TRABAJO FIN DE GRADO

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Federación de controladores SDN con DDS

Autor
José Ángel Expósito Arenas

Directores
Juan Manuel López Soler
Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Septiembre de 2016

Federación de controladores SDN con DDS

José Ángel Expósito Arenas

Palabras clave: SDN, DDS, federación, recuperación

Resumen

El innegable aumento del tráfico en la red durante los últimos años ha llevado a desarrollar una arquitectura de red alternativa a las ya existentes. Las redes tradicionales no están preparadas para satisfacer los requisitos actuales ni futuros de los operadores de telecomunicaciones, ni de los usuarios finales. Debido a esto, algunas empresas de telecomunicaciones, y otras de la importancia de Google, están empezando a implementar la arquitectura de red conocida como *Sotfware-Defined Network*.

SDN crea una red más flexible, escalable, potente y programable que las arquitecturas de red actuales. Mediante el uso de APIs, permite que las aplicaciones instaladas en el controlador, el cual desempeña la función de cerebro de la red, puedan modificar de forma sencilla y dinámica todos los servicios proporcionados por la red. Con la separación del plano de control y del plano de datos de la red, esta se vuelve mucho más eficaz y rápida.

Aunque como se ha dicho, las SDN son muy escalables, el control de la red sigue siendo centralizado, y se materializa en el controlador, el cual se sirve del protocolo OpenFlow para gestionar todos los *switches* de la red. Esto supone un punto fuerte de vulnerabilidad en caso de fallo de dicho controlador, así como un periodo importante durante el cual la red está inoperativa.

Para dar solución, o al menos paliar este problema, en este proyecto se propone una solución basada en la federación de dos controladores SDN en una topología de red cualquiera. En el diseño propuesto es clave utilizar un sistema de publicación/suscripción que permita el envío de información en tiempo real.

El uso de este paradigma implica un desacoplamiento a nivel de aplicación entre ambos controladores, por lo que existe una mayor flexibilidad a la hora de programar dichas aplicaciones. En este sentido, este proyecto busca conseguir que ambos controladores compartan una única imagen, lo más exacta posible, de toda la red, a fin de reducir el tiempo de recuperación cuando se produce algún fallo y aumentar la escalabilidad de la misma.

Analizando la solución desarrollada para los problemas descritos, en este trabajo se propone un modelo de federación de controladores SDN mediante el uso de la tecnología DDS. Esta tecnología, basada en el esquema de publicación/subscripción, ha sido usada para intercambiar datos referentes a la topología, así como las reglas necesarias para el correcto enrutamiento de paquetes en la red, los llamados "flujos", entre ambos controladores. Esto

permite que los controladores compartan y mantengan una imagen común de la red.

Además de incluir el diseño, en este trabajo se incluye todo el proceso necesario para la implementación de ambos controladores junto al simulador de redes *Mininet*. Por otra parte, también se incluye todas las pruebas realizadas a fin de ver como la federación ha sido exitosa e incluso es más eficaz, en términos de tiempo de recuperación y tráfico generado, que utilizar un solo controlador en la red, o usar dos pero sin ningún tipo de federación.

SDN Controllers Federation with DDS

Expósito Arenas, José Ángel

Keywords: SDN, DDS, federation, recuperation

Abstract

The undeniable increase in network traffic in recent years has led to develop an alternative architecture to the existing network. Traditional networks are not prepared to meet current or future requirements of telecom operators, or end users. Because of this, some telecommunications companies, and others of the importance of Google, are beginning to implement network architecture known as Sotfware-Defined Network.

SDN creates a more flexible, scalable, programmable powerful that current network architecture. By using APIs, allowing applications installed in the network controller, which acts as the "brain.ºf the network, can easily and dynamically modify all services provided by the network. With the separation of the control plane and the data plane of the network, this becomes much more efficient and faster.

Although as stated above, the SDN are highly scalable, network control remains centralized, and is embedded in the controller, which uses the OpenFlow protocol to manage all network switches. This is a strong point of vulnerability in case of failure of the controller, as well as an important period during which the network is inoperative.

To solve or at least alleviate this problem, this project is based on a federation of two SDN controllers solution in a given network topology. In the proposed design it is key to use a system of publication/subscription that allows sending information in real time.

Using this paradigm involves a decoupling application level between the two controllers, so that there is greater flexibility in scheduling these applications. In this sense, this project aims to get both controllers to share a single image, as accurate as possible, to the entire network, to reduce the recovery time when a failure occurs and increase network scalability.

This paper describes a SDN controllers federation model using DDS technology. DDS technology, based on the scheme of publication / subscription, has been used to exchange data concerning the topology and rules necessary for the proper routing of packets on the network, called "flows", between the two controllers. This allows controllers to share and maintain a common picture of the network.

In addition to including the design, this paper includes all necessary for the implementation of both controllers together with Mininet simulator. Moreover, all tests to see how the federation has been successful and is even

more effective in terms of recovery time and generated traffic, that use a single controller on the network, or use two but without any federation is also included.

Yo, **José Ángel Expósito Arenas**, alumno de la titulación de la **Grado en Ingeniería de Tecnologías de Telecomunicación**, con DNI XXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José Ángel Expósito Arenas

Granada a 10 de Septiembre de 2016.

D. **Juan Manuel López Soler**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Jorge Navarro Ortiz**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Federación de controladores SDN con DDS***, ha sido realizado bajo su supervisión por **José Ángel Expósito Arenas**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 10 de Septiembre de 2016.

Los directores:

Juan Manuel López Soler **Jorge Navarro Ortiz**

Agradecimientos

En primer lugar, agradecer a mis tutores en este proyecto. A Juanma, por su mano experta a la hora de guiarme en la realización de este trabajo, además de sus constantes ánimos durante el mismo. A Jorge, por su ayuda clave cuando más difícil veía que esto saliera adelante.

A mis compañeros y amigos de toda la vida, por su apoyo incondicional a pesar de la distancia, que con su habitual sentido del humor han sabido ayudarme en algún que otro mal momento.

Como no, agradecer también a todas las magníficas personas que he tenido la gran suerte de conocer durante estos cuatro años en Granada. A Pedro, Victor, Fran, Carlos, con nuestras míticas tardes en la biblioteca sufriendo el lado oscuro de nuestros proyectos, Manolo, Jesús, Antonio, Juan David y Julio, por cada una de las experiencias y recuerdos que he tenido la oportunidad de vivir con vosotros, cada uno de vosotros me ha ensañado algo que no creo que olvide jamás, no se puede pedir mejores amigos. Pero gracias sobre todo por vuestros ánimos y apoyo en cada uno de los altibajos de este proyecto.

Al grupo de los *hipsters* de teleco, por todas las experiencias que hemos compartido este último año, por esas noches y esas cervecillas que suponían un auténtico desahogo cuando necesitaba olvidarme por un rato de los problemas del día a día.

Finalmente, a mis padres, Jose y Ana, porque sin su apoyo en cada uno de mis decisiones, seguramente estos cuatro años habrían sido mucho más complicados y duros de afrontar. Y a mi hermana Beatriz, por hacerme ver que a una parte de mí también le gusta enseñar además de aprender.

A todos, MIL GRACIAS.

Índice general

1. Introducción **1**
- 1.1. Contexto y motivación . . . 1
  - 1.1.1. La necesidad de una nueva arquitectura de red . . . 2
  - 1.1.2. Limitaciones de las redes actuales . . . 3
- 1.2. Introduciendo el *Sotware-Defined Networking* . . . 5
- 1.3. Introduciendo el *Data Distribution Service* . . . 6
- 1.4. Objetivos principales del proyecto . . . 7
- 1.5. Estructura de la memoria . . . 7

2. Conocimientos teóricos y herramientas utilizadas **9**
- 2.1. Software-Defined Networking . . . 9
  - 2.1.1. Funcionamiento . . . 10
  - 2.1.2. Arquitectura . . . 12
- 2.2. Data Distribution Service . . . 13
- 2.3. Herramientas utilizadas . . . 14
  - 2.3.1. RTI Connext DDS . . . 14
  - 2.3.2. OpenFlow . . . 17
  - 2.3.3. MiniNet . . . 20
  - 2.3.4. Opendaylight . . . 21
  - 2.3.5. Otras herramientas . . . 22

3. Estado del Arte **25**
- 3.1. HyperFlow . . . 25
  - 3.1.1. Diseño . . . 26
  - 3.1.2. Funcionamiento . . . 27
- 3.2. SDNi . . . 28
  - 3.2.1. Implementaciones . . . 29
  - 3.2.2. Arquitectura . . . 30
  - 3.2.3. Ejemplos de uso . . . 31
- 3.3. CPRecovery . . . 32
  - 3.3.1. Procesos de replicación y recuperación . . . 32
  - 3.3.2. Comparación entre implementaciones . . . 34

4. Planificación y Costes **35**

5. Diseño **37**
5.1. Diseño de la arquitectura 37
5.1.1. Publisher 37
5.1.2. Subscriber 39
5.1.3. Topología SDN 41
5.2. Módulos del controlador *Publisher* 42
5.2.1. Módulo creador de flujos 42
5.2.2. Módulo de comunicación 45
5.3. Módulos del controlador *Subscriber* 48
5.3.1. Módulos creador de flujos 49
5.3.2. Módulo de comunicación 49
5.3.3. Módulo de datos 51

6. Implementación **53**
6.1. Entorno de desarrollo estable 53
6.2. Diagrama de clases 57
6.3. Ejecución de la solución propuesta 59

7. Evaluación **63**
7.1. Evaluación cualitativa 63
7.2. Evaluación cuantitativa 68

8. Conclusiones y Trabajo Futuro **73**
8.1. Conclusiones 73
8.2. Trabajo Futuro 75

Bibliografía **80**

Índice de figuras

1.1. Aumento del tráfico IP 2013-2018. [2] 4
1.2. Arquitectura de red SDN. [15] 6

2.1. Elementos red SDN. [4] 11
2.2. Capas de red SDN. [3] 12
2.3. Esquema RTI Connext DDS. [17] 16
2.4. *Switches* tradicionales frente a sistemas abiertos. [10] 18
2.5. Arquitectura de un *switch* OpenFlow. [13] 19
2.6. Arquitectura de un switch OpenFlow. 22

3.1. Arquitectura de HyperFlow. [18] 26
3.2. Tipos de mensaje en HyperFlow. 27
3.3. Implementación vertical SDNi. [4] 29
3.4. Implementación horizontal SDNi. [4] 30
3.5. SDNi en Opendaylight. [12] 30
3.6. Funcionamiento CPRecovery. [16] 33

4.1. Planificación del proyecto. 35

5.1. Esquema elementos Publisher. 39
5.2. Esquema elementos Subscriber. 40
5.3. Esquema de la red hecha con Mininet. 42
5.4. Diagrama de flujo del módulo creador de flujos. 45
5.5. Plantilla de las muestras DDS. 46
5.6. Diagrama de flujo del módulo de comunicación. 48
5.7. Diagrama de flujo del módulo de comunicación. 50

6.1. Generar proyecto con Maven. 54
6.2. Instalar librería DDS. 55
6.3. Dependencia DDS. 55
6.4. Librería DDS de forma automática. 56
6.5. Estructura de ambos proyectos. 58

7.1. Base de datos del controlador *Publisher* 64
7.2. Base de datos del controlador *Subscriber* 65

7.3. *Ping* entre dos nodos de la red. 65
7.4. Resumen de las muestras DDS recibidas. 66
7.5. *Operational Data-Store después de recibir las muestras DDS.* 67
7.6. Ping despues de conectar con el controlador *Subscriber*. . . . 68
7.7. Gráfica del tiempo de recuperación. 69
7.8. Gráfica del tráfico generado durante la recuperación. 70

Índice de tablas

Lista de acrónimos

SDN Software-Defined Network

DDS Data Distribution Service

SDNi Software-Defined Network Interface

API Application Programming Interface

CDN Content Delivery Network

QoS Quality of Service

DoS Denegation of Service

AD-SAL API-Driven System Abstraction Layer

MD-SAL Model-Driven System Abstraction Layer

BGP Border Gateway Protocol

DCPS Data-Centric Publish-Subscribe

IDL Interface Language Description

NAT Network Address Translation

OSGi Open Services Gateway initiative

ONF Open Networking Foundation

SAL Service Abstraction Layer

LLDP Link Layer Discovery Protocol

ACL Access List

ODL-BGP Opendaylight-Boarder Gateway Protocol

Capítulo 1

Introducción

En este primer capítulo se introducen los aspectos fundamentales del proyecto. Para empezar, se analiza en el contexto en el que se desarrolla este trabajo, así como la motivación que ha permitido su realización. Después se describe brevemente uno de los pilares del proyecto, las Software-Defined Network (SDN). Por último, se describen los objetivos a alcanzar inicialmente planteados en el anteproyecto, junto con la estructura de todo el proyecto.

1.1. Contexto y motivación

En esta sección se dan las razones de la elección de este proyecto, describiendo la importancia de las dos principales tecnologías sobre las que se asienta el proyecto, así como su uso en la actualidad. Para ello, se aportan informes y estadísticas de todos los ámbitos a los que es aplicable el proyecto.

Este proyecto esta orientado a mejorar la robustez, fiabilidad y también escalabilidad de las SDN, ya que el utilizar Data Distribution Service (DDS) puede permitir pasar de un controlador centralizado a un conjunto de controladores federados y sincronizados que sean capaces de cooperar para compartir una misma imagen de la red. Si la red solo consta de un controlador centralizado, esta puede verse comprometida fácilmente, ya que la cantidad de tráfico que puede soportar un único controlador es limitada, lo que lleva a un funcionamiento de la red menos eficaz. Si la red sufre algún fallo o caída, el tiempo de recuperación con un solo controlador es bastante más elevado que si dos controladores colaborando entre sí actúan frente a fallos del sistema.

Por otra parte, también se analiza el estado actual de las redes definidas por software, viendo como su implantación, en sustitución de las redes tradicionales, puede mejorar la robustez, fiabilidad y escabalabilidad de las

redes.

1.1.1. La necesidad de una nueva arquitectura de red

La explosión del número de dispositivos móviles y el impresionante auge en la demanda de contenidos digitales, la virtualización de servidores, y la llegada de servicios en la nube son algunas de las tendencias que impulsan a la industria de las telecomunicaciones para volver a examinar el diseño y operación de las arquitecturas de red tradicionales. Muchas redes convencionales son jerárquicas, construidas con niveles de conmutadores Ethernet dispuestos en una estructura de árbol. Este diseño tiene sentido cuando la computación cliente-servidor era dominante, pero tal arquitectura estática no se adapta necesariamente bien a las necesidades de cómputo y de almacenamiento dinámico de los centros de datos empresariales de hoy en día, los campus, y demás ambientes de soporte. Algunas de las tendencias de la computación clave que impulsan la necesidad de un nuevo paradigma de red incluyen:

- **Patrones de tráfico cambiantes:** En los centros de datos empresariales, los patrones de tráfico han cambiado en gran medida. En contraste con las aplicaciones cliente-servidor, las aplicaciones de hoy en día acceden a diferentes bases de datos y servidores, creando una ráfaga de tráfico máquina-a-máquina antes de devolver los datos al dispositivo final del usuario. Al mismo tiempo, los usuarios están cambiando los patrones de tráfico de red a medida que impulsan el acceso a los contenidos corporativos y aplicaciones desde cualquier tipo de dispositivo (incluido el propio), conectándose desde cualquier lugar, en cualquier momento. Por último, muchos gerentes de centros de datos empresariales están contemplando un modelo de *utility computing*, que podría incluir una nube privada, nube pública, o alguna combinación de ambos, lo que resulta en más tráfico a través de la red.
- **Aumento de servicios *cloud*:** Las empresas han adoptado con entusiasmo tanto los servicios de nube pública y privada, lo que resulta en un crecimiento sin precedentes de estos servicios [1]. Los departamentos de negocio de las empresas ahora quieren que la agilidad para acceder a las aplicaciones, la infraestructura y otros recursos estén en la demanda y "a la carta". Para añadir más complejidad, esta planificación de servicios en la nube debe realizarse en un entorno de aumento de las necesidades de seguridad y auditoría. Proporcionar la capacidad de auto-servicio a un usuario, ya sea en una nube privada o pública, requiere de una escala elástica de computación, almacenamiento y recursos de red, idealmente, desde un punto de vista común y con un conjunto común de herramientas.

- **"Big Data"significa más ancho de banda:** El manejo del "Big Data"de hoy o de conjuntos de datos masivos requiere un procesamiento paralelo masivo de miles de servidores, todos los cuales necesitan conexiones directas entre sí. El aumento de los conjuntos de datos masivos está alimentando una demanda constante de la capacidad de red adicional en los centros de datos. Los operadores de redes de centros de datos hiperescala se enfrentan a la difícil tarea de ampliar la red, de un tamaño inimaginable ya de por sí, manteniendo la conectividad sin irse a la quiebra.

1.1.2. Limitaciones de las redes actuales

Cumplir con los requisitos actuales del mercado en cuanto a servicios de telecomunicación es prácticamente imposible con las arquitecturas tradicionales de red. Frente a los presupuestos planos o reducidos, los departamentos de las empresas de Telecomunicaciones están tratando de exprimir al máximo las redes usando herramientas de gestión y procesos manuales. Hay retos similares en cuanto a movilidad y demanda de ancho de banda, los beneficios se están viendo fuertemente reducidos debido a la escalada de costes de equipamiento de las redes. Las arquitecturas de red actuales no están diseñadas para cumplir con los requerimientos de los usuarios de hoy en día, empresas, etc. Además, los desarrolladores están "bloqueados" por las limitaciones de estas redes, que según la Open Networking Foundation (ONF) [15] incluyen:

- **Complejidad:** La tecnología de red hasta la fecha ha consistido principalmente en conjuntos discretos de protocolos diseñados para conectar los *hosts* a grandes distancias, tipos de enlaces y topologías. Para satisfacer las necesidades técnicas durante los últimos años, las empresas han tenido que evolucionar para ofrecer protocolos con un mayor rendimiento y fiabilidad, conectividad más amplia, y la seguridad más estricta. Cada protocolo está diseñado para resolver un problema concreto, por lo que no hay ningún tipo de abstracción en su diseño. Esto supone una de las principales limitaciones de las redes actuales. Para realizar cualquier cambio, por ejemplo en un dispositivo de la red hay que tener en cuenta todo el *hardware* y *software* de la misma, ya que pueden ser de distintos fabricantes. Debido a esta complejidad, las redes actuales son relativamente estáticas, lo que está las antípodas de las necesidades de los usuarios de hoy en día.
- **Políticas inconsistentes:** Para poner en marcha una política de red a gran escala, puede ser necesario modificar miles de dispositivos y mecanismos. Por ejemplo, cada vez que una máquina virtual entra en el sistema, puede llevar horas reconfigurar todas las Access List (ACL)

en la red. La complejidad de las redes actuales hace que sea muy difícil implantar políticas de acceso, seguridad y QoS consistentes.

