TRABAJO FIN GRADO

CONSTRUCCIÓN DE UN EQUIPO DE COMUNICACIONES SDN Y VALIDACIÓN DEL COP COMO INTERFAZ PARA CONTROLADORES SDN

18 de Diciembre de 2015

José Manuel Gran Josa

Tutores:

Ignacio de Miguel Jiménez

Ramón J. Durán Barroso

Óscar González de Dios

ÍNDICE

Construcción de un equipo de comunicaciones SDN y validación del COP como interfaz para controladores SDN 1
1. Agradecimientos 4
2. Resumen 5
3. Abstract 5
4. Introducción 6
4.1. Estructura 6
4.2. SDN 7
4.3. OpenFlow 8
4.4. Artuitectura PCE 8
5. Construcción de un equipo de comunicaciones SDN 10
5.1. Equipos de trabajo 10
5.2. Test de rendimiento 12
5.2.1. Test del procesador 12
5.2.2. Test de escritura en disco 13
5.3. Test de red 14
5.3.1. Iperf 15
5.3.2. TCP 19
5.3.3. UDP 26
5.3.4. TCP vs UDP 29
5.4. Switch virtual 31
5.4.1. OpenVSwitch 31
5.4.2. Controlador externo 32
5.4.3. Openvswitch - Controlador externo 37
5.5. Replica soekris 39
5.5.1. Preparando Imagen 39
5.5.2. Replicando Imagen 41
5.5.3. Configurando equipo 42
5.5.4. Conexión al controlador externo 43
5.6. Conclusiones 45
6. Implementación SDN 46
6.1. Conexión PCE – Controlador externo Floodlight 46
6.2. Proyecto europeo Strauss 48
6.3. Preparando escenario 49
6.4. Demostración COP 54
6.5. Conclusiones 57

6.6. Paper OFC 2016 58
6.6.1. Resúmen 58
6.6.2. Introducción 58
6.6.3. VM dinámicas y servicio de transporte E2E desarrollado con provisionamiento con QoS 59
6.6.4. Conclusiónes 60
7. Referencias 61

1. AGRADECIMIENTOS

Para finalizar esta memoria del trabajo de fin de grado, me gustaría agradecer a todas las personas que me han aportado tiempo, conocimiento y cariño para poder llegar hasta aquí. Mi primera dedicatoria tiene que ser para mis padres, que sin ellos nada de esta hubiera sido posible, a los que estoy enormemente agradecido de que me hayan convertido en una persona con valores humildes, con los que con trabajo y esfuerzo todo se puede conseguir. A mi hermana y mis abuelos que han sido otro pilar fundamental en mi desarrollo como persona, especialmente a mi abuela que cada día la echo más de menos. También tengo palabras de agradecimiento a mis amigos de Salamanca del colegio y a los nuevos amigos de esta etapa universitaria, de la facultad y por supuesto de la Residencia de Estudiantes "SANTIAGO", en la que convivir con ellos durante los últimos años ha sido una de las experiencias más inolvidables. Destacar la labor de todos y cada uno de los profesores que conformáis a esta escuela, que con la suma de todos he podido lograr acabar obteniendo esta titulación. Especialmente agradecer a los tutores que me han ayudado en este trabajo, al Prof. Ignacio de Miguel y al Prof. Ramón Durán, por su dedicación, ayuda y conocimiento en este año. Por último también dar las gracias a Óscar González de Dios que me dio la posibilidad de realizar un beca en Telefónica I+D en la que bajo su supervisión y la de Víctor López Álvarez y junto todos los demás compañeros, estoy disfrutando y aprendiendo día a día.

A TODOS ELLOS, GRACIAS.

2. RESUMEN

En este trabajo que vamos a presentar a continuación sobre tecnología SDN, podemos diferenciar dos partes. Una primera de ellas en la que vamos a comentar paso a paso la construcción de una maqueta SDN basada en equipos de comunicaciones Soekris y una segunda parte, en la que también dentro de la tecnología SDN participamos en una demostración *multipartner* sobre un protocolo de orquestación para controladores SDN. Volviendo a la primera parte, para construir dicha maqueta SDN partiendo de unos primeros análisis intrínsecos de los dispositivos, vamos a disponer de un switch virtual OpenFlow gracias a la utilidad OpenVSwitch, en el que a través de su tabla de flujo vamos a poder realizar ciertas operaciones con los datos que atraviesan dicho dispositivo, modificando dichos parámetros a través de un controlador externo. También comentaremos como a partir de un dispositivo, replicar la información para disponer de la misma configuración y utilizar varios switches para utilizar topologías más complejas. Una vez dispuestos estos dispositivos conseguimos hacer una integración con servicios SDN como un PCE. En una segunda parte, aunque no seguimos utilizando los equipos construidos, si utilizamos el concepto del controlador y colaboramos en la validación de un protocolo innovador, COP (*Control Orchestration Protocol*), que funciona como una API de transporte para manejar un conjunto de funciones de plano de control. Esta demostración plantea una orquestación integrada para los recursos de la red utilizando una sintaxis común para el provisionamiento y servicios de recuperación extremo a extremo con QoS.

3. ABSTRACT

In this document that is going to presented below about SDN technology, it is possible to distinguish two different parts. The first one describes step by step the building of a SDN model Soekris communications devices based, and the second one, within SDN technology too, it consist a multipartner demonstration about an orchestration protocol for SDN controllers. Going back to the first part and starting with the first tests of devices to build this SDN prototype, it is going to be got an OpenFlow virtual switch through OpenVSwitch utility. With this utility's flow table it is going to be possible to perform different operations in the network data, modifying this parameters using an external controller. In the next section, it is explained how to replicate the information of all the software in order to get others virtual switches to use more complex networks. After the building of those devices, it will be shown the integration of SDN services such as PCE. In this second part, although the built devices will not be used anymore, the controller concept is still used and we collaborate in the validation of an innovative protocol called Control Orchestration Protocol (COP), which works like a transport API to manage a set of control plane functions. This demonstration contemplates an integrated orchestration for network resources using a common syntax to provision and recovery QoS-aware end to end services.

4. INTRODUCCIÓN

En estas primeras páginas de la memoria final del trabajo de fin de grado, vamos a realizar una introducción sobre el concepto de la tecnología SDN (*Software-defined networking*) y algunos protocolos relacionados, aunque en primer lugar vamos comentar la estructura que dispone este documento.

4.1. ESTRUCTURA

En el presente documento sobre la realización del trabajo de fin de grado vamos a encontrar la siguiente estructura en cuanto al contenido.

En primer lugar una introducción en la que vamos a hablar sobre conceptos un poco más generales o tecnologías de lo que luego desarrollaremos como parte de este trabajo. En esta introducción hablaremos de la tecnología SDN, explicando su concepto general y su arquitectura. En el siguiente punto pasaremos a hablar acerca del protocolo OpenFlow para la comunicación entre el controlador externo y el switch virtual. Seguiremos comentando sobre la arquitectura PCE, el protocolo PCEP y el elemento ABNO.

Después de dicha introducción pasamos al trabajo en sí de dicho TFG, desarrollando paso a paso la construcción de un equipo de comunicación SDN a partir de dispositivos Soekris. En ese capítulo (5), en primer lugar veremos una sección (5.2) donde realizamos un análisis de las características de los dispositivos y cuantificaremos los recursos que podemos disponer a la hora de crear redes con ellos. Dentro de esté análisis de los recursos, en primer lugar veremos la realización de un test de rendimiento (5.2) (CPU y escritura en disco) y en segundo lugar la realización de test de red (5.3). En este test de red, realizaremos multitud de pruebas, variando los protocolos, las interfaces de red, el tamaño de los datagramas enviados, etc.

Siguiendo la construcción de esta maqueta, en el punto 5.4, podremos leer como disponer en el dispositivo una utilidad (OpenVSwitch) que dotará de un switch virtual que junto a un controlador externo podremos añadir reglas para tratar los flujos de datos que lo atravieses como se le especifiquen.

En el siguiente punto (5.5) vamos a poder observar como a partir de un dispositivo en el que le hemos ido configurando los servicios necesarios, vamos a replicar dicha información para disponer de otros switches virtuales y poder utilizar redes con más elementos.

En el apartado 6 vamos a disponer de una segunda parte de este trabajo que deja a un lado los equipos construidos en líneas generales. En el primer punto (6.1), todavía utilizando las maquetas SDN, intentamos comunicar el controlador SDN con el PCE para que este obtenga información como la topología. A partir del punto 6.2, dejamos completamente a un lado los dispositivos Soekris y empezando por realizar una introducción a un proyecto de investigación de participación internacional, pasamos a realizar una serie de tareas previas a tener en cuenta para realizar una validación de un protocolo para controladores SDN. En el punto 6.4 presentamos dicha demostración y los pasos que hemos seguido para realizarla.

En el último apartado con contenido técnico (6.6) escribimos acerca de que en una colaboración *multiparner* internacional hemos presentado un *paper* al OFC 2016 el cual ha sido admitido. En este apartado también comentamos los resultados correspondientes.

4.2. SDN

SDN (*Software-defined Network*) es un concepto de crear las infraestructuras de la red de una manera modular y con diferentes planos, para desacoplar el control de los dispositivos físicos y dárselo a una parte lógica, un controlador. Esto nos permite separar las funcionalidades y servicios de la red de sus diferentes tecnologías (Ethernet, Microondas, PON…) obteniendo así redes más flexibles que además nos permite realizar, con mayor facilidad, ingeniería de tráfico (control y administración de recursos de la red).

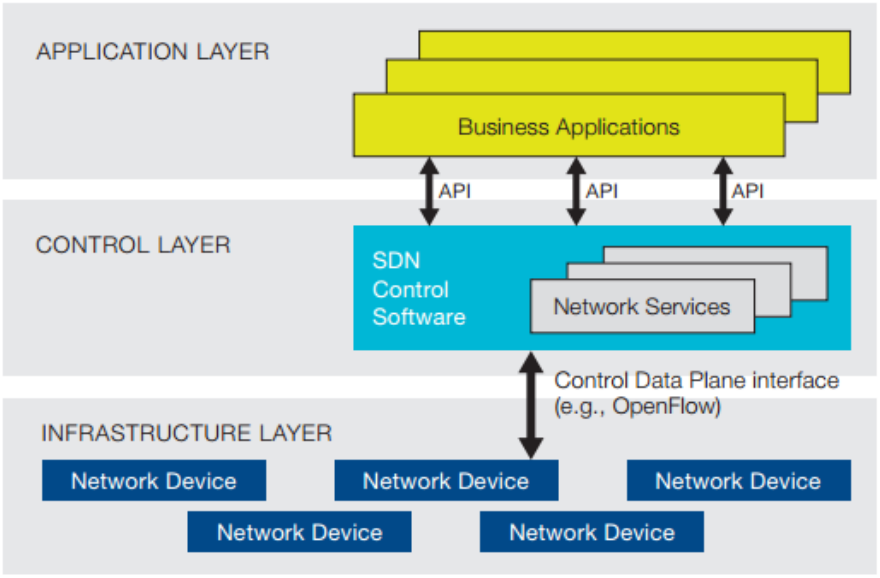


Imagen 1: Arquitectura SDN

[1].

Los dispositivos lógicos encargados de gestionar la red son los denominados controladores. Con dicho dispositivos conseguimos una separación del plano de control con el plano de datos. Los controladores, como se pueden observar en la imagen anterior (Imagen 1) tienen dos interfaces, la API *Application Programming Interface* o también conocida como *Northbound interface* y la *Control data plane interface* o *Southbound interface*. La interfaz superior (*Northbound interface*) es la encargada de recibir las órdenes de la capa de aplicación. Aunque existen protocolos privativos para realizar esta comunicación como el OpFlex – [último acceso: 9 de Diciembre de 2015]

[2] propiedad de Cisco Systems, Inc. existe un uso muy extendido en la utilización del protocolo OpenFlow del que hablaremos en el siguiente punto.

La interfaz inferior (*Southbound interface*) facilita el control en la red, permitiendo al controlador realizar cambios dinámicos de acuerdo a las demandas en tiempo real y las necesidades. Estos cambios se realizan sobre la tecnología específica de la red.

4.3. OPENFLOW

Como hemos mencionado en la introducción sobre SDN, OpenFlow es un estándar utilizado para las comunicaciones del *northbound interface* entre el plano de control y el plano de aplicación. Este protocolo fue el primer estándar definido para este tipo de comunicaciones. Este protocolo está bajo el desarrollo de la *Open Networking Foundation* (ONF) (Universidad de Stanford antes de la creación de esta nueva institución), que según en su documentación web [3], comenta que, OpenFlow permite un acceso directo hacia y la manipulación del plano de control y del encaminamiento de los distintos dispositivos de red como pueden ser los switches y routers, tanto como físicos o virtuales. Tecnologías SDN basadas en OpenFlow, permiten direccionar el ancho de banda, crear flujos naturales de aplicaciones de uso diario, adaptar la red a cada cambio que se necesite y reduce significativamente el tiempo, las operaciones y la complejidad de gestión.

Uno de los elementos de red más importantes en este ámbito es el switch OpenFlow, que no es nada más de una integración de un switch Ethernet de uso común con la implementación de este protocolo. En este elemento se pueden diferenciar 3 partes:

1. Controlador, es el dispositivo (en el caso de un controlador externo) o la parte del dispositivo que se encarga de recibir la información procedente del usuario de una aplicación superior y modificar la tabla de flujos.
2. Canal, es el medio por el que se comunican el switch y el controlador. Este medio debe ser un canal seguro para no perder el control del elemento. En el caso de este trabajo esta comunicación se hacía a través una comunicación TCP.
3. Tabla de flujos, es una lista de acciones que debe aplicar el switch a los flujos de datos que lo atraviesen en función de parámetros de entrada como pueden ser interfaces de entrada, salida, dirección destino, protocolo encapsulado, etc.

4.4. ARTUITECTURA PCE

La arquitectura basada en el *Path Computation Element* (PCE) [4] es un sistema de componentes, aplicaciones o nodos de red capaces de determinar y encontrar un camino adecuado para el transporte de datos entre dos extremos (*endpoints*). Concretamente un PCE es un elemento de la red que es capaz de calcular y encontrar un camino en una malla de red. El cálculo de esta ruta se denomina *Path Computation*. Otro tipo de elemento en este tipo de arquitecturas es el PCC (*Path Computation Client*) quien es todo aquella aplicación cliente que realice una petición *path computation* que pueda entender el PCE.

Todas las comunicaciones entre los elementos de una arquitectura basada en PCE se realizan a través del protocolo PCEP [5] (*Path Computation Element Protocol*).

En Marzo de 2015 en el rfc791 [6] se definió el uso de este tipo de arquitecturas para aplicaciones basadas en operaciones de red, lo que dio lugar a un nuevo elemento llamado ABNO (*Application-Based Network Operations*). Este elemento proporciona una optimización entre de los flujos de tráfico de aplicaciones, un control remoto de los componentes y recursos de la red a través de las técnicas de encaminamiento, separación de elementos de control, OpenFlow o a través de un plano de control coordinado mediante el protocolo PCEP. Con ABNO también podemos coordinar para obtener una mejor interconexión entre redes de entregas de contenido y las redes de distribución a través del establecimiento y posibles ajustes posteriores. Además de coordinar los recursos de red para automatizar el provisionamiento de las rutas y facilitar la programabilidad del ancho de banda y otros recursos, es posible planificar redes privadas virtuales (VPN).

En la siguiente imagen (Imagen 2) podemos observar cómo es la arquitectura ABNO basada en PCE.

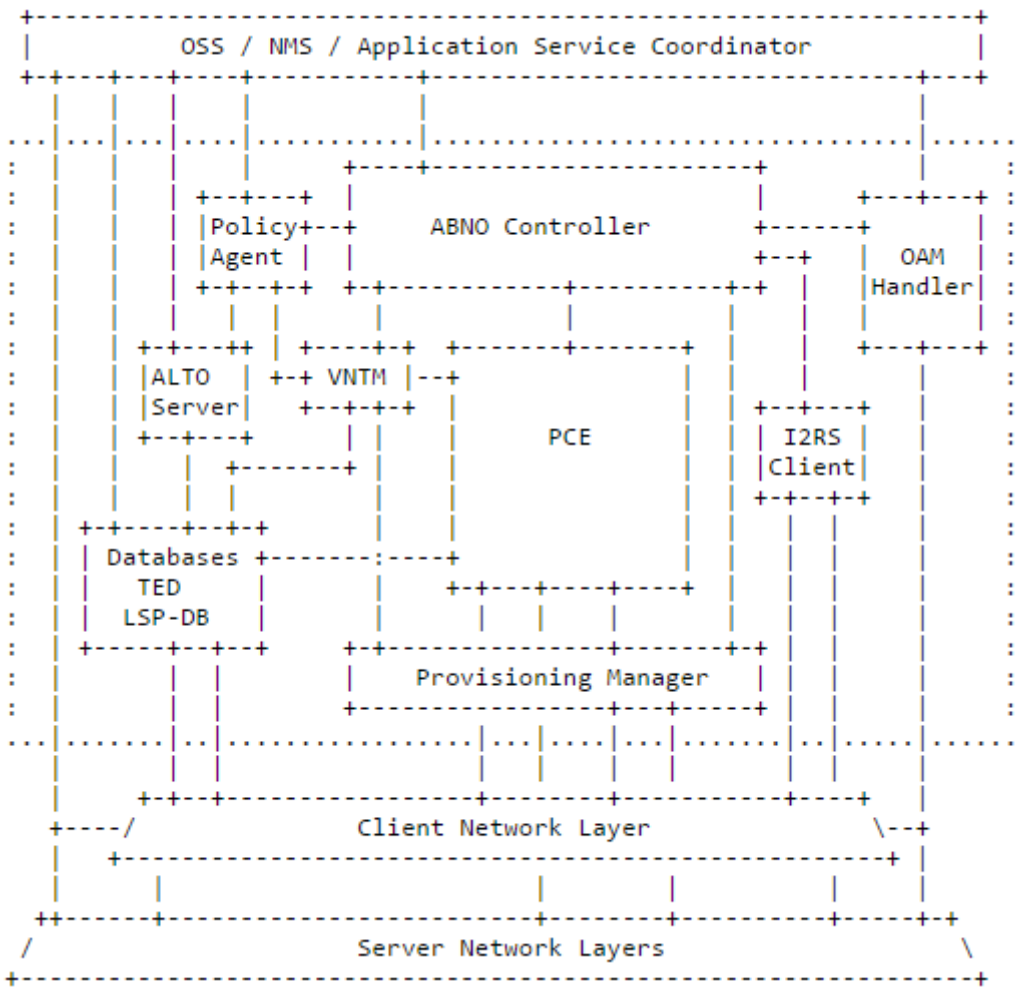


Imagen 2: Arquitectura ABNO.

5. CONSTRUCCIÓN DE UN EQUIPO DE COMUNICACIONES SDN

En esta primera parte vamos a comentar, de una manera organizada, la búsqueda de información de los dispositivos involucrados, las pruebas realizadas y como convertiremos unos dispositivos en un equipo de comunicaciones SDN.

