jan/2023 21:56:43] "POST /qos/rules/0000000000000001 HTTP/1.1" 400 124 0.000320
packet in 0000000000000001 02:fd:00:04:01:01 c2:65:8f:7a:80:4e 1
packet in 0000000000000001 c2:65:8f:7a:80:4e 02:fd:00:04:01:01 2
packet in 0000000000000001 02:fd:00:04:01:01 c2:65:8f:7a:80:4e 1
```

Como se puede ver, el conmutador ha aceptado las peticiones a la API de creación de colas y definición de flujos mínimos y máximos, creando las dos colas esperadas: 0 y 1. En ese momento, se puede ver cómo, al iniciar un ping desde la máquina h11, se empiezan a ver mensajes PACKET_IN, propios de OpenFlow, por pantalla. Una vez mostrada esta imagen de cara al

lector de esta memoria, se ha redirigido la salida de los logs de ryu-manager a un fichero para evitar la inundación de la consola.

Se muestra a continuación la dirección ether de h11 en su interfaz eth1, que, como se puede ver, coincide con los paquetes recibidos en el conmutador de id 00…01: `ether 02:fd:00:04:00:01  txqueuelen 1000  (Ethernet)`

[1] http://osrg.github.io/ryu-book/en/html/rest_qos.html

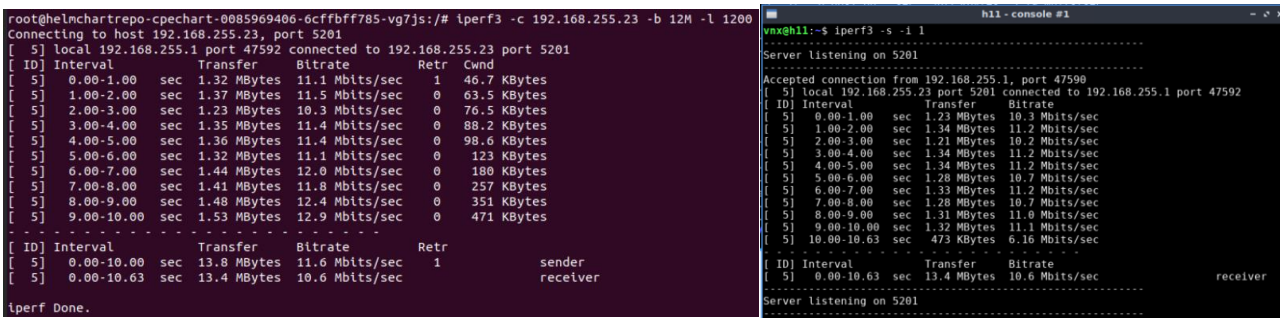
5.3. PRUEBAS DE PRESTACIONES CON IPERF3 PARA EL TRÁFICO DE BAJADA

Para comprobar la QoS de bajada desde K8s hacia las redes residenciales se va a utilizar la herramienta *iperf3*, la cual hemos instalado en los KNF en su instanciación en el Dockerfile, y en los dispositivos de las redes residenciales mediante una modificación de su XML añadiendo un nuevo comando ejecutable.

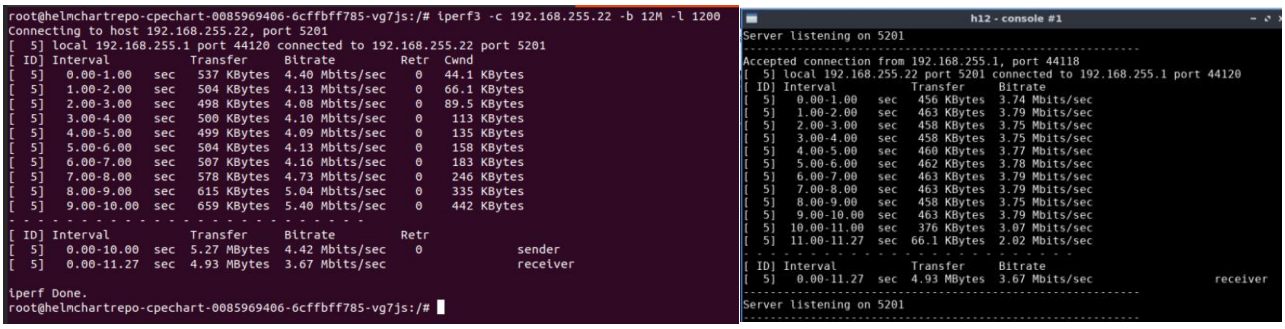
Para instalar iperf3 en los equipos de la red residencial tras la creación de los comandos asociados en VNX se utiliza el fichero automatizado `./kubernetes/installIperf3.sh`

Definimos que el tráfico de bajada viajará como máximo a 12 Mbps, y que el referente a los hx1 tendrá como mínimo 8 Mbps, mientras que el tráfico que va encaminado a hx2 tendrá como máximo 4 Mbps.

Comenzamos haciendo pruebas desde el KNF:cpe hacia h11. El tráfico esperado estará comprendido entre 8 – 12 Mbps.



Las pruebas son satisfactorias. Como se puede ver, el tráfico tiene de media 10.6 Mbps, perteneciendo al rango esperado. Continuamos las pruebas desde el KNF:cpe hacia h12. El tráfico esperado deberá ser ligeramente inferior a 4 Mbps.



Nuevamente el resultado obtenido es el esperado, con una media de tráfico a 3.67 Mbps. Cabe denotar que todas estas pruebas se han realizado utilizando el KNF:cpe asociado a renes1, por lo que se han repetido para renes2, utilizando los dispositivos h21 y h22, obteniéndose los mismos resultados correctos.

5.4. PRUEBAS DE PRESTACIONES CON IPERF3 PARA EL TRÁFICO DE SUBIDA

Análogamente, se va a limitar el tráfico desde la red residencial hacia el exterior. Para ello, debemos definir en los equipos brg1 y brg2 la IP del controlador, y, utilizando *ip link* para que funcione correctamente QoS, establecer el puerto vxlan1 para br0. A continuación se puede ver la modificación del *on_boot* de brg1, diferenciándose de brg2 en la IP del controlador, que será 10.255.0.3 para la red residencial 2. También se muestra el resultado en brg1 del controlador definido.

