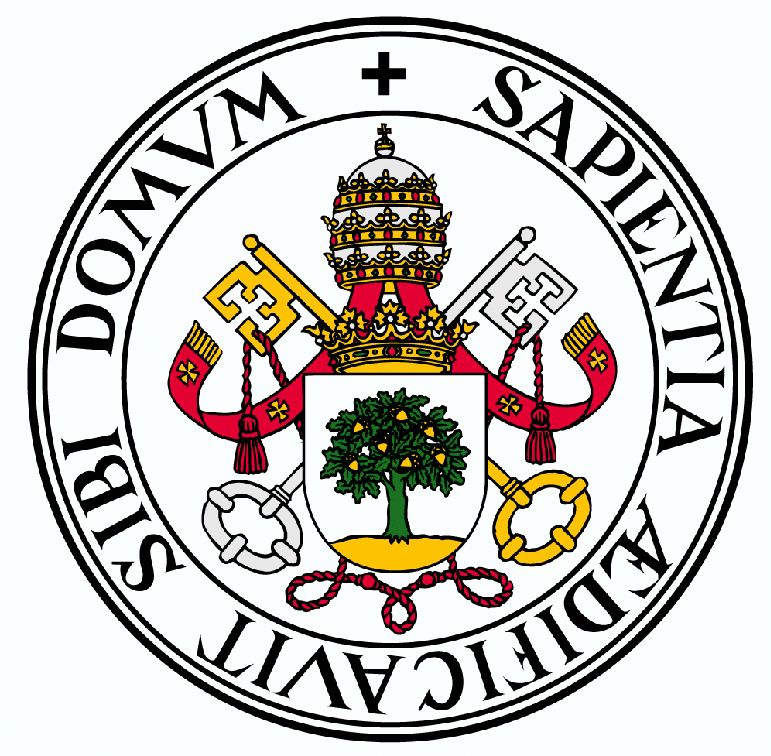
****

**Universidad de Valladolid**

Escuela Técnica Superior de Ingenieros de Telecomunicación (Valladolid)

Grado de Ingeniería en Tecnologías de Telecomunicación

**Avanzando hacia una red auto-adaptativa: simulación de redes definidas por software (SDN) mediante el simulador GNS3**

Alumno: Rubén Blanco Pérez

Tutores: Rubén Ruiz González

Jaime Gómez Gil

**DESCRIPCIÓN DEL TFG**

**TÍTULO**: Avanzando hacia una red auto-adaptativa: simulación de redes definidas por software (SDN) mediante el simulador GNS3

**AUTOR**: Rubén Blanco Pérez

**TUTOR**: Rubén Ruiz González, Departamento de Teoría de la Señal y

Comunicaciones e Ingeniería Telemática, E.T.S.I.

Telecomunicación, Universidad de Valladolid

**COTUTOR**: Jaime Gómez Gil, Departamento de Teoría de la Señal y

Comunicaciones e Ingeniería Telemática, E.T.S.I.

Telecomunicación, Universidad de Valladolid

**DEPARTAMENTO**: Departamento de Teoría de la Señal y Comunicaciones e

Ingeniería Telemática

**MIEMBROS DEL TRIBUNAL**

**PRESIDENTE**: José Fernando Díez Higuera

**VOCAL**: Jaime Gómez Gil

**SECRETARIO**: Isabel de la Torre Díez

**SUPLENTE**: Javier Manuel Aguiar Pérez

**FECHA** **DE** **LECTURA**:

**CALIFICACIÓN**:

Resumen

**INTRODUCCIÓN**

Las redes tradicionales requieren de cambios de cada elemento individual. El uso de redes definidas por software (SDN) permite disponer de in controlador que gestiona de manera simultánea todos los elementos de la red. Este enfoque permite adaptar los elementos de la red de manera dinámica, habilitando la optimización de la red a las necesidades concretas según varían en tiempo real. En este TFG se pretende simular, mediante el simulador gráfico de redes GNS3, diferentes esquemas SDN que permitan que la red se auto-adapte a las condiciones actuales. Con este trabajo se espera conseguir desarrollar nuevos mecanismos que puedan implementarse en entornos reales, avanzando así hacia el despliegue de redes auto-adaptativas.

**OBJETIVOS**

El objetivo principal de este trabajo es configurar una red con el simulador GNS3 e incorporarle a un controlador diferentes aplicaciones que gestione de diferentes formas el comportamiento de dicha red, consiguiendo de esta forma una red auto-adaptativa.

**MATERIAL**

Entre los materiales que se han utilizado se encuentran: Un ordenador portátil que contiene una distribución Linux instalada , el entorno gráfico GNS3, utilizado para la configuración de la red, el *software* Eclipse, utilizado para programar las diferentes aplicaciones que se le mandarán al controlador de la red.

**METODOLOGÍA**

**RESULTADOS**

**CONCLUSIONES**

**Palabras clave**: red, GNS3, controlador, ONOS, aplicación, SDN, simulación

Abstract

**INTRODUCTION**

**OBJECTIVES**

**MATERIALS**

**METHODOLOGY**

**RESULTS**

**CONCLUSIONS**

**Keywords**:

Agradecimientos

*A nadie 😊*

Índice

Índice de figuras y tablas

Capítulo 1. Introducción

1.1 Objetivos del proyecto

1.2 Organización de la memoria

1.3 Historia de las redes SDN

Las redes SDN *(Software Defined Networking)* permiten a los ingenieros responder más rápidamente a diferentes requisitos a través de un control centralizado. Esta tecnología surgió hace aproximadamente 20 años y su historia se puede dividir en 3 etapas.

En primer lugar, el periodo denominado redes activas que abarca entre el año 1995 y el año 2000. Las redes activas surgieron como una forma de evitar los procesos de estandarización por parte de la IETF ya que era un mecanismo muy lento y no permitía avanzar tanto como se deseaba en esa época. Este tipo de redes están orientadas hacia el control de la red incluyendo una API (interfaz de programación) que expone recursos en los nodos de la red individuales, como por ejemplo el almacenamiento o las colas de paquetes y permite ejecutar código sobre ellos con el objetivo de poder procesar el flujo de datos. Fue el primer intento de hacer redes programables.

Una segunda etapa que transcurre durante los años 2001 y 2007 consiste en la separación de los planos de control y datos. El plano de control está destinado al tráfico que ocupan los equipos para gestionar, mantener y modificar el estado de la red, por ejemplo los protocolos de enrutamiento o la configuración de firewall están en este plano, mientras que el plano de datos se refiere a aquel destinado a los servicios, básicamente a hacer efectivo lo establecido en el plano de control, como por ejemplo el IP forwarding. El motivo fundamental de esta separación se debió a que el volumen de tráfico empezó a aumentar exponencialmente y los protocolos de enrutamiento convencionales dejaron de ser eficientes. Además, separar ambos planos permite también una independencia a la hora de desarrollo y, realizar un control desde un programa de alto nivel de toda una red lo que simplifica el debug.

La tercera etapa fue la aparición del protocolo Openflow que es considerado uno de los primeros estándares de las redes SDN y surgió a raíz de separar los planos.

Openflow es una tecnología de switching que empezó en la Universidad de Stanford y consiste básicamente en un protocolo de comunicación entre los controladores de la red, que deciden el encaminamiento y los switches que dispone la red. Cabe resaltar que los switches tradicionales no soportan este protocolo, por tanto fue necesario desarrollar unos nuevos que lo soportaran.

Algunos beneficios que introdujo Openflow son:

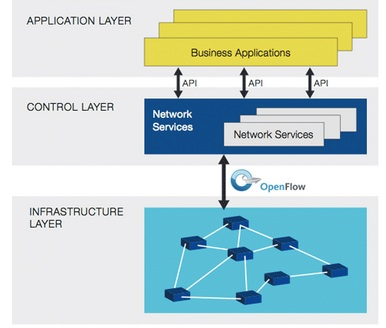
* Flexibilidad en el uso de la red, así como en la forma de operar sobre ella.
* El ser de código abierto, por tanto puede ser modificado de forma muy sencilla.
* Reducir el gasto de operación.
* Mejorar la integración de la red. Permite que la red sea controlada con un solo punto de vista.

1.2 Arquitectura de las redes SDN

Todas las redes SDN están caracterizadas por tener estos elementos:

* Aplicación SDN: Son programas que comunican el comportamiento de la red al controlador SDN
* Controlador SDN: Es el cerebro de la red y encargado de traducir las peticiones de la aplicación y enviarlas a los switches.
* Interfaz hacia el norte: Es la API que comunica la aplicación SDN y los controladores.
* Capa de datos: Compuesta por los diversos elementos de red como los switches o sistemas finales.
* Interfaz hacia el sur: Es el protocolo que comunica el controlador con la capa de datos, generalmente es Openflow.

En la Ilustración 1 se puede ver un esquema sencillo de la arquitectura de una red SDN.



1.3 Funcionamiento de una red SDN

Para comprender mejor el funcionamiento de una red SDN vamos a poner un pequeño ejemplo que clarificará los conceptos. Para ello supongamos que tenemos una red formada por 2 hosts que denominaremos Host 1 y Host 2, intercomunicados por un switch que soporta el protocolo OpenFlow (openVswitch) y el controlador.