5.1. EQUIPOS DE TRABAJO

- Equipo de trabajo (NERUDA).
  - Procesador:
    - AMD PHENOM™ II X6 955.
    - http://www.amd.com/es-es/products/processors/desktop/phenom-ii
  - RAM:
    - 4GB.
  - Almacenamiento:
    - Seagate ST3500418AS SATA II 500GB
  - Sistema operativo:
    - Linux Mint 17 Cinnamon 64bits.
  - Interfaces de red
    - RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller (rev03)
    - eth0: 48:5b:39:a3:da:c0
    - RTL8169 PCI Express Gigabit Ethernet Controller (rev10)
    - eth1: 00:80:5a:4a:90:cc

- Equipo de trabajo secundario (DICKENS).
  - Procesador:
    - Intel® Core™2 Duo Processor E6550.
    - http://ark.intel.com/es-es/products/30783/Intel-Core2-Duo-Processor-E6550-4M-Cache-2_33-GHz-1333-MHz-FSB
  - RAM:
    - 2GB.
  - Almacenamiento:
    - Maxtor STM325031 SATA II 250GB
  - Sistema operativo:
    - Linux Mint 17 Cinnamon 64bits.
  - Interfaces de red
    - NVIDIA MCP51 Ethernet Controller
    - eth0: 00:1d:60:0c:03:eb

- Soekris (Soekris1)
  - Procesador:
    - Intel® Atom™ Processor E640.
    - http://ark.intel.com/es-es/products/52493/Intel-Atom-Processor-E640-512K-Cache-1_00-GHz
  - RAM:
    - 1GB.
  - Almacenamiento:
    - Memoria USB 4GB.
  - Sistema operativo:
    - Ubuntu Server 14.04.01 LTS 32bits.
  - Interfaces de red:
    - Intel 82574L Gigabit Network Connection.
    - eth0: 00:00:24:d0:3a:90
    - eth1: 00:00:24:d0:3a:91
    - eth2: 00:00:24:d0:3a:92
    - eth3: 00:00:24:d0:3a:93

Neruda y Dickens los vamos a utilizar en casi todo momento como equipos auxiliares para centrarnos en analizar los dispositivos Soekris. Por eso vamos a comentar unos primeros aspectos de estos equipos.

El entorno de pruebas se ha construido con equipos **Soekris net6501**, como el que se puede observar en la siguiente imagen.

Imagen 3: Soekris net6501

A través del conector serie RS-232 del equipo y con la ayuda de la aplicación Minicom instalada en otro equipo auxiliar es posible comunicarse con el equipo, y de esta forma se va a realizar una instalación de un sistema operativo. El sistema operativo elegido es Ubuntu Server 14.04.1 LTS, dado que bajo este sistema operativo hay una gran comunidad que aporta tanto desarrollo como documentación específica, que nos va a ser útil para solucionar problemas, que nos podamos encontrar, de las herramientas que iremos a utilizar.

5.2. TEST DE RENDIMIENTO

En esta sección vamos a realizar una serie de pruebas del el procesador y la escritura en disco para ver realmente cómo se comportan los dispositivos respecto a las características teóricas de los mismos.

5.2.1. TEST DEL PROCESADOR

En primer lugar buscamos una pequeña comparativa entre los procesadores de los equipos Neruda (AMD PHENOM™ II X6 955) y Soekris (Intel® Atom™ Processor E640), para ver qué podemos esperar de ellos cuando se realice el test experimental.

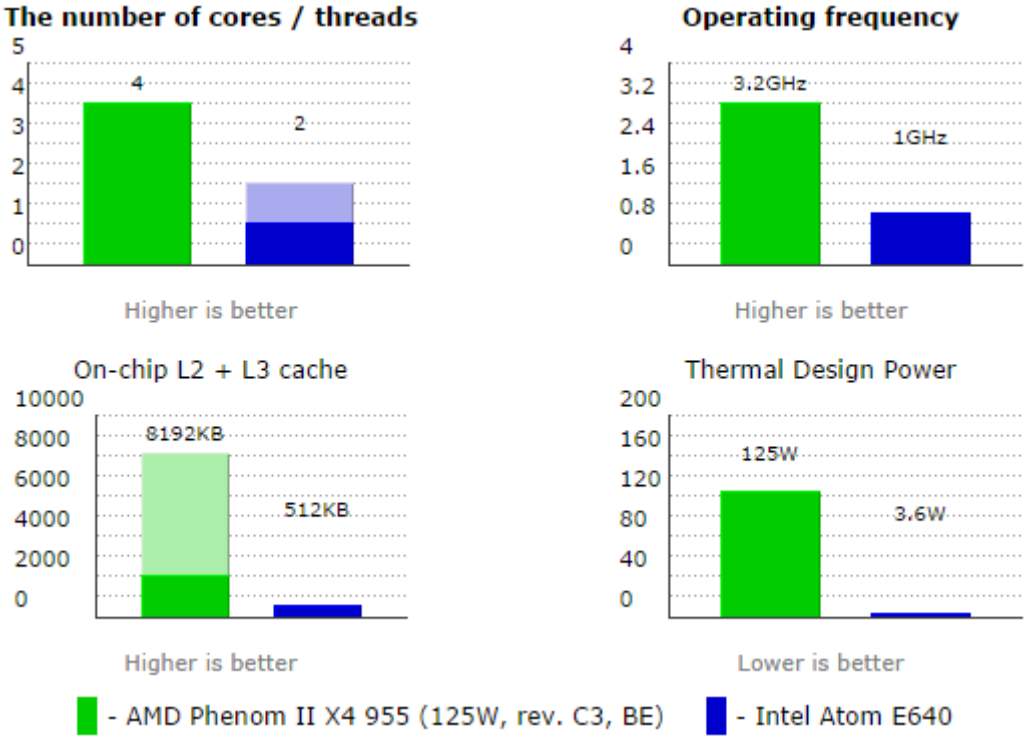


Imagen 4: Principal comparativa gráfica entre los procesadores [7].

En verde tenemos el procesador de Neruda y en azul el de Soekris. En la primera gráfica en color más claro muestra el rendimiento extra que aporta la tecnología Hyper-Threading. (En la gráfica de abajo muestra con color oscuro o más claro, la comparación con L2 o L3 caché respectivamente).

En un primer lugar antes de hacer unas pruebas reales en los equipos, observamos las gráficas y vemos que en relación a la frecuencia de reloj de ambos microprocesadores, podemos esperar una relación aproximada de 3 a 1 en cuanto a rendimiento.

Creamos un script básico, que realice una serie de multiplicaciones, para comprobar el tiempo que tarda en terminar dichas ejecuciones.

rendimiento_cpu.sh

```
#!/bin/bash
inicio_ns=`date +%s%N`
inicio=`date +%s`
for i in {1..10000}
  do
     sol=`expr $i \* $i`
 done
echo $sol
fin_ns=`date +%s%N`
fin=`date +%s`
let total_ns=$fin_ns-$inicio_ns
let total=$fin-$inicio
echo "ha tardado: -$total_ns- nanosegudos, -$total- segundos"
```

Se realizan 3 pruebas en cada equipo (Neruda y Soekris) con un número de iteraciones de 3000, 5000, 10000.

Iteraciones	**Neruda**	**Soekris**
3000	4s	12s
5000	7s	20s
10000	14s	40s

Tabla 1: Tiempo ejecución script test.

Se puede concluir que el test de rendimiento del procesador es el esperado (aproximadamente una relación 3 a 1).

5.2.2. TEST DE ESCRITURA EN DISCO

Se realiza una serie de pruebas con los equipos Neruda y Soekris en distintas unidades de almacenamiento.

Con la siguiente línea de comando, medimos las velocidades de escritura: *dd if=/dev/zero of=prueba count=100 bs=512k*

Variamos el valor del parámetro *count* para realizar varios tests con distintos tamaños de fichero.

Fichero			Neruda						Soekris	
count	bs	Tamaño	Disco Duro SATA		Memoria USB 1		Memoria USB 2		Memoria USB 1	
100	512	52MB	0.05s	960MB/s	15.56s	3.4MB/s	1.13s	46.5MB/s	0.368s	142MB/s
800	512	419MB	3.38s	124MB/s	121.4s	3.5MB/s	11.2s	37.3MB/s	124s	3.4MB/s

Tabla 2: Comparativa escritura en disco.

La memoria USB 1 es la unidad de almacenamiento primaria utilizada en el equipo Soekris (memoria USB genérica con sistema de ficheros EXT4).

Memoria USB 2: Transcend Jet Flash 700 USB 3.0 con sistema de ficheros FAT32 (utilizado en ambos equipos en puerto USB 2.0).

5.3. TEST DE RED

Diseñamos una red inicial para configurar los equipos y realizar un test de medida de los enlaces entre los equipos.

En la siguiente imagen se puede observar cómo es la topología y su configuración.

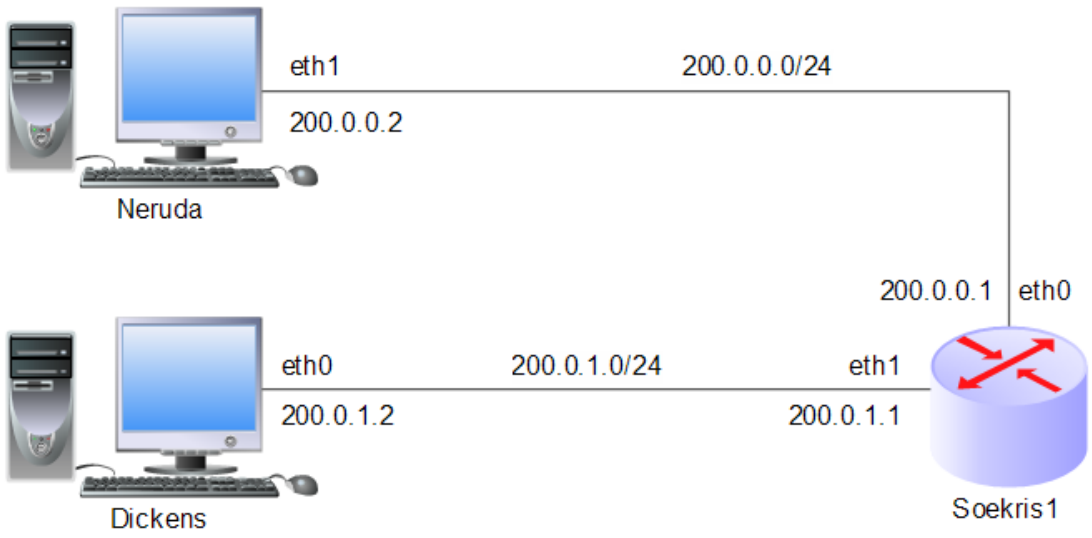


Imagen 5: *Topología de red.*

Con el escenario planteado, se realiza un script para configurar los equipos.

`config_red.sh`

```
#! /bin/bash
# ejecutar como root. sudo.
maquina=`uname -n`
if [ "$maquina" == "neruda" ]; then
      service network-manager stop
      ifconfig eth0 down
      ifconfig eth1 200.0.0.2 up
      route del default
      route add default gw 200.0.0.1
      echo "Se ha configurado la red en la maquina neruda"

elif [ "$maquina" == "dickens" ]; then
      service network-manager stop
      ifconfig eth0 200.0.1.2 up
```

```
      route del default
      route add default gw 200.0.1.1
      echo "Se ha configurado la red en la maquina dickens"

elif [ "$maquina" == "soekris1" ]; then
      bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'
      ifconfig eth0 200.0.0.1 up
      ifconfig eth1 200.0.1.1 up
      echo "Se ha configurado la red en la maquina soekris1"

else
      echo "Error al configurar la maquina $maquina"
fi
```

5.3.1. IPERF

Iperf [8], es una utilidad para medir el ancho de banda disponible en una red IP. Esta herramienta soporta varios protocolos para realizar las medidas. Para este trabajo vamos a utilizar el protocolo TCP y UDP.

Una vez configurados los 3 equipos que vamos a utilizar (Neruda, Dickens y Soekris1), y habiendo probado con la utilidad *ping* que el encaminamiento funciona correctamente, pasamos a instalar la utilidad Iperf para medir la capacidad de la red. A continuación vamos a realizar una serie de pruebas para familiarizarnos con esta utilidad e ir obteniendo datos.

a) En primer lugar hacemos un test entre Neruda y Soekris1:

   En el equipo Soekris1 lanzamos la utilidad en modo servidor.

```
jgrajos@soekris1:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 200.0.0.1 port 5001 connected with 200.0.0.2 port 60823
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec   619 MBytes   519 Mbits/sec
[  5] local 200.0.0.1 port 5001 connected with 200.0.0.2 port 60824
[  5]  0.0-60.0 sec  3.62 GBytes   518 Mbits/sec

CTRL-A Z for help | 19200 8N1 | NOR | Minicom 2.7 | VT102 | Conectado
```

Imagen 6*: Soekris en modo servidor.*

   Y a continuación en Neruda lanzamos el test en modo cliente (por defecto transmitirá tráfico TCP durante 10s). Lanzamos 2 pruebas, una con duración de

10s (por defecto) y otra de 60s, para comprobar que el ancho de banda se mantiene estable.

```
jgrajos@neruda ~ $ iperf -c 200.0.0.1
------------------------------------------------------------
Client connecting to 200.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 200.0.0.2 port 60823 connected with 200.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   619 MBytes   519 Mbits/sec
jgrajos@neruda ~ $ iperf -t 60 -c 200.0.0.1
------------------------------------------------------------
Client connecting to 200.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 200.0.0.2 port 60824 connected with 200.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-60.0 sec  3.62 GBytes   518 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 7 *Neruda en modo cliente.*

Como se puede observar en ambas capturas de pantalla, para los 2 tests aproximadamente nos ofrece el mismo resultado, y decimos que el ancho de banda del enlace es de 518Mb/s.

b) En segundo lugar realizamos el mismo test pero entre Soekris1 y Dickens.

```
jgrajos@dickens ~ $ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 200.0.1.2 port 5001 connected with 200.0.1.1 port 53165
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  1.10 GBytes   941 Mbits/sec
[  5] local 200.0.1.2 port 5001 connected with 200.0.1.1 port 53166
[  5]  0.0-60.0 sec  6.57 GBytes   941 Mbits/sec
^Cjgrajos@dickens ~ $
```

Imagen 8*: Dickens en modo servidor.*

```
jgrajos@soekris1:~$ iperf -c 200.0.1.2
------------------------------------------------------------
Client connecting to 200.0.1.2, TCP port 5001
TCP window size: 43.8 KByte (default)
------------------------------------------------------------
[  3] local 200.0.1.1 port 53165 connected with 200.0.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.10 GBytes   944 Mbits/sec
jgrajos@soekris1:~$ iperf -t 60 -c 200.0.1.2
------------------------------------------------------------
Client connecting to 200.0.1.2, TCP port 5001
TCP window size: 43.8 KByte (default)
------------------------------------------------------------
[  3] local 200.0.1.1 port 53166 connected with 200.0.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-60.0 sec  6.57 GBytes   941 Mbits/sec
jgrajos@soekris1:~$
CTRL-A Z for help | 19200 8N1 | NOR | Minicom 2.7 | VT102 | Conectado
```

Imagen 9: *Soekris en modo cliente.*

Se observa que en este enlace el ancho de banda es de 941Mb/s.

c) Por último volvemos a realizar el mismo test entre Neruda y Dickens para comprobar si se produce pérdida de capacidad en el encaminamiento. En este caso lanzamos Iperf con la opción *–i 1* para que nos muestre la información, durante el proceso de transmisión de datos, cada 1s.

```
jgrajos@dickens ~ $ iperf -i 1 -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 200.0.1.2 port 5001 connected with 200.0.0.2 port 37324
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 1.0 sec  61.2 MBytes   514 Mbits/sec
[  4]  1.0- 2.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  2.0- 3.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  3.0- 4.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  4.0- 5.0 sec  61.3 MBytes   515 Mbits/sec
[  4]  5.0- 6.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  6.0- 7.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  7.0- 8.0 sec  61.2 MBytes   514 Mbits/sec
[  4]  8.0- 9.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  9.0-10.0 sec  61.3 MBytes   514 Mbits/sec
[  4]  0.0-10.0 sec   613 MBytes   514 Mbits/sec
^Cjgrajos@dickens ~ $
```

Imagen 10: Dickens servidor – Neruda cliente.

```
jgrajos@neruda ~ $ iperf -i 1 -c 200.0.1.2
------------------------------------------------------------
Client connecting to 200.0.1.2, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 200.0.0.2 port 37324 connected with 200.0.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  61.8 MBytes   518 Mbits/sec
[  3]  1.0- 2.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  2.0- 3.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  3.0- 4.0 sec  61.4 MBytes   515 Mbits/sec
[  3]  4.0- 5.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  5.0- 6.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  6.0- 7.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  7.0- 8.0 sec  61.4 MBytes   515 Mbits/sec
[  3]  8.0- 9.0 sec  61.2 MBytes   514 Mbits/sec
[  3]  0.0-10.0 sec   613 MBytes   514 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 11: Neruda cliente – Dickens servidor.

En este caso observamos que entre Neruda y Dickens obtenemos un ancho de banda de 514Mb/s, ligeramente inferior al menor ancho de banda entre los dos enlaces por separados (518Mb/s). En este caso no percibiríamos una pérdida substancial del ancho de banda por el encaminamiento.

d) Como observamos que el enlace *Neruda – Soekris* a través de la interfaz de red eth1, nos ofrece un rendimiento bastante inferior al esperado al ser una interfaz de red Gigabit, probamos a realizar el test anterior pero utilizando en Neruda la interfaz eth0.

- Test entre Neruda y Soekris1:

```
jgrajos@neruda ~ $ iperf -c 200.0.0.1
------------------------------------------------------------
Client connecting to 200.0.0.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 200.0.0.2 port 54628 connected with 200.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.10 GBytes   942 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 12: Neruda cliente (eth0)- Soekris servidor.

Ahora si observamos que funciona a una velocidad mucho mejor, al usar la interfaz eth0 (aprox. a la esperada) que la utilizada anteriormente (eth1).

- Test entre Neruda y Dickens:

```
jgrajos@neruda ~ $ iperf -c 200.0.1.2
------------------------------------------------------------
Client connecting to 200.0.1.2, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 200.0.0.2 port 43277 connected with 200.0.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   922 MBytes   774 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 13: Neruda cliente (eth0)- Dickens servidor.

Ahora comprobamos que cada enlace por separado nos ofrece un ancho de banda de 940Mb/s y que obtenemos un ancho de banda entre extremos de 774Mb/s, cuya pérdida de rendimiento es debida al procesar la información a la hora de encaminar los paquetes en Soekris.

