Universidad de la República
Facultad de Ingeniería

Overlay Network Routing Application: ONRApp

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Ignacio Brugnoli, Martín Fernández, Diego Mazzuco

en cumplimiento parcial de los requerimientos para la obtención del título de Ingeniero Electricista.

Tutor

Gabriel Gómez ..... Universidad de la República

Tribunal

Eduardo Cota ..... Universidad de la República
Eduardo Grampín ..... Universidad de la República
Alvaro Valdés ..... Universidad de la República

Montevideo
lunes 22 julio, 2019

Overlay Network Routing Application:

ONRApp, Ignacio Brugnoli, Martín Fernández, Diego Mazzuco.

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 167 páginas.
Compilada el lunes 22 julio, 2019.
http://iie.fing.edu.uy/

Agradecimientos

Queremos agradecer principalmente a nuestras familias por darnos día a día el soporte moral y emocional que necesitamos para afrontar retos complejos que nos exigen rendir al máximo. En particular, este trabajo no podría haber sido realizado de no ser por ellos.

Agradecemos a todos nuestros compañeros, amigos y familiares que ayudaron a construir el documento aquí presente.

Esta página ha sido intencionalmente dejada en blanco.

Resumen

Las reglas de enrutamiento de tráfico impuestas por los proveedores de servicio de Internet pueden establecer canales de comunicación cuyas características sean sub óptimas desde el punto de vista de la calidad de servicio (QoS de sus siglas en inglés). Por esta razón surgen las redes sobrepuestas a Internet que posibilitan definir reglas de encaminamiento distintas a las preestablecidas, permitiendo posibles mejoras en la calidad de servicio de forma independiente a los proveedores subyacentes.

Este trabajo propone un algoritmo de identificación y encaminamiento de trafico TCP/UDP basado en la utilización de identificadores instalados en los puertos de capa de transporte, permitiendo que la MTU de los paquetes pertenecientes al tráfico tratado no se vea afectada y por tanto, brindando la posibilidad de ser implementado en un ambiente multidominio como lo es Internet. El algoritmo diseñado se basa en el concepto de identificadores locales utilizado en el protocolo *MPLS* y la técnica de traducción de direcciones *NAT*.

Si bien el algoritmo presenta ventajas teóricas respecto a métodos tradicionales, en la práctica y bajo el paradigma actual de las redes de datos, la administración distribuida de redes sobrepuestas a Internet posee limitaciones importantes debido al dinamismo de esta red y a la complejidad que genera la toma de decisiones de forma distribuida.

Para afrontar la dificultad de administración, en este proyecto se hace uso de las ideas de separación de los planos de control y de datos planteadas por el paradigma de las redes definidas por software (*SDN* de sus siglas en inglés) en conjunto con *OpenFlow*, el protocolo con mayor adopción en la actualidad dentro del paradigma para la comunicación entre ambos planos para implementar el algoritmo diseñado.

En este proyecto se define una arquitectura de red sobrepuesta a Internet compuesta por un sistema de medición de calidad de servicio y encaminamiento de datos distribuido en cada nodo de la red, controlado de forma centralizada.

Se desarrolla una aplicación sobre un controlador *SDN* la cual brinda servicios de red que, a partir de políticas impuestas externamente enruta el tráfico implementando el algoritmo de encaminamiento diseñado y ejecuta mediciones de QoS en una red sobrepuesta. Esta aplicación es diseñada de forma tal que el sistema sea tolerante a fallas de ruteo.

Se realizan pruebas de funcionalidad y rendimiento de la aplicación en un ambiente de emulación como *Mininet*, validando el sistema de medición y logrando tiempos de implementación de políticas en la red del orden de milisegundos.

Esta página ha sido intencionalmente dejada en blanco.

Tabla de contenidos

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 1

Introducción al problema

En el presente capítulo capítulo se introducen los principales problemas que el proyecto pretende solucionar y los objetivos concretos que se plantean como criterios de éxito.

1.1. Descripción

En la actualidad, el consumo de servicios brindados desde la nube ha crecido de forma exponencial. En particular, las redes de distribución de contenido a demanda exigen un alto nivel de calidad de servicio (Quality of Service, QoS). Ejemplo de esto son los servicios *Live Streaming*, donde la calidad de la experiencia (Quality of Experience, QoE) del usuario es altamente sensible a parámetros que determinan la QoS, como el *Delay*, *Jitter* o pérdida de paquetes.

Los nuevos servicios brindados en la nube, como *IaaS* (Infraestructure as a Service), *SaaS* (Software as a Service) o *PaaS* (Platform as a Service), pueden requerir transferencias de datos entre *data centers* distribuidos geográficamente, interconectados a través de Internet. En este caso, un parámetro relevante en la QoS es el ancho de banda disponible para transferencia de grandes volúmenes de información.

En Internet los flujos de datos se enrutan de acuerdo a las políticas del protocolo *Border Gateway Protocol* (BGP). Sin embargo, los caminos generados por este protocolo pueden no ser óptimos desde el punto de vista de la QoS.

Consideremos tres puntos dispersos en Internet (A, B, C) como se muestra en la figura 1.1. El hecho anterior implica que el camino de A hasta B, determinado por el conjunto de proveedores de servicios de Internet (Internet Service Providers, ISPs), puede tener una QoS inferior que si consideramos el camino de A hasta C y luego de C hasta B. Debido a esto y a razones de privacidad o estrategias para la distribución de contenido, se han desarrollado desde hace varios años las redes sobrepuestas (Overletworks, ON).

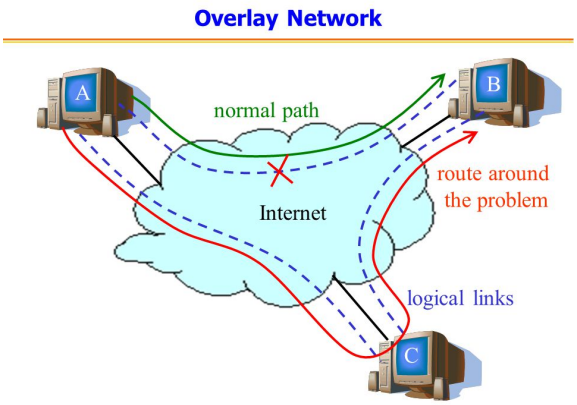


Figura 1.1: Camino alternativo usando una red sobrepuesta. [referencia]

Una *ON* es una red virtual construida lógicamente sobre una red física subyacente. Estas son compuestas por nodos o puntos de presencia (Point of Presence, PoP) dentro de los cuales el administrador tiene dominio total para la toma de decisiones. Las *ON* permiten un control de los flujos de datos sin necesidad de tener acceso a los equipos de los operadores o *ISPs*.

Soluciones clásicas se basan en el concepto de circuitos virtuales, implementados usando tecnologías como ATM o IP/MPLS. Estos mecanismos permiten enrutar tráfico por caminos diferentes a los que indica el enrutamiento basado en la dirección IP de destino, tal como se realiza de forma estándar en Internet. Si bien estas tecnologías son capaces de proveer QoS, aplican únicamente dentro de un mismo dominio. Cuando los flujos deben atravesar distintos dominios, como ocurre usualmente en Internet, las soluciones de QoS se tornan complejas y costosas debido a la intervención de múltiples *ISPs*.

Una posible solución tanto al problema del ambiente multidominio, como a la independencia frente a los *ISPs*, podría ser la utilización de túneles. A pesar de ser una solución válida, este método presenta varias debilidades. Una de ellas es que el encapsulamiento utilizado para construir estos túneles puede generar paquetes cuyo tamaño excede la *MTU* en alguna zona de la red, dando lugar a la fragmentación, lo que genera un mayor tráfico de datos para enviar la misma información original.

Otra debilidad importante de los mecanismos de túneles IP es que el proceso de establecimiento, gestión y mantenimiento de los mismos es una tarea compleja.

Encaminar el tráfico por un camino distinto al determinado por los *ISPs* no es el único problema. Si se supone que existe una solución al problema del encaminamiento, surge la duda: ¿cómo se determina el mejor camino a utilizar? o de forma más especifica, ¿cuál es la QoS de cada uno de los caminos posibles en la *ON*?

En este proyecto, la aproximación a los problemas mencionados se basa en la aplicación del paradigma de redes definidas por software (Software Defined Networks, SDN) para definir una arquitectura que brinde el soporte y permita desarrollar un sistema de encaminamiento de datos y medición de QoS sobre una *ON*, independiente de la colaboración de los *ISPs* por donde pasa el tráfico entre los nodos de la *ON* y que prometa autonomía, dinamismo y escalabilidad gracias a su estructura de control centralizado.

A continuación, se especifican los objetivos puntuales que tiene este proyecto frente a los problemas planteados.

1.2. Objetivos del proyecto

Como solución a los problemas mencionados, se plantean los siguientes objetivos:

- Diseño de la arquitectura de red:
  Tras investigar las distintas soluciones de encaminamiento posibles, definir una arquitectura basada en el paradigma *SDN* que cumpla con los requerimientos previamente mencionados, implementar dicha arquitectura y probarla a nivel de prototipo. Se toma como punto de partida la arquitectura presentada en trabajos relacionados. [1, 2]
- Algoritmo de encaminamiento sobre una *ON*:
  Diseñar un algoritmo capaz de encaminar el tráfico en una *ON* sobre Internet utilizando el paradigma *SDN*. Este algoritmo debe evitar los problemas que genera el encapsulamiento de tráfico al crear caminos virtuales, teniendo en consideración aspectos como la escalabilidad de la *ON* respecto a puntos de presencia e identificación de flujos entre los mismos, así como también el ambiente multidominio en el que se trabaja.
- Metodología de medición de QoS sobre caminos de una *ON*:
  En base al algoritmo de encaminamiento creado, diseñar un método de medición de caminos en una *ON* que permita la aplicación de técnicas de medición de QoS estandarizadas.
- Elección del controlador:
  Se investigará sobre controladores *SDN* disponibles en el mercado, así como estudios previos que hayan sido realizados sobre los mismos y en base a esto, se seleccionará uno para ser utilizado en el diseño de la aplicación.
- Diseño de aplicación de encaminamiento y medición de QoS:
  Diseñar una aplicación capaz de utilizar los servicios de red del controlador para encaminar el tráfico acorde a las decisiones tomadas en una aplicación externa que realizará ingeniería de tráfico. A su vez, esta aplicación debe brindar las herramientas necesarias para evaluar la QoS de las rutas definidas sobre la *ON*.
- Protocolo de pruebas de funcionalidad y rendimiento:
  Se realizará un protocolo de pruebas que evalúe la correcta funcionalidad y rendimiento de la aplicación.

El resto del documento se estructura como sigue: en el capítulo 2 se presenta el paradigma de redes definidas por software, junto con el protocolo de comunicación entre plano de datos y plano de control con mayor adopción en la actualidad.

En el capítulo 3 se presenta el sistema diseñado en este proyecto, desde su arquitectura hasta los algoritmos de encaminamiento y medición de QoS.

En el capítulo 4 se presenta el controlador elegido para realizar el desarrollo de las aplicaciones, su estructura interna y la forma de interacción de este con el plano de datos y el plano de control definidos en el capítulo 2.

En el capítulo 5 se exhibe *ONRApp*, la aplicación implementada para llevar a cabo el encaminamiento sobre una *ON*, mostrando su arquitectura de software, bloques que la componen y funcionalidades que brinda hacia capa de aplicación.

En el capítulo 6 se analiza el *PPG*, software destinado a la medición de QoS a partir de instrucciones de *ONRApp*. Allí se presenta la arquitectura de software utilizada, funcionalidades y forma de comunicación con *ONRApp* y otros *PPGs*

En el capítulo 7 se presenta como se configuran los *Switches OpenFlow* presentados en el capítulo 2 para lograr encaminar el tráfico en el plano de datos, utilizando las reglas presentadas en el capítulo 3.

En el capítulo 8 se presentan los resultados de las pruebas de concepto de los algoritmos desarrollados en el capítulo 3, así como los resultados de las pruebas de validación del sistema diseñado e implementado expuesto en los capítulos 5, 6 y 7. A su vez, se presentan resultados cuantitativos de la evaluación de rendimiento de *ONRApp*

Finalmente, en el capítulo 9 se realizan las conclusiones del proyecto y se presentan los posibles trabajos a futuro.

Capítulo 2

Redes definidas por Software

El objetivo de este capítulo es dar al lector conocimientos teóricos sobre las redes definidas por software (o por sus siglas en inglés *SDN*) y brindar una versión resumida de *OpenFlow*, el protocolo de comunicación con mayor adopción para la comunicación con los dispositivos de red dentro del paradigma *SDN*.

El capítulo se estructura de la siguiente manera: en la sección 2.1 se presentan las principales motivaciones que hacen necesaria la aparición del paradigma *SDN*. En la sección 2.2 se brinda una breve descripción de los antecedentes de *SDN*, así como también una descripción del principio fundamental detrás del paradigma y su arquitectura. Finalmente, en la sección 2.3 se realiza un resumen de las principales funcionalidades que se agregan en cada versión de *OpenFlow*, el protocolo de comunicación entre plano de datos y plano de control con mayor adopción en la actualidad.

2.1. Motivación

Los exigentes requerimientos de QoS en redes actuales y la necesidad de brindar servicios de valor agregado, han implicado la necesidad de un cambio de paradigma para afrontar nuevos retos. En concreto, alguno de los principales problemas existentes son:

- Cambios poco predecibles en los patrones de tráfico:
  En la actualidad es normal que aplicaciones necesiten acceder a bases de datos y servidores geográficamente distribuidos. Esto exige la existencia de un manejo flexible del tráfico y la posibilidad de acceder a demanda a diferentes niveles de ancho de banda, es decir, de forma variable y poco predecible.
- Surgimiento de nuevas tecnologías:
  Pequeñas y medianas empresas tienden a migrar sus servicios hacia la nube. Esto implica que servicios de procesamiento de datos, almacenamiento, *streaming* y otros, se procesen cada vez más en servidores remotos.
  Este hecho se ve acompañado del concepto *Big Data*, que implica un aumento considerable en los anchos de banda necesarios debido a la necesidad de

mayor capacidad de procesamiento paralelo y distribuido.

Las principales problemáticas que surgen cuando se desea solucionar estos temas utilizando los paradigmas de redes actuales son:

- Complejidad generalizada:
  Modificar físicamente la red o modificar políticas de toda la red es complejo. Esto consume mucho tiempo y termina siendo manual, lo cual implica un alto riesgo de errores y, por tanto, estos cambios necesarios se terminan estancando.
- Dependencia con los vendedores:
  Las capacidades del hardware de dispositivos de red y la forma de administración de los mismos no se encuentra estandarizada. Esto implica que, una vez que se implementa una solución con un fabricante particular o un conjunto de fabricantes particulares, esta asociación no suele ser modificada. Esto se traduce en que estos fabricantes pautan la velocidad de avance de la tecnología.

Como respuesta a todos estos problemas surge la idea de separar el plano de control del plano de datos, en donde *SDN* aparece como uno de los grandes actores.

2.2. Paradigma *SDN*

2.2.1. Antecedentes

Los principios de SDN se observaron por primera vez en la red telefónica digital en la cual se desacopla el plano de control del plano de transporte de datos para simplificar la administración de la misma.

En el 2004, la "Internet Engineering Task Force (IETF)" consideró distintas formas de separar ambos planos en las redes de datos, proponiendo un estandar para la interfaz entre el plano de datos y el de control, en una publicación titulada "Forwarding and Control Element Separation (ForCES)". [3]

Por otra parte, la utilización de Software *Open-Source* en el paradigma de separación de planos de datos y control surge con el proyecto *Ethane* creado por el departamento de ciencias de la computación de la Universidad de Stanford. Este proyecto resultó en la creación del protocolo de comunicación *OpenFlow* en el 2008. En la actualidad, este resulta ser el protocolo con mayor adopción para la interacción entre el plano de datos y el plano de control dentro del paradigma *SDN*.

En el 2011, Deutsche Telekom, Facebook, Google, Microsoft, Verizon y Yahoo! crearon la "Open Networking Foundation (ONF)", una organización sin fines de lucro dedicada a la promoción y adopción de *SDN* a través del desarrollo de estándares abiertos, basados en el trabajo previo de investigadores de las universidades de Stanford y Berkeley. Además de apoyar a la estandarización del protocolo *OpenFlow* como interacción entre los planos de control y de datos, esta organización forma parte del desarrollo de otras piezas dentro del paradigma, como es el caso del controlador *ONOS*. [4]

2.2.2. Principio fundamental

La idea fundamental detrás de este paradigma es separar el plano de control del plano de datos en los dispositivos físicos (routers, switches, etc), desplazando el control de la red a elementos particulares: los controladores.

En lugar de que todos los dispositivos participen en la toma de decisiones (decisiones distribuidas), el controlador es el único elemento encargado de construir las tablas de encaminamiento de los dispositivos de red. Inteligencia centralizada implica que este elemento posee una visión completa de la red, lo cual brinda beneficios para la toma de decisiones. Esto implica una simplificación a la hora de configurar funciones de red (NAT, Firewalls, etc), ya que es posible realizarlas desde un único punto con efecto en toda la red.

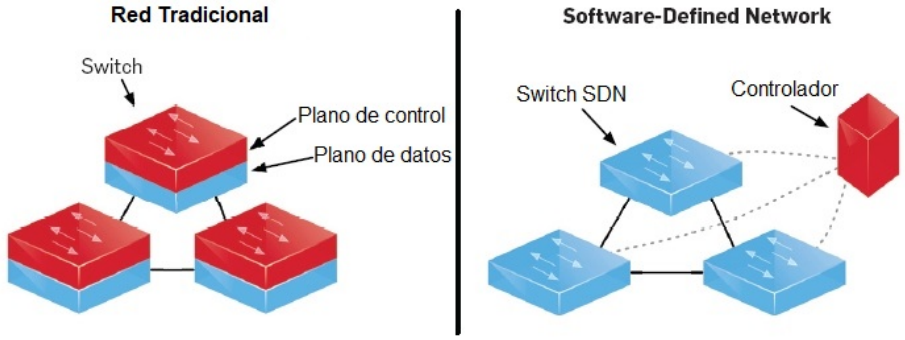


Figura 2.1: Separación de planos de control y datos en el paradigma *SDN*. Figura obtenida de [5]

Concretamente, la ONF define a las *SDN* como:

> *" Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications.*
> *This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.*
> *The OpenFlow protocol is a foundational element for building SDN solutions. "*
>
> ONF [6]

Resumiendo, las características que definen a las *SDN* son:

- Programable
  El control de la red es programable de forma simple ya que se encuentra separado de las funciones de encaminamiento de datos.

- Ágil
  La centralización del control de la red permite ajustar el flujo de tráfico de forma dinámica, minimizando los tiempos de respuesta.

- Administración centralizada
  El hecho de que la inteligencia se encuentre centralizada también implica la posibilidad de una visión global de la red. Desde el punto de vista de las aplicaciones y usuarios que generan políticas de alto nivel, la red aparenta ser un único punto lógico.

- Permite automatización
  El control se encuentra centralizado y por tanto, los administradores pueden generar programas que se ejecuten en el controlador, de forma de automatizar las políticas deseadas, permitiendo mayor dinamismo en las configuraciones.

- Simplifica los elementos de infraestructura (*Switches SDN*)
  La centralización de las funciones de control de red implica que, en el paradigma *SDN*, los dispositivos de infraestructura son menos complejos y posean mayor especialización en el *forwarding*. Esto se traduce en un menor costo de los dispositivos y una posible mejora de rendimiento.

- Orientada a estándares libres y neutral frente a fabricantes
  Uno de los mayores problemas de las redes actuales a nivel de administración y mantenimiento es la configuración de los elementos de red, ya que tiende a ser particular para cada fabricante debido a la no estandarización de la forma de administración. Esto implica el conocimiento de diferentes comandos para obtener mismos resultados en distintos equipos.
  En *SDN*, la independencia frente al fabricante implica que todos los dispositivos se puedan configurar de la misma manera, facilitando en gran medida su gestión.

2.2.3. Arquitectura

La mayor diferencia que tiene *SDN* respecto a las redes actuales es el cambio en la arquitectura que plantea. La ONF define esta arquitectura como aquella que, desde el punto de vista más simple, se puede dividir en tres planos principales: el plano de aplicación, el plano de control y el plano de datos o infraestructura. [7]

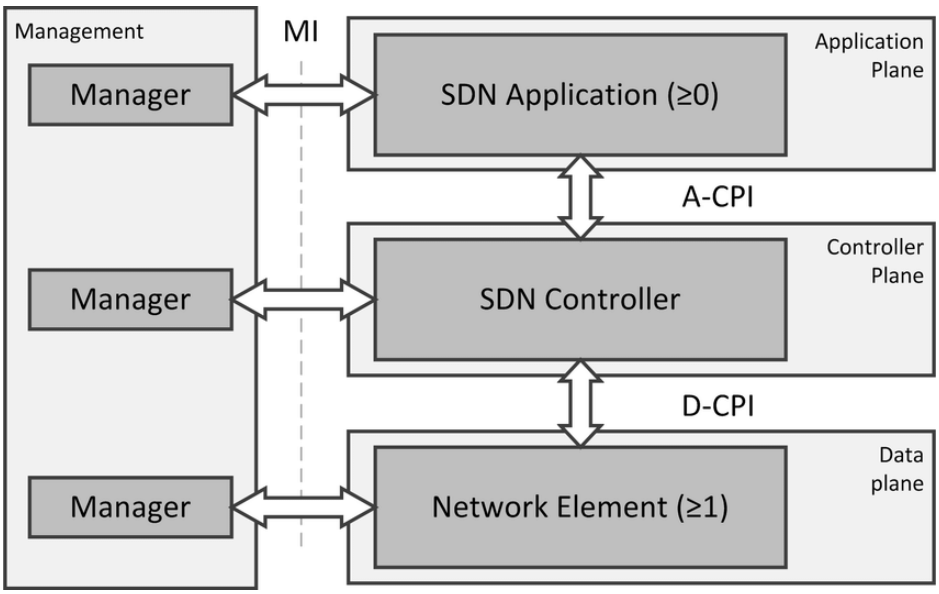


Figura 2.2: Planos del paradigma *SDN*. Imagen extraída de [7]

Plano de datos o infraestructura

Es el plano inferior, compuesto por los elementos de red encargados de procesar y encaminar el tráfico dentro de ella. Estos elementos de red son interpretados como objetos lógicos por el plano superior, lo que permite considerar elementos físicos o virtualizados indistintamente como parte de este plano.

Plano de Control

En el núcleo de la arquitectura encontramos el controlador, siendo este el elemento con mayor importancia de toda la arquitectura, ya que es el elemento mediador entre el plano de aplicación y el plano de infraestructura.

Entre cada borde de planos existe una *API*, cierto protocolo de comunicación que permite la interacción entre los planos. A la interfaz Controlador-Aplicación, se le asocia el nombre de interfaz norte (*NorthBound Interface* o *NBI*), mientras que a la interfaz Controlador-Infraestructura, se le asocia el nombre de interfaz sur (*SouthBound Interface* o *SBI*).

El controlador se encuentra encargado de dos principales tareas:

- Traducir los requerimientos de las aplicaciones que soporta hacia el plano de infraestructura.
- Proveer a las aplicaciones una visión global y abstracta de la red, que puede incluir estadísticas sobre la misma, así como eventos que ocurran en ella (pérdida de un *link*, etc)

Es importante observar que en la definición no se especifica (no se obliga a poseer ni se obliga a prescindir) la presencia de más de un controlador que puedan servir como respaldo o réplica, o la existencia de un orden jerárquico entre controladores.

Plano de aplicación

El plano de aplicación en *SDN* contiene todas las aplicaciones que van a hacer uso de los recursos que provee el controlador. Concretamente, se define una aplicación *SDN* como un programa que explícita, directa y automáticamente comunica sus requerimientos de red y el comportamiento de red deseado al controlador *SDN*, a través de la *NBI*. A su vez, pueden consultar el estado de la red para tomar decisiones.

Plano de Administración

Hasta este punto, se ha considerado una arquitectura de tres planos para *SDN*. Sin embargo, además de definir esta arquitectura de tres planos, se define una arquitectura más profunda: a los tres planos anteriores se agrega un plano vertical que interactúa con cada uno de los otros, el plano de administración. [7]

La razón de que en un modelo más profundo de la arquitectura aparezca este cuarto plano, proviene del hecho que ciertas funciones administrativas resultan imprescindibles para el funcionamiento de una red *SDN*:

- En el plano de datos, se necesita de la función administrativa al menos en la configuración inicial de los elementos de red, asignando interfaces, puertos, elementos a ser controlados y asignaciones de controladores.
- En el plano de control, las funciones administrativas deben configurar políticas que indiquen los permisos de control y monitoreo del sistema que pueden tener las aplicaciones que utilizan del mismo.
- En el plano de aplicación, el plano de administración típicamente configura los *Service-Level Agreements (SLAs)*. [8]

2.3. *OpenFlow*

Tengamos en cuenta la arquitectura de las *SDN* presentada en la sección 2.2. Una pieza clave que resulta necesario definir es el comportamiento de las entidades que componen el plano de datos y como estas interactúan con el controlador (*SBI*). Es aquí donde aparece *OpenFlow* (*OF*), el protocolo de comunicación con mayor adopción en la actualidad destinado a la programación del plano de datos.

En esta sección se presenta la estructura interna de los dispositivos de red que implementan el protocolo *OpenFlow*, los cuales se definen como *Switches OpenFlow*. La sección continúa describiendo el protocolo *OpenFlow*, resaltando las características más relevantes del protocolo desde la versión 1.0 hasta la 1.5.

Todo el contenido presentado en esta sección es un resumen de la especificación del protocolo. En particular se hizo uso de las especificaciones 1.0 y 1.5, ya que estas contienen todos los coneceptos de las versiones intermedias. [9, 10]

En esta sección se hace uso de los anglicismos ***match*** *y* ***matching****. refiriendo a la acción de encontrar coincidencias entre el cabezal de un paquete y valores preestablecidos.*

2.3.1. *Switch OpenFlow*

La especificación del protocolo *OpenFlow* define a grandes rasgos la estructura y el comportamiento de un dispositivo de red que lo implemente. Un dispositivo que implementa *OpenFlow* es comúnmente llamado *Switch OpenFlow*. Un esquema de su estructura se muestra en la figura 2.3.

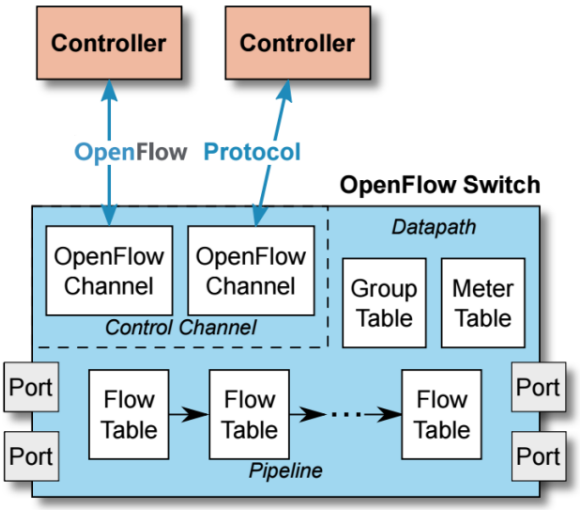


Figura 2.3: *Switch OpenFlow* v1.5. Figura extraída de [10]

Las diferentes partes que se muestran en la figura 2.3 serán descriptas en la siguiente sección, en donde se presentan las diferentes versiones del protocolo.

2.3.2. Versiones del protocolo

El fin de este protocolo es reprogramar la lógica de los switches con el objetivo de encaminar flujos de datos según un criterio predefinido. Con el fin de brindar un mayor entendimiento de este protocolo, se describirán las diferentes versiones de *OpenFlow* existentes hasta la fecha, presentando las características más relevantes añadidas en cada una. Se debe tener en cuenta que la versión mínima necesaria para ejecutar la aplicación realizada es la 1.3 debido a las funcionalidades que se agregan hasta esta versión, necesarias para la implementación descripta en el capítulo 7.

2.3.2.a. *OpenFlow* v1.0

Puerto *OpenFlow*

Se define un puerto *OpenFlow* como un puerto físico al que se le asocian distintas colas de forma de clasificar los paquetes según un criterio de prioridad. En particular, en esta versión el único método para controlar la QoS es a través del manejo de estas colas.

Flow tables y *flow entry*

Las *flow tables* y *flow entries* son la base de la lógica de encaminamiento de los datos del *Switch OpenFlow*. Las *flow tables* son tablas compuestas por *flow entries*. Estas modifican, encaminan o enrutan paquetes entrantes según las coincidencia entre el paquete que ingresa al switch y alguna de las *flow entries* configuradas en el mismo.

En esta versión sólo se contempla la existencia de una única *flow table*. Una *flow entry* está compuesta por los siguientes tipos de campos:

- *Header fields*
  Utilizados para realizar el *matching* con un paquete de datos. Describe ciertas características que debe cumplir un paquete para ser procesado.
- *Counters*
  Son utilizados para llevar estadísticas relacionadas con las coincidencias con la *flow entry* configuradas en el switch. De esta forma, es posible transferir información al controlador para que esta sea accesible desde las aplicaciones.
- *Actions*
  Acciones a ejecutar sobre los paquetes que hacen *match* con esta *flow entry*.

En esta versión del protocolo, el *maching* se puede realizar utilizando un subconjunto de los siguientes campos:

- Puerto de entrada
- VLAN ID
- Prioridad de VLAN
- Dirección Ethernet de origen
- Dirección Ethernet de destino
- Tipo de trama Ethernet
- Dirección IP origen
- Dirección de IP destino

- Versión de protocolo IP
- Bits de Tipo de Servicio (Type of Service, ToS) de IP
- Puerto TCP/UDP origen
- Puerto TCP/UDP destino

Es importante comprender que en el proceso de *matching* no es necesario especificar todos los campos del encabezado, si no que se pueden omitir de forma de aumentar la flexibilidad.

En las especificaciones *OpenFlow*, la ONF define como debe comportarse un switch, pero no define la implementación específica. En contraste, define tres niveles de conformidad. Todo switch que soporte *OpenFlow* 1.0 debe cumplir una de las tres categorías:

- Completa
  Debe soportar los 12 campos para el *matching*
- Conformidad de capa 2
  Debe soportar únicamente el *maching* con cabezales de capa de enlace de datos (capa 2)
- Conformidad de capa 3
  Debe soportar únicamente el *maching* con cabezales de capa de red (capa 3)

En esta versión, si no existe ningúna *flow entry* contra la cual se haya realizado *maching*, entonces el paquete es encaminado hacia el controlador para que este último lo procese y decida la acción a realizar. Cuando existe un *match*, cada *flow entry* tiene asociado un conjunto de acciones a realizar sobre el paquete.

En esta versión, se definen las siguientes acciones:

- *Forwarding*
  El switch debe soportar el *forwarding* a puertos físicos y los siguientes puertos virtuales:
  - CONTROLLER
  - ALL
  - LOCAL
  - TABLE
  - IN_PORT
  - NORMAL
  - FLOOD
- *Drop*
  Indica que los paquetes deben ser descartados.
- *Queue*
  Encola el envío de un paquete a un puerto específico.
- *Modify header*
  Algunos campos del encabezado que pueden ser modificados en esta versión son ciertos cabezales Ethernet, VLAN, IPv4 y campos de TTL.

Mensajes entre switch y controlador

Los mensajes entre el switch y el controlador transitan en un canal seguro. Este canal es implementado vía *Transport Layer Security* (TLS) sobre TCP. El switch debe conocer la IP del controlador para poder establecer la conexión con el mismo y de esta forma estar bajo su dominio.

Cada mensaje entre un switch y el controlador mediante el protocolo *OF* comienza con un *header* que contiene información de la versión de *OpenFlow*, tipo del mensaje, largo del mensaje, ID de la transacción, entre otros.

A grandes rasgos, existen tres tipos de mensajes entre switch y controlador [4]:

- Simétrico:
  Estos mensajes pueden ser iniciados desde el controlador o desde el switch.
- Controlador a switch:
  Este tipo de mensajes se generan en el controlador y permiten configurar y administrar las *flow entries* de un switch, así como generar nuevos paquetes a ser transmitidos por una interfaz específica.
- Asíncrono:
  Estos mensajes son enviados por el switch sin solicitud del controlador, por ejemplo, a causa de un evento generado en la red. Por ejemplo, estos mensajes son enviados al controlador cuando no se encuentra un *match* en la *flow table* para un paquete entrante.

Message Type	Category	Subcategory
HELLO	Symmetric	Immutable
ECHO_REQUEST	Symmetric	Immutable
ECHO_REPLY	Symmetric	Immutable
VENDOR	Symmetric	Immutable
FEATURES_REQUEST	Controller-switch	Switch configuration
FEATURES_REPLY	Controller-switch	Switch configuration
GET_CONFIG_REQUEST	Controller-switch	Switch configuration
GET_CONFIG_REPLY	Controller-switch	Switch configuration
SET_CONFIG	Controller-switch	Switch configuration
PACKET_IN	Async	NA
FLOW_REMOVED	Async	NA
PORT_STATUS	Async	NA
ERROR	Async	NA
PACKET_OUT	Controller-switch	Cmd from controller
FLOW_MOD	Controller-switch	Cmd from controller
PORT_MOD	Controller-switch	Cmd from controller
STATS_REQUEST	Controller-switch	Statistics
STATS_REPLY	Controller-switch	Statistics
BARRIER_REQUEST	Controller-switch	Barrier
BARRIER_REPLY	Controller-switch	Barrier
QUEUE_GET_CONFIG_REQUEST	Controller-switch	Queue configuration
QUEUE_GET_CONFIG_REPLY	Controller-switch	Queue configuration

Figura 2.4: Tipos de mensajes en *OpenFlow* v1.0. Figura extraída de [4]

Algunos de los tipos de mensajes que se muestran en la comunicación son:

- HELLO: Es un tipo de mensaje simétrico generado por el *Switch OpenFlow* para iniciar una nueva conexión.
- PACKET_IN: Es un tipo de mensaje asíncrono generado por el switch para informar al controlador de cierto evento del plano de datos.

- PACKET_OUT: Este tipo de mensajes es enviado desde el controlador a un switch, comúnmente como respuesta a un PACKET_IN.
- FLOW_MOD: Utilizado para cambiar el estado de las tablas del *Switch OpenFlow*.

En la figura 2.5 se muestran las diferentes etapas de una comunicación típica entre un controlador y un switch.

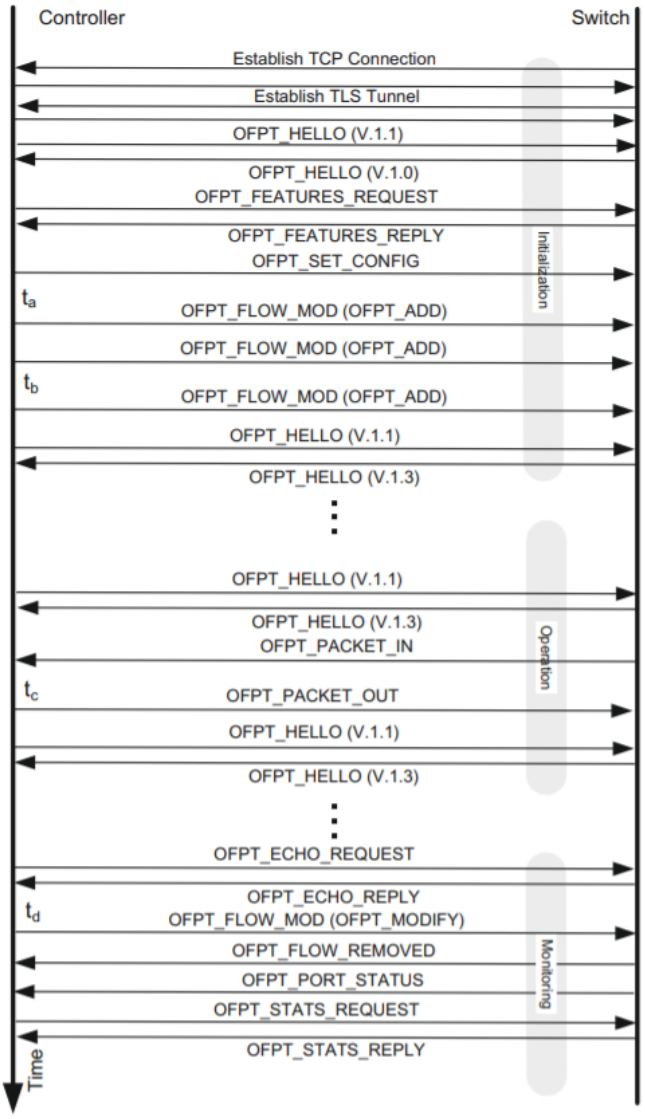


Figura 2.5: Etapas de comunicación en *OpenFlow* v1.0. Figura extraída de [4]

2.3.2.b. *OpenFlow* v1.1

La adición más relevante en esta versión del protocolo es la existencia de múltiples *flow tables*. Las diferentes *flow tables* pueden ser identificadas unívocamente a través de un ID asociado a cada una. Aquí se introduce un nuevo concepto: el *pipeline*.

Pipeline es el conjunto de *flow tables* ordenadas por ID, por las que un paquete debe atravesar al ser procesado, comenzando por la *flow table* cuyo ID es 0. Existe un tipo de acción creada en esta versión que permite enviar el paquete a una *flow table* con un ID mayor al de la *flow table* actual.

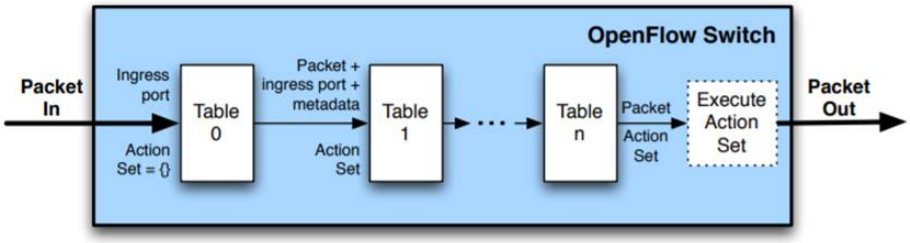


Figura 2.6: Pipeline en *OpenFlow* v1.1. Figura extraída de [9]

Instructions

Cada *flow entry* contiene una serie de *instructions* que son ejecutadas cuando se realiza el *matching* de un paquete. Estas definen las acciones a realizar en el momento de *matching* (acciones dentro de un *Action Set*) y las acciones a realizar al finalizar el *pipeline* (acciones dentro de un *Set of Actions*). Además, las *instructions* permiten controlar cómo será el flujo de datos entre las *flow tables* mediante la *instruction Go-to table* y qué información se transfiere entre estas. Los tipos de *instructions* posibles son:

- *Apply-Actions*
  Son acciones a ser realizadas sobre el paquete de forma inmediata. Esta *instruction* puede ser utilizada para modificar el paquete entre dos *flow tables*. Las acciones se especifican como una lista o *Actions Set*.
- *Write-Actions*
  Escribe una acción en el *Set of Actions* actual. Si existe una acción del tipo dado en el conjunto actual, se sobre escribe, de lo contrario la agrega.
- *Clear-Actions*
  Borra todas las acciones en el *Set of Actions* de forma inmediata.
- *Write-Metadata*
  El *metadata* es una variable de 64 bits que permite transferir información entre *flow tables*. La acción escribe el valor deseado en el campo *metadata*.
- *Goto-Table*
  Indica la siguiente tabla en el *pipeline*. El ID de tabla debe ser mayor que el ID de tabla actual.

En cada *flow entry* no puede existir dos o más instrucciones del mismo tipo.

Set of Actions

La implementación de varias *flow tables* produjo la aparición del *Set of Actions*. Estas son un conjunto de acciones asociadas a un paquete, que se ejecutan cuando el mismo llega al final del *pipeline*. Esto ocurre cuando el paquete realiza el *maching* con una *flow entry* que no contiene la *instruction Go-to table*.

Una *flow entry* puede modificar el *Set of Actions* usando una *instruction* del tipo *Write-Actions* o *Clear-Actions*. Al comenzar el *pipeline*, este conjunto se encuentra vacío y se conserva al cambiar de *flow table*.

Las acciones dentro de un *Set of Actions* son aplicadas en orden, dependiendo del tipo de acción, independientemente del orden en el cual fueron agregadas al *Set of Actions*.

Virtual ports

Si bien en la versión anterior ya existían algunos puertos virtuales reservados, todos los puertos de salida se encuentran asociados a puertos físicos. En esta versión se agrega la posibilidad de *switch-defined virtual ports*, mediante el cual se agregan puertos de salida virtuales con los que se puede llevar a cabo acciones más complejas tal como túneles, o utilizarlos como *Link Agregation (LAG)*, etc.

2.3.2.c. *OpenFlow* v1.2

Múltiples controladores

Versiones anteriores no soportan múltiples controladores. Esto implica que, en caso de pérdida de comunicación de un switch con el controlador, el switch únicamente puede operar en dos modos: *secure* o *standalone*. El primero consiste en continuar con el funcionamiento normal, descartando los paquetes que deben ir al controlador. En el segundo, el switch se comporta como switch *legacy*. Sin embargo, ambos implican un funcionamiento sin control centralizado.

A partir de esta versión, el switch puede configurarse para mantener conexiones simultáneas a múltiples controladores. Esto permite que, en caso de pérdida de conexión con uno, continúe el control centralizado con otro. El switch debe llevar el registro de cuáles mensajes corresponden a cada controlador, duplicando los mismos si es necesario enviarlo a más de uno.

Para que los controladores no entren en conflicto se subdividen entre 3 roles diferentes respecto a cada switch: *master*, *equal* y *slave*. En rol *slave* el controlador solo puede pedir datos al switch, no pudiendo realizar modificaciones en este. En *equal* y *master* permiten al controlador programar el switch. En caso que un controlador pase a *master*, el antiguo *master* asumirá el rol de *slave*, mientras que el modo *equal* permite varios controladores del mismo tipo.

2.3.2.d. *OpenFlow* v1.3

Soporte más flexible para *table miss*

En versiones anteriores, cuando un paquete no realiza *matching* con alguna *flow entry*, el switch puede realizar 3 funciones:

- Descartar el paquete
- Enviar el paquete encapsulado en un *PACKET_IN* al controlador
- Enviar el paquete a la siguiente *flow table*

En esta versión, a cada *flow table* se agrega una *flow entry* llamada *table miss flow entry*. Esta *flow entry* tiene la particularidad de tener prioridad 0 y todos los *match* en *wildcard*. Esto implica que, luego de que un paquete llegue a una *flow table* y no encuentre ningún *match*, este siempre realizará *matching* con la *table miss flow entry*.

Cookie en el paquete *PACKET_IN*

La *cookie* es una variable de 64 bits asociada a la *flow entry* que es asignada por el controlador y generalmente se usa para realizar tareas de identificación. En versiones anteriores, cuando un controlador recibe un *PACKET_IN*, este realiza el *matching*. Sin embargo, este *matching* ya fue realizado en el switch y por tanto, esta acción es repetida.

A medida que la comercialización de *OpenFlow* aumenta, mejorar la *performance* se vuelve un aspecto importante a tener en cuenta. En esta versión se incluye la *cookie* en los *PACKET_IN* lo cual permite que, una vez que el controlador recibe por primera vez la *cookie*, el *matching* completo del paquete no se vuelve a realizar. Esto se traduce en una mejora en la *performance* respecto al tiempo de procesamiento.

2.3.2.e. *OpenFlow* v1.4

Puertos ópticos

A partir de esta versión, *OpenFlow* soporta puertos ópticos. Se incluyen campos para configurar y monitorear la frecuencia de transmisión y recepción de los láseres, así como también su potencia.

Bundle

Se agregan objetos llamados *Bundle*. Estos son una secuencia de solicitudes de modificación de *flow entries* realizadas por el controlador, que se aplica como una única operación en un switch. Si las modificaciones en el paquete tienen éxito, todas las modificaciones se conservan, pero si surge algún error en el switch durante alguna de las modificaciones, ninguna es efectuada.

2.3.2.f. *OpenFlow* v1.5

Egress Tables

En versiones anteriores, el procesamiento en el *pipeline* no permite realizar un análisis según el puerto salida. A partir de esta versión es posible realizar este tipo

de procesamiento, por lo cual el proceso completo del *pipeline* se separa en dos: *ingress process* y *egress process*.

Ingress process sucede cuando un paquete ingresa en el *pipeline* y *egress process* cuando a través de una acción, se determina el puerto de salida de un paquete.

Todas las *flow tables*, con excepción de la *flow table* 0, pueden ser utilizadas como *ingress tables* o *egress tables*, según la configuración existente. A pesar de las similitudes entre ambos procesos, existen algunas diferencias importantes. A modo de ejemplo, las *egress tables* no pueden cambiar el puerto de salida del paquete.

El procesamiento de salida es opcional. En caso de que un switch no lo implemente o que no se encuentren configuradas las *egress tables*, se transmitirá el paquete como en las versiones anteriores, por el puerto de salida.

La figura 2.7 presenta el procesamiento de paquetes en la versión 1.5, en donde se aprecia el *egress process* y las *egress tables*, junto con el *ingress process* ya conocido.