```
<exec seq="on_boot" type="verbatim">
    service openvswitch-switch start
    sleep 5
    ovs-vsctl add-br br0
    ovs-vsctl add-port br0 eth1
    ifconfig eth1 up

    ip link add vxlan1 type vxlan id 0 remote 10.255.0.1 dstport 4789 dev eth2
    ovs-vsctl add-port br0 vxlan1
    ifconfig vxlan1 up

    # Definir versiones OpenFlow
    echo "Definiendo version OpenFlow 1.0, 1.1, 1.2, 1.3 ..."
    ovs-vsctl set Bridge br0 protocols=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13

    # Definir el puerto del manager y controlador de OpenFlow:
    ovs-vsctl set-manager ptcp:6632
    ovs-vsctl set-controller br0 tcp:10.255.0.1:6633
</exec>
```

```
Manager "ptcp:6632"
Bridge br0
    Controller "tcp:10.255.0.1:6633"
        is_connected: true
```

Adicionalmente, se realiza un fichero automatizado `./osm/upQos.sh` para la definición de las reglas asociadas al QoS de subida. Se podría haber hecho juntamente con el QoS de bajada, pero se ha hecho en un fichero separado para diferenciar la entrega obligatoria de la opcional de esta práctica. Su código se puede observar junto con los ficheros adjuntos en esta entrega.

Finalmente, se muestran los resultados, utilizando renes2 de las prestaciones con la herramienta iperf3 del QoS de subida, ejecutado desde h21 y h22 y capturado en el KNF:cpe asociado a esa red residencial. Comenzamos haciendo pruebas desde el h21 hacia KNF:cpe. El tráfico esperado estará comprendido entre 4 – 6 Mbps.