5.3.2. TCP

En estos siguientes apartados vamos a realizar unas pruebas utilizando el protocolo TCP. En contra de las pruebas realizadas en el apartado anterior, que era para tener una idea general del funcionamiento de la red montada, ahora vamos a realizar unas pruebas más específicas, ya que vamos a realizarlas en las combinaciones de todas las interfaces y además incluiremos otra sección utilizando el protocolo UDP, que nos va a permitir modificar el tamaño del datagrama, para ver como varía el ancho de banda de la red en función de este.

5.3.2.1. CONEXIÓN DIRECTA

En primer lugar se realiza un test para medir el ancho de banda a través de una conexión directa (sin nodos intermedios) entre Neruda y Dickens. Utilizamos las interfaces de red y direcciones IP tal cual se observan en la siguiente representación esquemática de la red.

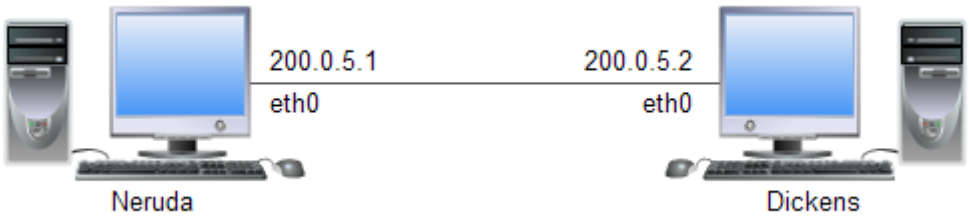


Imagen 14: conexión directa; Neruda (eth0)- Dickens (eth0).

Con dicha configuración de red obtenemos el siguiente test:

```
jgrajos@neruda ~ $ iperf -c 200.0.5.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.5.2, TCP port 5001
TCP window size:  170 KByte (default)
------------------------------------------------------------
[  5] local 200.0.5.1 port 53669 connected with 200.0.5.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  1.10 GBytes   942 Mbits/sec
[  4] local 200.0.5.1 port 5001 connected with 200.0.5.2 port 51565
[  4]  0.0-10.0 sec  1.10 GBytes   941 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 15: Test Iperf; Neruda (eth0)- Dickens (eth0).

Observamos que obtenemos un buen ancho de banda.

Ahora cambiamos en Neruda la interfaz eth0 por la eth1. Mostramos el cambio en el siguiente esquema de red.

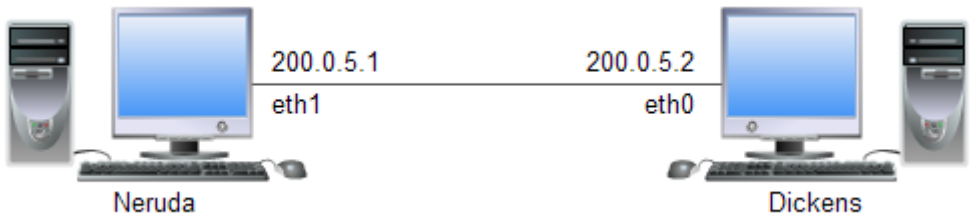


Imagen 16: conexión directa; Neruda (eth1)- Dickens (eth0).

El análisis del ancho de banda se observa en la siguiente captura:

```
jgrajos@neruda ~ $ iperf -c 200.0.5.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.5.2, TCP port 5001
TCP window size:  442 KByte (default)
------------------------------------------------------------
[  5] local 200.0.5.1 port 53675 connected with 200.0.5.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   610 MBytes   511 Mbits/sec
[  4] local 200.0.5.1 port 5001 connected with 200.0.5.2 port 51570
[  4]  0.0-10.0 sec   789 MBytes   662 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 17: Test Iperf; Neruda (eth1)- Dickens (eth0).

Al usar en Neruda la interfaz eth1 observamos que el ancho de banda obtenido en el test es significativamente inferior (de 900Mb/s a 500Mb/s), por lo que a partir de ahora utilizaremos la interfaz eth0 de Neruda para las tareas posteriores.

5.3.2.2. CONEXIÓN A TRAVÉS DE UN EQUIPO SOEKRIS

Con el enlace directo analizado, pasamos a incorporar el equipo Soekris1 para que actúe como enrutador y comprobaremos si existe pérdida de ancho de banda al encaminar.

Configuramos el equipo Soekris1 con las siguientes direcciones de red en cada interfaz para que cada vez que se hable de otro equipo, dada su dirección de red, se asuma en que interfaz de red de Soekris1 está conectado.

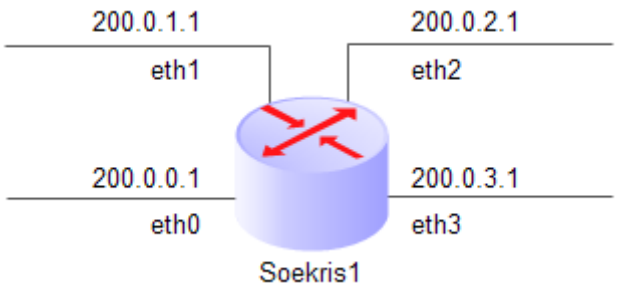


Imagen 18: Soekris1 – Interfaces de red.

En la siguiente imagen se puede ver la representación de la red al incluir el equipo Soekris1 en ella. Las siguientes modificaciones de red de este informe sólo cambiarán en que puertos se conectan los equipos al router (Soekris1) y por consiguiente las direcciones IP de los equipos (Neruda y Dickens).

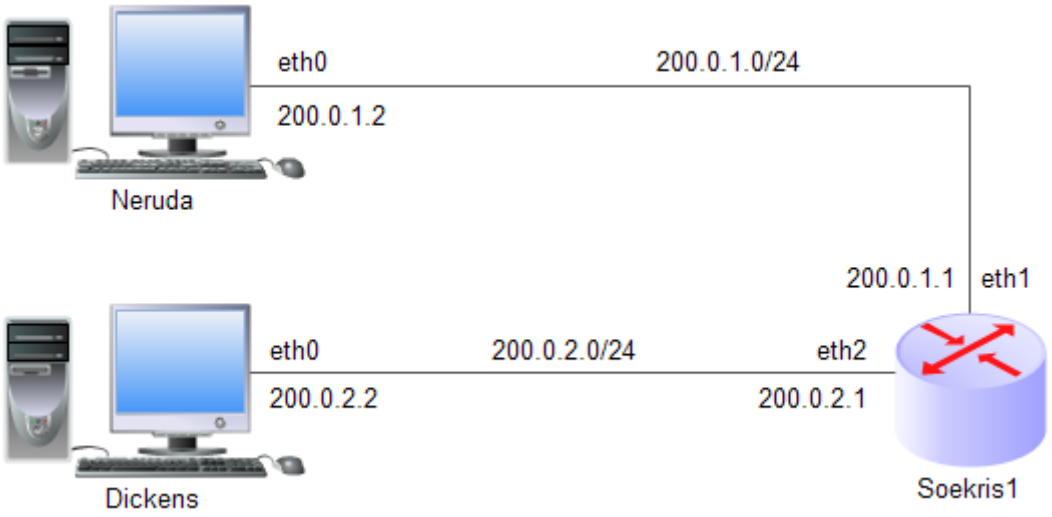


Imagen 19: Topología de red tests.

Teniendo definida la red en la que se van a realizar las pruebas para determinar el ancho de banda posible de extremo a extremo, se muestran a continuación las capturas de los test realizados para algunas combinaciones de pares de interfaces utilizadas (en Soekris). Y al final de estas capturas se incorporará una tabla para una visualización más representativa.

1) Neruda 200.0.0.2 y Dickens 200.0.1.2

```
jgrajos@neruda ~ $ iperf -c 200.0.1.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.1.2, TCP port 5001
TCP window size:  314 KByte (default)
------------------------------------------------------------
[  5] local 200.0.0.2 port 51487 connected with 200.0.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   922 MBytes   773 Mbits/sec
[  4] local 200.0.0.2 port 5001 connected with 200.0.1.2 port 44472
[  4]  0.0-10.0 sec   910 MBytes   762 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 20: Test Iperf; Neruda 200.0.0.2 - Dickens 200.0.1.2.

2) Neruda 200.0.1.2 y Dickens 200.0.0.2

```
jgrajos@neruda ~ $ iperf -c 200.0.0.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.0.2, TCP port 5001
TCP window size:  484 KByte (default)
------------------------------------------------------------
[  5] local 200.0.1.2 port 59023 connected with 200.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   908 MBytes   761 Mbits/sec
[  4] local 200.0.1.2 port 5001 connected with 200.0.0.2 port 43941
[  4]  0.0-10.0 sec   900 MBytes   755 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 21: Test Iperf; Neruda 200.0.1.2 - Dickens 200.0.0.2.

3) Neruda 200.0.0.2 y Dickens 200.0.2.2

```
jgrajos@neruda ~ $ iperf -c 200.0.2.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.2.2, TCP port 5001
TCP window size:  332 KByte (default)
------------------------------------------------------------
[  5] local 200.0.0.2 port 52526 connected with 200.0.2.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  1.05 GBytes   906 Mbits/sec
[  4] local 200.0.0.2 port 5001 connected with 200.0.2.2 port 49801
[  4]  0.0-10.0 sec  1.08 GBytes   925 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 22: Neruda 200.0.0.2 y Dickens 200.0.2.2

4) Neruda 200.0.1.2 y Dickens 200.0.2.2

```
jgrajos@neruda ~ $ iperf -c 200.0.2.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.2.2, TCP port 5001
TCP window size:  280 KByte (default)
------------------------------------------------------------
[  5] local 200.0.1.2 port 48381 connected with 200.0.2.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  1.08 GBytes   929 Mbits/sec
[  4] local 200.0.1.2 port 5001 connected with 200.0.2.2 port 37769
[  4]  0.0-10.0 sec  1.09 GBytes   934 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 23: Neruda 200.0.1.2 y Dickens 200.0.2.2

5) Neruda 200.0.2.2 y Dickens 200.0.3.2

```
jgrajos@neruda ~ $ iperf -c 200.0.3.2 -r
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.3.2, TCP port 5001
TCP window size:  272 KByte (default)
------------------------------------------------------------
[  5] local 200.0.2.2 port 54737 connected with 200.0.3.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   923 MBytes   774 Mbits/sec
[  4] local 200.0.2.2 port 5001 connected with 200.0.3.2 port 55832
[  4]  0.0-10.0 sec   906 MBytes   759 Mbits/sec
jgrajos@neruda ~ $
```

Imagen 24: Neruda 200.0.2.2 y Dickens 200.0.3.2

Se incluyen los valores obtenidos en una tabla para una mejor visualización.

En cada combinación de interfaces utilizada, en la primera interfaz se ha conectado el cliente y en la segunda el servidor. En este caso Neruda como cliente y Dickens como servidor.

El primer valor del ancho de banda en cada fila corresponde al test en sentido cliente -> servidor y el segundo valor en sentido inverso.

eth0	eth1	eth2	eth3
773Mb/s	762Mb/s		
906Mb/s		925Mb/s	
920Mb/s			934Mb/s
	929Mb/s	934Mb/s	
	924Mb/s		925Mb/s
		774Mb/s	759Mb/s

Tabla 3: Tabla Ancho de banda en interfaces.

Al analizar la tabla se observa que se pueden diferenciar los resultados en dos grupos, los que funcionan a un valor muy próximo al ideal (marcados en verde) y los que no (en rojo).

5.3.3. UDP

En este apartado se van a realizar diferentes pruebas modificando el tamaño de los datagramas, para observar si varía o no la capacidad de la red en función de ello.

5.3.3.1. CONEXIÓN DIRECTA

Partiendo del tamaño máximo de datagrama (1470 bytes) que podemos enviar sin que se fragmenten los paquetes, se harán más simulaciones disminuyendo el tamaño del datagrama hasta un mínimo de 12bytes (mínimo soportado por Iperf).

```
jgrajos@neruda ~ $ iperf -c 200.0.5.3 -u -r -b 1000m -l 1470
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.5.3, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 200.0.5.2 port 52262 connected with 200.0.5.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   968 MBytes   812 Mbits/sec
[  5] Sent 690326 datagrams
[  5] Server Report:
[  5]  0.0-10.0 sec   967 MBytes   811 Mbits/sec   0.085 ms  315/690325 (0.046%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
[  3] local 200.0.5.2 port 5001 connected with 200.0.5.3 port 60340
[  3]  0.0-10.0 sec   960 MBytes   806 Mbits/sec   0.019 ms 6549/691613 (0.95%)
[  3]  0.0-10.0 sec  1 datagrams received out-of-order
jgrajos@neruda ~ $
```

Imagen 25: Iperf, conexión directa, test UDP, 1470bytes/datagrama.

```
jgrajos@neruda ~ $ iperf -c 200.0.5.3 -u -r -b 1000m -l 1000
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1000 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.5.3, UDP port 5001
Sending 1000 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 200.0.5.2 port 47757 connected with 200.0.5.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   656 MBytes   550 Mbits/sec
[  5] Sent 687836 datagrams
[  5] Server Report:
[  5]  0.0-10.0 sec   656 MBytes   550 Mbits/sec   0.016 ms  387/687835 (0.056%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
[  4] local 200.0.5.2 port 5001 connected with 200.0.5.3 port 42640
[  4]  0.0-10.0 sec   656 MBytes   550 Mbits/sec   0.022 ms 3694/691495 (0.53%)
[  4]  0.0-10.0 sec  1 datagrams received out-of-order
jgrajos@neruda ~ $
```

Imagen 26: Iperf, conexión directa, test UDP, 100bytes/datagrama.

```
jgrajos@neruda ~ $ iperf -c 200.0.5.3 -u -r -b 1000m -l 12
WARNING: option -l has implied compatibility mode
------------------------------------------------------------
Client connecting to 200.0.5.3, UDP port 5001
Sending 12 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 200.0.5.2 port 33602 connected with 200.0.5.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  7.85 MBytes  6.58 Mbits/sec
[  3] Sent 685664 datagrams
jgrajos@neruda ~ $
```

Imagen 27: Iperf, conexión directa, test UDP, 12bytes/datagrama.

En estas pruebas con conexión directa ya observamos que al disminuir el tamaño del datagrama, el número de datagramas enviados son aproximadamente los mismos, por lo que el ancho de banda disminuye.

5.3.3.2. CONEXIÓN CON SOEKRIS

Incorporamos el equipo Soekris a la red y realizamos las pruebas como en el apartado anterior, disminuyendo el tamaño del datagrama para observar como varía el ancho de banda y el número de datagramas enviados.

```
jgrajos@neruda ~ $ iperf -c 200.0.2.2 -u -r -b 1000m -l 1470
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 200.0.2.2, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 200.0.1.2 port 58192 connected with 200.0.2.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   960 MBytes   806 Mbits/sec
[  5] Sent 685030 datagrams
[  5] Server Report:
[  5]  0.0-10.0 sec   960 MBytes   806 Mbits/sec   0.019 ms    0/685029 (0%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
[  4] local 200.0.1.2 port 5001 connected with 200.0.2.2 port 45472
[  4]  0.0-10.0 sec   960 MBytes   805 Mbits/sec   0.016 ms 7109/691674 (1%)
[  4]  0.0-10.0 sec  1 datagrams received out-of-order
jgrajos@neruda ~ $
```

Imagen 28: Iperf, test UDP, 1470bytes/datagrama.

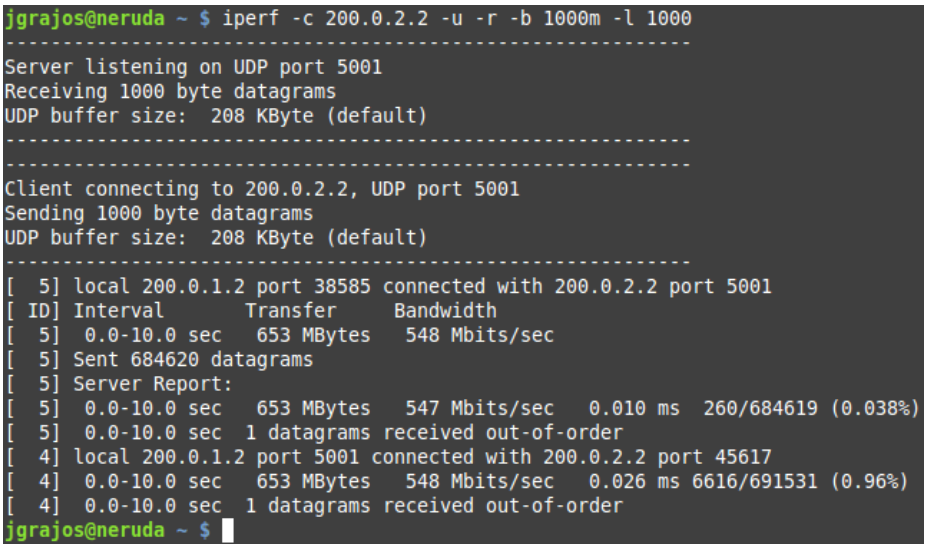


Imagen 29: Iperf, test UDP, 100bytes/datagrama.

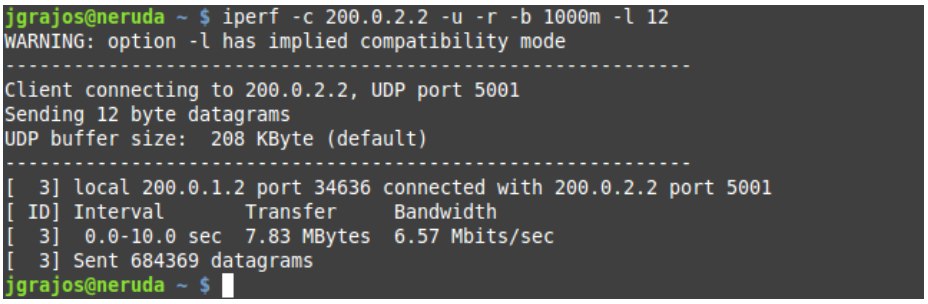


Imagen 30: Iperf, test UDP, 12bytes/datagrama.

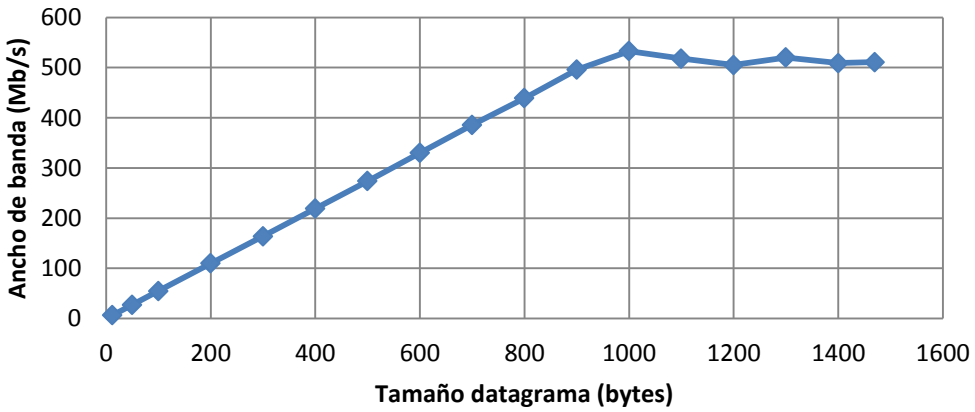
Como se puede ver en las capturas, al igual que en las pruebas con conexión directa, se mantiene constante el número de datagramas enviados y por ello disminuye el ancho de banda.