Supongamos que el Host 1 envía una trama al Host 2. La trama en primer lugar llega al openVswitch. Si dicho switch posee una entrada de flujo para algún campo de capa 2, 3 o 4 se actúa según dicha regla, mientras que si el switch no posee una entrada en sus tablas envía el paquete al controlador para que este detalle cómo se debe conmutar dicha trama.

Capitulo 2: Primeros pasos

En este capítulo se van a detallar los primeros pasos previos a poder empezar a simular redes SDN.

2.1 Instalación del SO

Para poder realizar el proyecto correctamente es necesario la instalación de un sistema operativo Linux funcional. En este caso se ha elegido la distribución Linux Fedora 29 basada en GNOME 3, aunque cualquier distribución es válida.

El motivo de haber elegido este sistema operativo es debido a diferentes problemas con distribuciones más conocidas como Ubuntu o Debian. En concreto, cabe resaltar que el Ubuntu sufría de bloqueos, cada vez más habituales, que impedían una continuidad a la hora de trabajar con él.

Por su parte el sistema operativo Debian tiene problemas con los repositorios *non-free*

2.2 Instalación de GNS3

Una vez tenemos una distribución de Linux instalada, el siguiente paso es instalar el simulador gráfico GNS3. Este simulador es gratis y se puede descargar de forma sencilla desde la página web.

Una de las opciones que da el programa es utilizar una máquina virtual para que todo el proceso de simulación de las redes quede virtualizado. Para ello es recomendable instalar una herramienta de virtualización como VMWare y descargar la máquina virtual para GNS3 disponible en la página web.

2.3 Instalación de ONOS

NO RECUERDO, ONOS SE INSTALABA SOBRE GNS3? ERA INSTALANDO LA IMAGEN DIRECTAMENTE NO?

Y, TUVIMOS QUE INSTALARLO EN EL PC TAMBIEN NO?

2.4 Instalación de Eclipse

Una vez tenemos GNS3 instalado el siguiente paso es instalar Eclipse para poder programar las diferentes aplicaciones que queramos mandar al controlador y que veremos en capítulos posteriores.

2.5 Instalación de maven

BUSCAR, NO RECUERDO SI IBA CON ONOS O A PARTE

Capítulo 3: Creando la primera red con GNS3

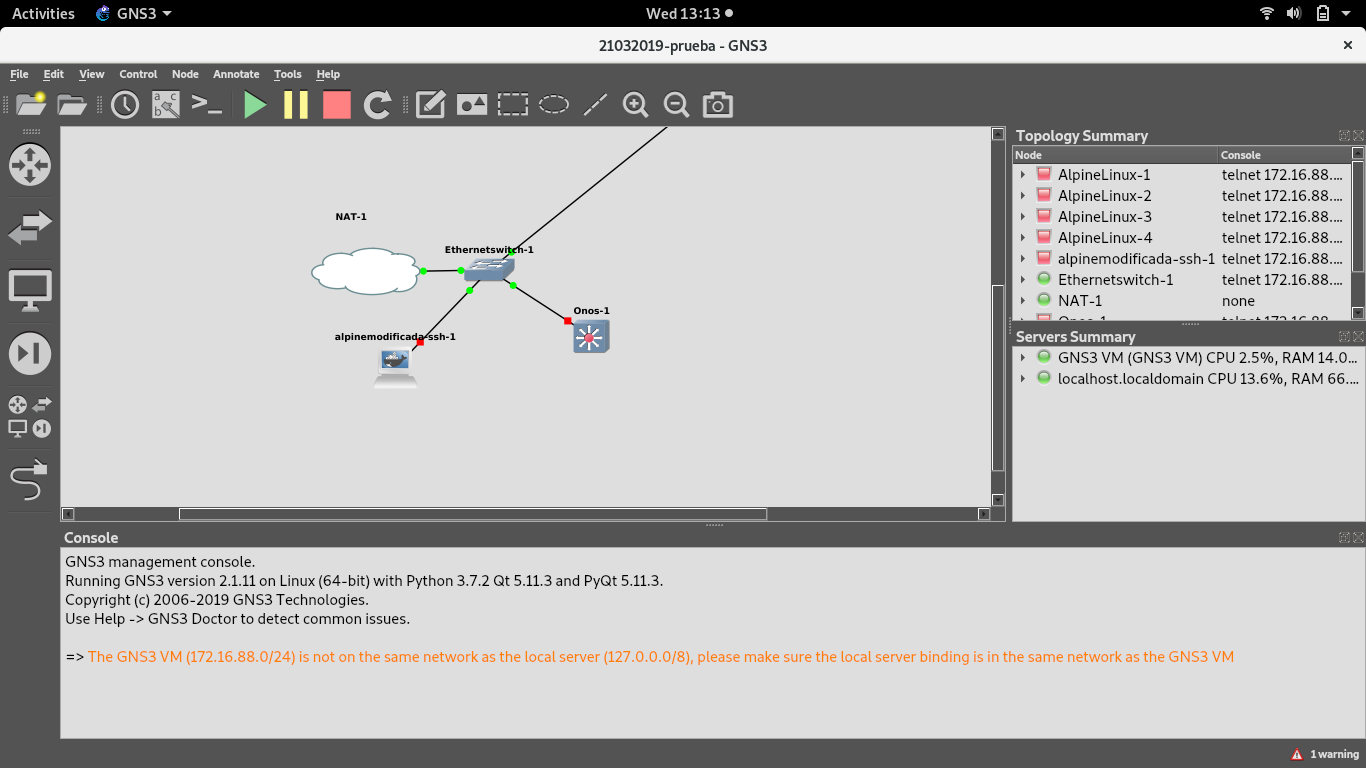
En este capítulo se va a detallar la red que se va a simular y sobre la cual instalaremos las diferentes aplicaciones que irán componiendo el proyecto.

La red se puede dividir en 2 partes diferenciadas:

Por un lado la conexión con el controlador y con el exterior y por otro los hosts conectados por un openVswitch

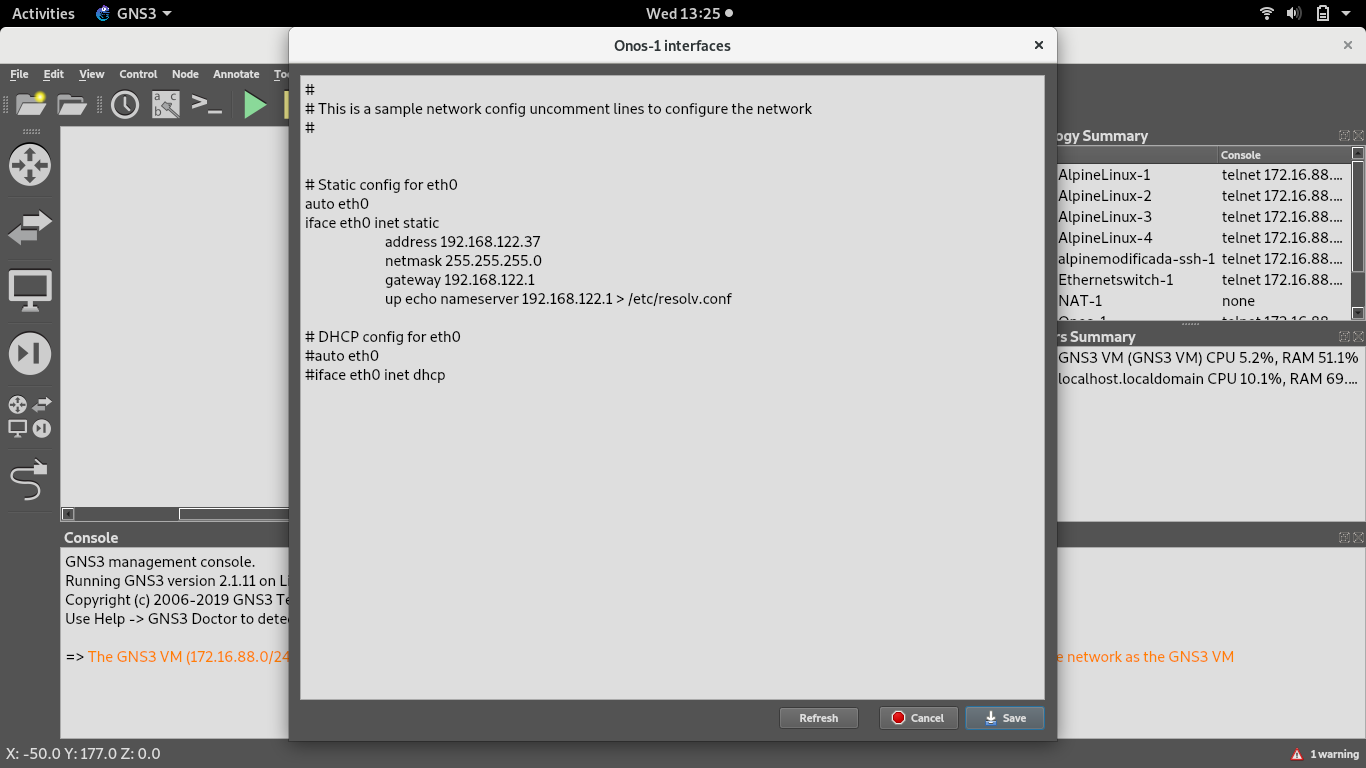
3.1 Conexión con el controlador y con el exterior

La parte de la red que concierne a este apartado se puede ver en la Ilustración siguiente:



Para poder realizar esta parte es necesario tener acceso a Internet desde nuestro propio portátil (recordemos que todo este proceso se está realizando de forma virtual), para ello es necesario instalar una NAT (Network Address Translation) que nos permita acceder a nuestra red privada a través de una IP pública.