- **Incapacidad para escalar:** Como la demanda de *Data Centers* crece muy rápido, también debe hacerlo la red. Sin embargo, la red se vuelve enormemente más compleja cuando se añaden cientos o miles de dispositivos que deben ser configurados y gestionados. Los enlaces también se sobrecargan porque aunque se pueden definir patrones de tráfico, en nuestros días, los flujos de datos son muy dinámicos. Las grandes compañías como Google, Yahoo o Facebook necesitan redes enormemente escaladas que puedan ofrecer alto rendimiento y conectividad a bajo coste con millones de servidores.
- **Dependencia del fabricante:** Las nuevas capacidades y servicios perseguidos por proveedores y empresas en respuesta rápida a las necesidades dinámicas de negocios y demanda de clientes, se ven frenadas por los ciclos de producción de los equipamientos por parte de los fabricantes, que pueden implicar periodos de hasta más de tres años.

En la Figura 1.1 se puede ver un informe realizado por Cisco, donde se detalla la evolución del tráfico IP en la red desde el año 2013 hasta la previsión que se ofrece para el año 2018. En incremento por año supone multiplicar por tres el dato del año anterior, lo que indica un aumento exponencial difícil de gestionar para las redes actuales.

Figura 1.1: Aumento del tráfico IP 2013-2018. [2]

No es difícil observar que las nuevas arquitecturas de red que se implementen deben soportar un volumen mucho mayor de tráfico, ser más flexibles y aumentar su velocidad. Algunos de estos requerimientos son descritos por la *Open Data Center Alliance* [14]:

- **Adaptabilidad:** Las redes futuras tiene que ser capaces de ajustarse y responder dinámicamente en función de las condiciones de la red y las necesidades de las aplicaciones. Esto está en relación con la dinámica de los flujos de datos descrita anteriormente.
- **Seguridad:** Las aplicaciones de red deben integrar seguridad como un servicio imprescindible, ya que cada día se descubren nuevas formas de aprovechar la vulnerabilidad de la red.
- **Mantenimiento:** La introducción de nuevas características y capacidades en la red (actualizaciones de software y firware) deben de ser transparentes con la mínima interrupción posible de las operaciones.
- **Gestión de red:** El software de configuración de la red debe permitir su gestión de forma conjunta, evitando la configuración de elementos individuales.
- **Escalabilidad:** Las implementaciones deben de tener la capacidad de escalar en toda su infraestructura, es decir, tanto en la parte referente a los *hosts* como en la parte referente al *core* de la misma.
- **Automatización:** Los cambios de políticas de red deben propagarse de forma automática de forma que los errores se reduzcan de forma notoria.

Una vez estudiadas las necesidades de los usuarios de hoy en día y las limitaciones de las redes actuales, queda demostrado que estas no suponen una buena solución de cara a un futuro inmediato. Por lo tanto, es en este escenario donde entra en juego un nuevo modelo de arquitectura de red, que parece ser el que se adoptará mayoritariamente en toda la red y que algunas empresas del calibre de Google ya implementa, las redes definidas por software (SDN).

1.2. Introduciendo el *Sotware-Defined Networking*

Las Redes definidas por software (SDN) son una arquitectura de red emergente en la cual el control de la red está desacoplada del plano de datos y es directamente programable. Esta migración del plano de control, anteriormente fuertemente unido al plano de control en los dispositivos de red, hacia dispositivos de computación más accesibles permite a la infraestructura subyacente ser extraída para aplicaciones y servicios de red, los cuales pueden tratar a la red como una entidad lógica o virtual.

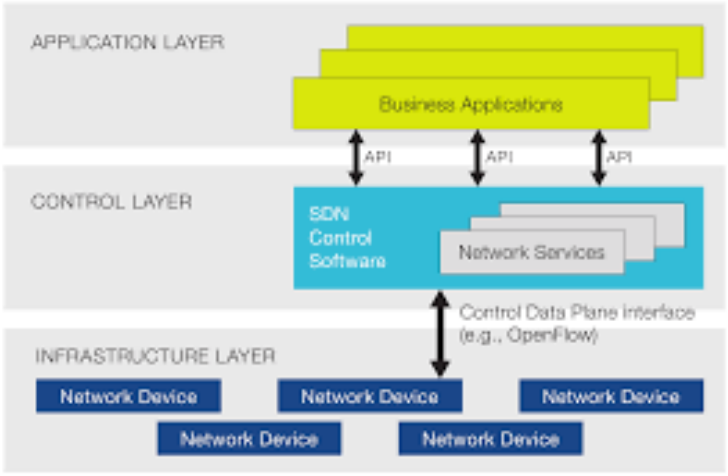

Figura 1.2: Arquitectura de red SDN. [15]

La Figura 1.2 representa una vista lógica de la arquitectura SDN. La inteligencia de red es (lógicamente) centralizada en los controladores SDN basados en software, que mantienen una visión global de la red. Como resultado, la red se muestra a las aplicaciones y políticas de ejecución como un sólo switch lógico. Con SDN, empresas y operadores aumentan la independencia del vendedor a la hora de diseñar la red, y se gana control sobre toda la red, ya que esta se controla desde un único punto lógico. SDN también simplifica en gran medida los dispositivos de red en sí, puesto que ya no tienen que entender y procesar miles de estándares de protocolos, sino simplemente aceptar instrucciones que reciben de los controladores SDN.

1.3. Introduciendo el *Data Distribution Service*

El *Data Distribution Service* DDS es una especificación de middleware que permite estandarizar el modelo de programación *Data-Centric Publish-Subscribe* (DCPS). Al implementar el modelo de publicación-suscripción para enviar y recibir datos y eventos entre los nodos, facilita enormemente la programación de la red. Los nodos publicadores son aquellos que crean los tópicos y crean muestras de estos tópicos. Los suscriptores a su vez se suscriben a estos tópicos para recibir las muestras publicadas. El estándar DDS gestiona todas las fases de la transferencia de información, así como la seguridad en dicha comunicación o la calidad de servicio.

Este estándar permite por lo tanto distribuir datos en una red Ethernet con dispositivos de bajo coste y con altos requerimientos de tiempo de entrega, ya que DDS trabaja en tiempo real. Otro aspecto interesante es que las

aplicaciones que incorporan DDS no necesitan ninguna información sobre el resto de aplicaciones participantes en el esquema descrito.

1.4. Objetivos principales del proyecto

En este apartado se establecen los objetivos principales del proyecto, para en los Capítulos 3 y 4 analizar su grado de cumplimiento.

El principal objetivo de este TFG es estudiar el problema de la federación de controladores SDN para que compartan una imagen única y común de un conjunto de *switches*. Como subobjetivos en este trabajo fin de grado se consideran los siguientes:

- Conocer y comprender la tecnología SDN y DDS.
- Revisión del estado del arte en cuanto a implementaciones.
- Instalación y pruebas de la implementación de un escenario con varios switches SDN controlados por varias instancias colaborativas interconectadas mediante DDS.
- Pruebas y evaluación de la implementación realizada.

1.5. Estructura de la memoria

En esta sección se enumeran los distintos capítulos que conforman el proyecto, realizando una breve descripción de cada uno de ellos. En concreto la memoria técnica esta formada por ocho capítulos:

- **Capítulo 1. Introducción.** En este capítulo se llevará a cabo una introducción de los aspectos más relevantes del proyecto, así como las motivaciones que han llevado a su implementación y los objetivos a conseguir con su realización.
- **Capítulo 2. Conocimientos teóricos y herramientas utilizadas.** Descripción de las tecnologías SDN y DDS, así como de todas las herramientas y programas que se usan en el desarrollo del proyecto.
- **Capítulo 3. Estado del arte.** En este capítulo se analizan algunas de las soluciones existentes similares a este proyecto, sirviendo como punto de partida para el desarrollo del mismo.
- **Capítulo 4. Planificación y estimación de costes.** En este capítulo se detalla una planificación a seguir con todas las etapas de desarrollo del proyecto, además de una estimación de los costes previstos en el diseño del mismo.

- **Capítulo 5. Diseño.**Aquí se describe todo el diseño de la arquitectura de la solución propuesta así como de los módulos desarrollados para los controladores SDN.
- **Capítulo 6. Implementación: Federación de controladores SDN.** En este apartado, se describen los pasos que se han llevado a cabo para implementar el diseño descrito en el capítulo anterior.
- **Capítulo 7. Evaluación.** Aquí se incluyen diversas pruebas realizadas sobre el escenario ya implementado, para comprobar su rendimiento y beneficios.
- **Capítulo 8. Conclusiones y Trabajo Futuro.** En este último capítulo se incluyen las conclusiones obtenidas una vez finalizado el proyecto, proponiendo mejoras y posibles trabajos futuros.
- **Bibliografía**. En la parte final se detalla la bibliografía con todo el contenido consultado durante la realización del proyecto.

Capítulo 2

Conocimientos teóricos y herramientas utilizadas

En este capítulo se describen, en primer lugar, todos los conocimientos necesarios para poder entender llevar a cabo la realización del proyecto. Por una parte, se analizan todos los aspectos relacionados con SDN, permitiendo conocer la arquitectura y funcionamiento de la red sobre la que se ha desarrollado la solución. Por otra parte, se explican los fundamentos del estándar DDS, ampliamente usado para comunicaciones en tiempo real, y que en este proyecto se ha usado con el fin de permitir la federación entre controladores SDN. Además, también se describen, de forma algo más resumida, todas las herramientas necesarias que se han usado para hacer posible la realización del trabajo.

2.1. Software-Defined Networking

Las redes definidas por software (SDN) son un conjunto de técnicas relacionadas con el área de redes computacionales.Tienen por objetivo facilitar la implementación e implantación de servicios de red de forma determinista, dinámica y escalable, evitando así al administrador tener que gestionar todo a bajo nivel. Todo esto se asienta sobre la característica fundamental de las redes definidas por software, la separación del plano de control (software) y del plano de datos (hardware).

Durante los últimos años, y gracias al uso de SDN, el diseño y gestión de redes se ha vuelto más innovador. Empresas del calibre de Google las usa en sus redes. Sin embargo, esta tecnología no ha tenido una aparición súbita, sino que es una larga historia de avances desde hace más de 20 años. La historia de las redes se divide en tres etapas fundamentalmente: redes activas, separación del plano de control del plano de datos y la aparición de

la interfaz de programación de aplicaciones de OpenFlow.

La principal idea de SDN es externalizar la responsabilidad del control de flujo y la inteligencia de red fuera del hardware de los *switches* y *routers*, centralizándolo en un servidor llamado "controlador", que realiza todas las tareas que en las redes tradicionales realizaban los protocolos de forma separada, es decir, actúa como el cerebro de la red.

Las SDN tienen múltiples ventajas. Al poder tener una visibilidad completa de la red desde los controladores, en lugar de la actual vista parcial a la que están restringidos los *switches* y *routers* de las redes tradicionales, es posible redirigir flujos con gran agilidad y conseguir una optimización máxima de los recursos en menor tiempo, incluso mediante comportamientos que son imposibles de manejar por los protocolos independientes que actualmente manejan la red. También se mejora la respuesta ante fallos o incluso congestión en la red, ya que al tener una mejor visibilidad de la topología de la red, el controlador puede calcular el camino más rápido alternativo extremo a extremo, en lugar del procesamiento local que se hace actualmente.

2.1.1. Funcionamiento

Cualquier red SDN consta siempre de los mismos elementos, el plano de datos (esta parte puede estar formada por *switches* OpenFlow o por *switches* tradicionales que incorporan adaptadores OpenFlow), el controlador, una API "hacia el sur" (*southbound* API) y una API "hacia el norte" (*northbound* API). En la Figura 2.1 se puede ver como se interconectan todos estos elementos.

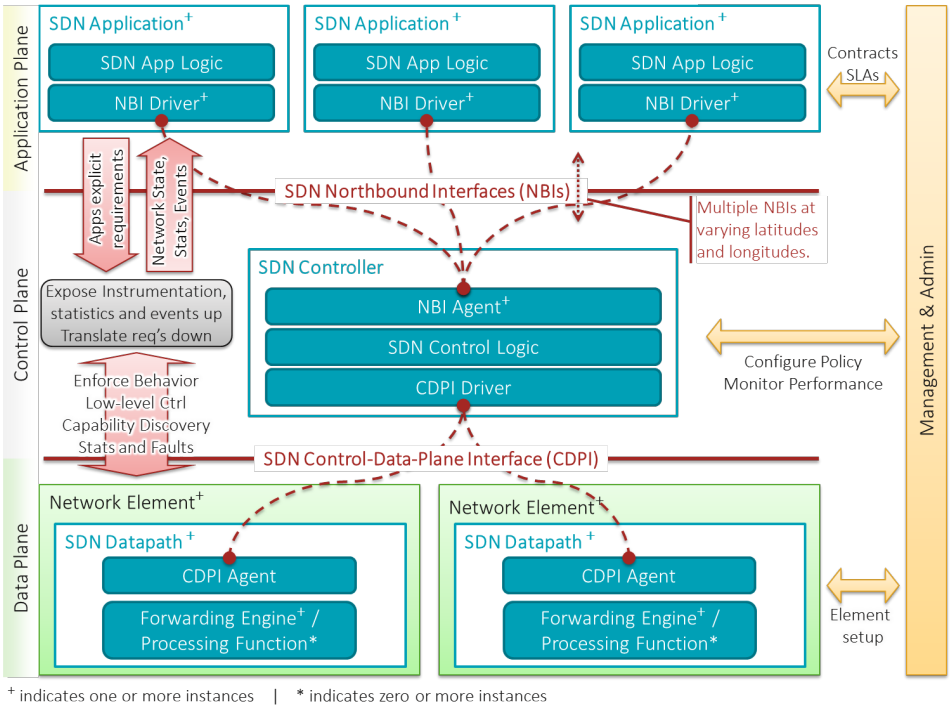

Figura 2.1: Elementos red SDN. [4]

Un controlador SDN toma el papel de "cerebro" de dicha red, es decir, el punto de control estratégico que retransmite la información de los *switches* de abajo (a través de la *southbound API*) y a las aplicaciones y la lógica de negocio encima (a través de la *northbound API*). Recientemente, se viene persiguiendo la tarea de asociar dominios de controladores SDN, que es objetivo principal de este proyecto. Un controlador SDN está basado en un conjunto de módulos *pluggable* que realizan diversas tareas de red, como realizar un inventario de todos los aparatos de red disponibles en ésta, recolectar sus capacidades, agrupar estadísticas de red, etc.

Las interfaces *northbound* y *southbound* sirven para conectar el controlador SDN a los aparatos de red por debajo, como pueden ser los *switches* OpenFlow, y a los servicios y aplicaciones por encima.

Las *Southbound APIs* facilitan el control en la red, permitiendo al controlador realizar cambios dinámicos de acuerdo a las demandas en tiempo real. OpenFlow, desarrollado por la ONF (Open Networking Foundation) es la primera y más conocida de estas interfaces.

Las *Northbound APIs* en cambio, pueden ser usadas para facilitar la innovación y permitir la organización y automatización de la red para suplir las necesidades de las diferentes aplicaciones a través de la programabilidad

de la red SDN. Estas interfaces son las más críticas, ya que tienen que soportar multitud de aplicaciones y servicios.

Aunque, como se ha dicho antes, el controlador es el que se encarga de la lógica de toda la red, es el protocolo OpenFlow el que se encarga de la programación de los *switches* (según como indique el controlador) y la gestión de toda la red. También es el encargado de intercambiar los mensajes entre los *switches* y el controlador y viceversa. Fundamentalmente, hay dos modelos de funcionamiento: proactivo y reactivo. Cuando un flujo llega a un *switch* se realiza un mapeo de la tabla de flujos. En el caso de que no se encuentre coincidencia, se envía una petición al controlador para instrucciones más extensas. En modo reactivo, el controlador actúa después de estas peticiones y crea una regla en la tabla de flujos para el paquete correspondiente si es necesario. En modo proactivo, sin embargo, el controlador llena entradas de la tabla de flujos para cada posible coincidencia de tráfico para ese switch, algo parecido con las típicas entradas de tablas de enrutamiento. Además de estos modelos por separado, se puede combinar en una red ambos modelos en forma de un híbrido, para aportar las ventajas de ambos. [9]

2.1.2. Arquitectura

La arquitectura de SDN se compone de tres capas bien diferenciadas, las cuáles pueden verse fácilmente en la Figura 2.2.

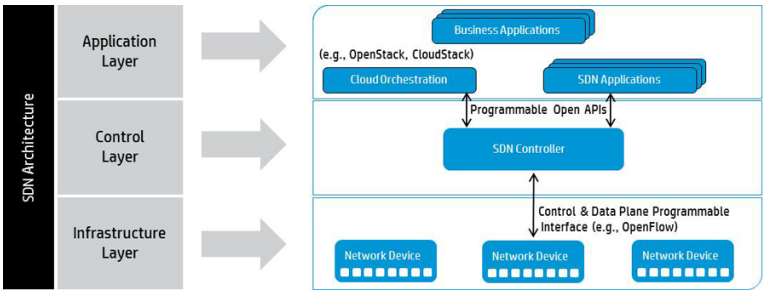

Figura 2.2: Capas de red SDN. [3]

- **Capa de infraestructura:** en la que reside el hardware. Los componentes de esta capa son dispositivos de red, pero solo se tiene en cuenta el hardware que lo compone, no los protocolos que contenga ni ningún tipo de inteligencia, por lo que pueden ser equipos de muy bajo coste y no tendrían que ser de un único fabricante, tan solo deben contar con el soporte del método que les conecta con el controlador, es decir, el protocolo OpenFlow.