Para una mejor visualización de los datos, se muestran dos gráficas, para ello se medirán los datagramas/s enviado y el ancho de banda obtenido variando el tamaño del datagrama en UDP, limitando el ancho de banda a 500Mbits/s.

Conectamos Neruda a la interfaz eth1 de Soekris , Dickens a la interfaz eth2 y ejecutamos el comando "`iperf -c 200.0.2.2 -b 500M -l 1470`" variando el tamaño de datagrama desde el tamaño mínimo (12bytes) hasta el tamaño máximo (1470bytes).

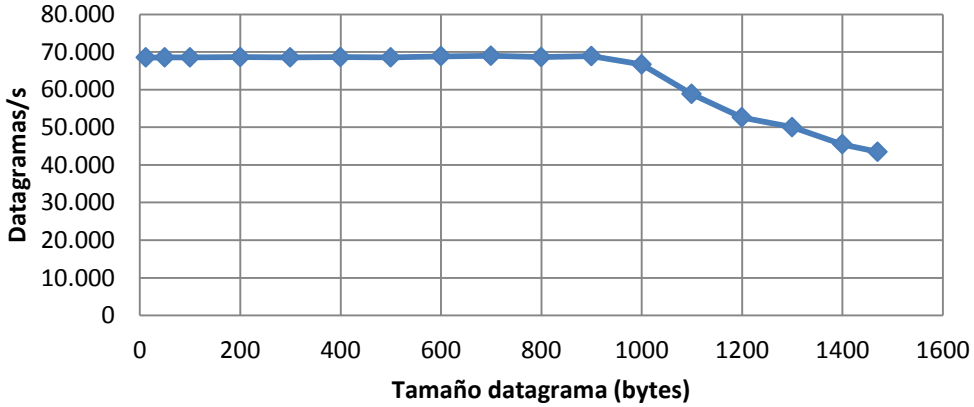
Para una mejor visualización de los datos, se muestran dos gráficas con los datos obtenidos.

En esta primera gráfica comparamos el ancho de banda en función del tamaño del datagrama.



Gráfica 1: Ancho de banda / Tamaño datagrama.

Por último mostramos una gráfica que compara el número de datagramas enviados por segundo, frente al tamaño del datagrama.



Gráfica 2: Número datagramas/s / Tamaño datagrama.

En las gráficas se observa lo esperado. Hasta un ancho de banda aproximado de 500Mbits/s se comporta igual que lo visto en el informe anterior (el ancho de banda aumenta al aumentar los datagramas y los datagramas por segundo transmitidos permanecen constantes) y a partir del ancho de banda que hemos limitado, este permanece constante y los datagramas por segundo enviados disminuyen.

5.3.4. TCP VS UDP

Por último, en la siguiente imagen se observa el tipo de tráfico que transmite Iperf cuando utiliza el protocolo TCP. Así podemos observar el tipo de tráfico que hemos estado analizando.

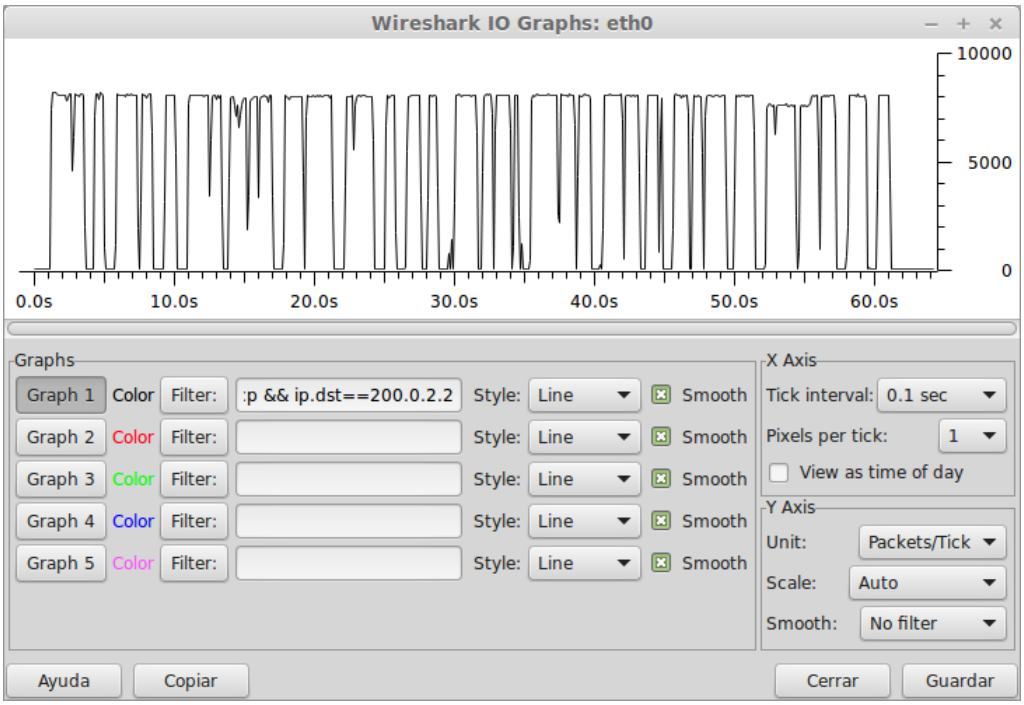


Imagen 31: Tráfico TCP Gráfica Wireshark.

Iperf utiliza un tiempo constante entre el envío de datagramas o paquetes (CBR) según está comentado en la referencia [9]. Aun así se ha comprobado analizando el tráfico generado. En este análisis ya podemos observar que el envío constante de datagramas es aproximadamente 70.000 datagramas/s (en la captura aparece 7000/0.1s).

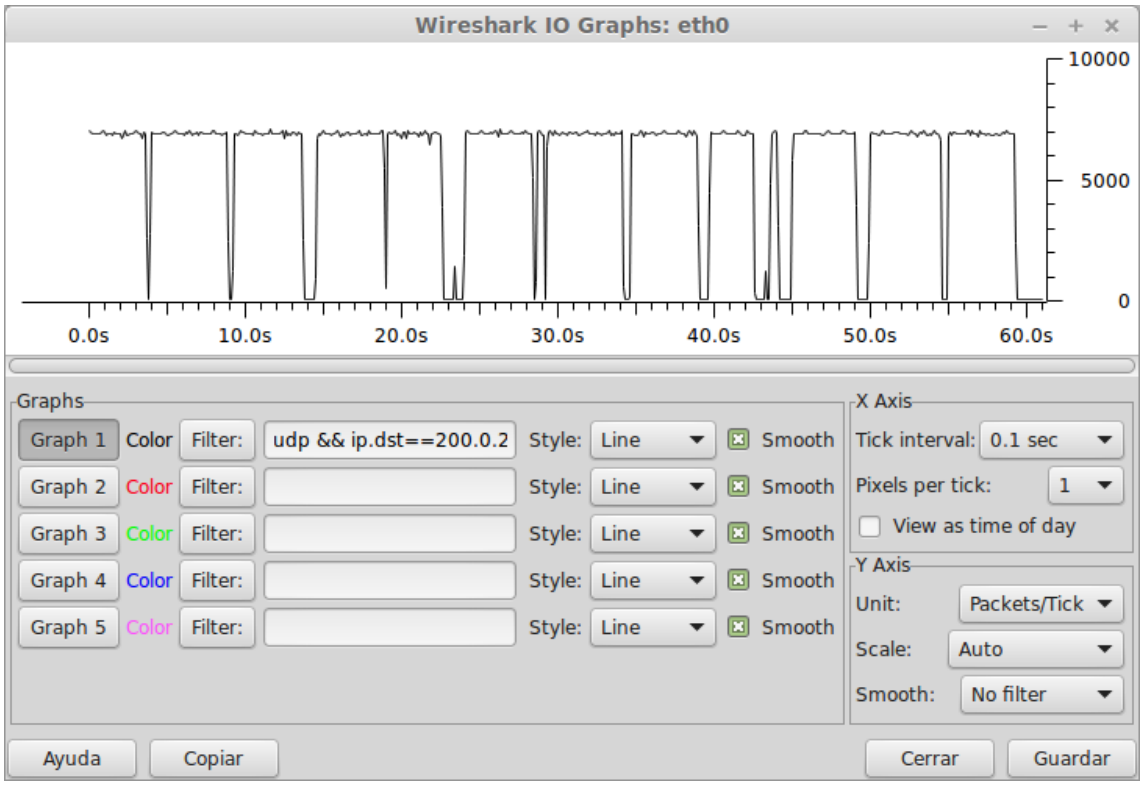


Imagen 32: Tráfico UDP Gráfica Wireshark.

En TCP a diferencia de UDP, comprobamos que obtenemos un tráfico más rafagoso. El número de paquetes transmitidos por segundo, aumenta en TCP con respecto a UDP, en el que obtenemos un valor aproximado de 80.000 paquetes/s (en UDP conseguimos 70.000 datagramas/s).

5.4. SWITCH VIRTUAL

En esta sección vamos a describir los pasos que hemos seguido y las tareas realizadas para configurar en un equipo Soekris un switch virtual a través de la utilidad OpenVSwitch. A continuación, como, mediante un controlador externo podemos gestionar este switch virtual.

5.4.1. OPENVSWITCH

En Soekris1 instalamos openvswitch con la siguente línea de comandos.

```
apt-get install openvswitch-switch
```

A través de los repositorios incluidos en ubuntu server, se ha instalado la versión 2.0.2 de openvswitch. Si necesitásemos una versión más actual podemos descargarla desde la página web oficial (http://openvswitch.org/).

Con la utilidad instalada, creamos un puente y añadimos las interfaces eth1 y eth2.

```
ovs-vsctl add-br puente
ovs-vsctl add-port puente eth1
ovs-vsctl add-port puente eth2
```

Todavía sin controlador externo probamos el funcionamiento añadiendo las siguientes reglas, que harán que todo lo que entre por el puerto 1 (eth1) salga por el puerto 2 (eth2) y viceversa.

```
sudo ovs-ofctl add-flow puente in_port=1,actions=output:2
sudo ovs-ofctl add-flow puente in_port=2,actions=output:1
```

Probamos su funcionamiento con un ping y observamos el tráfico con Wireshark y vemos que funciona según lo esperado.

Ahora cambiamos las reglas anteriores para que todo lo que entre por cada interfaz se descarte.

```
sudo ovs-ofctl add-flow puente in_port=1,actions=drop
sudo ovs-ofctl add-flow puente in_port=2,actions=drop
```

Volvemos a analizar el tráfico al intentar hacer un ping, pero en este caso (según lo especificado) el ping no recibe respuesta, ya que la petición no llega a su destino.

5.4.2. CONTROLADOR EXTERNO

En la siguiente imagen se muestra el diagrama de red, indicando la incorporación del controlador externo situado en Neruda.

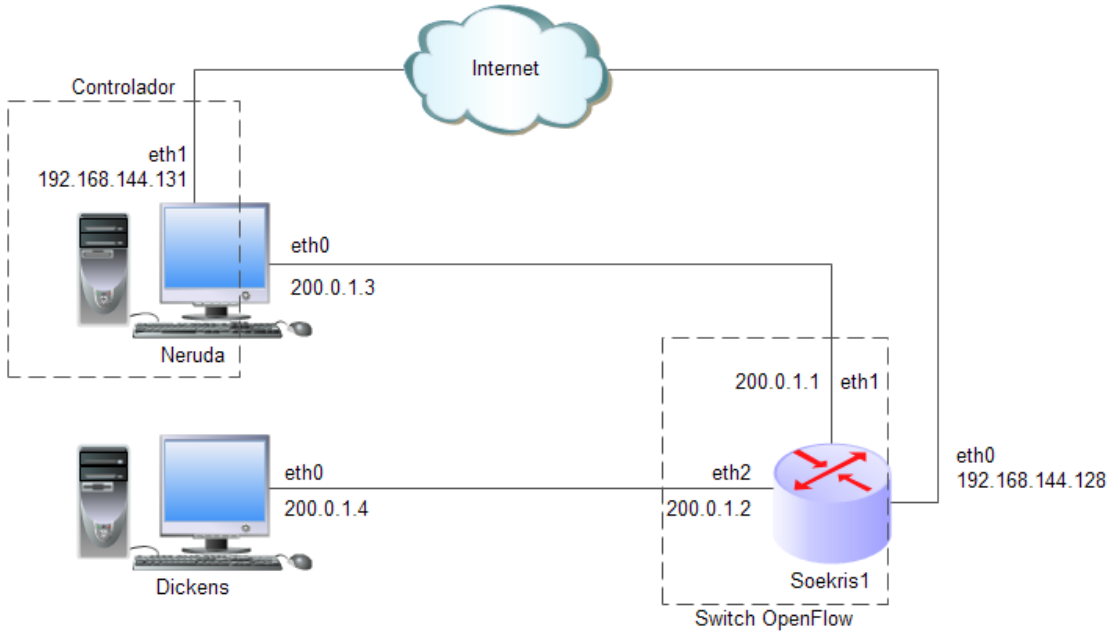


Imagen 33: Diagrama de red con controlador.

En Neruda instalamos el controlador externo Floodlight. En primer lugar instalamos los prerrequisitos que nos indican en la documentación oficial de Floodligth [10].

```
sudo apt-get install build-essential default-jdk ant python-dev
     eclipse
git clone git://github.com/floodlight/floodlight.git
cd floodlight
git checkout stable
ant;
```

Ahora en Soekris1, debemos indicar al openvswitch la ubicación del controlador externo. El puerto utilizado por defecto es el 6633.

```
ovs-vsctl set-controller puente tcp:192.168.144.131:6633
```

Y observamos que la configuración ha tenido efecto con la siguiente línea:

```
ovs-vsctl show
```

```
jgrajos@soekris1:~$ sudo ovs-vsctl show
04cc3013-636f-4f6e-8da7-61b0a9a9ab8b
    Bridge puente
        Controller "tcp:192.168.144.131:6633"
        Port "eth1"
            Interface "eth1"
        Port "eth2"
            Interface "eth2"
        Port puente
            Interface puente
                type: internal
    ovs_version: "2.0.2"
jgrajos@soekris1:~$ _
```

Imagen 34: Configuración Openvswitch.

Volvemos a Neruda y cada vez que queramos iniciar el controlador debemos ejecutar (estando en el directorio *'~/floodlight'* , ya que el código anterior lo hemos ejecutado desde el directorio de usuario.) la siguiente línea de comandos.

```
java -jar target/floodlight.jar
```

```
.168.144.128:58133 DPID[00:00:00:0c:29:f2:bb:77]] bound to class class net.floodlightcontrol
ore.internal.OFSwitchImpl, writeThrottle=false, description Switch Desc - Vendor: Nicira, In
odel: Open vSwitch  Make: None  Version: 2.0.2  S/N: None
20:03:55.018 INFO [n.f.c.OFSwitchBase:New I/O server worker #2-1] Clearing all flows on swit
SwitchBase [/192.168.144.128:58133 DPID[00:00:00:0c:29:f2:bb:77]]
20:03:55.021 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:0c:29:f2:bb:77 connected.
```

Imagen 35: Iniciando Floodlight.

Observamos al iniciar el controlador externo, que se ha conectado el puente cuyo DPID (Data Path ID) es: 00:00:00:0c:29:f2:bb:77

El DPID es un identificador único del switch, que según las especificaciones [11] OpenFlow los 48bits más bajos corresponden con la MAC del switch que hemos creado. Los 16 bits más significativos están definidos en la implementación (por ejemplo esos 16 bits pueden usarse para diferenciar múltiples instancias virtuales del mismo switch, para el uso de redes virtuales).

La MAC que se asocia al puente, es la MAC del primer puerto que se añade (hasta entonces no proporciona el DPID).

También podemos ver información [12] obtenida por el controlador a través de una interfaz web incluida en la utilidad Floodlight.

```
http://127.0.0.1:8080/ui/index.html
```

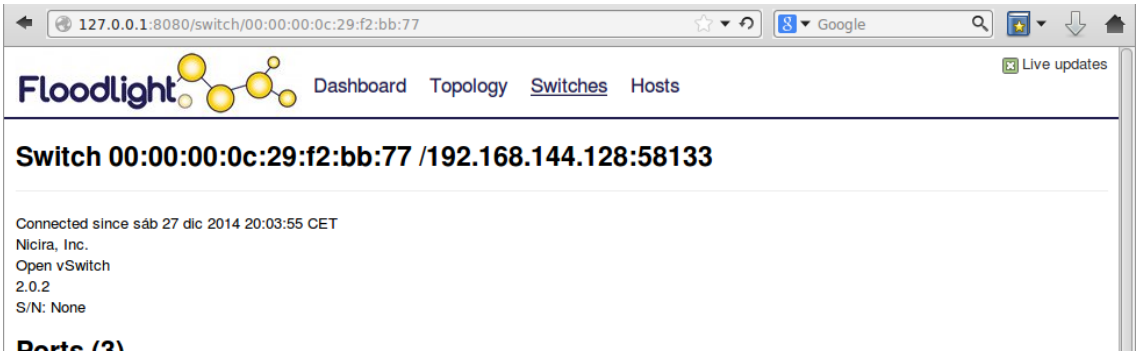


Imagen 36: Floodlight interfaz web.

Instalamos la utilidad *curl*, la cual la utilizaremos para comunicarnos con el controlador externo para enviar órdenes y recibir información. Disponemos de más información de la API de Floodlight en la referencia [13].

```
apt-get install curl
```

Ahora procederemos a introducir reglas a través del controlador externo.

En primer lugar borramos las anteriores.

```
curl
     http://127.0.0.1:8080/wm/staticflowentrypusher/clear/00:00:
     00:0c:29:f2:bb:77/json
```

Introducimos dos reglas, una para que todo lo que entre por el puerto 1 del puente salga por el puerto 2 y la segunda para que todo lo que entre por el puerto 2 salga por el 1 (igual que se ha hecho en el caso sin controlador).

```
curl -d '{"switch":"00:00:00:0c:29:f2:bb:77", "name":"flujo1",
     "ingress-port":"1", "active":"true", "actions":"output=2"}'
     http://127.0.0.1:8080/wm/staticflowentrypusher/json

curl -d '{"switch":"00:00:00:0c:29:f2:bb:77", "name":"flujo2",
     "ingress-port":"2", "active":"true", "actions":"output=1"}'
     http://127.0.0.1:8080/wm/staticflowentrypusher/json
```

Comprobamos que funciona mediante Wireshark y ping.

Ahora borramos las reglas anteriores e introducimos otras reglas.

Una primera regla, que realice que todo lo que entre por el puerto 1, cambie su dirección IP de destino a 200.0.1.4 (eth0 de Dickens) y se envíe por el puerto 2.