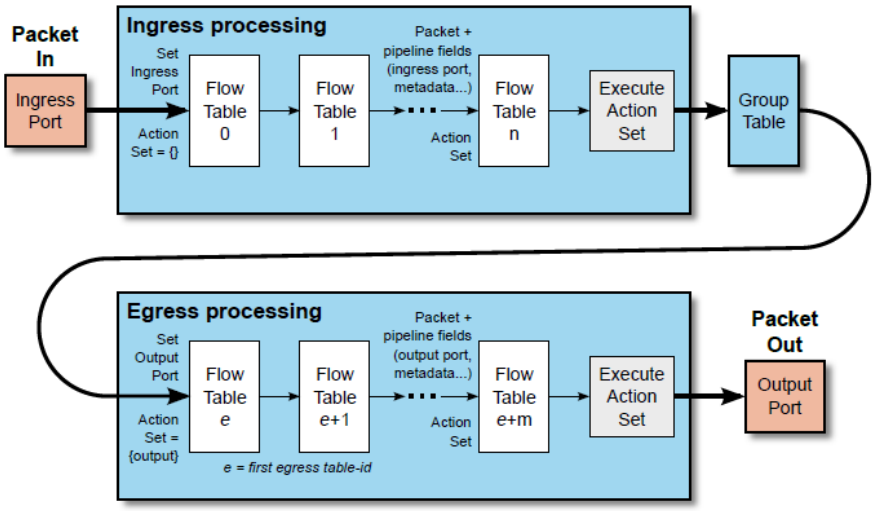


Figura 2.7: *Pipeline* en *OpenFlow* v1.5. Figura extraída de [10]

Capítulo 3

Diseño del sistema

Este capítulo presenta los diseños teóricos realizados durante el proyecto. Aquí se presenta la arquitectura del sistema completo, el algoritmo de identificación y encaminamiento de datos y la metodología de medición de *QoS* de circuitos virtuales sobre una *ON*.

El capítulo se estructura de la siguiente forma: en la sección 3.1 se presenta la arquitectura de sistema, incluyendo la arquitectura de red y su vinculación con las aplicaciones a desarrollar en próximos capítulos. En la sección 3.2 se presenta el proceso de diseño del algoritmo de encaminamiento junto con su especificación y análisis de funcionamiento. Finalmente, en la sección 3.3 se introduce el software de medición *PPG* y su vinculación con el algoritmo antes presentado.

3.1. Arquitectura de sistema

Arquitectura del sistema refiere al conjunto de entidades (físicas o virtuales) que lo componen, así como también a las interacciones entre estas entidades y los requerimientos de diseño necesarios para lograr el funcionamiento esperado. Se destaca que la arquitectura del sistema propuesta en esta sección utiliza como punto de partida propuestas en trabajos relacionados. [1, 2]

El sistema se divide en dos principales conjuntos: el conjunto central, que involucra todas las instancias de controladores y el conjunto distribuido, que involucra todos los *PoP* (*Point of Presence*) conectados a Internet y pertenecientes a la *ON*.

3.1.1. Conjunto central

En el caso particular de este proyecto, el conjunto central se encuentra compuesto por una única instancia de un controlador. El servidor sobre el cual será ejecutado dispondrá de una IPv4 pública, de forma de ser accesible desde cualquier *PoP*.

Con el fin de brindar servicios de valor agregado en la *ON* como el encaminamiento y la medición de QoS, el conjunto central incorpora cuatro aplicaciones:

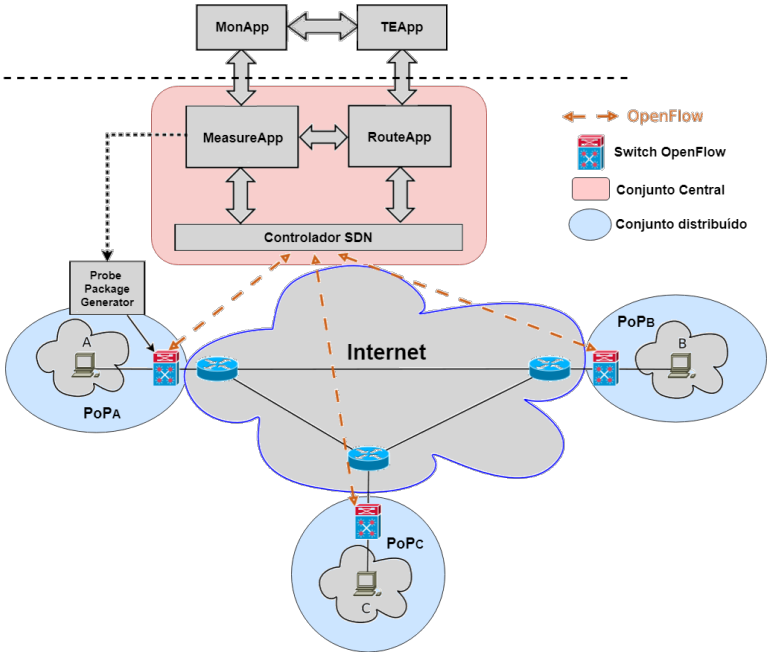


Figura 3.1: Arquitectura general del sistema. Imagen editada a partir de [2]

- Traffic Engineering Application (*TEApp*):
  Esta aplicación es la encargada de, a partir de algoritmos preestablecidos y en base a resultados de medición de QoS, configurar políticas de encaminamiento del tráfico entre *PoPs* de la *ON*.
- Monitoring Application (*MonApp*):
  Esta aplicación es la encargada de, a partir de algoritmos preestablecidos, solicitar mediciones de QoS de los posibles caminos sobre la *ON*.
- Routing Application (*RouteApp*):
  Esta aplicación se encarga de gestionar los flujos de datos, permitiendo la configuración de politicas de encaminamiento sobre la ON, utilizando las funcionalidades que brinda el controlador.
- Measurements Application (*MeasureApp*):
  Esta aplicación es la encargada de gestionar el sistema de medición, permitiendo la obtención de QoS de posibles caminos sobre la *ON*. Para esto hace uso de los servicios de *RouteApp* y envía órdenes a los *Probe Packet Generator (PPG)*. Los *PPG* son definidos en la sección 3.1.2.

Este proyecto propone el diseño e implementación de una plataforma flexible y escalable que permita la utilización de diferentes algoritmos de monitoreo e ingeniería de tráfico sobre la *ON*. Respecto al conjunto central este proyecto propone la implementación de *RouteApp* y *MeasureApp*, dejando la elección e implementación de algoritmos de monitoreo e ingeniería de tráfico para trabajos futuros.

3.1.2. Conjunto distribuido

El conjunto distribuido se compone por los diferentes puntos de presencia cuyo tráfico se desea controlar. En este proyecto, se define un *PoP* como una subred pública que contiene los siguientes elementos:

- Un *Switch OpenFlow*:
  Es el elemento de red que tiene la capacidad de modificar y encaminar el tráfico según las ordenes del conjunto central, mediante el protocolo *OpenFlow*.
- Un *Probe Package Generator (PPG)*:
  Un *PPG* es una entidad de software que tiene como objetivo la medición, según un criterio predefinido, de la QoS de los caminos en la *ON*. Los *PPG* son orquestados desde el conjunto central por *MeasureApp* de forma de ejecutar las medidas.
- LAN Cliente:
  Es la LAN que genera el tráfico que se desea controlar. Definimos "Cliente" como el conjunto de hosts que poseen una IP pública dentro de la subred asociada al *PoP*.
- Router del *ISP*:
  Por simplificación, el router del ISP es la única puerta de acceso a Internet que tiene el *PoP*.

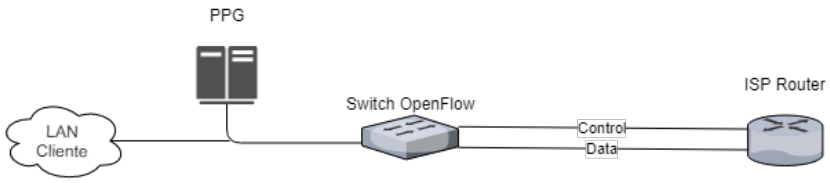


Figura 3.2: Arquitectura interna de un *PoP*

Cada punto de presencia se organiza como se presenta en la figura 3.2. Como se observa, el *Switch OpenFlow* posee dos vínculos asociados al router del proveedor de servicios de Internet. Esto implica que para estos switches es necesario reservar dos IPs públicas. Mientras una dirección IP se utiliza para establecer el canal de control para la comunicación con el controlador, la otra se utiliza para encaminar el tráfico de datos desde y hacia la subred a controlar. La fundamentación es que la interfaz de datos necesita una IP se explica en la sección 3.2.1.

Por otra parte, tanto el *PPG* como el resto de la subred pública a controlar se encuentran "detrás" del *Switch OpenFlow* con respecto al *ISP*. Esto fue diseñado de forma que todo tráfico saliente pueda ser tratado por el switch.

A modo de resumen, para cada uno de los *PoPs* el sistema necesita hacer uso de un conjunto de tres IP públicas (reservadas para el sistema), donde una de ellas se encuentra destinada al canal de control del switch, otra al canal de datos del switch y la última al software de medición *PPG*.

3.2. Algoritmo de identificación y encaminamiento

Esta sección comienza con la exposición del proceso de diseño del algoritmo que permite encaminar flujos de datos en una *ON*. En la segunda parte se realiza la especificación del algoritmo, presentando ejemplos de uso.

3.2.1. Diseño del algoritmo

Para lograr que un paquete sea encaminado a través de un nodo de la *ON*, evitando las rutas establecidas por BGP y los problemas introducidos en la sección 1.1, una posible solución es modificar la dirección IP destino. Este cambio debería ser por alguna de las direcciones de la subred asociada al *PoP* perteneciente a la *ON* por el que se desea encaminar el paquete. Utilizando el control central provisto por la arquitectura *SDN*, se podría realizar este cambio sucesivamente en todos los nodos de la *ON* por los que se desea que transite el paquete, reintegrando la dirección IP del destino original al final del recorrido.

Tradicionalmente un flujo de datos sobre IP se identifica por la quíntupla: IP origen, IP destino, puerto origen, puerto destino y protocolo de capa de transporte. Al modificar la IP destino se está modificando el flujo de datos y por lo tanto, es necesario un método que permita que distintos flujos sean identificables.

Si se está trabajando en el contexto de IPv6, cada nodo de la *ON* podría tener un rango de direcciones exclusivamente para este fin. Esto permitiría que durante la vida de un flujo, este sea unívocamente identificable.

Pero, ¿qué sucede si se está en un contexto como IPv4?, ¿es siempre posible identificar unívocamente el flujo utilizando únicamente sus direcciones IPv4?, ¿utilizando sus direcciones IP, puertos de capa de transporte y protocolo? o ¿es necesario establecer una marca en el cabezal del paquete IP que permita identificar el flujo durante su vida?

En este proyecto, se aborda un análisis de posibles soluciones para IPv4, dejando el encaminamiento en IPv6 para trabajos futuros. **A partir de aquí, todo lo que refiera a IP será IPv4**.

Con el fin de evaluar distintas alternativas de encaminamiento, considere la figura 3.3. En esta se muestra una *ON* sobre Internet que cuenta con cuatro *PoPs*. Cada PoP_i se encuentra compuesto por una subred 172.16.i.0/24, un host h_i, un *Switch Openflow* s_i y un router r_i perteneciente al *ISP* el cual representa el acceso a Internet (como simplificación no se agrega el *PPG* de cada *PoP*).

Si bien se resalta que en general la conexión entre todos los routers del *ISP* se encuentra compuesta por un gran número de routers intermedios, aquí se modela como una conexión directa, de forma de abstraerse de la red subyacente.

3.2.1.a. Primera aproximación: identificación usando las direcciones IP del paquete

Supongamos que se desea encaminar el tráfico desde h_1 a h_3, pasando por PoP_2. En este contexto, una forma de que los paquetes lleguen hasta s_2 es que la dirección IP destino pertenezca a la subred de este punto de presencia. Para

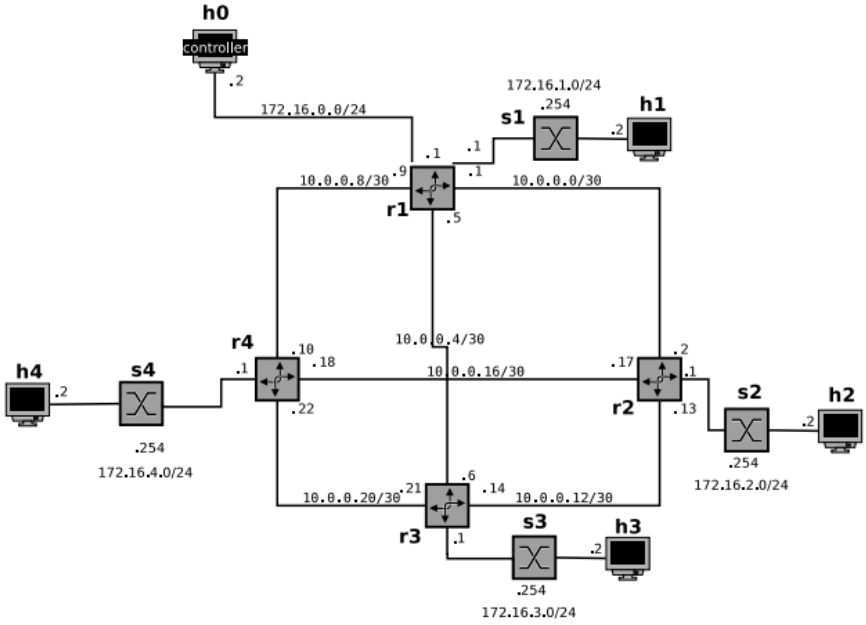


Figura 3.3: Red FullMesh con 4 PoPs. Imagen extraída de [11]

que esto suceda, una vez que un paquete enviado desde h_1 hacia h_3 pasa por s_1, este debe modificar la IP destino para que sea encaminado hacia PoP_2. En el momento que el paquete ingresa en s_2, se debe encaminar hacia el destino final h_3. La forma de lograr esto último es modificando nuevamente la dirección IP destino, configurando la dirección de h_3.

Surge ahora la duda, ¿qué pasa con la dirección IP origen al enviar el tráfico hacia s_3?

Como el paquete ahora se encuentra en PoP_2, existen dos posibilidades:

1. Mantener la IP origen incambiada

2. Modificar la IP origen, configurando una perteneciente a la subred de PoP_2

Tomemos en consideración el primer caso. Este tiene la simpleza de que cualquier paquete enviado conservará la IP origen en todo el camino y por tanto, cuando h_3 recibe el paquete, conocerá que el origen de este es h_1. Este caso implicaría que en s_3 no se realice procesamiento extra.

A pesar de la simpleza de la propuesta, rápidamente se encuentra un problema: la existencia de *"Reverse Path Filtering"* como una de las formas de protección que implementan los router.

Dado un paquete entrante, esta forma de protección cumple la regla de, en caso que su dirección IP de origen no sea ruteable por la interfaz por donde llegó, este es descartado [12–14]. Esto implica que los paquetes que sean encaminados

de la forma (1) tienen una gran probabilidad de ser descartados por los routers de los proveedores de servicios de Internet. Por lo tanto, este método no asegura la funcionalidad requerida.

Se concluye que, a la hora de hacer rebotar un paquete, tanto la dirección IP origen como la de destino deben ser modificadas de forma que la dirección destino sea la del siguiente punto de presencia por el que debe pasar el paquete mientras que la de origen pertenezca al PoP en el que se está rebotando.

Definimos *rebote* como el pasaje de un paquete por un PoP_C, que llega desde PoP_A y es redirigido hacia PoP_B, como se aprecia en la figura 1.1.

Con lo anterior presente, supongamos que mientras una aplicación desea comunicar h_1 y h_3, otra pretende comunicar h_1 y h_2. En el último caso, s_1 no debe realizar ninguna modificación sobre el paquete (no rebota en ningún punto de presencia) y debe ser encaminado directamente hacia h_2. En este contexto, se tienen paquetes que rebotan en s_2 con destino h_3 y paquetes que no rebotan con destino h_2, cuyas direcciones IP origen son h_1 y destino son h_2. Concretamente, ambos paquetes tienen las mismas direcciones IP origen y destino.

Entonces, ¿Cómo se diferencian estos dos tipos de paquetes o flujos?
Este caso se soluciona con un poco de ingenio y sin mayor complejidad. Si establecemos que cada *Switch OpenFlow* tenga asociada una IP pública perteneciente a la subred de cada punto de presencia, en el switch se puede crear la regla:

"Si el paquete recibido tiene como IP origen, la IP asociada a un switch de otro punto de presencia e IP destino, la IP que me asignaron, entonces aplicar la siguiente acción, si no, no hacer nada"

Esta regla presenta la restricción que a cada switch de cada PoP se le debe asignar una IP pública. Sin embargo, la identificación de tráfico se torna más complicada en el siguiente caso:

Supongamos que se desea enviar tráfico desde h_1 a h_3 pasando por el PoP_2, y también se desea encaminar otro tráfico desde h_4 a h_3, pasando por el PoP_2. En teoría, estos son dos caminos independientes, por lo tanto no debería haber ambigüedad alguna a la hora de identificarlos. Para ambos encaminamientos se cumple que, luego de rebotar en el switch del PoP_2, los paquetes tendrán configurado $IP\_Src = IPs_2$, $IP\_Dst = IPs_3$, y al ser recibidos en el s_3, este no tiene como reconocer si el tráfico fue originado en h_1 o h_4.

Por lo tanto, únicamente modificando las direcciones IP resulta imposible satisfacer los requisitos de identificación de flujos de datos que posee el sistema.

3.2.1.b. Segunda aproximación: identificación utilizando campos del encabezado IP

La segunda aproximación a la solución del problema antes descrito fue buscar un campo del cabezal IP que pudiese ser utilizado como identificador. Los *Switches OpenFlow* podrían utilizarlo para hacer *matching* contra los paquetes entrantes y

así identificar el camino a seguir. La motivación de utilizar un campo del cabezal de IP proviene de que el método resultaría generalizado para todos los protocolos de capa de transporte encapsulados sobre IP.

Con esto en mente, se comenzó investigando el campo *Differentiated Services Code Point (DSCP)*, antes conocido como *Type of Service (ToS)*. Este, con un largo de 6 bits, es utilizado para brindar QoS dentro de un mismo dominio, en el cual es posible configurar los elementos de infraestructura según un criterio unificado.

Sin embargo, este proyecto requiere que la identificación sea multidominio y por tanto, el identificador debe mantenerse invariante durante todo el recorrido por los diferentes *ISPs*. En el caso del campo *DSCP*, no es posible asegurar la invarianza en un contexto de múltiples dominios, por lo que se tuvo que descartar el método. [15]

Buscando una alternativa al campo anterior, surge una nueva idea: En el campo **protocolo** solo se utilizan los primeros 140 valores, dejando más de 100 sin asignar que podrían ser utilizados para instalar un identificador [16]. Sin embargo, la idea no floreció puesto que, si bien se puede hacer el *matching* de paquetes contra este valor, el mismo no es modificable utilizando una acción de *OpenFlow*. [10]

Otra posibilidad evaluada fue utilizar los campos *identification* y *offset*, con un tamaño de 16 y 13 bits respectivamente [17]. Esto brindaría gran escalabilidad para la identificación de flujos. Sin embargo, recordando que la *ON* trabaja sobre Internet, no es posible asegurar que el tamaño del paquete en el origen sea menor al tamaño de la menor *MTU* del trayecto hasta destino. Esto implica que en algún punto del trayecto sobre Internet pueda ser necesario fragmentar el paquete y por consiguiente, perder la información de encaminamiento de flujos.
El caso de poder asegurar el tamaño de la MTU en todo el trayecto, lo conveniente sería implementar un túnel de origen a destino por simplicidad de gestión.

3.2.1.c. Tercera aproximación: identificación utilizando el cabezal de capa de transporte

Del análisis anterior se concluye que, de usar únicamente el encabezado IP, no sería posible realizar la identificación. Por esta razón, la única alternativa es identificar los flujos de datos utilizando la información de capa de transporte.

En la última especificación de *OpenFlow* existente hasta la fecha V1.5 [10], el protocolo solo puede realizar acciones sobre campos de 3 protocolos de capa de transporte: TCP, UDP y SCTP.

Si bien acotar el encaminamiento a 3 tipos de tráfico es una clara desventaja, este conjunto de protocolos representan más del 90 % del tráfico total en Internet [18–20]. Como ventaja de los mismos, ambos definen la noción de puertos de origen y destino de capa de transporte. Teniendo en cuenta que en Internet únicamente se trabaja hasta capa de red, los campos se ven inalterados durante el recorrido multidominio y por tanto, es posible modificarlos en la medida que sea necesario.

Si bien el algoritmo diseñado será aplicado únicamente para los protocolos TCP y UDP, resulta interesante buscar una solución que escale a todos los protocolos de transporte que incluyan puertos en el encabezado, previendo un posible agregado

de protocolos en próximas versiones de *OpenFlow*.

3.2.2. Especificación del algoritmo

A continuación se detalla el algoritmo de encaminamiento de flujos diseñado. Para ello, serán definido algunos conceptos:

- *Flujo o Flujo de datos*
  Un *Flujo* queda determinado por los siguientes parámetros:

  $$Flujo = \{Ip\_Src, Ip\_Dst, Port\_Src, Port\_Dst, protocol\_L4\}$$

- *Link*
  Un *Link* $L_{i,j}$ queda determinado por una dupla ordenada de *PoPs*.

  $$L_{i,j} = [PoP_i, PoP_j] \quad / \ i, j \in \mathbb{N}$$

  Observar que esta dupla es direccional, es decir, $L_{j,i} = [PoP_j, PoP_i] \neq L_{i,j}$.

  El número de *Links* de la *ON* queda determinado por la cantidad de *PoPs* y la topología permitida. En el caso que la *ON* trabaje sobre Internet y no se tengan restricciones adicionales, se asume una topología *Full Mesh* entre los *PoPs*. Esto da como resultado una cantidad total de $n * (n - 1)$ *Links* dirreccionales.
- *Path*
  Un camino o *Path* refiere al conjunto de *PoPs* por los que un flujo de datos debe transitar al ser encaminado desde origen a destino.
- *Overlay Network Routing Policy (ONRP)*
  Es el objeto central de estudio en el algoritmo de encaminamiento propuesto. Se encuentra compuesto por un *Path* y parámetros relevantes que permiten diferenciar los flujos desde origen a destino. Una *ONRP* queda totalmente definida por los siguientes parámetros:

  $$ONRP = \{Subnet\_Src, Subnet\_Dst, Protocol, Port\_Src, Port\_Dst, Path\}$$

  Con el objetivo de incluir varios tipos de tráfico en una misma *ONRP*, los campos *Protocol*, *Port_Src* y *Port_Dst* pueden no estar especificados. Se destaca que una *ONRP* puede tener uno o más flujos asociados. En contraposición, un flujo de datos de la *ON* siempre tiene una única *ONRP* asociada.

Para brindar mayor flexibilidad en cuanto a implementación de políticas de QoS, debe ser posible asignar distinto grado de prioridad a distintos tipos de tráfico asociado a un punto de presencia. Esto se traduce en la necesidad de diferenciar subconjuntos de tráfico generado en un *PoP*, permitiendo el encaminamiento por diferentes *Paths*, según sea requerido.

Recordando que únicamente se considerarán protocolos de capa de transporte que utilicen puertos, la forma de diferenciar flujos podría ser especificando puerto

origen, puerto destino y protocolo, tal como se muestra en la definición de *ONRP*.

Una vez definidos estos conceptos, se procede a detallar el algoritmo de encaminamiento implementado.

Una *ONRP* tiene asociado un determinado tipo de tráfico que se desea encaminar. Una vez que una *ONRP* es establecida en el *Switch OpenFlow* del PoP_{origen}, el tráfico hará *match*. A este tráfico se le modificará la IP destino a la asignada al switch del *PoP* siguiente e IP origen la asignada al switch del PoP_{origen}, tal como se mencionó anteriormente.

Cada *ONRP* tiene asociado m *PoPs* especificados en el *Path*, y por lo tanto $m-1$ *Links*, *con* $m \in \mathbb{N}$.

Para lograr identificar el tráfico asociado a distintas *ONRP*, a cada *Link* de la *ON* se le asocia una colección de números enteros acotada por un número máximo N_{max} y un número mínimo N_{min}, llamados *Local ONRP Id*.

Con esta construcción lógica, a una *ONRP* que pasa por k *Links* se le asociarán k identificadores locales, uno por cada *Link*. Estos identificadores no pueden volver a asignarse a ninguna otra *ONRP* mientras estén en uso.

Esta lógica de identificadores locales se basa en los principios de MPLS, en el cual se asocia un identificador local entre cada par de enrutadores mediante el mecanismo *Push* y *Pull* de *tags* en cada paquete [21]. La ventaja que tiene el algoritmo propuesto respecto a MPLS es que en el último se agrega información extra en cada *Push* de identificadores locales, con los problemas de fragmentación que puede generar en un ambiente multidominio como es Internet. En este proyecto, esta información es enviada en el puerto origen de capa de transporte, siendo modificado en cada *Switch OpenFlow*, por lo que la carga útil permanece inalterada.

Cabe destacar que la gestión de todos los identificadores asignados para cada *Link* se torna simple en el paradigma *SDN* debido a que la única pieza que gestiona toda esta información es el controlador.

Este método de encaminamiento se vuelve totalmente escalable, con un límite teórico de 2^{16} *ONRPs* por *Link*. Este límite es impuesto por la cantidad de bits disponibles en el campo puerto origen de capa de transporte.

Si bien la identificación entre distintas *ONRPs* queda totalmente determinada y sin ambigüedad, puede ocurrir que en una *ONRP* no se especifique protocolo, puerto origen o puerto destino, o se especifique subred origen a subred destino, o que un host particular de origen dentro de esta subred quiera comunicarse con un host destino, por ejemplo, a un servidor web. En este caso, ¿Cómo se recupera esta información en el extremo final del *Path* de la *ONRP*?

En cada *ONRP* es necesario identificar estos flujos con el fin de recuperar la información una vez que el paquete llega al nodo destino. Para lograr esto, la información será enviada en el puerto destino de capa de transporte, que permanece inalterado en todo el *Path*.

A modo de ejemplo, supongamos que se quiere implementar una política de encaminamiento en una *ON* compuesta de cuatro reglas. Estas reglas se especifican en la tabla 3.1 y representan los cuatro *Paths* de la figura 3.3.

Recordando la definición de *Link*, en este caso existen 12. Cada uno de estos es de la forma:

$$L_{i,j} = [PoP_i, PoP_j] \quad \forall\ i, j = [1, 2, 3, 4], i \neq j$$

Tabla 3.1: Cuatro ejemplos de *ONRPs*

# ONRPolicy	1	2	3	4
Red origen	172.16.1.8/32	172.16.1.0/24	172.16.2.0/24	172.16.1.0/24
Red destino	172.16.3.10/32	172.16.4.0/28	172.16.4.4/32	172.16.3.0/24
Protocolo	UDP	*	TCP	*
Puerto L4 origen	15194	*	*	*
Puerto L4 destino	15193	*	80	*
Path	[PoP1,PoP2,PoP3]	[PoP1,PoP2,PoP4]	[PoP2,PoP1,PoP4]	[PoP1, PoP2, PoP4, PoP3]

Supongamos que un host h_1 en el PoP_1 quiere comunicarse con un host particular h_3 en PoP_3. Los campos del encabezado desde origen a destino de todos estos paquetes se muestran en la tabla 3.2.

Tabla 3.2: Traducción de un *flujo* perteneciente a la $ONRP_1$ en el recorrido del *Path*

	Origen	L1,2	L2,3	Destino
IP origen	172.16.1.8	172.16.1.254	172.16.2.254	172.16.1.8
IP destino	172.16.3.10	172.16.2.254	172.16.3.254	172.16.3.10
Protocolo	UDP	UDP	UDP	UDP
Puerto L4 Origen	15194	1	1	15194
Puerto L4 Destino	15193	1	1	15193

Este tráfico hará *match* en s_1 y encaminado al próximo salto por $L_{1,2}$ con las características mostradas en la tabla 3.2 en la columna $L_{1,2}$.

Observar que una vez que un paquete haga *matching*, tanto la IP origen como la IP destino son referidas a las IPs públicas asignadas a los *Switches OF* de cada *PoP*. Este tráfico atraviesa de origen a destino los *Links* $L_{1,2}$ y $L_{2,3}$. El identificador local que tiene asignado esta *ONRP* en cada *Link* es $[L_{1,2}, L_{2,3}] = [1, 1]$.

Una vez que el paquete llegue al switch ubicado en el PoP_2, este hará *match* y será enviado hacia el PoP_3 con las características mostradas en la tabla 3.2 en $L_{2,3}$.

Luego que el paquete llegue al switch ubicado en el PoP_3 y haga *match*, gracias a la información del puerto origen y destino de capa 4 se podrán recuperar los valores iniciales. Por lo tanto, el encaminamiento sobre la *ON* resulta transparente a los host de las puntas.

Supongamos ahora que h_1 pretende comunicarse con h_4 ubicado en PoP_4. En todo el recorrido el tráfico tiene las características indicadas en la tabla 3.3.

Tabla 3.3: Traducción de un *flujo* perteneciente a la $ONRP_2$ en el recorrido del *Path*

	Origen	L1,2	L2,4	Destino
IP origen	172.16.1.8	172.16.1.254	172.16.2.254	172.16.1.8
IP destino	172.16.4.3	172.16.2.254	172.16.4.254	172.16.4.3
Protocolo	TCP	TCP	TCP	TCP
Puerto L4 Origen	43000	2	1	43000
Puerto L4 Destino	8080	1	1	8080

Se aprecia que este tráfico pertenece a la $ONRP_2$ y seguirá sus reglas estipuladas. Al atravesar el $Link_{1,2}$, el tráfico sobre Internet tendrá las características mostradas en la tabla 3.3.

Observar que el puerto origen de capa de transporte toma el valor 2. Esto se debe a que la $ONRP_1$ consumió anteriormente el identificador local 1 para este *Link* y, por lo tanto, mientras el Id esté en uso no se podrá reutilizar. Por este motivo, el identificador local en $L_{1,2}$ para la $ONRP_2$ es 2.

Si bien el algoritmo de encaminamiento queda especificado, resta definir que sucede con la información de los flujos que ingresan a la *ON*. Como ya se mencionó, para realizar la identificación de un flujo ya ingresado a la *ON* asociado a una *ONRP*, es necesario modificar los campos *IP src*, *IP dst*, *Port src* y *Port dst*. Esto implica que es necesario almacenar la información original que se encontraba en los campos previo a la modificación.

Para cada flujo asociado a cada *ONRP*, esta información se almacena en una entrada llamada *Overlay Network Address Translation* u *ONAT*, a la que se le asocia un identificador único llamado *ONAT Id* (el identificador se puede repetir entre *ONRPs* distintas).

A los ejemplos ya planteados en esta sección, agreguemos el caso que otro host pretende comunicarse desde PoP_1 a PoP_4. Observando la tabla 3.1, esta comunicación pertenece a la $ONRP_2$. La particularidad es que van a existir dos flujos distintos para una misma *ONRP*. Si bien el tráfico para ambos flujos seguirá el mismo *Path* utilizando los mismos identificadores locales, la información en destino será recuperada a partir del puerto destino de capa de transporte, que necesariamente debe ser único. Es en este campo donde se introducirá el identificador de la entrada *ONAT* asociada al flujo.

En la tabla 3.4 se presenta una transición entre origen y destino de un caso que cumple las características antes mencionadas.

Tabla 3.4: Traducción *ONAT* de dos *flujos* perteneciente a la $ONRP_2$ en el recorrido del *Path*

	Origen		$L_{1,2}$		$L_{2,4}$		Destino	
Overlay Network Address Translation	***Flujo 1***	***Flujo 2***	***Flujo 1***	***Flujo 2***	***Flujo 1***	***Flujo 2***	***Flujo 1***	***Flujo 2***
IP origen	172.16.1.8	172.16.1.10	172.16.1.254	172.16.1.254	172.16.2.254	172.16.2.254	172.16.1.8	172.16.1.10
IP destino	172.16.4.3	172.16.4.3	172.16.2.254	172.16.2.254	172.16.4.254	172.16.4.254	172.16.4.3	172.16.4.3
Protocolo	TCP	TCP	TCP	TCP	TCP	TCP	TCP	TCP
Puerto L4 Origen	43000	49000	2	2	1	1	43000	49000
Puerto L4 Destino	8080	22	1	2	1	2	8080	22

ONAT Id	1	2
IP origen	172.16.1.8	172.16.1.10
IP destino	172.16.4.3	172.16.4.3
Protocolo	TCP	TCP
Puerto L4 Origen	43000	49000
Puerto L4 Destino	8080	22

Observar que la identificación del flujo se preserva durante todo el *Path* en el puerto destino de capa de transporte y que, dado que los flujos pertenecen a la misma *ONRP*, tienen los mismos identificadores locales en cada *Link*.

En la figura 3.4 se muestra el encaminamiento de tráfico generado por las primeras 3 *ONRPs*.

Consideremos ahora la $ONRP_4$. Si comparamos los *match* respecto a la $ONRP_1$, resulta evidente que estos últimos se pueden considerar como un subconjunto de todos los *match* posibles en la $ONRP_4$ y lógicamente se le podría atribuir mayor prioridad por el hecho de ser más específico. Para considerar estos casos, así como también maximizar la eficiencia de búsquedas de *ONRPs* en el controlador, se diseñó un algoritmo de prioridad. Dado que este algoritmo de prioridad forma parte de la implementación y no del algoritmo de encaminamiento, será explicado en la sección 5.2.1.b asumiendo que la identificación por subconjuntos puede realizarse.

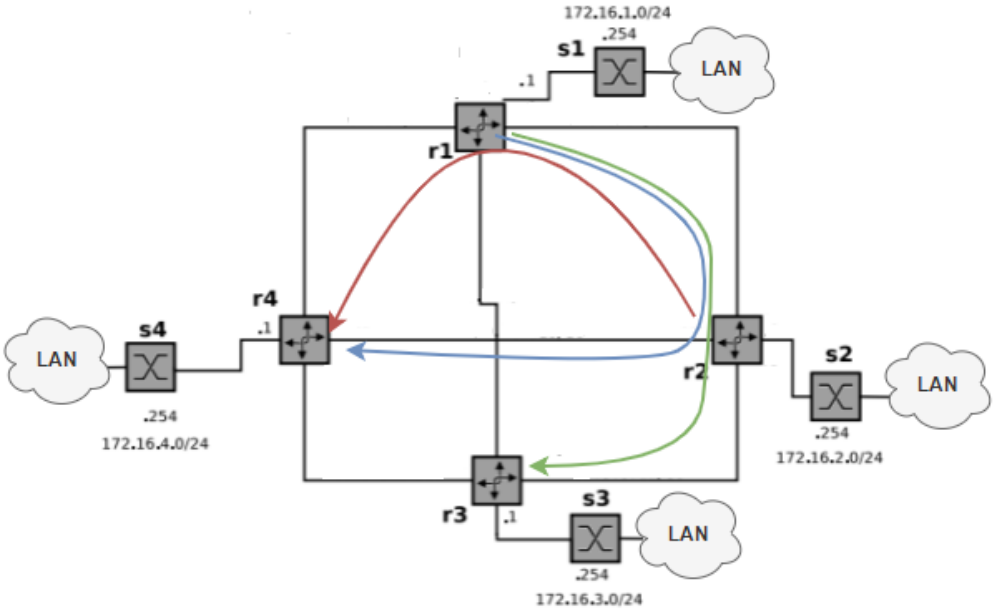


Figura 3.4: *Paths* de tres *ONRPs* sobre Internet. Imagen editada a partir de [11]

3.3. Metodología de medición de QoS

En esta sección se presenta el método diseñado para obtener una medida de la QoS de un camino o *path* sobre una *ON*. El método diseñado no solo involucra la técnica de medición (*RTT*, *Bandwidth*, etc), sino también como aplicar la técnica de forma que sea compatible con el algoritmo de encaminamiento propuesto en 3.2.

Recordemos la arquitectura presentada en la sección 3.1. Con ella puede surgir la duda:
¿Por qué es necesario tener instancias distribuidas del software de medición (*PPG*) y no es posible realizar las medidas utilizando directamente los+54163. *Switches OpenFlow* o desde el controlador?

Para cuantificar la QoS de la red es necesario realizar mediciones activas (inyección de tráfico) sobre cada *Path* particular. Parámetros como retardo y *jitter* se hacen indispensables para analizar el desempeño de una red. Sin embargo, hasta el momento los *Switches OpenFlow* no presentan la capacidad de estampar marcas de tiempo en los paquetes, lo que agrega una dificultad extra al momento de estimar estos parámetros bajo el paradigma *SDN*. En particular, esto implica la imposibilidad de utilizar propiamente el protocolo *OpenFlow* para solucionar el problema.

Dos trabajos interesantes se han centrado en el problema de la medición de QoS [22, 23]. Sin embargo, estas propuestas obtienen estimaciones de retardo y *jitter* poco precisas. Además, están pensadas para la red interna de un *data center* donde se puede diseñar la red de forma que el retardo entre los switches y el

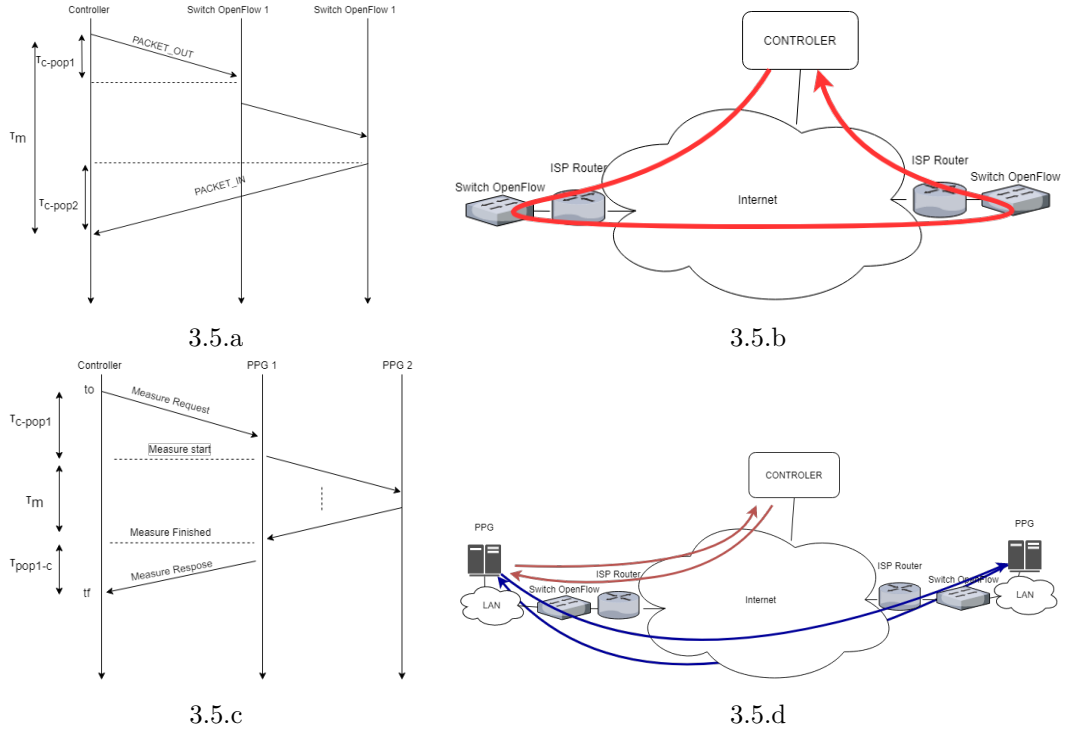


Figura 3.5: Comparativa de dos métodos de medición

controlador sea muy pequeño y razonablemente estable. En una *ON* sobre Internet resulta complejo estimar con precisión estos retardos y pueden ser comparables al existente entre switches pertenecientes a *PoPs* que se desea medir.

Por este motivo, incluir elementos que realicen medidas en cada punto de presencia se vuelve indispensable. En la figura 3.5 se muestra la diferencia sustancial entre realizar las medidas utilizando el controlador y colocar elementos de software destinados a esto.

Supongamos que la medida se realiza utilizando únicamente el controlador. En la figura 3.5.b se muestra el camino que debe seguir el tráfico de medida (*Probes*) para poder obtener información de la QoS, como por ejemplo el retardo entre *PoPs*. En este caso, el controlador envía un *PACKET_OUT* al *Switch OF* ubicado en el PoP_1 y este lo redirige al switch del PoP_2. Allí es enviado como *PACKET_IN* hacia el controlador tal como se muestra en la figura 3.5.a, obteniendo un tiempo de viaje del paquete τ_m. Para obtener el tiempo de retardo τ_R entre los *PoPs* se debería aplicar:

$$\tau_R = \tau_m - \tau_{C-PoP_1} - \tau_{C-PoP_2}$$

Sin embargo, no se conoce el tiempo de retardo entre el controlador y los puntos de presencia τ_{C-PoP_1} y τ_{C-PoP_1}. Como se mencionó anteriormente, el ambiente de trabajo subyacente es Internet, por lo que no es posible despreciar estos tiempos frente al tiempo de interés τ_R.

El agregado de un *PPG* en cada *PoP* permite obtener medidas más precisas de

QoS. En la figura 3.5.d, se muestra un flujo de datos de control entre controlador y un *PPG* y un segundo flujo entre *PPGs* que representa la medida. A los paquetes de este últlimo flujo se le asocia el nombre de *Probes*. En la figura 3.5.c se puede observar el diagrama de tiempos asociado al proceso de medición usando *PPGs*.

El proceso comienza con un mensaje de control *Measure_Request* hacia el *PPG* encargado de iniciar la medida. Una vez que el *PPG* recibe e interpreta la petición desde el controlador, inicia la medida inyectando tráfico en la red. Durante un tiempo τ_m ambos *PPGs* mantienen la comunicación para obtener parámetros relevantes del canal. Una vez recolectados los datos necesarios, el *PPG* de origen envía un mensaje al controlador *Measure_Response* con los resultados de la medida realizada.

La arquitectura general del sistema asume la presencia externa de una aplicación de monitoreo de la red (*MonApp*) y una aplicación de ingeniería de trafico (*TEApp*) que toman decisiones de cuando y hacia donde medir. Teniendo en cuenta que el ambiente multidominio (Internet) es sumamente dinámico, *TEApp* puede tomar decisiones no solo en base al valor de QoS, si no también considerando el instante de tiempo en el que fue realizada la medida.

El reloj interno del *PPG*, el del Controlador y el del servidor donde se ejecuta *TEApp* no necesariamente están sincronizados. Esto implica que, para conocer el tiempo exacto en el cual se inicia y finaliza la medida, es necesario hacer uso de un único reloj para determinarla. A partir de la figura 3.5.c resulta claro que se necesitan conocer los tiempos τ_{C-PoP_1}, τ_{PoP_1-C}, τ_m, t_o y t_f. Por lo tanto, indiferentemente de la métrica utilizada para QoS, el *PPG* de origen siempre debe devolver el valor τ_m en el mensaje *Measure_Response*.

Asumiendo que $\tau_{C-PoP_1} \simeq \tau_{PoP_1-C}$, es posible obtener el tiempo de retardo Controlador-PoP_1 como:

$$\tau_{C-PoP_1} = \frac{t_f - t_i - \tau_m}{2}$$

Con estos parámetros, es posible calcular:

$$T_{MeasureStarted} = t_f - \tau_{C-PoP_1} - \tau_m$$

$$T_{MeasureFinished} = t_f - \tau_{C-PoP_1}$$

Detalles de sincronización entre *TEApp*, *MonApp* y el controlador quedan fuera del alcance de este proyecto.

Este proyecto se centra en la creación de la plataforma de medición, involucrando el desarrollo de partes como la *API* expuesta desde *MeasureApp* hacia *MonApp*, las comunicaciones *MeasureApp-PPG* y *PPG-PPG*. Este diseño se realizó abstrayéndose de la técnica de medición en si misma, como *delay*, *jitter*, *throughput*, etc.

El alcance de este proyecto únicamente incluye *RTT* como técnica de medición de QoS. En este caso este valor puede obtenerse directamente como:

$$RTT = \tau_m$$

Hasta este punto se tiene una primera aproximación al método de obtención de la medida, así como las dificultades de realizarlas bajo el paradigma *SDN*. La pregunta que queda responder es: ¿Qué método se utiliza para especificar el *Path* en el que se inyectarán *probes* de medición?

Para responder esto es necesario recordar el algoritmo de identificación y encaminamiento desarrollado en la sección 3.2. Por cada *Path* que se desee medir se generará un flujo de datos distinto a ser identificado por *RouteApp*. Resulta importante entender que, dado un origen A y un destino B, si se desea medir dos *Path* distintos entre ellos, como pueden ser $A \leftrightarrow C \leftrightarrow B$ y $A \leftrightarrow D \leftrightarrow B$, es necesario generar dos políticas de encaminamiento para cada una de ellas (cuatro *ONRPs*), como se muestra en la figura 3.6.