- **Capa de control:** aquí se encuentran los controladores, quienes mantienen la inteligencia de la red. Los controladores se comunican con los equipos de red de la capa de infraestructura mediante el uso del protocolo OpenFlow, y brindan una API abierta para que puedan ser programados desde la capa de aplicación.
- **Capa de aplicación:** es la de más alto nivel y desde la cual se controla el comportamiento de la red. Para ello se utiliza la API del controlador, que puede ser utilizada por usuarios, por otro software de gestión o aplicaciones de infraestructura corporativas.

Aunque SDN se compone de muchas tecnologías habilitadoras, SDN no es una tecnología, sino una arquitectura novedosa de red. Ya sea de estructura o de superposición, la virtualización de redes puede considerarse como una aplicación SDN. Sin embargo, SDN tiene otros beneficios potenciales incluyendo aliviar la carga administrativa de funcionalidades de aprovisionamiento tales como Quality of Service (QoS) y la seguridad.

2.2. Data Distribution Service

El servicio de distribución de datos para sistemas de tiempo real (DDS) es un *middleware machine-to-machine* que tiene como objetivo permitir intercambio de datos entre *Publishers* y *Subscribers* de forma escalable, en tiempo real, fiable y de alto rendimiento. DDS atiende las necesidades de aplicaciones como el comercio financiero, el control de tráfico aéreo, la gestión de redes inteligentes, y otras aplicaciones como el *Big Data*. Este estándar se utiliza también en aplicaciones como los sistemas operativos de los *smartphones*, los sistemas de transporte y vehículos, de radio definida por software, y por dispositivos médicos. Actualmente, DDS esta tomando gran relevancia en el apartado del Internet de las Cosas (IoT).

Unos pocos sistemas propietarios de DDS llevan disponibles bastante tiempo. Las empresas que tienen un desarrollo más avanzado en este estándar son la americana *Real Time Innovations* (el software usado en este proyecto es de esta empresa) y PrimsTech, entre otras. La especificación de DDS incluye dos niveles de interfaces:

- Una capa inferior Data-Centric Publish-Subscribe (DCPS) cuyo objetivo consiste en la prestación eficiente de la información adecuada a los destinatarios adecuados.
- Una capa opcional superior DLRL (Data Local Reconstruction Layer), cuyo objetivo es permitir una integración simple de DDS en la capa de aplicación.

DDS podría describirse como un *middleware* de redes que simplifica la compleja programación de la red. Implementa un modelo de publicación/-suscripción para enviar y recibir datos, eventos y comandos entre varios nodos. Los nodos que producen información crean *topics* y publican "muestras". DDS entrega las muestras a los suscriptores que declaran interés en ese tópico. En el siguiente apartado, en el cual se describe el software de la empresa RTI basado en DDS, se da más información sobre este estándar

2.3. Herramientas utilizadas

2.3.1. RTI Connext DDS

Connext DDS es un *middleware* de red para aplicaciones distribuidas en tiempo real desarrollado por la mundialmente conocida empresa líder en su sector *Real Time Innovations*. Provee de servicios de comunicaciones que los programadores necesitan para distribuir datos con un tiempo crítico entre dispositivos o nodos embebidos. Connext DDS usa el sistema de comunicación de **publicación-suscripción**, para hacer la distribución de datos eficiente y robusta.

Connext DDS implementa el Data-Centric Publish-Subscribe (DCPS) API con el estándar DDS para sistemas de tiempo real. La aproximación *data-centric* implica que el *middleware* es capaz de filtrar los datos y poderlos almacenar en la memoria caché. DDS es el primer estándar desarrollado para las necesidades de tiempo real. DCPS provee una manera eficiente de transferir datos en un sistema distribuido.

Con Connext DDS, los diseñadores y programadores de sistemas empiezan con una infraestructura flexible y tolerante a fallos que trabaja sobre una amplia variedad de hardware, sistemas operativos, lenguajes, y protocolos de transporte. Connext DDS es altamente configurable, así que los programadores pueden adaptarlo a su gusto para encontrar los requerimientos de comunicación necesarios.

Como se ha dicho antes, Connext DDS usa el modelo de comunicación de publicación-suscripción. En este modelo, las aplicaciones del ordenador (nodos) se "subscriben" a los datos que necesitan y "publican" los datos que quieren compartir. Los mensajes pasan directamente entre el publicador y el subscriptor, en vez de moverlos dentro y fuera de un servidor localizado. La información con más restricciones de tiempo usa este modelo.

Middleware es una capa software situada entre la aplicación y el sistema operativo. Se encarga de aislar la aplicación de los detalles de las capas subyacentes, como la arquitectura del nodo, el sistema operativo, etc. El uso de la capa *middleware* simplifica el desarrollo de sistemas distribuidos

permitiendo a las aplicaciones enviar y recibir información sin tener que usar protocolos de bajo nivel como TCP o UDP.

Connext DDS soporta mecanismos que van mas allá del modelo básico de publicación-suscripción. Las aplicaciones que usan este software están completamente desacopladas. Las aplicaciones nunca necesitan información sobre las otras aplicaciones, incluyendo su existencia o localizaciones. Connext DDS automáticamente gestiona todos los aspectos de la entrega de mensajes, sin requerir ninguna intervención de la aplicación de usuario. Además de todo esto, Connext DDS incluye algunas características que buscan satisfacer las características de las aplicaciones de tiempo real.

- **Tipos de datos definidos por el usuario:** Habilita al programador para modificar el formato de la información que va a ser transmitida por cada aplicación. Para hacer esto posible se usa el lenguaje Interface Language Description (IDL). Este lenguaje va a permitir la creación de una plantilla sobre la cual se crearán todas las muestras DDS que sea necesario compartir entre los controladores. Estas plantillas son conocidas como *extensyble-types* y permiten aumentar el número de formatos o de tipos de datos que se pueden transmitir usando DDS. Por defecto solo se puede transmitir un *String* o un *Byte* en cada muestra.
- **Intercambio de mensajes seguro:** La seguridad también es un aspecto muy importante de Connext DDS. Las comunicaciones pueden ser cifradas y se puede incluir autenticación entre las aplicaciones participantes.
- **Múltiples redes de comunicación:** Diferentes redes de comunicación pueden ser usadas sobre la misma red física usando Connext DDS. Las aplicaciones son solo capaces de participar en el dominio DDS al que pertenecen, aunque las aplicaciones individuales pueden ser configuradas para participar en múltiples dominios DDS.
- **Arquitectura simétrica:** Permite que la aplicación sea más robusta, eliminando posibles fuentes de fallo.
- **Soporte para múltiples lenguajes:** Incluye APIs para C, C++ y Java.
- **Soporte multiplataforma:** Incluye soporte para UNIX, sistemas operativos de tiempo real y Windows, aunque en cada versión nueva que lanzan aumenta el número de plataformas compatibles.

La arquitectura del modelo publicación-suscripción se compone de varios elementos que interaccionan entre sí. A continuación, se irán explicando

todos los elementos que forman dicha arquitectura, la cual puede verse esquematizada en la Figura 2.3.

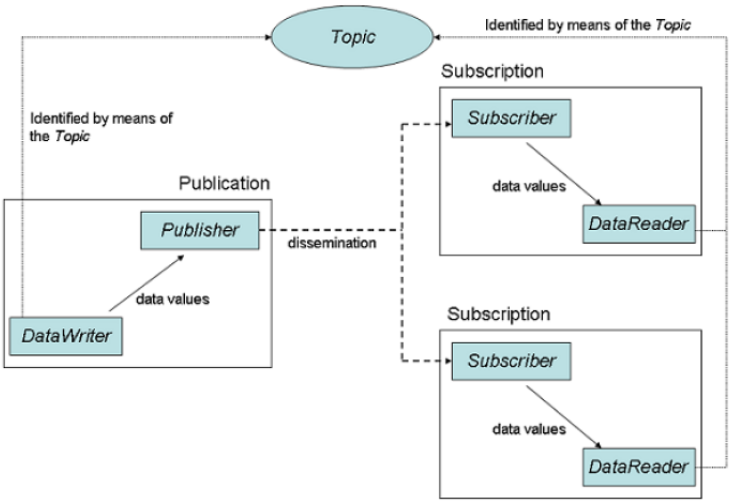

Figura 2.3: Esquema RTI Connext DDS. [17]

Aunque ya se han dado algunas pinceladas sobre algunos de estos elementos, no se ha comentado nada sobre la base del modelo de publicación-suscripción: los tópicos. Compartir el conocimiento sobre los tipos de datos es un requerimiento para diferentes aplicaciones para comunicarse mediante DCPS. Estas aplicaciones deben tener también un modo de identificar que datos van a compartirse. Los datos se distinguen únicamente usando un nombre llamado *Topic*. Por definición, un *Topic* corresponde a un único tipo de datos. Sin embargo, varios *Topics* pueden hacer referencia al mismo tipo de datos.

Una aplicación usa *DataWritters* para enviar datos. Un *DataWritter* se asocia con un *Topic*. Se pueden tener múltiples *DataWritters* y *Topics* en una misma aplicación. Además se pueden tener varios *DataWritters* asociados a un *Topic* en una aplicación.

El *Publisher* es el objeto DCPS responsable del envío de datos. Los *Publishers* gestionan los *DataWritters*. Un *Publisher* puede gestionar varios *DataWritters*, además, el *Publisher* puede enviar datos de diferentes *Topics*. Cuando el código del usuario llama al método *write()*, la muestra de datos DDS se pasa al *Publisher*, el cual se encarga de diseminar los datos por la red.

Una aplicación usa *DataReaders* para acceder a los datos recibidos por DCPS. Un *DataReader* se asocia con un único *Topic*. Se pueden tener múlti-

ples *DataReaders* y *Topics* en una misma aplicación. Al igual que pasaba en el lado de envío, aquí también se pueden tener mas de un *DataReader* para un *Topic*.

El *Subscriber* es similar al *Publisher*, aunque este se encarga de la recepción de los datos publicados. Cuando los datos son enviados a las aplicaciones, primero son procesados por el *Subscriber*. Después, la muestra DDS es almacenada en el *DataReader* que corresponda. El usuario, mediante código, puede registrar un *listener* que será llamado cuando lleguen nuevos datos y cuando se tomen datos que hayan sido previamente almacenados usando los métodos *read()* o *take()*.

Toda la estructura descrita anteriormente, se engloba dentro del concepto de "Dominio DDS". Los dominios DDS representan redes lógicas de comunicación aisladas entre si. Múltiples aplicaciones funcionando en el mismo conjunto de *hosts*, pero en dominios DDS diferentes, están complemente aisladas unas de otras (aunque estén en la misma máquina). Una aplicación puede pertenecer a varios dominios DDS simultáneamente creando un *DomainParticipant* por dominio. Sin embargo los *Publishers* y *Subscribers* solo pertenecen al dominio DDS en el que fueron creados.

Connext DDS supone un software muy potente para trabajar en entornos en los que se usa el estándar DDS, gracias a sus múltiples características para intercambiar datos en tiempo real en entornos distribuidos. Esto lo convierte en la herramienta idónea para el diseño de la parte de comunicación entre controladores.

2.3.2. OpenFlow

OpenFlow [10] es el protocolo que se encarga de la gestión de los *switches* en las redes SDN. La idea sobre la que se asienta OpenFlow es bastante simple, se explota el hecho de que la mayor parte de los *swtiches* y *routers* contienen tablas de flujo que funcionan en línea para implementar *firewalls*, Network Address Translation (NAT), QoS y para recolectar estadísticas. Sin embargo, la tabla de enrutamiento de cada vendedor puede ser diferente, por lo tanto el objetivo principal de OpenFlow es explotar estas funcionalidades comunes.

A pesar de que la virtualización está empezando a tomar gran peso, cuando comparamos los sistemas abiertos con los *switches* comerciales, se observa cierta diferencia entre los atributos existentes y los deseados por el cliente de cada una de estas tecnologías. Para paliar este defecto se creó el protocolo OpenFlow. En la Figura 2.4 se puede observar los atributos que ofrecen los sistemas abiertos en relación a los *switches* comerciales.

	Performance Fidelity	Scale	Real User Traffic?	Complexity	Open
Simulation	medium	medium	no	medium	yes
Emulation	medium	low	no	medium	yes
Software Switches	poor	low	yes	medium	yes
NetFPGA	high	low	yes	high	yes
Network Processors	high	medium	yes	high	yes
Vendor Switches	**high**	**high**	**yes**	**low**	no

Figura 2.4: *Switches* tradicionales frente a sistemas abiertos. [10]

OpenFlow es básicamente un protocolo abierto para programar la tabla de flujo en diferentes *switches* y *routers*, es decir, un administrador de red que puede dividir el tráfico en flujos de producción e investigación. Los *switches* controlan sus propias tablas de flujo, eligen las rutas que siguen los paquetes entrantes y el procesamiento que reciben.

Switch OpenFlow

El camino de datos (datapath) de un *switch* OpenFlow consiste de una tabla de flujo y una acción asociada con cada entrada de flujo. El conjunto de acciones soportadas por un *switch* OpenFlow es extensible pero se establecen unos requisitos mínimos para todos los *switches*. Para que sea posible la implementación de este protocolo debemos partir de la premisa de que se debe contar con un *Switch* OpenFlow, cuyos requerimientos se explican a continuación:

Un *switch* OpenFlow consiste básicamente de 3 partes: una **Tabla de Flujo**, con una acción asociada con cada entrada de la misma, la cual le indica al *switch* como procesarlo; un **Canal Seguro** que conecta al *switch* con un "controlador", permitiendo que los paquetes y comandos puedan ser intercambiados entre ambos usando el protocolo **OpenFlow**, que es la tercera parte, el cual se encarga de controlar la comunicación, pudiendo el controlador, a través de dicho protocolo, añadir, actualizar y borrar entradas de la tabla de flujo tanto de forma reactiva como proactiva.

Hay dos tipos de *switches* que soportan el protocolo: por un lado están los Switches OpenFlow, que no soportan procesamiento de nivel 2 y 3, es decir, están *pensados* únicamente para ser utilizados con OpenFlow, por lo que su uso es exclusivo en SDN; por otro lado, están los switches convencionales a los que se les añaden "habilitadores" OpenFlow, que permiten aumentar sus prestaciones, asemejando su rendimiento al de los de primer tipo. En la

Figura 2.5 se puede ver la estructura de un *Switch* OpenFlow.

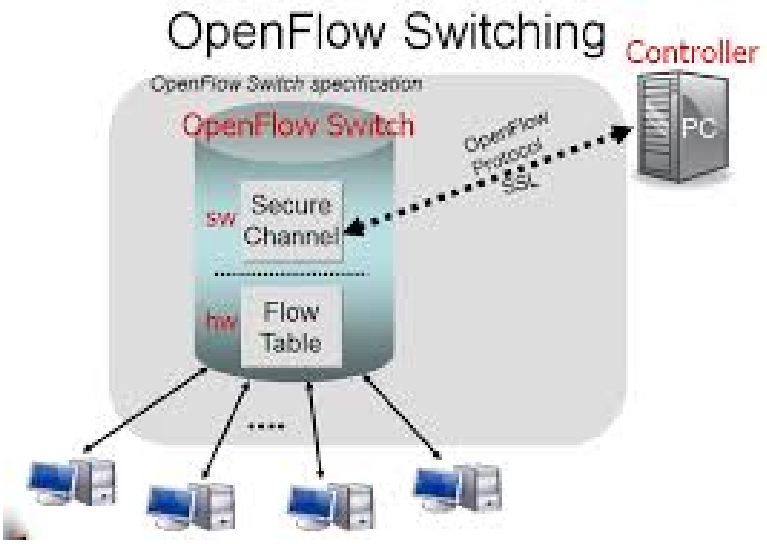

Figura 2.5: Arquitectura de un *switch* OpenFlow. [13]

Cada entrada de la tabla de flujo puede tener asociada una acción, las tres básicas (soportadas por todos los switches OpenFlow) son:

- **Enviar el flujo de paquetes a un puerto (o puertos) establecidos**. Esto permite que los paquetes sean enrutados a través de la red. En la mayoría de los *switches* esta función se realiza a la velocidad de línea.
- **Encapsular y enviar dichos flujos de paquetes al controlador.** El paquete se pone en el canal seguro, donde se encapsula y se envía al controlador. El controlador decide si el flujo debe ser añadido a la tabla de flujos.
- **Lanzamiento del flujo de paquetes.** Estos pueden ser utilizados para seguridad, para frenar ataques de Denegation of Service (DoS) o para reducir el descubrimiento de tráfico falso de *broadcast* por parte de los usuarios finales.

Hay varios tipos de mensajes contemplados en el protocolo Openflow. Por un parte están los mensajes "switch-a-controlador", que son los mensajes iniciados por el controlador que pueden o no requerir una respuesta del *switch*. También están los mensajes asíncronos, que son los mensajes que los *switches* transmiten al controlador sin previo aviso, como por ejemplo los mensajes para denotar la llegada de un paquete. Por último, también existen

los mensajes simétricos, que son mensajes utilizados en el establecimiento de la conexión.

El uso de este protocolo para la realización de este proyecto es esencial, ya que OpenFlow podría considerarse como el "gestor"de la red SDN, por lo que permite controlar todos los switches de la red creada así como su comunicación con el controlador Opendaylight. Esta herramienta se utiliza en conjunción con la siguiente, Mininet.

2.3.3. MiniNet

MiniNet es un sistema de red de emulación. Se ejecuta una colección de *hosts* finales, conmutadores, enrutadores y enlaces en un solo núcleo de Linux. Se utiliza la virtualización ligera para hacer que un solo sistema se parezca a una red completa, ejecutando el mismo kernel, sistema y código de usuario. Un *host* en MiniNet se comporta como una máquina real; puede entrar mediante SSH en él (si inicia *sshd* y establece un puente desde la red hasta el *host*) y ejecutar programas arbitrarios (incluyendo cualquier cosa que se instala en el sistema Linux subyacente). Los programas que se ejecutan pueden enviar paquetes a través de lo que parece ser una verdadera interfaz Ethernet, con una velocidad de enlace y retardo dados. Los paquetes se procesan por un elemento muy parecido a un conmutador Ethernet real, un *router* o *middlebox*, con un tamaño de cola dado. Cuando dos programas, como un cliente y un servidor *iperf*, se comunican a través de MiniNet, el rendimiento medido debe coincidir con el de las dos máquinas nativas (más lento por separado).