```
-----------------------------------------------------------
Server listening on 5900
-----------------------------------------------------------
Accepted connection from 192.168.255.28, port 49192
[  5] local 192.168.255.1 port 5900 connected to 192.168.255.28 port 49194
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   681 KBytes  5.58 Mbits/sec
[  5]   1.00-2.00   sec   682 KBytes  5.58 Mbits/sec
[  5]   2.00-3.00   sec   692 KBytes  5.68 Mbits/sec
[  5]   3.00-4.00   sec   691 KBytes  5.66 Mbits/sec
[  5]   4.00-5.00   sec   695 KBytes  5.70 Mbits/sec
[  5]   5.00-6.00   sec   685 KBytes  5.61 Mbits/sec
[  5]   6.00-7.00   sec   695 KBytes  5.69 Mbits/sec
[  5]   7.00-8.00   sec   682 KBytes  5.59 Mbits/sec
[  5]   8.00-9.00   sec   693 KBytes  5.67 Mbits/sec
[  5]   9.00-10.00  sec   673 KBytes  5.51 Mbits/sec
[  5]  10.00-10.60  sec   415 KBytes  5.70 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.60  sec  7.11 MBytes  5.63 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5900
-----------------------------------------------------------
```

```
vnx@h21:~$ iperf3 -c 192.168.255.1 -b 6M -l 1200 -p 5900
Connecting to host 192.168.255.1, port 5900
[  5] local 192.168.255.28 port 49194 connected to 192.168.255.1 port 5900
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   732 KBytes  6.00 Mbits/sec    0   45.4 KBytes
[  5]   1.00-2.00   sec   732 KBytes  5.99 Mbits/sec    0   79.1 KBytes
[  5]   2.00-3.00   sec   732 KBytes  6.01 Mbits/sec    0    114 KBytes
[  5]   3.00-4.00   sec   732 KBytes  6.00 Mbits/sec    0    148 KBytes
[  5]   4.00-5.00   sec   732 KBytes  6.00 Mbits/sec    0    183 KBytes
[  5]   5.00-6.00   sec   732 KBytes  6.00 Mbits/sec    0    217 KBytes
[  5]   6.00-7.00   sec   732 KBytes  6.00 Mbits/sec    0    252 KBytes
[  5]   7.00-8.00   sec   732 KBytes  6.00 Mbits/sec    0    287 KBytes
[  5]   8.00-9.00   sec   731 KBytes  5.99 Mbits/sec    0    323 KBytes
[  5]   9.00-10.00  sec   734 KBytes  6.01 Mbits/sec    0    416 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  7.15 MBytes  6.00 Mbits/sec    0             sender
[  5]   0.00-10.60  sec  7.11 MBytes  5.63 Mbits/sec                  receiver

iperf Done.
```

Las pruebas son satisfactorias. Como se puede ver, el tráfico tiene de media 5.63 Mbps, perteneciendo al rango esperado. Continuamos las pruebas desde el h22 hacia KNF:cpe. El tráfico esperado deberá ser ligeramente inferior a 2 Mbps.

```
root@helmchartrepo-cpechart-0023898006-6d79c4878b-vqjhf:/# iperf3 -s -i 1 -p 5900
-----------------------------------------------------------
Server listening on 5900
-----------------------------------------------------------
Accepted connection from 192.168.255.26, port 42874
[  5] local 192.168.255.1 port 5900 connected to 192.168.255.26 port 42876
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   234 KBytes  1.91 Mbits/sec
[  5]   1.00-2.00   sec   233 KBytes  1.91 Mbits/sec
[  5]   2.00-3.00   sec   232 KBytes  1.90 Mbits/sec
[  5]   3.00-4.00   sec   232 KBytes  1.90 Mbits/sec
[  5]   4.00-5.00   sec   232 KBytes  1.90 Mbits/sec
[  5]   5.00-6.00   sec   232 KBytes  1.90 Mbits/sec
[  5]   6.00-7.00   sec   231 KBytes  1.89 Mbits/sec
[  5]   7.00-8.00   sec   233 KBytes  1.91 Mbits/sec
[  5]   8.00-9.00   sec   232 KBytes  1.90 Mbits/sec
[  5]   9.00-10.00  sec   233 KBytes  1.91 Mbits/sec
[  5]  10.00-11.00  sec   233 KBytes  1.91 Mbits/sec
[  5]  11.00-11.74  sec   169 KBytes  1.88 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-11.74  sec  2.66 MBytes  1.90 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5900
-----------------------------------------------------------
```