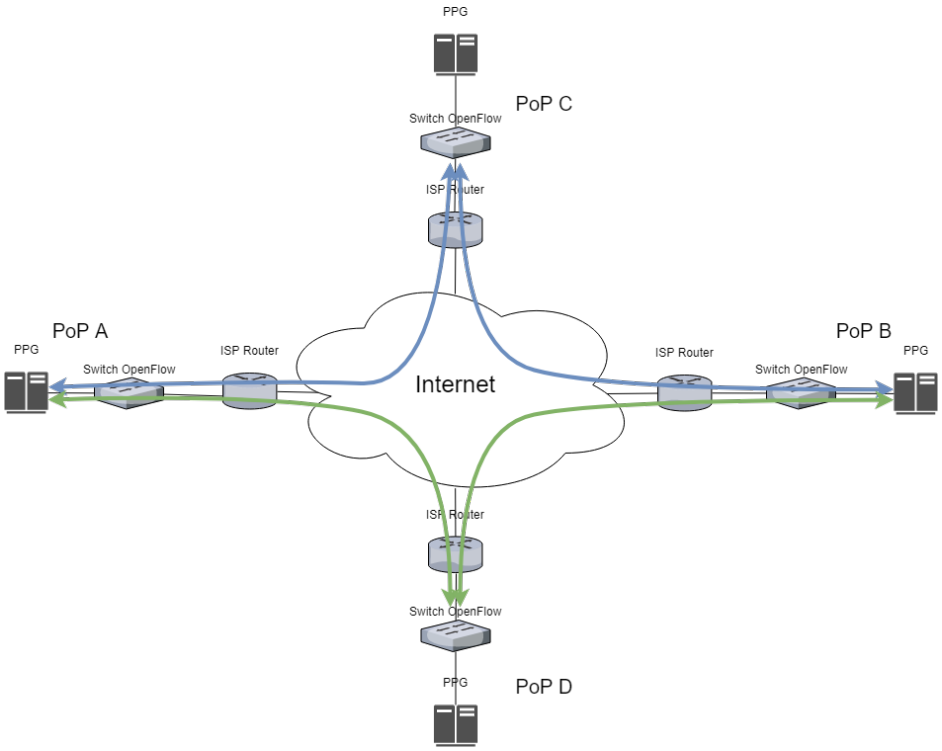


Figura 3.6: Medición simultanea de dos *Paths* entre PPG_A y PPG_B

Para poder acoplar el algoritmo de encaminamiento, la solución propuesta es la siguiente:

- Cada *PPG* posee un único puerto de capa de transporte de recepción por el que pueden llegar *Probes*.
- Para discernir entre dos *Paths* entre un mismo origen y un mismo destino, se utiliza el campo puerto origen de capa de transporte.
- El puerto de origen que un *PPG* debe utilizar para realizar cierta medición es decidido por *MeasureApp* e indicado al *PPG* de origen en el mensaje *Measure_Request*.

- La medición de QoS de un *Path* en base a *RTT* se realiza mediante la inyección de paquetes UDP. Existen dos razones principales por las que este es el protocolo utilizado para realizar la medida:
  1. La única forma de que paquetes transiten un *Path* específico a través de la red sobrepuesta es utilizando el algoritmo propuesto en la sección 3.2. Este algoritmo trata únicamente el tráfico generado con los protocolos TCP y UDP.
  2. Existen excelentes herramientas para realizar pruebas de *QoS* en una red desarrolladas para inyectar tráfico particularmente UDP y TCP.
  3. Entre los protocolos disponibles, TCP es un protocolo confiable y orientado a conexión. Por tanto, agrega una sobrecarga de tiempo generada en el establecimiento y finalización de la conexión y en las eventuales retransmisiones. Esto se traduce en una mayor demora en la transmisión de un *Probe*.
- Para la medición de un *Path* en base a *RTT* se crean dos *ONRPs* (ver sección 3.2), una para la ida y otra para la vuela. En la figura 3.6 estas dos *ONRPs* se exponen con la flecha doble de cada medición.

Con estas características definidas, *MeasureApp* puede solicitar a *RouteApp* la creación de políticas de encaminamiento específicas que permitan la medición de QoS de las rutas requeridas. Una vez establecidos los circuitos virtuales en la *ON*, *MeasureApp* puede dar inicio al proceso de medición presentado en las figuras 3.5.c y 3.5.d.

Hasta aquí se han presentado los diseños teóricos realizados, cumpliendo con tres de los objetivos planteados en la sección 1.2. En los próximos capítulos se presentará el controlador elegido y la aplicación desarrollada para implementar los algoritmos aquí presentados.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 4

Introducción al controlador ONOS

Este capítulo brinda una breve introducción al controlador elegido para realizar el desarrollo de las aplicaciones requeridas. Se presenta la arquitectura interna del controlador y se analiza como se deben integrar las aplicaciones a realizar.

Se destaca que, si bien en este capítulo se presenta específicamente el controlador utilizado para desarrollar las aplicaciones, se realizó un minucioso proceso en la elección del controlador. En el proceso se evaluaron parámetros como *performance*, modularidad, documentación disponible, adopción en el mercado, etc. Esto se realizó a través de la búsqueda y análisis de trabajos previos que evaluaran los controladores *SDN* más populares. [24–27]

El análisis concluye con la elección de *ONOS* como controlador apropiado para el desarrollo de las aplicaciones requeridas en este proyecto. Si desea conocer en profundidad el análisis realizado para su elección, dirigirse al anexo A.

4.1. Modelo de capas de la arquitectura

Toda la información acerca del controlador *ONOS* expuesta en esta sección es extraída de [28]. Aquí solo se pretende exponer un resumen del controlador utilizado para desarrollar la aplicación.

La arquitectura de *ONOS* se encuentra dividida en varias capas que progresivamente abstraen las operaciones de los protocolos y los dispositivos, de forma de presentar a las aplicaciones un conjunto uniforme y unificado de funcionalidades que se encuentren libres de ambigüedades.

La capa fundamental del esquema es el núcleo de *ONOS* (*CORE*) que, mientras permite la separación entre las funciones de datos y las de control, es responsable de presentar una vista centralizada del estado de la red y de brindar un acceso centralizado a las funciones de control de la misma. Este núcleo se encuentra separado de las demás capas por dos interfaces límites, que lógicamente resultan distintas.

La interfaz sur es una *API* de alto nivel de abstracción, a través de la cual el núcleo puede interactuar con los dispositivos presentes en el medio de red. En vez de interactuar directamente con los dispositivos, *ONOS* depende de adaptadores

específicos de cada protocolo de comunicación, con los que el núcleo puede realizar las interacciones haciendo uso del protocolo necesario (*OpenFlow*, *NetConf*, *OVSDB*, etc). Por tanto, la interfaz sur actúa de puente para enviar y recibir información de los distintos proveedores de protocolo (*Providers*). La independencia de esta *API* frente a los protocolos de comunicación garantiza que ninguna característica específica de protocolos se filtre hacia el núcleo de *ONOS* ni hacia las aplicaciones.

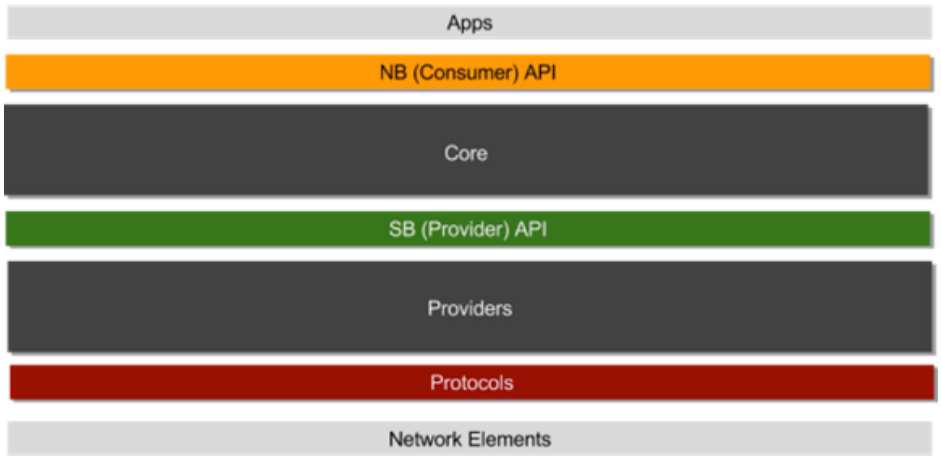


Figura 4.1: Arquitectura de capas de *ONOS*. Imagen extraída de [28]

En la interfaz norte, el núcleo de *ONOS* expone un conjunto de abstracciones hacia las aplicaciones para que estas accedan a servicios de red. Estas abstracciones brindan cierto rango de acceso a la información de red, comenzando por abstracciones de bajo nivel de la topología en donde se brinda información de dispositivos, enlaces y hosts, hasta abstracciones de alto nivel como por ejemplo la gráfica de la topología de red. De forma similar, brinda un conjunto de abstracciones para modificar el estado de red.

4.2. Estructura del *CORE*

En la literatura generalmente el núcleo de *ONOS* se identifica como un único bloque como en la figura 4.1. Sin embargo, en la realidad es un ensamblaje de subsistemas individuales, donde cada uno de ellos es responsable de sus propios servicios. La figura 4.2 presenta los conjuntos subsistemas antes mencionado.

Muchos de estos subsistemas son utilizados por otros subsistemas que, en conjunto brindan las funcionalidades expuestas por la *API* norte. Algunos de estos subsistemas se encargan del control interno de datos del conrolador, estos se encuentran en la imagen 4.2 coloreados en gris. Coloreado con rojo se encuentran los subsistemas que controlan y gestionan la red.

El objetivo de esta figura es demostrar que el núcleo de *ONOS* no es un único bloque, sino que es modular y, por tanto, permite ser modificado con el nivel de

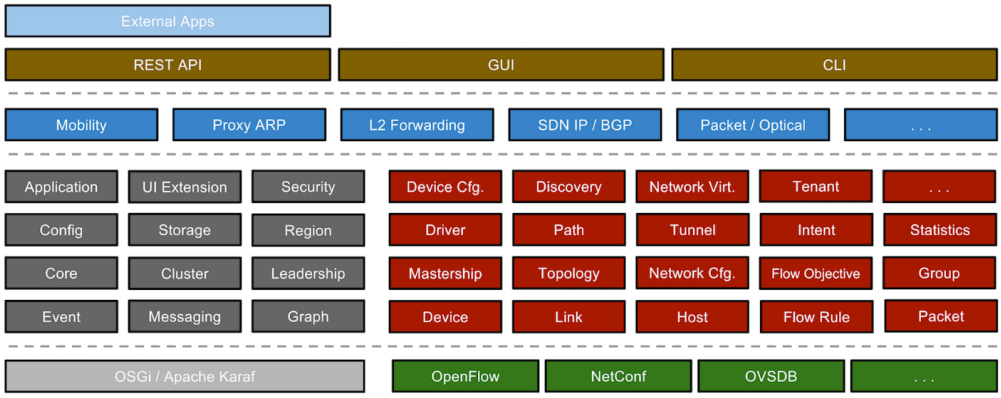


Figura 4.2: Diagrama de bloques de la arquitectura de de *ONOS*. Imagen extraída de [28]

granularidad requerido, siempre y cuando se preserven las interfaces de comunicación.

4.3. Integración de nuevas aplicaciones a *ONOS*

La esencia de las redes definidas por software es que la red sea programable a partir de aplicaciones que utilicen funciones que ofrece el controlador. Sin embargo, si se observa la figura 4.2 se puede diferenciar dos tipos de aplicaciones:

- Aplicaciones externas: son las aplicaciones estandar del paradigma *SDN*. Estas aplicaciones son externas al controlador, y se comunican con este a través de una *API* mediante la cual el controlador expone sus servicios de red.
- Aplicaciones del controlador: Estas aplicaciones no solo utilizan funcionalidades del controlador, sino que exponen servicios de red propios y por lo tanto deben estar instanciadas dentro del controlador tal como se muestran en color azul en la figura 4.2. Hacen uso directo de los servicios de *CORE* del controlador.

En el caso del proyecto, resulta imposible cumplir con los requerimientos planteados en el capítulo 1 si la aplicación es externa al controlador debido a la necesidad de que el sistema implemente nuevas funcionalidades de red, como el encaminamiento de tráfico automático mediante la imposición de políticas sobre una red sobrepuesta, utilizando el algoritmo planteado en la sección 3.2.

Por lo tanto, la aplicación diseñada es instanciada en el controlador, entre *Northbound API* y el *CORE* del mismo.

En el capítulo 5 se aborda una descripción formal de la aplicación diseñada, presentando su arquitectura de software, servicios del *CORE* de *ONOS* que utiliza y funcionalidades de red que expone a aplicaciones externas.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 5

ONRApp - Overlay Network Routing Application

En el presente capítulo se describe la arquitectura de software utilizada en la implementación de la aplicación.

Se detalla como esta responde ante eventos externos generados desde el plano de datos, así como también desde el plano de aplicación del paradigma *SDN*, finalizando el capítulo con una descripción de las funcionalidades de red que expone de aplicación por intermedio de *API REST*.

5.1. ¿Qué es *ONRApp*?

ONRApp es una aplicación diseñada con el objetivo de brindar servicios de encaminamiento de tráfico y medición en una *ON* de forma centralizada. Mediante una *API* diseñada acorde a los requerimientos del sistema expuestos en la sección 1.2, *ONRApp* funciona como capa de abstracción para entes externos que deseen establecer políticas de ruteo sobre una *ON*.

La aplicación utiliza servicios del controlador *ONOS* para traducir las políticas externas a reglas aplicables en el plano de datos, realizando la orquestación del tráfico y administrando de forma autónoma la creación y medición de QoS de caminos virtuales.

Se destaca que, si bien las políticas pueden ser establecidas manualmente, la *API* es diseñada de forma tal de permitir que los entes externos sean otras aplicaciones que, haciendo uso de las funcionalidades de *ONRApp* implementen decisiones tomadas a partir de algoritmos de ingeniería de tráfico y monitoreo de una *ON*.

En el anexo A se muestra un análisis exhaustivo del proceso de elección del controlador *SDN* utilizado en este proyecto. Allí se remarca la importancia del controlador en cuanto a la flexibilidad de desarrollar aplicaciones que, a partir de *Northbound APIs* expongan los servicios de red. Estos servicios permiten, por ejemplo, administrar flujos de datos y monitorear el estado de la red.

Como se muestra en 4.1, las entidades que corren en capa de aplicación pueden

comunicarse con el controlador a través de peticiones cliente-servidor mediante *REST API*.

En el modelo *RESTful* cada transacción contiene toda la información necesaria ya sea para modificar (método *POST*) o para pedir información (método *GET*) del estado de un recurso. Con este modelo resulta imposible desarrollar una aplicación fuera del controlador que utilice servicios de red expuestos por el mismo que se ajuste dinámicamente y de forma "instantánea" a eventos del plano de datos, dado que mediante API REST no se puede informar de forma reactiva a aplicaciones externas eventos en tiempo real.

ONRApp no solo hace uso de funcionalidades del controlador, sino que debe exponer servicios de red particulares como por ejemplo el encaminamiento en una *ON*, pudiendo interpretar peticiones de alto nivel para luego traducirlas en la creación de rutas de tráfico en el plano de datos utilizando el algoritmo planteado en 3.2

Por estas razones, *ONRApp* no es una aplicación estándar en el modelo *SDN*, sino que debe ser instanciada por el controlador. Es por esto que la aplicación es implementada en *JAVA* al igual que la arquitectura de software del controlador ya que forma parte de este y por ende utiliza servicios internos del mismo.

5.2. Arquitectura de software

El diseño de la arquitectura de software de *ONRApp* se basa en un patrón de arquitectura en capas, con el objetivo de lograr modularización, jerarquía e independencia. [29] Este enfoque de capas soporta el desarrollo incremental de sistemas acorde al avance en la implementación de cada capa. Esto permitió desarrollar cada capa de forma independiente, con la posibilidad de agregar nuevos servicios siempre y cuando se mantengan las interfaces entre las mismas.

Para brindar un servicio atractivo y que presente beneficios reales, *ONRApp* debe acompañar los requerimientos funcionales de ruteo automático y sistema de medición centralizado con requerimientos no funcionales, como por ejemplo, rendimiento en cuanto al tiempo requerido en ejecutar procesos.

Llamamos requerimientos no funcionales a aquellos requerimientos que no se especifican como criterio de éxito del sistema, pero por el marco de trabajo deben tenerse en consideración.

Recordemos que *ONRApp* tiene como finalidad brindar servicios de red a entidades del plano de aplicación. Minimizar el tiempo de ejecución es crítico al momento de diseñar el sistema.

En el contexto de este trabajo, no se debe olvidar la importancia de la seguridad del sistema, parámetro fundamental en cualquier sistema multidominio.

Si bien en esta primera versión de *ONRApp* no se hace énfasis en vulnerabilidades de seguridad que presenta el sistema, la elección del modelo de capas permite escalar en versiones futuras a consideraciones en este parámetro. Destacamos que *ONOS* presenta módulos que exponen servicios de seguridad, por lo que en versiones futuras podrían integrarse en la aplicación.

En la figura 5.1 se presenta la arquitectura de software de *ONRApp* con sus principales subsistemas:

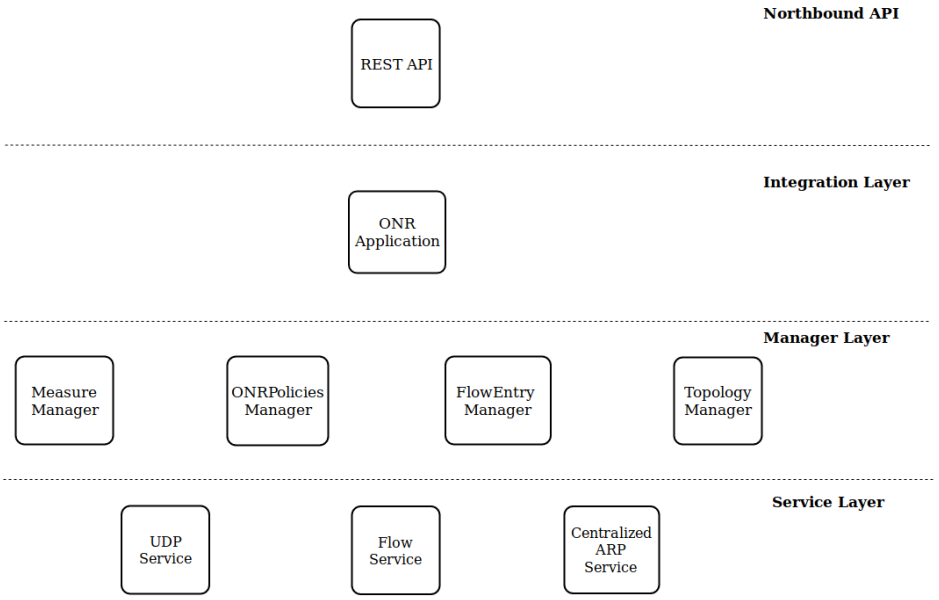


Figura 5.1: Arquitectura de *ONRApp* separada en sus diferentes capas internas

Vale la pena destacar que los subsistemas mostrados en la figura 5.1 son propios de la aplicación. Para lograr brindar las funcionalidades expuestas por la aplicación estos subsistemas deben comunicarse con el núcleo del controlador, tal como se explica luego en esta sección.

Como se observa en 5.1, el sistema se organiza en 4 capas de abstracción:

- *Northbound API*
  Se utilizan estructuras brindadas por *ONOS* que facilitan la exposición de funcionalidades a la capa de aplicación. *ONOS* brinda la capacidad de exponer los servicios de red mediante *API REST*. Utilizando la arquitectura *RESTful* brindada por *ONOS* para exponer servicios, se diseña una *API* que expone las funcionalidades de encaminamiento y medición sobre una *ON*.
  En esta versión solo se contempla la interacción con *ONRApp* mediante *API REST* desde capa de aplicación.
  Detalles de la *API* diseñada hacia capa de aplicación se exponen en el anexo C.
- *Integration Layer*
  Esta es la capa donde se encuentra *ONRApp*. En esta capa se instancian todos los *Managers* y por lo tanto, es quien establece la comunicación entre estos.
  Se encarga de transmitir el flujo de información de Petición-Respuesta desde *Northbound API* hacia capas más bajas de la aplicación.
- *Manager Layer*

Esta capa está compuesta por distintos subsistemas *Managers* quienes se encargadan de administrar todo el flujo de información que maneja la aplicación. Cada *Manager* tiene un dominio de responsabilidad bien definido, acorde a su funcionalidad y tipo de información que maneja.
Todos los *Managers* tienen asociada su propia base de datos. Son capaces de acceder, consultar y modificar la información únicamente de su propia base de datos. Se debe generar una petición a otro *Manager* en caso de necesitar su información.

- *Service Layer*
  La capa de servicio es la encargada de traducir requerimientos de alto nivel a acciones de control en el plano de datos. Brinda servicios a *Manager Layer* para que la comunicación con el plano de datos sea posible.
  Los *Manager* se suscriben a eventos de estos servicios, esperando que un evento se dispare para informar actualizaciones del plano de datos.
  A diferencia de los *Manager*, almacenan información de bajo nivel. Los tipos de datos que administran son orientados a información dinámica con validez local en cada *PoP*.
  Es la capa más baja de la aplicación, por lo tanto es quien utiliza directamente servicios del *CORE* de *ONOS*.

Antes de realizar una descripción de los servicios expuestos por cada capa, resulta conveniente definir los tipos de comunicación entre subsistemas de la aplicación. Se definen tres tipos de comunicación posible:

- Comunicación horizontal
  Esta comunicación se genera entre bloques o subsistemas de una misma capa. Subsistemas de una misma capa no pueden tomar acciones sobre datos administrados por otros. Sí pueden realizar peticiones para comenzar una acción o consultas sobre determinada información. Esta comunicación es bidireccional.
- Comunicación descendente
  Esta comunicación es unidireccional. Comienza desde un bloque de una capa superior hacia un bloque de una capa inferior. Bloques de capas superiores tienen la capacidad de tomar acciones y ordenar a bloques de capas inferiores que realicen alguna acción en base al conjunto de servicios que exponen.
- Comunicación ascendente
  Esta comunicación es unidireccional. Las comunicaciones ascendentes se generan exclusivamente como respuesta, ya sea en respuesta a una petición o el anuncio de un evento.

Vale la pena destacar que la comunicación descendente y ascendente está permitida únicamente entre capas adyacentes.

A continuación se describen los subsistemas que componen cada capa.

5.2.1. Gestores de la aplicación

5.2.1.a. Topology Manager

Este bloque es el encargado de gestionar la topología de la red. Administra información de puntos de presencia que pertenecen a la misma, así como sus parámetros relevantes. Vale la pena destacar que este subsistema es independiente del subsistema "topology" instanciado en el CORE de ONOS debido a que este solo brinda información de la topología de la red si existen conexiones entre switches en capa 2, en cambio *Topology Manager* brinda información de la topología de una red sobrepuesta.

Mediante información externa es posible configurar la topología de la *ON*, es decir, qué *PoPs* están directamente conectados por un *Link*.

En el caso del presente proyecto, se asume que la red subyacente a la *ON* es Internet, por lo cual se admite una topología *full Mesh*. Esto implica que todos los *PoPs* se encuentran directamente conectados. Sin embargo, este *Manager* tiene la capacidad de imponer que *Links* están permitidos. De esta forma, *ONRApp* puede usarse imponiendo otra topología en la *ON*.

En la figura 5.2 se muestran los tipos de datos relevantes que administra *Topology Manager*:

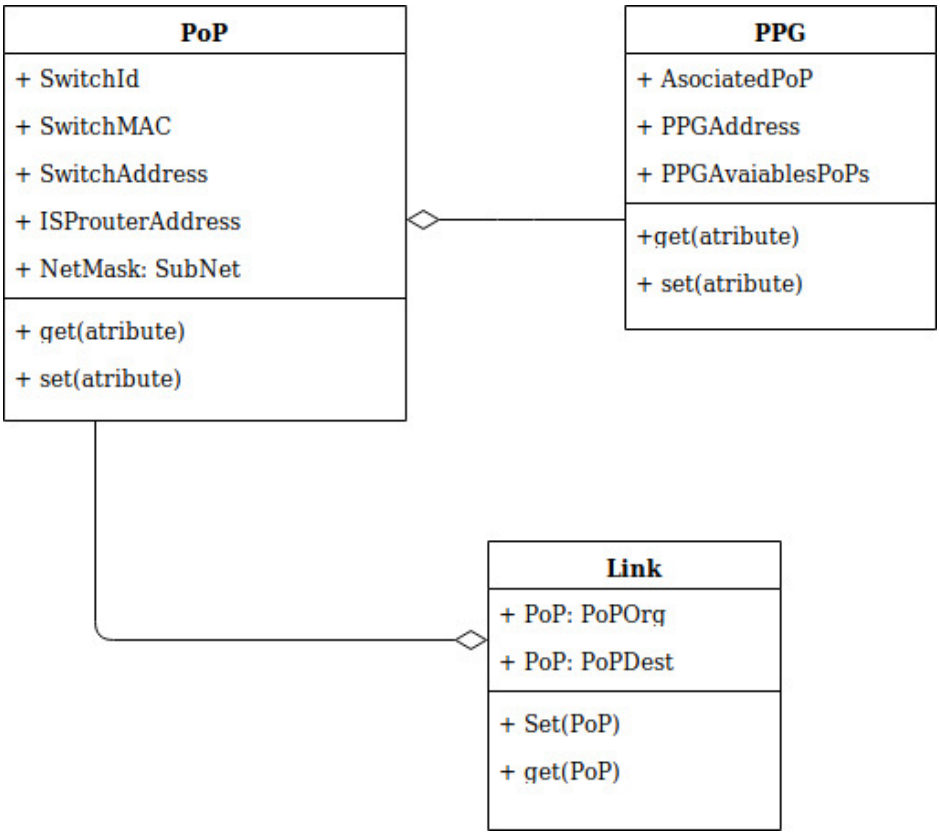


Figura 5.2: Representación *UML* del tipo de dato que administra *Topology Manager*

En la figura 5.2 se muestran los tipos de datos relevantes que administra *Topology Manager*.

La clase *PoP* está compuesta principalmente por los atributos mencionados en 3.1.2, necesarios para lograr implementar el algoritmo de encaminamiento.

Los atributos del *PoP* son los siguientes:

- SwitchId: Todo *Switch OpenFlow* tiene asociado un identificador el cual es utilizado por el controlador para comunicarse con el switch.
- SwitchMAC: Es la MAC a la cual el *switch* responde y envía mensajes ARP. La utilidad de este parámetro será explicada en 7.
- SwitchAddress: Es la dirección IP a la cual un *Switch OF* perteneciente a un *PoP* responde. La necesidad de esta dirección IP se explica en 3.2
- ISPRouterAddress: Es la dirección IP del router del *ISP*.
- NetMask: Subred asociada al *PoP*.
- PPG: Como se menciona en 3.1, todo *PoP* tiene asociado un *PPG*. La clase *PPG* contiene información útil para el sistema de medición.

Los atributos de un *Link* son dos *PoPs*, uno de origen y otro de destino. Como los *Links* son direccionales igual que los *ONRPolicies*, se crean dos *Link* por cada par de *PoPs*.

5.2.1.b. Overlay Network Routing Policies(ONRP) Manager

Este subsistema es el encargado de administrar toda la información de encaminamiento de alto nivel que existe en la *ON*. Es quien recibe peticiones de políticas de encaminamiento de tráfico de capas superiores e inicia las acciones pertinentes.

Todas las peticiones de encaminamiento de tráfico en la *ON* comienzan en este bloque. Es el único bloque que conoce en tiempo real el estado de cada *ONRPolicy*. Gestiona el *pool* de asignaciones de identificadores de cada *ONRP*.

Cada *ONRPolicy* está compuesto por el siguiente pool de identificadores:

- *Global ONRP Id*
  Identificador único para cada *ONRPolicy* creado. Este identificador no está estrechamente relacionado con el algoritmo de encaminamiento diseñado detallado en la sección 3.2. Sin embargo, tiene funcionalidades importantes en la implementación, por ejemplo, para optimización de búsquedas.
- Conjunto de *Local ONRPolicy Id*
  Un *ONRPolicy* compuesto por un *Path* de largo k, cuenta con k identificadores locales, tal como se explica en la sección 3.2.

Resulta evidente que el tipo de dato central con el que trabaja *ONRPolicies Manager* es la clase *ONRPolicy*.

En la figura 5.3 se ilustra en UML la representación de este tipo de datos.

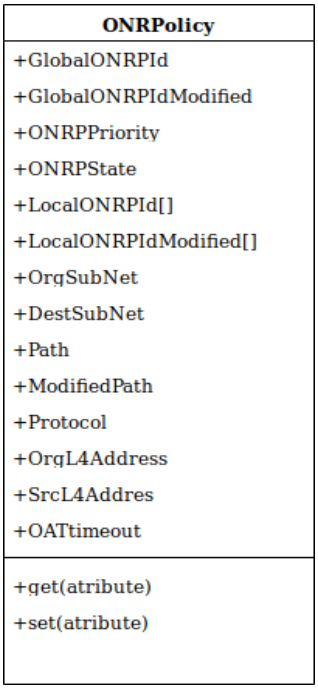


Figura 5.3: Representación *UML* del tipo de dato que administra *ONRP Manager*

En la figura 5.3 puede identificarse claramente los atributos ya mencionados en la sección 3.2. Sin embargo aparecen algunos atributos que no están presentes en la construcción teórica de una *ONRP* para lograr el encaminamiento de flujo sobre una *ON*. Estos son:

- LocalONRPModified, ModifiedPath, ModifiedGlobalONRP
  Estos parámetros están destinados exclusivamente a la función *Modificar ONRP*. Esta funcionalidad es descrita en la sección 7.
- *Overlay Network Address Traslation (ONAT) timeout*
  Este valor hace referencia al tiempo de ocio máximo de una *flow entry* vinculada a un flujo a partir del cual la *flow entry* desaparece. Se define tiempo de ocio de una *flow entry* como el período en el cual el *Switch OpenFlow* no encuentra ningún *match* en dicha *flow entry*.
- ONRPPriority
  Como se menciona en 3.2 este parámetro brinda una gran riqueza a la implementación, por lo que se dedica particular atención en 5.2.1.b a su diseño.
- ONRPState
  Toda *ONRP* tiene un parámetro asociado que describe el estado actual del mismo. Los estados posibles son:
  - ACTIVE
  - BEING_CREATED
  - BEING_MODIFIED
  - BEING_DELETED
  - DOWN

ONRP Priority

El ejemplo planteado en la sección 3.2 evidencia la necesidad discernir entre *ONRPs* asignándoles un valor de prioridad. Para los casos particulares en que los parámetros de una *ONRP* caen en un subconjunto específico de los parámetros de otra, esto se vuelve indispensable.

Para comprender la necesidad de este valor, considere los siguientes casos:

1. El primer caso ocurre al realizar las búsquedas en la bases de datos de *ONRP Manager*. La asignación de prioridad permite comparar una *ONRP* únicamente con otras de su misma prioridad, minimizando los tiempos de búsqueda necesarios en el controlador. De no existir la asignación de prioridad, al momento de crear una nueva *ONRP*, se debería iterar en todos las *ONRP* creadas y realizar comparaciones en todos sus atributos para evaluar por ejemplo, si ya existe en la base de datos.

2. Las *flow entries* de un *Switch OpenFlow* tienen una prioridad configurable. En caso de no especificar este valor correctamente, podría suceder que las *flow entries* de dos *ONRP* diferentes cuyos *match* coinciden, presenten la misma prioridad y por consiguiente, según la especificación de *OpenFlow* el orden queda criterio del proveedor del *switch*. [10]

Si bien en esta versión *ONRApp* solo contempla encaminamiento de flujos para tráfico TCP y UDP debido a que hasta hasta el momento *ONOS* solo permite el tratamiento de estos protocolos de capa de transporte, la solución debe contemplar la posibilidad de la inclusión de nuevos protocolos en versiones futuras.

Con el objetivo de mitigar los problemas planteados, se diseña un algoritmo *Priority Algorithm*, el cual se explica a continuación:

Dada una *ONRP* genérica:

$$ONRP = \{Subnet_{origen}, Subnet_{destino}, protocolo, Port_{src}, Port_{dst}, Path\}$$

X_1 = Largo de la mascara en la subred de origen
X_2 = Largo de la mascara en la subred de destino
X_3 = Vale 1 si el protocolo de capa transporte esta especificado sino 0
X_4 = Vale 1 si el puerto origen esta especificado sino 0
X_5 = Vale 1 si el puerto destino esta especificado sino 0

$$\Rightarrow ONRP_{Priority} = \{X_1 + X_2 * MaxNetMaskLenght + MaxNetMaskLenght^2(4X_4 + 2X_5 + X_3)$$

Donde:

$$MaxNetMaskLength = 32 \quad X_1, X_2 \in \{1, .., 32\} \quad X_3, X_4, X5 \in \{0, 1\} \tag{5.1}$$

Como puede observarse en la ecuación 5.1, la prioridad aumenta a medida que especificamos puerto de origen y destino, y algún tipo de protocolo de capa de transporte. Esto lo hace escalable a agregar nuevos protocolos al encaminamiento de flujo.

Tomemos el ejemplo de 3.2 en la cual se especifican las siguientes *ONRP*:

Tabla 5.1: Cálculo de $ONRP_{Prority}$ para distintas *ONRPs*

# ONRPolicy	1	2	3	4
Red origen	172.16.1.8/32	172.16.1.0/24	172.16.2.0/24	172.16.1.0/24
Red destino	172.16.3.10/32	172.16.4.0/28	172.16.4.4/32	172.16.3.0/24
Protocolo	UDP	*	TCP	*
Puerto L4 origen	15194	*	*	*
Puerto L4 destino	15193	*	80	*
Path	[PoP1,PoP2,PoP3]	[PoP1,PoP2,PoP4]	[PoP2,PoP1,PoP4]	[PoP1, PoP2, PoP4, PoP3]
Priority	8224	920	4120	792

En la tabla 5.1 queda claro que si bien el $ONRP_1$ es un subconjunto del $ONRP_4$, presenta mayor prioridad. Este algoritmo resuelve los problemas planteados, logrando discernir entre *ONRPs* con facilidad.

5.2.1.c. Measure Manager

Este bloque es el encargado de orquestar todo el flujo de información vinculado con la medición de QoS en la *ON*.

Es capaz de interpretar peticiones de medición de QoS de capas superiores, comunicarse con *ONRPolicies Manager* para encaminar el tráfico inyectado por los *PPG* desde cada *PoP*. Utiliza servicios de *Service Layer* para intercambiar información con los *PPG*.

La base de datos de *Measure Manager* contiene información del resultado de medidas realizadas, métrica utilizada en cada medida y entidades relacionadas en la misma, los *PPG*.

Vale la pena destacar que en esta versión de *ONRApp*, si bien su estructura está diseñada para escalar a nuevos métodos, el sistema de medición compuesto por *Measure Manager* y *PPG* solo contempla la métrica *RTT*.

5.2.1.d. FlowEntry Manager

Es el encargado de traducir toda la información de los otros *Managers* en *flow entries* e instalarlas en los *switchs*. Además, procesa los *PACKET_IN* y en caso de ser necesario genera una respuesta con un *PACKET_OUT*.

Hace uso de *ONRAFlowServices* para instalar nuevas *flow entries* y *CentralizedARP* para procesar los mensajes ARP. Por lo tanto, es el *Manager* que opera más cerca del plano de datos.

Cuando *Topology Manager* registra un *PoP* le solicita a *FlowEntry Manager* que inicialice el *switch* asociado, configurando las reglas básicas. Le agrega las *flow entries* necesarias en la primer *flow table* del *pipeline* con el objetivo de que los paquetes de interés para la aplicación sean enviados al controlador correspondiente para ser procesados. En caso contrario seguirán su destino sin ser modificados. Todo el manejo de *flow entry* y *flow tables* se describe en profundidad en el capítulo 7.

ONRPolicies Manager le indica cuándo debe crear, borrar o modificar una *ONRP*, e instala las *flow entries* en los *switch* según las características de la misma.

El tipo de dato de mayor importancia que administra *FlowEntry Manager* es la clase *FlowRule* de ONOS. Esta clase es una representación de alto nivel de las *flow entries*, y por lo tanto se tiene una correspondencia entre las *Flow Rule* almacenadas por este *Manager* y las *flow entries* instaladas por los *Switch OpenFlow*.

5.2.2. Servicios de la aplicación

5.2.2.a. UDP Service

Este bloque es el encargado de traducir mensajes hacia el plano de datos generando tráfico UDP. Gestiona estado de *sockets*, threads para paralelizar distintas comunicaciones, etc. Sus servicios son utilizados por *Measure Manager* para comunicarse con los *PPG*.

5.2.2.b. Flow Service

Este servicio es una simplificación del servicio *FlowRuleService* de *ONOS*. El servicio brindado por *ONOS* permite crear cualquier tipo de *flow entry*, con la complejidad que esto implica. Este servicio exige que la *FlowRule* siga determinadas reglas, lo que permite filtrar potenciales errores al crear una *flow entry*.

5.2.2.c. CentralizedARP Service

Este bloque brinda funcionalidades para extraer información del plano de datos mediante el envío y recepción de mensajes ARP de tipo *request* y *reply*. Cuenta con un bloque de almacenamiento local el cual contiene información recolectada a través de mensajes ARP. Este almacenamiento es el análogo a la tabla ARP bajo el paradigma *SDN*, ya que contiene información de la asociación IP-MAC de todos los *host* de cada *PoP* de la *ON* que fueron consultados por intermedio del protocolo ARP, incluyendo el router del *ISP*. La información de esta base de datos se mantiene actualizada, enviando *ARP Request* a los hosts cada determinado período, y en caso de notar algún cambio en la MAC la información de la tabla ARP centralizada se actualiza automáticamente.

En caso de notar un cambio en la MAC de router del *ISP*, actualiza su almacenamiento y genera un evento hacia *FlowEntry Manager* para actualizar la *flow entry* en el *Switch Open Flow* correspondiente, permitiendo que el encaminamiento en la *ON* no se vea degradado ante el posible dinamismo de la red subyacente. Los servicios brindados por este bloque son relevantes para asegurar que todo el encaminamiento de tráfico sobre la *ON* funcione automáticamente.

Dada su importancia, en 7 se explica la implementación para que un *Switch OpenFlow* realice y responda consultas ARP.

Si bien con la descripción abordada queda totalmente definida la funcionalidad que agrega cada subsistema en la aplicación, para brindar servicios a entidades de capa de aplicación y comunicarse con el plano de datos, resulta indispensable establecer interfaces de comunicación con elementos externos. A continuación se detalla como *ONRApp* se comunica con el exterior.

5.2.3. Interacciones externas

En esta sección se exponen los dos tipos de interacciónes externas que posee la aplicación *ONRApp*. La figura 5.10 expone la arquitectura completa, incluyendo las interacciónes con los servicios del *CORE* de ONOS y las interacciónes por *REST API*.

Se presenta la interacción con ONOS, describiendo los servicios de *CORE* propios del controlador que fueron necesarios para el desarrollo de *ONRApp*.

Posteriormente, se presentan las principales funcionalidades que se exponen a traves de *REST API* que permiten el control de políticas de encaminamiento y el sistema de medición de QoS.

5.2.3.a. Servicios del *CORE* de *ONOS* utilizados

Core Service

Como su nombre lo indica, permite comunicarse con el *CORE* de *ONOS* para registrar una nueva aplicación. Una vez registrada, *Core Service* habilita a la aplicación a utilizar otros servicios del *CORE* del controlador.

Device Service

Brinda información del estado de los *switch* en la red. *Topology Manager* se suscribe a eventos de este servicio para conocer si un *switch* mantiene establecido el canal de comunicación con el controlador, mientras que *FlowEntry Manger* lo utiliza para conocer las interfaces del *switch* y en base a esta información crear las *flow entries* correspondientes.

Host Service

Este servicio obtiene la información de los host conectados a cada uno de los *switchs*. A este bloque se le consulta mediante *FlowEntry Manager* para conocer las MACs de los routers de borde de los *ISP* y ahorrarse el tiempo de usar el servicio de c*entralized ARP*.

Cabe destacar que la necesidad de desarrollar *centralizedARP* radica en que *HostService* no realiza consultas ARP sino que obtiene la información de los host a partir de los paquetes que estos envían. A modo de ejemplo, en caso que un host no inicie comunicaciones, es necesario realizar una consulta ARP para tener información sobre su MAC.

Packet Service

Uno de los mensajes *OpenFlow* más utilizados por ONRApp son los *PACKET_IN*. Este servicio es el encargado de recibir los *PACKET_IN* desde el plano de datos, los analiza y envía estos al bloque correspondiente como por ejemplo a *CentralizedARP* en caso de ser un mensaje ARP para que sea procesado, y si es un mensaje UDP o TCP y cumple con determinadas características, lo envía a *FlowEntry Manager* para que instale la *flow entry* correspendiente.

FlowRule Service

Este es un servicio indispensable que brinda el controlador *ONOS* a *ONRApp*. Es utilizado para que las *flow entries* diseñadas por *FlowEntry Manager* sean instaladas o borradas correctamente en los switchs usando el protocolo de *OpenFlow*. Además, genera eventos relacionados con estas *flows*.

Una funcionalidad a destacar es que este servicio no se limita instalar las *flow entries*, sino que verifica que estén correctamente instaladas. Por ejemplo, si un switch falla y se reinicia completamente, este servicio reinstala todas las *flow entries* sin que la necesidad de que*FlowEntry Manager* tenga que participar, evitando la sobrecarga y pérdida de tiempo que generaría enviar esta información hasta la capa de administración para ser procesada y posteriormente tomar acciones.

5.2.3.b. Funcionalidades expuestas hacia entes externos

En esta sección se muestran algunos de los servicios de red expuestos por *ONRApp* hacia capa de aplicación.

Como se menciona en este capítulo, la forma de exponer estos servicios es por intermedio de *REST API*.

En esta primera versión de la aplicación, solo se contempla en métodos *POST* el pasaje de información a partir de archivos *.JSON*. Para que esto sea posible, se diseña un traductor *JSON traslator* el cual recibe información en formato *JSON* y lo traduce a tipos de objetos que administran los *Managers*. La *API* diseñada para acceder a los recursos expuestos, sintaxis de los archivos *.JSON* y código de errores son descriptos en los anexos C y D.

Establecimiento y consultas de políticas de ruteo

Una de las funciones de red de mayor relevancia que expone la aplicación es la capacidad de implementar políticas de ruteo, generando rutas en el plano de datos de forma tal que el proceso de establecimiento sea ocultado a la capa de aplicación.

La capacidad que brinda *ONRApp* en cuanto a la generación de rutas se reduce a 3 funciones posibles:

- Implementación de una nueva *ONRP*:
  Esta función recibe los parámetros que definen una política de ruteo en una *ON*, y en base a estos crea una nueva ruta en el plano de datos. Esta función solo aplica si la política de ruteo no está establecida aún en el plano de datos. En caso que la sintaxis sea correcta y no se detecten problemas, devuelve un identificador único asociado a una *ONRP*. La única forma que tienen las aplicaciones externas de consultar parámetros de la política una vez esta esté implementada es mediante la utilización de este identificador. Un parámetro de interés puede ser el estado de una política, la cual por ejemplo puede estar siendo creada, activa o desactivada si se detectó un problema en alguno de los puntos de presencia pertenecientes al *Path*.
- Borrado de una *ONRP*:
  Esta función solo puede ser aplicada si la política ya está implementada en el plano de datos. Recibe como entrada el identificador global asociado en el momento de creación. Sirve para solicitar el proceso de borrado de una política implementada en la red.
- Modificación de una *ONRP*:
  Esta función solo puede ser aplicada si la política ya está implementada en el plano de datos. Recibe como entrada el identificador global asociado en el momento de creación y el nuevo *Path* por donde se quiere encaminar el tráfico. Si bien el proceso de modificación de un *Path* podría generarse a partir de un borrado de la *ONRP* y una nueva creación, la implementación de esta funcionalidad reduce la pérdida de paquetes generada en el transitorio de modificación de un *Path*.

Para mayor entendimiento de la dinámica generada ante estas peticiones y detalles de la mejora en la QoS al modificar una *ONRP* en lugar de crearla nuevamente, dirigirse a la sección 5.3 y al capítulo 7 respectivamente.

Funcionalidades exclusivamente orientadas a consulta de recursos asociados a políticas de ruteo implementadas son detalladas en el anexo E.

Medición de QoS

Otra de las funcionalidades principales de *ONRApp* es que permite realizar medidas de *QoS* en una red sobrepuesta de forma centralizada.

Estas acciones se ejectuan partir de peticiones de agentes externos como pueden ser aplicaciones de monitoreo e ingeniería de tráfico, ocultando el proceso de establecimiento de rutas y control del conjunto distribuido perteneciente al sistema de medición: Los *PPG*.

El procedimiento de medición consta de dos etapas las cuales son expuestas como métodos independientes:

- Inicialización de la medida:
  Resulta claro que antes de realizar una medición de QoS de un *Path* perteneciente a la *ON*, deben generarse las condiciones para que el tráfico sea encaminado por el mismo. Para esto, recordando el algoritmo de encaminamiento propuesto en 3.2 y el sistema de medición presentado en 3.3, para inicializar la medida debe enviarse por *REST API* una solicitud con paráme-

tros asociados como por ejemplo la métrica de medición, el *Path* a medir, entre otros. Una vez que se recibe la petición y esta es validada como correcta, se devuelve a la aplicación que envió la solicitud un identificador asociado y la medida pasa a un estado de inicialización.