En resumen, los *hosts* virtuales, *switches*, enlaces, y los controladores de MiniNet son como elementos reales (que sólo se crean utilizando software en lugar de hardware) y en su mayor parte su comportamiento es similar a los elementos de hardware discretos. Por lo general, es posible crear una red MiniNet que se asemeja a una red de hardware, o una red de hardware que se asemeja a una red MiniNet, para ejecutar el mismo código binario y aplicaciones en cualquier plataforma. MiniNet tiene ciertas ventajas que hacen su uso muy recomendable para crear redes y escenarios de simulación. Algunas de estas ventajas son:

- **Es rápido:** Arrancar una topología simple sólo tarda unos pocos segundos. Esto significa que se puede ejecutar el bucle de *run-edit-debug* en muy poco tiempo.
- **Se pueden crear topologías personalizadas:** El abanico de topologías que se pueden crear es muy amplio, desde un simple *switch*, grandes topologías como en Internet, hasta DataCenters, o lo que el usuario desee.

- **Ejecutar programas reales:** Todo lo que pueda ser ejecutado en Linux puede ser ejecutado también en los *hosts* de Mininet, desde servidores web hasta herramientas de monitorización como Wireshark.
- **Es posible compartir resultados:** Cualquiera con un ordenador puede ejecutar código hecho por otra persona una vez que este ha sido empaquetado.
- **Mininet es *open-source*:** El código completo de MiniNet esta a disposición de todos los usuarios, para que estos puedan modificarlo, resolver problemas y *bugs*, etc.

Es este trabajo, MiniNet se ha usado para poder crear las topologías sobre las cuales se despliegan los escenarios a estudiar, ya que es posible elegir que el controlador de la red sea remoto, lo que permite la conexión de MiniNet con Opendaylight.

2.3.4. Opendaylight

Anunciado en Abril del 2013, el proyecto OpenDayLight [7], un proyecto SDN *open source* alojado en la Linux Foundation, fue creado para avanzar en la adopción de las redes definidas por software. Es un proyecto con una plataforma de controlador flexible, modular y *pluggable* como núcleo. El controlador está implementado en su totalidad en software y contiene su propia máquina virtual de Java (JVM). Además, el controlador Opendaylight puede ser desplegado en cualquier plataforma hardware que soporte Java. El controlador expone APIs "northbound" abiertas que son usadas por las aplicaciones. Opendaylight soporta el *framework* Open Services Gateway initiative (OSGi) y comunicación REST bidireccional con la API "northbound". El *framework* OSGi se usa para aplicaciones que se ejecutan en el mismo dominio que el controlador, mientras que REST (basado en web), lo usan las aplicaciones que no se ejecutan en la misma página que el controlador. La lógica y los algoritmos residen en las aplicaciones. Estas aplicaciones usan el controlador para reunir información de la red, ejecutar algoritmos para realizar análisis, y luego usar el controlador para orquestar la nueva normativa, en su caso, a través de la red.

El controlador en sí mismo contiene una colección de módulos dinámicamente conectables para poder llevar a cabo las tareas de la red. Hay una serie de funciones de red básicas que vienen por defecto, las cuales permiten reunir estadísticas de la red, ver la topología, etc. Estas funciones son conocidas como las *core network functions*. Además, en la arquitectura del controlador, hay otros elementos muy importantes como los los *plugins* "southbound", que se conectan al controlador mediante la capa de abstracción de servicio

(Service Abstraction Layer (SAL)), y los *plugins* "northbound", en su mayor parte aplicaciones, que se conectan al controlador mediante la REST API. En el Figura 2.6 se puede observar la arquitectura esquematizada del controlador Opendaylight.

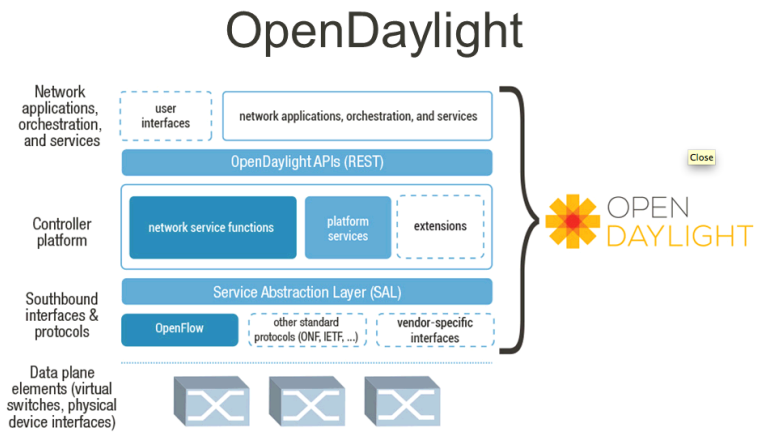

Figura 2.6: Arquitectura de un switch OpenFlow.

El proyecto Opendaylight está siempre en constante desarrollo debido a la contribución de su comunidad de desarrollo. Además su uso en este proyecto se debe a su gran estabilidad y fácil manejo. Cada día son más las empresas que se unen al grupo de desarrollo, ya que el proyecto está en auge, saliendo nuevas versiones cada poco tiempo. Aun así, hay que destacar que para los usuarios que no pertenecen a grandes empresas y que necesitan usar este controlador para sus proyectos, como es el caso de este trabajo, la falta de información para desarrollar componentes y aplicaciones para el controlador es muy alarmante.

2.3.5. Otras herramientas

Aparte de las herramientas que ya se han descrito, las cuales han sido usadas para implementar la solución propuesta en este proyecto, se han usado otras herramientas de apoyo. Los casos más importantes a destacar son:

- **Maven:** Herramienta de software para la gestión y construcción de proyectos Java. Esta herramienta ha sido usada en el trabajo principalmente para poder compilar todos los programas generados con fin de ser incluidos en el controlador Opendaylight. También se ha usado para generar el arquetipo, es decir, la estructura de carpetas y archivos de dichos programas.

- **Github:** Herramienta de sobra conocida, GitHub es una plataforma de desarrollo colaborativo, utilizada para alojar proyectos utilizando el sistema de versiones de Git. En este proyecto, se ha utilizado para descargar todos los repositorios que contenían las dependencias del controlador Opendaylight, necesarias para que las aplicaciones creadas funcionasen correctamente.

- **IDE Eclipse:** Es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados, como el IDE de Java. En este trabajo ha sido elegido como el IDE a utilizar para diseñar los programas que posteriormente se cargan en el controlador Opendaylight. El *plugin* de Maven para este IDE hace mucho más fácil la gestión de librerías.

Una vez expuestos los conocimientos teóricos necesarios para el correcto desarrollo del proyecto, así como un vista de las herramientas y aplicaciones que se han usado en la implementación del mismo, el siguiente paso es pasar al diseño de la solución propuesta.

Capítulo 3

Estado del Arte

En este capítulo se pretende analizar las soluciones alternativas más relevantes relacionadas con la federación de controladores en redes SDN, con el objetivo de que supongan un punto de partida para poder realizar una comparación apropiada.

3.1. HyperFlow

Openflow, el protocolo que se encarga de la gestión de la red, asume un controlador centralizado lógico, que idealmente puede ser físicamente distribuido. Sin embargo, los despliegues actuales confían en un solo controlador, lo que provoca problemas de escalabilidad y de retardo.

HyperFlow [18], desarrollado por Amin Tootoonchian y Yashar Ganjali, presenta un panel de control distribuido basado en eventos para OpenFlow. HyperFlow es lógicamente centralizado pero físicamente distribuido; provee escalabilidad manteniendo las ventajas de la red centralizada. Sincronizando pasivamente las vistas de toda la red de los controladores individuales, minimizando así el tiempo de respuesta a las peticiones del panel de datos. HyperFlow, según sus creadores, es resistente al fallo de componentes individuales. También habilita la interoperabilidad entre redes independientes gestionadas con OpenFlow. Todos los controladores comparten la misma vista de la red, y sirven peticiones localmente sin conectarse a ningún nodo remoto, minimizando así el retardo. Adicionalmente, HyperFlow no requiere implementar ningún cambio en el estándar de OpenFlow.

HyperFlow está diseñado para ser implementado en el controlador Nox [5]. La aplicación es la encargada de sincronizar todas las vistas de la red de los controladores (propagando eventos especiales generados localmente en cada controlador), redirigir comandos OpenFlow a *switches* que no están directamente conectados a su controlador correspondiente y redirigir las res-

puestas de los *switches* al controlador que originó la petición. Para facilitar la comunicación entre controladores, se usa un sistema de publicación/subscripción de mensajes.

3.1.1. Diseño

Cada *switch* está conectado al controlador más próximo, todos los controladores usan el mismo software y comparten la misma imagen de la red. Cada controlador gestiona los *switches* conectados directamente a él, e indirectamente programa o almacena datos del resto. En la Figura 3.1 se muestra la arquitectura de HyperFlow.

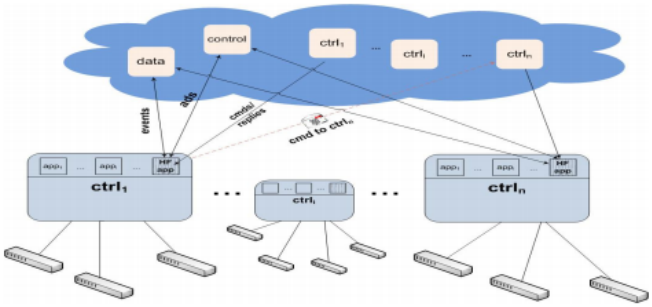

Figura 3.1: Arquitectura de HyperFlow. [18]

Para conseguir tener una vista distribuida de la red, cada controlador publica los eventos que cambian el estado de todo el sistema, el resto de controladores replica el mensaje, reconstruyendo así el estado del sistema. Este diseño está basado en las siguientes observaciones: (a) cualquier cambio en la vista de la red de los controladores se origina con un evento de la misma. Un simple evento puede cambiar el estado de muchas aplicaciones. (b) Sólo una fracción muy pequeña de los eventos de la red causan cambios en ella. La mayoría de los eventos consisten solamente en routing, por ejemplo. (c) El orden temporal de los eventos en un *swtich* no afecta a la vista de la red. (d) Las aplicaciones solo necesitan ser ligeramente modificadas para detectar los eventos que afectan a su estado.

Para propagar los eventos de un controlador al resto, HyperFlow usa el paradigma de mensajes publicación/subscripción, en concreto, WheelFS. WheelFS es un sistema distribuido de archivos diseñado para ofrecer un área flexible de almacenamiento para aplicaciones distribuidas. En WheelFS se representan canales con directorios y mensajes con ficheros. Para implementar la notificación de un mensaje entrante, HyperFlow comprueba el estado de los directorios para ver si ha habido algún cambio.

3.1.2. Funcionamiento

Cada controlador se subscribe a tres canales en la red: el canal de control, el canal de datos y su propio canal. Todos los controladores de la red tienen permiso para publicar y recibir mensajes en los tres canales. La aplicación HyperFlow publica los eventos de las aplicaciones y los controladores que son de interés general en el canal de datos. Los eventos y los comandos OpenFlow dirigidos a un controlador son publicados en el canal de control. Adicionalmente, cada controlador debe, periódicamente, anunciarse al resto de la red para poder detectar posibles fallos. A continuación se describen las funciones de la aplicación HyperFlow de los controladores:

- **Inicialización:** La aplicación HyperFlow inicia el cliente WheelFS y el servicio de almacenamiento, suscribe al controlador en los canales de datos y control, y empieza a anunciarse periódicamente en la red. Este mensaje contiene, entre otras cosas, el identificador de los switches directamente gestionados por el controlador.
- **Publicar eventos:** La aplicación HyperFlow captura todos los eventos generados por los mensajes OpenFlow y los que generan las aplicaciones que usan HyperFlow. Luego, la aplicación publica los eventos que han sido generados localmente y afectan al estado del controlador o la red. Usando este método, el número de eventos propagados está limitado por el número de mensajes Openflow generados por el controlador. El nombre de los mensajes publicados contiene el identificador del controlador y el identificador del propio evento. En la Figura 3.2 se puede ver la convención existente para el nombre de los mensajes.

Message Type	Message Name Pattern
Event	e : *ctrl_id* : *event_id*
Command	c : *ctrl_id* : *switch_id* : *event_id*
Advertisement	*ctrl_id*

Figura 3.2: Tipos de mensaje en HyperFlow.

- **Reproducir eventos:** La aplicación HyperFlow reproduce todos los eventos publicados, porque los controladores filtran y solo publican los eventos necesarios para reconstruir el estado de la red.
- **Redirigir comandos a un *non-local switch*:** Un controlador puede solo programar los *switches* que tiene directamente conectados. Para programar un *switch* que no está bajo control, la aplicación HyperFlow detecta cuando se va a enviar un mensaje OpenFlow a dicho *switch*

y publica el comando en el canal de control. El nombre del mensaje publicado muestra que es un comando, el identificador del controlador, el del *switch* destino y el del propio comando.

- **Encaminar mensajes OpenFlow:** La aplicación HyperFlow toma un comando dirigido a un *switch* bajo la gestión del controlador y lo envía. Para enrutar la respuesta, HyperFlow usa un "mapping" entre los identificadores de los mensajes de la transacción y el identificador del controlador de origen.
- **Prueba de estado:** La aplicación HyperFlow escucha los anuncios del controlador en el canal de control. Si el controlador falla, la aplicación manda un switch leave a cada uno de los *switches* conectados al controlador que ha fallado, y se encarga de conectarlos a un nuevo controlador.

Las aplicaciones instaladas en el controlador también tienen que cumplir con unos requerimientos para que puedan ser usadas en conjunción con HyperFlow. Algunos de estos requerimientos son: asegurar el correcto funcionamiento bajo una posible reordenación de eventos y estados transitorios de la red, capacidad de corrección y escalabilidad. Por lo tanto, en algunos casos es necesario realizar algunas modificaciones en las aplicaciones para que cumplan con estos requerimientos.

Los creadores, según afirman en su artículo [18], llevaron a cabo diversas pruebas y evaluaciones para medir el rendimiento de la aplicación. HyperFlow puede leer y deserializar 987 eventos por segundo, y escribir y serializar 233 eventos por segundo. Por lo tanto, a la luz de los resultados anteriores, HyperFlow puede garantizar una ventana de consistencia entre controladores siempre y cuando el número de eventos por segundo sea menor a 1000. Según sus creadores, la capacidad de HyperFlow puede ser mejorada modificando WheelFS. Aun así, no todo son ventajas. El retardo cada vez que la red debe converger a un estado nuevo va aumentando en cada iteración. Además, las modificaciones que requieren las aplicaciones para soportar HyperFlow no son compatibles con todos los plugins del controlador.

3.2. SDNi

Las SDN, previsiblemente se desplegarán en redes a gran escala, por lo tanto tendrá que haber varios dominios SDN interconectados entre sí, para mejorar la seguridad y la escalabilidad. Para utilizar los recursos de la red de forma eficiente, los controladores necesitarán comunicarse entre sí. Posibilitando este tipo de conexión aumenta el uso y la necesidad de incorporar más controladores. Todas las necesidades comentadas anteriormente llevaron al

desarrollo de la aplicación Software-Defined Network Interface (SDNi) por parte de la empresa Tata Consultancy Services, que desarrollo esta aplicación en 2012 como parte del proyecto Opendaylight.

3.2.1. Implementaciones

Hay dos implementaciones potenciales de SNDi: La implementación vertical y la implementación horizontal [4]. En las Figuras número 3.3 y 3.4 se explican a grandes rasgos ambas propuestas.

En la implementación vertical, existe un controlador SDN "master", localizado un nivel por encima del resto de controladores individuales. Este controlador tiene una visión general de toda la red a través de todos los dominios interconectados. Además, puede configurar el controlador de cada dominio SDN individualmente. Esta implementación provee de una mejor jerarquía a la red, sin embargo, en caso de fallo del controlador maestro, la red entera caería. Por este motivo, esta es la menos usada de las dos variantes disponibles.

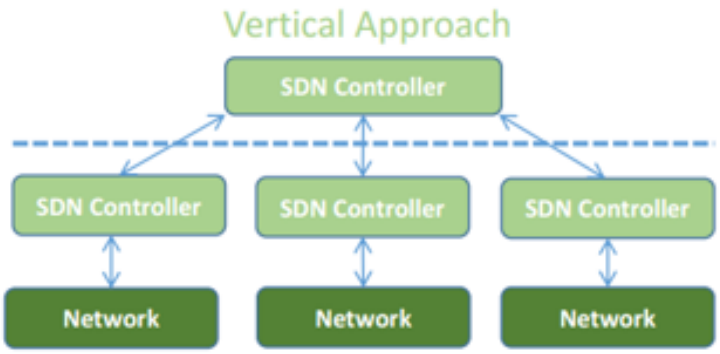

Figura 3.3: Implementación vertical SDNi. [4]

En la implementación horizontal, los controladores SDN establecen una comunicación *peer-to-peer*. Cada uno de estos controladores puede recibir información o peticiones de conexión de los controladores del resto de dominios SDN. Esta implementación también recibe el nombre de "*East-West Interface*". Posibilitando una comunicación controlador a controlador se consigue aumentar la resistencia a fallos de red, así como reducir el área a la que afecta dicho fallo. Por estos motivos, esta implementación es la más usada.

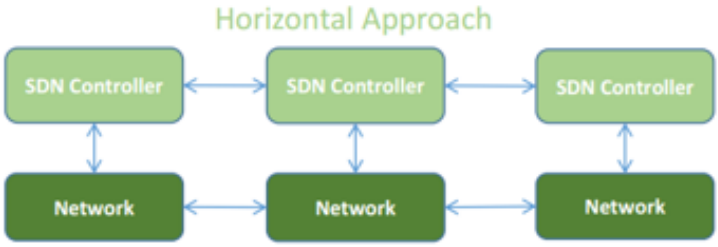

Figura 3.4: Implementación horizontal SDNi. [4]

3.2.2. Arquitectura

La arquitectura de la aplicación SDNi está integrada dentro de la propia estructura del controlador Opendaylight [4], utilizando varios de sus elementos, por ejemplo los *core network functions* (*statics manager*, *topology manager*...), para llevar a cabo su función. Los tres elementos que componen SDNi son: SDNi Aggregator, SDNi REST API y SDNi Wrapper [12]. En la Figura 3.5 puede verse como están integrados estos elementos dentro de la arquitectura del controlador Opendaylight.