Otro paso previo es configurar una IP a nuestro controlador, ello se puede hacer en GNS3 clicando botón derecho sobre nuestro controlador, editando la configuración del mismo y asignándole una IP estática (también se le puede asignar por DHCP pero ello implica que cada vez que ejecutemos el proyecto la IP cambia). En nuestro caso el fichero de configuración queda de la siguiente forma:

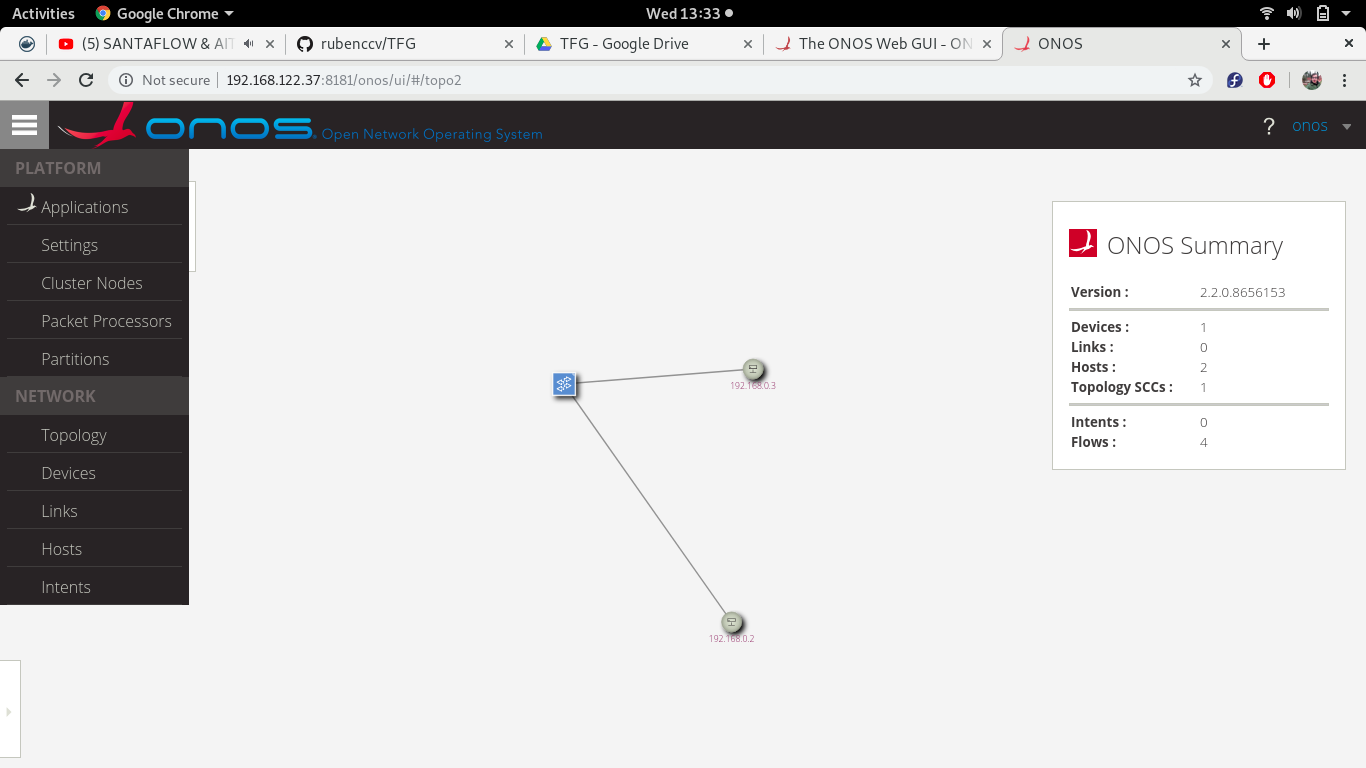


Hemos elegido arbitrariamente la dirección IP 192.168.122.37, dentro de la subred 192.168.122.0/24

Una vez realizados estos pasos previos, ya podemos acceder al controlador para acceder a sus comandos y funciones. Hay 2 formas de acceder a él:

Una de ellas es accediendo desde un host conectado a él a la interfaz gráfica mediante la página web <http://IP-ONOS:8181/onos/ui>. Donde IP-ONOS es la IP que hemos definido previamente en el fichero de configuración. Las credenciales de acceso son user: *onos* y password: *rocks*.

Lo que vemos una vez que accedemos es lo siguiente:

En primer lugar vemos la topología que el controlador ha detectado. En nuestro ejemplo podemos ver una red muy simple que consta de un switch conectado a 2 hosts.

En el recuadro de la derecha podemos ver un resumen acerca de la red, así como los flujos que han generado diferentes aplicaciones.

Finalmente en el menú de la izquierda podemos ver las aplicaciones disponibles, ver en detalle los dispositivos o los hosts...

La segunda opción para acceder al controlador, y la que más usaremos, es mediante consola. Para ello debemos instalar un host que tenga un cliente SSH (el Alpine por defecto no lo trae instalado), así que lo que haremos será modificar la imagen almacenada en Docker.

Para empezar hacemos un *pull* de la imagen alpine original:

docker pull alpine

A continuación, ejecutamos una instancia de dicha imagen en nuestro shell:

docker run --name alpinemodificada2 -it alpine /bin/sh

Una vez dentro de la instancia que acabamos de ejecutar, vamos a instalar las aplicaciones necesarias para nuestro proyecto:

apk add openssh wget curl

Y guardamos la imagen modificada con los paquetes instalados con el nombre deseado:

docker commit alpinemodificada2 alpinemodificada

Por último podemos borrar la instancia ejecutada para liberar recursos:

docker rm alpinemodificada2

Ahora que ya tenemos un host con un cliente SSH instalado, procedemos a acceder desde él al controlador mediante consola con el comando:

ssh -p 8101 [onos@IP-ONOS](mailto:onos@IP-ONOS)

Nuevamente la contraseña de acceso es *rocks*

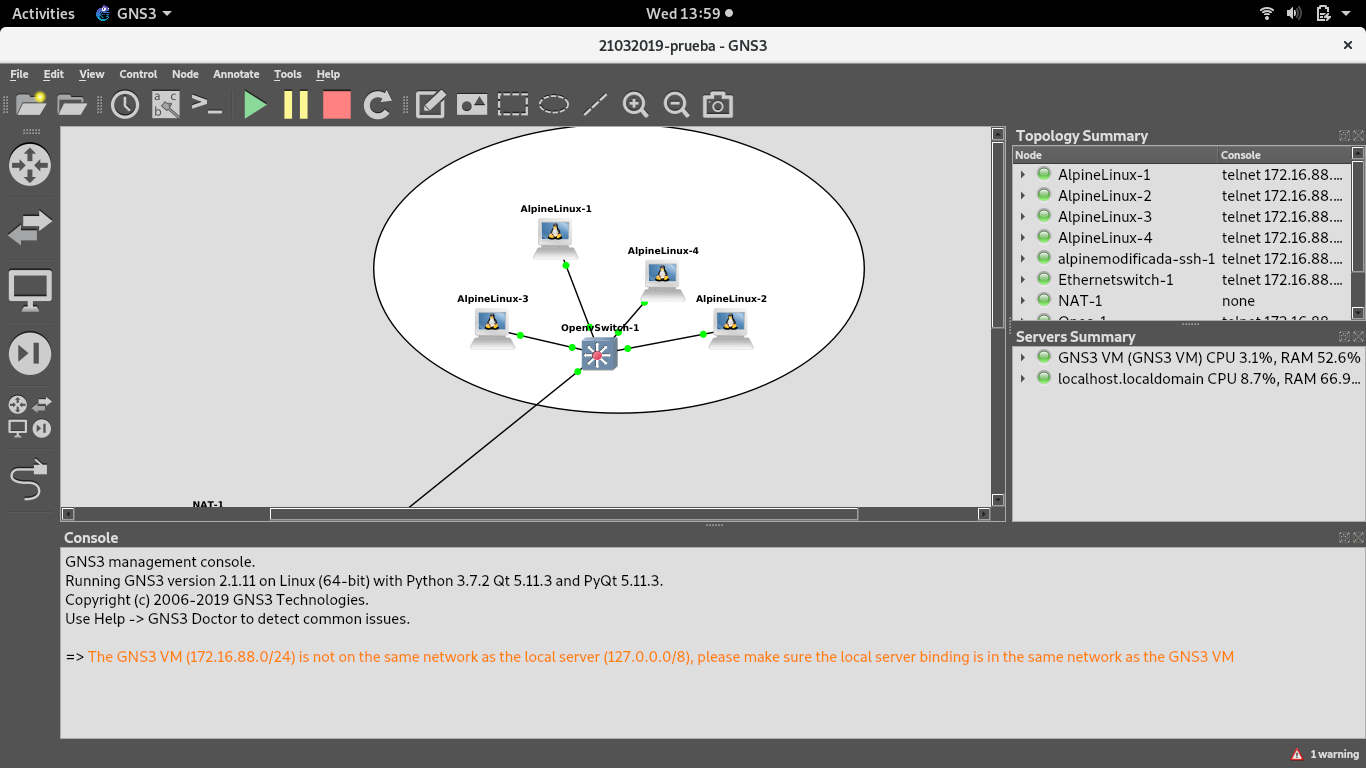
Algunos comandos de utilidad que son necesarios para comprobar el funcionamiento son:

* devices: Devuelve los dispositivos que son controlados por el controlador. En las redes que simularemos son los switches Open vSwitch.
* hosts: Devuelve los hosts que ha reconocido (aquellos que han enviado algún paquete y están conectados a los dispositivos).
* flows: Devuelve los flujos OpenFlow que se han instalado sobre los dispositivos las diferentes aplicaciones y que gestionan el comportamiento de la red.
* log:set debug: Accedemos al modo debug del registro (log) que nos devuelve más información acerca de lo que está ocurriendo en el controlador.
* app activate *nombreAplicacion*: Activa la aplicación en el controlador para que se inicie su funcionamiento.
* app deactivate *nombreAplicacion*: Desactiva la aplicación en el controlador para que se finalice su funcionamiento.