- Disparo de una medida: Este proceso puede ejecutarse una vez halla finalizado el proceso de inicialización, momento en el cual la medida a realizar pasa de estado de inicialización a inicializada. A partir de este momento, al recibir una petición HTTP con el Identificador asociado, se ejecuta la medida correspondiente. Luego de finalizada la ejecución, el resultado estará disponible para ser consultado a partir de la *API* definida con el identificador correspondiente.

Si bien el resultado de la medida está disponible como recurso a ser consultado desde entidades externas mediante *REST API*, como se explicó en la sección 3.3, resulta de vital importancia para aplicaciones de ingeniería de tráfico saber en que momento fueron efectuadas las mediciones. Puede que un algoritmo de ingeniería de tráfico necesite conocer en tiempo real el resultado de la medida.

Dado que la arquitectura *RESTful* no mantiene estados luego de una petición, *MonApp* debe realizar de forma intermitente consultas cada pequeños intervalos de tiempo al controlador para poder obtener lo más pronto posible el resultado de la medida.

Por este motivo, al iniciar la medida, *ONRApp* brinda la posibilidad de especificar una dirección IP y puerto de capa de transporte en los parámetros de inicialización con el objetivo de, una vez que esté disponible un resultado asociado a una medición, este pueda ser enviado inmediatamente al *socket* definido, minimizando el tiempo de espera o la sobrecarga de tráfico que podría generar el *pooling* que debe hacerse constantemente si se quisiera recibir la información a partir de consultas HTTP.

Los detalles de la dinámica de *ONRApp* en el proceso de medición son presentados en 5.3.

Registro de nuevos puntos de presencia en la red

Una funcionalidad básica que debe ofrecer la aplicación es registrar o borrar el registro de un nuevo punto de presencia. *ONRApp* además de contar con esta funcionalidad, permite realizar consultas de los parámetros de cada *PoP*, inclusive el estado actual del mismo. Esto permite conocer si los nodos de la red se encuentran operativos o el sistema detectó un problema en algún *PoP*.

Detalles de la respuesta del sistema ante la pérdida de conexión de un *PoP* son expuestos en 5.3.

Si bien en esta sección se resumen las funcionalidades más importantes expuestas por *ONRApp*, el detalle completo de la *API* se muestra en anexo C.

5.3. Análisis de dinámicas

En la presente sección se describe la dinámica de la aplicación como respuesta a eventos externos. Se expone un modelo de la secuencia lógica que sigue el conjunto central del sistema ante excitaciones externas, como la solicitud de registro de un nuevo punto de presencia en la *ON*, peticiones de implementación de nuevas políticas de ruteo, ejecución de mediciones de QoS en la red o respuesta del sistema ante detección de fallas en un punto de presencia.

5.3.1. Registro de *PoPs*

En esta sección se detalla el proceso de registro de un nuevo nodo en la *ON*. En la figura 5.4 se muestra en UML un diagrama de eventos que suceden al ejecutar la acción.

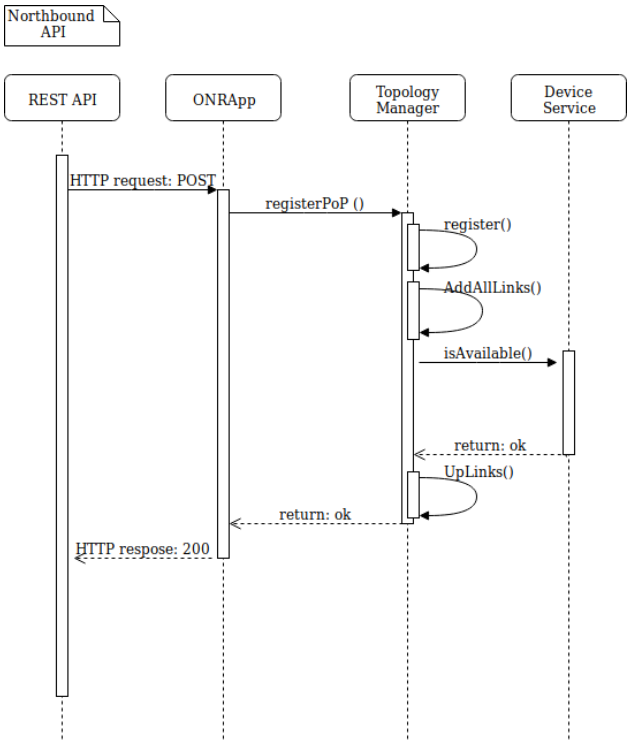


Figura 5.4: Diagrama de eventos en *ONRApp*: Registro de un nuevo *PoP*

En la figura 5.4 se muestra que la petición de registro de un nuevo nodo en la *ON* se genera a partir de un *request* HTTP. La petición es procesada por *Topology Manager*, quien se encarga de validar si es posible el registro de este nuevo punto de presencia. Recordando la sección 5.2.1.a, este subsistema se encarga de generar la topología de red a partir de los *PoPs* registrados.

Se admite el registro y se generan los *Links* asociados a todos los *PoPs* previamente registrados. Luego, le consulta a *Device Service* si este nuevo *PoP* se

encuentra operativo para poder generar rutas que pasen por el.

Una vez que *Device Service* responde que este nuevo nodo se encuentra en condiciones, se dan de alta los *Links* permitiendo su utilización en la implementación de futuras políticas.

5.3.2. Creación de políticas de ruteo

A continuación se presenta un diagrama de la secuencia que sigue la aplicación al recibir una petición de una nueva *ONRP*.

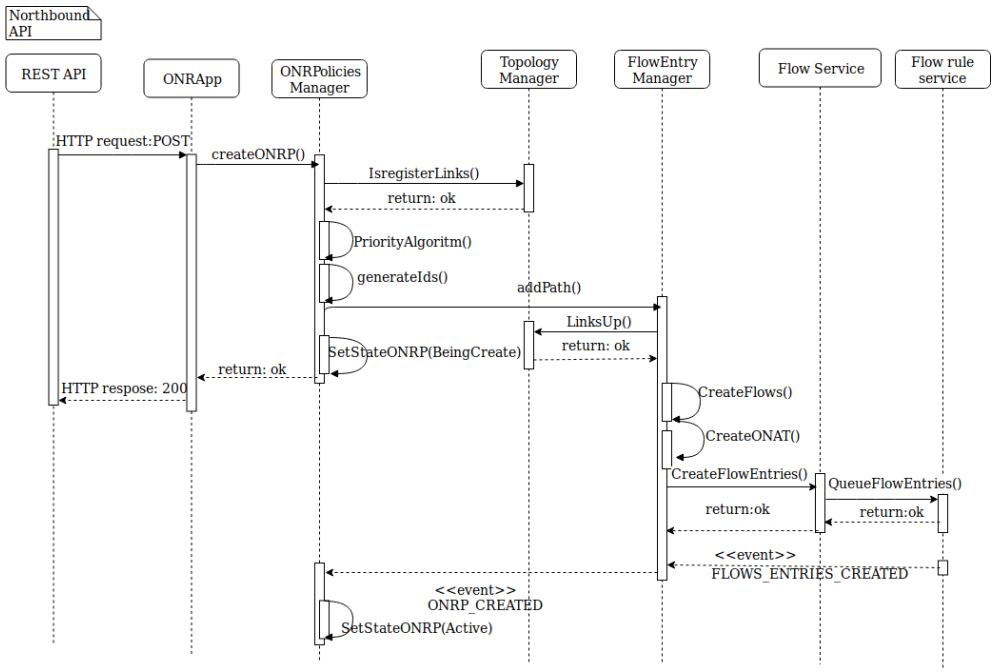


Figura 5.5: Diagrama de eventos en ONRApp: Creación de una *ONRP*

En la figura 5.5 se aprecia que, luego de un *request* HTTP en el cual llega una petición de implementación de una nueva política de ruteo, se ejecuta el método *createONRP()*, dando inicio a la secuencia.

Una vez que *ONRP Manager* recibe la petición de crear una *ONRP*, le consulta a *Topology Manager* si todos los *Links* del *Path* se encuentran registrados. *Topology Manager* informa que el *Path* es válido, *ONRP Manager* genera los identificadores mencionados en 3.2 y 5.2.1 y su prioridad asociada ejecutando el *Priority Algorithm*.

Luego que la política de alto nivel se encuentra creada, le solicita a *FlowEntry Manager* que programe el plano de datos, instalando las *flow entries* necesarias.

En el diagrama se refleja claramente el uso de *threads* en *ONRApp* para mejorar el rendimiento de la aplicación. En este punto, *ONRP Manager* abre un *thread* para ejecutar las tareas de *FlowEntry Manager* y continúa su ejecución. *ONRP*

Manager configura el estado del *ONRP* e informa a la aplicación externa que la política se encuentra en proceso de creación.

FlowEntry Manager antes de comenzar la ejecución de creación en el plano de datos, consulta a *Topology Manager* si todos los *Links* están activos. Luego, procesa y almacena la información de *flow entries* y *ONAT* a instalar.

Mediante el uso de *Flow Service* se crean las *flow entries* y se solicita a *FlowRule Service* que sean encoladas, para su posterior configuración en los *Switch OpenFlow* de cada *PoP*.

Una vez que *FlowEntry Manager* recibe confirmación desde los *Switches OpenFlow* que todas las *flow entries* asociadas al *Path* fueron creadas correctamente, genera un evento hacia *ONRP Manager* informando que la política de ruteo ha sido implementada correctamente en el plano de datos, momento en el cuál se modifica el estado de la *ONRP* a *ACTIVE*. La aplicación externa debe consultar a *ONRApp* para saber cuando la *ONRP* se encuentra implementada en el plano de datos.

5.3.3. Modificación de politicas de ruteo

Este servicio brindado por *ONRApp* tiene la misma finalidad que la creación de políticas de ruteo, pero presenta ventajas gracias a su implementación como se menciona en el capítulo 7.

En la figura 5.6 se muestra el diagrama de eventos que se ejecutan al llegar la petición de modificar una política de ruteo ya implementada.

Una de las diferencias apreciables entre la figura 5.5 y la figura 5.6 es que en el proceso de modificación no se ejecuta *Priority Algorithm* dado que los parámetros de la *ONRP* se mantienen constantes, modificándose únicamente el *Path*.

Si bien en la sección 7.4.4 se expone la complejidad del proceso de modificación de una *ONRP*, por simplicidad en la figura 5.6, a nivel de *FlowEntry Manager* se muestra un proceso de borrado e instalación de nuevas *flow entries* para modificar el *Path*.

5.3.4. Inicialización de medición de QoS

Como se menciona en 5.2.3, el proceso de medición consta de dos etapas.

La primera es una etapa de inicialización que adapta el sistema para que, al momento de ejecutar las medidas, el sistema se encuentre en condiciones de llevar a cabo el proceso.

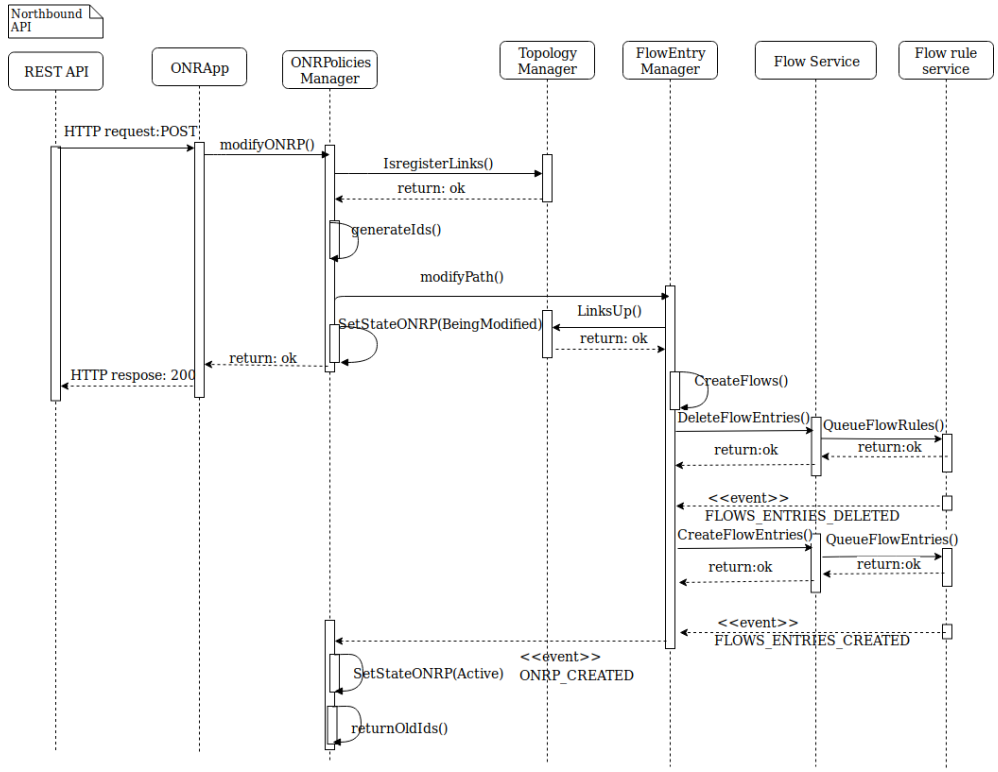


Figura 5.6: Diagrama de eventos en *ONRApp*: Modificación de una *ONRP*

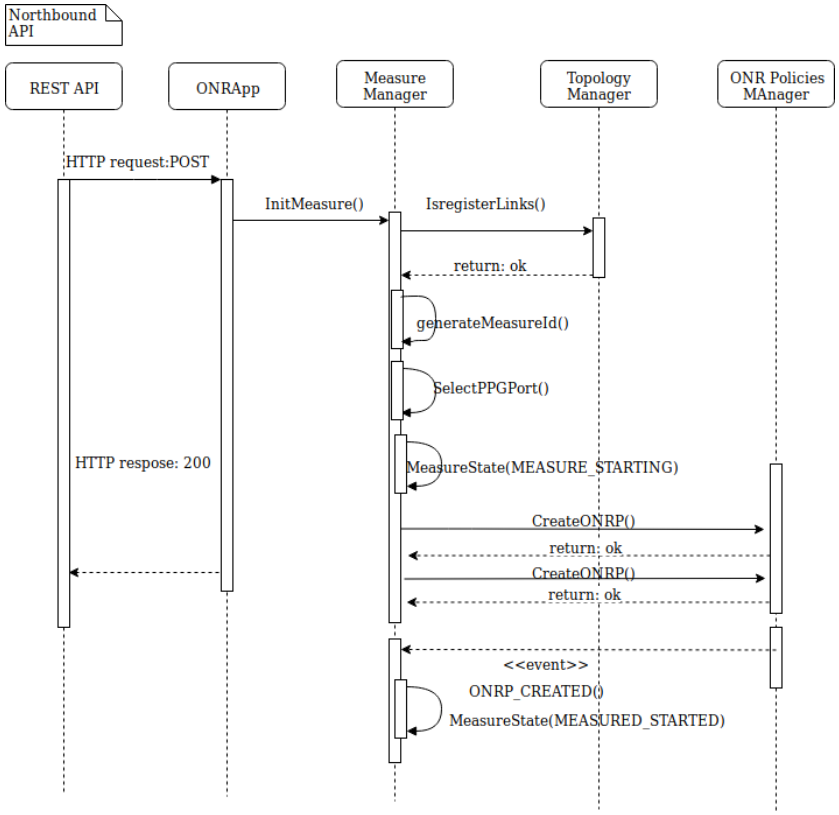


Figura 5.7: Diagrama de eventos en *ONRApp*: Inicialización de medición

En la figura 5.7 se detalla como reacciona la aplicación ante una petición HTTP para inicializar una medida. Allí se observa que la solicitud de inicialización de una nueva medida comienza por *Measure Manager*.

Measure Manager consulta a *Topology Manager* si todos los *Links* del *Path* que se desea medir están registrados en la *ON* y luego comienza a determinar y almacenar parámetros relevantes de la medida.

En particular, en el diagrama se muestra la generación de un identificador único asociado a la medida y la selección de los puertos de capa de transporte que los *PPGs* deben utilizar para establecer la comunicación.

A partir de esta información y de las direcciones IP asignadas a los *PPGs*, se solicita a *ONRP Manager* la creación de dos políticas de ruteo. Luego, se actualiza el estado de la medida a *MEASURE_STARTING*.

La petición de creación de dos nuevas políticas hacia *ONRP Manager* se debe a que, para realizar la medida es necesario establecer una *ONRP* para el envío del tráfico de medición y otra para el retorno de la respuesta desde el *PPG* destino.

Una vez que ambas *ONRPs* son establecidas en el plano de datos, se genera un evento hacia *Measure Manager*, quien actualiza el estado de la medida a *MEASURE_STARTED*.

En este estado se habilita la ejecución de mediciones de QoS para el *Path* solicitado.

5.3.5. Medición de QoS

A continuación se detalla el comportamiento del sistema al recibir una petición de ejecución de una medición que ha sido previamente inicializada.

En la figura 5.8 se observa que, una vez que *Measure Manager* recibe la petición de ejecución de una medida, se verifica que esta halla sido inicializada. Esto es cierto si el *Measure Id* que contiene la solicitud de medición fue asignado.

A partir del *Measure Id* de la solicitud, *Measure Manager* utiliza servicios de *UDP Service* para solicitar al *PPG* de origen que dé inicio al proceso de medición. Luego, responde al ente externo que realizó la solicitud que la medida ha pasado a estado *MEASURING*.

Una vez que el proceso finaliza, el *PPG* de origen envía al controlador el resultado. En el momento que este resultado llega, se genera un evento hacia *Measure Manager* de forma que este lo procese y almacene la información en su base de datos.

Measure Manager implementa un método agregado que permite el registro de entidades externas a través de la especificación de una dirección IP y un puerto de capa de transporte. El método implementado permite enviar los resultados de las mediciones, en el momento que es recibido desde el *PPG*. Esto evita la necesidad de realizar un *polling* continuo del estado de la medición. Este hecho es apreciable en la figura con el evento final destinado al *LISTENER_PORT*.

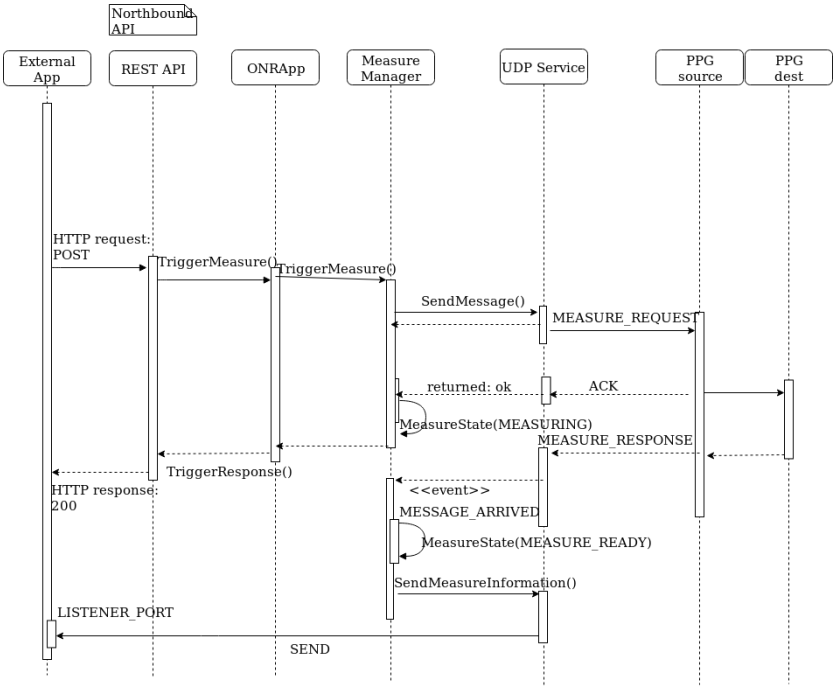


Figura 5.8: Diagrama de eventos en *ONRApp*: Ejecución de una medición

5.3.6. Toleracia a fallas de ruteo

Un parámetro relevante para cuantificar la QoS del sistema es la disponibilidad de los servicios brindados por el mismo. Un valor directamente relacionado con la disponibilidad es el tiempo de recuperación de un sistema ante una falla en algún punto de la red.

Al realizar mediciones en la red se puede cuantificar la QoS en base a una métrica, ejecutando en base al resultado obtenido determinadas decisiones. Sin embargo, como se menciona en 1, ejecutar medidas constantemente sobre todos los *Path* posibles puede impactar directamente en los recursos de la red. Por lo tanto, se deben tomar políticas de cómo y cuándo ejecutar mediciones.

Si bien los algoritmos empleados para la toma de decisiones quedan fuera del alcance de este proyecto, resulta clara la necesidad del sistema de reaccionar de forma ágil y dinámica ante catástrofes.

En esta versión de la aplicación, el único evento crítico al que el sistema responde de forma autónoma es la pérdida total de conexión con un *PoP*, considerando como tal la pérdida de conexión entre controlador y *Switch OpenFlow*.

Una vez detectada y aceptada la conexión de un *Switch OpenFlow*, el controlador mantiene una conexión TCP para poder realizar acciones sobre este. En caso que ocurra un evento inesperado a partir del cual la conexión TCP se da por finalizada, es posible suscribirse a un evento para recibir el aviso de una posible falla, por intermedio de *Device Service*.

En la figura 5.9 se presenta la respuesta del sistema a este evento. Una vez que

Device Service informa del evento a *Topology Manager*, este cambia el estado de todos los *Links* de la ON en los que interviene el *PoP* e informa a *ONRP Manager* del suceso. En este caso, todas las políticas establecidas cuyo *Path* incluye este *PoP* pasan a estado *DEACTIVATE*, es decir, se transforman en *ONRPs* inactivos. Este estado es explicado en el capitulo 7.

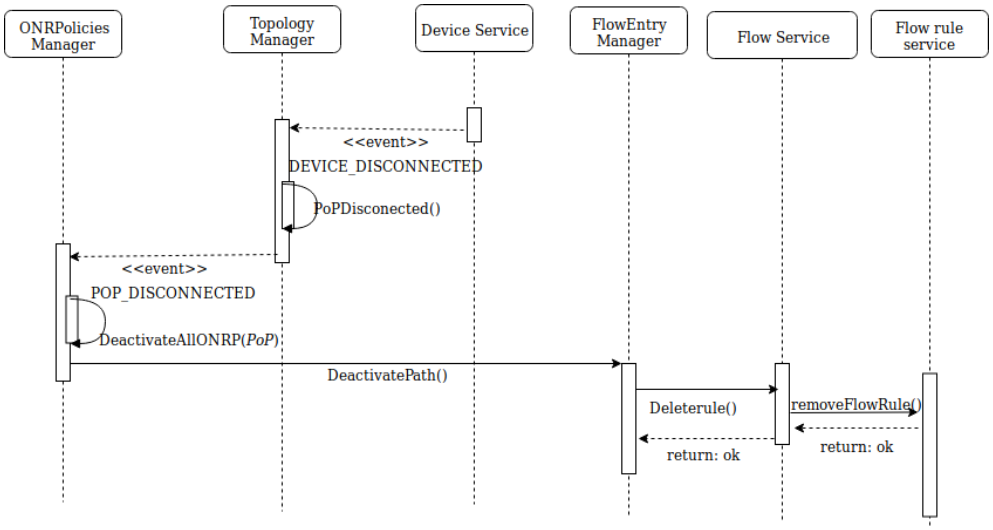


Figura 5.9: Diagrama de eventos en *ONRApp*: Respuesta a falla en conexión de un *PoP*

5.4. Resumen de la arquitectura

Durante todo este capítulo se dio a conocer el diseño de la arquitectura y funcionalidades expuestas, a demás de la dinámica del sistema como respuesta a eventos externos. A modo de resumen de la arquitectura de la aplicación y entidades externas con las que interactúa, en la figura 5.10 se muestra un diagrama indicando los bloques e interacciones ya mencionadas a lo largo del capítulo, dando por concluida la presentación formal de la aplicación implementada.

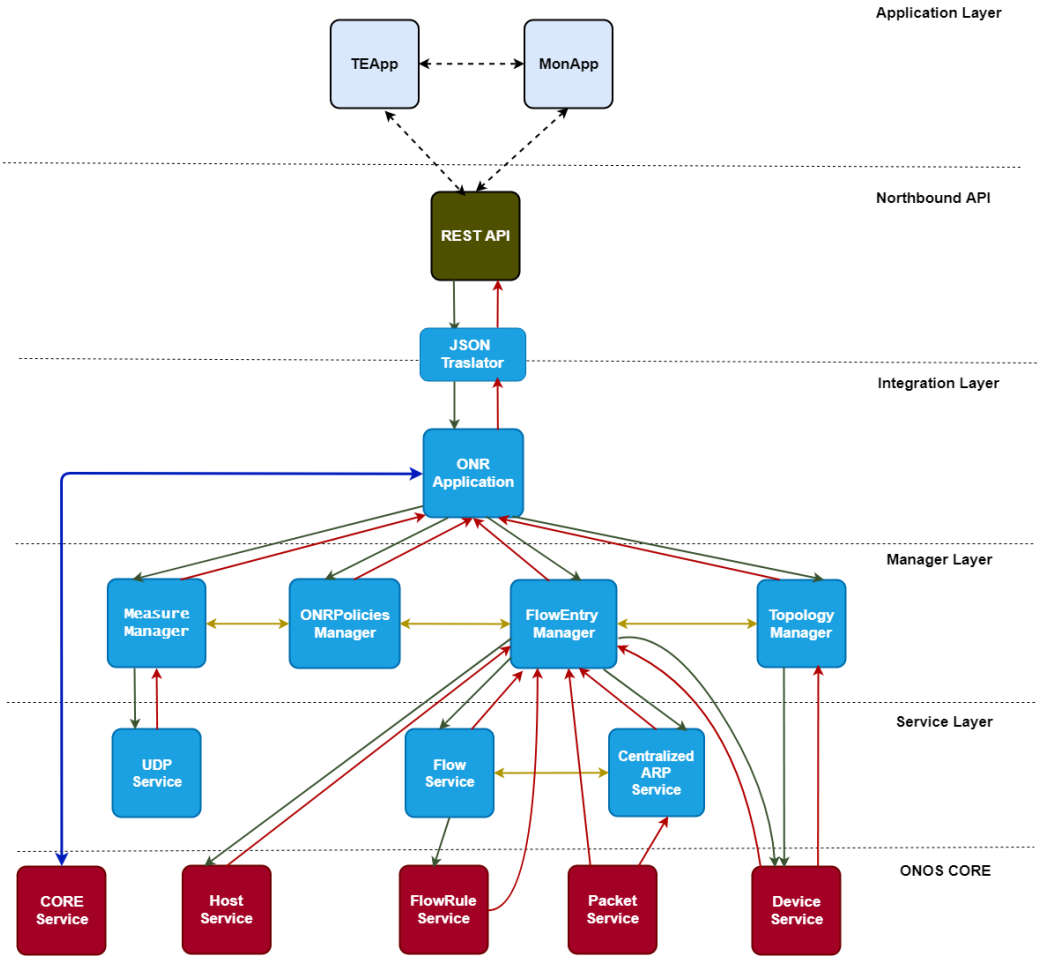


Figura 5.10: Arquitectura de *ONRApp* con vínculos al controlador y a las aplicaciones externas

Capítulo 6

PPG- Probe Packet Generator

6.1. ¿Qué es un *PPG*?

Los *PPG* son la parte distribuida del sistema de medición de QoS diseñado en este proyecto. En concreto, es una pieza de software que tiene como principal objetivo la obtención de la QoS de un *Path* de la *ON* .El software aquí presentado es la implementación de la metodología introducida en 3.3. A pesar de que en la versión presentada en este proyecto el *PPG* implementa una única técnica de medición, este se encuentra diseñado de forma que, en trabajos futuros, puedan ser incluidos criterios alternativos.

6.2. Arquitectura de software

En esta versión, el desarrollo fue implementado utilizando el lenguaje de programación *Python*, versión 2.7. El software utiliza una arquitectura orientada a la separación de objetivos y se encuentra conformada por cuatro bloques principales: *Task Manager*, *Measure Initializer*, *Receiver of probes*, *Cleaner*.

Cada bloque representa una tarea específica que son ejecutadas en paralelo gracias a la existencia de cuatro threads, cada thread destinado a cada tarea.

En la figura 6.1 se presenta un esquema de la arquitectura diseñada. Además de los cuatro bloques principales, se presentan los demás subsistemas que componen el *PPG*.

Tres de estos bloques son servicios utilizados por el resto para lograr sus respectivos objetivos individuales. El bloque restante es la implementación de la técnica de medición de QoS.

Como se mencionó, si bien esta versión únicamente implementa la técnica *RTT*, la arquitectura permite el agregado de nuevas técnicas como bloques independientes. Esto hace que el *PPG* sea un software escalable en cuanto al agregado de nuevas técnicas de medición.

El resto de esta sección presenta una descripción de cada uno de los bloques, en donde se indica los roles que cada uno posee en el funcionamiento del *PPG*.

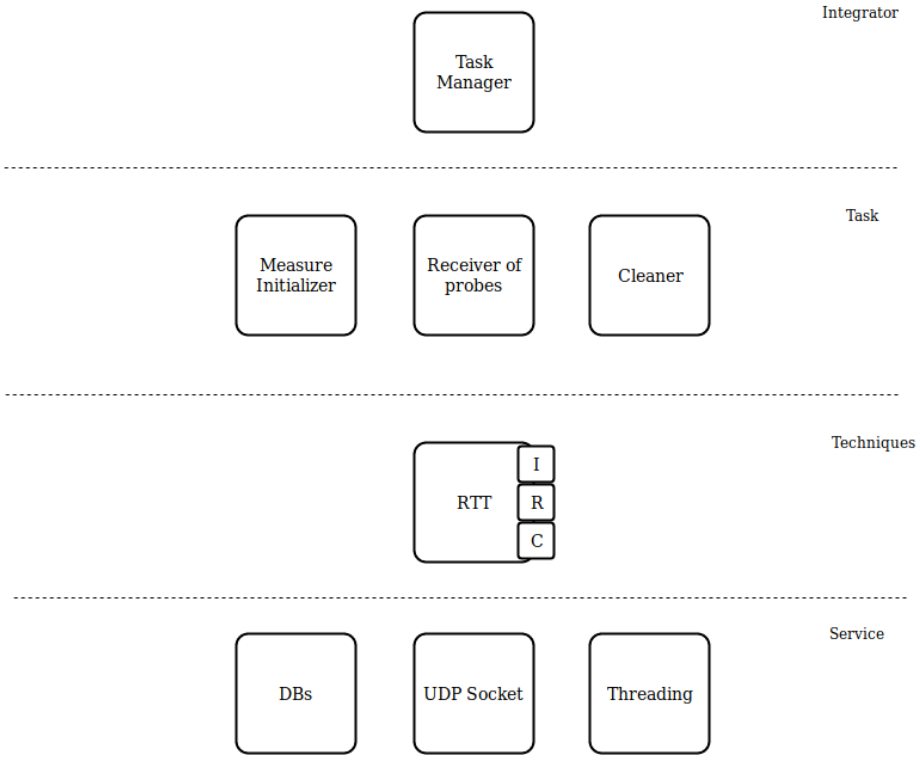


Figura 6.1: Arquitectura de software del *PPG*

6.2.1. Bloques del *PPG*

Task Manager

Este bloque es el encargado de inicializar el resto del sistema. Para ello, utiliza las funcionalidades brindadas por el servicio *Threading* (bibloteca estándar de *Python*) para solicitar la ejecución paralela de las tres tareas principales al sistema operativo donde se ejecute el *PPG*. Una vez que inicializa las tareas y el sistema se encuentra en régimen, este bloque pasa a un estado de espera, en el que escucha una orden de apagado del administrador.

En caso de solicitud de apagado, este bloque se encarga de terminar las tareas creadas anteriormente y posteriormente el *PPG* se apaga.

Measure Initializer Task

Una vez que *Task Manager* inicia este bloque, se realiza la lectura desde un archivo de configuración. Para este bloque, el archivo de configuración contiene información de la dirección IP desde la cual se puede controlar esta instancia de *PPG* y del puerto en el que debe escuchar los mensajes de control. Haciendo uso de esta información y del servicio *Socket* (bibloteca estándar de *Python*), este bloque se encarga de recibir instrucciones de control, así como de inicializar las medidas indicadas en estas instrucciones.

Recordar que, como fue explicado en 3.3, las instrucciones que este bloque debe procesar incluyen parámetros que permiten la elección del *Path* a medir (IP del

PPG destino y puertos de capa de transporte de origen y destino) y parámetros asociados a la técnica de medición que debe utilizar. En particular, la dirección IP de destino en conjunto con el puerto origen de capa de transporte son utilizados para seleccionar el *Path* a medir, permitiendo la medida simultanea tanto hacia distintos destinos, como a hacia un mismo destino por diferentes *Paths*.

Finalmente, según los parámetros de las técnicas de medición, este bloque distribuye las instrucciones hacia bloques específicos que implementan la técnica indicada en la instrucción.

Probes Receiver Task

Una vez que *Task Manager* inicia este bloque, se realiza una lectura desde un archivo de configuración. Haciendo uso del servicio *Socket* se inicializa el receptor usando los parámetros obtenidos del archivo, como por ejemplo el puerto a donde escuchar.

Una vez que se recibe un mensaje de medición (*Probe*), se analiza el contenido para determinar el tipo de técnica. Luego se ejecuta la medida según la técnica correspondiente, en este proyecto se limita a *RTT*.

Cleaner Task

Este último bloque se encarga de consultar periódicamente los tiempos iniciales de todas las mediciones que fueron inicializadas. En caso que alguna de ellas no cumpla ciertas condiciones, se contemplan dos opciones:

1. La medida no alcanzó el máximo de reintentos, y por tanto se repite el último envío.

2. La medida alcanzó el número máximo de reintentos y por tanto se envía un aviso al controlador indicando que por alguna razón desconocida, el destino no es alcanzable.

6.2.2. Técnicas de medición de QoS

Como se observa en la figura 6.1, cada bloque de técnica de medición brinda 3 funcionalidades, cada una ejecutada por una de las tareas. Para cada una de estas, el comportamiento específico depende de la técnica asociada. De esta forma se logra que las técnicas de medición de QoS funcionen como "*Plugins*", facilitando la escalabilidad del sistema.

Funcionamiento con múltiples técnicas

Aquí se explicará cómo funcionaría el *PPG* si se implementasen múltiples tipos de técnicas de medición de QoS.

Supongamos que se agregan dos técnicas de medición de QoS, como por ejemplo *Jitter* y *Bandwidth*. En este caso, el nuevo esquema de la arquitectura se muestra en la figura 6.2.

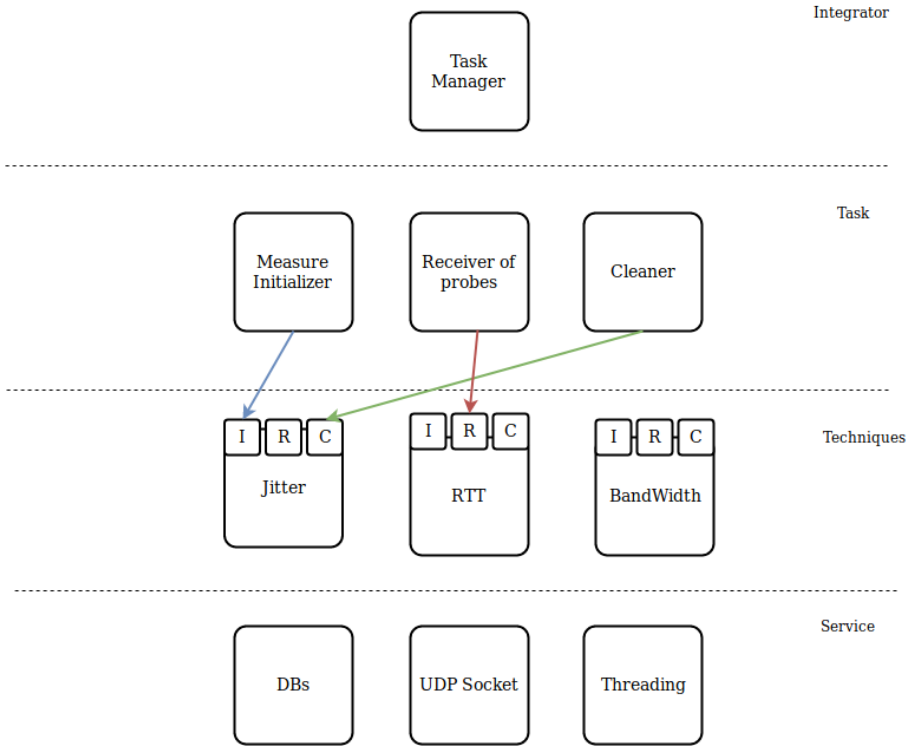


Figura 6.2: Ejemplo de funcionamiento con múltiples técnicas

En esta figura, *Measure Inicializer* se encuentra ejecutando el proceso de inicialización de una medida de la técnica *Jitter*, mientras *Cleaner* se encuentra ejecutando el proceso de limpieza de la misma métrica. Al mismo tiempo, *Probes Receiver* se encuentra recibiendo *Probes* enviados desde otro *PPG* utilizando la técnica *RTT*.

En resumen, la arquitectura diseñada permite la utilización de técnicas de medición en la forma de *plugins*, lo cual permite que el *PPG* sea una plataforma independiente de la o las técnicas implementadas.

Técnica implementada: *RTT*

Este bloque es el encargado de implementar la técnica de medición *Round-Trip Time* o *RTT*. Para ello, internamente define tres funciones que permiten inicializar medidas de *RTT*, recibir *probes RTT* y realizar la limpieza de mediciones *RTT*.

Se ha optado por la implementación de la técnica RTT frente a la técnica retardo direccional, de forma de evitar la sincronización temporal que requiere la última. Recordando que el sistema se encuentra implementado de forma de permitir la medición simultanea con diferentes destinos, esto implicaría que todos los *PPGs* deben estar sincronizados simultaneamente.

Estas son las tres funciones que debe utilizar cada una de las tareas descriptas en 6.2.1. Cada función realiza las acciones presentadas a continuación:

- RTT Measure Initializer (I)
  Almacena la solicitud de una medición originada en el controlador en una base de datos interna al bloque *RTT*. Además genera los *Probes* asociados a la medición, enviándolos hacia otro *PPG* usando *UDP Socket*.
  En los mensajes de control de inicialización de mediciones *RTT*, este bloque puede recibir los parámetros definidos a continuación. Tener presente que en la sección 6.3 se presentan los diagramas temporales asociados a la definición de cada parámetro.

  1. Parámetro *average*
     Este parámetro indica la cantidad de viajes *origen* $\rightarrow$ *destino* $\rightarrow$ *origen* que deben realizarse antes de calcular la medida.

  2. Parámetro *throughput*
     Este parámetro indica la cantidad de *Probes* a ser enviados secuencialmente, simulando un tren de mensajes contiguos.

  3. Parámetro *packet_size*
     Es un entero positivo que indica el tamaño de la carga útil del protocolo UDP a ser enviado en un *probe* medido en bytes. Si bien el valor puede ser hasta 2^{16} para entrar un mensaje IP, para evitar la fragmentación a nivel de capa de red es fuertemente recomendable utilizar un valor menor a la MTU Ethernet menos el encabezado Ethernet, encabezado IP y encabezado UDP (asumiendo que se trabaja en capa 2 sobre red Ethernet)

- RTT Probes Receiver (R)
  Como se menciona en la sección 3.3, la técnica utilizada para medir QoS de un *Path* es *RTT*. Esto implica que todo *Probe* que es enviado desde un PPG_1 hacia un PPG_2 retorna hacia PPG_1, en donde se calcula el resultado. Esto implica que, una vez inicializado y en estado de recepción, el bloque gestiona dos eventos:

  1. En caso que sea el destino de una medición (PPG_2 en el ejemplo anterior), este bloque recibe el *Probe* y lo retransmite hacia el origen (PPG_1) utilizando la dirección IP origen como destino y el puerto de origen que se encuentra indicado en el contenido del *Probe* recibido.

  2. En caso que el bloque pertenezca al origen de la medición (PPG_1 en el ejemplo anterior), el bloque evalúa los valores de promedio y *throughput* (parámetros particulares de la medida que serán explicados a continuación) y en caso de haber finalizado la medición, genera un mensaje con el resultado y lo envía a *ONRApp* en el controlador.

- RTT Cleaner (C)
  Este bloque define un *timeout* como parámetro de decisión. Se considera que un intento ha fallado cuando, en caso que se envíe un *probe RTT*, la respuesta desde el destino demore más que el tiempo predefinido en el archivo de configuración.

Con esto presente, la limpieza de mediciones RTT actúa según los siguientes criterios:

1. Si la medida no alcanzó la cantidad máxima de intentos se repite el último envío. La cantidad de intentos es un parámetro definido en el archivo de configuración antes mencionado.
2. Si la medida alcanzó el número máximo de reintentos se envía un aviso al controlador que, por alguna razón, el destino no es alcanzable.

6.3. Comunicación de *RTT Probes*

En esta sección se describirán tres formas de medir QoS que implementa el bloque RTT. Es importante comprender que las tres formas aquí presentadas resultan un subconjunto dentro de las posibilidades: los tres campos explicados en 6.2.2 pueden ser configurados en conjunto, por lo que sería posible obtener una combinación de las formas 6.3.2 y 6.3.3.

Se debe tener presente que en todos los tipos de medición que presentaremos a continuación, en caso de perder un *Probe*, luego de un tiempo determinado en el archivo de configuración, se vuelve a iniciar la medida repitiendo todos los pasos asociados a la misma.

6.3.1. Medición básica

Es la medida mínima que permite obtener el RTT del canal de comunicación. Se realiza el envío de un único *Probe* que realiza el viaje de ida y vuelta por el canal. Con respecto a los parámetros de control antes mencionados, en necesario configurarlos como sigue:
$\Rightarrow average = 1$
$\Rightarrow throughput = 1$

En la figura 6.3 se presenta el diagrama asociado a una medición de RTT básica. En la figura se remarca la diferencia de tiempo entre el envío y el retorno del *Probe* que es medido por el PPG, como el parámetro τ_m.

En este caso, se observa que el valor de RTT se puede obtener directamente usando el valor medido por los PPG:

$$RTT = \tau_m$$

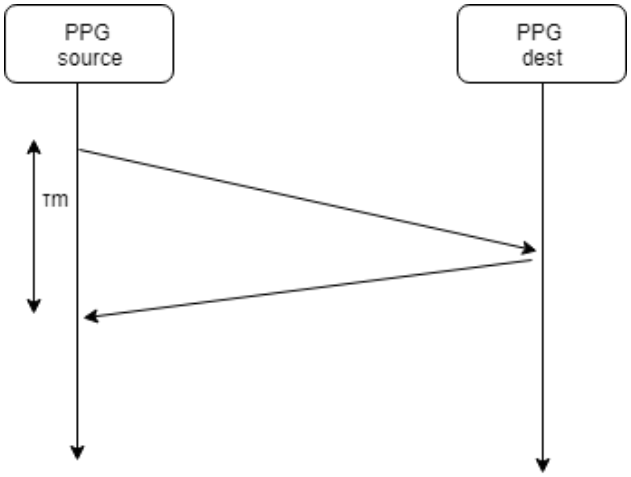


Figura 6.3: Diagrama de comunicación de un *Probe* básico

6.3.2. Medición de *RTT* con parámetro *average*

La modificación del parámetro *average* pretende un aumento en la precisión de la medida. Esto se logra imponiendo que un *Probe* transite el *Path* a medir una cantidad N de veces y midiendo el tiempo total de viaje. Con respecto a los parámetros de control antes mencionados, es necesario configurarlos como sigue:
$\Rightarrow average = N$
$\Rightarrow throughput = 1$

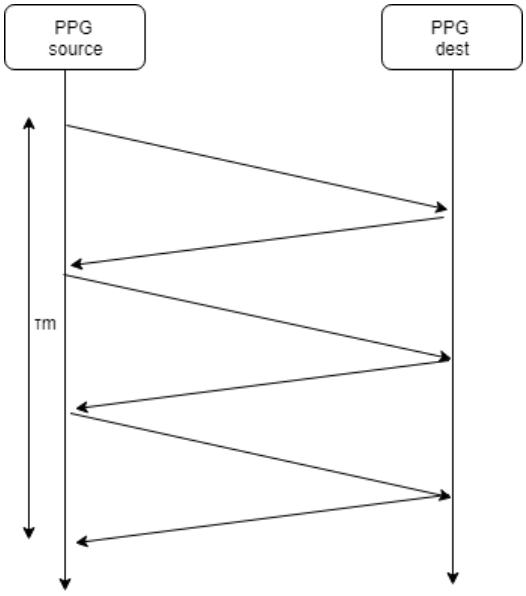


Figura 6.4: Diagrama de comunicación de un *Probe* con parámetro $average = 3$

En la figura 6.4 se presenta el diagrama asociado a una medición de este tipo.

En este caso, se observa que la obtención del *RTT medio* se debe realizar a través del siguiente cálculo:

$$RTT = \frac{\tau_m}{average} \tag{6.1}$$

La ventaja de usar *average* como parámetro es que al medir el mismo camino múltiples veces la medida se vuelve mas tolerante ante cambios efímeros de la red.

6.3.3. Medición de *RTT* con parámetro *throughput*

Como fue explicado en 6.2.2, la modificación de este parámetro implica el envío consecutivo de *Probes*. Este tipo de medida es un agregado que potencialmente permite obtener características más detalladas del canál.