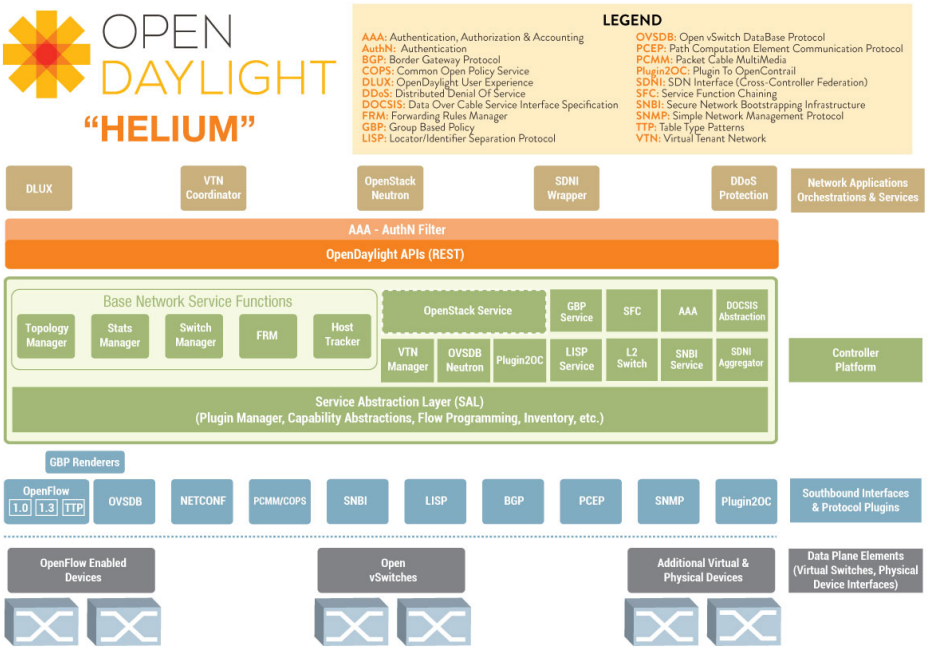

Figura 3.5: SDNi en Opendaylight. [12]

El **SDNi Aggregator** es un *plugin* "northbound" del controlador, es decir, la parte que interactúa con las aplicaciones integradas en dicho controlador, encargado de colectar todos los parámetros y características de la red, como por ejemplo la topología. Este *plugin* es ampliable según las necesidades de parámetros de red que tengan que ser compartidos.

La **RestAPI** se implementa para extraer la información agregada que proporciona el SDNi Aggregator. Esta Application Programming Interface (API) está en constante desarrollo para aumentar el número de parámetros que pueden ser soportados.

El **SDNi BGP Wrapper** es el elemento encargado de compartir y recolectar información intercambiada entre los controladores SDN conectados. Esto se hace a través del *plugin* Opendaylight-Boarder Gateway Protocol (ODL-BGP).

3.2.3. Ejemplos de uso

Según sus creadores, la aplicación SDNi puede ser usada en múltiples escenarios, resumidos a continuación:

- **Demanda de ancho de banda:** Cuando los recursos de la red están distribuidos en múltiples dominios SDN, el controlador de cada dominio necesita comunicarse con el resto para compartir los parámetros de la red. Esto posibilita que el controlador origen pueda confirmar y procesar el requerimiento de ancho de banda para la comunicación.
- **Content Delivery Network (CDN):** Los proveedores de servicio tienen que cumplir con los requisitos de entrega del contenido de acuerdo con la QoS acordada. Si el servidor CDN más cercano al cliente presenta alta carga y no puede servir la petición, esta es reenviada a otro servidor CDN que puede estar en otro dominio SDN. Por lo tanto, los distintos controladores de los dominios implicados tienen que comunicarse entre sí (usando SDNi), para poder crear un camino al otro servidor CDN.
- **DataCenters:** Si un cliente demanda un servicio de un Data Center, los controladores de acceso, distribución y *core* deben comunicarse entre sí para pasarse información como los parámetros de la red, información de calidad de servicio o incluso la misma política QoS utilizada.

Al ser una solución desarrollada para el proyecto del controlador Opendaylight, está en constante evolución. En concreto los desarrolladores actualmente están moviendo el código de SDNi desde el modelo API-Driven

System Abstraction Layer (AD-SAL) [11], que está empezando a quedar anticuado, al modelo Model-Driven System Abstraction Layer (MD-SAL) [8]. Hay diversos modos de establecer comunicación entre controladores SDN, pero al utilizar BGP se aprovecha el *plugin* que incluye el controlador, facilitando así de forma importante el desarrollo. Aun así, el protocolo Border Gateway Protocol (BGP) no es en tiempo real, por lo que el retardo y el tiempo de recuperación ante fallos es una desventaja a tener en cuenta.

3.3. CPRecovery

El principal objetivo de este componente según describen sus creadores Paulo Fonseca, Ricardo Bennesby, Edjard Mota y Alexandre Passito, en el artículo [16] consiste en el diseño de un mecanismo que permita incrementar la resistencia en la utilización de los componentes que forman una SDN, y su implementación e integración con un componente de *core* del sistema operativo del sistema. CPRecovery es un componente basado en la técnica "Primary-Backup", que permite fortalecer la red frente a fallos en un sistema centralizado y reducir el tiempo de transición entre el controlador que ha fallado y el de *backup*.

3.3.1. Procesos de replicación y recuperación

En esta sección, se describe el proceso de replicación entre el *switch* del controlador primario y del secundario. Además se describe el proceso de recuperación cuando el controlador primario falla. La primera fase se denomina fase de replicación, la segunda es la fase de recuperación.

En la fase de replicación el CPRecovery actúa durante la operación normal del sistema, es decir, la tabla de flujos de los *switches* se va completando conforme se envían peticiones y se reciben las respuestas oportunas, si un controlador con el *flag* "isPrimary" en true recibe una petición este la procesa, pero si el *flag* está en false (servidor de respaldo) la petición se ignora. El *switch* también manda un mensaje llamado "inactivity probe" si ve que el controlador ha entrado en un estado "ocioso" y espera durante un tiempo determinado. Si transcurrido ese tiempo el controlador no envía una respuesta, el *switch* asume que el controlador ha fallado, entrando así en la fase de recuperación. Durante este periodo ocurren las siguientes acciones:

- El *switch* busca al siguiente controlador en la lista e inicia una conexión con él.
- El controlador secundario recibe la petición, envía un mensaje "*data-path join event*" y cambia su estado interno a "*Primary*", el *flag* de

"*isPrimary*" debe ser puesto a true. Una vez se ha completado este proceso, el controlador pasa a gestionar el *switch*, convirtiéndose en primario. Si el antiguo controlador primario entrara de nuevo en funcionamiento, pasaría a ser secundario. En la Figura 3.6 se puede ver todo el intercambio de mensajes comentado anteriormente.

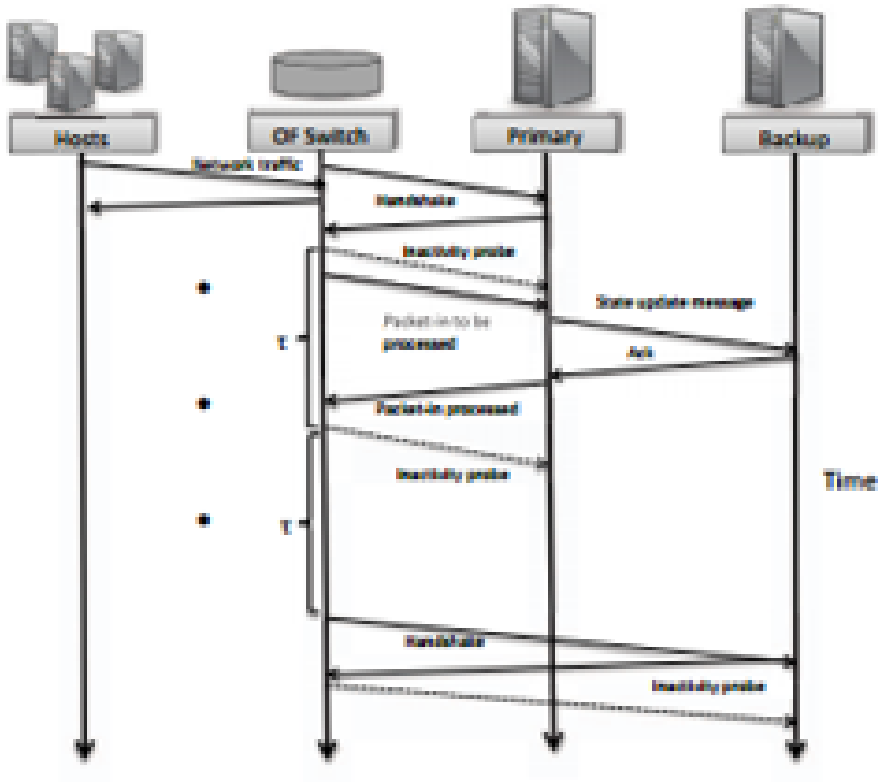

Figura 3.6: Funcionamiento CPRecovery. [16]

El componente CPRecovery ha sido probado y usado con Mininet. Para evaluar el rendimiento de este componente, sus creadores estudiaron el rendimiento de la red cuando el controlador primario sufría un fallo. Midieron parámetros como el tiempo de recuperación de la red, o el tiempo de variación de respuesta de las aplicaciones. Los resultados que obtuvieron fueron buenos, aun así, sí que había un cierto retardo desde que el controlador primario caía hasta que el secundario tomaba el control. Por otra parte, la mayoría de la información de la red (enlaces, direcciones IP, tabla de flujos...) se pierde al hacer este cambio, por lo que esto implica una mayor carga en la red para que el nuevo controlador primario adquiera toda esa información de nuevo.

3.3.2. Comparación entre implementaciones

Los tres apartados anteriores describen las implementaciones similares a lo que se pretende conseguir en este proyecto. El problema que aborda este proyecto es un tema que está todavía por desarrollar en plenitud, ya que aunque hay algunas propuestas, todavía no existe un "estándar" para llevar a cabo la federación de controladores SDN de forma completamente satisfactoria. Si bien la primera y la tercera solución (HyperFlow y CPRecovery) ofrecen, según sus creadores, muy buenos resultados, toda la información y bibliografía que se puede encontrar sobre ambas es el artículo en el que se definen, en los cuales solo se explica de forma general el funcionamiento de la solución propuesta. En cuanto a SDNi, esta solución sí que está implementada en las últimas versiones del controlador Opendaylight y disponible para descargar para todo aquel que lo precise, además dado a la importante comunidad de desarrollo que tiene detrás, está en constante evolución, suponiendo la principal alternativa para la solución que se propone en este proyecto. Aun así, la información sobre su funcionamiento es bastante escasa, lo que dificulta en gran manera su uso y comprensión por parte de los usuarios.

Capítulo 4

Planificación y Costes

Un aspecto esencial en el desarrollo de cualquier proyecto, un tema de vital importancia para que este se lleve a cabo de la mejor forma posible, es la planificación de todas y cada unas de las etapas que conforman el proyecto, como pueden ser las etapas de diseño, implementación, evaluación, etc.

Para la elaboración de una planificación correcta, una buena forma de hacerlo consiste en dividir todo el trabajo en tareas, e irlas realizando una a una. Hay un diagrama que permite representar la planificación de cualquier proyecto de forma muy acertada, el diagrama de Gantt. A continuación, en la Figura 4.1 se muestra la planificación que se ha seguido en este trabajo a través de uno de estos diagramas

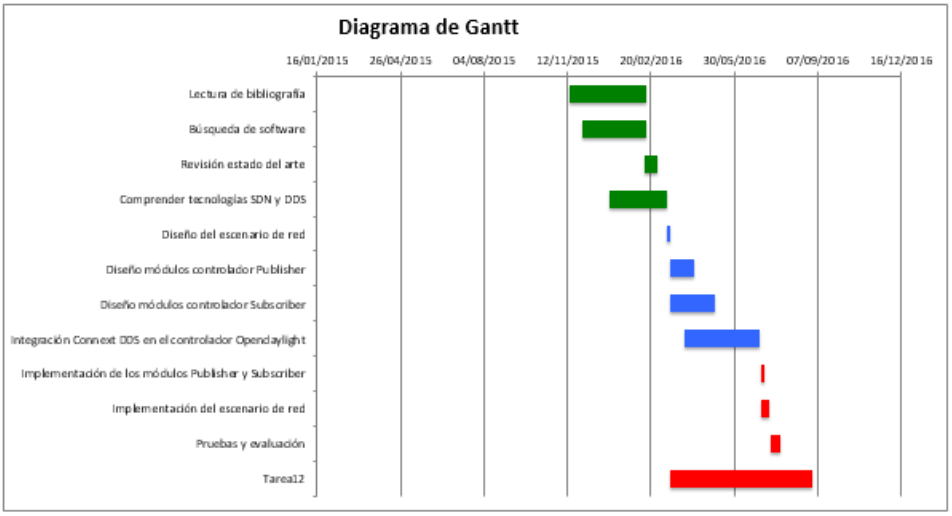

Figura 4.1: Planificación del proyecto.

Estudiando esta planificación, puede verse como este proyecto se ha di-

vidido en tres bloques principalmente: un primer bloque de investigación, tanto de bibliografía como de estado del arte, un segundo bloque relativo al diseño de la solución propuesta, y un tercer bloque en el cual se recoge la implementación de dicha solución, la evaluación realizada y la redacción de esta memoria.

Un aspecto importante a destacar es que la planificación aquí mostrada no es exactamente igual a la esbozada al principio de este trabajo. Esto es debido principalmente a los numerosos problemas que se han encontrado durante el diseño de la solución propuesta, pero sobre todo durante la implementación, por la importante falta de información, ya que los objetivos propuestos en en este trabajo no se habían logrado antes.

En cuanto al tema de costes, decir que no se ha tenido gastar nada ni en compra de equipos para la evaluación y pruebas, ya que todo se ha realizado mediante el uso de herramientas que permiten simular redes, ni en tema de licencias para todas las herramientas usadas, porque estas han sido dadas en el ámbito de investigación que supone este proyecto.

Capítulo 5

Diseño

A continuación, se establece el diseño de la solución propuesta dividido en dos partes: en primer lugar y de forma más resumida, se procede al diseño de la arquitectura sobre la cual se implementará la solución propuesta; en segundo lugar, se analizan con más detalle todos los módulos que componen la solución desarrollada para conseguir la federación de controladores SDN.

5.1. Diseño de la arquitectura

Tal y como se ha dicho anteriormente, la arquitectura de la solución escogida es de tipo publicación/suscripción, con una topología SDN subyacente. Uno de los controladores Opendaylight actúa como *Publisher* y el otro como *Subscriber*.

5.1.1. Publisher

En este apartado se describen las funciones que le han sido añadidas al controlador Opendaylight para hacer posible el envío de información al *Subscriber*, así como la creación de flujos para el enrutamiento de paquetes cuando se tiene una topología *multiswitch*.

Como se ha comentado en capítulos anteriores, el controlador Opendaylight ya incluye muchas funciones que no es necesario implementar de nuevo (como pueden ser las *network core functions*). Estas funciones o módulos facilitan el desarrollo de nuevas aplicaciones para el controlador, ya que cualquier aplicación que se desarrolle puede usarlas, sin más dificultad que importar los paquetes necesarios a la hora de programar la aplicación. A continuación se describen los módulos que forman parte de la solución que se ha diseñado en el lado del *Publisher*:

- **Creación de los flujos:** Cuando el controlador de la red SDN empieza a funcionar, envía a todos los *switches* que tiene directamente conectados unos mensajes Link Layer Discovery Protocol (LLDP), cuya función principal reside en construir una imagen de la red, es decir, conocer la topología de la red subyacente. Cuando a un *switch* directamente conectado al controlador llega un paquete para el cual el *switch* no tiene ningún flujo instalado, es decir, no sabe que hacer con ese paquete, este es reenviado al controlador. El controlador consulta sus propias tablas, si no existe una ruta para el paquete entrante la crea e inunda todos los enlaces con el paquete. Una vez hecho esto, el controlador devuelve el paquete al *switch* correspondiente y además instala el flujo en la tabla de enrutamiento de dicho *switch*. Esta última parte abarca una versión modificada de la aplicación llamada *learning-switch*.

- **Envío de datos mediante DDS:** En las funciones predeterminadas del controlador, no hay ninguna que permita enviar información o datos a otro controlador. Por este motivo, se ha diseñado un módulo, que basado en el estándar DDS y usando el software Connext DDS, permita la adecuación de la información para su envío, así como el control de todos los elementos (seguridad, QoS,formato de los datos...) que forman parte de cualquier comunicación basada en el modelo publicación/suscripción. La comunicación se estructura mediante el uso de *Topics*. En el diseño propuesto, se cuenta con un *Topic* para enviar datos relativos a las notificaciones de los paquetes entrantes,topología, flujos, etc. Este módulo es una de las partes más importantes de este proyecto, pues junto con el del *Subscriber*, es el que permite llevar a cabo la federación de ambos controladores, y el que habilita a los controladores para compartir una imagen común de todos los *switches* de la red y sus rutas.

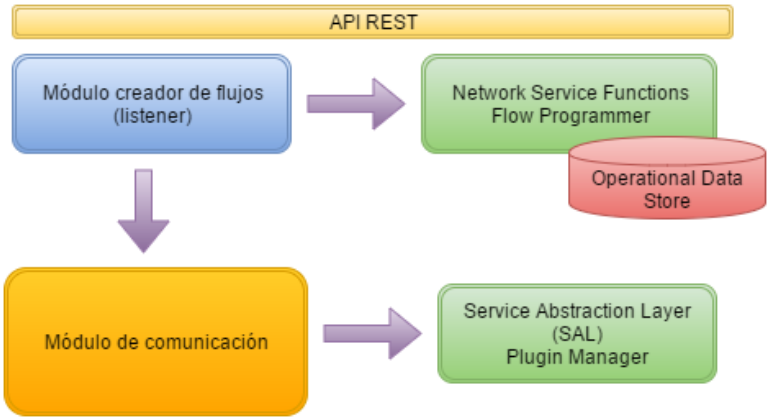

Figura 5.1: Esquema elementos Publisher.