3.2 Hosts conectados a Open vSwitch

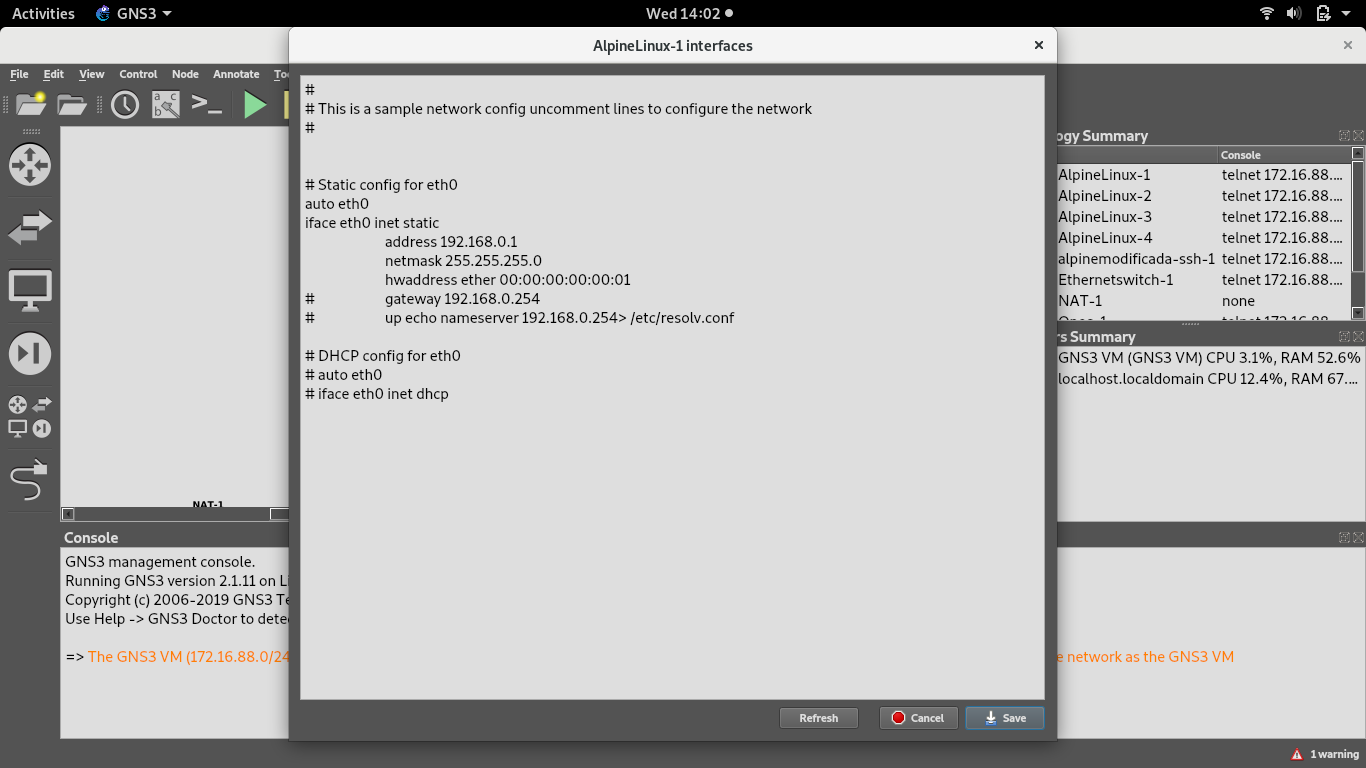
Esta parte de la red es la que nos permitirá probar el funcionamiento de nuestras aplicaciones y se irá modificando en función de los requerimientos de dichas aplicaciones.

Para las primeras tendrá la siguiente topología:



Como podemos ver, tenemos 4 hosts conectados a un openVswitch sobre el que van instaladas las aplicaciones. Para poder trabajar más cómodamente sobre cada uno de los hosts se ha modificado la configuración creando una subred propia.

El fichero de configuración queda de la siguiente forma:



Tal y como se puede ver se le ha asignado una IP y una MAC estática (en orden ascendente por cada host que utilicemos).

Finalmente, para configurar el openVswitch es necesario indicarle quién es el controlador. Como la IP de éste es estática, este proceso sólo tendremos que realizarle una vez, en caso de que se eligiese por DHCP habría que realizarlo cada vez que arrancamos el programa. Para poder indicárselo abriremos una consola desde el openVswitch y ejecutaremos los siguientes comandos:

ovs-vsctl set-manager ptcp:IP-ONOS:6640

ovs-vsctl set-controller br0 tcp:IP-ONOS:6633

Una vez realizado todo esto podremos proceder a realizar nuestras propias aplicaciones y ejecutarlas en esta red.

Capítulo 4: Aplicaciones desarrolladas

En este capítulo se van a detallar las aplicaciones que se han realizado asi como las utilidades que tienen.

Antes de ello, tenemos que crear el proyecto para luego poder modificarlo. Esto se realiza desde nuestra terminal ejecutando el comando.

onos-create-app app org.onosproject nombreApp version org.onosproject.nombreApp

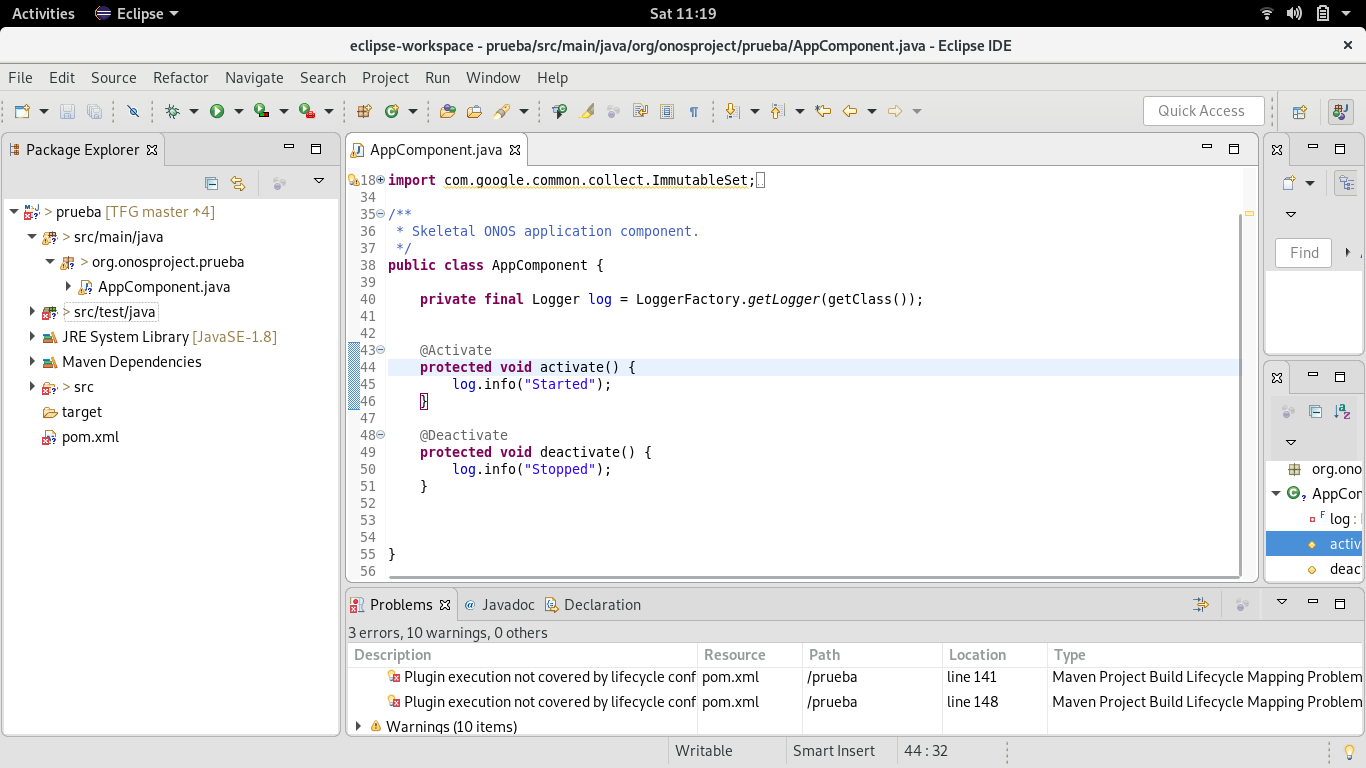
donde nombreApp será el nombre que decidamos dar a nuestra aplicación y la versión es la fase en la que estemos de la aplicación. Si es una aplicación que estamos empezando se pone 1.0-SNAPSHOT.