Si bien el resultado calculado no se ajusta específicamente a la definición de *RTT*, la medida se realiza en base a los *RTT* de cada uno de los *Probes* enviados.

Para realizar este tipo de medida es necesario configurar los parámetros de control como sigue:
$\Rightarrow average = 1$
$\Rightarrow throughput = 3$

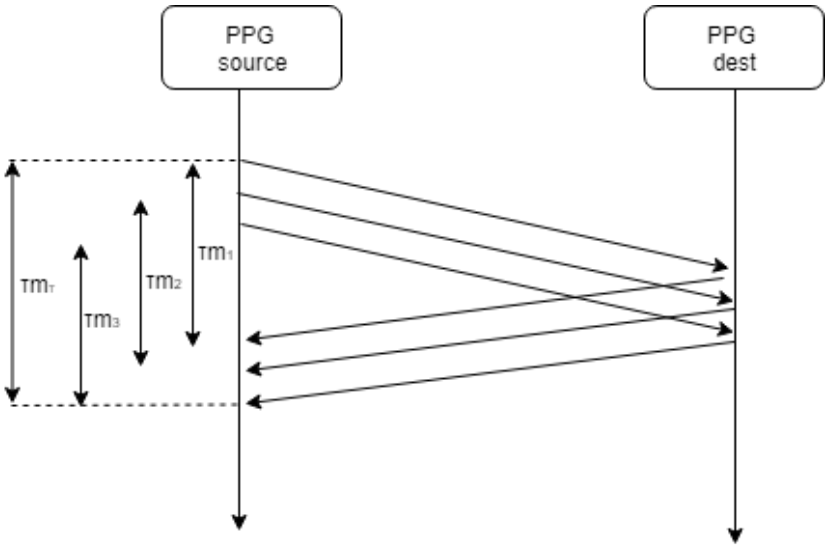


Figura 6.5: Diagrama de comunicación de un *Probe* con parámetro $throughput = 3$

En la figura 6.5 se presenta un diagrama asociado a una medición de este tipo. En este caso, se observa que el resultado calculado depende de los *N Probes* enviados. La medida en sí representa el promedio de los *RTT* de todos los *Probes* enviados. Para el calculo se plantea:

$$RTT = \frac{\sum_{i=1}^{N} \tau_i}{N} \qquad / \; N = throughput \tag{6.2}$$

En contraste con las otras formas de medida, el tiempo total de la medición no puede ser deducido a partir del resultado calculado. Con el fin de cumplir con

los requerimientos del diseño teórico planteado en 3.3, se toma el tiempo τ_{m_T} (ver figura 6.5) y se envía al controlador como otro parámetro del resultado.

La utilización de este parámetro puede permitir obtener el valor de RTT bajo condiciones de carga de la red. El parámetro permite la inyección simultanea de paquetes de medición a través de los cuales se puede simular un aumento de carga de la red.

En caso que se defina tanto *average* como *throughput*, el proceso se implementa realizando N mediciones paralelas según el parámetro *throughput* donde cada una de estas mediciones realiza M idas y vueltas según el parámetro *average*. Cada vez que termina una medición bajo *average*, se calcula el resultado parcial según la ecuación 6.1. Cuando todas las medidas bajo *average* terminan, se utiliza la formula 6.2 para obtener el resultado final a partir de todos los resultados parciales.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 7

Implementación usando *Switches OpenFlow*

Este capítulo desarrolla como se lleva a cabo la implementación en el plano de datos de los diseños teóricos expuestos en el capítulo 3 utilizando los conceptos de *Switch OpenFlow* vistos en la sección 2.3.

Se remarca que las funcionalidades aquí descriptas son administradas por el bloque *FlowEntry Manager*, introducido en la sección 5.2.1.d.

7.1. Definiciones previas

Para continuar con la sección es necesario realizar una serie de definiciones:

Switch origen
Dada una *ONRP*, es el primer switch del *Path* asociado.

Switch destino
Dado una ONRP el *switch destino* es el ultimo del *Path*.

Switch intermedio
Dada una *ONRP*, puede ser cualquier switch que no se encuentra en la primera, ni última posición del *Path* asociado.

Switch actual
Referencia al switch en donde se encuentra la *flow table* que se está especificando.

7.2. Ordenamiento de *flow tables*

Uno de los principales desafíos tratados en esta tesis fue la organización de las *flow entries* y las *flow tables*, ya que un incorrecto ordenamiento de las mismas puede llevar a un comportamiento no deseado.

7.2.1. Resumen de objetivos de las *flow tables*

La organización de las *flow tables* realizado tiene como principal enfoque la separación de objetivos. Existen cuatro tipos de objetivos. Esto implica la utilización de cuatro categorías de *flow tables* que se definen a continuación. En las primeras tres *flow tables*, su nombre corresponde a su identificador de tabla en el protocolo OpenFlow. Las Flow tables de *ONATs* las cuales se describen en esta sección, corresponden a las *Flow Table* del *Switch OpenFlow* cuyo identificador vale entre 3 y 255.

- *Flow table* 0
  Su objetivo es determinar si los paquetes entrantes corresponden a la administración de *ONRApp*. Concretamente, se determina si los mensajes son ARP o comunicaciones entre *PoPs*, que sean específicamente UDP o TCP.
- *Flow table 1*
  Los paquetes entrantes a esta tabla se originaron en el *PoP* en el que se encuentra *Switch OpenFlow* actual. En esta tabla se determina si los paquetes pertenecen a una *ONRP*, y en caso de que así sea lo envía a una *Flow tables* de *ONATs* para ser procesado.
- *Flow table* 2
  Los paquetes entrantes a esta tabla se encuentran en tránsito sobre la *ON*. Por ende, esta se encuentra asociada a los *switch intermedios y de destino*. Su principal objetivo es determinar para cada paquete a que *ONRP* pertenece. En caso de que el mensaje pertenezca a una ONRP y transite por un *switch intermedio* lo enviará hacia el próximo salto. Si se encuentra en un switch destino lo enviará a una *Flow table* de *ONATs* para ser procesado.
- *Flow tables* de *ONATs*
  Estas realizan la inicialización del encaminamiento en los *switch origen* y la recuperación de la información en los *switch destino*, haciendo uso de los *ONATs Ids*.

7.2.2. Acciones que implementan las *flow tables*

Cada una de las *flow tables* realiza una serie de acciones con el fin de cumplir los objetivos antes planteados. A continuación, se especifica las funcionalidades que implementa cada tipo de *flow table* por medio de las *flow entries*. En la figura 7.1 se brinda un diagrama de las posibles acciones aquí especificadas.

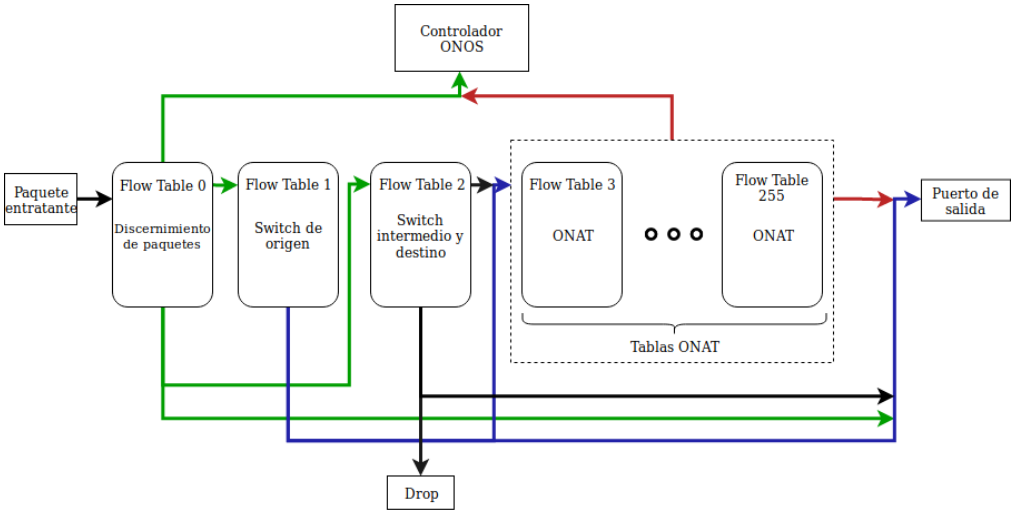


Figura 7.1: Diagrama de flujo del procesamiento de un mensaje en las diferentes *flow tables*

Flow table 0

- Continuar el *pipeline* en la *flow table* 1
  Esta acción se aplica sobre paquetes que no han ingresado a la *ON* y potencialmente puedan pertenecer a una *ONRP*.
  Se verifica que el paquete sea UDP/TCP, que la IP origen pertenezca a la subred del *PoP* en el que se encuentra el *Switch OF actual* y que la IP destino pertenezca a la subred de otro *PoP* de la *ON*.
- Continuar el *pipeline* en la *flow table* 2
  Esta acción se aplica sobre paquetes que previamente han ingresado a la *ON* (que ya han sido encaminados).
  Se verifica que el paquete sea UDP/TCP, que la IP origen sea de otro switch perteneciente a la *ON* y que la IP destino sea la del *Switch OF actual*.
- Encapsular el paquete como *PACKET_IN* y enviarlo al controlador
  Esta acción verifica si el paquete entrante es un *ARP request* solicitando la IP del *Switch OF actual*, en cuyo caso se envía al controlador de forma que este realice el *ARP response*.
- Enviarlo al puerto *ALL*. A su vez encapsular el paquete como *PACKET_IN* y enviarlo al controlador.
  La acción de enviarlos al puerto ALL se aplica sobre los paquetes *ARP response*, con el objetivo de que el sistema sea transparente y que este llegue al host para quien va dirigido el paquete. Se envía al controlador para que este puede obtener la información de la asociación IP-MAC de los host del *PoP* sin necesidad de realizar una consulta ARP.
- Enviar el paquete al puerto *ALL*
  Cualquier otro tipo de mensaje que no corresponda a administración de *ONRApp* es enviado por el puerto *ALL*, de forma de ser transparentes a estos.

Flow table 1

- Continuar el *pipeline* en la *flow table* de *ONAT*
  Esta acción se aplica a paquetes que no han ingresado a la *ON* y pertenecen a una *ONRP* particular.
  Se verifica que el paquete esté incluido en los parámetros que especifica la *ONRP* en cuestión.
  Como fue explicado en la sección 3.2, cada *ONRP* tiene asociada una única tabla de *ONATs* en el controlador, pero en el switch esta tiene el nombre de *flow table ONAT*. En contraposición, una *flow table ONAT* puede estar asociada con más de una *ONRP* simultáneamente.
- Encaminar paquete por un *Path* directo
  Esta acción se aplica a paquetes que no han ingresado a la *ON* y pertenecen a una *ONRP* cuyo *Path* es directo (ver sección 7.3).
- Enviar el paquete al puerto *ALL*
  Esta acción se aplica a paquetes que no han ingresado a la *ON* y no pertenecen a ninguna *ONRP*.

Flow table 2

- Encaminar paquete por un *Path* estándar
  Esta acción se aplica a paquetes que previamente han ingresado a la *ON* y se encuentran en un *switch intermedio*.
  Como fue explicado en la sección 3.2, para realizar el encaminamiento se modifica el *Local ONRP Id* del mensaje y se envía por la misma interfaz por donde llegó.
- Encapsular el paquete como *PACKET_IN* y enviarlo al controlador
  Esta acción se aplica a paquetes que previamente han ingresado a la *ON* (las direcciones IP del paquete son de los *Switches OF*), se encuentran en un *switch intermedio* y no se ha configurado la *flow entry* de encaminamiento por *Path* estándar por falta de información.
- Continuar el *pipeline* en la *flow table* de *ONAT*
  Esta acción se aplica sobre paquetes que previamente han ingresado a la *ON*, pertenecen a una *ONRP* y se encuentran en el *switch destino*.
- *Drop* de paquetes
  Esta acción se aplica a paquetes que no pertenecen a ninguna *ONRP* y tienen la IP de destino del switch.
  Esta acción se lleva a cabo por razones de seguridad, ya que todo tráfico que llegue a un switch, cuya dirección IP de destino sea la dirección IP asociada al switch, debería tener una *ONRP* asociada. Por lo tanto no se desea que este tráfico sea enviado al controlador.

Flow tables de *ONATs*

- Realizar traducción de *ONAT* y encaminar
  Esta acción se aplica a paquetes que no han ingresado a la *ON*, para los cuales ya existe una *ONAT* particular.
  Se verifica que el paquete esté incluido en los parámetros que especifica la *ONAT* en cuestión.
- Reconstruir el mensaje original y enviarlo al host correspondiente
  Esta acción se aplica a paquetes que previamente han ingresado a la *ON*, se encuentran en el *switch destino*, poseen una *ONRP* y una *ONAT* asociadas.
  Se verifica que el puerto origen de capa de transporte coincida con el último *Local ONRP Id* del *Path* de la *ONRP* y que el puerto destino de capa de transporte coincida con la *ONAT Id*.
- Encapsular el paquete como *PACKET_IN* y enviarlo al controlador
  Esta acción se aplica a paquetes que no tengan una *ONAT* asociada, indiferentemente si han ingresado a la *ON* o no.
- Enviar directamente al router del *ISP* sin realizar encaminamiento
  Se agrega la protección para el caso en que estén ocupados los 2^{16} *ONATs* correspondientes a una *ONRP* de enviar los paquetes sin encaminar hasta que se libere una.

7.2.3. Optimización de *flow tables*

El protocolo *OpenFlow* posee diversas herramientas que permiten la optimización del uso de *flow entries* y de las *flow tables*. En este proyecto se hizo uso de los campos *metadata* y *cookie*, para generar simplificaciones tanto a nivel del *Switch OpenFlow*, como a nivel de la aplicación *ONRApp*.

A continuación, será analizada la forma en la que estos campos son utilizados:

Campo *cookie*

Recordando lo expuesto en la sección 2.3, el campo *cookie* posee un largo de 64 bits.

El controlador *ONOS* hace uso de este campo, configurando un identificador llamado *Flow Id*, que tiene como objetivo la identificación de *flow entries*. En particular, *ONOS* asocia los primeros 16 bits del campo a identificar la aplicación que solicitó la instalación de la *flow entry*, por lo que resulta imprescindible no modificar esos bits.

Sin embargo, los 48 bits restantes pueden ser modificados. Con esto presente, estos bits son utilizados para asociar una *flow entry* a su respectiva *ONRP*. Considerando que el *matching* de paquetes entrantes es realizado a nivel de los *Switches OpenFlow*, esta optimización permite evitar la repetición del *matching* del paquete a nivel del controlador, en caso de un *PACKET_IN* o un evento relacionado con *flow entries*.

La desventaja de esta optimización es que la cantidad máxima de *ONRPs* resulta acotada por 2^{48}.

Campo *metadata*

El uso de este campo simplifica la transición entre *flow tables* de un mismo switch. Dado un paquete entrante, una vez determinado la *ONRP* asociado, se configura el *Global ONRP Id* en el *metadata*.

En *flow tables* que ya se conoce el *ONRP* asociado, se evita la repetición del *matching* contra los mismos campos, simplemente realizando una comprobación contra el *metadata* antes configurado (por ejemplo en las *flow tables* de *ONATs*).

7.3. Clasificación de *ONRPolicies*

Hasta esta sección, siempre que se menciona una *ONRP*, el *Path* asociado posee al menos tres *PoPs*: un origen, al menos un rebote y un destino.

Sin embargo, como se analiza a continuación, no solo existen *ONRPs* con este tipo de *Path*. Aquí se presenta la clasificación de *ONRPs* según las *flow entries* instaladas en los *Switch OF*:

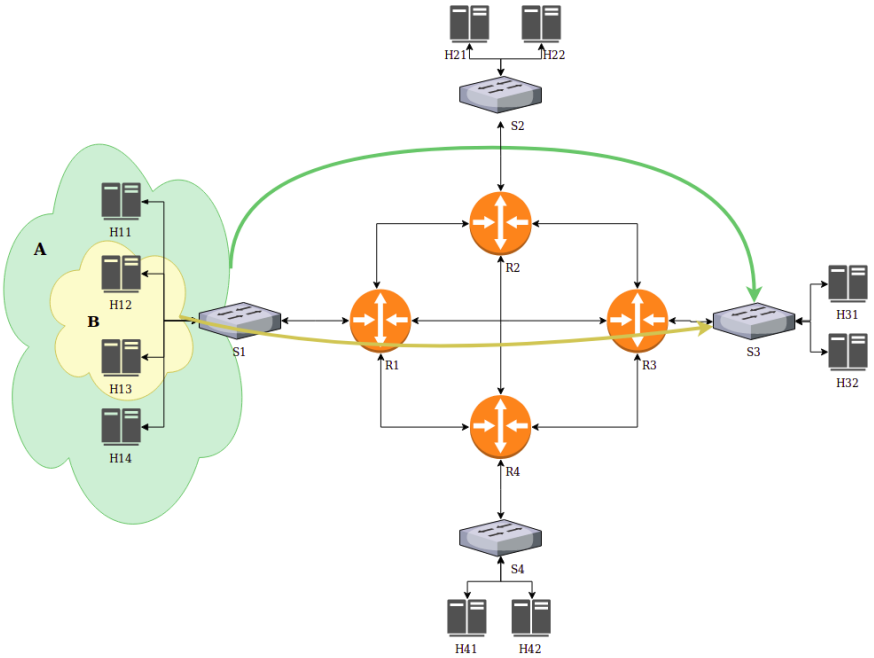


Figura 7.2: Topología con ejemplos de *Path* estandar y *Path* directo

ONRP estándar

Es el caso típico de uso del algoritmo de encaminamiento. Se define como una *ONRP* cuyo *Path* tiene un *switch origen*, al menos un *switch intermedio* y un destino.

ONRP directa

Supongamos que se tiene una topología compuesta por cuatro *PoPs* y dos conjuntos de hosts A y B del PoP_1, tal que B es un subconjunto de A, como se muestra en la figura 7.2.

En principio, si el subconjunto de hosts B se pretende comunicar con H_{32} sin realizar ningún rebote (marcado en amarillo en la figura 7.2), queda claro que no es necesario establecer una *ONRP* ya que el *Path* es directo.

Sin embargo, esto no es cierto en el siguiente caso:

Considere que previamente se ha creado una *ONRP* definiendo que la comunicación entre el conjunto A y H_{32} debe rebotar en el PoP_2 (marcado en verde en la figura 7.2). Para tal fin, si se desea establecer una comunicación sin rebotes para el subconjunto B, resulta imprescindible la existencia de una *ONRP*.

Si bien esto es solucionable con una *ONRP* con su respectiva *flow table ONAT*, la solución no es óptima. Como optimización a este problema, se instala una única *flow entry* en la *flow table* 1 que, para cada *ONRP* con *Path* directo, evita la traducción *ONAT* reduciendo el tiempo de procesamiento y la cantidad de *flow entries* asociadas. De esta forma la *flow entry* enviará los paquetes directamente a Internet sin realizar cambios en sus campos de los encabezados.

ONRP inactiva

Es una *ONRP* que no tiene *flow entries* instaladas en el *switch origen* y por ende, no se realiza el encaminamiento que esta define.

Las *ONRPs* inactivas son utilizadas en dos casos:

- Desconexión de un *Switch OF* del *Path* de una *ONRP*.
- Solicitud de modificación de una *ONRP*.

7.4. Análisis de dinámicas

Esta sección describe de forma minuciosa el proceso de creación y modificación de *ONRPs* y sus *ONATs* asociadas. La sección comienza con la presentación del proceso de registro de un nuevo *PoP*, acción previa necesaria para la creación de políticas de encaminamiento.

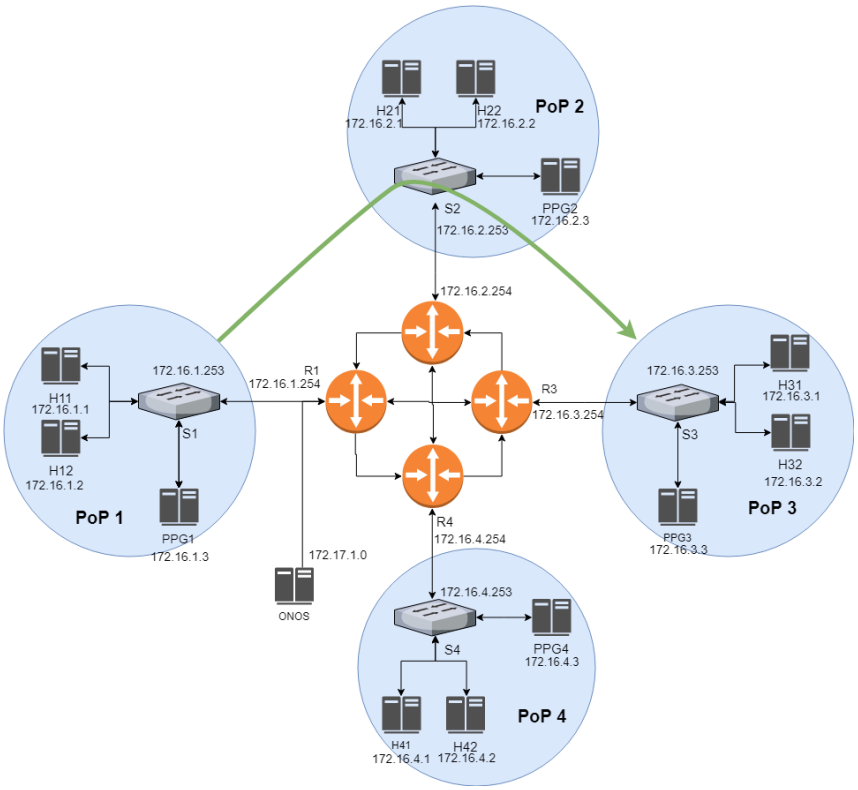


Figura 7.3: Topología detallada con *Path* estandar

7.4.1. Registro de *PoPs*

La implementación de una *ONRP* requiere la especificación de un *Path*, entre otros parámetros. Este parámetro sólo puede ser definido si se conocen los *PoPs* por los que pasa. Esto implica que el primer paso para la creación de una *ONRP* es el registro de todos los *PoPs* del *Path*.

A nivel de los elementos de red o *Switch OpenFlow*, esto implica la instalación de un conjunto mínimo de *flow entries* que, entre otras acciones, permiten la identificación de tráfico intercambiado entre *PoPs*.

Consideremos la arquitectura de red de la figura 7.3. Para el *Path* allí indicado es necesario que el switch en el PoP_1 identifique el tráfico saliente dirigido al PoP_3. Es importante comprender que esta identificación es previa e independiente a la existencia de una *ONRP*, por lo que las *flow entries* asociadas a este tipo de identificación deben aparecer en la *flow table* 0, como es presentado en 7.2.2.

En la tabla 7.1 se presenta un ejemplo de *flow entry* a instalar en la *flow table* 0 del switch del PoP_1, cuyo objetivo es la identificación de tráfico saliente de PoP_1 con destino a PoP_3.

Tabla 7.1: *Switch* S_1- *Flow entry* de identificación de tráfico saliente de PoP_1 hacia PoP_3

	Opción	Valor	Comentario
Match Header Fields	Ethernet Type	IPv4	
	IP orígen	172.16.1.0/24	Subred del PoP1
	IP destino	172.16.3.0/24	Subred del PoP3
	Protocol	UDP/TCP	Una *flow entry* por cada protocolo
Actions	Go-to table	1	*Flow table* de *ONRPs*

7.4.2. Implementación de *ONRPs*

Una *ONRP* puede ser implementada solo si previamente se registraron los *PoPs* del *Path* que define, como fue explicado en 7.4.1. Una vez que estas condiciones son cumplidas, se puede proceder a la implementación de la política deseada. Con el fin de comprender el proceso de implementación de una *ONRP* en el plano de datos, se presenta un ejemplo a ser analizado en detalle.

Supongamos que previamente en *ONRApp* no existe ninguna *ONRP* configurada. Entonces, consideremos una *ONRP* que cumple $Path = \{PoP_1, PoP_2, PoP_3\}$, sin especificación de protocolo ni puertos de capa de transporte, como se presenta en la tabla 7.2.

Tabla 7.2: Tabla de *ONRPs* en *ONRApp*

Global ONRP Id	Prioridad	IP src	IP dst	Path	Protocolo	Puerto Org	Puerto Dst
1	792	172.16.1.0/24	172.16.3.0/24	[1,2,3]	*	*	*

El proceso de implementación de esta *ONRP* comienza en *FlowEntry Manager* en *ONRApp*. Como se menciona en 5.2.1.d, este gestor se encarga de clasificar los switch pertenecientes al *Path* y posteriormente, solicitar la instalación de las *flow entries* necesarias según la clasificación.

Los switch se clasifican en los tres grupos: el *switch origen*, los *switch intermedios* y el *switch destino* (ver 7.1 por sus definiciones). En el ejemplo, S_1 es el *switch origen*, S_2 el *switch intermedio* y S_3 el *switch destino*.

El resto de la sección se presentará separando el análisis según cada switch de la clasificación anterior.

Implementación en el *switch origen*

El *switch origen* es el encargado de detectar y encaminar los nuevos paquetes que ingresarán a la *ON*. Para lograr este objetivo, se instala una *flow entry* (con las características presentadas en la tabla 7.3) en la *flow table* 1. Es en esta etapa del proceso que a la *ONRP* se le asigna una *flow table* de *ONATs* en el switch.

Tabla 7.3: *Switch* S_1- *Flow entry* para la detección de paquetes pertenecientes a la $ONRP_1$

	Opción	Valor	Comentario
Match Header Fields	Ethernet Type	IPv4	
	IP orígen	172.16.1.0/24	Subred del PoP1
	IP destino	172.16.3.0/24	Subred del PoP2
Actions	Write-Metadata	1	*Global ONRP Id*
	Go-to table	ONAT table	Flow table de ONATs asociada

Mientras no exista ningún flujo de datos que requiera ser traducido, la única *flow entry* que contiene la *flow table* de *ONATs*, es la encargada de enviar mensajes *PACKET_IN* hacia el controlador. Estos mensajes permiten la generación de nuevas *ONATs* como será explicado en 7.4.3. Retomando el ejemplo, esta *flow entry* se presenta en la tabla 7.4.

Tabla 7.4: *Switch* S_1- *Flow entry* por defecto en la *flow table* de *ONATs* asociado

	Opción	Valor	Comentario
Match	Metadata	1	*Global ONRP Id*
Actions	Send to port	CONTROLLER	*Enviar como PACKET_IN*

Un criterio tomado en la implementación es que, cada vez que se instale una *flow entry* como respuesta a un *PACKET_IN*, se envía un *PACKET_OUT* conteniendo el mensaje. Este mensaje es modificado según las acciones que realizaría la *flow entry* instalada, permitiendo que la información contenida en el mensaje llegue a destino sin necesidad de retransmisión desde el origen.

Implementación en el *switch intermedio*

El procesamiento en el *switch intermedio* se realiza en la *flow table* 2. Para ello,

se hace uso del *Local ONRP Id* para obtener la información del próximo salto y encaminar el paquete como se especifica en el capítulo 3.

En la tabla 7.5 se presenta la *flow entry* que implementa lo anterior. En contraste con lo que el algoritmo de encaminamiento estipula, en esta tabla se aprecia que la acción sobre el paquete es enviarlo al controlador. Esto se atribuye a la necesidad de conocer la MAC del router del *ISP* para poder invertir los campos direcciones Ethernet (el paquete debe volver por donde llego al switch).

Tabla 7.5: *Switch* S_2- *Flow entry* asociada al rebote intermedio sin MAC del router del *ISP*

	Opción	Valor	Comentario
Match Header Fields	Ethernet Type	IPv4	
	IP orígen	172.16.1.253/32	IP del switch1
	IP destino	172.16.2.253/32	IP del switch2
	Protocolo	UDP/TCP	Una *flow entry* por cada protocolo
	Layer 4 Port Src	0	*Local ONRP Id* para el link $L_{1,2}$
Actions	Enviar	Puerto *CONTROLLER*	Enviar como PACKET_IN

Una vez que un primer paquete del circuito virtual llega al switch, hace *match* con la *flow entry* anterior y llega al controlador, la direccion Ethernet del router pasa a ser conocida. Dadas estas condiciones, se instala la *flow entry* presentada en la tabla 7.6.

Tabla 7.6: *Switch* S_2- *Flow entry* asociada al rebote intermedio con MAC del router del *ISP*

	Opción	Valor	Comentario
Match Header Fields	MAC Src	ISP Router MAC	
	Mac Dst	Switch OF MAC	
	Ethernet Type	IPv4	
	IP orígen	172.16.1.253/32	IP del switch1
	IP destino	172.16.2.253/32	IP del switch2
	Protocolo	UDP/TCP	Una *flow entry* por cada protocolo
	Layer 4 Port Src	0	*Local ONRP Id* para el link $L_{1,2}$
Actions	Set MAC Src	Switch OF MAC	Se invierten las MAC Src-Dst
	Set Mac Dst	ISP Router MAC	Se invierten las MAC Src-Dst
	Set IP Src	172.16.2.253/32	IP del switch2
	Set IP Dst	172.16.3.253/32	IP del switch3
	Set Layer 4 Port Src	0	*Local ONRP Id* para el link $L_{2,3}$
	Send to port	*IN_PORT*	Mismo puerto por el que entró

Es importante comprender que la *flow entry* de la tabla 7.5 sigue siendo necesaria ya que, si por alguna razón el router conectado al switch cambia de MAC, el mensaje no se ajustará con la de la tabla 7.6, pero si lo hará con la de la tabla 7.5.

Implementación en el *switch destino*

El procesamiento en el *switch destino* también se realiza en la *flow table* 2. La acción de la *flow entry* presentada en la tabla 7.7 es continuar el procesamiento en la *flow table* de *ONATs* correspondiente a la *ONRP*.

El comportamiento de los *switches destino* durante la traducción de *ONATs* se realiza en la sección siguiente (7.4.3).

Tabla 7.7: *Switch* S_3- *Flow entry* de identificación de *ONRP* en destino

	Opción	Valor	Comentario
Match Header Fields	Ethernet Type	IPv4	
	IP orígen	172.16.2.253/32	IP del switch2
	IP destino	172.16.3.253/32	IP del switch3
	Protocolo	UDP/TCP	Una *flow entry* por cada protocolo
	Layer 4 Port Src	0	*Local ONRP Id* para el link L2,3
Actions	Write-Metadata	1	*Global ONRP Id*
	Go-to table	ONAT table	Flow table de ONATs asociada

7.4.3. Implementación de *ONATs*

Las *flow tables* de *ONATs* son las que realizan las traducciones de paquetes necesarias para realizar el encaminamiento de origen a destino. Cada *ONRP* posee una *flow table* de *ONATs* asignada. Estas son asignadas en orden creciente y de forma cíclica, desde la *flow table* 3 hasta la 255.

Para esta etapa, se considera que la *ONRP* asociada ya se encuentra totalmente implementada. El proceso de traducción con *ONATs* se divide en dos etapas a ser analizadas a continuación: una primer etapa que implica el ingreso a la *ON* en origen y una segunda que implica el egreso de la *ON* en destino.

ONAT origen

El primer flujo de datos que realiza el *matching* en una *flow table* de *ONATs* en el origen, es enviado al controlador de forma que se genere una nueva *ONAT* que permita almacenar la información del flujo y genere los identificadores necesarios. Continuando con el ejemplo introducido en la sección anterior, en la tabla 7.8 se aprecia la entrada *ONAT* que es generada en esta etapa.

Tabla 7.8: Entrada de traducción *ONAT*

Global ONRP Id	1				
OATId	IP src	IP dst	Proto	Puerto Org	Puerto Dst
1	172.16.1.2	172.16.3.2	TCP	53000	80

Una vez que se asigna un *ONAT Id* a la traducción, *FlowEntry Manager* instala en la *flow table* de *ONATs* asociada a la *ONRP*, la *flow entry* que permite realizar la traducción en el *switch origen*. Esta *flow entry* se presenta en la tabla 7.9.

Tabla 7.9: *Switch* S_1- *Flow entry* que implementa la traducción de ingreso a la *ON*

	Opción	Valor	Comentario
Match Header Fields	Metadata	1	Metadata asociado a la ONRP1
	Ethernet Type	IPv4	
	IP orígen	172.16.1.2/32	IP origen original
	IP destino	172.16.3.2/32	IP destino original
	Protocolo	TCP	
	Layer 4 Port Src	53000	*Puerto origen original*
	Layer 4 Port Dst	80	*Puerto destino original*
Actions	Set Ethernet Type	IPv4	
	Set MAC Src	Switch OF MAC	
	Set Mac Dst	ISP Router MAC	
	Set IP Src	172.16.1.253/32	IP del switch1
	Set IP Dst	172.16.2.253/32	IP del switch2
	Set Layer 4 Port Src	0	*Local ONRP Id* para el link L1,2
	Set Layer 4 Port Dst	0	*ONAT Id* asociado a este flujo
	Send to port	*ALL*	Puerto donde está el router del ISP

Una vez que los paquetes realizan el *matching* contra esta *flow entry*, estos se consideran encaminados sobre la *ON*.

ONAT Destino

De forma similar al caso del origen, el primer paquete sube al controlador, este solicita la MAC del host destino utilizando *Centralized ARP*. La MAC del host destino debe estar presente en la *flow entry* para que esta pase tener el valor de la MAC destino del switch a tener la MAC del host de destino. Finalmente, se instala la *flow entry* presentada en la tabla 7.10.

Tabla 7.10: *Switch* S_3- *Flow entry* que implementa la traducción de salida de la *ON*

	Opción	Valor	Comentario
Match Header Fields	Metadata	1	Metadata asociado a la ONRP1
	Ethernet Type	IPv4	
	IP orígen	172.16.2.253/32	IP del switch2
	IP destino	172.16.3.253/32	IP del switch3
	Protocolo	TCP	
	Layer 4 Port Src	0	*Local ONRP Id* para el link L2,3
	Layer 4 Port Dst	0	*ONAT Id* asociado al flujo
Actions	Set Ethernet Type	IPv4	
	Set MAC Src	Switch OF MAC	
	Set Mac Dst	host MAC	MAC del host destino
	Set IP Src	172.16.1.2/32	IP origen original
	Set IP Dst	172.16.3.2/32	IP destino original
	Set Layer 4 Port Src	53000	*Puerto origen original*
	Set Layer 4 Port Dst	80	*Puerto destino original*
	Send to port	*ALL*	Puerto donde está el host

Por defecto, en todas las *flow entries* asociadas a las *flow tables* de *ONATs* se utilizó el criterio de implementar un tiempo de inactividad para determinar cuando

borrarse. Cuando estas *flow entries* son eliminadas por inactividad, el switch en cuestión envía un evento a *FlowEntry Manager* a partir del cuál borra el registro de esa *ONAT Id*, permitiendo ser reutilizada por un nuevo flujo de datos.

7.4.4. Modificación del *Path* de una *ONR*

Modificar una *ONRP* es la funcionalidad que permite alterar el *Path* de la misma. Este proceso presenta una ventaja en cuanto a la pérdida de paquetes con respecto a borrar y crear la ONRP, ya que si se borra una *ONRP* todos los mensajes en tránsito se pierden.

El proceso de modificar de una ONRP se describe a continuación. Sea una *ONRP* cuyo $Path_{old}$ desea ser modificado a un nuevo $Path_{new}$ tal que:

$$Path_{old} = [PoP_0, PoP_1, .., PoP_{k-1}] \quad / \; PoP_i \in ON \forall \; i = [0, ..k-1]$$

$$Path_{new} = [PoP_0, PoP_1, .., PoP_{n-1}] \quad / \; PoP_j \in ON \forall \; j = [0, ..n-1]$$

$$PoP_{k-1} = PoP_{n-1}$$

Este proceso consta de tres etapas:

1. Primer paso
   Se usan los parámetros *ModifiedPath*, *ModifiedGlobalONRP* y otros del objeto *ONRP* de *ONRApp* para instalar un *ONRP* inactivo en paralelo con la política anterior.

   Esto implica que todas las *flow entries* son instaladas a excepción de las del switch S_1. Se tomó el criterio de no instalar las *flow entries* en el S_1 debido a que se tendrían dos grupos de *flow entries* con los mismos *match* pero acciones diferentes, dando posibilidad a errores.

2. Segundo paso
   Se desactiva el antiguo *ONRP* y posteriormente se instalan las *flows entries* del nuevo *Path* para que el tráfico sea redirigido por $Path_{new}$. Estas acciones son las que modifican las *flow entries* instaladas en el *Switch OF* de origen del camino.

3. Tercer paso
   Se espera un tiempo τ configurable en el comando de *REST API* y posteriormente se eliminan las *flows entries* de los *Switch OpenFlow* en los $PoP_1, .., PoP_{k-1}$ asociados a $Path_{old}$.

   Observar que si el tiempo τ es suficientemente grande para que todos los mensajes en tránsito lleguen a destino, se evita la perdida de paquetes.

Capítulo 8

Pruebas de validación

En el presente capítulo se describen y analizan los resultados obtenidos en pruebas de funcionalidad y rendimiento de *ONRApp*, con el objetivo de comprobar el funcionamiento descrito en el capítulo 5.

En la sección 8.1 se detallan las pruebas de concepto realizadas que validan el algoritmo de encaminamiento planteado en el capítulo 3.

En la sección 8.2 se resumen las diferentes pruebas de funcionalidad realizadas. En estas pruebas se tiene en cuenta pruebas externas haciendo uso de servicios web expuestos por la *API*, reconocimiento de potenciales errores generados por usuarios que utilicen la aplicación, correcto encaminamiento del tráfico en la red, entre otros.

Finaliza el capítulo en la sección 8.3 donde se exponen los resultados obtenidos en las pruebas de rendimiento realizadas. Las pruebas de rendimiento cuantifican parámetros relevantes como tiempo de respuesta del sistema ante fallas de ruteo, precisión de sistema de medición y tiempo que tarda la aplicación en registrar nuevos puntos de presencia o implementar de políticas de ruteo.

Es importante destacar que todas las topologías de red han sido simuladas usando *Mininet*, una herramienta destinada a la emulación de redes. Esta es ampliamente usada en la simulación de redes definidas por software. [30]

8.1. Prueba de concepto: algoritmo de encaminamiento

En esta sección se muestra la validez del algoritmo de encaminamiento diseñado a partir de pruebas de concepto simples que permiten visualizar claramente cómo se modifica el tráfico en la red para ser encaminado acorde a lo expuesto en 3.2. La veracidad de cada prueba será probada por intermedio de la herramienta de análisis de tráfico *Wireshark*. Todas las pruebas se llevan a cabo simulando la red en *Mininet*.

La topología de red seleccionada se muestra en la figura 8.1. En esta se observa que la red está compuesta por cuatro puntos de presencia. El controlador *ONOS* se encuentra corriendo en un host virtualizado en *Mininet*. En este caso el controlador se encuentra conectado a uno de los routers del *ISP*, dado que puede estar ubicado

en cualquier punto de la red.

8.1.1. Primer prueba de concepto: Validez del encaminamiento

Esta prueba funciona como primer acercamiento al algoritmo de encaminamiento.

La política de ruteo implementada en la red cumple con los parámetros mostrados en la tabla 7.9.

Tabla 8.1: Prueba de concepto: *ONRP* implementada en la red

Global ONRP Id	Prioridad	IP src	IP dst	Path	Protocolo	Puerto Org	Puerto Dst
1	792	172.16.1.0/24	172.16.3.0/24	[1,2,3]	*	*	*

Dado el *Path* especificado en la *ONRP*, el flujo de datos debe seguir la ruta marcada en 8.1 color verde.

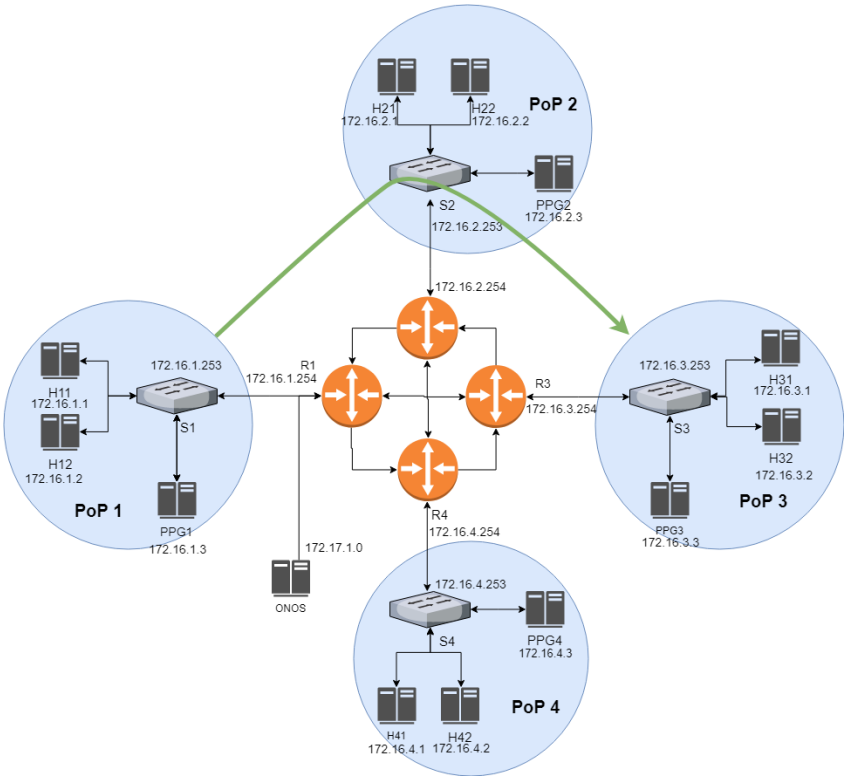
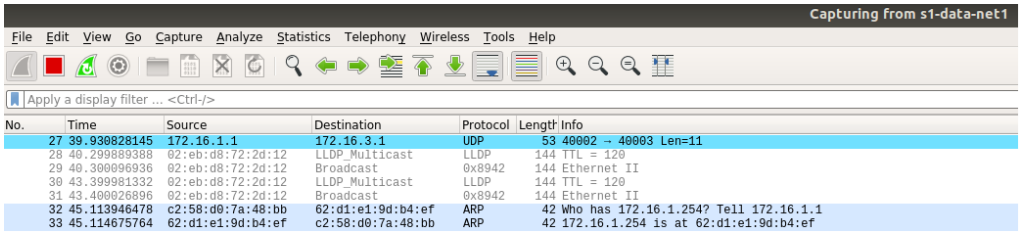
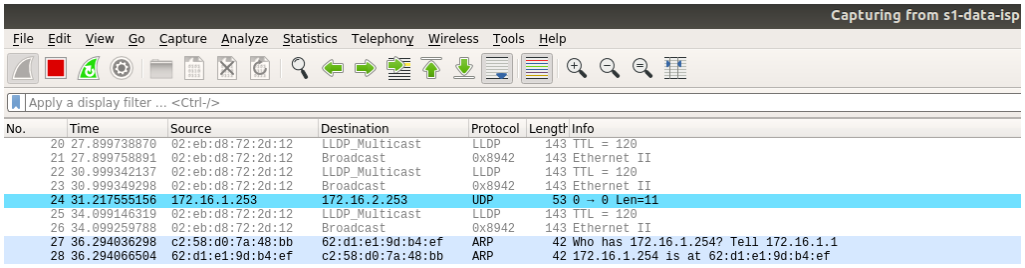


Figura 8.1: Topologia simulada en *Mininet* para prueba de concepto

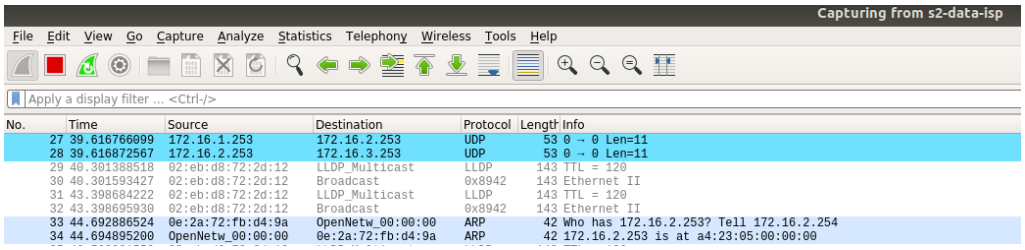
En las siguientes figuras se muestra utilizando *Wireshark* como se modifica en los *Switches OpenFlow* un paquete UDP enviado desde H11 a H31 al transitar por el *Path* especificado.