En la Figura 5.1 se puede ver con que elementos del controlador Opendaylight debe interactuar cada módulo, así como las relaciones entre los dos módulos desarrollados. Los módulos verdes y rojo representan elementos del propio controlador Opendaylight. Destacar que el módulo de envío usa su propia librería, por lo que también usa unos métodos específicos.

5.1.2. Subscriber

De forma análoga al *Publisher*, en este apartado se describen de forma resumida las funciones añadidas al controlador Opendaylight que desempeña el rol de *Subscriber* en la comunicación, con el fin de hacer posible la recepción de la información enviada por el otro controlador y su uso en el funcionamiento de la red.

- **Recepción de datos mediante DDS:** Al igual que en el *Publisher*, se ha diseñado un módulo que permite la recepción de la información enviada por el otro controlador. Como se explicó en capítulos anteriores, cuando el software Connext DDS transmite la información, la deposita en un *buffer*, garantizando el orden de entrega de los paquetes. Este módulo se encarga de coger los datos de esta cola, para ello, es necesario implementar un *listener* que avise a la aplicación cuando haya datos disponibles en dicho *buffer*.
- **Creación de flujos:** Este módulo tiene una función similar al módulo creado en la parte del *Publisher*, pero con una ligera diferencia, y es que permite añadir a la tabla de flujo del controlador no solo los flujos de los switches directamente conectados, sino que también permite instalar flujos a partir de los datos recibidos por DDS. Para añadir estos flujos,

la información recibida debe ser "parseada."[a]l formato original de envío, de esto se encarga también este módulo.

- **Reconstrucción de la topología:** Entre los datos que se reciben por DDS, se encuentran los referidos a la topología, como pueden ser los nodos, los puertos de cada nodo (*NodeConnector o Termination-Point*), o incluso los *Links* entre varios nodos. La función principal de este módulo consisten en recoger los datos recibidos referentes a la topología e incluirlos en la *MD-SAL Data-Store*, permitiendo así que el controlador "secundario"pueda tener la misma imagen de la red que el controlador "primario".

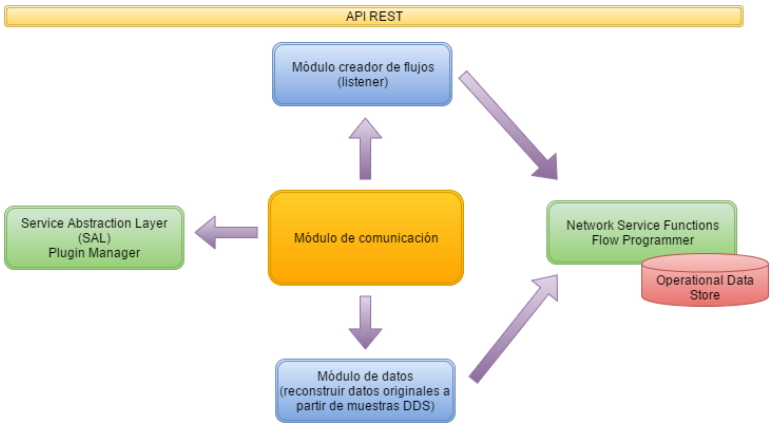

Figura 5.2: Esquema elementos Subscriber.

Al igual que se hizo en la parte del *Publisher*, la Figura 5.2 muestra como interactúan los módulos desarrollados con la arquitectura del controlador Opendaylight en la parte del *Subscriber*.Los módulos en verde y rojo representan elementos propios del controlador de Opendaylight, mientras que el resto son los módulos desarrollados para la solución propuesta.

Hay un aspecto importante a destacar y sobre el que se asienta la necesidad de implementar estos módulos, principalmente en el lado del *Subscriber*. Como se explica en el siguiente apartado, los principales switches de la topología creada están conectados a ambos controladores, pero sólo uno actúa como MASTER, mientras que el otro actúa como SLAVE, a la espera de un fallo del MASTER [10].

El controlador que actúa como MASTER (*Publisher*), tiene acceso completo al switch, es decir, puede modificarlo a su antojo ya que tiene permisos tanto de escritura como de lectura, los mensajes LLDP para descubrir la topología son respondidos y a él se le envían los mensajes con los paquetes

desconocidos para que cree una ruta. De la misma manera, puede enviar mensajes a los switches con la rutas o flujos a instalar

El controlador que actúa como SLAVE (*Subscriber*), tiene un acceso muy limitado al switch, y no recibe los mensajes asíncronos del mismo ni responde a los mensajes LLDP.Esto implica que aunque el controlador SLAVE este conectado al switch, no tiene ninguna información de los flujos que este posee, ni la topología de la red. En este escenario es donde los módulos desarrollados permiten que ambos controladores compartan la misma información de la red, es decir, que sean controladores espejo.

5.1.3. Topología SDN

Una vez que se ha descrito de forma resumida el diseño de los dos controladores que aglutinan la parte del plano de control de la red, así como los módulos más importantes que han sido diseñados para cada uno de ellos, es hora de analizar cual va a ser el diseño de la arquitectura de red SDN subyacente, la cual constituye la parte del plano de datos de la red.

La topología de una red representa la disposición de los enlaces que interconectan nodos de la red. La arquitectura de la red puede tomar muchas formas diferentes en función de como se conecten estos nodos entre sí. Existen principalmente dos formas para describir una topología: física o lógica. La topología física, normalmente suele referirse a la configuración de cables, antenas, computadores y otros dispositivos de red, mientras que la topología lógica hace referencia a un nivel más abstracto, considerando por ejemplo el método y flujo de la información transmitida entre nodos.

Para la realización de este proyecto se ha diseñado una topología en árbol. La topología en árbol puede verse como una combinación de varias topologías en estrella interconectadas entre sí, solo que no hay un nodo central. Sin embargo, si que existe un nodo de enlace troncal, a partir del cual se ramifican todos los demás. Por otra parte, cada uno de los nodos que se ramifican pueden tener *hosts* conectados.

La topología diseñada incluye un total de tres *switches*. El principal (S1) está directamente conectado con los dos controladores descritos anteriormente, siendo el único de la topología que tiene conexión con ambos controladores. Los otros dos *switches* (S2 y S3) cuelgan del principal. Los *hosts*, a su vez, están directamente conectados a estos dos *switches*. Más concretamente, en la topología se incluyen cuatro *hosts*, dos conectados al *switch* S2 y dos conectados al *switch* S3. Un aspecto importante a destacar es que aunque los enlaces entre *switches* son bidireccionales, cuando se analiza la topología, se ve como Mininet no crea un solo enlace que permita transmitir datos en ambos sentidos, sino que crea dos enlaces, cada uno de los cuales transmite la información en un único sentido.

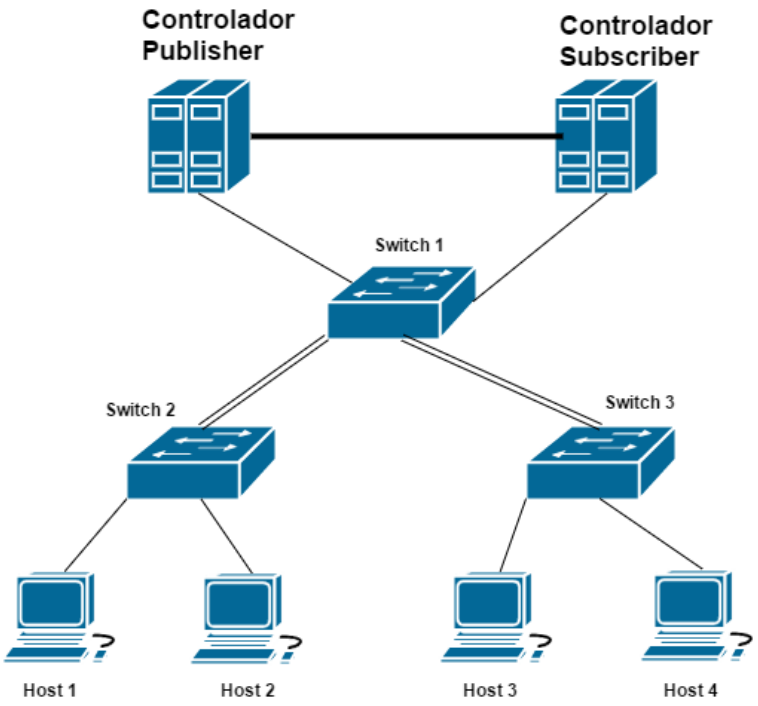

Figura 5.3: Esquema de la red hecha con Mininet.

Como puede verse en la Figura 5.3, aunque la topología no es excesivamente grande, es lo suficientemente amplia como para poder ver los resultados deseados y para que la ingente cantidad de información que se obtiene no dificulte la tarea de analizar los datos útiles.

Una vez que se ha descrito la arquitectura que tiene tanto la red como los controladores, es hora de describir de forma más precisa cada uno de los módulos que forman parte de cada uno de los controladores.

5.2. Módulos del controlador *Publisher*

En esta sección se describen en más profundidad el diseño de los módulos que forman parte del controlador que actúa como *Publisher*.

5.2.1. Módulo creador de flujos

En cualquier arquitectura de red, uno de los principales problemas a resolver es la forma en la que se realiza el enrutamiento y la transmisión de datos entre los distintos elementos que forman la red. En el caso de las SDN, este problema es aún más importante, ya que una de las premisas para este

tipo de redes es que buscan una clara separación entre el plano de control y el plano de datos. Ya que el elemento que aglutina la lógica de la red es el controlador Opendaylight, es en este donde se implementa el módulo que capacite al controlador para poder crear rutas e interconectar todos los elementos de la red.

Como ya se explicó anteriormente, cuando un *switch* recibe un paquete para el que no tiene ningún flujo instalado, reenvía este paquete al controlador. A continuación se describen todos los pasos que sigue el controlador con la lógica implementada para procesar el paquete:

1. **Captura del paquete:** En primer lugar, cuando el *switch* reenvía el paquete al controlador, este tiene que implementar un *listener* para capturar todos los paquetes entrantes. En el caso de la solución propuesta, este *listener* se implementa en el método *onPacketReceived()*. Una vez que *listener* captura el paquete, este método ejecuta la lógica correspondiente a los siguientes pasos.

2. **Extraer datos del paquete:** Para empezar a trabajar con el paquete es esencial saber que tipo de paquete es, el tamaño de su *payload* y las distintas cabeceras que incorpora. Para lograr esto, se usan algunas funciones del paquete *GenericUtils* (el cual debe ser importado como una dependencia tanto en la aplicación en sí, como en el controlador) que permiten extraer datos útiles del paquete como las direcciones MAC de origen y destino, así como el nodo y el puerto (*NodeConnector*) por el que se ha reenviado dicho paquete.

3. **Comprobar si ya existe una tabla para el nodo:** Cuando la topología subyacente a los controladores solo tiene un *switch* este paso se hace menos importante, ya que solo existe una tabla para ir almacenando todas la direcciones MAC de los paquetes recibidos y sus destinos. Sin embargo, cuando se tienen varios *switches* en la red, que suele ser el escenario más común, la capacidad de diferenciar a que *switch* pertenece el flujo que se va a crear es fundamental. Por ello, cuando el controlador recibe un paquete de un switch para el cual no existe esta tabla, la crea. La función de esta tabla es realizar un *mapping* entre las direcciones MAC origen y destino, así como entre los *Node* y *NodeConnector* para cuando se pase a construir el flujo, saber todos los datos relativos al destino. Para crear estas tablas se usan los *Maps* de Java.

4. **Comprobar si hay alguna entrada en la tabla para el paquete recibido:** Una vez que se comprueba que existe una tabla para el nodo que ha enviado el paquete, hay que comprobar si dicha tabla contiene alguna entrada que coincida con los datos del paquete. LLegado este punto, pueden ocurrir dos cosas; que no haya ninguna entrada en la

tabla que coincida con la MAC de origen o que sí la haya. Según cual de los dos casos se dé, el controlador debe realizar unas acciones u otras:

- **Si no hay coincidencia de la MAC:** Este caso se dará cuando el controlador reciba un paquete para el que no tiene información ninguna en la tabla del switch correspondiente,por lo que tendrá que crear una entrada nueva para asociar la MAC con algún nodo.
- **Si hay coincidencia de MAC:** En el caso de que una de las entradas de la tabla coincida con la dirección MAC del paquete entrante, el controlador sabe donde debe redirigir el paquete, por lo que puede crear un flujo e instalarlo en el *switch* correspondiente, y reenviar el paquete al destinatario correspondiente.

5. **Programar flujo:** Este paso solo se lleva a cabo en el caso de si haya coincidencia. Para construir el flujo se usa el método *programL2Flow()*, a este método es necesario pasarle tanto los identificadores del *Node* y los *NodeConnectors*, como la dirección MAC de destino. Esta función primero crea un objeto *flow* que contiene una lista de instrucciones y datos predeterminada según los parámetros de entrada. Después, se añade este objeto como parte del nodo en el inventario del *datastore* del controlador, para que quede constancia del flujo. Por último, el módulo *FlowProgrammer* de OpenflowPlugin (feature instalado por defecto en Opendaylight) detectará esta cambio y, eventualmente, será el encargado de programar el *switch*.

6. **Enviar paquete:** Según haya coincidencia o no en la tabla del switch, pueden ejecutarse dos métodos. Si hay coincidencia, se llama al método *packetOut()*, el cual se encarga de transmitir el paquete al nodo que se le indique en los parámetros de entrada. Si no hay coincidencia, el método que se ejecuta es el *floodingPacket()*. Es bastante similar al método que se encargaba de transmitir los paquetes en el caso anterior, solo que ahora, esto se repite para todos los nodos que el controlador conoce, inundando el paquete por toda la red.

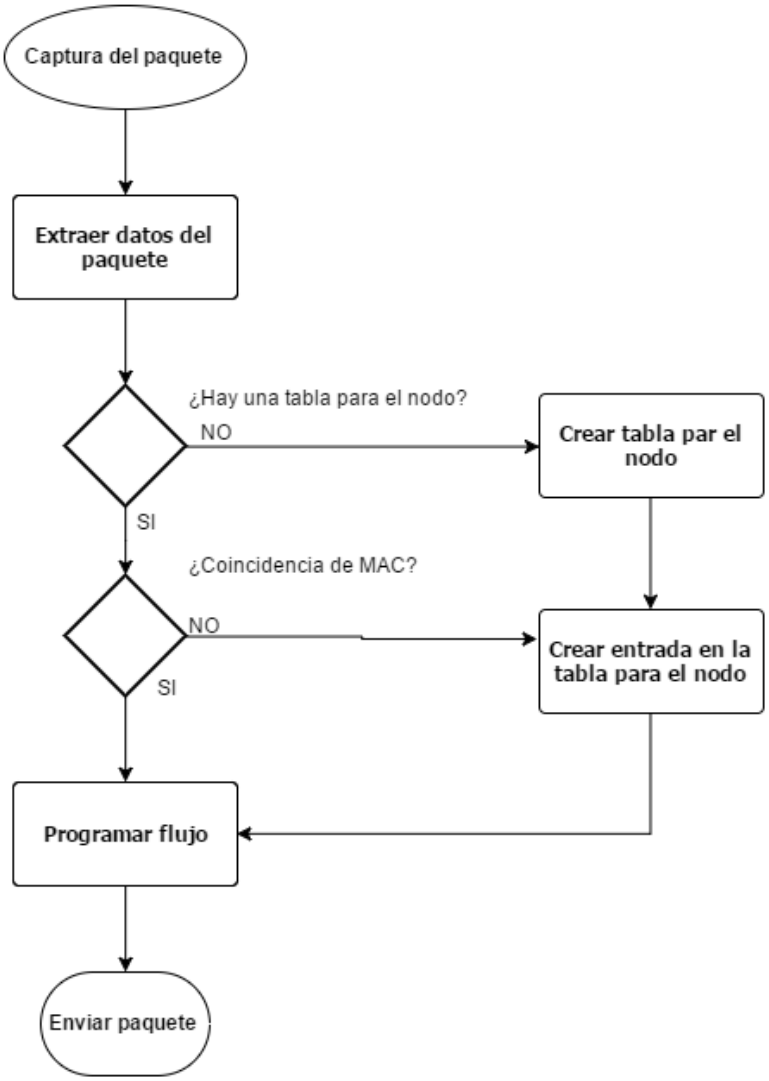

Figura 5.4: Diagrama de flujo del módulo creador de flujos.

En la Figura 5.4 puede verse mediante un diagrama de flujo, como es el diseño de este módulo y la forma en la que va avanzando su ejecución.

5.2.2. Módulo de comunicación

El objetivo principal de este proyecto consiste en la federación de controladores SDN mediante el uso del estándar DDS. Además también se busca proveer de redundancia y recuperación ante fallos a toda la red. Aunque el módulo anterior tiene gran importancia, ya que permite realizar el enrutamiento entre todos los elementos de la red, el módulo de comunicación,

implementado con el *software* RTI Connext DDS es el eje sobre el que se asienta el proyecto. Esto se debe a que las aplicaciones desarrolladas que permitan federar controladores son prácticamente inexistentes, por lo que todo el proceso y código desarrollado constituye la primera integración del middleware antes comentado con las SDN.

Al igual que el módulo anterior, este estará completamente integrado dentro de la arquitectura del controlador Opendaylight, lo que permitirá que el módulo interaccione con todos los elementos necesarios del mismo. A continuación se detallan todos los pasos se han seguido para el diseño y el funcionamiento de este módulo, el cual permite enviar información al otro controlador que participa de la comunicación:

1. **Descripción de la plantilla con el lenguaje IDL:** Antes de empezar a crear todos los elementos que participan en la comunicación, hay que crear una plantilla para que las muestras que se encarga de enviar el software RTI Connext DDS estén adaptadas a las necesidades requeridas. Por defecto el software solo puede enviar formatos simples como *Strings* o *Bytes*. Sin embargo, es posible crear un archivo IDL que permita enviar una combinación de ambos o formatos como *doubles* o *int*.