Una vez ejecutado el comando se crea una carpeta con nuestro proyecto y en primer lugar vemos el fichero *pom.xml*. En este fichero tenemos toda la información acerca de las versiones que estamos utilizando. Si en algún momento se desea cambiar alguna configuración de este estilo se debe modificar en este fichero.

A continuación vamos a la carpeta *src* donde se sitúan los código fuente que podremos modificar. Aparecerá el fichero *AppComponent.java* que es el que vamos a editar en nuestras sucesivas aplicaciones.

Para ello, modificamos el fichero *AppComponent.java* importando el proyecto desde el programa Eclipse.

Nos aparecerá por defecto un código similar al de la Ilustración

En él se destacan los 2 métodos principales: *activate* y *desactivate.* El método *activate* será aquel que se ejecute cuando activamos la aplicación, mientras que el *deactivate* se ejecuta cuando desactivamos la aplicación en ONOS.

Una vez tengamos las aplicaciones finalizadas, o simplemente queramos compilar y enviársela al controlador ONOS lo que debemos hacer es situarnos en la carpeta principal de nuestro proyecto y compilar con el comando:

*mvn clean install*

Esto lo que nos genera es una carpeta *target* con un fichero .oar que será el que enviemos al controlador con el comando:

*onos-app IP-ONOS install! target/fichero.OAR*

La exclamación es opcional e implica que la aplicación se active automáticamente en la aplicación. Si no queremos que ocurra esto simplemente la quitamos manteniendo el resto del comando.

Como detalle, cabe destacar que si la aplicación ya ha sido instalada previamente y estamos enviándola otra vez el comando se modifica ligeramente quedando de la siguiente forma:

*onos-app IP-ONOS reinstall! target/fichero.OAR*

Una vez detallado el proceso previo, nos centramos en las aplicaciones desarrolladas

4.1 Aplicacion *severalpings*

Esta aplicación es una adaptación de la aplicación *oneping (PONER REFERENCIA)* cuyo objetivo principal es limitar el número de pings que se pueden enviar entre 2 hosts cualesquiera.

Esto tiene como principal función el no saturar la red o evitar ataques por inundación.

En primer lugar vamos a detallar el funcionamiento de los *pings*.

4.1.1 Explicación teórica

El *ping* es un comando que permite comprobar si existe conexión entre 2 hosts cualesquiera. Para ello se basa en el envío de paquetes ICMP. El host que ejecuta el comando envia un paquete ICMP de tipo REQUEST al destino. En caso de que el host reciba este paquete responde con un paquete ICMP de tipo REPLY.

La forma de ejecuta el comando en Linux es:

ping -opciones IP -HOST

Y las opciones más utilizadas que usaremos son:

* -c *count:* Indica en la variable *count* el número de pings que se van a enviar.
* -s *size:* Modifica el tamaño del paquete a enviar. Por defecto son 84 bytes

Una vez explicado el funcionamiento teórico se procede a detallar la programación de la aplicación. Para ello se va a dividir la misma en 3 partes diferenciadas que se proceden a detallar.

4.1.2 Interceptar los paquetes

En primer lugar para poder realizar la aplicación es necesario que todos los pings del tipo ICMP REQUEST sean enviados al controlador. Esto tiene como objetivo poder contabilizar los que se han enviado y saber si son más o menos del límite prefijado.

Para poder interceptar estos paquetes es necesario definir un servicio de paquetes (*packetService*) de la siguiente forma:

@Reference(cardinality = ReferenceCardinality.**MANDATORY**)

**protected** PacketService packetService;

Notar que todos los servicios que iremos definiendo tanto en esta como en las aplicaciones sucesivas se definirán de la misma forma.

Con este packetService accedemos al método requestPackets cuya finalidad es enviar los paquetes que cumplan un determinado selector al controlador.

Este selector tiene que identificar los paquetes ICMP tipo REQUEST que lleguen al openVswitch de nuestra red para enviarlos al controlador de la manera que se indica a continuación:

**private** **final** TrafficSelector intercept = DefaultTrafficSelector.builder()

.matchEthType(Ethernet.**TYPE\_IPV4**).matchIPProtocol(IPv4.**PROTOCOL\_ICMP**).matchIcmpType(ICMP.**TYPE\_ECHO\_REQUEST**)

.build();

4.1.3 Procesando los paquetes

Una vez tenemos controlado que todos los ICMP van a llegar al controlador es momento de procesarlos. Para ello definimos una clase que implemente a *PacketProcessor* y comprobamos que de todos los paquetes que llegan al controlador cogemos solo los que nos interesan, esto se accediendo al paquete que se está procesando en cada momento. Para ello definimos el método que is*IcmpPing* que tiene como argumento el paquete Ethernet procesado. Una vez realizada la comprobación se llama al método *processPing*. Este método obtiene la dirección MAC origen y destino del paquete procesado y accede a un hashMap definido previamente que contiene el número de pings realizados previamente entre esa correspondencia MAC origen-destino. Si el número de pings es menor que el máximo predefinido se realizan 2 operaciones:

* Se aumenta en 1 el valor de pings enviados en el hashMap.
* Se crea una tarea utilizando la clase *schedule* programada para el tiempo de baneo máximo prefijado que quite del hashMap el ping enviado.

Por contra, si el número de pings es mayor que el máximo predefinido lo que se hace es bloquear ese paquete y llamar al método banPings.

Este método crea una regla en el controlador que este enviará al openVswitch que regula el tráfico. La parte de código correspondiente a esta parte se presenta a continuación:

TrafficSelector selector = DefaultTrafficSelector.builder()

.matchEthSrc(src).matchEthDst(dst).matchEthType(Ethernet.**TYPE\_IPV4**).matchIPProtocol(IPv4.**PROTOCOL\_ICMP**).matchIcmpType(ICMP.**TYPE\_ECHO\_REQUEST**).build();

TrafficTreatment drop = DefaultTrafficTreatment.builder()

.drop().build();

flowObjectiveService.forward(deviceId, DefaultForwardingObjective.builder()

.fromApp(appId)

.withSelector(selector)

.withTreatment(drop)

.withFlag(ForwardingObjective.Flag.**VERSATILE**)

.withPriority(**DROP\_PRIORITY**)

.makeTemporary(**TIMEOUT\_SEC**)

.add());

Fundamentalmente una regla de flujo se compone de 2 elementos.

* El selector de tráfico que en este caso lo que hace es coger todos los paquetes tipo ICMP REQUEST que vayan entre ese origen y destino, permitiendo de esta forma poder enviar el resto de tráfico (incluidos los ICMP REPLY que se puedan originar de otros pings)
* El tratamiento a realizar para ese tráfico seleccionado previamente. En este caso se trata simplemente de descartarlo, pero otras acciones que pueden ocurrir son por ejemplo enviar el tráfico por otro puerto.

A mayores de estos 2 elementos seleccionamos la prioridad que tendrá la regla (que será superior al de un envío normal) y la temporalidad de la regla que vendrá predefinida previamente.

4.1.4 Escuchando los flujos

Una vez creamos la regla que sabemos que tiene una duración limitada creamos un *Listener* cuya implementación se puede ver más abajo con el objetivo de que esté pendiente de cuando una regla es eliminada (porque ha pasado su tiempo de vida determinado) para, de esta forma, crear un mensaje en el *log* que nos informe que el enlace entre los 2 hosts baneados vuelve a estar disponible para mandar otra vez pings.

flowRuleService.addListener(flowListener);

4.1.5 Parámetros configurables

Finalmente dentro de esta aplicación se han creado 2 parámetros configurables mediante la CLI de onos para evitar tener que modificar el código en caso de querer variar su valor. En concreto los parámetros configurables son el número máximo de pings que permitiremos entre 2 hosts y el tiempo que mantendremos el enlace baneado para el envío de mas pings. Los valores por defecto se pueden encontrar en el fichero *OsgiPropertyConstants.java* y son 7 pings como máximo y 60 segundos de baneo.

La forma de declarar parámetros configurables es así:

@Component(

immediate = **true**,

service = AppComponent.**class**,

property = {

**MAX\_PINGS** + ":Integer=" + **MAX\_PINGS\_DEFAULT**,

**TIME\_BAN** + ":Integer=" + **TIME\_BAN\_DEFAULT**,

}

)

Resaltando que el servicio que hemos de declarar es cfgService

Cuando hayamos declarado los parametros declaramos un método modified cuya implementación puede verse en el código que se ocupa de actualizar el valor de los parámetros una vez los hayamos cambiado.

Para poder ver los parámetros que tiene una aplicación concreta y modificarlos es necesario ir a la CLI de ONOS. Los comandos fundamentales son:

* cfg: Lista todos los nombres de las clases que tienen parámetros configurables
* cfg get componentClass: Lista todas las propiedades de la clase especificada
* cfg get componentClass name: Lista el valor de la propiedad especificada
* cfg set componentClass name value: Modifica el valor de la propiedad especificada
* cfg set componentClass name: Reestablece el valor de la propiedad especificada a su valor por defecto

PONER CAPTURA DE LA ALPINEMODIFICADA CON ESTOS COMANDOS PARA QUE QUEDE MAS CLARO

4.1.6 Banco de pruebas

En este apartado final se va a probar la aplicación mostrando fundamentalmente capturas de la red y del funcionamiento de la aplicación.