La segunda regla hace que todo lo que entre por el puerto 2 salga por el puerto 1.

```
curl -d '{"switch": "00:00:00:0c:29:f2:bb:77",
     "name":"uno","ingress-port":"1", "active":"true",
```

```
      "actions":"set-dst-ip=200.0.1.4,output=2"}'
      http://localhost:8080/wm/staticflowentrypusher/json
curl -d '{"switch": "00:00:00:0c:29:f2:bb:77",
      "name":"dos","ingress-port":"2", "active":"true",
      "actions":"output=1"}'
      http://localhost:8080/wm/staticflowentrypusher/json
```

Antes de probar estas reglas, introducimos en la cache ARP de Neruda una entrada manualmente para que crea que la dirección MAC 00:0C:29:6b:23:f7 (eth0 de Dickens) corresponde a la dirección IP 200.0.1.10.

```
sudo arp -i eth0 -s 200.0.1.10 00:0c:29:6b:23:f7
```

Ahora desde Neruda realizamos un ping a la dirección IP 200.0.1.10 y observamos mediante Wireshark las reglas introducidas al openvswitch.

Primero observamos que el ping funciona correctamente porque aunque las respuestas lleguen con dirección IP origen diferente a la dirección destino de las peticiones del ping (ya que en las reglas en sentido contrario no se ha hecho el cambio de IP), el ping lo que tiene en cuenta es un ID y un número de secuencia.

```
jm@neruda ~ $ ping -c2 200.0.1.10
PING 200.0.1.10 (200.0.1.10) 56(84) bytes of data.
64 bytes from 200.0.1.4: icmp_seq=1 ttl=64 time=61.8 ms
64 bytes from 200.0.1.4: icmp_seq=2 ttl=64 time=0.832 ms

--- 200.0.1.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.832/31.331/61.830/30.499 ms
```

Imagen 37: Ping desde Neruda a 200.0.1.10

En esta captura de pantalla observamos que es lo que envía y recibe Neruda.

```
3 6.24347100 200.0.1.3          200.0.1.10          ICMP   98 Echo (ping) request  id=0x1dfb, seq=1/25
4 6.27731200 Vmware_6b:23:f7    Broadcast           ARP    60 Who has 200.0.1.3?  Tell 200.0.1.4
5 6.27733000 Vmware_ac:2a:d2    Vmware_6b:23:f7     ARP    42 200.0.1.3 is at 00:0c:29:ac:2a:d2
6 6.30528600 200.0.1.4          200.0.1.3           ICMP   98 Echo (ping) reply    id=0x1dfb, seq=1/25
7 7.24552300 200.0.1.3          200.0.1.10          ICMP   98 Echo (ping) request  id=0x1dfb, seq=2/51
8 7.24631200 200.0.1.4          200.0.1.3           ICMP   98 Echo (ping) reply    id=0x1dfb, seq=2/51
```

Imagen 38: Wireshark en Neruda.

En esta otra captura de pantalla observamos lo que recibe y envía Dickens.

```
1 0.00000000 200.0.1.3          200.0.1.4           ICMP   98 Echo (ping) request  id=0x1dfb, seq=1/25
2 0.00003700 Vmware_6b:23:f7    Broadcast           ARP    42 Who has 200.0.1.3?  Tell 200.0.1.4
3 0.06081100 Vmware_ac:2a:d2    Vmware_6b:23:f7     ARP    60 200.0.1.3 is at 00:0c:29:ac:2a:d2
4 0.06083400 200.0.1.4          200.0.1.3           ICMP   98 Echo (ping) reply    id=0x1dfb, seq=1/25
5 1.00208400 200.0.1.3          200.0.1.4           ICMP   98 Echo (ping) request  id=0x1dfb, seq=2/51
6 1.00211800 200.0.1.4          200.0.1.3           ICMP   98 Echo (ping) reply    id=0x1dfb, seq=2/51
```

Imagen 39: Wireshark en Dickens.

Observando las dos capturas de Wireshark se ven los cambios que se han producido según las reglas introducidas al puente.

Por último vamos a comprobar las reglas que hemos añadido. Primero con la utilidad openvswitch (desde la máquina que tenemos el puente).

```
sudo ovs-ofctl dump-flows puente
```

```
jgrajos@soekris1:~$ sudo ovs-ofctl dump-flows puente
NXST_FLOW reply (xid=0x4):
 cookie=0xa000000e4a9ae3, duration=1133.075s, table=0, n_packets=13, n_bytes=1892, idle_age=241, pri
ority=32767,in_port=1 actions=mod_nw_dst:200.0.1.4,output:2
 cookie=0xa000000e3f0f41, duration=1107.127s, table=0, n_packets=7, n_bytes=648, idle_age=399, prior
ity=32767,in_port=2 actions=output:1
jgrajos@soekris1:~$
```

Imagen 40: Reglas desde OpenVswitch.

Observamos en la captura de pantalla que se pueden ver e interpretar claramente las reglas que hemos introducido.

Ahora vamos a intentar obtener la información a través del controlador externo.

```
curl
     http://localhost:8080/wm/staticflowentrypusher/list/00:00:0
     0:0c:29:f2:bb:77/json
```

```
jm@neruda ~ $ curl http://localhost:8080/wm/staticflowentrypusher/list/00:00:00:0c:29:f2:bb:77/js
on
{"00:00:00:0c:29:f2:bb:77":{"uno":{"version":1,"type":"FLOW_MOD","length":88,"xid":0,"match":{"da
taLayerDestination":"00:00:00:00:00:00","dataLayerSource":"00:00:00:00:00:00","dataLayerType":"0x
0000","dataLayerVirtualLan":-1,"dataLayerVirtualLanPriorityCodePoint":0,"inputPort":1,"networkDes
tination":"0.0.0.0","networkDestinationMaskLen":0,"networkProtocol":0,"networkSource":"0.0.0.0","
networkSourceMaskLen":0,"networkTypeOfService":0,"transportDestination":0,"transportSource":0,"wi
ldcards":4194302},"cookie":45035996513475299,"command":0,"idleTimeout":0,"hardTimeout":0,"priorit
y":32767,"bufferId":-1,"outPort":-1,"flags":0,"actions":[{"type":"SET_NW_DST","length":8,"network
Address":-939523836,"lengthU":8},{"type":"OUTPUT","length":8,"port":2,"maxLength":32767,"lengthU"
:8}],"lengthU":88},"dos":{"version":1,"type":"FLOW_MOD","length":80,"xid":0,"match":{"dataLayerDe
stination":"00:00:00:00:00:00","dataLayerSource":"00:00:00:00:00:00","dataLayerType":"0x0000","da
taLayerVirtualLan":-1,"dataLayerVirtualLanPriorityCodePoint":0,"inputPort":2,"networkDestination"
:"0.0.0.0","networkDestinationMaskLen":0,"networkProtocol":0,"networkSource":"0.0.0.0","networkSo
urceMaskLen":0,"networkTypeOfService":0,"transportDestination":0,"transportSource":0,"wildcards":
4194302},"cookie":45035996512718657,"command":0,"idleTimeout":0,"hardTimeout":0,"priority":32767,
"bufferId":-1,"outPort":-1,"flags":0,"actions":[{"type":"OUTPUT","length":8,"port":1,"maxLength":
32767,"lengthU":8}],"lengthU":80}}}jm@neruda ~ $
jm@neruda ~ $
```

Imagen 41: Reglas desde controlador externo.

Observando detenidamente la captura de pantalla, se puede llegar a interpretar las reglas introducidas, pero no nos ofrece la información de una manera clara.

5.4.3. OPENVSWITCH - CONTROLADOR EXTERNO

En este apartado vamos a comentar aspectos que nos hemos encontrado cuando durante el periodo de realización de estas tareas, los desarrolladores de Floodlight han actualizado esta utilidad..

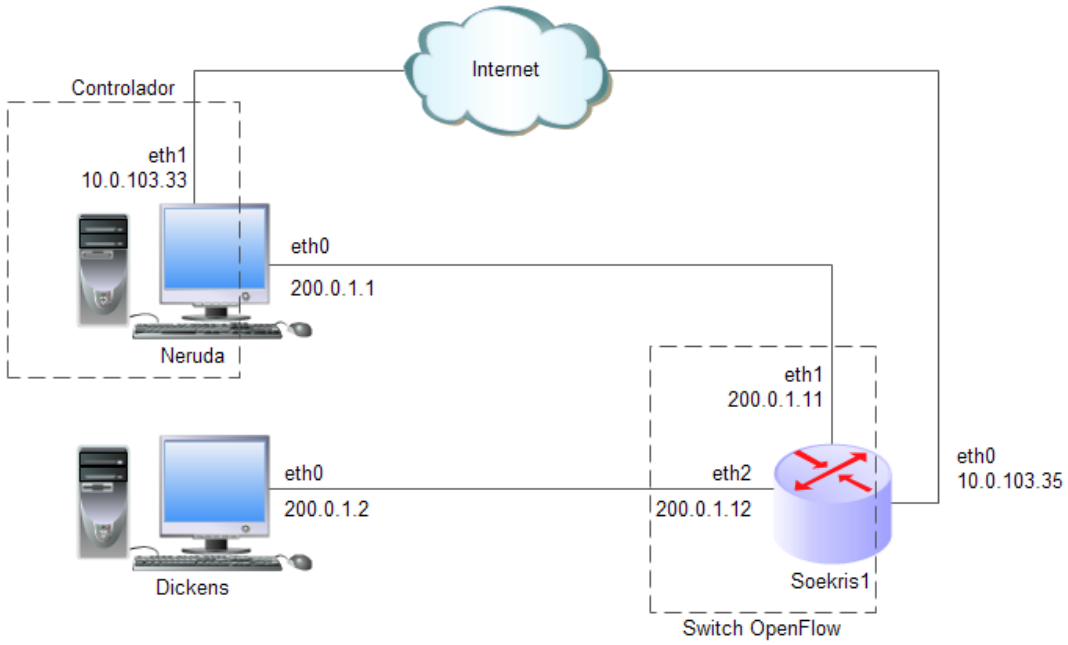


Imagen 42: Diagrama de red OpenVSwitch - controlador.

Observamos al descargar el controlador externo la disponibilidad de una nueva versión de Floodlight (v1.0) que implementa una nueva API REST (Interfaz para comunicarse con el controlador).

Escogemos utilizar esta nueva versión por utilizar la nueva sintaxis de su API ya que es como nos entenderemos en las tareas futuras con el controlador.

En un primer lugar se plantea actualizar el Openvswitch (de la versión 2.0.2 a la versión 2.3.1). No consiguiendo un funcionamiento esperado del switch virtual a causa de una supuesta instalación no completamente correcta y además de la observación de cambios en el funcionamiento del Openvswitch en la versión 2.3.1, se escoge utilizar la versión por defecto (2.0.2) de los repositorios oficiales de Ubuntu 14.04.01 LTS (versión de SO que estamos utilizando).

Tendremos en cuenta que la versión 2.0.2 de Openvswitch trabaja sobre la versión de OpenFlow 1.0.

Los campos de los datagramas sobre los cuales podemos trabajar los podemos encontrar en la siguiente referencia a la documentación del controlador [14].

Comentamos que en la documentación existen varios errores que dificultan la comprensión de la sintaxis de la nueva API REST. Se han conseguido solucionar los problemas planteados, pero destacamos este aspecto para que no se siga la

documentación punto por punto. Por ejemplo, la orden que muestra la documentación para consultar las reglas que se han añadido no funciona correctamente.

Imagen 43: Ejemplo de documentación errónea.

Entendemos que estos errores vienen de la mezcla de aspectos de su versión anterior.

Como comentaremos más adelantes el comando utilizado para obtener dichas reglas es el siguiente:

```
curl http://localhost:8080/vm/core/switch/all/flow/json
```

Probamos un ejemplo para probar el funcionamiento entre el controlador y el Openvswitch.

Especificamos con las siguientes reglas que todo en Soekris1 lo que entre por la interfaz 1 (eth1) salga por la interfaz 2 (eth2) y todo lo que entre por la interfaz 2 (eth2), se cambie su dirección IP de origen y salga por la interfaz 1 (eth1).

```
curl -d '{"switch": "00:00:00:00:24:d0:3a:91", "name":"flow1",
      "in_port":"1","active":"true", "actions":"output=2"}'
      http://127.0.0.1:8080/wm/staticflowpusher/json

curl -d '{"switch": "00:00:00:00:24:d0:3a:91", "name":"flow2",
      "in_port":"2","active":"true",
      "actions":"set_ipv4_src=200.0.1.55,output=1"}'
      http://127.0.0.1:8080/wm/staticflowpusher/json
```

A continuación realizamos un ping desde Neruda (200.0.1.1) hacia Dickens (200.0.1.2) y observamos que funcionan correctamente las reglas introducidas en la siguiente captura de Wireshark.

1	0.000000000	Olicom_d0:3a:91	LLDP_Multicast	LLDP	61 Chassis Id = 00:00:24:d0:3a:91 Port Id = TTL
2	0.097830000	Olicom_d0:3a:91	Broadcast	0x8942	69 Ethernet II
3	6.622462000	AsustekC_a3:da:c0	Broadcast	ARP	42 Who has 200.0.1.2? Tell 200.0.1.1
4	6.623148000	AsustekC_0c:03:eb	AsustekC_a3:da:c0	ARP	60 200.0.1.2 is at 00:1d:60:0c:03:eb
5	6.623175000	200.0.1.1	200.0.1.2	ICMP	98 Echo (ping) request id=0x0dfe, seq=1/256, ttl
6	6.623929000	200.0.1.55	200.0.1.1	ICMP	98 Echo (ping) reply id=0x0dfe, seq=1/256, ttl
7	7.624140000	200.0.1.1	200.0.1.2	ICMP	98 Echo (ping) request id=0x0dfe, seq=2/512, ttl
8	7.624311000	200.0.1.55	200.0.1.1	ICMP	98 Echo (ping) reply id=0x0dfe, seq=2/512, ttl

Imagen 44: Captura de Wireshark.

Por último observamos a través del controlador las órdenes establecidas.

```
jgrajos@neruda ~ $ curl http://localhost:8080/wm/core/switch/all/flow/json | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   533    0   533    0     0    524      0 --:--:--  0:00:01 --:--:--   525
{
    "00:00:00:00:24:d0:3a:91": {
        "flows": [
            {
                "actions": {
                    "output": "2"
                },
                "byteCount": "628",
                "cookie": "45035998409453772",
                "durationSeconds": "98",
                "hardTimeoutSec": "0",
                "idleTimeoutSec": "0",
                "match": {
                    "in_port": "1"
                },
                "packetCount": "4",
                "priority": "65535",
                "tableId": "0x0",
                "version": "OF_10"
            },
            {
                "actions": {
                    "output": "1",
                    "set_ipv4_src": "200.0.1.55"
                },
                "byteCount": "0",
                "cookie": "45035998409453773",
                "durationSeconds": "97",
                "hardTimeoutSec": "0",
                "idleTimeoutSec": "0",
                "match": {
                    "in_port": "2"
                },
                "packetCount": "0",
                "priority": "65535",
                "tableId": "0x0",
                "version": "OF_10"
            }
```

Imagen 45: Visualización de las reglas de flujo introducidas.

En este caso destacar que se utilizado el comando "`python -m json.tool`" para mostrar la salida con un aspecto más amigable.

5.5. REPLICA SOEKRIS

En esta sección vamos a comentar el proceso seguido una vez que hemos recibido los nuevos discos SSD para los equipos Soekris, que van a sustituir a las memoria USB utilizadas hasta ahora. En primer lugar describiremos una serie de puntos a tener en cuenta y realizar antes del cambio de disco. Y por último comentaremos las tareas a realizar después del cambio de disco para dejar los equipos preparados.

5.5.1. PREPARANDO IMAGEN

Analizamos el dispositivo de bloques de la memoria USB en la cual está contenida la instalación del sistema operativo (Ubuntu Server) y observamos que en los primeros sectores del disco encontramos una partición EXT4 y a continuación la partición SWAP. Como en el resultado final vamos a disponer de una memoria de mayor tamaño y su principal uso lo queremos para la partición EXT4, realizaremos una serie de operaciones para preparar la imagen del disco en el que dispongamos en primer lugar la partición SWAP y a continuación la partición EXT4 para que cuando finalicemos la copia de la imagen en el nuevo disco nos sea más sencillo realizar la expansión de la partición.

Estas operaciones las realizaremos en un ordenador auxiliar y utilizando la herramienta GParted.

En primer lugar eliminamos la partición SWAP, a continuación podemos redimensionar la partición EXT4 si consideramos oportuno y movemos esta partición desplazándola hacia la derecha hasta tener a su izquierda el espacio que queramos establecer en la partición SWAP. En el espacio libre sin asignar, en los primeros sectores del disco creamos una nueva partición con un sistema de ficheros "Linux-swap". Por último aplicamos todas las operaciones.

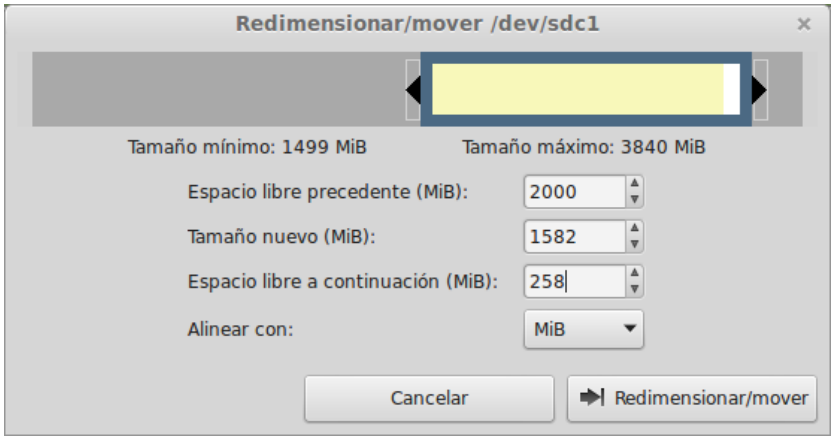


Imagen 46: Gparted redimensionar disco.

Después de estas modificaciones, volvemos a poner la memoria USB en el equipo Soekris e iniciamos. Desde el dispositivo Soekris cambiamos el identificador de la partición SWAP para que el sistema operativo haga utilidad de ella. Conseguimos este UUID con el comando "`sudo blkid`" y editamos el fichero `/etc/fstab` para remplazar el anterior UUID por el nuevo. Y por último activamos la SWAP con el comando "`sudo swapon –U 'nuevo_UUID' `" [15].

La última cuestión a considerar antes utilizar los nuevos discos SSD, para no tener problemas al reconocer las interfaces de red al iniciar la imagen de disco que estamos preparando borramos el fichero `/etc/udev/rules.d/70-persistent-net-rules` para que al iniciar esta imagen de disco en un Soekris diferente no tengamos problemas con la identificación de las interfaces de red.