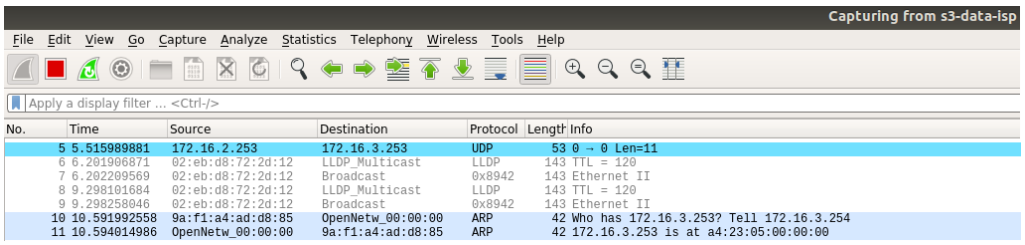
8.2.a- Análisis de tráfico en Wireshark: Interfaz de s_1 hacia la LAN interna del PoP



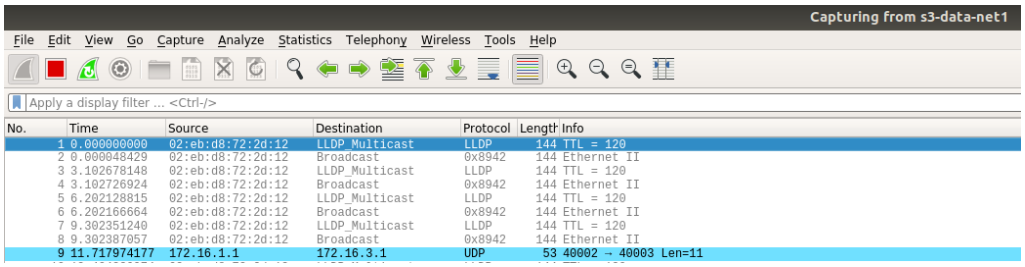
8.2.b- Análisis de tráfico en Wireshark: Interfaz de s_1 hacia R_1



8.2.c- Análisis de tráfico en Wireshark: Interfaz de s_2 hacia R_2



8.2.d- Análisis de tráfico en Wireshark: Interfaz de s_3 hacia R_3



8.2.e- Análisis de tráfico en Wireshark: Interfaz de s_3 hacia H_{31}

Figura 8.2: Análisis de tráfico en Wireshark: prueba de concepto del algoritmo de encaminamiento

En la figura 8.2.a se muestra la interfaz del switch s_1, conectada a la red privada del PoP. El paquete UDP llega con IP origen H_{11}, IP destino H_{31}, puerto

origen de capa de transporte 40002 y puerto destino 40003. Observando la política implementada en la tabla 8.1, este paquete debería ser encaminado en la *ON*.

En la figura 8.2.b se observa que el paquete UDP es modificado y enviado desde PoP_1 al PoP_2 con las direcciones asignadas a los switchs. Siguiendo el camino del paquete, en la figura 8.2.c, se muestra como en la interfaz de s_2 llega desde R_2 el mismo paquete que salió desde PoP_1. Luego, este paquete es modificado y enviado al router nuevamente.

Recordar que en este punto, no solo se debe modificar los encabezados de capa de red y capa de transporte. Para que el router acepte el paquete, debe enviarse el paquete con dirección MAC la MAC del router del *ISP*.

El paquete transita por la red y llega a la interfaz del switch s_3 conectada al router del *ISP* como se muestra en la figura 8.2.d, donde es modificado y enviado a H_{31}. En la figura 8.2.e se muestra como el paquete llega al host H_{31} idéntico a como salió desde H_{11}.

Con estos resultados se valida la prueba de concepto encaminamiento del algoritmo propuesto.
En la siguiente sección se continúa con pruebas de concepto para visualizar aspectos que se generan al aumentar el número de políticas en la red, como por ejemplo visualización de identificación de flujos y método de *ONAT*, concepto explicado en la sección 3.2.

8.1.2. Segunda prueba de concepto: identificación de flujos

En esta sección se muestra como a partir de la implementación de un conjunto de *ONRPs* en un sistema cuya arquitectura se detalla en 3.1, se modifica el tráfico generado desde los puntos de presencia a partir de las reglas impuestas por el algoritmo expuesto en 3.2. En particular, se muestra aspectos relevantes del algoritmo como son la aplicación de *ONATs* y la utilización de identificadores locales.

En esta sección, gracias a las funcionalidades brindadas por *ONRApp* se establecen las siguientes 5 *ONRPs* mostradas en la figura 8.3. Destacar que los ONRPs que tienen la misma prioridad en el *switch* se diferencian en el switch por alguno de sus valores por ejemplo el puerto de capa 4.

En la figura 8.4 se muestra la topología junto con los flujos generados por las políticas implementadas.

# ONRPolicy	1	2	3	4	5
Red origen	172.16.1.2/32	172.16.1.2/32	172.16.1.2/32	172.16.1.0/32	172.16.1.0/24
Red destino	172.16.3.2/32	172.16.3.2/32	172.16.3.2/32	172.16.3.2/32	172.16.3.0/24
Protocolo	UDP	UDP	UDP	TCP	*
Puerto L4 origen	40005	40003	40002	*	*
Puerto L4 destino	40006	40004	40003	80	*
Path	[PoP1,PoP3]	[PoP1,PoP2, PoP4,PoP3]	[PoP1,PoP2, PoP3]	[PoP1,PoP2, PoP4,PoP3]	[PoP1,PoP2, PoP3]
Prioridad	8224	8224	8224	4120	792
Color del path	Rojo	Amarrillo	Verde	Amarrillo	Verde

Figura 8.3: *ONRPs* implementadas en prueba de concepto de identificación de flujos

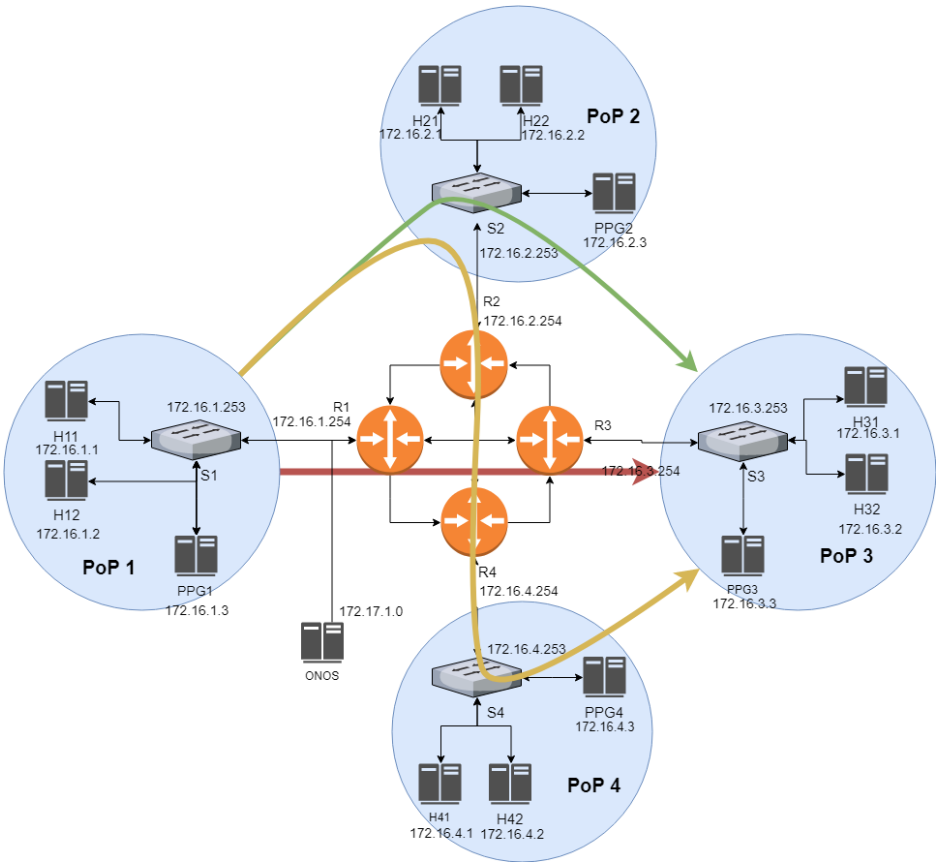


Figura 8.4: Caminos generados por *ONRPs* mostradas en tabla 8.3

Identificadores locales

En la figura 8.5 se muestra el tráfico que pasa por s_1 cuando H_{11} intenta comunicarse con H_{31}. Se observa que en la interfaz de s_1 conectada al router, el paquete sale dirigido hacia PoP_2. En este caso se le asigna al $ONRP_5$ el identificador local 0 en el *link* [PoP_1,PoP_2], información transportada en el puerto de origen.

En el puerto destino, donde se lleva información del *Flow*, se le asigna el identificador 1. Observar que en esta política no se especifican puertos ni protocolo, por lo tanto pueden generase distintos *Flows* asociados.

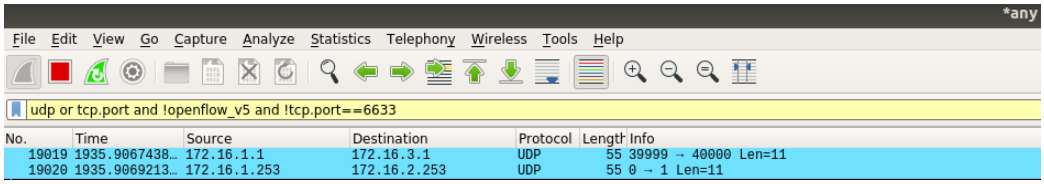


Figura 8.5: Análisis de tráfico en Wireshark: tráfico total por s_1 perteneciente a $ONRP_5$

En la figura 8.6 se muestra tráfico que será tratado en base a $ONRP_4$. A esta *ONRP* se le asigna el identificador local *1* en el primer *Link* del *Path*. Esto se debe a que el identificador local 0 ya fue consumido por $ONRP_5$ y se asignan en orden ascendente desde el 0 hasta 2^{16}.

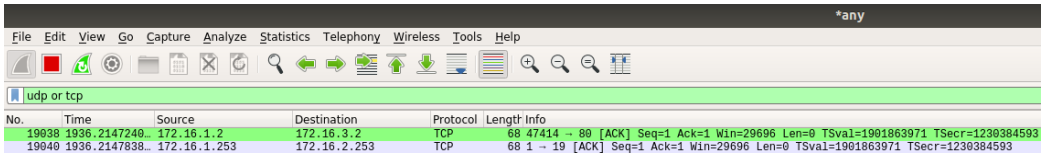


Figura 8.6: Análisis de tráfico en Wireshark: tráfico total por s_1 perteneciente a $ONRP_4$

En la figura 8.7 se muestra tráfico generado desde PoP_1 perteneciente a distintas políticas. En esta se puede ver como a los paquetes de los $ONRP_2$ y $ONRP_3$ se les asigna los identificadores locales 3 y 2 respectivamente. Además se puede observar como al mensaje perteneciente a la $ONRP_1$ que es directo, no se le modifican los puertos de capa 4, ya que este no es encaminado. Esta funcionalidad es explicada en la sección 7.

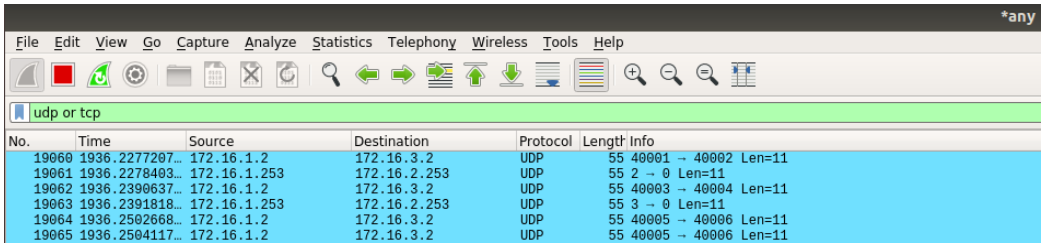


Figura 8.7: Análisis de tráfico en Wireshark: tráfico total por s_1 perteneciente a múltiples *ONRP*

En las figuras 8.5, 8.6 y 8.7 se verifica que el mecanismo de prioridad funciona ya que todos los mensajes cumplen con los parámetros de la $ONRP_5$, sin embargo, 4 de los 5 mensajes se encaminan por los otros *ONRPs* al tener mayor prioridad.

ONAT

Se presenta un ejemplo del mecanismo de traducción *ONAT* descrito en la sección 7.

Las figuras 8.8 y 8.9 muestran el funcionamiento de la traducción *ONAT* con 5 mensajes de diferentes características pero todos pertenecientes al $ONRP_5$.

En la figura 8.8 se muestra como en el switch origen, a cada mensaje se le asigna un *ONAT* Id diferente al tener diferentes puertos de capa 4.

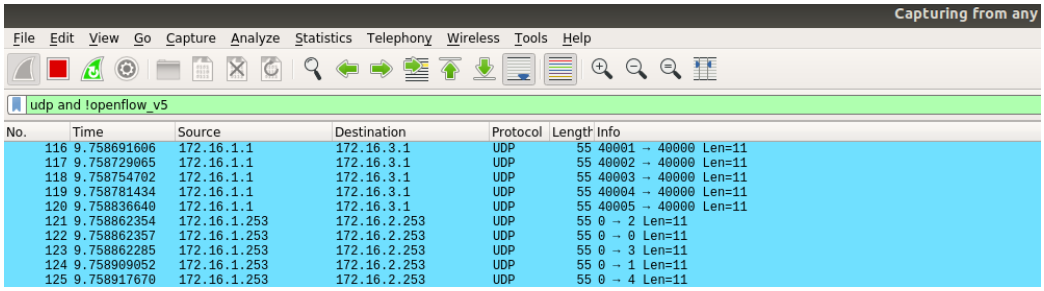


Figura 8.8: Análisis de tráfico en Wireshark: tráfico total por s_1 perteneciente a $ONRP_5$. Traducción ONAT.

Finalmente en la figura 8.9 muestra como usando los *ONAT Ids* el switch de destino recompone los mensajes, demostrando el funcionamiento del *ONAT*.

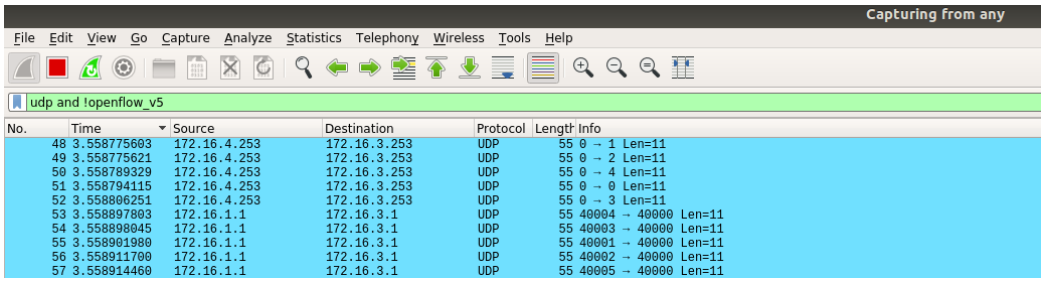


Figura 8.9: Análisis de tráfico en Wireshark: tráfico total por s_3 perteneciente a $ONRP_5$. Traducción ONAT.

Se concluye en esta sección el correcto funcionamiento del algoritmo de encaminamiento y traducción *ONAT* a nivel de plano de datos.

8.2. Pruebas de funcionalidad de ONRApp y PPG

En esta sección se presenta las pruebas de funcionalidad realizadas más representativas para validar el correcto funcionamiento del sistema. Si bien se pretende evaluar la funcionalidad de las piezas de software desarrolladas *ONRApp*, *PPG* y su interacción, interesa analizar el comportamiento de la arquitectura de red completa propuesta en 3.1.

Como se pretende evaluar el sistema completo, la técnica implementada en las pruebas de funcionalidad presentadas en la presente sección son "Test de Caja Negra". Este tipo de pruebas permiten evaluar la respuesta del sistema a partir de una *API*, entradas y salidas esperadas [31], ocultando la ejecución de procesos e interacción entre *ONRApp*, *PPG*, *ONOS* y red subyacente.

Las pruebas de funcionalidad son implementadas mediante la *API* expuesta por *ONRApp*, simulando ser un ente externo perteneciente al plano de aplicación dentro del paradigma *SDN*. El ambiente de estas pruebas esta explicitado en el Anexo B.

El sistema de verificación se realiza usando chequeos automatizados a partir de *scripts* diseñados en lenguaje *Python* que ejecutan peticiones HTTP y comparan las respuestas recibidas con las esperadas.

A las pruebas automáticas se le agregan algunos chequeos manuales para validar la correcta implementación de las políticas de ruteo en el plano de datos. Estos chequeos manuales se realizan mediante la herramienta de análisis de tráfico *Wireshark* tal como se muestra en la sección 8.1.2 y consultas de las *flow entries* instaladas a los OVS virtualizados en *Mininet*.

ONRAapp es diseñada utilizando principios de programación concurrente permitiendo la ejecución de varios procesos como respuesta a eventos externos, ya sean generados por el plano de datos o mediante peticiones por *REST API*. El objetivo de usar *threads* en la ejecución de un programa es optimizar el rendimiento en cuanto al tiempo de ejecución de tareas. La ejecución de estos eventos en paralelo puede generar errores al acceder a variables compartidas de forma simultánea. [32]

Los criterios tomados para la elección de pruebas de funcionalidad son los siguientes:

- Toda funcionalidad expuesta por *REST API* debe tener al menos dos instancias de prueba: una ejecutando peticiones de forma serial y otra usando *threads* para la ejecución en paralelo.
- Todas las funcionalidades expuestas que al ejecutarse acceden a los mismos registros deben ser probadas en una misma prueba, para comprobar que no se generan conflictos.
- Toda función expuesta por *REST API*, en caso de detectar un error debe responder con un código error. Los potenciales códigos de error identificados son expuestos en el Anexo D, Manual de códigos de error.

La topología usada en todas las pruebas de funcionalidad es la mostrada en la figura 8.1 correspondiente a la prueba de concepto. Las principales pruebas ejecutadas son:

- Prueba de registro de *PoPs*
  La prueba consiste en registrar los 4 *PoPs*, inicializando los *Switch OpenFlow* en *Mininet*. Usando *Wireshark* se comprueba que la red puede comunicar correctamente mensajes UDP,TCP e ICMP entre todos los nodos de la red. Por lo que el sistema resulta transparente al funcionamiento normal si no se establece ninguna nueva política en la red. Además, haciendo uso de *Mininet* se verifica que las *flow entries* de la *flow table* 0 fueran correctas

en la inicialización. Detalles de estas *flow entries* se explican en el capítulo 7.

- Creación de *ONRPs*
  La prueba consiste en crear 15 *ONRPs* con distinta cantidad de saltos, parámetros y por ende prioridad, pero con el mismo *PoP* de origen. Se generó tráfico desde los hosts virtualizados en *Mininet* para verificar que este tráfico se encamina correctamente para cada *ONRP* usando *Wireshark*.
- Pruebas de modificación de *ONRP*
  La prueba consiste en crear una serie de *ONRPs*, modificarlas y comprobar el encaminamiento del trafico.
- Prueba funcional del sistema de medición de QoS
  La prueba consiste en inicializar medidas, ejecutarlas y consultar los resultados mediante peticiones HTTP. También se valida el proceso opcional en el cual el resultado es entregado inmediatamente al ser recibido. Tal como se explica en la sección 5.3
- Pruebas de falla de conexión
  La prueba consiste en la instalación de un *ONRP* y la posterior desconexión de un switch perteneciente al *Path*. Se comprueba que los mensajes llegan a destino directamente luego de desconectarse el switch.
- Verificación de códigos de error
  Se realizaron minuciosas pruebas generando variantes de todos los potenciales errores identificados de pasaje de parámetros de todas las funciones expuestas por *REST API*, verificando que se devuelve el código de error y mensaje correspondiente.
- Prueba de registro, obtención y des-registro de *PoPs*
  La prueba consistió en registrar, obtener y borrar 1000 *PoPs*, comprobando que los registros en *ONRApp* queden vacíos luego de la prueba. Los 1000 *PoPs* no fueron simulados en *Mininet* debido que no se cuenta con un equipo que sea capaz de simular 1000 puntos de presencia(*PPG*, *SwitchOpen Flow*, *Router ISP*).
- Pruebas de creación, obtención, modificación y borrado de *ONRPs*
  La prueba consistió en crear, obtener, modificar y borrar 1000 *ONRPs*, comprobando que los registros en *ONRApp* queden vacíos luego de la prueba. También se verifico que todas las *flow entries* fueran instaladas y luego borradas mediante eventos que verifican el estado de las *flow entry* en *ONOS*.

Por intermedio del protocolo de pruebas de funcionalidad se detectaron los siguientes problemas no solucionados en la versión de la aplicación implementada hasta el momento:

- Se genera un error al ejecutar sobre una misma *ONRP* varias peticiones de *Modificar Camino* . La prueba consiste en ejecutar un *script* el cual modifica 100 veces seguidas el mismo ONRP en paralelo usando threads. El problema encontrado es que dos o más threads en la aplicación pedían un identificador en tiempos próximos consumiendo dos identificadores, pero como era el mismo ONRP se termina descartando el primero, por ende, este identificador queda ocupado y sin posibilidad que se libere. Sin embargo, no

genera conflictos en el uso del programa. El hardware donde se ejecuta el *script* se menciona en el anexo B.

- Al crear una nueva *ONRP*, no se consulta si todos los *PoPs* del *Path* se encuentran operativos. Esto no se acopla al diseño inicial mostrado en la figura 5.5.

Una duda que puede surgir respecto al algoritmo diseñado es que ocurre cuando los paquetes recibidos por un *Switch OF* se encuentran fragmentados y por tanto el cabezal de capa de transporte no se encuentre presente en todos. Para ello se realizó la prueba de enviar paquetes con un tamaño total que supere la *MTU* de la red y por tanto sean fragmentados.

En base a esta prueba, se encontró que por defecto los *Switches OF* procesan los paquetes fragmentados que no incluyen el cabezal de capa de transporte asumiendo el valor 0 para todos los campos inexistentes. Como se define en **??**, la solución a este problema es la utilización del modo de funcionamiento "OFPC_FRAG_REASM", el cual permite el reensamblado de paquetes fragmentados previamente a ser procesado por el *pipeline* de *OpenFlow*.

Sin embargo, esta capacidad resulta opcional según la especificación y en particular, los *Switches OF* emulados por *Mininet* no la incluyen y por tanto, no es posible utilizar el algoritmo definido si los paquetes son fragmentados.

El resultado del protocolo de pruebas de funcionalidad fue satisfactorio, sin detectar errores críticos que desestabilicen el funcionamiento del sistema.

Una vez validada la funcionalidad del sistema, se continúa en la siguiente sección con el análisis de *performance* del la aplicación.

8.3. Performance de *ONRApp*

En la presente sección se evalúa el rendimiento de *ONRApp* en relación a la ejecución de procesos más relevantes como registro y borrado de puntos de presencia, creación, modificación y borrado de políticas de ruteo. También se analiza el tiempo de respuesta ante una consulta de información del estado de la red.

Se realiza un análisis detallado con el objetivo de validar el proceso de medición variando el largo de *Path* y el retardo generado en la red subyacente a la *ON*.

Otro parámetro relevante cuantificado en esta sección es el tiempo de respuesta del sistema ante fallas de ruteo.

Es importante tener presente que las pruebas aquí presentadas se realizan en el ambiente de emulación *Mininet*, el cual es ejecutado sobre el hardware presentado en el anexo B. La emulacion permite definir comunicaciones ideales que posibilitan la evaluación del sistema en un ambiente en el que los retardos que este agrega son apreciables. De esta forma, se logra cuantificar características que potencialmente limiten el funcionamiento del sistema en un ambiente real, como por ejemplo, el retardo introducido por *ONRApp* al crear una nueva política de encaminamiento (*ONRP*).

Como el objetivo de esta sección es estimar el rendimiento de la aplicación, se desarrolló una versión de *debug* de *ONRApp* para medir los tiempos de ejecución de las operaciones planteadas. El tiempo de ejecución hace referencia al tiempo desde

que se recibe la solicitud desde *REST API* hasta el momento que se ejecutaron todas las tareas asociadas a esa petición, ya sea respuesta a la petición o momento en el que la aplicación encola mensajes *OpenFlow* al *CORE* de *ONOS*.

A menos que se indique lo contrario, la topología usada en las pruebas es la mostrada en la figura 8.1.

8.3.1. Tiempo de registro y borrado de *PoPs*

En esta prueba se pretende cuantificar el tiempo de demora de la aplicación en registrar, borrar o devolver información relacionada a un punto de presencia de la red. A partir de esta información se puede conocer la demora introducida por la aplicación ante una inicialización. A su vez permite conocer la escalabilidad del sistema.

La prueba consiste en variar la cantidad de *PoPs* registrados de 0 a 1000, y en cada una de estas cantidades adquirir el tiempo de registrar, obtener y borrar un *PoP* en la aplicación. Con el objetivo de obtener muestras significativas se repite la prueba 100 veces.

En la figura 8.10.a se muestra el tiempo que demora la aplicación en registrar un nuevo punto de presencia.

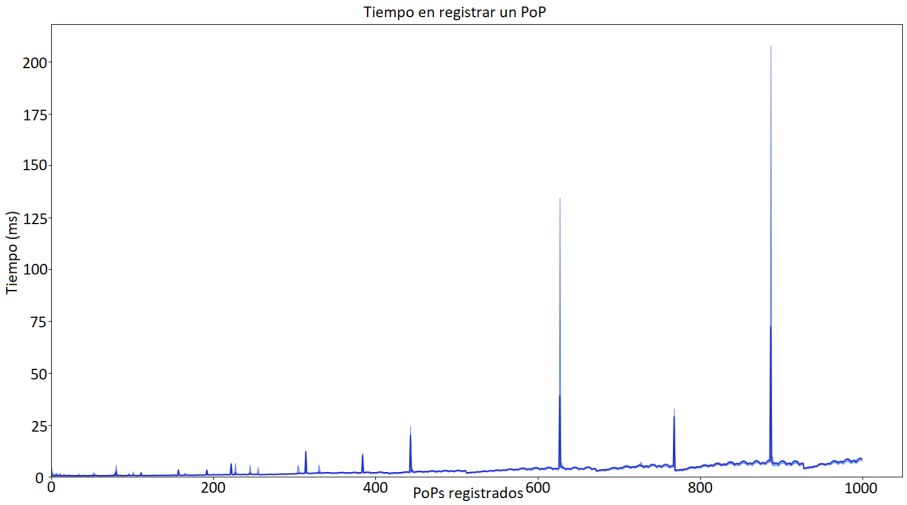
Como se observa en la figura 8.10.a, dada una cantidad de *PoPs* ya registrados en la aplicación, existen puntos particulares en los cuales al registrar un nuevo *PoP*, el tiempo de demora da un salto significativo de al menos un orden de magnitud. Estos saltos siguen un patrón el cual se repite siempre en los mismos puntos de las 100 pruebas, al intentar agregar los *PoPs* 442, 627, 768 y 887.

Para mejor visualización del comportamiento cualitativo del tiempo de registro de *PoPs*, evitando el patrón de saltos de órdenes de magnitud mencionados, se exponen las figuras 8.10.b y 8.10.c.

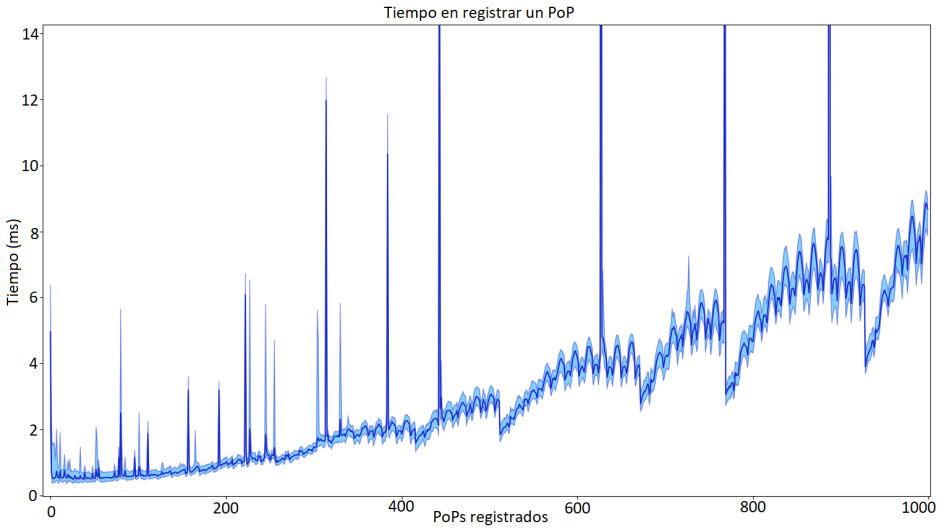
En la figura 8.10.b, se logra visualizar mayor cantidad de saltos que los mencionados anteriormente.

Se identifica un patrón extraño en determinados puntos de la gráfica, como por ejemplo al agregar el *PoP* 769, el tiempo de registro respecto al anterior disminuye hasta un 40 %.

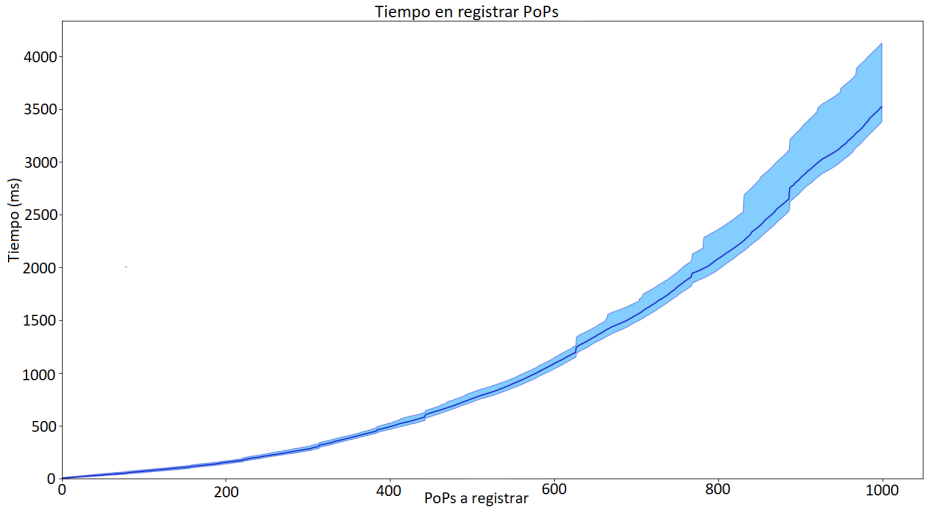
Si bien la función no crece de forma monótona, se observa una tendencia de crecimiento en el tiempo de registro a medida que aumenta en número de *PoPs* registrados en el sistema.



8.10.a-Tiempo de registro de un*PoP*.



8.10.b- Tiempo de registro de un *PoP*.



8.10.c- Tiempo total de registro de *PoPs*

Figura 8.10: Tiempo de registro de nuevos *PoPs* en función del número total del*PoPs*

En particular, la tendencia de crecimiento en media es lineal. Este comportamiento se debe a que la red permitida en la prueba es *Full Mesh*, y por lo tanto al registrar el *PoP* número n, se agregan $2 * (n - 1)$ *Links*.

En la figura 8.10.c se muestra el tiempo de demora total de la aplicación al agregar una determinada cantidad de *PoPs*.

Resulta evidente que el tiempo en función del número de *PoPs* a registrar describe un comportamiento parabólico. Esto se atribuye a que en este caso, dado que la red sobrepuesta es *Full Mesh*, el número de *links* tiene una relación cuadrática con el número de *PoPs*.

Si bien el dominio de las funciones es discreto, la curva de la figura 8.10.c puede verse como la integral de la curva descrita en 8.10.b.

En la figura 8.11 se muestra el tiempo de demora de la aplicación en desregistrar un *PoP*. En esta, el comportamiento cualitativo es similar a el de la figura 8.10.b, con algunas diferencias sustanciales.

La primer diferencia significativa es que luego de 200 *PoPs*, la demora de la acción de registro comienza a crecer de tal forma que para el *PoP* 1000 la diferencia de tiempo entre registrar y deregistrar es del 25 %, con un tiempo de $6ms$ y $8ms$ respectivamente.

La segunda diferencia es que si bien al comenzar a desregistrar los 1000 *PoPs* se ve unos saltos en el tiempo de demora, esto puede atribuirse a un transitorio del inicio de proceso de borrado el cual se estabiliza luego. En cambio a la hora de registrar estos picos se ven durante todo el proceso de escritura.

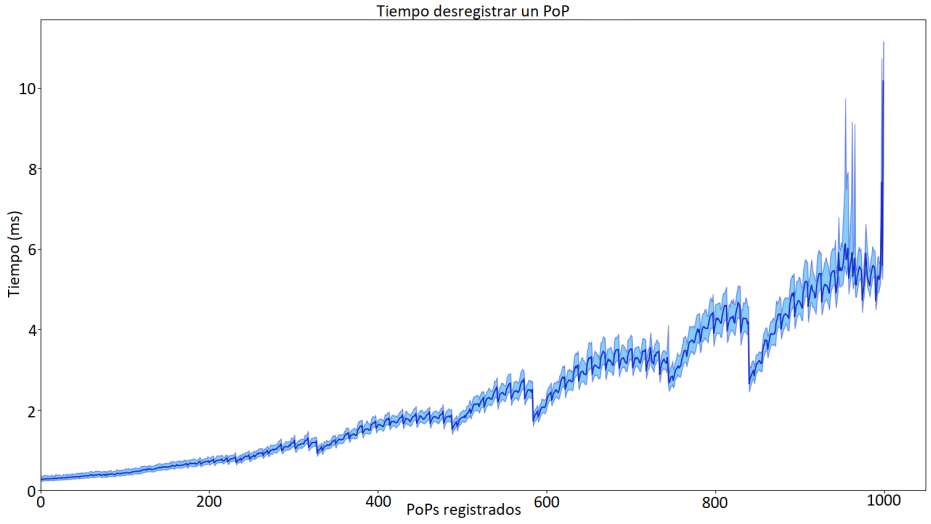


Figura 8.11: Tiempo en desregistrar un *PoP*

Debido a la complejidad del sistema y al desconocimiento de como procede *ONOS* a administrar sus aplicaciones internas, no se encontró una explicación concreta de la aparición de los picos.

Una posibilidad de este comportamiento es el *garbage collector* de *Java* ya que, el uso inadecuado del mismo puede producir este tipo de comportamiento [33]

En la figura 8.12 se muestra el tiempo de procesamiento que presenta la aplicación al recibir una consulta por información de un *PoP* ya registrado.

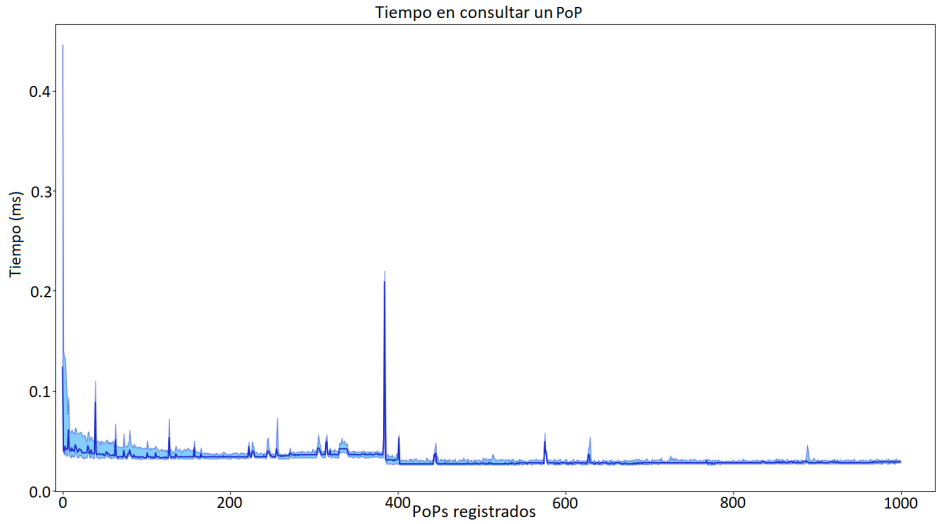


Figura 8.12: Tiempo de respuesta ante una consulta sobre información de un *PoP*

En la figura 8.12 se observa una demora $400\mu s$ al consultar por el primer *PoP* y una varianza mayor respecto a la media cuando el número de *PoPs* registrados varía de 0 a 200. Esto puede atribuirse a un proceso de inicialización del sistema, ya que las pruebas fueron ejecutadas un segundo después que el controlador comienza a correr y levanta la aplicación.

Si bien existe un comportamiento que en principio carece de sentido luego de registrar 384 *PoP*, al cual no se encontró una explicación, donde el tiempo de demora de lectura disminuye de $36\mu s$ a $26\mu s$, el orden de magnitud de este tiempo resulta despreciable respecto al tiempo de registro de nuevos *PoPs*.

8.3.2. Implementación de políticas de ruteo

En el sistema diseñado resulta indispensable cuantificar el tiempo en el cual la aplicación establece nuevas políticas de ruteo, ya sea a través de la creación o modificación de una ya existente.

En las figuras 8.13, 8.14 y 8.15 se muestra el tiempo de creación de una nueva *ONRP*, modificación y borrado de una ya existente en función del número de *ONRP* ya implementadas en la red.

El largo del *Path* de todas las *ONRP* es de 4 *PoPs*. El número de *ONRP* varía desde 0 a 1000. Para obtener valores significativos la prueba fue realizada 100 veces.

Si bien el comportamiento cualitativo de creación y modificación es similar, el cual tiene una tendencia constante al variar el número de *ONRPs*, presentan

diferencias en el valor obtenido. En promedio, el tiempo de creación de nuevas políticas de ruteo introducido por la aplicación se encuentra en 4,4ms, mientras que el percentil 10 a 90 se encuentra entre 4,1ms a 4,6ms tal como se muestra 8.13. Los resultados presentados en 8.14 muestran un mayor tiempo de ejecución en la modificación de una *ONRP* ya existente respecto a la creación, donde el promedio se ubica en 6,0ms, y el percentil 10 a 90 es de 5,0ms y 8,3ms.

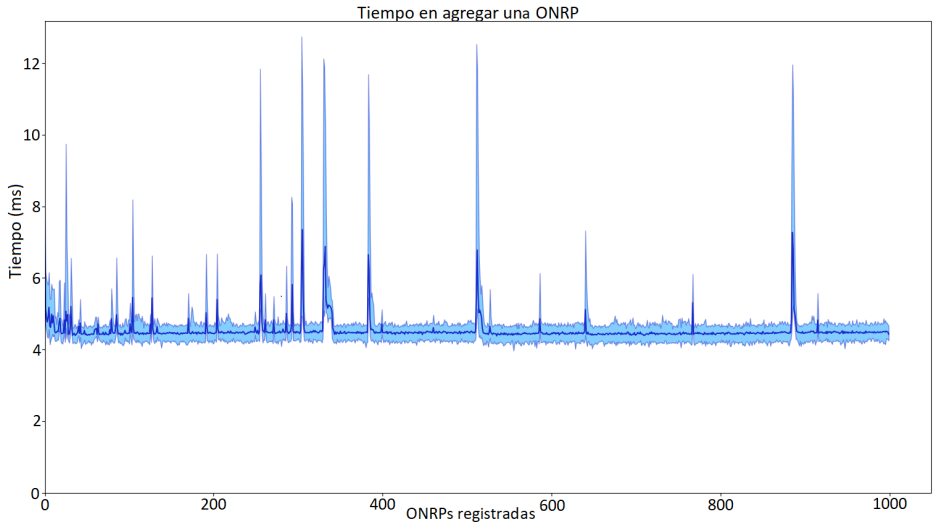


Figura 8.13: Tiempo de crear de un *ONRP*

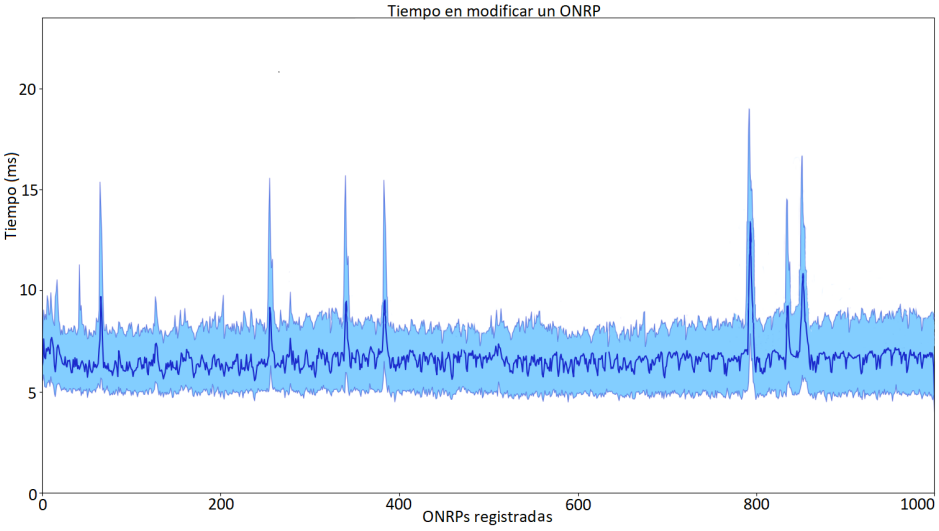


Figura 8.14: Tiempo en modificar un *ONRP*

En la figura 8.15 se muestra el tiempo de borrado de políticas en función del número de políticas implementadas en la red.

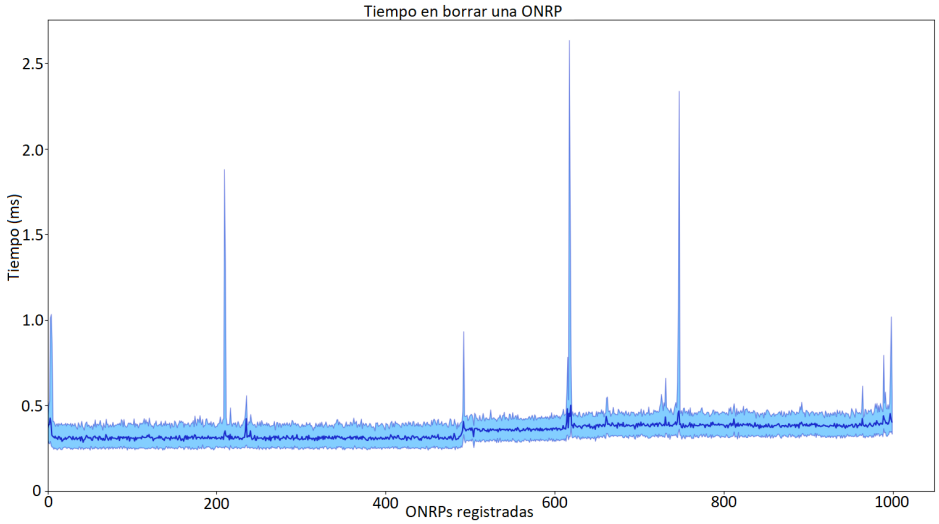


Figura 8.15: Tiempo en borrar una *ONRP*

Como se observa en 8.15 los tiempos de borrado son un orden de magnitud menor a los de creación y modificación, donde el percentil 10 y 90 son de 0,3ms y 0,4ms respectivamente.

Como se explica en el capítulo 7, el proceso de modificación fue realizado como una optimización respecto al proceso de creación de una *ONRP*, modificando el *Path*, manteniendo los otros parámetros de una *ONRP* ya creada constantes. Este proceso implica una secuencia de borrado y creación de una *ONRP* más procesos agregados. Por lo tanto, resulta coherente que el tiempo de borrado sumado al de creación sea siempre menor que el tiempo que consume la aplicación en ejecutar el proceso de modificación.

Para finalizar la prueba de rendimiento de la aplicación en relación a la implementación de políticas se muestra en la siguiente figura el tiempo de respuesta ante una solicitud de información de una *ONRP* implementada:

Todas las acciones que ejecuta la aplicación en torno a la implementación de *ONRPs* presentan un comportamiento similar. El tiempo de demora no varía respecto a la cantidad de políticas implementadas en la red.

Si bien en las gráficas presentadas también se observan puntos de grandes saltos en el tiempo de ejecución tal como sucede en el proceso de registro de *PoPs*, estas anomalías presentan saltos de 3 veces el valor esperado, mientras que en operaciones de registro de nodos en la aplicación los saltos son de al menos 1 oren de magnitud.

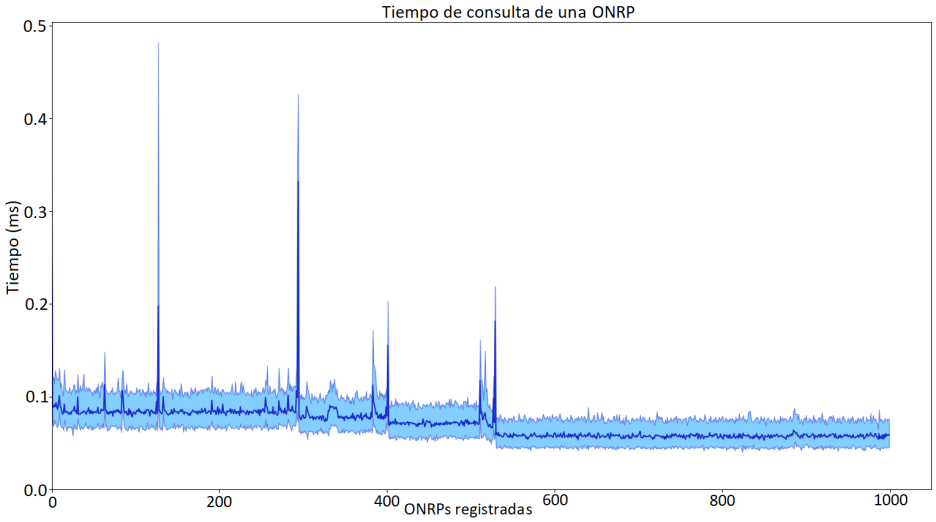


Figura 8.16: Tiempo en obtener la información de una *ONRP*

8.3.3. Sensibilidad del sistema de medición

Con el objetivo de validar el sistema de medición de QoS, se compara el método de medición diseñado utilizando la métrica RTT con el envío de mensajes ICMP para la obtención de este parámetro.