4.2 Analizando las estadísticas de tráfico

Con el objetivo de analizar las estadísticas de tráfico y poder usarlas para determinados objetivos se han realizado 3 aplicaciones que se proceden a explicar:

4.2.1 Aplicación *statsshow APLICACION TERMINANDA EXPLICACION Y CODIGO*

El objetivo de esta aplicación es informativo ya que muestra las estadísticas de tráfico de todos los puertos de los dispositivos que controlemos. En este caso los puertos del Open vSwitch. Esta aplicación es muy sencilla y únicamente tiene un apartado.

4.2.1.1 Tareas repetidas

Para llevar a cabo el funcionamiento correcto de la tarea es necesario crear una tarea que se repita periodicamente. Esto se hace creando un temporizador de la siguiente forma:

timer = new Timer("Timer");

Y le definimos una periodicidad y un retardo:

timer.scheduleAtFixedRate(repeatedTask, delay, period);

Donde:

* repeatedTask: es el nombre de la tarea y dentro de ella se ejecuta el método *run* donde está implementado el código que se debe repetirse.
* Delay: Es un atributo de tipo long que indica el retraso desde que se activa la aplicación hasta que se ejecuta por primera vez la tarea. Está definido a un segundo.
* Period: Es otro atributo de tipo long que indica la periodicidad de la tarea, es decir, cada cuanto tiempo se repite la tarea. Este atributo es un parámetro configurable, por tanto puede modificarse desde la CLI de ONOS tal y como se ha explicado en el apartado 4.1.5

Es importante destacar que en el método *deactivate* hay que cancelar el temporizador, de lo contrario la tarea se quedará ejecutandose indefinidamente. Para ello se ejecuta la orden:

timer.cancel();

Dentro de la tarea el objetivo, como se ha comentado, es obtener las estadisticas de tráfico de los puertos de cada dispositivo. Por tanto es necesario recorrer 2 bucles *for*; el primero que recorra todos los dispositivos y el segundo que recorra todos los puertos de ese dispositivo. Tanto los dispositivos existentes como los puertos se pueden obtener a través de *deviceService*

Una vez creados los 2 bucles necesarios obtenemos las siguientes estadisticas, en concreto obtendremos las siguientes estadísticas:

* bytes enviados por cada puerto desde que se activó la aplicación
* bytes recibidos por cada puerto desde que se activó la aplicación
* bytes enviados por cada puerto en cada ejecución de la tarea
* bytes recibidos por cada puerto en cada ejecución de la tarea

4.2.2 Aplicación detectHost

El objetivo de esta aplicación es conocer todos los hosts que tenemos conectados a los Open vSwitch de la red.

Hay que resaltar que para el buen funcionamiento de la aplicación es necesario instalar una aplicación predefinida en el controlador a parte de las habituales ya comentadas, esta es la app *hostprobingprovider*.

Para ello, la aplicación se divide en 2 partes:

4.2.2.1 Tarea repetida

En este caso, a diferencia de la aplicación anterior, se utilizan 2 tareas repetidas, dentro de la primera se utiliza el servicio definido previamente *hostService*. Este servicio nos devuelve todos los hosts que procedemos a detallar mediante un bucle que los recorra. Cabe destacar que para que un host sea detectado por este servicio es necesario que haya mandado algún paquete al controlador.

En la segunda tarea, lo que se hace es, mediante el servicio *hostProbing* mandar un paquete de prueba a todos los hosts para ver si siguen activos, o por el contrario se han desconectado. Nuevamente destacar que para que un hots se reconozca como desconectado debe estarlo físicamente, es decir, no estar conectado mediante un cable al Open vSwitch.

Finalmente, recordar que es necesario en el método *deactivate* cancelar los 2 temporizadores creados para evitar que se queden residuales.

4.2.2.2 Listener

Además de las tareas repetidas, el código incluye un *HostListener* cuyo objetivo es estar pendiente de cuando un host sea eliminado notificarlo mediante un mensaje informativo al usuario que aparecerá en el controlador ONOS.

Destacar que como HostListener viene definido como una interfaz es necesario instanciarlo previamente de la forma que procede:

private final HostListener hostListener = new InternalHostListener();

4.2.3 Aplicación detectHostBan

En esta aplicación el objetivo es realizar un seguimiento de los hosts conectados a la red actual y banear a aquellos que superen un determinado umbral durante un cierto umbral. La finalidad más clara es evitar que los hosts envíen una cantidad muy elevada de datos.

Para el desarrollo de la aplicación se utilizarán aspectos de las 2 aplicaciones mencionadas anteriormente ya que, en primer lugar se tienen que detectar los hosts que hay en la red para lo cual se usará lo explicado en la aplicación detectHost.

En segundo lugar, es necesario, para todos los hosts detectados conocer sus estadísticas, para lo cual se usará código de la aplicación statsshow, utilizando las delta estadísticas dentro de un temporizador. En cada iteración del bucle iremos sumando el valor e introduciendolo en un HashMap que relaciona la MAC del host con los datos acumulados que lleva desde que se activó la aplicación.

Una vez se ha superado el umbral, todo el tráfico es baneado. Destacar, que a diferencia de la aplicación *severalpings* se banea todo tipo de tráfico bidireccionalmente, es decir, el host no puede ni enviar ni recibir dato alguno durante un tiempo que está definido como un parámetro configurable del estilo a aplicaciones anteriores, para ello se llama al método *banDatas.*

Este método crea 2 reglas de flujo en el Open vSwitch,:

El objetivo del selector es capturar todo el tráfico que pase por el enlace; En la primera regla se captura el tráfico cuyo origen es la MAC baneada, mientras que en el segundo se captura todo el tráfico cuyo destino es la MAC baneada, de esta forma baneamos bidireccionalmente

El tratamiento de estas reglas es idéntico y consiste simplemente en descartar ese tráfico seleccionado.

Finalmente, hacemos que estas reglas sean permanentes y ponemos una tarea programada para el tiempo que hemos definido que la MAC este baneada.

El motivo principal por el que no se hace temporal se explica fácilmente con un breve ejemplo:

Supongamos que tenemos 2 hosts: host A y host B. El host A está enviando una serie de pings y supera el límite predefinido. Se llamaría al método *banDatas* y se crearían las reglas que descartan todo el tráfico.

Supongamos tambien que el tiempo de baneo es mayor que el tiempo de la tarea que actualiza las estadísticas y que el comando *ping* se sigue manteniendo en ejecución (aunque no se esté recibiendo respuesta). Por cada iteración de la tarea se volvería a ejecutar el método *banDatas* y se volverían a crear los flujos, otra vez con el temporizador de baneo reiniciado, por tanto nunca se eliminarían estos flujos y el baneo pasaría a ser a todos los efectos permanente.

Para evitar este problema, directamente creamos las reglas de flujo permanentes y creamos una tarea programada para el tiempo de baneo con el comando timer.schedule().

Dentro del este método se ejecuta a su vez método *run* que lo que hace es eliminar las 2 reglas y actualizar el HashMap volviendo a reiniciar el valor de los datos acumulados para la MAC que ya ha dejado de ser baneada.

Una opción alternativa para este proceso sería enviar todo el tráfico al controlador, hacer la regla temporal y bloquear los paquetes mediante un procesador del mismo con la orden *block,* pero esto aumentaría el flujo de datos enviado al controlador haciendo que se ralentizase la aplicación.

4.3 Aplicación *Vlan*

La siguiente aplicación desarrollada permite asignar diferentes Vlan a los hosts.

4.3.1 Explicación teórica

Una Vlan (red de área local virtual) es una forma de crear redes lógicas independientes dentro de una misma red física. Es decir, poder tener diferentes redes dentro de un mismo enrutador. Esto permite simplificar la administración de la red ya que permite separar que hosts no deben intercambiar datos. Por tanto, en resumen las ventajas que aporta son las siguientes:

* Mayor flexibilidad ya que se facilita el cambio y moviemiento de los dispositivos de la red. Simplemente con cambiar la Vlan en la que se encuentra un host cambiamos por completo la topología de la red virtual
* Mayor seguridad ya que los dispostivos están separados en diferentes Vlan, por tanto su comunicación está mas restringida
* Control del tráfico broadcast, ya que entre subredes no se produce.

A la hora de configurar las Vlan se utiliza el protocolo IEEE 802.1Q. Su propuesta principal es añadir 4 bytes a una trama Ethernet (en lugar de encapsularla)

PONER IMAGEN FORMATO DE TRAMA ETHERNET Y 802.1Q y ver diferencias

Este protocolo permite identificar a una trama perteneciente a una Vlan, de esta forma el tráfico se mandará unicamente por su Vlan.

Para interconectar Vlan se añadió un tipo de puerto en los switches:

* Por un lado están los puertos de acceso, estos conectan con los hosts directamente. Cuando entra una trama Ethernet se le añade la etiqueta de acuerdo con el protocolo 802.1Q, mientras que cuando sale una trama 802.1Q se le quita la etiqueta de tal forma que al host final le llega la trama Ethernet original.
* Por otro lado, están los puerto 1Q Trunk que son los que se utilizan para conectar los switches entre sí y que el tráfico de varias VLANs circule a través de él. En este caso, tanto las tramas entrantes al switch como las salientes van etiquetadas según 802.1Q

Por tanto, el objetivo de esta aplicación será implementar el funcionamiento de este protocolo estableciendo reglas de flujo.