```
vnx@h22:~$ iperf3 -c 192.168.255.1 -b 6M -l 1200 -p 5900
Connecting to host 192.168.255.1, port 5900
[  5] local 192.168.255.26 port 42876 connected to 192.168.255.1 port 5900
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   315 KBytes  2.58 Mbits/sec    0   45.4 KBytes
[  5]   1.00-2.00   sec   255 KBytes  2.09 Mbits/sec    0   57.1 KBytes
[  5]   2.00-3.00   sec   254 KBytes  2.08 Mbits/sec    0   68.7 KBytes
[  5]   3.00-4.00   sec   251 KBytes  2.05 Mbits/sec    0   80.4 KBytes
[  5]   4.00-5.00   sec   254 KBytes  2.08 Mbits/sec    0   92.1 KBytes
[  5]   5.00-6.00   sec   285 KBytes  2.33 Mbits/sec    0    122 KBytes
[  5]   6.00-7.00   sec   318 KBytes  2.60 Mbits/sec    0    170 KBytes
[  5]   7.00-8.00   sec   345 KBytes  2.82 Mbits/sec    0    231 KBytes
[  5]   8.00-9.00   sec   369 KBytes  3.02 Mbits/sec    0    307 KBytes
[  5]   9.00-10.00  sec   408 KBytes  3.34 Mbits/sec    0    403 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  2.98 MBytes  2.50 Mbits/sec    0             sender
[  5]   0.00-11.74  sec  2.66 MBytes  1.90 Mbits/sec                  receiver
```

Nuevamente el resultado obtenido es el esperado, con una media de tráfico a 1.90 Mbps. Cabe denotar que todas estas pruebas se han realizado utilizando el KNF:cpe asociado a renes2, por lo que se han repetido para renes1, utilizando los dispositivos h11 y h12, obteniéndose los mismos resultados correctos.

ANEXOS

ANEXO I: CONTENEDORES DOCKER

Para la realización de esta práctica se ha hecho uso de dos imágenes Docker con distinta funcionalidad: una para el KNF *access* y otra para KNF *cpe*. Dicha distinción se ha llevado a cabo debido a que cada VNF debe cumplir un rol diferente en la puesta en marcha del escenario, por lo que así minimizamos las dependencias requeridas de instalación en ambos y hacemos una distinción entre los ficheros a incluir en ellas. A continuación, se explica el contenido de cada una de ellas:

KNF ACCESS

El pod asociado al KNF *access* se ha solicitado que tenga una funcionalidad de conmutador OpenFlow para gestionar el QoS del tráfico de bajada hacia ambas redes residenciales.

Por tanto, este pod va a hacer uso de las funcionalidades de ryu, además de herramientas básicas de monitorización de tráfico en red. Para ello, partimos de un repositorio de código *open source* [1], el cual incluimos en la imagen para que disponga de los archivos de dicha librería. Adicionalmente, indicamos que se deben clonar los ficheros README.txt, con los nombres de los autores de este trabajo, y un *script* de inicio de OpenFlow automatizado, para su posterior uso.

Dockerfile

```
FROM ubuntu:20.04
# variables to automatically install tzdata
ARG DEBIAN_FRONTEND=noninteractive
ENV TZ=Europe/Madrid
# install required packages
RUN apt-get clean
RUN apt-get update \
    && apt-get install -y \
    net-tools \
    traceroute \
    curl \
    iptables \
    inetutils-ping \
    nano \
    build-essential \
    bridge-utils \
    isc-dhcp-server \
    tcpdump \
    openvswitch-switch \
    openvswitch-common \
    iperf3 \
    iproute2 \
    vim \
    ryu-bin

COPY vnx_config_nat /usr/bin/
COPY isc-dhcp-server /etc/default/isc-dhcp-server
COPY dhcpd.conf /etc/dhcp/dhcpd.conf

COPY README.txt .
COPY initOpenFlow.sh .
COPY ./ryu/ /ryu/
```

El fichero initOpenFlow.sh se explicará posteriormente, en la sección de QoS (Quality of Service) de este documento.

Dicha imagen ha sido construida y almacenada en un repositorio de Dockerhub [2] con los siguientes comandos:

```
docker build -t patriciaortuno28/vnf-img .

docker push patriciaortuno28/vnf-img
```

patriciaortuno28/vnf-img:latest

DIGEST: sha256:50e54d228b67ba00b6ff0df57541e57d7bc3ae61610616b6965483ed7f427910

OS/ARCH	COMPRESSED SIZE	LAST PUSHED	TYPE
linux/amd64	218.63 MB	3 days ago by patriciaortuno28	Image

Gracias a ello, se puede disponer de dicha imagen para construir los pods necesarios asociados a la funcionalidad del KNF *access*. Esto se debe a que en el fichero values.yaml asociado al *accesschart* de *Helm* de nuestro repositorio repo-rdsv público [3] utilizamos dicha imagen para instanciar estos pods:

```
image:
  repository: patriciaortuno28/vnf-img
  pullPolicy: Always
  # Overrides the image tag whose default is the chart appVersion.
  tag: "latest"
```

[1] https://github.com/faucetsdn/ryu [2] https://hub.docker.com/repositories/patriciaortuno28 [3] https://github.com/patriciaOrtuno28/repo-rdsv

KNF CPE

El pod asociado al KNF *cpe* se ha solicitado que tenga la funcionalidad de captura de tráfico ARP haciendo uso de la herramienta arpwatch.

Por tanto, el Dockerfile para construir una imagen que actúe como plantilla para los contenedores con dicha funcionalidad, debe diferir de la anteriormente explicada para el KNF *access*. Además de prescindir de la instalación del paquete ryu-bin, se procede a la instalación de arpwatch dentro del RUN del Dockerfile.

Adicionalmente, las 3 últimas sentencias COPY vistas anteriormente se han de sustituir por el siguiente extracto de código:

```
COPY README.txt .
COPY captureArp.sh .

RUN echo 'deb http://archive.ubuntu.com/ubuntu/ trusty main universe restricted multiverse' >> /etc/apt/sources.list

RUN apt-get update
RUN apt-get -f install sysv-rc-conf -y
RUN sysv-rc-conf --level 35 arpwatch on