5.5.2. REPLICANDO IMAGEN

Colocamos el disco SSD en un equipo Soekris y arrancamos el SO desde la memoria USB y en otra memoria USB colocamos la imagen que hemos preparado.

Con el comando `lsblk` consultamos los dispositivos de bloques que se reconocen en nuestra máquina y observamos que tenemos en `sda` el disco SSD, en `sdb` el sistema operativo que está funcionando y en `sdc` la imagen preparada.

Procedemos a copiar la imagen al disco SSD con el siguiente comando:

```
sudo dd if=/dev/sdc of=/dev/sda &
```

Para consultar el estado de la copia podemos ejecutar el siguiente comando:

```
sudo kill -USR1 'pidof dd'
```

Una vez finalizada la copia de la imagen en el disco SSD, vamos a proceder a extender la partición para aprovechar la nueva capacidad y pora ello ejecutamos los siguientes pasos[1]:

1. Lanzamos la aplicación `fdisk` para entrar en el menú para administrar las particiones de `sda`

   ```
   sudo fdisk /dev/sda
   ```

2. Pulsamos la tecla `p` y vemos las particiones del dispositivo de bloques.
3. Observamos que la partición que queremos extender es la número 1 y por ello pulsamos la tecla `d` y a continuación la tecla `1`.
4. Volvemos a pulsar la `p` para comprobar que hemos eliminado la partición correcta.
5. Pulsamos la tecla `n` para crear una partición y dejamos las 4 opciones siguientes por defecto (partición primaria, número 1 y primer y último sector por defecto para ocupar todo el espacio disponible).
6. Por ultimo volvemos a pulsar `p` para comprobar que hemos creado una nueva partición.
7. Finalmente pulsamos `w` para escribir los cambios en el disco.

Una vez realizada la expansión de la partición apagamos la máquina, desconectamos las dos memorias USB y comprobamos que arranca el sistema operativo desde el disco SSD sin problemas. Destacamos que no hemos tenido problemas con el MBR (Registro Principal de Arranque) ni modificar nada al respecto para que el sistema operativo arranque sin problemas desde esta nueva unidad de almacenamiento.

[1] Aunque se usan los términos borrar partición, destacar que no se pierden los datos.

5.5.3. CONFIGURANDO EQUIPO

A continuación procedemos a cambiar el nombre del equipo modificando los ficheros `/etc/hostname` y `/etc/hosts` y reiniciamos para comprobar los cambios. Los nombres de equipo de las máquinas situadas en el Laboratorio del GCO en Valladolid van a ser `soekris4` y `soekris5` y las máquinas situadas en Madrid en las oficinas de Telefónica se llamarán `soekris1`, `soekris2` y `soekris3`.

También se asignarán direcciones de red diferentes modificando el fichero `/etc/network/interfaces`, quedando su contenido por ejemplo para la máquina `soekris2` de la siguiente forma:

```
auto eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet static
address 200.0.1.21



auto eth2
iface eth2 inet static
address 200.0.1.22



auto eth3
iface eth3 inet static
address 200.0.1.23
```

Comentar que la dirección de red de la interfaz eth0 se asigna mediante DHCP, ya que es la interfaz de red por la cual cada máquina va tener acceso a Internet y se conectará al controlador externo (controlador *Floodlight* en Neruda). Las direcciones de red de las interfaces de red eth1, eth2 y eth3 de todos los Soekris realmente no son necesarias ya que estas interfaces van a pertenecer al *OpenVSwitch*, aunque recomendamos especificarlas para que por lo menos se activen.

También destacamos que el DPID del switch virtual se ha cambiado automáticamente al reconocer distintas interfaces de red de las de la máquina en la que preparamos la configuración inicial, asique en este sentido tampoco vamos a tener problemas.

Por último se harán unas pruebas de rendimiento de escritura en disco para analizar la mejora de utilizar el disco SSD respecto a la memoria USB.

Los resultados con el Nuevo disco SSD son:

```
jgrajos@soekris2:~$ dd if=/dev/zero of=prueba count=100 bs=512k
100+0 records in
100+0 records out
52428800 bytes (52 MB) copied, 0.400025 s, 131 MB/s
jgrajos@soekris2:~$ dd if=/dev/zero of=prueba count=800 bs=512k
800+0 records in
800+0 records out
419430400 bytes (419 MB) copied, 4.56829 s, 91.8 MB/s
```

Imagen 47 Rendimiento SSD en `soekris2`.

Recordamos los datos de rendimiento de escritura en disco de la *Tabla 2* de la sección *Test de escritura en disco* del equipo Soekris con la memoria USB para un mejor análisis de este nuevo resultado.

Fichero			Soekris			
count	bs	Tamaño	Memoria USB		Disco SSD	
100	512	52MB	0.368s	142MB/s	0.400s	131MB/s
800	512	419MB	124s	3.4MB/s	4.568s	91.8MB/s

Tabla 4: Comparativa escritura en disco Memoria USB vs Disco SSD.

Observando los datos vemos que para las copias de ficheros de tamaño más grande, obtenemos una mejora de rendimiento mayor.

Pasando de 3.4MB/s a 91.8MB/s obtenemos una mejora aproximada de 27 veces mejor.

5.5.4. CONEXIÓN AL CONTROLADOR EXTERNO

La interfaz de red eth1 de Neruda, por la cual se conectaban los Soekris al controlador se le ha asignado una dirección de red estática para poder establecer una configuración fija en los Soekris. Esta dirección de red es 10.0.103.27. Recordar que las interfaces eth0 de los Soekris son asignadas mediante DHCP.

A continuación se incluyen unas capturas de la interfaz web de Floodlight para comprobar que los 4 Soekris están conectados correctamente y que se reconoce la topología.

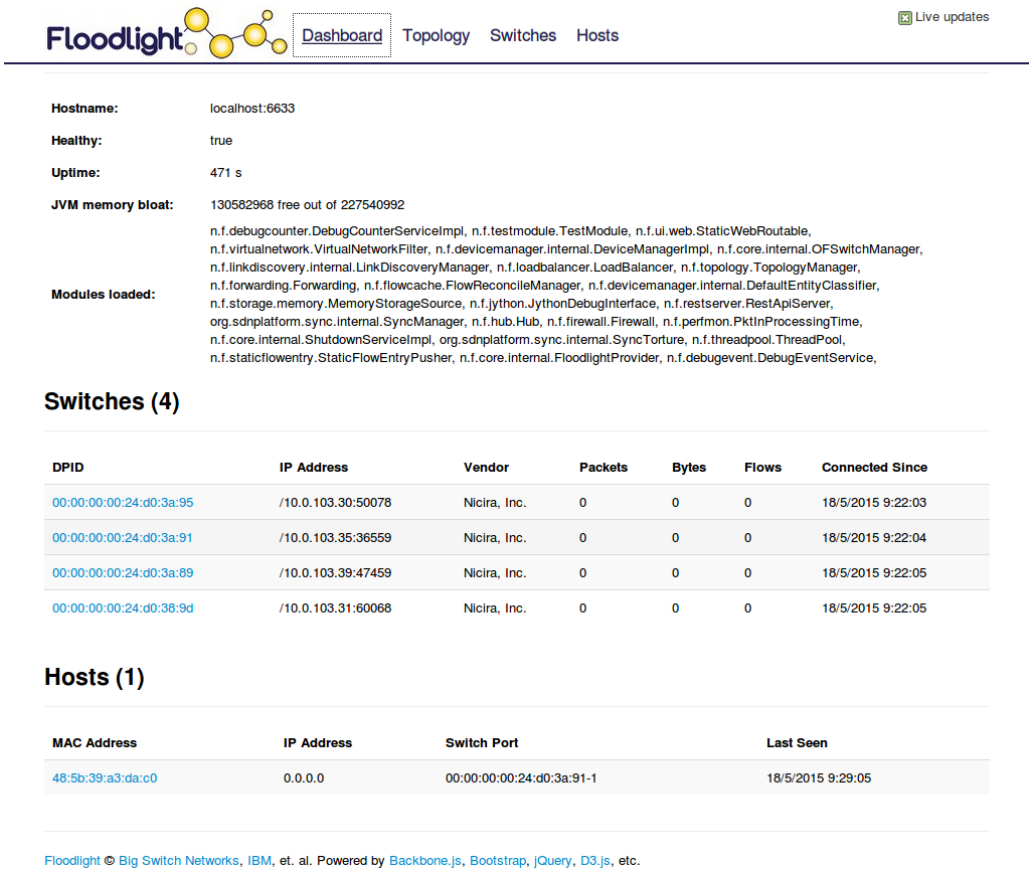


Imagen 48 Interfaz web Floodlight.

Para esta pequeña prueba se ha conectado Neruda a la interfaz eth1 de soekris2 y además se han conectado entre sí las interfaces eth3 y eth2 de soekris2 y soekris4 respectivamente.

Imagen 49 Topología reconocida en Floodlight.

5.6. CONCLUSIONES

Para concluir esta primera parte vamos a comentar el recorrido hecho con ciertas valoraciones personales. Desde el principio comentar que al estar acostumbrado a trabajar con dispositivos tales como PC, Smartphones y equipos de usuario, por el contrario estos nuevos dispositivos con los que hemos trabajado conocen poco el término *user friendly* y a veces un poco frustrante que se hiciesen de rogar ciertas tareas, como completar la instalación del sistema operativo de los equipos Soekris. Una vez solventados los primeros problemas he disfrutado mucho utilizando este tipo de tecnología (RS-232, SSH, etc…).

Siguiendo con los dispositivos físicos, comentar como hemos conseguido mejorar ciertas características como la escritura en disco del equipo durante la realización de este trabajo con la inclusión de los discos SSD. Sobre los test de rendimiento de las interfaces de red, aunque comprobamos que en determinadas condiciones podíamos obtener un rendimiento un poco inferior al óptimo pero tratándose de equipos para fines de desarrollo podemos decir que han cumplido perfectamente.

Quisiera volver a hacer hincapié cuando analizamos las características de la red al involucrar los equipos Soekris. Recordar la peculiaridad entre algunos pares de interfaces cuando eran usadas para recibir y enviar datos, que disminuía el rendimiento, probablemente porque en dichos pares de red compartieran ciertos recursos *hardware*.

Después de todas las pruebas realizadas, volvemos a destacar que teníamos una limitación en la red en número de tramas por segundo que la red podía tratar (~70.000 tramas/s) siempre y cuando estas no superasen el ancho de banda disponible de la red, que entonces pasaría a medirse la capacidad en Mb/s.

En los siguiente avances de este trabajo, en la implementación de un switch virtual OpenFlow en los equipos Soekris y la utilización de uno de los equipos auxiliares como controlador externo, hemos podido aprender, como a través de ciertas APIs, de una manera sencilla establecer reglas al switch para cambiar el comportamiento sobre el encaminamiento y el tratamiento de los datos atravesados.

Una vez que habíamos acabado de incorporar las funcionalidades comunes para los equipos Soekris y también debido a la recepción de los nuevos dispositivos de almacenamiento SSD, estuvimos aprendiendo como poder replicar el contenido de arranque y almacenamiento para luego llevarlo a cabo y poder utilizar varios dispositivos Soekris a la vez y con ello poder crear topologías más complejas. Esta parte del trabajo es de gran utilidad ya que nos ofrece escalabilidad al proyecto y poder ampliar en un futuro en número de switches Soekris en el caso que se necesitasen.

6. IMPLEMENTACIÓN SDN

En este capítulo vamos a hacer una introducción de cómo hemos utilizado algunos módulos, como PCE (*Path Computation Element*), ABNO Controller (*Application-Based Network Operations Controller*), *Provisionig Manager* implementados en JAVA.

Este proyecto, en el que he empezado a contribuir, tiene multitud de protocolos, algoritmos y funciones incorporadas.

Estos módulos propiedad de Telefónica I+D, se pueden consultar públicamente en la siguiente referencia [16]. En el momento de desarrollo de esta parte estos repositorios no eran públicos y las versiones del código utilizado todavía siguen sin serlo. Por eso nos centraremos en el funcionamiento y no tanto en el código. Pequeñas partes de código, bajo previa autorización y con el interés de mostrar que han sido desarrolladas personalmente, las mostraremos oportunamente.

6.1. CONEXIÓN PCE – CONTROLADOR EXTERNO FLOODLIGHT

En esta parte hemos tenido que familiarizarnos con el proyecto en código JAVA en la que Telefónica I+D tienen implementados ciertos algoritmos para el cálculo de rutas, y protocolos relacionados con SDN y concretamente con PCE (*Path Computation Element*).

El objetivo ha sido intercomunicar este proyecto con el controlador externo, para que éste a su vez obtenga información de la topología actual y pase dichos datos al software JAVA.

En primer lugar tenemos que editar uno de los ficheros de configuración para que el programa conozca donde interactuar con nuestro controlador.

A través de unos ficheros de configuración cómo el `ControllerListFile.xml` describimos al controlador externo, para que la aplicación JAVA se pueda comunicar con este.

```
<?xml version="1.0" encoding="UTF-8"?>

<controller_list>

      <controller>

            <ip>10.0.103.27</ip><!--NERUDA-->

            <port>8080</port>

            <type>Floodlight</type>

    </controller>

</controller_list>
```

Una de las partes más importantes del código, es la parte del servidor, que es la encargada de llamar a los demás módulos cuando son necesarios. En la siguiente imagen podemos

observar una pequeña parte del código y cómo en la consola del entorno de trabajo Eclipse se observa que el servidor está en funcionamiento.

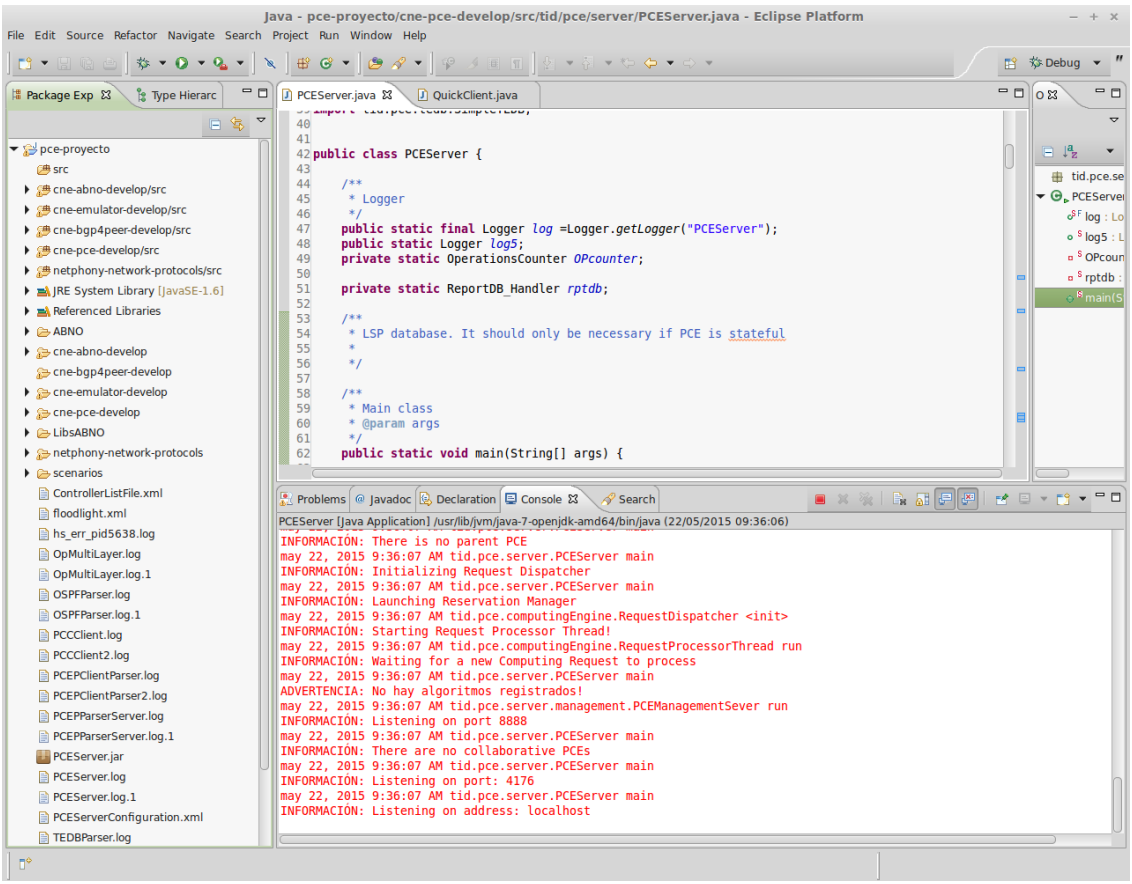


Imagen 50 PCEServer en Eclipse.

Después de pequeñas modificaciones para adaptar el código proporcionado, conseguimos con éxito el resultado de que dicha aplicación pueda obtener información de la topología dispuesta en ese momento.

Podemos observar la información que recaba el `PCEServer.java` en la siguiente imagen.

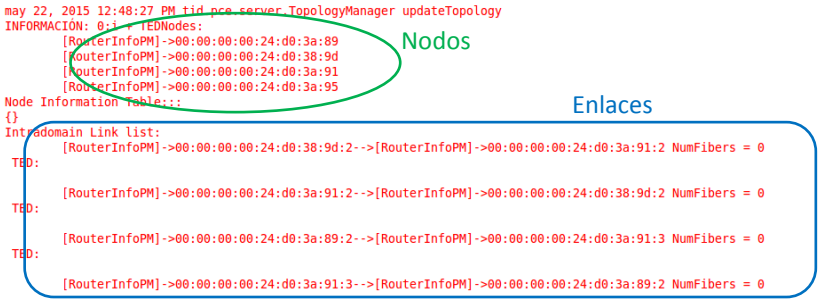


Imagen 51 Nodos y enlaces reconocidos por la aplicación JAVA.

Comprobamos que nos reconoce 4 nodos, los 4 equipos Soekris, y 4 enlaces. Estos 4 enlaces, son realmente 2, pero como son bidireccionales, nos reconoce 1 en cada sentido. Observamos que hemos conectado la interfaz 2 del router con DPID 00:00:00:00:24:d0:38:9d con la interfaz 2 del router con DPID 00:00:00:00:24:d0:3a:91 y la interfaz 2 del router con DPID 00:00:00:00:24:d0:3a:89 con la interfaz 3 del router con DPID 00:00:00:00:24:d0:3a:91.

6.2. PROYECTO EUROPEO STRAUSS

STRAUSS (*Scalable and efficient orchestration of Ethernet services using software-defined and flexible optical networks*), es un proyecto de investigación, financiado por la Comisión Europea, cuenta con colaboradores internacionales tanto de países de la Unión Europea como de Japón (Imagen 52).

Imagen 52: Mapa colaboradores – STRAUSS

El objetivo principal de este proyecto es mejorar la eficiencia de las infraestructuras ópticas multidominio para transporte Ethernet. STRAUSS propone una avanzada arquitectura de red de transporte Ethernet óptico definido por software.