4.3.2 Desarrollo de la aplicación

El funcionamiento de la aplicación se va a ir explicando paulatinamente.

Antes de nada, declaramos 3 HashMaps que serán necesarios:

* *macVlanMap*: HashMap que relaciona la Mac de un host con la Vlan a la que pertenece. A la hora de introducir los datos lo haremos con un comando que crearemos especificamente para este fin (ver sección PONER SECCION DONDE SE EXPLICA CUANDO ESTE)
* *vlanRuleMap*: Este HashMap devuelve las reglas de flujo que tiene cada Vlan creada. Más adelante se verá el fin de él
* *macRuleMap*: Este HashMap devuelve para cada Mac todas las reglas de flujo que tiene asociadas.

En primer lugar, nada mas activar la aplicación capturamos todo el tráfico ARP para mandarlo al controlador y que sea este el que decida que hacer con él. Esta regla será fundamental como se verá posteriormente y se instalará en todos los dispostivos que tengamos.

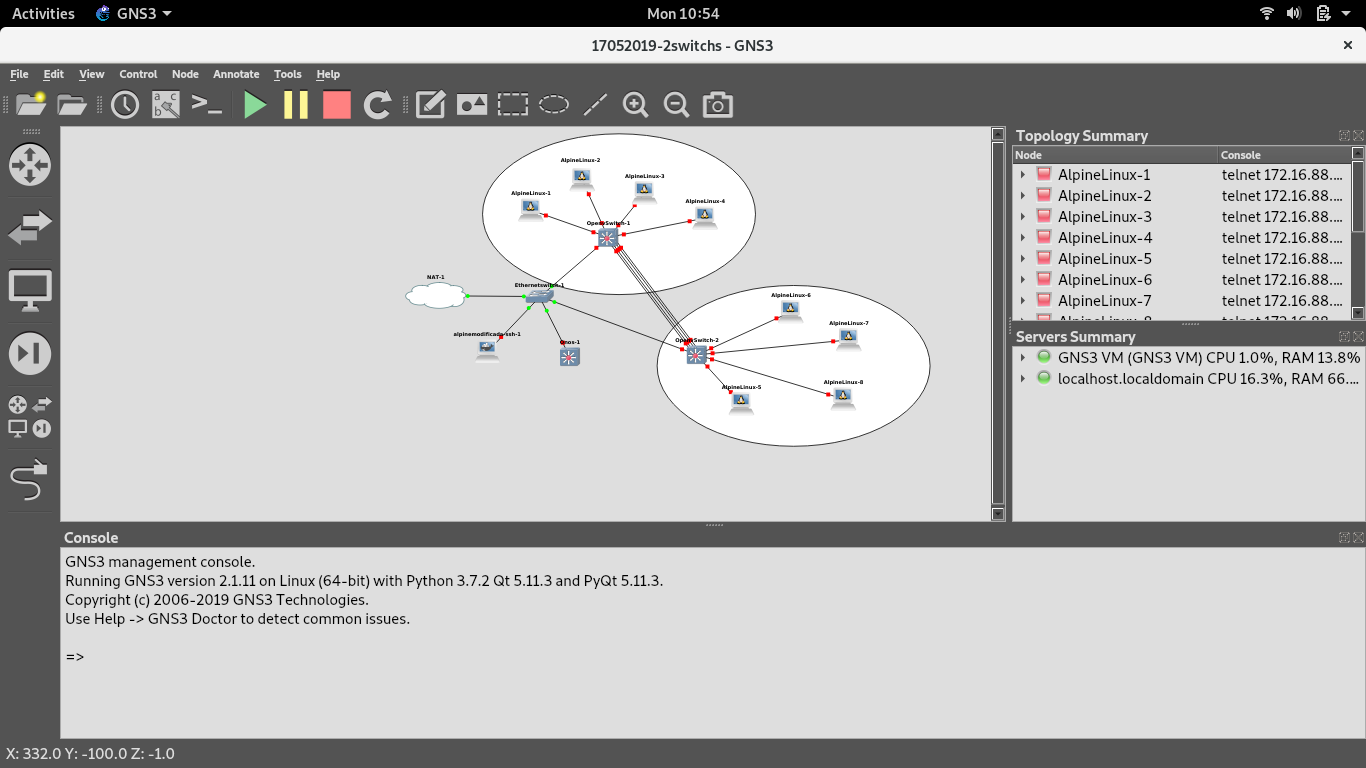
Seguidamente, añadimos tambien un *HostListener,* que tendrá como objetivo notificarnos cuando se añade un host nuevo a la red.

En cuanto se añade un host comprobamos la VLAN en la que está mediante *macVlanMap*

MIRAR EN CASA LA HOJA CON EL DESARROLLO APUNTADO

4.4 Aplicación fwdBalanceo

Esta aplicación es diferente, puesto que es una modificación de la fwd que contiene el controlador ONOS por defecto. El objetivo es realizar un balanceo de carga cuando tenemos varios enlaces que conectan 2 open V switch. Para ello, en primer lugar modificamos la red que tenemos en el GNS3 sustituyendola por la mostrada en la Ilustración



Como vemos, tenemos 2 open V switch conectados por 4 enlaces que será el punto donde realizaremos el balanceo de carga entre las 2 subredes que conectan 4 hosts cada una

Como se ha comentado, lo que se va a hacer es modificar la aplicación fwd. Esta aplicación la podemos encontrar dentro de la carpeta Home/onos/app que copiaremos y llevaremos a nuestro directorio de trabajo. Seguidamente modificamos el pom.xml para cambiar el nombre de la aplicación dejandolo de la siguiente forma:

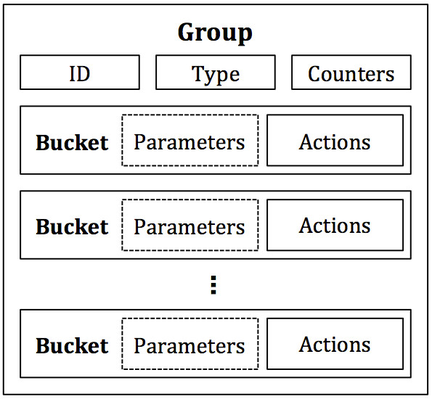


REMARCAR LOS CAMBIOS MODIFICANDO LA CAPTURA

Finalmente, modificamos el nombre de las carpetas que tienen el nombre de la aplicación sustituyendolas por fwdBalanceo (lo que modificará el paquete en el que está contenido), y una vez hecho esto ya podemos empezar a realizar nuestra aplicación, en concreto se van a modificar el *activate* y el método *installRule* del fichero java *ReactiveForwarding*

4.4.1 Explicación teórica

En este apartado se va a realizar la explicación acerca de como realizar el balanceo de carga. Para ello es necesario que los puertos que conecten ambos open Vswitch (puertos troncales) estén asociados un grupo. Un grupo es una abstracción que permite realizar operaciones más complejas (como lo es esta) que no se pueden realizar mediante las reglas de flujo que se han visto hasta el momento. Tampoco se pueden enviar a otras tablas. Cada grupo contiene una lista separada de acciones (*bucket list*). En nuestro caso el bucket serán los puertos por los que se enviará tráfico en cada momento en función del tráfico existente en la red.



Tal y como se vem un grupo está formado por un tipo. En concreto, hay 4 tipos diferentes que puede tener que son:

* ALL group: Si seleccionamos esta opción cuando un paquete llegue como entrada al grupo se duplicará para que se opere sobre él independientemente. De esta forma podemos operar en copias separadas del paquete original. Estas operaciones estarán definidas en las acciones en cada grupo. En este caso, el bucket solo está compuesto de acciones y no hay parámetros especiales.
* SELECT group: Esta opción está diseñada para equilibrar la carga. En este caso el parámetro consiste al peso que tendrá cada *bucket.* El algorimo de selección depende de la implementación del switch. En el apartado APARTADO BANCO DE PRUEBAS se indica como se realiza la selección de los buckets del grupo para el switch que manejamos, ya que esta es la sentencia que vamos a usar para nuestra aplicación
* INDIRECT group: Este caso es un poco especial, ya que solo contiene un único *bucket ,* por lo tanto siendo estrictos es díficil verlo como un grupo. Su fin es encapsular un conjunto de acciones utilizadas en muchos flujos. Por ejemplo, si tenemos diferentes tipos de flujos (por ejemplo que sean tipos diferentes de tráfico) pero las acciones que deben llevar son las mismas, se pueden enviar al grupo en vez de indicar la acción para cada flujo independiente. De esta forma creamos menos reglas de flujo y se ahorra en memoria.
* FAST-FAILOVER group: Este tipo de grupo está diseñado para detectar y superar fallos en los puertos. Cada *bucket* tiene un puerto de vigilancia que controla el “estado de vida” del puerto indicado. Si se considera que la vida está baja no se utilizará ese *bucket,* mientras que si su nivel de vida está alto se usa ese *bucket.* Como detalle indicar que solo se puede usar un *bucket* y que este no cambia a menos que se detecte que la vida del mismo desciende.