En esta prueba se genera tráfico desde el mismo host de origen hacia el mismo host destino, haciendo uso de el software *PPG* y la funcionalidad *ping* de *Linux*.

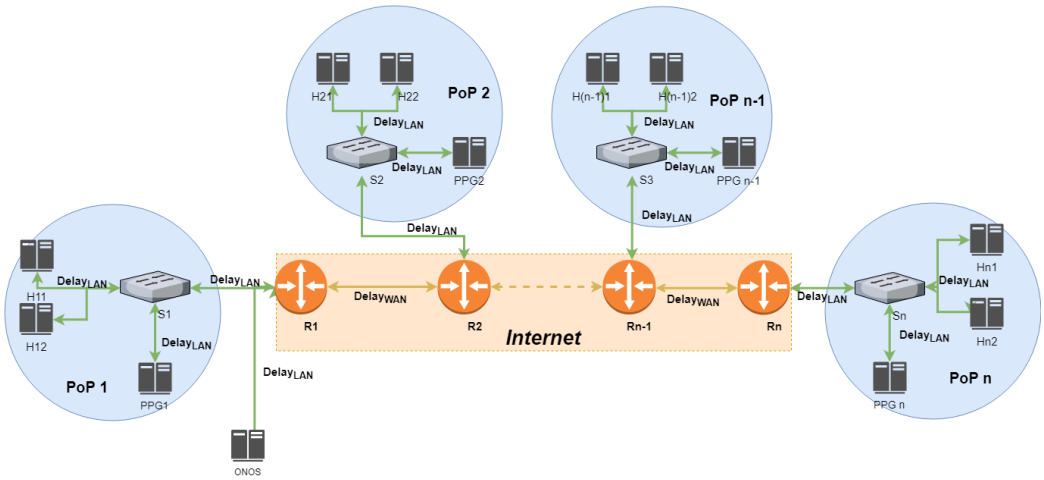


Figura 8.17: Topología utilizada en prueba de sensibilidad del sistema de medición.

Se optó por la topología mostrada en la figura 8.17 de forma que los paquetes ICMP recorran la misma ruta (en lo que simula ser Internet) que los *probes* del *PPG*. En esta, se hace referencia al tiempo de *delay* en enlaces entre routers de

los *ISPs* con el nombre de $Delay_{WAN}$ y para enlaces internos a los *PoPs* como $Delay_{LAN}$.

En la topología introducida anteriormente, la expresión teórica de RTT entre un host en PoP_1 y otro en PoP_N para cada método es:

$$RTT_{ping} = 2(d_W(N-1) + 4d_L + TP_R * N + 2\ TP_S) \tag{8.1}$$

$$RTT_{PPG} = 2(d_W(N-1) + 4d_L + 2d_L(N-2) + TP_S * N + TP_R(2N-2)) \tag{8.2}$$

Donde:

- $d_W = Delay_{WAN}$
- $d_L = Delay_{LAN}$
- $N = número\ de\ PoPs\ en\ el\ Path$
- $TP_R = tiempo\ de\ procesamiento\ en\ router\ de\ ISP$
- $TP_s = tiempo\ de\ procesamiento\ de\ un\ Switch\ OpenFlow$

La diferencia teórica entre el *ping* y el método de medición implementado radica en que el paquete enviado por el *PPG* debe pasar por el switch de todos los *PoPs*. Esto explica que en la segunda ecuación se agreguen el *delay* de los *links* de la LAN y el procesamiento interno en los *switch intermedios*.

La diferencia teórica de tiempo entre ambos métodos se resume en la siguiente expresión:

$$RTT_{Diff} = RTT_{PPG} - RTT_{ping} = 2(N-2)(2d_L + (TP_R + TP_s)) \tag{8.3}$$

Protocolo de prueba:

1. Inicializar una medida desde *REST API* para luego ejecutarla desde el *PPG* del PoP_1. Se asume que todos los *PoPs* se encuentran configurados, registrados y conectados al controlador con sus respectivos *PPGs*.

2. Una vez inicializada la medida, se ejecuta desde *REST API* una petición de ejecución de medida.

3. Se obtiene el resultado de la medida y se procede a medir enviando paquetes ICMP.

La prueba consiste en variar el número de puntos de presencia y el $delay_{WAN}$ introducido por los enlaces simulados en *Mininet*, relevando resultados obtenidos del RTT medido por el sistema y a través de paquetes ICMP. El $delay_{LAN}$ se configura en $1ms$.

En las tablas 8.2 y 8.3 se exponen los resultados obtenidos en *ONRApp* y la diferencia obtenida respecto al método utilizando ping. En base a estos es posible observar la sensibilidad del sistema propuesto en el capítulo 3 frente a *delays* agregados en las redes internas de los *PoPs*, a *delays* de Internet y a los tiempos de procesamiento de los routers de *ISPs* y los *Switch OpenFlow*.

Tabla 8.2: *RTT* obtenido de *ONRApp*

delay$_{WAN}$(ms) \ N$_{PoPs}$	2	3	4	5	6	7	8	9	10
10	28.81	53.67	78.03	102.09	126.45	150.91	175.32	199.72	224.78
20	48.84	93.39	137.82	182.11	225.67	270.04	314.43	358.53	404.13
50	108.7	213.73	317.73	421.62	525.68	630.49	733.15	839.25	944.76
100	208.92	413.3	617.77	821.52	1025.72	1229.84	1432.99	1639.01	1844.46

Tabla 8.3: Diferencia entre *RTT* obtenidos desde *ONRApp* y mensajes ICMP

delay$_{WAN}$(ms) \ N$_{PoPs}$	2	3	4	5	6	7	8	9	10
10	0.18	4.55	8.75	12.67	16.87	21.12	25.8	29.76	34.98
20	0.12	4.42	8.68	12.93	16.92	21.23	25.66	29.61	35.15
50	0.08	4.72	8.81	12.89	17.14	20.94	24.48	30.12	35.31
100	0.11	4.46	8.93	12.58	16.4	20.86	23.81	30.26	35.26

Como se observa en la tabla 8.3, cuando el numero de *PoPs* es 2, los resultados muestran específicamente la *performance* de la plataforma de medición, que incluye el software de medición *PPG* y la aplicación *ONRApp* presentada en el capítulo 5.

El aumento de la diferencia a medida que aumenta la cantidad de *PoPs* se ve reflejado en la expresión de la ecuación 8.3, en la cual se observa que el retardo agregado aumenta linealmente con la cantidad de *PoPs* en el *Path*.

8.3.4. Toleracia a fallas de ruteo

En esta sección se presenta cuantitativamente el tiempo de detección y corrección de fallas del sistema. En esta versión de la aplicación, el evento crítico al que el sistema responde de forma autónoma es la pérdida total de conexión con un *PoP*. Se considera una pérdida de conexión de un *PoP* cuando se pierde la conexión TCP establecida entre el *Switch OpenFlow* y el controlador. Vale la pena destacar que existen dos situaciones en las cuales el controlador pierde conexión con el switch:

1. Un agente externo decide administrativamente apagar el *Switch OpenFlow*, evento que no es ejecutado desde el controlador.

2. Se rompe el canal de comunicación por fallas producidas en la red.

La topología utilizada en esta prueba se muestra en la figura 8.1

Protocolo de prueba:

1. Se configura una política cuya *Path* se especifica en la figura 8.1 con color verde

2. De forma de cuantificar el tiempo de respuesta del sistema independizando al mismo del *delay* de la red, se configuran todos los enlaces con *delay*=0, con excepción del enlace R_1-R_3 y R_2-R_3. Estos últimos se configuran con un *delay* de $20ms$.

3. Se envían paquetes cuyo tráfico coincide la *ONRP* con una cadencia de $100\mu s$ desde un host de origen ubicado PoP_1 a un host ubicado en el PoP_3.

4. Los mensajes enviados tienen información del tiempo en que se envió el mensaje y un identificador del paquete el cual va aumentando en uno por paquete enviado. Estos mensajes utilizan como protocolo de capa de transporte UDP.

5. El host ubicado en el PoP_2 al recibir cada paquete toma el tiempo en el que fue enviado, lo compara con su reloj actual y almacena este resultado junto con el identificador del paquete. Observar que el reloj interno de ambos hosts es el mismo ya que se encuentran virtualizados en el mismo sistema operativo, por lo tanto el resultado almacenado es el *delay* introducido por el enlace.

6. Una vez el tráfico está establecido, desde *Mininet* se apaga switch virtualizado s_2 y se cuenta cuantos paquetes se perdieron debido a la caída del *PoP2*. La desconexión del switch se genera de dos formas: enviando un comando de apagado y luego se repite la prueba eliminando la interface del switch.

La elección del *delay* en los enlaces mencionados se realiza buscando cuantificar el tiempo de respuesta del sistema independizando la medida de la red subyacente. Esto se logra minimizando lo máximo posible el *delay* entre el controlador y s_2 para disminuir el tiempo de detección del problema en la aplicación y el *delay* entre controlador y s_1 para reducir el tiempo que demoran en transitar los mensajes de control para encaminar el tráfico por el camino directo una vez detectado el problema.

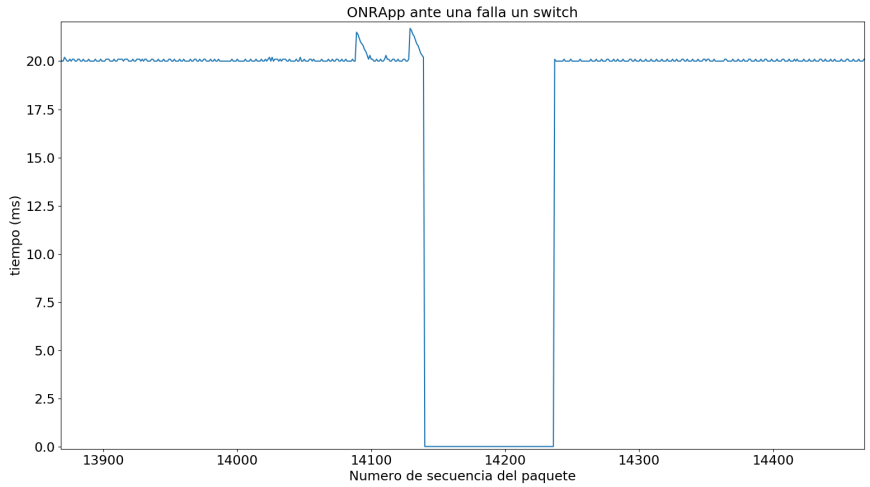


Figura 8.18: Tiempo de recuperación ante la desconexión del un switch generado administrativamente. *Delay agregado en la red para cada paquete enviado 20ms*

Desconexión de un Switch OpenFlow generada administrativamente

En este caso se analiza la respuesta del sistema ante la desconexión de un switch generada de forma administrativa a partir de la ejecución de una línea de comandos en su terminal, siguiendo el protocolo de pruebas mencionado.

En la figura 8.18 se muestra el resultado de la prueba. Se observa que el delay calculado por el receptor es de $20ms$, el cual coincide con el tiempo configurado en *Mininet* y la pérdida de 101 paquetes desde el momento que cae el punto de presencia hasta que vuelven a llegar los paquetes al receptor.

Se repite el protocolo de pruebas 10 veces. El promedio de pérdida de paquetes en las 10 pruebas realizadas es de 105 paquetes perdidos.

Dada la cadencia de envío especificada en el protocolo de prueba, se estima en media que el tiempo de recuperación del sistema ante este tipo de fallas es de $10{,}5ms$.

En la figura 8.19 se muestra una captura de tráfico realizada con Wireshark en la interface del controlador.

```
   9038 21.761385865  172.17.0.1       172.16.2.102     OpenFl…   146 Type: OFPT_FLOW_MOD
   9039 21.761405913  172.17.0.1       172.16.2.102     OpenFl…   178 Type: OFPT_FLOW_MOD
   9040 21.761450898  172.17.0.1       172.16.2.102     OpenFl…   154 Type: OFPT_FLOW_MOD
   9041 21.761546962  172.17.0.1       172.16.2.102     OpenFl…   162 Type: OFPT_FLOW_MOD
   9042 21.761621355  172.17.0.1       172.16.2.102     OpenFl…   162 Type: OFPT_FLOW_MOD
   9043 21.761647287  172.17.0.1       172.16.2.102     OpenFl…   162 Type: OFPT_FLOW_MOD
   9155 21.783513438  172.17.0.1       172.16.2.102     OpenFl…   898 Type: OFPT_FLOW_MOD
   9267 21.803511740  172.16.2.102     172.17.0.1       TCP        66 54800 → 6633 [ACK] Seq=49273 Ack=14153 Win=1224 Len=0 TSval
   9269 21.803674619  172.16.2.102     172.17.0.1       TCP        66 54800 → 6633 [RST, ACK] Seq=49273 Ack=14153 Win=1227 Len=0
```

Figura 8.19: Tráfico generado entre *Switch OpenFlow* y controlador al momento de una desconexión generada de forma administrativa.

Como se puede observar, al apagar el *Switch OpenFlow* de forma administra-

tiva, el último mensaje enviado al controlador es un mensaje TCP con los flags RST=1, ACK=1 y mismo número de secuencia que el paquete TCP enviado anteriormente, informando al controlador que se da por finalizada la conexión. Si bien resulta interesante contemplar este hecho, esta no es la falla de ruteo con mayor relevancia, debido a que este evento es generado por un ente que tiene acceso a la configuración del switch, y a su vez este antes de apagarse notifica al controlador. A continuación mostramos el caso de mayor importancia.

Desconexión de un PoP

Este caso es el de mayor importancia, dado que muestra el tiempo de respuesta del sistema ante una falla inesperada generada en algún punto de la red. En este caso, no se apaga el switch por intermedio de línea de comandos, si no que se le borra al mismo la interface que le da conexión con el router del ISP.

En la figura 8.20 se muestra el tiempo de recuperación del sistema ante este tipo de problemas

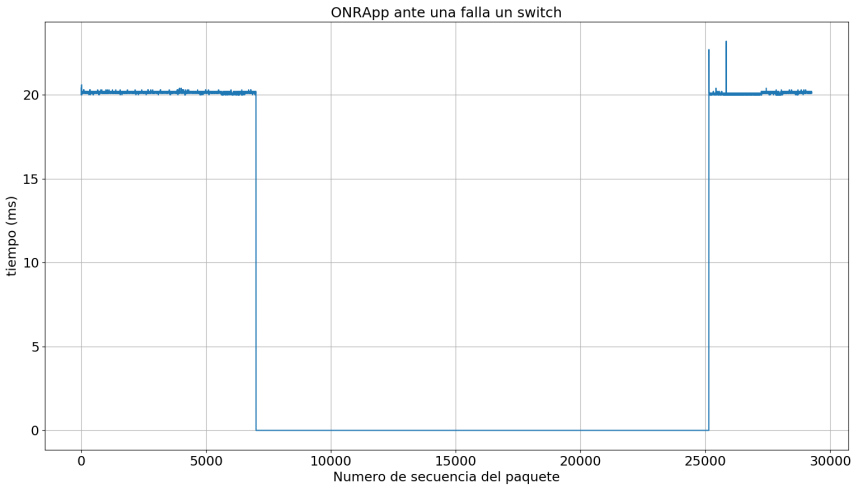


Figura 8.20: Tiempo de recuperación ante una falla en un *PoP*. *Delay agregado en la red para cada paquete enviado 20ms*

En la figura se muestra una pérdida total de 17619 paquetes. En este caso la tasa de transmisión es de *1ms*, por lo tanto, el tiempo de respuesta del sistema en este caso es de *17.62ms*. En promedio el resultado de todas las pruebas es de un tiempo de respuesta de *17.3ms* aproximadamente. Si comparamos el tiempo de respuesta en este caso respecto al caso en que se apaga el switch administrativamente, la diferencia del tiempo de respuesta es de 3 órdenes de magnitud. En este caso, el tiempo de respuesta es significativo. El motivo de este tiempo de respuesta se observa en la figura 8.21:

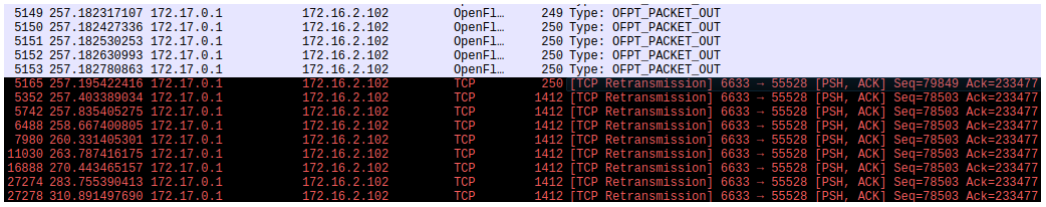


Figura 8.21: Wireshark que muestra los mensajes recibidos por el switch 2 por la interfaz de control cuando se pierda la comunicacion con el controlador

En la figura se observa que al fallar el switch, dado que el controlador en el momento antes de la falla se encontraba intercambiando información, al no recibir los reconocimientos antes de que expire el temporizador, comienza el envío de retransmisiones. La primer retransmisión la hace enviando un paquete con número de secuencia 79849. Dado que tampoco llegó el reconocimiento del paquete anterior, retrasmite 0.2 segundos despues un paquete con número de secuencia menor, en este caso número de secuencia es 78503. Comienza el proceso de retransmisión, duplicando el tiempo de expiración del temporizador en cada retransmisión, suceso esperable. Se observa que el tiempo de respuesta del sistema cae dentro del intervalo entre que se ejecuta la séptima y la octava retransmisión, momento en el cual se genera un evento a nuestra aplicación para tomar las acciones necesarias.

Luego de realizadas las pruebas de concepto planificadas se valida el algoritmo de encaminamiento implementado en la sección 3.2.

Las pruebas de funcionalidad de *ONRApp* no presentan problemas críticos para la utilización de la aplicación con bloques externos que deseen hacer uso de las funcionalidades de red que expone debido a la coherencia de las respuestas a las peticiones y la correcta comunicación con el plano de datos.

Respecto a las pruebas de rendimiento, el tiempo de registro de nuevos puntos de presencia, en el contexto en el que se realiza la medida, si bien aumenta con el número de *PoPs*, es menor a $10ms$ para menos de 1000 *PoPs*.

A diferencia del registro y borrado de *PoPs*, el tiempo en implementar nuevas políticas de ruteo en la red no varía en función del número de políticas implementadas. En este caso, todos los resultados también dieron un retardo en la ejecución menor a $10ms$.

Las pruebas de lectura a memoria, como por ejemplo consultas sobre *PoPs* y *ONRPs* almacenadas en la aplicación es al menos un orden de magnitud menor que todas las funciones expuestas que requieren escritura a memoria.

En las emulaciones realizadas durante las pruebas, fue posible eliminar los retardos que presenta Internet. Esto posibilitó la evaluación del sistema en un ambiente en el que los retardos que este agrega sean apreciables. De esta forma, se logró obtener características que potencialmente limiten la *performance*, como por ejemplo, que la latencia de crear una nueva política de encaminamiento (*ONRP*) sea de aproximadamente *4,4ms*.

Sin embargo, se debe considerar que *ONRApp* surge de la necesidad de encaminar y medir la QoS de una *ON* sobre Internet y en particular para cuantificar

el tiempo de transito del tráfico entre nodos de la red. Dado que los resultados obtenidos de rendimiento de las funcionalidades analizadas en este capítulo son menores a $10ms$ y la latencia media existente en Internet es de aproximadamente un orden mayor [34], se puede considerar que esta primera versión de la aplicación es funcional para el contexto de trabajo.

Capítulo 9

Conclusiones y trabajos a futuro

9.1. Conclusiones

Este documento presentó el trabajo realizado durante el proyecto de grado "*Overlay Network Routing Application: ONRApp*.

Se expone el estudio de *SDN*, un paradigma que plantea una reestructuración de las redes de datos que promete simplificación y automatización a través de la separación del plano de datos del plano de control.

Dentro del paradigma se presenta el protocolo *OpenFlow*, protocolo con mayor estandarización para la comunicación entre los planos que define el paradigma.

En este proyecto las ideas expuestas por el paradigma *SDN* no solo fueron utilizadas durante el desarrollo de la aplicación de encaminamiento en una *ON*, sino también para la creación del software de medición de QoS *(PPG)*. En él, se aprecian las ideas de control centralizado y funcionalidad distribuida directamente en las bases de su creación, como fue expuesto en 3.3 y en 6.

Mientras el *PPG* en sí mismo es capaz de realizar mediciones de QoS (funcionalidad), la decisión de realizar la medida y los parámetros asociados a la misma se generan en la aplicación *ONRApp* (elemento de control centralizado).

El protocolo *OpenFlow* fue de suma importancia para lograr traducir los diseños realizados en el capítulo 3 a una implementación práctica, como se explica en el capítulo 7. Esta implementación ejecutada por *ONRApp*, realiza configuraciones en los elementos de red de forma dinámica y automática permitiendo el encaminamiento de tráfico evitando arduas y complejas intervenciones por parte de los administradores.

En el capítulo 1 se presentaron soluciones actuales al problema de encaminamiento sobre una *ON*. Sin embargo, se explica que estos métodos poseen diversas desventajas como la intervención de los *ISPs* en el caso de MPLS, que implica costos operativos agregados, la disminución de la MTU en MPLS o VPNs, que aumenta la probabilidad de fragmentación de paquetes y por consiguiente, disminuyendo la QoS o la dificultad administrativa nuevamente en el caso de VPNs.

El algoritmo de identificación y encaminamiento de flujos de datos sobre una *ON* diseñado en el capítulo 3 soluciona los problemas anteriores a través del uso

de identificadores que son instalados en los puertos de capa de transporte de cada paquete que transita la *ON*. Este tipo de identificación, en conjunto con la arquitectura planteada por el paradigma *SDN*, permite independencia frente a los *ISPs* y a su vez que la MTU del paquete se mantenga incambiada. Además, de no ser por las restricciones que impone el protocolo *OpenFlow*, el algoritmo resulta escalable frente a otros protocolos de capa de transporte que definan el concepto de puertos.

Sin embargo, el algoritmo planteado tiene ciertas limitaciones teóricas:

- Cantidad máxima de *ONRPs* por *Link*
  El algoritmo planteado utiliza el campo puerto origen de capa de transporte para la identificación del *Path* a seguir. Por esta razón, existe la limitante teórica de 2^{16} *ONRPs* que comparten un mismo *Link* de forma simultánea.
- Cantidad máxima de flujos por *ONRP*
  El algoritmo planteado utiliza el campo puerto destino de capa de transporte para la traducción de la información de los flujos de datos en el ingreso y egreso a la *ON*. Esto implica que el tamaño de este campo limita la cantidad de flujos de datos que pueden coexistir simultáneamente en una *ONRP* a 2^{16}.

Por otra parte, este proyecto se estructuró en base a los resultados obtenidos en trabajos relacionados [2, 35]. En estos se propone el encaminamiento de paquetes sobre una *ON* superpuesta a Internet basado en la modificación de la dirección IP destino. Sin embargo, se dejan abiertos varios problemas:

- Identificación de flujos
  La propuesta no incluye un método concreto para la identificación de más de un flujo de datos que se deseara encaminar. En este proyecto, este problema fue abordado y solucionado con el algoritmo aquí presentado.
- Problemas de *reverse path filtering*
  Si bien se plantea un cambio de dirección de destino, no se especifican acciones concretas (solo se sugiere) sobre la dirección IP origen. En las pruebas de concepto realizadas en 8, los routers emulados en la topología implementan este filtro y considerando los resultados obtenidos, se concluye que este problema fue solucionado.
- Asociación de una dirección IP al *Switch OpenFlow*
  En los trabajos relacionados se plantea la imposibilidad de configurar una dirección IP a los *Switch OpenFlow*. Si bien esto no es posible utilizando un switch por sí solo, gracias al servicio *Centralized ARP* implementado, el conjunto *Switch-Controlador* puede responder consultas *ARP* utilizando paquetes *PACKET_OUT* y de esta forma, se simula la asignación de la IP al *Switch OpenFlow*.

Con respecto a la aplicación *ONRApp* desarrollada, en los resultados presentados en el capítulo 8 donde se mide el tiempo de ejecución de las funcionalidades expuestas por la aplicación, se obtiene un tiempo de demora de todas las acciones menor a $10ms$. Dado que el contexto en el que se desarrolla este proyecto es Internet, si bien los tiempos obtenidos en las pruebas no son despreciables, son de al menos un orden de magnitud menor que las latencias existentes en esta red [34].

Esto se traduce en que, a pesar de que se podría mejorar aún más la *performance*, la aplicación desarrollada no introduce retardos significativos, por lo tanto puede dar grandes beneficiosos de automatización y medición de *QoS* brindados.

En el capitulo 1 también se menciona la complejidad de realizar mediciones de QoS en un ambiente *SDN*. Gracias a la existencia de los *PPGs*, presentados en el capítulo 6 como sistema de medición distribuido, los resultados obtenidos en el capítulo 8 para la obtención de QoS basada en *RTT* resultan muy similares al *RTT* medido por la funcionalidad *ping* basada en el protocolo *ICMP*. Por esta razón, se considera validada la plataforma de medición dentro del ambiente de simulación.

9.2. Trabajo a futuro

En esta sección se describen las posibles mejoras a futuro que pueden agregarse al desarrollo expuesto en este documento.

ONRApp v1.1

En una nueva versión de *ONRApp* pueden considerarse los siguientes aspectos:

- Optimización de código
  En el desarrollo de *ONRApp* se tuvieron en cuenta aspectos relevantes como el rendimiento, utilización de memoria y consumo del CPU. Sin embargo no se dedicó un periodo exclusivamente a optimización de código fuente.
  Si bien los resultados expuestos en 8.3 garantizan una primera versión funcional de la aplicación. Se considera que se pueden mejorar estos resultados.
- Corrección de errores
  En la sección 8.2 se exponen los errores encontrados en el proceso de *test* de funcionalidad. Uno de los casos presenta una situación particular en una mala utilización de la función *ModONRP* expuesta por *REST API*, que si bien resulta compleja de solucionar, no es crítica. Sin embargo el otro error detectado sí puede generar problemas.
  Este último fue corregido en la versión de *debug* de la aplicación. Se entiende que, en una nueva versión vale la pena su corrección.
- Agregado de funcionalidades al sistema de medición
  - Actualmente solo se permite en el pasaje de parámetros por *REST API* la especificación de un *Path* de ida, a partir del cual también se establece el la vuelta.
    En versiones futuras este debe ser el parámetro por defecto, permitiendo especificar un *Path* de retorno distinto al de ida.
  - Agregado de nuevas técnicas de medición de QoS Actualmente la única técnica considerada es *RTT*. Se propone el agregado de otras técnicas como *jitter* y *throughput*.
  - Conexión TCP para comunicación con *PPG*
    En esta versión, por simplicidad, la conexión con los *PPG* para establecimiento e intercambio de peticiones y respuestas de mediciones de

QoS se realiza sobre tráfico UDP.
Esto presenta casos de borde como por ejemplo, ¿qué ocurre si no recibo un reconocimiento de un *PPG* al enviar una petición?, ¿qué ocurre si se pierde una respuesta con el resultado de una medida?. Claramente todos estos problemas se solucionan utilizando en capa de transporte un protocolo confiable. Se propone la utilización de TCP para las interacciones entre el conjunto central y conjunto distribuido del sistema de medición.

- Seguridad
  En esta nueva versión no se contemplan temas de seguridad en la aplicación. Dada la arquitectura de capas se propone incluir un nuevo bloque que contemple estos aspectos. Algunas ideas son:
  - La funcionalidad por defecto considerada en la aplicación es que si un paquete ingresa a un *Switch OpenFlow* perteneciente a un *PoP* que no presenta coincidencias, este sea reenviado sin aplicarse ninguna acción. Podrían establecerse filtros que a partir de *ONRPs* solo permitan el pasaje de algún tipo tráfico particular. Si se utilizan *Switches OpenFlow* implementados sobre hadware dedicado al reenvío de paquetes y enlaces de gran capacidad en la conexión a Internet, la utilización de filtros podrían llegar a hacer frente a un problema conocido como "Distributed Denial of Service", descartando rápidamente el tráfico que no caiga dentro de los filtros establecidos.
  - Conexión segura con los *PPG* basada en el protocolo "Transport Layer Security (TLS)"
- Agregado de nuevos protocolos de capa de transporte
  El diseño del algoritmo de encaminamiento propuesto en la sección 3.2 puede aplicarse a todo tipo de tráfico que contemple la utilización de puertos en capa de transporte. Si bien la aplicación contempla únicamente tráfico UDP y TCP debido a que *OpenFlow* permite el tratamiento de otros protocolos, puede extenderse una vez se tengan en consideración en nuevas versiones de *OpenFlow*.

PPG

Si bien el *PPG* es una parte importante de la arquitectura del sistema presentada en 3.1 y se obtuvieron resultados comparables con herramientas de test de *RTT*, en esta tesis se buscó una pieza de software funcional con el fin de validar el sistema de medición propuesto en 3.3. Se propone:

- Nuevo diseño de arquitectura de software
  Si bien la arquitectura planteada en 6.2 considera la posibilidad de agregar nuevas funcionalidades y técnicas de medición, se podría evaluar una nueva arquitectura de forma de mejorar la modularidad del software.
- Implementación en nuevo lenguaje de programación
  Como el objetivo del *PPG* en esta versión fue validar el sistema de medición distribuido, se buscó una versión funcional fácil de implementar. Para mejorar el tiempo de procesamiento puede implementarse en otros lenguajes como C o C++.
- Utilización de herramientas ya testeadas para medición de nuevos parámetros de QoS.
- Refinamiento del protocolo de comunicación *PPG-Controlador* tanto en capa de aplicación como en capa de transporte utilizando TLS sobre TCP.
  En base a la misma idea que con *Switches OpenFlow*, puede implementarse un protocolo mediante el cual el *PPG* sea quien comience la comunicación con el controlador.

Test de integración

- Integración de la plataforma con bloques de ingeniería de tráfico y monitoreo
  Recordar que esta aplicación no toma decisiones de políticas de encaminamiento y medición, si no que brinda estas funcionalidades para ser utilizadas por agentes externos. Se propone probar el sistema diseñado integrando a las pruebas entidades que trabajan en capa de aplicación del paradigma *SDN*.
- Repetir las pruebas de funcionalidad y *performance* llevando el plano de datos a un ambiente real. Agregar pruebas que determinen los tiempos de puesta en marcha o modificación de una *ONRP*.

Investigación a futuro

- Abordar nuevas investigaciones sobre posible identificación de tráfico en redes sobrepuestas a ambientes multidominio basado en el protocolo IP versión 6.
- Evaluar la implementación con el controlador *In-Band* [10]
- ¿Vale la pena implementar esta aplicación en *OpenDaylight*?

Esta página ha sido intencionalmente dejada en blanco.

Apéndice A

Elección del controlador

El presente anexo tiene como enfoque dar a conocer los parámetros determinantes al momento de la elección de un controlador si se quisiese trabajar bajo un ambiente SDN. Se analizan distintos controladores **a partir de los resultados que obtuvieron en trabajos relacionados**. Este anexo es una descripción detallada del proceso de elección del controlador. [24–27]

A.1. Aspectos relevantes

Todo controlador presenta un conjunto de características llamativas que impulsan a los administradores de red a su elección. En esta sección se presenta el conjunto de características con mayor importancia para elegir el controlador. El conjunto de características aquí presentado se basa en los criterios utilizados en trabajos relacionados. [24, 26]

1. Lenguaje de programación:
   El lenguaje de programación en el que se desarrollan tanto los controladores como módulos del mismo debe tener en cuenta:

   a) Posibilidad para correr en varias plataformas.

   b) *Multithreading*

   c) Facilidad de aprender

   d) Velocidad de acceso a memoria

   e) Calidad de manejo de memoria

2. Soporte *OpenFlow*:
   Hasta el momento, es el protocolo estandarizado para APIs entre el controlador y el plano de datos. A su vez se debe tener presente la versión del protocolo que soporta el controlador, ya que nuevas versiones admiten nuevas funcionalidades.

3. Programabilidad de Red:
   Es la esencia de las redes definidas por software y refiere a la capacidad de programabilidad del controlador a partir de scripts que permitan que este funcione dinámicamente y de forma autónoma.
   La existencia de interfaces que permitan la ejecución de aplicaciones permite extender las funcionalidades del mismo. Es por esto que se debe tener en cuenta las funcionalidades de red que exponen los controladores, así como aplicaciones ya creadas.

4. Seguridad:
   Las decisiones en SDN son tomadas por un controlador central. Por ende es necesario que este tenga herramientas seguridad para que no se produzca ninguna falla de la ON.

5. Escalabilidad:
   Un parámetro de medida de escalabilidad es el número máximo de dispositivos (nodos) que un controlador puede manejar de forma simultánea sin degradar su rendimiento. Idealmente, el número de dispositivos no debería ser relevante en el rendimiento del controlador. Sin embargo, la capacidad de administrar de forma simultánea determinado número de dispositivos disminuye a medida que la cantidad aumenta. Esto se debe a que se necesita mayor capacidad de procesamiento para manejar todos los dispositivos.
   La escalabilidad se puede dividir en tres temas:

   a) Capacidad de sesiones de control:
   Mide el número máximo de sesiones de control que el controlador puede establecer de forma simultánea.

   b) Descubrimiento del tamaño de la red:
   Mide el tamaño máximo de la red (cantidad de nodos, enlaces y host) que el controlador puede reconocer. Esto resulta clave al momento de diseñar una infraestructura de red completa.

   c) Capacidad de la tabla de reenvío:
   Número máximo de entradas de flujos que un controlador puede gestionar sin necesidad de descartar ninguna orden de configuración.

6. Confiabilidad:
   Los parámetros relevantes para medir la confiabilidad son:

   a) Tiempo de conmutación en caso de fallo del controlador:
   En caso de estar trabajando con redundancia, en donde se dispone de al menos un controlador de respaldo, es el tiempo que demora este en estar disponible.

   b) Tiempo de detección y corrección de errores en la red:
   Cabe destacar que este tiempo no solo depende del controlador, sino también de la aplicación encargada del tipo de error que se pueda generar.

7. Rendimiento:
   Para cuantificar el rendimiento del controlador se utilizan dos parámetros llamados *latencia* y *throughput*. Estos parámetros son extremadamente importantes a la hora de decidir qué tipo de controlador utilizar:

   a) Throughput:
   Se define como el número de mensajes por unidad de tiempo que el controlador puede enviar hacia el plano de datos con información de nuevas reglas de encaminamiento de flujo (mensaje *FLOW_MOD* del protocolo OpenFlow).

   b) Latencia:
   Se define como el tiempo de demora entre que se genera un *PACKET_IN* hacia el controlador, este lo procesa y retorna un *FLOW_MOD*.

8. Southbound APIs:
   La capacidad que tiene el controlador de comunicarse con el plano de datos es crítico en el diseño de una arquitectura. A pesar de que el protocolo de comunicación entre controlador y plano de datos más popular es OpenFlow, existen otros protocolos que agregan funcionalidades contra el plano de datos. Entre ellos encontramos:

   a) NETCONF (estandarizado por IETF)

   b) OF-Conf (ONF le brinda soporte)

   c) Oplex (Cisco)

   d) OVSDB

   e) COPS

9. Northbound APIs:
   Estas APIs permiten la comunicación con aplicaciones de automatización como por ejemplo OpenStack u OpenCloud utilizadas para administración de la nube.

10. Soporte:
    Se debe considerar las empresas y corporaciones que brindan soporte a los controladores, su vínculo con el mercado y el dinero invertido. Esto brinda una noción del tiempo que van a perdurar en el mercado.

A.2. Comparación de alto nivel

En base a los parámetros mencionados en la sección A.1, en esta sección se realiza una comparación de los controladores más populares. En la tabla A.1, se muestra el resumen comparativo con sus características más relevantes.

Tabla A.1: Comparativa de controladores más populares del mercado. Tabla actualizada a partir de [24]

Controlador	Lenguaje de programación	GUI	Nivel de documentación	Modularidad	Controladores Distribuidos o Centralizados	Sistemas operativos soportados	Productividad	SouthBound APIs	Northbound APIs	Alianzas	Soporte a Multithreading	Soporte a Openstack	Link a documentación
ONOS	Java	Basado en Web	Buena	Alta	Distribuido	Linux, MAC OS y Windows	Justa	Openflow: 1.5, 1.4, 1.3, 1.0 NETCONF / YANG, OVSDB, BGP, SNMP, REST, TL1	Rest API	Adtran, AT&T, Ciena, China Unicom, ComCast, DellEMC, edge-core, Google, intel, juniper, ntt, radisys, telekom, Samsung ,turktelekom	Si	No	https://wiki.onosproject.org/
OpenDayLight	Java	Basado en Web	Muy Buena	Alta	Distribuido	Linux, MAC OS y Windows	Justa	Openflow: 1.5 1.4, 1.3 ,1.0 NETCONF/YANG, OVSDB, PCEP, BGP-LS, LISP, SNMP	Rest API	Linux Foundation, Cisco, Erricon redHat y ZTe	Si	Si	https://wiki.opendaylight.org/view/Main_Page
NOX	C++	Es necesario usar una herramienta externa PyQt4	Pobre	Baja	Centralizado	Sobre todo Linux	Justa	Openflow 1.0	Rest API	Ncira	NOX_MT	No	https://media.readthedocs.org/pdf/nox/stable/nox.pdf
POX	Python	Es necesario usar una herramienta externa PyQt4	Media	Baja	Centralizado	Linux, Mac OS, y Windows.	Alta	Openflow 1.0	Rest API	Ncira	No	No	https://openflow.stanford.edu/display/ONL/POX+Wiki
RYU	Python	Si	Media	Media	Centralizado	Sobre todo Linux	Alta	Openflow 1.0 a 1.5 OF-config Netconf	Rest API	Nippo Telegragph y telephone Corporation	Si	Si	https://osrg.github.io/ryu/resources.html
Beacon	Java	Basado en Web	Justa	Media	Centralizado	Linux, Mac OS, y Windows.	Alta	Openflow 1.0	Rest API	Standfor University	Si	No	https://openflow.stanford.edu/display/Beacon/Home
Maestro	Java	----------	Pobre	Media	Centralizado	Linux, Mac OS, y Windows.	Pobre	Openflow 1.0	Rest API	RICE, NSF	Si	No	https://code.google.com/archive/p/maestro-platform/wikis/MaestroTutorial.wiki
FloodLight	Java	Web/ Java	Buena	Media	Centralizado	Linux, Mac OS, y Windows.	Pobre	Openflow 1.0 a 1.4	Rest API	Big switch Network	Si	SI	http://www.projectfloodlight.org/documentation/
Iris	Java	Basado en Web	Media	Media	Centralizado	Linux, Mac OS, y Windows.	Pobre	Openflow 1.0 a 1.3	Rest API	ETRI	Si	No	http://openiris.etri.re.kr/
MUL	C	Basado en Web	Media	Media	Centralizado	Sobre todo Linux	Pobre	Openflow 1.4,1.3,1.0 OVSDB, of-config	Rest API	Kulcloud	Si	Si	http://www.openmul.org/devdoc-center.html
Runos	C++	Basado en Web	Media	Media	Distribuido	Linux	Pobre	Openflow 1.3	REST API	ARCCN	Si	No	https://github.com/ARCCN/runos

Se puede observar que los más destacados son ONOS y OpenDaylight. Estos se encuentran desarrollados en Java, lo que les brinda gran capacidad de multitarea y posibilita la existencia de una muy buena GUI. Ambos presentan una gran modularidad a través del uso de contenedores OSGI que permiten, entre otras cosas, la instalación de aplicaciones agregadas en tiempo de ejecución y sin necesidad de reinicio. Sumado a esto, son de licencia libre y tienen el respaldo de grandes empresas y grupos de investigación líderes en el mercado, lo cual asegura su continuidad en el tiempo y la continua optimización de funcionalidades.

Por otra parte, en cuanto a las APIs sur, los más destacados continúan siendo ONOS y OpenDayLight (ODL), los cuales no solo admiten OF como protocolo para la interacción con el plano de datos, si no que también admiten otros como OF-Config y OVSDB. Entre ambos, ONOS destaca aun más dado que admite la última versión del protocolo OF V1.5.

Se busca que el controlador tenga capacidad para operar tanto redes SDN como redes tradicionales, implicando la posibilidad de gestionar una red híbrida. Para esto, el controlador debe tener integrado APIs sur que permiten utilizar protocolos tradicionales como BGP, OSPF u otros. Esta propiedad es cumplida por los controladores ODL y ONOS.

Finalmente, debido a la gran modularidad y cantidad de aplicaciones ya existentes que presentan los controladores ONOS y ODL, son estos los que resultan más utilizados en el mercado y por ello, este proyecto continuará con el análisis en profundidad de ambos, de forma de elegir uno de ellos.

A.3. Desempeño de controladores

Con el objetivo de analizar la performance de diferentes controladores, **se utilizaron los resultados obtenidos en [26, 27]**. En estos documentos se comparan los parámetros escalabilidad, latencia y throughput como se definen en la sección A.1.

Todas las pruebas presentadas en esta sección y en la sección A.4 son realizadas con la herramienta *Cbench*. *Cbench* o *Controller Benchmarker* es uno de los programas mas utilizados para evaluar el desempeño de controladores según los parámetros antes mencionados. [36]

La mayor desventaja del programa es que no se realiza un análisis granular de la performance de mensajes recibidos por cada switch, sino que se evalúa el promedio del conjunto de switchs. Esto se debe a la existencia de una única conexión TCP con el controlador para todos los switchs que están siendo simulados, hecho que claramente se aparta de la realidad.

El programa cuenta con dos modos de operación. El primero simula un estado de baja carga en el sistema denominado modo *Latencia* y el segundo simula un estado de carga alta llamado modo *throughput*. Para profundizar mas en el programa *CBench* dirigirse al github del mismo. [37]

En la imagen A.1 se expone el resultado del test en el modo *Latencia* realizado con *CBench*. Los detalles de la prueba y el escenario se puede encontrar en. [26]

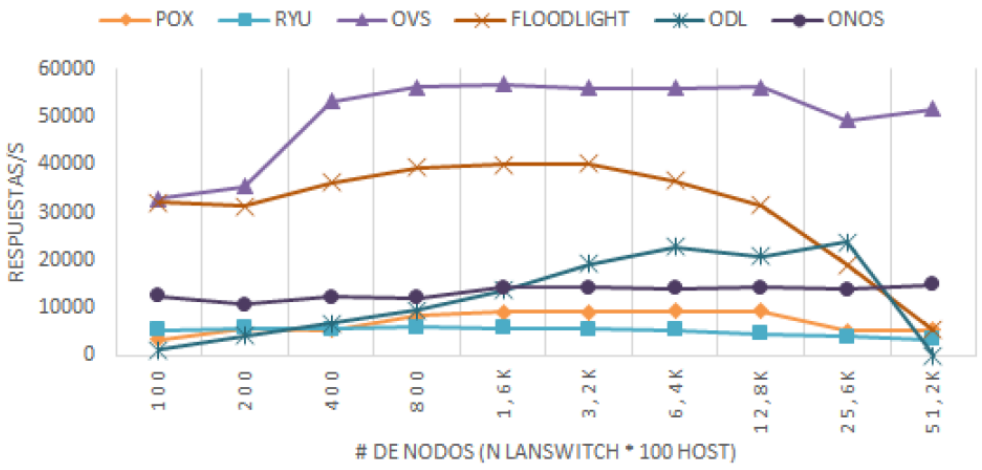


Figura A.1: Resultados en modo *latencia* de los controladores más populares. Imagen obtenida en [26]

En la imagen A.1 se pueden observar los distintos comportamientos que presentan los controladores al aumentar la cantidad de nodos conectados. En este caso, un nodo refiere a un *Switch OF* con 100 hosts conectados.

A modo de ejemplo, según estos resultados ONOS resulta estable con el aumento de nodos con los que se comunica, es decir, no presenta variaciones en la latencia al variar el número de hosts. En contraposición, en el caso del controlador ODL se observa un comportamiento variante. En este, la latencia disminuye con el aumento de hosts hasta aproximadamente 25600 hosts. A partir de ese punto, la latencia aumenta a medida que se incrementa la cantidad de nodos.