   En la solución propuesta, la plantilla está compuesta en su totalidad por *String*, ya que todo lo que necesitamos enviar se puede enviar con este formato, como los identificadores o las direcciones MAC. En la Figura 5.5 puede verse la descripción completa de este archivo, así como las variable que forman la plantilla.

```
1 /*Este archivo permite crear un formato para transmitir toda la informacion
2 de la topologia  y flujos de una red SDN, utilizando DDS*/
3
4 const long MAX_SIZE=64;
5
6 /*Identificadores de los distintos elementos de la topologia y de los flujos*/
7
8 struct topologia{
9         string<MAX_SIZE>                Identificador;
10        string<MAX_SIZE>                NodeId;
11        string<MAX_SIZE>                TerminationPointId;
12        string<MAX_SIZE>                LinkId;
13        string<MAX_SIZE>                SourceNode;
14        string<MAX_SIZE>                SourceNodeTp;
15        string<MAX_SIZE>                DestinationNode;
16        string<MAX_SIZE>                DestinationNodeTp;
17 };
```

Figura 5.5: Plantilla de las muestras DDS.

2. **Generar código con *rtiddsgen*:** Con la plantilla ya creada, el siguiente paso es generar todas las clases necesarias para que la aplicación pueda trabajar y enviar muestras DDS usando esta plantilla. Para realizar este procedimiento, RTI Connext DDS incorpora una herra-

mienta llamada *rtiddsgen*. Esta herramienta permite, a partir de un archivo IDL, e indicándole el lenguaje de programación de destino y la arquitectura del sistema operativo utilizado, generar todo el código y clases necesarias para poder crear instancias de esa plantilla. También construye las clases para crear *DataWritters* y *DataReaders* que permitan trabajar con estas instancias.

3. **Crear elementos DDS:** Habiendo realizado los dos pasos anteriores, ya es posible crear todos los elementos necesarios para que la parte de la comunicación del Publisher pueda realizarse. En primer lugar, en el constructor de la clase donde esta implementada toda la lógica de la aplicación, se crea el *DomainParticipant*, el *Topic* que implemente la plantilla creada anteriormente, y el *DataWritter*, que debe también implementar la anterior plantilla para poder publicar muestras de ese tipo.

4. **Instanciar las muestras**: Para poder transmitir datos mediante el estándar DDS, primero hay que definir la estructura de la muestra que se va a transmitir. Para ello, en la solución propuesta se instancia objetos del tipo *topología*, que permiten usar la plantilla anteriormente descrita. Cuando el objeto ya se ha creado, hay que ir rellenando sus campos según lo que se quiera transmitir:

   - **Flow:** Para el caso de que se quiera transmitir un flujo, es necesario indicar el identificador de la muestra, que en esta caso sera *Flow*, los identificadores del *Node* y *NodeConnector* por el que ha llegado el paquete y las direcciones MAC de origen y destino.
   - **NodeConnector:** Cuando se quiere transmitir un *NodeConnector* (también llamado *TerminationPoint*), el identificador de la muestra debe ser *TerminationPoint*, y deben transmitirse también los identificadores del *NodeConnector* y del *Node* al que pertenece.
   - **Node:** Para transmitir un nodo, el identificador de la muestra debe ser *Node*. Además, también se transmite el identificador de dicho nodo.
   - **Link:** Al transmitir un link entre dos nodos, o entre un nodo y el controlador, hay que rellenar la plantilla casi al completo. En identificador de la muestra será *Link*. También será necesario enviar los identicadores de los *Nodes* de origen y destino, así como los de los *NodeConnectors* de origen y destino.

5. **Enviar las muestras:** Este es el último paso para completar la parte de comunicación del *Publisher*. Lo único que queda por hacer es transmitir la muestra ya instanciada, para ello, se usa el método *write()* que incorpora el *DataWritter* anteriormente creado.

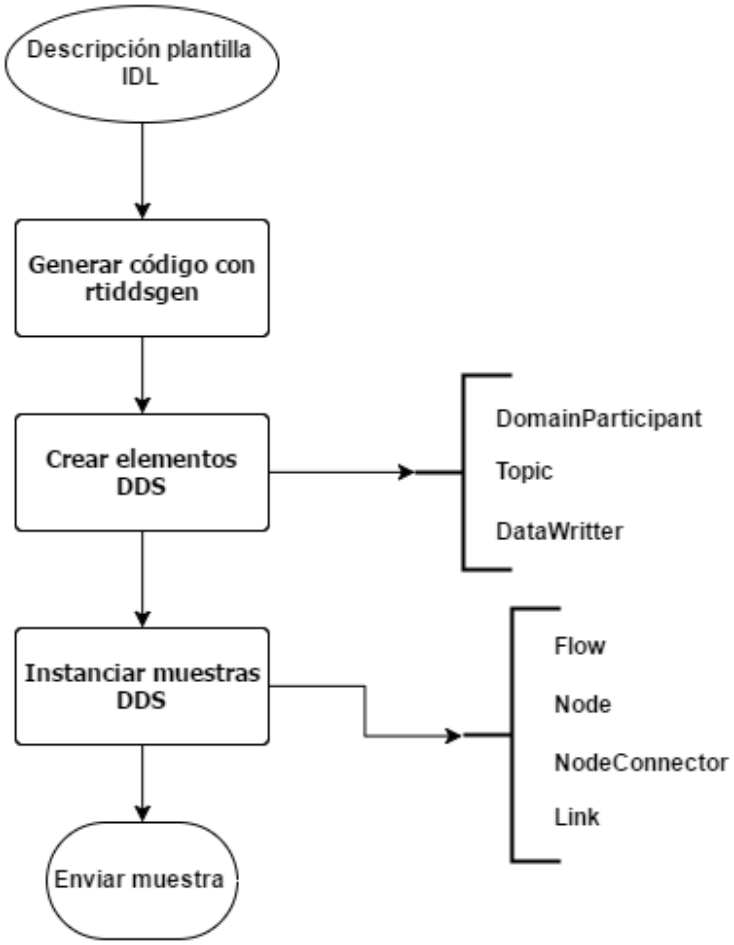

Figura 5.6: Diagrama de flujo del módulo de comunicación.

Para facilitar la comprensión de los pasos que se han seguido en el diseño de este módulo, en la Figura 5.6 puede verse todo esto de una forma más simplificada e intuitiva. Indicar también que los tres primeros pasos solo se realizan una vez, cuando se inicia la aplicación, ya que después no es necesario volver a crear, por ejemplo, la plantilla IDL.

5.3. Módulos del controlador *Subscriber*

Al igual que se ha hecho con el controlador que envía la información, en esta sección se describen los módulos que conforman la solución propuesta en el controlador que actúa como *Subscriber*.

5.3.1. Módulos creador de flujos

Este módulo, cuya finalidad es el enrutamiento de paquetes y el almacenar todas las rutas posibles que pueden seguir los paquetes de la red, no presenta ninguna diferencia con su módulo análogo en el controlador que desempeña la función de *Publisher*. Por lo tanto, al estar explicado en la sección anterior, no es necesario describir de nuevo todo el proceso de diseño. Recordar que el diseño del flujo de ejecución de este módulo puede verse en la Figura 5.4.

5.3.2. Módulo de comunicación

Una de las principales funciones que debe desempeñar el controlador *Subscriber* es la recoger la información, o mejor dicho, las muestras DDS, que el controlador *Publisher* envía. Esta funcionalidad queda implementada con este módulo. Cómo se verá un poco más adelante, el módulo de comunicación y el de reconstrucción de datos están fuertemente ligados. A continuación, se detallan todos los pasos que se han seguido para el diseño de este módulo:

1. **Descripción de la plantilla con el lenguaje IDL:** Antes de empezar a crear todos los elementos que participan en la comunicación, e igual que se hizo en el otro controlador, hay que crear una plantilla para las muestras que recibe el software RTI Connext DDS estén adaptadas a las necesidades requeridas. Esta plantilla debe ser exactamente igual a la anterior, ya que, de otra forma, la comunicación no es establecerá correctamente, imposibilitando así el intercambio de información.

2. **Generar código con *rtiddsgen*:** Este paso es exactamente igual al descrito en el otro controlador, por que no requiere de una explicación más profunda.

3. **Crear elementos DDS:** La parte de la comunicación del *Subscriber* tiene sus propios elementos DDS. En primer lugar, en el constructor de la clase donde esta implementada toda la lógica de la aplicación, se crea el *DomainParticipant*, el *Topic* que implemente la plantilla creada anteriormente, y el *DataReader*, que debe también implementar la plantilla IDL para poder recibir muestras de dicho tipo. Decir también, que al crear el *DataReader*, hay que indicar un *listener* que capture las muestras enviadas por el *Publisher*, esto se soluciona pasándole el constructor por defecto (el cual hay que crear, ya que hay varios constructores en la lógica implementada) como *listener*.

4. **Actuar cuando llega un paquete:** Con el *listener* implementado, el módulo ya es capaz de recibir paquetes. Cuando esto ocurre se ejecuta

el método que lleva asociado el *listener*. Este método es el *onDataAvailable()*. Lo primero que se hace al entrar en el método, es crear una instancia de la plantilla, es decir, una muestra en blanco. Después de esto, se lee la muestra recibida con una de las dos siguientes funciones: *read()*, que lee la muestra pero no la elimina del buffer, o *take()*, que lee la muestra y la elimina del buffer. En la solución propuesta se ha optado por la segunda función. Al leer la muestra lo que se hace es rellenar los campos de la muestra creada antes con los datos recibidos mediante DDS. Una vez hecho esto, es el turno del siguiente módulo, el módulo de reconstrucción de datos.

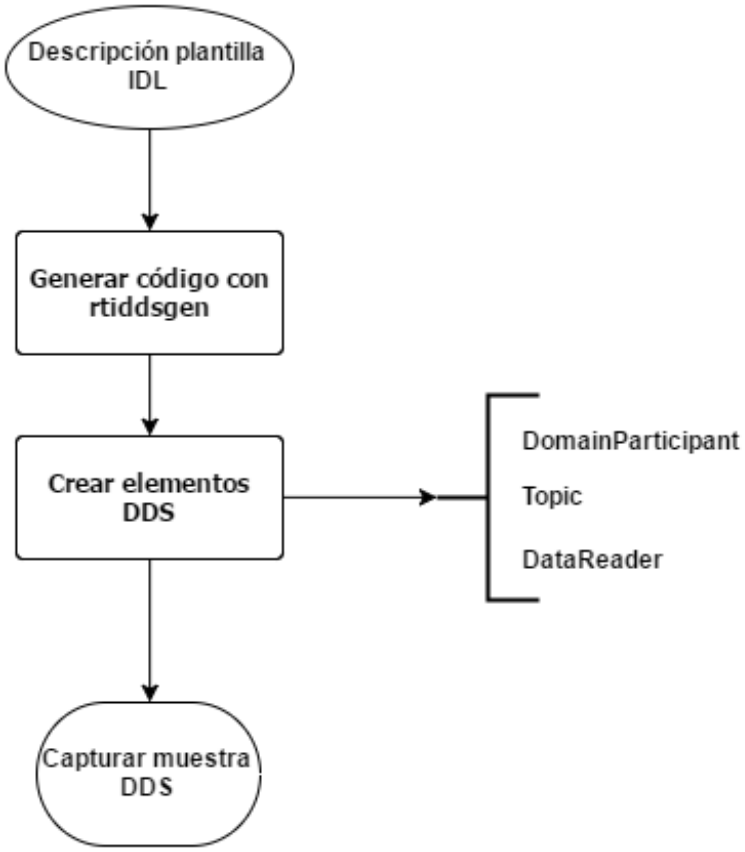

Figura 5.7: Diagrama de flujo del módulo de comunicación.

Como se ha hecho para el resto de módulos, en la Figura 5.7 se puede describe el flujo de ejecución de este módulo. Los dos primeros pasos solo es necesario hacerlos una vez, ya que se encargan de crear clases nuevas, mientras que el resto se ejecuta cada vez que llega una muestra nueva al controlador.

5.3.3. Módulo de datos

Con los módulos ya descritos, la federación de los dos controladores SDN ya es posible. Aun así, en la solución que se ha desarrollado, se incluyen más funciones aparte de la propia federación. Todas estas funciones están implementadas en este módulo, y permiten que el controlador que actúa como *Subscriber* tenga la misma información de la red que el controlador *Publisher*, en lo que a topología y flujos se refiere. Para ello, cuando ya se tiene la muestra recibida, se le realiza un análisis y procesamiento según el contenido de sus campos, una vez concluido el procesamiento, el elemento restante se incluye en el *data-store* del controlador. Este análisis, básicamente comprueba el identificador de la muestra que se está procesando, y se ejecuta una lógica u otra dependiendo de este identificador.

Antes de ver todos los casos posibles, es necesario definir el término *InstanceIdentifier*, ya que se usa en todos los casos y es un elemento importante a la hora de crear nuevos objetos. Todas los tipos de objetos que deben crearse para luego ser incluidos en la *data-store* del controlador (*Node, NodeConnector, Link...*) son objetos abstractos, es decir, no pueden ser definidos como el resto de objetos en programación. Un *InstanceIdentifier* no es sino una abstracción de estos objetos que permite manipularlos, ejecutar sus métodos e incluirlos en la *data-store*.

Así mismo, hay otro aspecto importante a la hora de escribir en la *data-store*, y es debe hacerse mediante el uso de un *DataBroker* que permita hacer la transacción entre el controlador y la base de datos. Una vez aclarado esto, es hora de describir los casos que pueden ocurrir al procesar la muestra:

- **Node:** En el caso de que tengamos un nodo, el procedimiento es bastante simple, ya que solo habrá que crear un *InstanceIdentifier* del *Node* que contenga el identificador del mismo, y posteriormente incluirlo en la *data-store* utilizando el método *writeData()*.

- **NodeConnector:** Para este caso, la forma de procesar es muy similar al anterior, la única diferencia es que a la hora de crear el *InstanceIdentifier* habrá que indicarle el identificador tanto del *NodeConnector* como del *Node* al que pertenece.

- **Link:** El caso del *Link* es el más complejo, ya que se trabaja con cinco identicadores. Hasta las últimas versiones del controlador, todavía no se ha implementado una forma de reconstruir el *Link* de forma que este pueda ser introducido en la *data-store*, por lo que aunque puede ser instanciado mediante un *InstanceIdentifier*, no se puede introducir de forma manual en la *data-store*. Esto no representa ningún tipo de problema, ya que los flujos no necesitan conocer ninguna información de los *links* para enrutar paquetes.

- **Flow:** Lo visto anteriormente se corresponde con las muestras recibidas que contienen información sobre la topología. Sin embargo, también es posible recibir información referente al enrutamiento de paquetes, con el fin de construir un nuevo flujo para incluirlo en la *data-store*. Para hacer esto, es necesario reutilizar una parte del código del módulo que se encargaba de crear los flujos. Esto se debe a que puede hacer falta crear tablas nuevas y aprender direcciones MAC de los switches de los que se tiene conocimiento, para reducir el tiempo de recuperación de la red frente a un fallo.

  Una vez se comprueba que no hay coincidencia de dirección MAC, se crea un *InstanceIdentifier* para instanciar el flujo, a fin de introducirlo también en la *data-store*.

Como se ha dicho antes, este módulo es vital para la redundancia y tolerancia a fallos de la red. Podría darse el caso, por ejemplo, de que el controlador primario sufre un fallo que lo deje inoperativo. El tiempo de recuperación de la red sería mínimo, ya que el controlador secundario dispone de todas las tablas y los flujos necesarios, además de toda la información de la topología para seguir enrutando paquetes de forma correcta.

Con este módulo concluye el análisis del diseño de la solución propuesta para la federación de controladores SDN mediante el uso del estándar DDS, además de un módulo extra que permite intercambiar información sobre la topología y los flujos de la red. Una vez hecho esto, se pasa a describir la implementación del proyecto, explicando todos los pasos que se han tenido que seguir para tener un entorno de desarrollo estable, así como la configuración necesaria para la ejecución del mismo.

Capítulo 6

Implementación

Con el diseño de la solución propuesta ya completamente descrito, el próximo paso en la elaboración del proyecto abarca la implementación del diseño explicado en el capítulo anterior. Esta fase se divide en dos partes; en la primera se explica todos los pasos necesarios para conseguir un entorno de desarrollo estable, lo cual ha supuesto una de las partes más complicadas del proyecto, y en la segunda se detalla la estructura de la aplicación desde el punto de vista de la programación, así como todo el proceso para ejecutar la solución.

6.1. Entorno de desarrollo estable

Antes de poder empezar a programar la solución propuesta, es necesario tener un entorno de desarrollo estable, a fin de evitar fallos en el sistema que complican la correcta ejecución tanto del controlador Opendaylight como del simulador de redes Mininet. Otra de las partes más importantes de este apartado, y que es de vital importancia para el proyecto, es que se describen como se ha llevado a cabo la integración del *software RTI Connext DDS* en la arquitectura del controlador Opendaylight. A continuación, se analizan todos los pasos que se han seguido para conseguir este entorno de desarrollo:

1. **Instalación de los programas necesarios:** En primer lugar, se han de instalar todos los programas que intervienen en el desarrollo y creación de la solución propuesta. En la máquina virtual que se ha utilizado como base para desarrollar el proyecto, programas como *Maven*, *Mininet* ó *Eclipse* ya vienen instalados de forma predeterminada de forma que no haya imcompatibilidades entre las versiones de todos ellos. Además, la máquina virtual cuanta con una distribución base del controlador Opendaylight. Esta máquina virtual puede encontrarse en [6]. El programa que sí es necesario instalar es el *RTI Connext*

DDS ya que este no viene instalado por defecto en la máquina virtual. Para ello, basta con entrar en la página oficial de la empresa *Real Time Innovations*, registrarse y descargar el archivo correspondiente al sistema operativo sobre el cual se va a usar el *software*.

2. **Generar arquetipo del proyecto con Maven:** Cuando ya se tienen todos los programas necesarios instalados, es hora de empezar a crear el proyecto Java en el cual se incluirá la solución propuesta. Este proyecto debe tener una estructura compatible con la estructura de las aplicaciones del controlador Opendaylight, para así evitar problemas a la hora de exportar el proyecto al controlador.