Esta arquitectura, como se puede apreciar en la Imagen 53, consta de 4 capas. Capa de infraestructura de transporte de red, de virtualización de transporte de red, de control y gestión de infraestructura virtual y la capa de servicio extremo a extremo y orquestación de red.

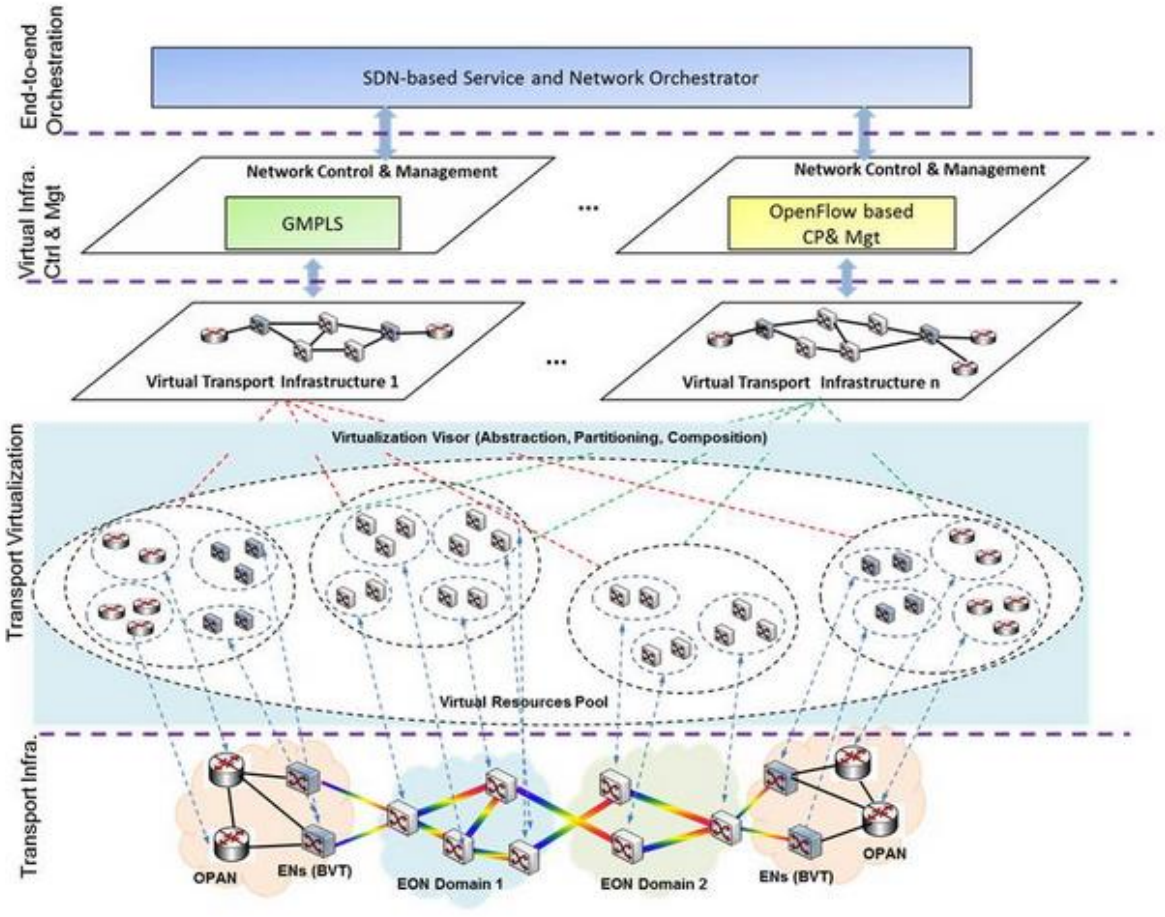


Imagen 53: Arquitectura de red – STRAUSS

6.3. PREPARANDO ESCENARIO

Una vez presentado el proyecto, se pretende realizar una demostración, pero primeramente vamos a realizar una serie de tareas previas para preparar una demostración del protocolo COP [17] (*Control Orchestration Protocol*), el cual utiliza un conjunto de funciones del plano de control para varios controladores SDN (*Software Defined Networking*).

En las tareas de preparación del escenario, para que a los Switch Openflow se les pueda identificar, se han tenido que incluir ciertos aspectos relacionados con el DPID.

Concretamente se han incorporado nuevos subobjetos como el *Unnumbered DataPath interface* para el protocolo PCEP *(Path Computation Element communication Protocol)*. Estas nuevas incorporaciones fueron programadas en JAVA.

Estos subobjetos todavía no estaban definidos. Concretamente el subobjeto mencionado se propone basándose en la RFC3209 [18] con el formato que se puede observar en la Imagen 54.

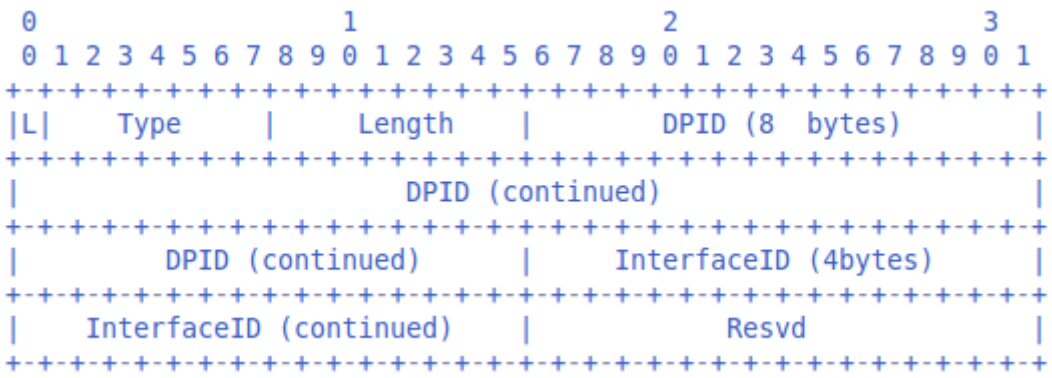


Imagen 54: Formato Subobjeto *Unnumbered DataPath Interface*

En la siguiente imagen (Imagen 55), podemos observar la implementación en java de dicho subobjeto.

```java
public class UnnumberedDataPathIDEROSubobject extends EROSubobject{

    public DataPathID dataPath;
    public long interfaceID;//32 bit Interface ID

    public long getInterfaceID() {
        return interfaceID;
    }

    public void setInterfaceID(long interfaceID) {
        this.interfaceID = interfaceID;
    }

    public UnnumberedDataPathIDEROSubobject(){
        super();
        this.erosolength=2+8+4+2; //header+dpid+interface+padding = 16
        this.setType(SubObjectValues.ERO_SUBOBJECT_UNNUMBERED_DATAPATH_ID);
    }

    public UnnumberedDataPathIDEROSubobject(byte[] bytes, int offset){
        super(bytes, offset);
        decode();
    }

    /**
     * Encode Unnumbered DataPathID ERO Subobject
     */
    public void encode(){

    /**
     * Decode Unnumbered DataPathID ERO Subobject
     */
    public void decode(){

    public DataPathID getDataPath() {
        return dataPath;
    }
    public void setDataPath(DataPathID dataPath) {
        this.dataPath = dataPath;
    }
    public String toString(){
        return dataPath.toString()+"::"+interfaceID;
    }
}
```

Imagen 55: UnnumberedDataPathIDEROSubobject.java

A través del repositorio de github del protocolo COP podemos descargar y autogenerar el código JAVA para disponer de un frontend con la sintaxis COP.

Para ello, siguiendo las instrucciones del repositorio podemos autogenerar la estructura de un API REST a partir de unos modelos yang, en los que están definidos los objetos del protocolo COP.

En la siguiente imagen podemos observar una parte de código del modelo YANG para definir la service-call, se puede consultar el código completo en la siguiente referencia [19].

```
grouping connection {
    description "The Connection represents an en

    leaf connection_id{
        type string;
    }
    container aEnd{
        uses endpoint;
    }
    container zEnd{
        uses endpoint;
    }
    container path{
        uses path-type;
    }
    container match {
        uses match-rules;
    }
    container traffic_params {
        uses traffic_params;
    }
    leaf controller_domain_id{
         type string;
    }
```

Imagen 56: service-call.yang

Una vez que hayamos autogenerado nuestros objetos, los obtendremos bajo un paquete llamado *"io.swagger.api"*. Una vez que tenemos la api rest generada proseguimos desarrollando la base de nuestro servidor para poder recibir las llamadas. Nos basamos en las librerías de un servidor web llamado JETTY [20] para generar este servidor y su servlet correspondiente. En la siguiente imagen (Imagen 57) podemos observar cómo hemos implementado dicho servicio.

```
ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);
context.setContextPath("/");

System.out.println("Abnoport: "+params.getAbnoPort());
Server jettyServer = new Server(params.getAbnoPort());
jettyServer.setHandler(context);
ServletHolder jerseyServlet =
        context.addServlet(com.sun.jersey.spi.container.servlet.ServletContainer.class, "/*");


jerseyServlet.setInitParameter(
        "com.sun.jersey.config.property.packages",
        "io.swagger.jaxrs.json;io.swagger.jaxrs.listing;io.swagger.api");

jerseyServlet.setInitParameter(
        "com.sun.jersey.spi.container.ContainerRequestFilters",
        "com.sun.jersey.api.container.filter.PostReplaceFilter");

jerseyServlet.setInitParameter(
        "com.sun.jersey.api.json.POJOMappingFeatures",
        "true");

jerseyServlet.setInitOrder(1);


try {
    jettyServer.start();
    jettyServer.join();
} finally {
    jettyServer.destroy();
}
```

Imagen 57: ABNOCOPController.java

Ahora podemos utilizar nuestro ABNO a través de este protocolo.

Realizamos una prueba para comprobar los cambios realizados.

Capturamos el tráfico de red generado y observamos el relacionado con el protocolo PCEP (Imagen 58).

```
1121 38.623063   127.0.0.1        127.0.0.1        PCEP   148 Path Computation Request (PCReq)
1135 38.842376   127.0.0.1        127.0.0.1        PCEP   168 Path Computation Reply (PCRep)
1140 38.921568   127.0.0.1        127.0.0.1        PCEP   176 Path Computation LSP Initiate (PCInitiate)
2745 45.596569   127.0.0.1        127.0.0.1        PCEP   176 Path Computation LSP State Report (PCRpt)
```

Imagen 58: Captura de tráfico PCEP

Podemos observar los mensajes intercambiados entre el ABNO Controller, el PCE y el *Provisioning Manager*.

En primer lugar el ABNO Controller envía una petición (PCReq) al PCE con la información de los puntos finales entre los que queremos hacer una provisión en la red. El PCE responde un PCRep con una ruta recomendada (*Explicit Route Object*, ERO). A continuación el ABNO Controller envía una solicitud LSP (*Label Switched Path*) al *Provisioning Manager* a través de un mensaje PCInitiate y este devuelve un PCRpt con el estado del LSP pedido.

A continuación (Imagen 59) mostramos el mensaje PCRep donde podemos observar el ERO y en detalle los subobjetos *Unnumbered DataPath Interfaces*. Como Wireshark no tiene noción de ellos los nombra como "*Non defined subobject*".

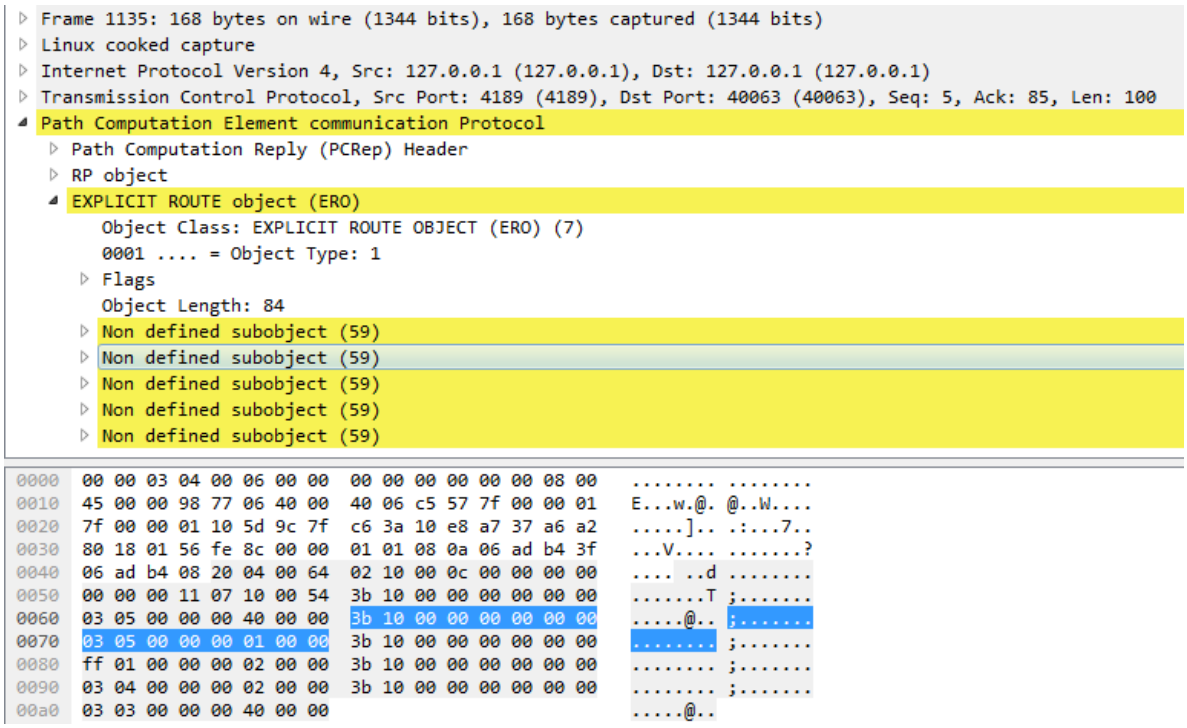


Imagen 59: Captura de tráfico Unnumbered DataPath Interface.

Comentamos el valor hexadecimal del subobjeto seleccionado observando que concuerda con la estructura definida en la Imagen 60. | 3B 10 | 00 00 00 00 00 00 03 05 | 00 00 00 01 | 00 00 |. El primer grupo de 2 bytes corresponden a la cabecera. El siguiente grupo de 8 bytes se corresponde al DataPathID del nodo recomendado. El tercer grupo de 4 bytes indican la interfaz de salida del nodo. Por último los 2 bytes son de relleno, ya que estos subobjetos como se indica en el RFC que nos hemos basado deben ser múltiplos de 4 bytes.

6.4. DEMOSTRACIÓN COP

En primer lugar ponemos en contexto la demostración a realizar. En la Imagen 61 podemos observar la topología disponible con nodos repartidos en las instalaciones de varios colaboradores.

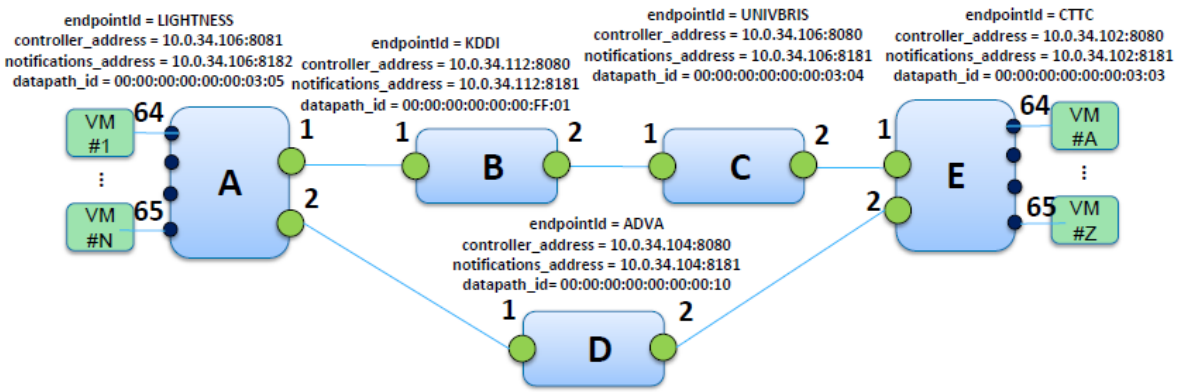


Imagen 61: Topología

Para realizar el test necesitamos hacer una petición al ABNO Controller para que provisione el camino. En el ejemplo hacemos la petición para el camino entre los nodos LIGHTNESS (A) y CTTC (E). Destacamos que, según se definió entre los participantes del proyecto, el camino con menos coste es: A→B→C→E. Con el siguiente comando empezamos la demostración.

```
curl -X POST -H "Content-type:application/json" -u admin:pswd1
     http://localhost:8080/restconf/config/calls/call/call_1/ -d
     '{"callId":"call_1","aEnd":{"routerId":"00:00:00:00:00:00:0
     3:05","interfaceId":"64","endpointId":"ep1"},
     "zEnd":{"routerId":"00:00:00:00:00:00:03:03","interfaceId":
     "64","endpointId":"ep2"},
     "trafficParams":{"latency":100,"reservedBandwidth":10000000
     0},"transportLayer":{"layer":"ethernet"}}'
```

En la siguiente imagen podemos observar el workflow a seguir en la demostración. Nuestro trabajo corresponde al *Multi-domain SDN Orchestrator*.

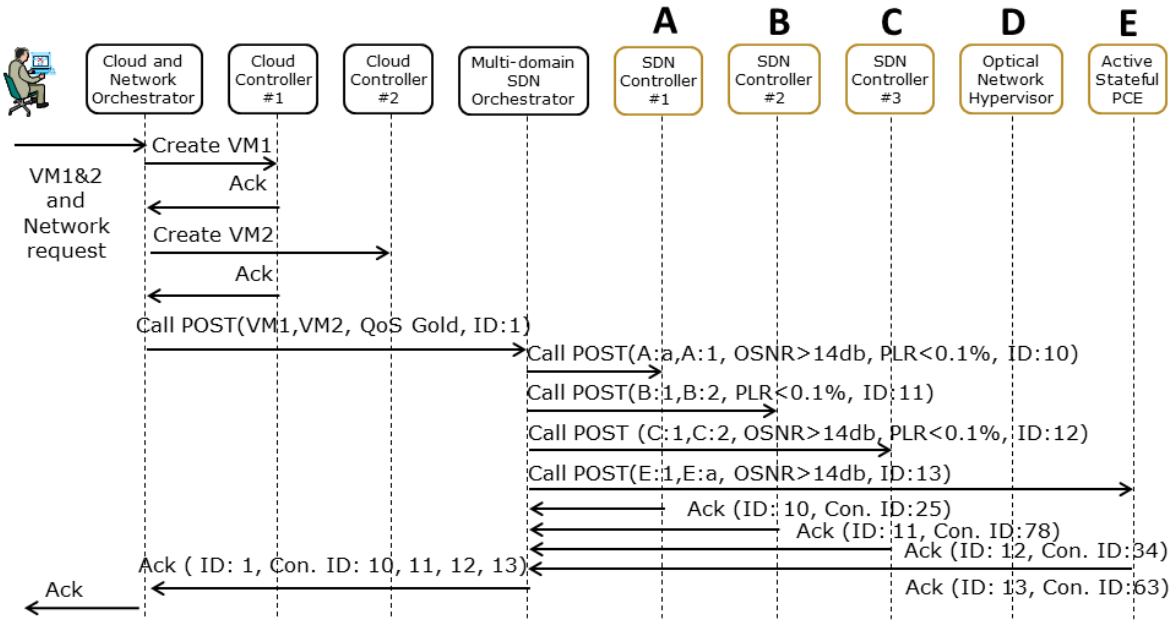


Imagen 62: Workflow COP

Como se observa, nuestra función es que al recibir una petición por parte del *"Cloud and Network Orchestrator"* procesarla y enviar a cada controlador la petición correspondiente para que se realice la cross-conexión.