Si se clasifica los controladores por su máximo desempeño basado en la imagen anterior, se llega a la conclusión que los mejores controladores son:

1. OVS
2. Floodlight
3. ODL
4. ONOS

Se procede a analizar el segundo parámetro con relevancia. En la imagen A.2 se presentan los resultados obtenidos del test de *throughput*. [26]

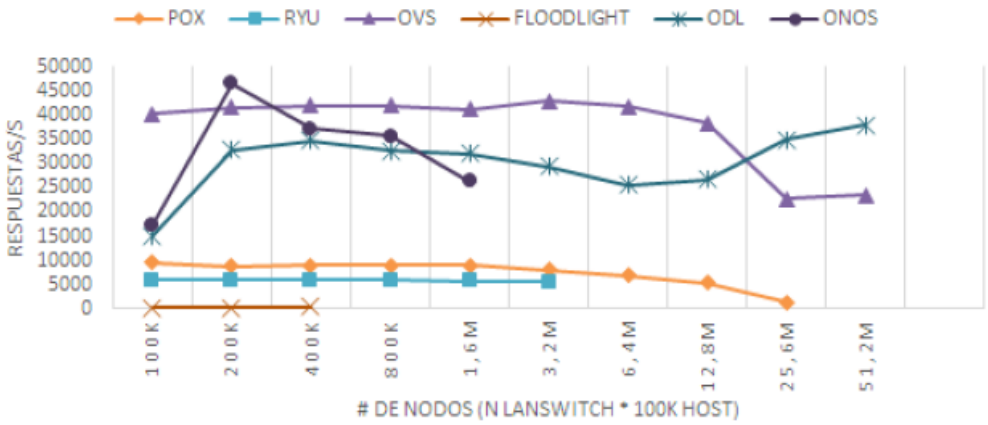


Figura A.2: Modo *throughput* de los controladores más populares. Imagen obtenida de [26]

La imagen A.2 refleja la escalabilidad en cuanto al *throughput* de los diferentes controladores. En esta imagen, el último punto de cada controlador representa la cantidad máxima de hosts que este logró administrar con éxito. Los tres controladores con mejor desempeño en la cantidad de respuestas en este caso son

1. OVS
2. ONOS
3. ODL

De los tests anteriores, se puede apreciar que los controladores más destacados son OVS, ODL y ONOS. El gran desempeño demostrado por OVS se debe a que se encuentra desarrollado en lenguaje C. Sin embargo, no se continuará con el análisis en profundidad de OVS dado que es un controlador de testing brindado por OpenVSwitch, con menores funcionalidades que ONOS y ODL. Además, posee una menor modularidad en su estructura, lo que dificulta el desarrollo de nuevas aplicaciones.

En cuanto a la escalabilidad de los controladores se podría deducir que ODL resulta más escalable que ONOS, dado que para una cantidad de aproximadamente 1.6 millones de host, en este ambiente de pruebas, ONOS no logra gestionar de forma correcta los mensajes de todos los hosts.

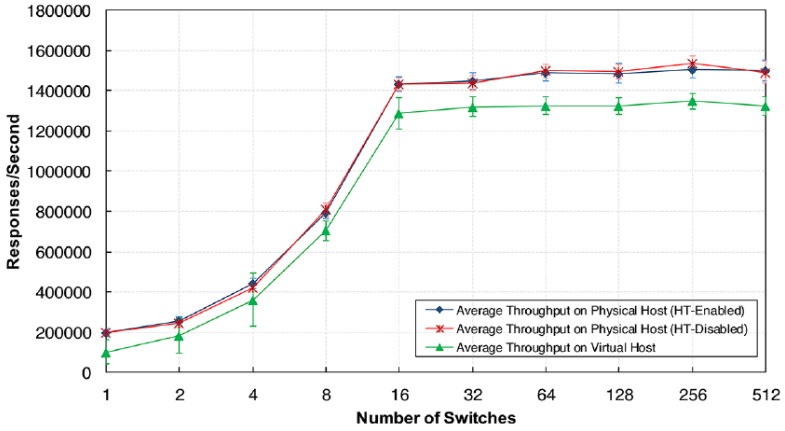
Como ya se ha mencionado al comienzo de este capítulo, el objetivo es seleccionar el controlador en el cual se realizará el desarrollo de las aplicaciones. Por ello, en la sección siguiente se realizará un análisis en profundidad de los controladores más destacados de esta sección: ODL y ONOS.

A.4. Profundización en ONOS y OpenDayLight

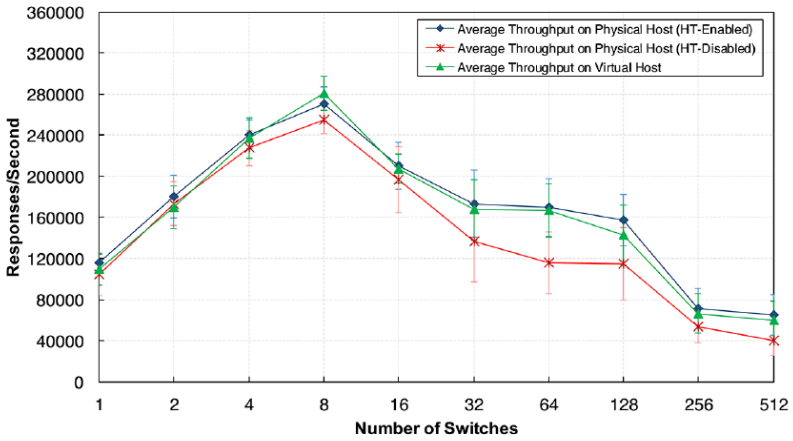
En esta sección se describen los resultados obtenidos en [27], donde se compara en detalle los parámetros relevantes del análisis de *ODL* y *ONOS*.

A.4.1. Análisis de Throughput

Esta prueba evalúa el impacto que tiene modificar el número de switchs controlados sobre la cantidad de respuestas por unidad de tiempo. A su vez, parámetros como hosts por switch, duración de la prueba, número de *loops* en la red a detectar por el controlador, número de *threads* en el CPU y otros permanecen constantes.



A.3.a- Resultados para ONOS



A.3.b- Resultados para ODL

Figura A.3: Resultados de *throughput* obtenidos en [27]

Como se puede observar de la imagen A.3.a, el *throughput* del controlador ONOS presenta dos comportamientos cualitativamente distintos:

Primero, un crecimiento a medida que aumenta la cantidad de switchs conectados hasta que se alcanzan los 16 switchs. Posteriormente, un valor constante de aproximadamente 1.4 Millones de respuestas/s.

El caso donde el servidor es virtualizado presenta una forma similar a los otros escenarios. Sin embargo se aprecia que su performance decae considerablemente, en particular cuando el controlador entra en la etapa de "meseta" luego de los 16 switchs, con una diferencia apreciable de aproximadamente 13.5 %. Al mismo tiempo, se observa que la varianza al virtualizar el host es considerablemente mayor respecto a correr el controlador sobre un servidor físico.

A pesar de las últimas observaciones, el comportamiento cualitativo comparando servidor físico y virtualizado resulta muy simular, pudiendo ser un indicador de estabilidad de la estructura del controlador.

Por otra parte, observando la imagen A.3.b de ODL, se observa que el *throughput* inicialmente aumenta con el número de switchs hasta 8, punto en el que comienza a decrecer drásticamente. Este comportamiento aparenta ser normal, por lo menos hasta la versión *Berilyum-sr2* [27]. A su vez, se identifica un comportamiento llamativo: el rendimiento del servidor virtualizado parece ser superior al servidor con la opción *hyperthreading* inactiva.

Observando los resultados anteriores, resulta evidente que el controlador ONOS es ampliamente superior a ODL. Desde el punto de vista del *throughput* máximo, mientras ODL alcanza un valor pico de 260k respuestas/s para 8 switchs conectados, ONOS responde con un *throughput* de 800k respuestas/s para la misma cantidad de switchs.

Por otra parte, se concluye que ONOS resulta más estable que ODL ya que, mientras ONOS alcanza una meseta y se estabiliza en un valor de *throughput* máximo, ODL comienza a decaer a medida que aumentamos la cantidad de switchs. Estos resultados, llevan a la conclusión que según este parámetro, ONOS es el controlador preferible a elegir si lo que se desea diseñar es sensible a la performance del controlador.

A.4.2. Análisis de Latencia

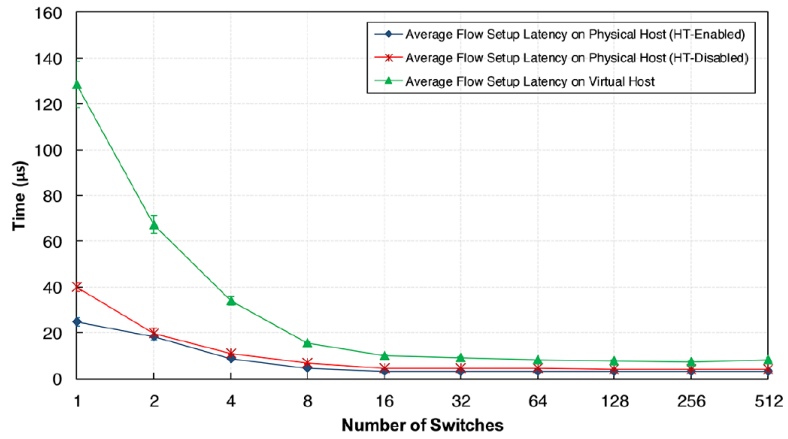
El siguiente parámetro a evaluar con el fin de realizar la selección es la *latencia* del controlador. Al igual que para el parámetro anterior, se realizan pruebas variando la cantidad de switchs conectados al controlador, dejando los demás parámetros fijos. [27]

La característica más importante a destacar de las imágenes A.4 es la considerable reducción en la latencia que ocurre con el aumento de los switchs presentes en la prueba. Utilizando como ejemplo la imagen A.4.a obtenida para el controlador ONOS, en donde la característica HT del servidor se encuentra apagada (curva roja), es posible apreciar que cuando un único switch se encuentra conectado, la *latencia* para el paquete que este envía es de $40\mu s$. Este tiempo representa el mínimo tiempo de procesamiento que el controlador posee para un paquete y por tanto, para todos los paquetes que deban ser procesados, este es el tiempo de demora del paquete individual resultará a ser al menos este.

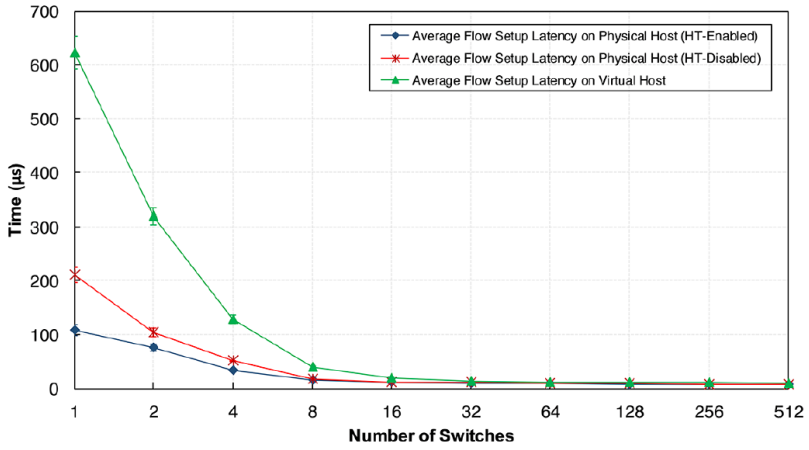
Surge entonces la duda, ¿Por qué la gráfica tiende a decrecer a medida que aumenta la cantidad de switchs conectados?

Para comprender el origen de este efecto, es importante recordar que ambos controladores poseen características de *Multithreading* y por tanto pueden procesar mensajes en paralelo. Con esto presente, cuando existen 2 switchs conectados, se generan 2 mensajes que el controlador puede procesar en paralelo. Entonces, si bien la *latencia* de cada uno de los mensajes es de $40\mu s$, estos se procesan ambos al mismo tiempo, por lo que en promedio la *latencia* del sistema disminuye a aproximadamente la mitad, $20\mu s$ como es apreciable en la imagen A.4.a.

En base a las imágenes A.4 que muestran el comportamiento de ambos controladores en relación a la *latencia*, se destaca que si bien el comportamiento cualitativo es similar, en donde las curvas para el controlador virtualizado presentan



A.4.a- Resultados para ONOS



A.4.b- Resultados para ODL

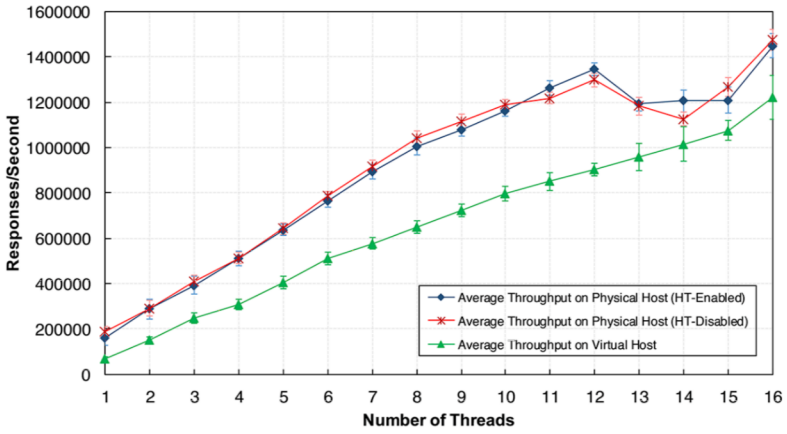
Figura A.4: Resultados de *latencia* obtenidos en [27]

un claro aumente frente a ejecutar el controlador sobre un servidor físico, ONOS resulta ampliamente superior a ODL (imagen A.4.b). Esta superioridad se aprecia particularmente para un número de switchs menor a 16. Para una cantidad mayor a 16 switchs la curva de ODL converge a un valor de $7\mu s$, un 75 % mayor a la *latencia* que presenta ONOS, con una tendencia a $4\mu s$.

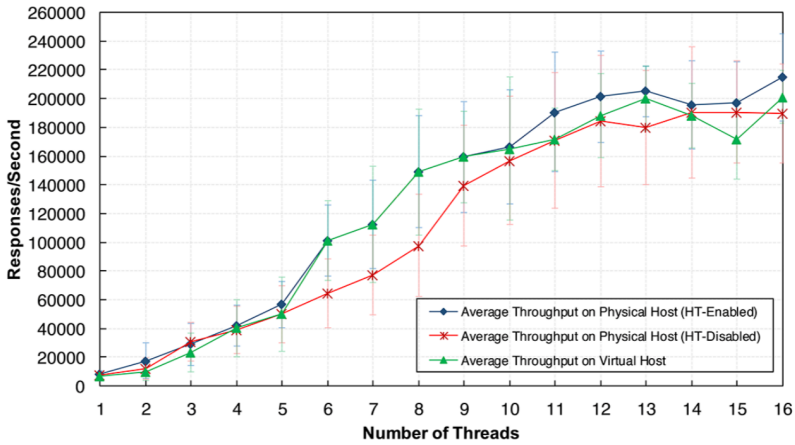
A.4.3. Análisis de capacidad de Multithreading

En este documento, el último parámetro que será tomado en cuenta para la comparación de los controladores será la escalabilidad respecto a la característica de *Multithreading*, es decir, manteniendo una cantidad fija de switchs conectados al controlador, se aumenta la cantidad de *threads* disponibles para procesamiento

paralelo y se observa el impacto que tiene sobre los parámetros antes analizados.



A.5.a- Resultados para ONOS



A.5.b- Resultados para ODL

Figura A.5: Resultados de *throughput* con *multithreading* obtenidos en [27]

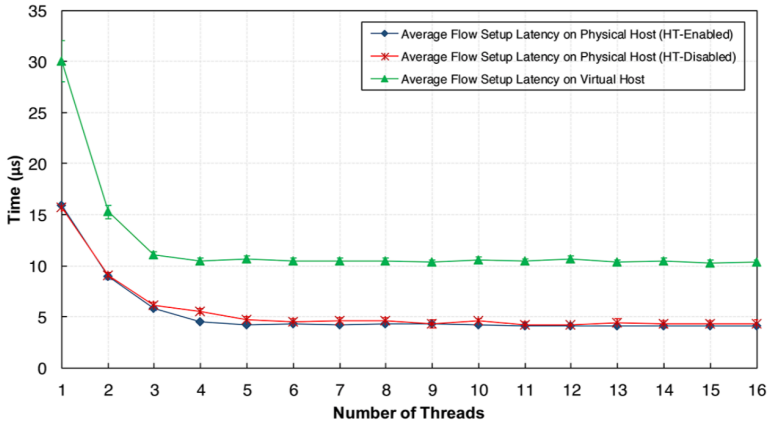
En las imágenes A.5 se presenta como se ve afectado el parámetro de *throughput* con el aumento de la cantidad de *threads* disponibles. Como se puede apreciar, a medida que aumentan los recursos de procesamiento paralelo, el *throughput* de ambos controladores tiende a aumentar. Esto remarca el hecho que ambos controladores se encuentran diseñados con el fin de soportar altos niveles de procesamiento paralelo.

Existe un efecto apreciable sobre el controlador ONOS cuando la cantidad de *threads* se encuentra entre 12 y 15:

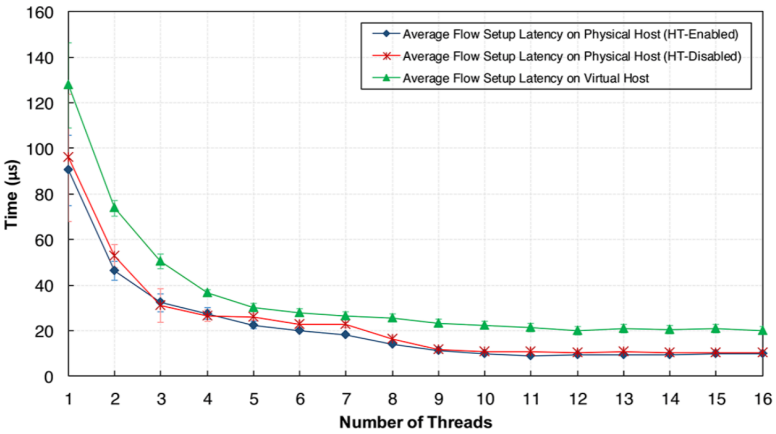
Aparece una inesperada caída del *throughput*. Es importante comprender que este efecto no es producido por el controlador en sí mismo, sino que es producto de limitaciones en el hardware subyacente. Estas pruebas se realizaron con un servidor que posee 2 CPUs, cada uno con 12 núcleos físicos. Si se desea utilizar más de

12 *threads* simultáneos, en donde cada *thread* se encuentra asociado a un núcleo físico, cuando se superan los 12 threads es necesario la utilización de ambos CPUs. Esto implica la utilización de protocolos de comunicación entre ambos CPUs que agregan una sobrecarga de tiempo que disminuye la performance del sistema.Este efecto se deja de observar para 16 threads ya que, a partir de ese momento, las ventajas de procesamiento paralelo introducida por la utilización de 4 núcleos del segundo CPU, supera las sobrecargas de tiempo introducidos el hecho anterior. [27]

Tomemos una de las zonas de la gráfica A.5.a con pendiente relativamente continua, como por ejemplo entre 1 y 8 threads. En esta zona, el aumento del *throughput* que posee ONOS es de aproximadamente 100000 respuestas/s por *thread* agregado. En comparación, el máximo aumento que presenta ODL en la gráfica A.5.b se da entre los *threads* 5 y 6 y es de 40000 respuestas/s por cada *thread* agregado.



A.6.a- Resultados para ONOS



A.6.b- Resultados para ODL

Figura A.6: Resultados de *latencia* con *multithreading* obtenidos en [27]

Posteriormente, se evalúa el efecto de la cantidad de *threads* sobre la *latencia* de los controladores como se presenta en las imágenes A.6. Recordando que *CBench* mide *latencia* en términos del promedio de todos los switches simulados, es esperado que a medida que se aumentan los recursos destinados a los controladores, el valor del parámetro tienda a disminuir.

En las imágenes se destaca el valor de *latencia* asociado a un único *thread* ya que representa la mínima latencia que el controlador genera para cada mensaje. Para el controlador ONOS, esta característica resulta ampliamente superior, donde incluso la latencia con ONOS virtualizado es menor a la mitad de la latencia del controlador ODL no virtualizado.

A.5. Resumen ONOS vs ODL

En términos de *throughput* y *latencia*, independientemente del escenario elegido (cantidad de switchs conectados, cantidad de threads disponibles, etc) ONOS demostró ser superior a ODL ya que presenta una mayor performance general, permitiendo mayor escalabilidad. [24–27]

Un hecho no menor que se aprecia en cada uno de los parámetros analizados es la diferencia en la estabilidad entre los controladores: las varianzas de los resultados de ODL presentadas en las gráficas de la sección A.4 resultaron ampliamente mayores a las presentadas para ONOS, por lo que se deduce que la arquitectura de ONOS es más estable que la de ODL. [27]

Como se observa en la tabla A.1, ambos controladores se encuentran desarrollados en el mismo lenguaje de programación (Java), presentan una gran modularidad y un gran conjunto de aplicaciones ya desarrolladas que permiten brindar servicios de valor agregado. Además, grandes corporaciones impulsan el desarrollo de los mismos, lo que induce a ser ampliamente utilizados en el mercado cuando se pretende desplegar servicios bajo el paradigma SDN. [24]

Si bien ambos controladores resultan interesantes para desarrollar una aplicación y brindar servicios en una red SDN, teniendo en cuenta el análisis de rendimiento, se concluye que el controlador más adecuado para brindar servicios de red sensible a requerimientos de tiempo y en el cuál se desarrollan las aplicaciones requeridas, es el controlador ONOS.

Esta página ha sido intencionalmente dejada en blanco.

Apéndice B

Ambiente de pruebas

El presente anexo da a conocer el contexto del ambiente de pruebas utilizado en el capitulo 8 de Testing. Este consisten en dos computadoras conectadas por un cable Gigabit Ethernet de categoría 5. En las pruebas el controlador ONOS se ejecutó en el equipo A, y la red fue simulada en Mininet por el equipo B. Cada uno de los equipos se pasan a exponer en la siguiente tabla.

Tabla B.1: Ambiente de Pruebas

	Equipo A	Equipo B
Sistema operativo	Ubuntu 18.04 64 bits	Ubuntu 18.04 64 bits
CPU:	Intel Core i5-8300H @2.3GHz	Intel Core i7-6500 @ 2.5 GHz
RAM	8 GbB DDR4	8 GbB DDR4
Tarjeta de red	1 Gbps	1 Gbps
Conexion	Ethernet categoría 5e	
Disco duro	Serial ATA 5500 rpm	THNSN5256GPUK NVMe PCIe M.2 256GB

Esta página ha sido intencionalmente dejada en blanco.

Apéndice C

Funciones de *REST API*

A continuación se detalla los parámetros de las funcionalidades implementadas en REST API. Las siguientes cuatro tablas hablan de la descripción, método, URL, entradas y salidas de cada una de las funcionalidades.

Tabla con la descripción de cada método implementado en la interfaz de REST API

Id	Función	Descripción
1	RegisterPop	Registra un PoP en la ON
2	GetPoP	Devuelve la información de un PoP en la ON
3	UnregisterPop	Desregistra un PoP en la ON
4	GetRegisteredPoPs	Devuelve los registros de los PoPs registrados
5	GetActivatedPoPs	Devuelve los registros de los PoPs que se han conectado
6	GetConnectedPoPs	Devuelve los registros de los PoPs que están conectados
7	PrintRegisterPoPs	Muestra en ONOS los registros de los PoPs registros
8	PrintActivatedPoPs	Muestra en ONOS los registros de los PoPs activos
9	PrintConnectedPoPs	Muestra en ONOS los registros de los PoPs Conectados
10	CreateONRP	Crea un ONRP
11	GetONRP	Devuelve la información de un ONRP
12	ModONRP	Modifica un ONRP
13	DeleteONRP	Borra un ONRP
14	GetAllONRP	Devuelve la información de todos los ONRPs
15	InitMeasure	Inicializa todos los parámetros para realizar la medida
16	GetMeasure	Obtiene la información de la inicialización de medida
17	TriggerMeasure	Manda un mensaje a un PPG para que se inicie la medida
18	StopMeasure	Des registra la medida
19	GetResult	Pide que se devuelva la ultima medida obtenida
20	GetAllMeasure	Devuelve los datos de todos los Measures
21	DeleteStorage	Borra los registros de PoPs en la memoria no volátil
22	saveOn	Activa el guardado automático de los PoPs
23	saveOff	Desactiva el guardado automático de los PoPs

La siguiente tabla detalla el método HTTPS de cada función con su url. Se omite que cada URL empieza con: http://< $IPdelcontrolador$ >: 8181/$onos/onra$.

Se usa la notación de <switchid> para indicar que se debe remplazar ese parámetro por el Id del switch (por ejemplo por of 0000000000000001). Todos los otros indicadores con "<>" son números enteros que hacen referencia a un valor que fue previamente entregado en la creación del objeto correspondiente. Por ejemplo GONRP es un numero entero que representa el Id del ONRP.

Id	Nombre	Método	URL
1	RegisterPop	POST	topology/register
2	GetPoP	GET	topology/get&id= <switchid>
3	UnregisterPop	POST	topology/unregister&id= <switchid>
4	GetRegisteredPoPs	GET	topology/get&registered
5	GetActivatedPoPs	GET	topology/get&activated
6	GetConnectedPoPs	GET	topology/get&connected
7	PrintRegisterPoPs	POST	topology/onosPrintRegisterPoPs
8	PrintActivatedPoPs	POST	topology/onosPrintActivatedPoPs
9	PrintConnectedPoPs	POST	topology/onosPrintConnectedPoPs
10	CreateONRP	POST	onrp/create
11	GetONRP	GET	onrp/get&id=<GONRP>
12	ModONRP	POST	onrp/modify
13	DeleteONRP	POST	onrp/delete&id=<GONRP>
14	GetAllONRP	POST	onrp/get&all
15	InitMeasure	POST	measure/init
16	GetMeasure	GET	measure/get=path&id=<Mid>
17	TriggerMeasure	POST	measure/trigger
18	StopMeasure	POST	measure/stop&id=<Mid>
19	GetResult	GET	measure/get=result&id=<Mid>
20	GetAllMeasure	GET	measure/get=all&path
21	DeleteStorage	POST	storage/delete
22	saveOff	POST	storage/save-on-exit=off
23	saveOn	POST	storage/save-on-exit=on

Tabla que marca los parámetros del JSON de entrada de cada función.

Id	Nombre	.JSON
1	RegisterPop	"switchId":"<switchid>", "switchMac":"<MacPoP>", "switchIp":"<IPPoP>", "ispRouterIp":"<ispRouterIP>", "subNet":"<SubNetPoP>", "ppgIp":"<IP_PPG>", "ppgMinPort":"<PPG_min_port>", "ppgMaxPort":"<PPG_MAX_port>"
2	GetPoP	
3	UnregisterPop	
4	GetRegisteredPoPs	
5	GetActivatedPoPs	
6	GetConnectedPoPs	
7	PrintRegisterPoPs	
8	PrintActivatedPoPs	
9	PrintConnectedPoPs	
10	CreateONRP	"orgnet":"<SubNetSrc>", "dstnet":"<SubNetDst>", "path": "<path>", "protocol":"<Protocol>", "orgport":"<OrgPort>", "dstport":"<DstPort>", "nattimeout":"<NatTimeout>"
11	GetONRP	
12	ModONRP	"orgnet": "<SubNetSrc>", "dstnet":"<SubNetDst>", "path":"<path>" "protocol":"<Protocol>", "orgport":"<OrgPort>", "dstport":"<DstPort>", "nattimeout":"<NatTimeout>"
13	DeleteONRP	
14	GetAllONRP	
15	InitMeasure	"path": <Path>
16	GetMeasure	
17	TriggerMeasure	"id":<Mid> "method": "RTT_METHOD", "packetsize": <packetSieze> "throughput": <througput> "average": <average>
18	StopMeasure	
19	GetResult	
20	GetAllMeasure	
21	DeleteStorage	
22	saveOff	
23	saveOn	

Esta ultima tabla muestra las respuestas de cada una de las funcionalidades. Se usan los paréntesis rectos para indicar que se entregan una array del objeto correspondiente.

Id	Nombre	Message
1	RegisterPop	PoP registered:<switchid>
2	GetPoP	PoP found "switchId": <Switchid>", "switchMac":"<MacPoP>", "switchIp":"<IPPoP>", "ispRouterIp":"<ispRouterIP>", "subNet":"<SubNetPoP>", "ppgIp":"<IP_PPG>", "ppgMinPort":"<PPG_min_port>", "ppgMaxPort":"<PPG_MAX_port>"
3	UnregisterPop	PoP Unregistered: "<switchid>"
4	GetRegisteredPoPs	Registered PoPs: [PoPs]
5	GetActivatedPoPs	Activated PoPs: [PoPs]
6	GetConnectedPoPs	Connected PoPs: [PoPs]
7	PrintRegisterPoPs	PoPs Printed
8	PrintActivatedPoPs	PoPs Printed
9	PrintConnectedPoPs	PoPs Printed
10	CreateONRP	ONRP successfully created <GONRP>
11	GetONRP	ONRP: <GONRP> "orgnet": "<SubNetSrc>", "dstnet":"<SubNetDst>", "path": "<Path>", "protocol":"<Protocol>", "orgport":"<OrgPort>", "dstport":"<DstPort>", "nattimeout":"<NatTimeout>"
12	ModONRP	ONRP successfully modified: <GONRP>
13	DeleteONRP	ONRP deleted
14	GetAllONRP	ONRPS: [ONRPs]
15	InitMeasure	Measure Initialized
16	GetMeasure	Measuring path found: <Path>
17	TriggerMeasure	Measure Triggered: <Trigger>
18	StopMeasure	Measure Stopping: <Mid>
19	GetResult	Measure result found: <Result>
20	GetAllMeasure	Measuring path found [Measure]
21	DeleteStorage	Succesfull
22	saveOff	Succesfull
23	saveOn	Succesfull

Apéndice D

Codigos de Error

En la implementación del sistema presentado en 3, se introdujo una codificación de error en las respuestas de REST API. Por ello este anexo describe los códigos de error implementados en una tabla.
Estos códigos se usan en todos los mensajes de respuesta en el encabezado ONRAcode, a modo de ejemplo en la respuesta ante una consulta sobre la información de un *PoP* desconocido, la respuesta con su código error es el siguiente:

$$u''message'' : u''unknownPoP'', u''result'' : u''switchIdn'', u''ONRAcode'' : 201$$

En próxima tabla se observan todos los códigos de error actuales.

Code	Message
200	Successful
201	PoP unknown
202	ONRP unknown
203	MeasureId unknown
204	Already Measuring
205	Measure stopping
400	The SwitchId is in wrong format.
401	The SwitchId must start with “of:”
402	The SwitchId must contain 16 hexadecimal digits
403	The SwitchId in wrong characters after “of:”
404	The MacAddres is in wrong format
405	The SwitchIP is in wrong format
406	The ispRouterAddr is in wrong format
407	The IpPrefix is in wrong format
408	The ppgAddr is in wrong format
409	The ppgMinPort is in wrong format
410	The ppgMaxPort is in wrong format
411	The ppgMinPort is in wrong format
412	The ppgMinPort is bigger than the ppgMaxPort

Code	Message
413	The ppgMinPort is bigger than $2^{16} - 1$
414	The structure of PoP is in wrong format
415	The GlobalONRPId can only be positive decimal numbers
416	The GlobalONRPId must be less than 2^{48}
417	One SwitchId of path is in wrong format
418	A PoP in the Path is unknown
419	The path of ModifyONRP must have more than one PoP
420	The path of ModifyONRP has a duplicate PoP
421	The origin and destination PoPs of the new path and old path must be the same
422	The newpath must be different than original Path
423	The path of ModifyONRP is in wrong format
424	The Timeout can only be positive decimal numbers
425	The Timeout must be less than 2^{31}
426	The source Subnet must be IPv4 with mask
427	The destination Subnet must be IPv4 with mask
428	The path of ONRP has a duplicate PoP
429	The path of ONRP must have more than one PoP
430	The path of ONRP is in wrong format
431	The protocol must be TCP(6), UDP(17) or WildCard(-1):
432	The sourcePort can only be positive decimal numbers or WildCard(-1)
433	The sourcePort must be less than 2^{31}
434	The destinationPort can only be positive decimal numbers or WildCard(-1):
435	The destinationPort must be less than 2^{16}
436	The NatTimeout can only be positive decimal numbers or 0(for permanent)
437	The NatTimeout must be less than 2^{16}
438	The structure of ONRP is in wrong format
439	The source and destination Subnets are wrong
440	Maximum number of ONRPs was reached: {Max_ID}
441	The MeasureId can only be positive decimal numbers
442	The MeasureId must be less than 2^{47}
443	Only RTT Method is implemented
444	The Packetsize can only be positive decimal numbers
445	The Packetsize must be less than 1464
446	The Throughput can only be positive decimal numbers
447	The Throughput must be less than 2^{31}
448	The Average can only be positive decimal numbers
449	The Average must be less than 2^{31}
450	The path of InitMeasure has a duplicate PoP
451	The path of InitMeasure must have more than one PoP
452	The path of InitMeasure is in wrong format
453	The structure of InitMeasure is in wrong format

Code	Message
500	Unknown error
501	Error creating ONRP
502	Error modifying ONRP
503	Some of the links are non-existent:
504	Measure has not started

Esta página ha sido intencionalmente dejada en blanco.

Apéndice E

Manual de Instalación

Este anexo es una guía para la instalación e iniciación del sistema ONRApp. Es de relevancia que esta guía ha sido corroborada en 3 equipos diferentes con Ubuntu 18.04.Cada vez que se haga referencia al repositorio de git se estara refiriendo al que se ubica en:

```
https://gitlab.fing.edu.uy/tesis/entrega.git
```

Instalación de ONOS

El inicio de la puesta en marcha radica en iniciar ONOS. A continuación se describen los pasos necesarios para su instalación. [38] Antes de continuar es necesario poseer siguientes paquetes: git,zip,curl,unzip,python2.7.
Lo primero es instalar JDK8, para ello usar los siguientes comandos en la terminal:

```
sudo apt-get install software-properties-common -y && \
sudo add-apt-repository ppa:webupd8team/java -y && \
sudo apt-get update && \
echo "oracle-java8-installer
shared/accepted-oracle-license-v1-1 select true"
| sudo  debconf-set-selections && \
sudo apt-get install
oracle-java8-installer oracle-java8-set-default -y
```

Finalizada la instalación de JDK8, es necesario instalar Bazel para poder realizar la compilación de ONOS. Se ha utilizado la versión 0.19.2 de Bazel, sin embargo no todas las versiones son compatibles con ONOS. [39] La última herramienta necesaria es Maven, la cual permite compilar la aplicación. Se instala con el siguiente comando:

```
sudo apt install maven
```

En esta etapa ya se tienen todas las herramientas para instalar y ejecutar ONOS. Se procede a descargarlo con los siguientes comandos:

```
git clone https://github.com/opennetworkinglab/onos
cd onos
sudo chmod 777 ~/onos/tools/dev/bash\_profile
sudo chmod a+x ~/onos/tools/dev/bash\_profile
echo '. ~/onos/tools/dev/bash\_profile' >> ~/.bashrc
source ~/.bashrc
```

Una vez instalado ONOS será necesario agregarlo al ambiente de desarrollo:

```
cd onos
cat << EOF >> ~/.bash\_profile
export ONOS\_ROOT="`pwd`"
source ONOS\_ROOT/tools/dev/bash\_profile
EOF
. ~/.bash\_profile
```

Por último se compila ONOS:

```
 cd \$ONOS\_ROOT
 bazel build onos
```

Para ejecutar ONOS se puede usar el siguiente comando en el repositorio ONOS:

```
ok clean
```

Instalación de ONRApp

El siguiente paso a la compilación de ONOS es ejecutar la aplicación realizada en esta tesis. Para esto es necesario descargar la ultima versión de ONRApp del Branch Master.

```
git clone https://gitlab.fing.edu.uy/tesis/onra
```

Una vez descargado ONRApp será necesario compilarlo con "Maven Clean Install". Con la compilación del programa y ONOS ejecutándose, se pasa a instalar y activar ONRApp en ONOS con [40]:

```
onos-app onos@localhost install! target/onra-1.0-SNAPSHOT.oar
```

Instalacion de mininet

Para instalar Mininet primero se necesita obtener el código fuente [41]:

```
git clone git://github.com/mininet/mininet
```

Lo siguiente es realizar el comando de instalación:

```
mininet/util/install.sh -a
```

Este comando instala Mininet y todas los paquetes que contiene. Para la comprobación de la correcta instalación de Mininet se puede utilizar el siguiente comando:

```
sudo mn --test pingalls
```

Inicialización del ambiente de pruebas

Existen dos posibilidades para ejecutar ONRApp con Mininet:

- La primera posibilidad es usar las funcionalidades por defecto de Mininet. En este caso los switchs se conectan directamente al controlador, y este último debe estar afuera de la red. Para esta opción alcanza con repetir los pasos antes indicados.
- La segunda posibilidad es correr ONOS en un host de Mininet. Esta opción es más compleja debido a que los switch de *OpenFlow* están diseñados para conectarse con un controlador afuera de Mininet. Por esta razón se optó por usar script de *OVS* para simular los switch en Mininet.

Para la segunda opción, se creó un script para automatizar el proceso, el cual contiene la topología usada en los pruebas del capitulo 8.1. Para ejecutar este programa es necesario descargar la carpeta Topología/outTest del repositorio de git. Una vez descargado se debe ejecutar outbandTest.py e iniciar el controlador en el host he.

Desactivación de Apps que producen conflicto con ONRApp

Esta guía es para desactivar tres aplicaciones por defecto que crean conflicto con ONRA. Ellas son fwd,proxyarp y mobility. Las dos primeras, al encaminar mensajes que también encamina ONRApp, produciendo duplicados de los mismos. Mientras que la última, borra cualquier *flow entry* que use la MAC de un host que se haya cambiado de posición en la Topología recientemente. Esto es muy problemático porque si se repite la dirección MAC asignada switch esta app puede borrar las *flow entries* usadas por ONRApp. Los pasos para solucionar estos problemas son:

1. Probar que ONOS se ejecute correctamente y luego detenerlo.
2. Guardar un respaldo del archivo “onos-service” de la carpeta /onos/tools/package/bin
3. Obtener el archivo “onos-service” proporcionado por el equipo de ONRA en onra-archivos-auxiliares en la carpeta deactivateAPPs del repositorio de gitlab.
4. Remplazar el archivo “onos-service” en la carpeta /onos/tools/package/bin
5. Ejecutar ONOS y verificar con el CLI que las apps fwd, proxyarp y mobility están desactivas.

Ejemplos basicos

En la carpeta ejemplos en el repositorio de git de la tesis se encuentra al menos un ejemplo de uso de cada funcionalidad de REST API habladas en C.

Se aconseja empezar con el script setupEnviroment.py que registra todos los PoPs, de la topología BasicTopo.py. El siguiente paso recomendado es usar el programa createPathway.py para crear un ONRP que pase por los PoPs 1,2 y 3 y verificar su funcionamiento.

Pruebas de funcionalidad

En la carpeta "pruebas" en el repositorio de git antes mencionado, se encuentran todas las pruebas de funcionalidad(muchas de ellas automáticas) realizadas con su respectivos comentarios. Se recomienda descargar estas pruebas y repetirlas para detectar errores de instalación o configuración.

Referencias

[1] Isabel Amigo, Gabriel Gómez, and Pablo Belzarena, "Ruteo y metrología en redes sobrepuestas utilizando el paradigma de redes definidas por software," 2017.

[2] Isabel Amigo, Gabriel Gómez, Marwa Chami, and Pablo Belzarena, "An sdn-based approach for qos and reliability in overlay networks," in *TMA Conference 2018 : Network Traffic Measurement and Analysis Conference, Vienna, Austria, 26-29 jun*, 2018, pp. 1–2. [Online]. Available: https://iie.fing.edu.uy/publicaciones/2018/AGCB18

[3] Wikipedia. Software defined networking. (Fecha de acceso 21-07-2019). [Online]. Available: https://en.wikipedia.org/wiki/Software-defined_networking

[4] Paul Goransson and Chuck Black, *Software Defined Networks: A Comprehensive Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[5] Datacomm. Software defined network. (Fecha de acceso 21-07-2019). [Online]. Available: https://www.datacomm.co.id/en/telco/sdn/

[6] The Open Networking Foundation, "Sdn architecture issue 1.1," 2016.

[7] The Open Networking Foundation, "Sdn architecture issue 1.0," 2014.

[8] Margaret Rouse, "service-level agreement (sla)," (Fecha de acceso 21-07-2019). [Online]. Available: https://searchitchannel.techtarget.com/definition/service-level-agreement

[9] The Open Networking Foundation, "Openflow switch specification," Jun. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf

[10] The Open Networking Foundation, "Openflow switch specification 1.5.1," 2015. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[11] Isabel Amigo and Gabriel Gómez, "Testbed for sdn-based overlay networks for qos-aware routing," 2016.

[12] THE URBAN PENGUIN, "rp_filter and lpic-3 linux security," (Fecha de acceso 21-07-2019). [Online]. Available: https://www.theurbanpenguin.com/rp_filter-and-lpic-3-linux-security/

[13] The Linux Document Project, "Reverse path filtering," (Fecha de acceso 21-07-2019). [Online]. Available: http://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.kernel.rpf.html

[14] VMware Docs, "Use ipv4 reverse path filtering." [Online]. Available: https://docs.vmware.com/en/vRealize-Automation/7.2/com.vmware.vra.planning.doc/GUID-7F464983-7A2D-46BB-8115-9E00AEB0B6E5.html

[15] A. Custura, A. Venne, and G. Fairhurst, "Exploring dscp modification pathologies in mobile edge networks," in *2017 Network Traffic Measurement and Analysis Conference (TMA)*, June 2017, pp. 1–6.

[16] IANA, "Protocol numbers," (Fecha de acceso 21-07-2019). [Online]. Available: http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

[17] Wikipedia. Cabezal ipv4. (Fecha de acceso 21-07-2019). [Online]. Available: https://en.wikipedia.org/wiki/IPv4#Header

[18] Félix Iglesias Vázquez and Tanja Zseby, "Time-activity footprints in ip traffic," *Computer Networks*, vol. 107, 03 2016.

[19] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian, "Internet inter-domain traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 75–86, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851194

[20] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho, "Seven years and one day: Sketching the evolution of internet traffic," in *IEEE INFOCOM 2009*, April 2009, pp. 711–719.

[21] IETF, "Multiprotocol label switching architecture," (Fecha de acceso 21-07-2019). [Online]. Available: https://tools.ietf.org/html/rfc3031

[22] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha, "Flowsense: Monitoring network utilization with zero measurement cost," vol. 7799, 03 2013, pp. 31–41.

[23] Curtis Yu, Cristian Lumezanu, Abhishek Sharma, Qiang Xu, Guofei Jiang, and Harsha V. Madhyastha, "Software-defined latency monitoring in data center networks," 03 2015, pp. 360–372.

[24] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "Sdn controllers: A comparative study," in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, April 2016, pp. 1–6.

[25] Alejandro García Centeno, Carlos Manuel Rodriguez Vergel, Caridad Anías Calderón, Frank Camilo, Casmartiño Bondarenko, and Uci , "Controladores sdn, elementos para su selección y evaluación," *Telemática*, vol. 13, pp. 10–20, 01 2014.

[26] Yanko Antonio Muro and Felix Alvarez Paliza, "Plataforma de pruebas para evaluar el desempeÑo de las redes definidas por software basadas en el protocolo openflow," Ph.D. dissertation, 12 2016.

[27] M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two openflow controllers," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017, pp. 1–6.

[28] ONF, "Onos: System components," (Fecha de acceso 21-07-2019). [Online]. Available: https://wiki.onosproject.org/display/ONOS/System+Components

[29] Ian Sommerville, *Ingeniería de software*, séptima edición. ed. Madrid :: Pearson Educación,, 2005.

[30] Mininet, "Introducción a mininet," (Fecha de acceso 21-07-2019). [Online]. Available: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[31] MIT, "Testing," (Fecha de acceso 21-07-2019). [Online]. Available: https://ocw.mit.edu/ans7870/6/6.005/s16/classes/03-testing/

[32] MIT, "Concurrency," (Fecha de acceso 21-07-2019). [Online]. Available: http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/

[33] PARASOFT, "Take control of threading issues impacting the performance of your java web api application," (Fecha de acceso 21-07-2019). [Online]. Available: https://blog.parasoft.com/take-control-of-threading-issues-impacting-performance-of-your-java-web-api-application

[34] Verizon, "Ip latency statistics," (Fecha de acceso 21-07-2019). [Online]. Available: https://enterprise.verizon.com/terms/latency/

[35] Pablo Belzarena, Gabriel Gómez, Isabel Amigo, and Sandrine Vaton, "Sdn-based overlay networks for qos-aware routing," in *LANCOMM 16 Proceedings of the 2016 workshop on Fostering Latin-American Research in Data Communication Networks, Florianopolis, SC, Brazil, aug 22-26*. ACM, 2016, pp. 19–21. [Online]. Available: https://iie.fing.edu.uy/publicaciones/2016/BGAV16

[36] Liehuang Zhu, Md Monjurul Karim, Kashif Sharif, Fan Li, Xiaojiang Du, and Mohsen Guizani, "Sdn controllers: Benchmarking and performance evaluation," 02 2019.

[37] Mininet, “Cbench,” (Fecha de acceso 21-07-2019). [Online]. Available: https://github.com/mininet/oflops/tree/master/cbench

[38] ONF, “Onos: Repositorio github,” (Fecha de acceso 21-07-2019). [Online]. Available: https://github.com/opennetworkinglab/onos

[39] Bazel. Bazel. (Fecha de acceso 21-07-2019). [Online]. Available: https://docs.bazel.build/versions/master/install-ubuntu.html

[40] ONF, “Onos: Application tutorial,” (Fecha de acceso 21-07-2019). [Online]. Available: https://wiki.onosproject.org/display/ONOS/Template+Application+Tutorial

[41] Mininet, “Mininet,” (Fecha de acceso 21-07-2019). [Online]. Available: http://mininet.org/download/

Índice de tablas

Esta página ha sido intencionalmente dejada en blanco.

Índice de figuras

Índice de figuras

Esta es la última página.
Compilado el lunes 22 julio, 2019.
http://iie.fing.edu.uy/