RUN sed -i 's@INTERFACES=""@INTERFACES="brint"@g' /etc/default/arpwatch

RUN echo 'IFACE_ARGS="-m p.ortuno@alumnos.upm.es"' > /etc/arpwatch/brint.iface
```

Por una parte, modificamos las fuentes permitidas de instalación para Ubuntu, siendo así capaces de instalar el paquete *sysv-rc-conf*. Gracias al mismo, podemos activar el servicio arpwatch.

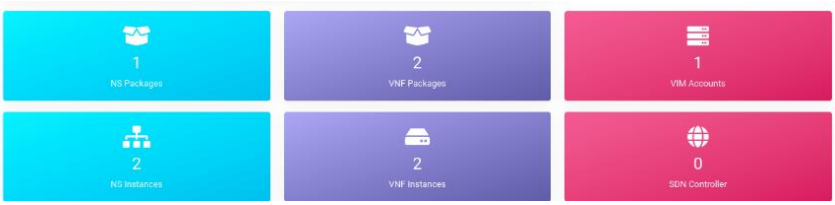
Modificamos el fichero por defecto de configuración de arpwatch para indicar que la interfaz a escuchar será brint, ya que es el túnel de comunicación con las redes residenciales, siendo estas el objetivo de la captura ARP prevista. Adicionalmente, definimos un fichero de configuración para dicha interfaz, indicando el correo electrónico que debe recibir alertas en el caso de una nueva detección de tráfico ARP por parte de un pod *cpe*.

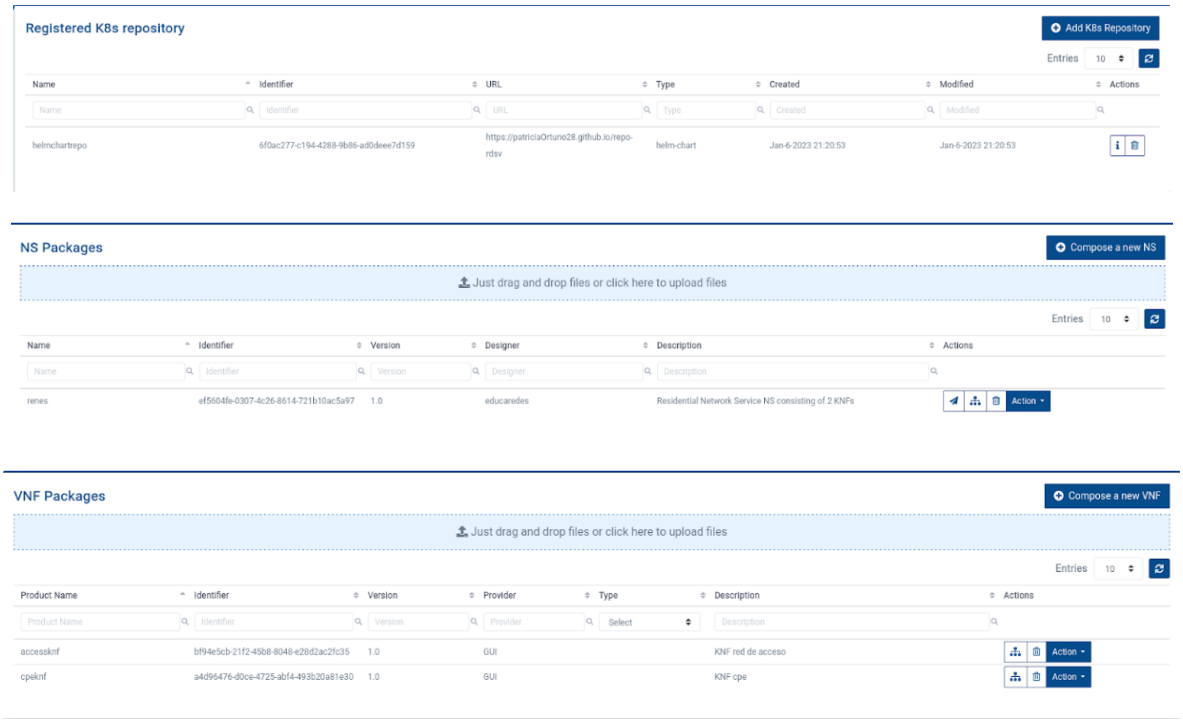
Por tanto, en esta imagen queda instanciada la configuración del servicio arpwatch para el KNF *cpe*, pero su inicio y captura de tráfico se realizarán más adelante junto con la creación de dichos pods, una vez se han inicializado los túneles VXLAN previstos.

El fichero captureArp.sh se explicará posteriormente, en la sección de Arpwatch.

Al igual que en el VNF *access*, dicha imagen se ha subido a Dockerhub, pero, esta vez, bajo el nombre vnf-img-cpe, utilizable por el *cpechart* de *Helm*.

ANEXO II: RESULTADOS DEL ONBOARDING





ANEXO III: DESTRUCCIÓN DE LOS ESCENARIOS

OSM

Opcionalmente se ofrece un *script* para la destrucción de los servicios de red renes1 y renes2, junto con sus pods asociados, a la par que se deshaga el *onboarding* realizado. Para ello, se ha creado un fichero automatizado ejecutable como `./osm/destroyRenes.sh`

destroyRenes.sh

```
#!/bin/bash


echo "Borrando el repositorio de K8s ..."
osm repo-delete helmchartrepo

echo "Borrando renes1 y renes2 ..."
osm ns-delete $NSID1
osm ns-delete $NSID2

sleep 25

echo "Borrando el NS Package ..."
osm nspkg-delete renes

sleep 1

echo "Borrando los VNF Package ..."
osm nfpkg-delete accessknf
osm nfpkg-delete cpeknf
```

K8S

Opcionalmente se ofrece un *script* para parar la ejecución de los equipos de las redes residenciales, del router y del servidor.

Para ello, se ha creado un fichero automatizado ejecutable como `./kubernetes/stopScenario.sh`

stopScenario.sh

```
#!/bin/bash

cd /home/upm/shared/rdsv-final
sudo vnx -f vnx/nfv3_home_lxc_ubuntu64.xml -P
sudo vnx -f vnx/nfv3_server_lxc_ubuntu64.xml -P
```