4.4.2 Método activate

En este método en primer lugar definimos como parámetros configurables los puertos que van a ser troncales. Se ponen como configurables para que sea más sencillo en caso de que se modifique la topología de la red modificarlos, y así mantener el código. En nuestro caso, los puertos seleccionados para ser troncales son los 10, 11, 12 y 13 de la topología del open V switch. A la hora de programar hay que sumar 1 a estos valores ya que no contabiliza el 0 como puerto y empieza en el 1 directamente.

Por tanto, nada más activar la aplicación creamos el primer grupo. Como el proceso puede parecer un poco complejo se va a explicar detalladamente.:

Un grupo se crea a partir de un constructor que contiene un GroupId (arbitrariamente hemos elegido el 137, y un *DefaultGroupDescription*:

DefaultGroup dgd = **new** DefaultGroup(**new** GroupId(137),degd);

A su vez, un *DefaultGroupDescription* tiene varios constructores. Usaremos el utilizado para las aplicaciones de la API del norte. Está formado por el dispositivo, el tipo de grupo que va a ser, los *GroupBuckets*, una clave para el grupo, el groupId definido previamente y la aplicación en la que se va a instanciar.

DefaultGroupDescription degd = **new** DefaultGroupDescription(d.id(), GroupDescription.Type.***SELECT***,gbs,k,137,appId);

Nuevamente, a su vez, los *GroupBuckets* está formado por un ArrayList que contiene todos los buckets individuales que tiene el grupo

gbs = **new** GroupBuckets(listGroup);

List<GroupBucket> listGroup = **new** ArrayList<GroupBucket>();

Finalmente, a este ArrayList de GroupBucket creado le iremos añadiendo cada bucket individual:

GroupBucket gb = DefaultGroupBucket.*createSelectGroupBucket*(sendPort);

Cada GroupBucket tendrá como argumento la acción a llevar por él, en nuestro caso enviarlo por el primer puerto troncal que tenga la red:

TrafficTreatment sendPort = DefaultTrafficTreatment.*builder*()

.setOutput(PortNumber.*portNumber*(puerto))).build();

Por tanto, lo que hemos hecho es crear un grupo que por el momento tiene un único bucket, que consiste en enviar por el puerto 10 cuando se utilice.

El siguiente paso es determinar 3 umbrales de tráfico acordes (al igual que los puertos son parámetros configurables). Cuando el tráfico agregado en toda la red supere un determinado umbral añadiremos un nuevo bucket que contenga el siguiente puerto.

Para poder obtener el tráfico agregado creamos una tarea y en ella obtenemos las delta estadísticas de cada puerto que iremos almacenando de acuerdo a la siguiente ecuación:

bitsPorSegundo=bitsPorSegundo+(traffic.bytesSent()\*8)/temporizadorTarea;

PONER MODO ECUACION EN WORD

Para añadir *buckets* simplemente creamos un nuevo tratamiento con el puerto correspondiente que esté almacenado en un nuevo *GroupBucket* ylo añadiremos al ArrayList ya creado.

Una vez tenemos actualizado en cada iteración del temporizador predefinido los *buckets* por los que se mandará el tráfico entre los 2 open Vswitch hay que crear las reglas de flujo que indiquen que el tráfico se mande por el grupo en vez de por los puertos propiamente dicho. Esto se tiene que realizar en el método *installRule.*

4.4.3 Método installRule

Dentro de este método el objetivo es crear una regla que indique que cuando el tráfico tenga como destino alguno de los puertos troncales, lo cual haremos dentro de una sentencia *if.*

El tratamiento cuando entremos en esta sentencia condicional será indicar al tráfico que se envie por el grupo de la siguiente forma:

TrafficTreatment treatment= DefaultTrafficTreatment.*builder*()

.group(**new** GroupId(137))

.build();

Esto, como se ha comentado, lo que hace es enviar por al grupo previamente creado y que sea este el que según el tipo y los *buckets* que contenga.

Por otro lado, en el caso de que el tráfico tenga como destino cualquier otro puerto el tratamiento a seguir será enviarlo por el puerto de salida que haya aprendido simplemente.

4.5 Aplicación FakeDHCP

4.5.1 Explicación teórica

En primer lugar se va a explicar el funcionamiento del protocolo DHCP (*Dynamic Host Configuration Protocol)*  es un protocolo de la capa de aplicación en el cual un servidor DHCP asigna de forma dinámica direcciones IP a los dispositivos de una red para poder comunicarse. Este protocolo permite 3 formas de asignación de direcciones IP:

* Asignación manual: Se asigna una IP a una máquina de forma estática, es decir, cada vez que se conecte recibirá la misma dirección
* Asignación automática: Se asigna una IP a una máquina la primera vez que hace la solicitud al servidor y se mantiene hasta que se desconecta
* Asignación dinámica: En este caso el administrador determina un rango de direcciones IP y cada máquina solicita su dirección IP al servidor cuando la tarjeta de interfaz de red se inicializa y durante un tiempo limitado. Cuando este tiempo finaliza la IP es eliminada y la máquina no puede funcionar hasta que pida otra.

El funcionamiento del protocolo se basa en el envío de 4 tramas diferentes intercambiadas entre cliente y servidor. Cabe destacar que el servidor utiliza el puerto 67 UDP y el cliente utiliza el puerto 68 UDP

* DHCP Discovery: Es una solicitud DHCP realizada por un cliente con la dirección destino MAC broadcast para que el servidor DHCP le asigne una dirección IP y el resto de parámetros como servidor DNS
* DHCP Offer: Es el paquete de respuesta del servidor DHCP ante la petición previa en el que le ofrece una dirección IP. Contiene una oferta de configuración con todos los detalles que el servidot tiene almacenados para el cliente. Este puede aceptarla o ignorarla
* DHCP Request: El cliente confirma la oferta recibida con sus detalles
* DCHPACK: El servidor confirma la asignación y una vez recibida el cliente empieza a usar la red.

PONER CAPTURAS WIRESHARK DEL INTERCAMBIO DE TRAMAS

Una vez explicado el funcionamiento del protocolo se va a detallar el de la aplicación en concreto:

Esta aplicación está basada en aumentar la seguridad de la red con el objetivo de evitar conectarnos a servidores DHCP no fiables.

Este fenómeno conocido como *rogue DHCP* tiene el siguiente funcionamiento:

En primer lugar el cliente se conecta a la red, en este momento tanto el servidor DHCP legítimo como el falso que está conectado a la red les ofrecen direcciones IP entre otros. En el momento en el que algún cliente acepta la dirección IP del DHCP falso puede detectar todo el tráfico que envia violando todo tipo de privacidad y pudiendo acceder a datos comprometidos o realizar ataques del tipo *man in the middle*.

4.5.2 Procesador de paquetes

Para el funcionamiento de la aplicación es necesario crear un procesador de paquetes muy similar al utilizado en la aplicación *severalpings*.

En primer lugar se interceptaran todos los paquetes que sean del tipo DHCP. Como es un protocolo de la capa de aplicación ONOS no llega a sus cabeceras ya que se queda en el nivel 3, por tanto, lo que haremos será interceptar todo el tráfico que tenga como origen el puerto UDP 67 y destino el puerto UDP 68

TrafficSelector selector = DefaultTrafficSelector.*builder*()

.matchEthType(Ethernet.***TYPE\_IPV4***)

.matchIPProtocol(IPv4.***PROTOCOL\_UDP***)

.matchUdpDst(TpPort.*tpPort*(UDP.***DHCP\_CLIENT\_PORT***))

.matchUdpSrc(TpPort.*tpPort*(UDP.***DHCP\_SERVER\_PORT***))

.build();

Cuando el paquete está en el controlador añadimos un procesador de paquetes que será instanciado:

**private** **final** PacketProcessor packetProcessor = **new** DHCPProcessor();

Dentro del método DHCPProcesor lo primero es comprobar que de todo el tráfico que llega al controlador estamos seleccionando el DHCP. Para ello habrá que obtener el paquete que acaca de llegar y obtener la carga útil e igualarlo al mensaje que queremos comprobar, DHCPOffer

((DHCP)(((UDP)((IPv4) eth.getPayload()).getPayload())).getPayload()).getPacketType() == DHCP.MsgType.***DHCPOFFER***;

Cuando se ha comprobado que efecivamente, el paquete que se está procesando es un DHCPOffer comprobamos el puerto por el que venía ese paquete. Si e el mismo que aquel que tenemos prefijado (el puerto del router es un parámetro configurable) permitimos el envío del paquete, en caso de que ese DHCPOffer venga por cualquier otro puerto se bloquea, el host no recibe ese paquete, por tanto no acepta la oferta y se evita que cualquiera que intente realizar un ataque a nuestra red.

Referencias

<https://es.wikipedia.org/wiki/Redes_definidas_por_software>

[https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html#~stickynav=1](https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html" \l "~stickynav=1)

<https://openzen.wordpress.com/2015/02/12/historia-del-sdn/>

<https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/>

<https://blogthinkbig.com/sdn-software-defined-networking-cambiando-de-paradigma-en-la-red>

[https://riunet.upv.es/bitstream/handle/10251/16310/Art%C3%ADculo%20docente%20configuraci%C3%B3n%20b%C3%A1sica%20VLANs.pdf](https://riunet.upv.es/bitstream/handle/10251/16310/Artículo docente configuración básica VLANs.pdf)

<https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/7995427/How+to+Work+with+Fast-Failover+OpenFlow+Groups>