Una vez preparados todos los colaboradores nos disponemos a realizar la prueba. En la Imagen 63 mostramos la captura de tráfico de la prueba realizada y podemos ver los mensajes en los que nuestro nodo estaba involucrado.

```
10.0.34.122     10.0.34.110     HTTP   746 POST /restconf/config/calls/call/1/ HTTP/1.1  (application/json)
10.0.34.110     10.0.34.102     HTTP   580 POST /restconf/config/calls/call/13/ HTTP/1.1
10.0.34.102     10.0.34.110     HTTP    73 HTTP/1.1 200 Successful operation  (application/json)
10.0.34.110     10.0.34.106     HTTP   607 POST /restconf/config/calls/call/12/ HTTP/1.1
10.0.34.110     10.0.34.106     HTTP   611 POST /restconf/config/calls/call/10/ HTTP/1.1
10.0.34.106     10.0.34.110     HTTP    73 HTTP/1.1 200 Successful operation  (application/json)
10.0.34.106     10.0.34.110     HTTP    73 HTTP/1.1 200 Successful operation  (application/json)
10.0.34.110     10.0.34.112     HTTP   585 POST /restconf/config/calls/call/11/ HTTP/1.1
10.0.34.112     10.0.34.110     HTTP    73 HTTP/1.1 200 Successful operation  (application/json)
10.0.34.110     10.0.34.122     HTTP    73 HTTP/1.1 200 Successful operation  (application/json)
```

Imagen 63: Captura de tráfico COP

En esta captura de Wireshark obtenida podemos observar los distintos mensajes que nuestro ABNO ha intercambiado, desde la petición recibida hasta las peticiones enviadas a los distintos controladores y todas sus respectivas respuestas.

6.5. CONCLUSIONES

Finalizando esta segunda parte de esta memoria, vamos a comentar unas conclusiones finales. En primer lugar hemos podido observar y comprobar cómo hemos podido conectar la información recopilada por el controlador SDN implementado por la utilidad Floodlight para que la implementación del PCE en JAVA desarrollado por Telefónica I+D pueda conocer información acerca de la topología. Esta interconexión entre el controlador externo y el PCE hace que este pueda realizar operaciones de *Path Computation* realizando estas consultas previas a los controladores.

Siguiendo por el orden de trabajo de esta memoria, el trabajo realizado dentro del marco del proyecto STRAUSS, cómo parte en la colaboración de Telefónica I+D ha conseguido afianzar aún más los conocimientos relacionados sobre la tecnología SDN. Con este trabajo he contribuido a mejorar las implementaciones que TID dispone para sus contribuciones y ofrece en forma de código abierto cada cierto tiempo a través de sus repositorios en la plataforma GitHub como hemos ido comentando. Concretamente recordamos las mejoras realizadas relacionadas con los identificadores DataPathID para el caso de los subobjetos UnnumberedDataPathIDEROSubobject heredados de la clase EROSubobject. Estas nuevas implementaciones nos han dado la posibilidad de solicitar peticiones extremo a extremo entre controladores SDN con DPID como identificadores.

También hemos aprendido a crear una instancia de un servidor web en código java para que a través del código generado de los repositorios del COP obtengamos un API para nuestro ABNO que podemos utilizar para recibir peticiones solicitadas. Esta integración hace que podamos reutilizar la estructura de la sintaxis común para este tipo de controladores que nos ofrece el COP.

Una vez hemos conseguido realizar estas modificaciones y mejoras, al poder observar el funcionamiento y el intercambio de mensajes entre los distintos actores nos ha proporcionado entender y comprender el funcionamiento de estas comunicaciones, como ha sido el ejemplo que explicamos en la memoria de como se le solicita un servicio extremo a extremo al ABNO y este mediante mensajes PCEP solicita al PCE mediante *Path Computation* el cálculo de una ruta y con este ERO el ABNO solicita mediante un mensaje PCInitiate al *Provisioning Manager* el provisionamiento extremo a extremo del camino a través de la solicitud en los distintos controladores.

Por último en el trabajo de construir un *testbed* para realizar una validación experimental del COP. Este nuevo protocolo presentado ofrece una interfaz común abstraída de la tecnología para controladores SDN. Además de todo el conocimiento técnico destacar el trabajo con colaboradores internacionales que ha hecho disfrutar de una manera especial de la realización de esta prueba experimental.

6.6. PAPER OFC 2016

OFC (*Optical Fiber Communication Conference*) es un congreso internacional sobre temas innovadores de comunicaciones ópticas. Este congreso está bajo la organización de la OSA (*The Optical Society*). Con los resultados obtenidos de la demostración que se ha comentado, se han sido incluidos como parte de un *paper* [21] , enviado y admitido al congreso OFC 2016 que se celebrará en el próximo marzo. A continuación se comenta el artículo presentado. En primer lugar agradecer la posibilidad de participar en él a Arturo Mayoral (CTTC), Alejandro Aguado (University of Bristol), Víctor López (TID) y Óscar González de Diós (TID) a los que también les tengo en estima la ayuda recibida. También quiero citar las instituciones participantes CTTC, University of Bristol, Telefónica I+D, ADVA Optical Networking, KDDI R&D Laboratories, NICT, Fujitsu LTD., Osaka.

6.6.1. RESÚMEN

Una API común de transportes es propuesta para enlazar la orquestación de nube/red, permitiendo interconectar los heterogéneos planos de control para dotar de provisionamiento y restablecimiento de servicios extremo a extremo con consciencia de QoS. En esta primera demostración se presenta en un entorno de pruebas con la colaboración de varias instituciones en la que incluimos una monitorización del plano de datos.

6.6.2. INTRODUCCIÓN

Para ofrecer un servicio de transporte extremo a extremo provisionando y orquestando a través de múltiples dominios con tecnologías heterogéneas de plano de transporte y control, un plano de control multidominio que gestione el trabajo es obligatorio. La OIF (*Optical Internetworking Forum*) y la ONF (*Open Networking Foundation*) presentó los resultados en su *Global Transport SDN Prototype Demonstration* [22] el pasado año, donde la mayoría de los controladores SDN y niveles jerárquicos fueron analizados. Actualmente, una API de transporte está estandarizada por la ONF.

El COP (*Control Orchestration Protocol*) es un buen candidato a API de transporte como una abstracción de un conjunto de funciones de plano de control comúnmente usadas por los controladores SDN, permitiendo trabajar de manera conjunta los paradigmas de un plano de control heterogéneo (por ejemplo: OpenFlow o GMPPLS/PCE). La propuesta COP proporciona una común NorthBound / SouthBound Interface (NBI/SBI) para los controladores SDN. Además, COP habilita la unión de la orquestación en la nube (computación, almacenamiento) y los recursos en red, al ser proporcionados en el provisionamiento de servicios de transporte extremo a extremo (E2E) entre máquinas virtuales (VM) atravesando múltiples dominios de red.

La definición del COP cubre una información topológica sobre la red, un servicio de llamada para el establecimiento de conexión E2E y un servicio de *Path Computation*. El modelo de información del COP es descrito a través de modelos YANG y con RESTconf como protocolo de transporte.

Las notificaciones de los parámetros ópticos supervisados están también incluidos de la definición de los modelos COP. El *service-call* del COP está definido como una interfaz de provisionamiento E2E. El objeto *Call* describe el tipo de servicio que es requerido o

servido (enlace DWDM, transporte extremo a extremo Ethernet, MPLS). Esto contiene los *endpoints* entre los cuales se crea el servicio. El objeto *Call* también incluye la lista efectiva de conexiones creadas en el plano de datos, para implementar el servicio extremo a extremo. El objeto de conexión (*Connection*) es utilizado en el ámbito de una red de un sólo dominio. La conexión incluye el camino de la topología de la red por el que los datos fluirán, el cual debe de estar plenamente detallado o abstraído dependiendo de la configuración utilizada en la orquestación o control. Finalmente la llamada introduce los parámetros de TE necesarios (ancho de banda, clase de QoS, latencia) que el servicio debe pedir.

Este artículo, es la primera propuesta arquitectónica y se realiza una validación experimental de una orquestación integrada de IT y recursos heterogéneos de red utilizando una común API de transporte para el provisionamiento y restauración de servicios con QoS. El mecanismo de recuperación sin QoS y sin COP ha sido previamente presentado en [3]. En este artículo se presenta el provisionamiento y resultados de la restauración en una prueba experimental *multi-partner* con *Data Centers* (DCs) distribuidos desde el proyecto LIGHTNESS hasta las instalaciones de CTTC y una red multidominio óptico de conmutación de circuitos o paquetes que cubre España, Reino Unido, Alemania y Japón. La calidad del plano de datos de transmisión (QoT) monitorizada está integrada dentro del plano de control para la primera parte para monitorizar las conexiones interdominios, el cual habilita recuperación de QoS tanto para dominios y extremo a extremo.

6.6.3. VM DINÁMICAS Y SERVICIO DE TRANSPORTE E2E DESARROLLADO CON PROVISIONAMIENTO CON QOS

En la Imagen 64 A se puede observar el escenario integrado, donde los DCs de LIGHTNESS y CTTC pueden estar orquestalmente unidos con un conjunto heterogéneo de redes de transporte, incluyendo dominios OPS/OCS. Cada dominio de red está controlado por su controlador SDN, un ONH (*Optical Network Hypervisor*) o un PCE *Stateful* activo. Cada dominio proporciona su topología abstraída (abstracción de nodo) para el orquestador SDN multidominio, el cual está basado en ABNO [25] (Imagen 64: A) Escenario LIGHTNESS-STRAUSS propuesto; B) Escenario abstracto de red/cloud; C) Ejemplo objeto *Call*; D) Clase QoS.Imagen 64 B muestra la topología observada por el orquestador SDN). Finalmente, un integrado *cloud* y orquestador de red es introducido para proporcionar la unión de orquestación en IT y los recursos de red.

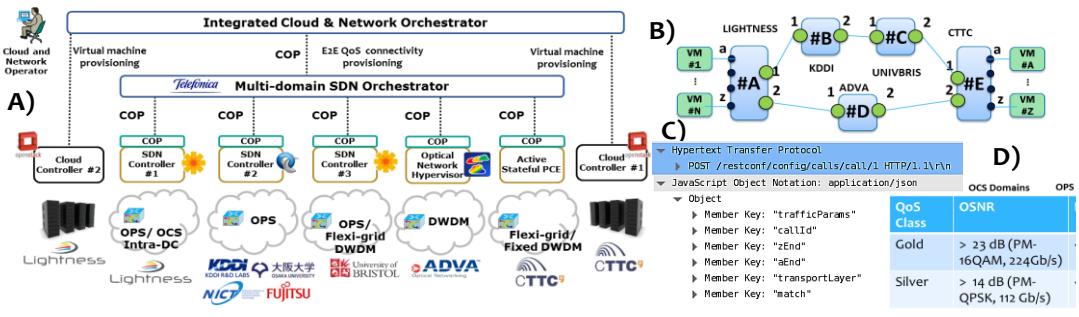


Imagen 64: A) Escenario LIGHTNESS-STRAUSS propuesto; B) Escenario abstracto de red/cloud; C) Ejemplo objeto *Call*; D) Clase QoS.

Con el objetivo de proporcionar un servicio de transporte E2E entre dos DC con QoS, nosotros hemos introducido dos clases QoS (Imagen 64 D) en la definición del objeto *Call* en el COP (Imagen 64 C, *trafficParams*). Cada clase de QoS define un cierto PLR (*Packet Loss Ratio*) para dominios OPS, y un cierto OSNR para dominios OCS, para obtener un ancho de banda pedido. El orquestador SDN podrá traducir la clase de nivel alto de QoS dentro de los correspondientes parámetros en las peticiones de llamadas para enviar a los diferentes controladores SDN. En la Imagen 65 A se muestra el mensaje intercambiado entre los elementos de computación y de red involucrados para conjuntamente proporcionar una interconexión entre VMs con QoS. El provisionamiento de las VMs es solicitada por cada controlador responsable de *cloud*, mientras que la interconexión de VM es pedida por el orquestador SDN con una llamada E2E (ID: 1) incluyendo una clase QoS. El orquestador SDN computa el camino extremo a extreme y solicita las llamadas necesarias (IDs: 10, 11, 12, 13) para los diferentes controladores SDN. En la Imagen 65 B vemos una captura de tráfico del integrado *cloud,* el orquestador de red y el orquestador SDN.

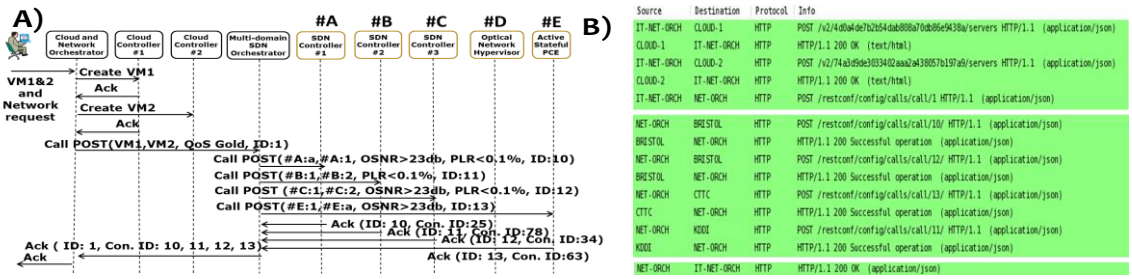


Imagen 65: A) Workflow de provisionamiento de conectividad de una VM; B) Captura Wireshark.

6.6.4. CONCLUSIÓNES

Este artículo primero introduce y valida un arquitectura para unir la orquestación de *cloud* y recursos de red con un protocolo de orquestación de control de código abierto en una colaboración internacional de plano de datos y control para realizar dichas pruebas, incluyendo recuperación interdominio y extremo a extremo basado en una monitorización QoT del plano de datos.

7. REFERENCIAS

[1] Software-Defined Networking: The New Norm for Networks. Disponible en: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf – [último acceso: 9 de Diciembre de 2015]

[2] OpFlex: An Open Policy Protocol White Paper. Disponible en: http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html – [último acceso: 9 de Diciembre de 2015]

[3] OpenFlow - Open Networking Foundation. Disponible en: https://www.opennetworking.org/sdn-resources/openflow – [último acceso: 9 de Diciembre de 2015]

[4] RFC 4655 A Path Computation Element (PCE)-Based Architecture. Disponible en: https://tools.ietf.org/html/rfc4655 – [último acceso: 9 de Diciembre de 2015]

[5] RFC 5440 Path Computation Element (PCE) Communication Protocol (PCEP). Disponible en: https://tools.ietf.org/html/rfc5440 – [último acceso: 9 de Diciembre de 2015]

[6] RFC 7491 A PCE-Based Architecture for Application-Based Network Operations. Disponible en: https://tools.ietf.org/html/rfc7491 – [último acceso: 9 de Diciembre de 2015]

[7] Mini Review AMD Phenom II X4 955 vs Intel Atom E640. Disponible en: http://www.cpu-world.com/Compare/851/AMD_Phenom_II_X4_955_(125W__rev._C2)_vs_Intel_Atom_E640.html – [último acceso: 9 de Diciembre de 2015]

[8] Web oficial de la utilidad Iperf. Disponible en: https://iperf.fr/ – [último acceso: 9 de Diciembre de 2015]

[9] Iperf Mailing Lists. Disponible en: http://sourceforge.net/p/iperf/mailman/message/19148111/ – [último acceso: 9 de Diciembre de 2015]

[10] Floodlight Getting Started. Disponible en: http://www.projectfloodlight.org/getting-started/ – [último acceso: 9 de Diciembre de 2015]

[11] OpenFlow Switch Specification v1.1.0. Disponible en: http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf – [último acceso: 9 de Diciembre de 2015]

[12] OpenFlow Starter Tutorial Lab. Disponible en: http://networkstatic.net/openflow-starter-tutorial-lab-1/ – [último acceso: 9 de Diciembre de 2015]

[13] Documentación Floodlight REST API. Disponible en: https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Floodlight+REST+API+pre-v1.0– [último acceso: 9 de Diciembre de 2015]

[14] Documentación Static Flow Pusher API. Disponible en: https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Static+Flow+Pusher+API+pre-v1.0– [último acceso: 9 de Diciembre de 2015]

[15] Cambiar partición SWAP. Disponible en: http://trastetes.blogspot.com.es/2013/11/cambiar-particion-swap-en-ubuntu-1204.html – [último acceso: 9 de Diciembre de 2015]

[16] Repositorios Github Telefónica I+D. Disponible en: https://github.com/telefonicaid?utf8=%E2%9C%93&query=netphony – [último acceso: 9 de Diciembre de 2015]

[17] Repositorios Github Telefónica I+D. Disponible en: https://github.com/ict-strauss/COP – [último acceso: 9 de Diciembre de 2015]

[18] RFC 3209 RSVP-TE: Extensions to RSVP for LSP Tunnels 4.3.3. Subobjects . Disponible en: https://tools.ietf.org/html/rfc3209#section-4.3.3 – [último acceso: 9 de Diciembre de 2015]

[19] Código service-call.yang . Disponible en: https://github.com/ict-strauss/COP/blob/master/yang/yang-cop/service-call.yang – [último acceso: 9 de Diciembre de 2015]

[20] Documentación oficial JETTY. Disponible en: http://www.eclipse.org/jetty/ – [último acceso: 9 de Diciembre de 2015]

[21] A. Mayoral et al., First experimental demonstration of distributed cloud and heterogeneous network orchestration with a common Transport API for E2E services with QoS, OFC 2016

[22] OIF-ONF white paper, Global Transport SDN Prototype Demonstration, 2014.

[23] R. Vilalta et al., The Need for a COP in Research Projects on Optical Networking, EuCnC, 2015.

[24] A. Aguado et al., Dynamic Virtual Net. Reconfiguration over SDN Orchestrated Multi-Technology Optical Transport Domains, ECOC 2015

[25] Y. Yoshida et al., SDN-based Network Orch. of Variable-capacity OPS Network over Programmable Flexi-grid Elastic Optical Path Network, JLT, vol. 33, n. 3, 2015.