   Por lo tanto, hay dos maneras de crear este arquetipo, o bien se crean todas las carpetas y ficheros a mano, lo que supone una pérdida de tiempo inmensa, o se utiliza el programa *Maven* para generar un arquetipo de proyecto personalizado en cuestión de minutos. Obviamente, la opción seleccionada ha sido la segunda, aunque gran parte de los archivos generados por *Maven* no se usan en la mayoría de las aplicaciones. El comando que se usa para generar estos arquetipos es el que aparece en la Figura 6.1:

```
1 GENERAR PROYECTO CON MAVEN
2
3 mvn archetype:generate -DarchetypeGroupId=org.opendaylight.controller -DarchetypeArtifactId=opendaylight-startup-archetype \
4 -DarchetypeRepository=http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/ \
5 -DarchetypeCatalog=http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/archetype-catalog.xml \
6 -DarchetypeVersion=1.2.0-SNAPSHOT
7
```

Figura 6.1: Generar proyecto con Maven.

   La carpeta que más interesa dentro del proyecto, ya que será la que posteriormente se exportará al controlador en formato *.jar*, es la carpeta *impl*. Esta carpeta será el lugar donde se incluyen todas las clases necesarias para la elaboración de la solución que se ha propuesto.

   Dentro de la carpeta mencionada, se encuentra el archivo *pom.xml*. Este archivo es importantísimo para el correcto funcionamiento de la aplicación, ya que es en este archivo donde se especifican todas las dependencias (paquetes incluidos en los repositorios de *Maven* y *Opendaylight*) que debe cargar la aplicación, tanto en tiempo de compilación como en tiempo de ejecución. En este capítulo se describen los aspectos más importantes de este archivo.

3. **Añadir librería DDS en el repositorio de Maven:** Con la estructura del proyecto ya generada, el siguiente paso consiste en instalar la librería *nddsjava.jar* (que contiene todas las funciones necesarias para utilizar este software en programas como *Eclipse*) dentro del repositorio de *Maven*, ya que con este programa también se lleva a cabo la compilación. Hay un comando que permite instalar librerías externas en el repositorio de *Maven*, y puede verse en la Figura 6.2:

```
8 INSTALAR LIBRERIA EXTERNA EN EL REPOSITORIO DE MAVEN
9
10 mvn install:install-file -Dfile=C:\AKS\Tools\RTI\ndds.5.2.0\class\nddsjava.jar -DgroupId=com.rti.dds -DartifactId=nddsjava -Dversion=5.2.0
   -Dpackaging=jar
11
12
```

Figura 6.2: Instalar librería DDS.

Una vez hecho esto, se debe modificar el archivo *pom.xml* anteriormente comentado para que incluya esta dependencia al proyecto cuando se de la orden compilar. Esto se hace añadiendo unas lineas al fichero, que pueden verse en la Figura 6.4.

```
32   <dependency>
33       <groupId>com.rti.dds</groupId>
34       <artifactId>nddsjava</artifactId>
35       <version>5.2.0</version>
36   </dependency>
37 <dependency>
38       <groupId>org.omg.dds</groupId>
39       <artifactId>java5-psm</artifactId>
40       <version>1.0</version>
41   </dependency>
```

Figura 6.3: Dependencia DDS.

Como se va a explicar en el siguiente paso, el hecho de instalar la librería DDS en el repositorio de *Maven* no es solo útil a la hora de compilar el proyecto, sino también a la hora de ejecutar la aplicación dentro del controlador.

4. **Añadir librería DDS en el controlador Opendaylight:** Al igual que *Maven*, el controlador Opendaylight tiene si propio repositorio, en el que se recogen todas las dependencias necesarias para poder ejecutar las distintas aplicaciones que incluye el controlador. En este repositorio no es posible instalar librerías de terceros directamente, sin embargo, en la elaboración de este proyecto, se han desarrollado dos formas posibles de resolver este problema.

   - **Añadir librería de forma manual:** Esta opción en la más simple, ya que basta con copiar el fichero *nddsjava.jar* dentro de la carpeta *deploy* del controlador. Esta carpeta se encarga de cargar archivos externos en el controlador para que este los ejecute de forma automática al iniciarse. Aunque esta forma es la más sencilla, cada vez que se recompila el controlador es necesario copiar el archivo de nuevo.
   - **Añadir librería de forma automática:** Opción más compleja de desarrollar, pero que evita que cada vez que se recompila el

controlador haya que copiar el archivo de nuevo. Como se ha dicho anteriormente, Opendaylight tiene su propio repositorio, aun así, es posible enlazar dicho repositorio con el repositorio de *Maven*. Esto se hace con el fin de poder coger la librería desde este repositorio cada vez que la aplicación se ejecuta. Para poder llevar esto a cabo es necesario modificar de nuevo el fichero *pom.xml* añadiéndole el código que puede verse en la Figura 6.4.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>${bundle.plugin.version}</version>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Export-Package>
                com.rti.dds.nddsjava
            </Export-Package>
            <Import-Package>*</Import-Package>
            <Embed-Dependency>
                nddsjava;type=!pom;inline=false
            </Embed-Dependency>
            <Embed-Transitive>
                true
            </Embed-Transitive>
        </instructions>
    </configuration>
</plugin>
```

Figura 6.4: Librería DDS de forma automática.

A fin de optimizar el proceso de ejecución del proyecto, se ha optado por usar la segunda opción y realizar el proceso de forma automática.

5. **Compilar el proyecto:** Siguiendo todos los pasos anteriores se consigue un entorno estable para el desarrollo de las aplicaciones que posteriormente se cargarán en el controlador Opendaylight. El último paso que hay que realizar es compilar la aplicación que se desarrolle de forma que pueda ser procesada por el controlador, es decir, empaquetar con Maven la aplicación en un *.jar*. Hay otro comando que incorpora Maven que permite hacer esto:

   mvn clean install -nsu -DskipTests

   Al introducir este comando se le han añadido tres argumentos extra para garantizar que la compilación no falle. El argumento *clean* permite eliminar todos los archivos existentes creados por anteriores compilaciones. El argumento *-nsu* impide que se descarguen nuevas versiones de las dependencias instaladas, a fin de evitar que haya problemas de compatibilidad. Por último, el argumento *-DskipTests* provoca que a la hora de compilar se omita el proceso de testeo. En el caso que se

está estudiando este testeo es completamente innecesario, además de que aumenta en gran medida el tiempo de compilación.

Una vez se han visto todos los pasos necesarios para conseguir en entorno de desarrollo estable, es hora de explicar la estructura de clases que tiene la aplicación desarrollada, ya que todo el funcionamiento de la misma ya ha sido explicado en el capítulo de diseño.

6.2. Diagrama de clases

Como se ha comentado en el apartado anterior, la estructura del proyecto es automáticamente generada por *Maven* según los parámetros que se le indiquen. Recordar en este punto que son dos los proyectos que tiene que generar *Maven*, uno para el controlador que actúa como *Publisher* y otro para el controlador que actúa como *Subscriber*. Ambos proyectos cuentan con la misma estructura, encontrándose las diferencias entre ambos en el código de sus clases, por lo que se va a analizar solo la de uno de ellos. La organización de todas las clases y carpetas puede verse en la Figura 6.5.

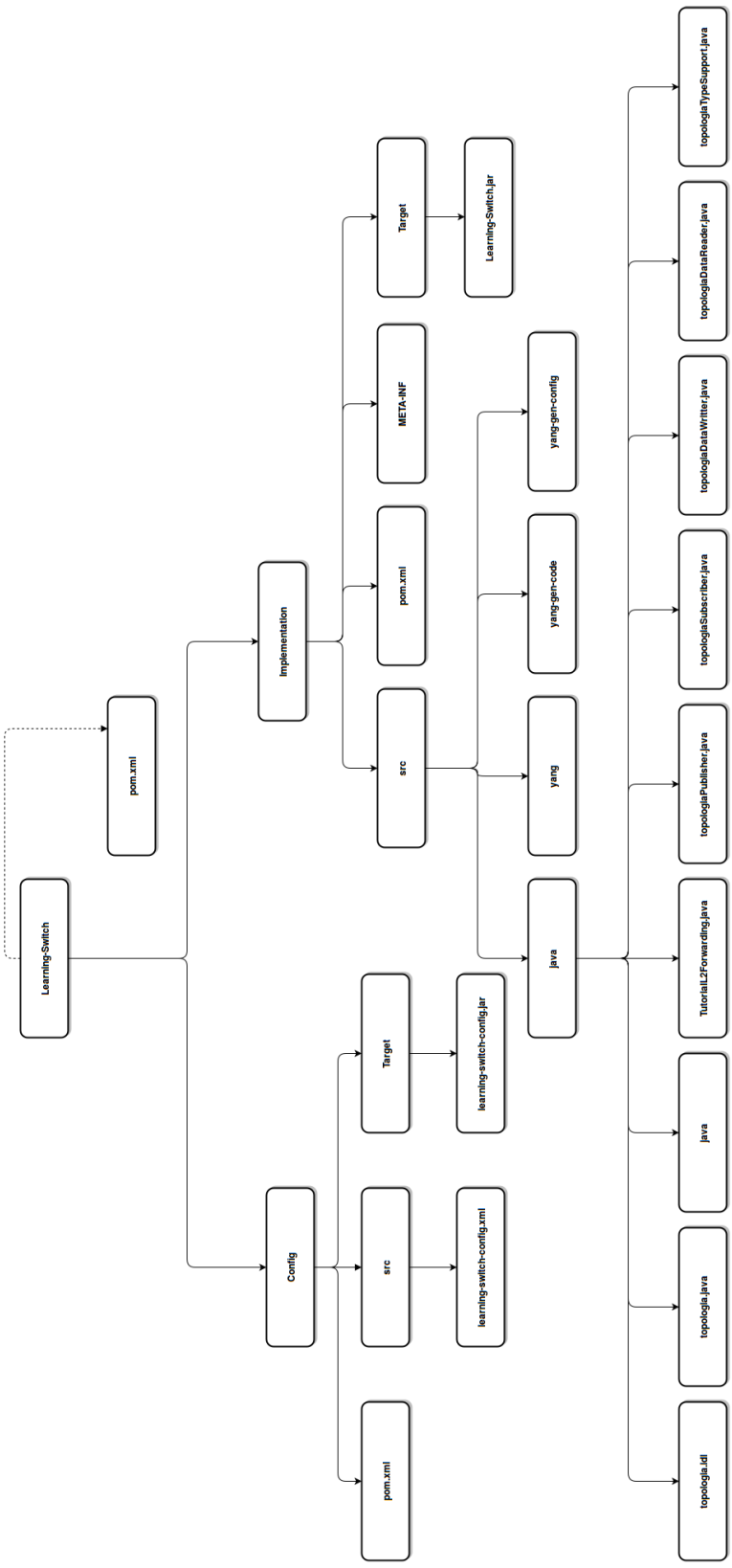

Figura 6.5: Estructura de ambos proyectos.

6.3. Ejecución de la solución propuesta

En este apartado se detalla todo el proceso a seguir para montar todo el escenario de la aplicación, así como para la ejecución de ambos controladores y la red simulada con Mininet. Para diferenciar las dos máquinas virtuales que componen el escenario sobre el que se realizan las pruebas, de aquí en adelante se van a utilizar las abreviaturas MV-Publisher (que corresponde con el controlador que actúa como *Publisher*) y MV-Subscriber (que se corresponde con el controlador que actúa como *Subscriber*). Esto se hace así para facilitar la comprensión del proceso de ejecución, ya que en este es necesario cambiar de máquina virtual en varias ocasiones. Es muy importante seguir el orden que se detalla a continuación para no obtener ningún error durante la ejecución:

1. **Compilar la aplicación de cada controlador:** El primer paso es compilar la aplicación desarrollada para cada uno de los controladores, es decir, tanto en *MV-Publisher* como en *MV-Subscriber*. Para ello se usa el comando indicado en el primer apartado de este capítulo. La primera vez que se compila cada uno de los proyectos, el tiempo de espera es largo, en torno a viente minutos, ya que todas las dependencias necesarias deben ser descargadas de los repositorios correspondientes.

2. **Compilar los controladores:** Para la correcta ejecución de la aplicación es recomendable recompilar el controlador cada vez que queramos construir una nueva topología. Este paso en obligatorio cuando la aplicación que se va a cargar en el controlador sufre alguna modificación. El método para compilar los controladores es el mismo que se seguía para compilar las aplicaciones y empaquetarlas en *bundles*, que es el nombre que reciben estas aplicaciones cuando han sido empaquetadas para su uso por el controlador.

3. **Ejecutar ambos controladores:** Con todo ya compilado, el siguiente paso es ejecutar los controladores de *MV-Publisher* y *MV-Subscriber*. Hay un aspecto importante a destacar durante este proceso. La aplicación que se ha desarrollado para cada controlador se ejecuta de forma autómatica cuando el controlador esta completamente iniciado porque así se ha programado, de igual forma que está programado que el controlador obtengan la librería DDS de forma automática como se indicó en apartados anteriores. Debido al elevado número de *bundles* y paquetes que tiene que cargar cada controlador al arrancar, la ejecución de la aplicación no es inmediata. Para estar seguros de que la aplicación se esta ejecutando hay que esperar hasta que en pantalla aparezcan unas líneas de texto relativas a la licencia del *software* RTI Connext DDS. Con todos los pasos anteriores se consigue la integración de RTI

Connext DDS dentro de ambos controladores Opendaylight, quedando así establecida la conexión para el futuro intercambio de información y muestras.

4. **Creación de la topología con Mininet:** Este paso debe hacerse en *MV-Publisher*, ya que el controlador alojado en esta máquina será el encargado de publicar las muestras DDS referentes a la topología y a los flujos necesarios para un correcto enrutamiento de paquetes. Como ya se comentó en el capítulo de diseño, la topología que se ha creado para este trabajo ha sido una topología en arból, con dos niveles de *switches* y cuatro *hosts*. Hay dos formas de crear topologías en Mininet; a partir de un script en Python (este caso suele ser más útil para topologías muy complejas) o mediante comandos en la terminal. El caso elegido ha sido segundo, ya que la topología que se maneja no es excesivamente compleja. Un hecho importante a destacar es que deben indicarse la dirección IP y el puerto en el que escuchan los dos controladores. Para el caso particular de este trabajo el comando necesario es el siguiente:

   sudo mn –topo tree,depth=2,fanout=2 –mac –arp –switch ovsk,protocols=OpenFlow13 –controller ip1:puerto1 ip2:puerto2

5. **Instalar flujos por defecto:** Como se ha comentado anteriormente, tanto la creación como la instalación de los flujos que permiten el correcto enrutamiento de los paquetes en la red SDN se realiza de forma automática por el controlador. Sin embargo, al iniciar la red, es necesario instalar de forma manual un flujo en cada uno de los *switches* que componen la topología. Este flujo es el que indica al *switch* que todos los paquetes para los que no tenga una regla instalada sean reenviados al controlador correspondiente. En el caso que se está estudiando, este controlador sería el alojado en *MV-Publisher*. Si se produjese un fallo, aunque el controlador de *MV-Subscriber* pasa a ser el controlador primario de forma automática, también sería necesario reinstalar estos flujos de forma que ahora redirijan los paquetes desconocidos a este controlador. Este paso también se lleva a cabo mediante comandos en la terminal de Mininet, en concreto hay que repetir el siguiente comando para cada *switch* de la topología:

   sudo ovs-ofctl add-flow -OOpenFlow13 s(i) priority=1, actions=output:controller

Una vez llegado este punto, la implementación del escenario para realizar las pruebas correspondientes ya está completa. Como ya se comentó anteriormente, esta parte del trabajo ha sido sin duda la de mayor dificultad, así como la que más porción del tiempo empleado en la realización del proyecto se ha llevado.

El siguiente capítulo describe la evaluación del diseño y la implementación descritos, mostrando el proceso a seguir para realizar las pruebas, así como los resultados tanto cualitativos como cuantitativos.

Capítulo 7

Evaluación

Una vez explicado en profundidad el diseño propuesto y descrito todos los detalles de la implementación realizada, en este capítulo se analiza la evaluación de la solución propuesta en el escenario creado para tal efecto.

Para ello en la primera parte del capítulo se realiza una evaluación a fin de obtener resultados cualitativos, es decir, resultados referentes a la federación del controladores SDN mediante el uso del estándar DDS. Y en la segunda parte del capítulo se detallan los resultados cuantitativos que se desprenden del resultado de implementar esta federación con múltiples controladores frente a escenarios en los que solo exista un controlador.

7.1. Evaluación cualitativa

Antes de empezar a desarrollar este apartado, y aunque ya se han mencionado con anterioridad en esta memoria, se va a describir el escenario sobre el que se van a realizar las pruebas y como estás se van a llevar a cabo.

El escenario sobre el que se ejecutan las pruebas consiste en un despliegue sencillo con 4 *hosts*, dos de ellos conectados directamente a un *switch* y los otros dos a otro. Estos dos *switches* a su vez están conectados a otro *switch* que se encuentra en un nivel superior. Para terminar, este *switch* está directamente conectado a los dos controladores existentes en el despliegue, el que actúa como *Publisher* (primario), y el que actúa como *Subscriber* (secundario). Al iniciarse la topología sólo el controlador primario tendrá acceso a todos los datos de la topología, y será el encargado de crear los flujos que deban ser instalados en los *switches*. Llegado cierto momento se provocará un fallo en el controlador primario, pasando así el controlador secundario a primario. Este paso lo lleva a cabo el *switch* que ha detectado el fallo enviando un mensaje OpenFlow a ambos controladores cambiando su estado de SLAVE a MASTER y viceversa. Si el funcionamiento es el correcto,

el controlador que antes era secundario debería tener toda la información necesaria de la topología y de los flujos instalados.

Sobre este escenario y partiendo de la base de que la topología ya esta creada y tiene ambos controladores correctamente conectados, lo primero que se comprueba, al igual que en el resto de redes, es que haya conectividad entre los nodos que forman el despliegue. Para ello el modo más sencillo es hacer *ping* entre dos nodos que se encuentren conectados a distintos *switches*. Pero antes de realizar este paso, se deben comprobar la base de datos de ambos controladores Opendaylight, a fin de ver que la base de datos del controlador primario está repleta de datos sobre los nodos representados en lenguaje XML, mientras que la del controlador secundario está completamente vacía. Para realizar esta comprobación basta utilizar la API REST que proprocionan los controladores Opendaylight a través de cualquier navegador. En las Figuras 7.1 y 7.2 se puede ver el estado de ambas bases de datos al inicializar la topología.

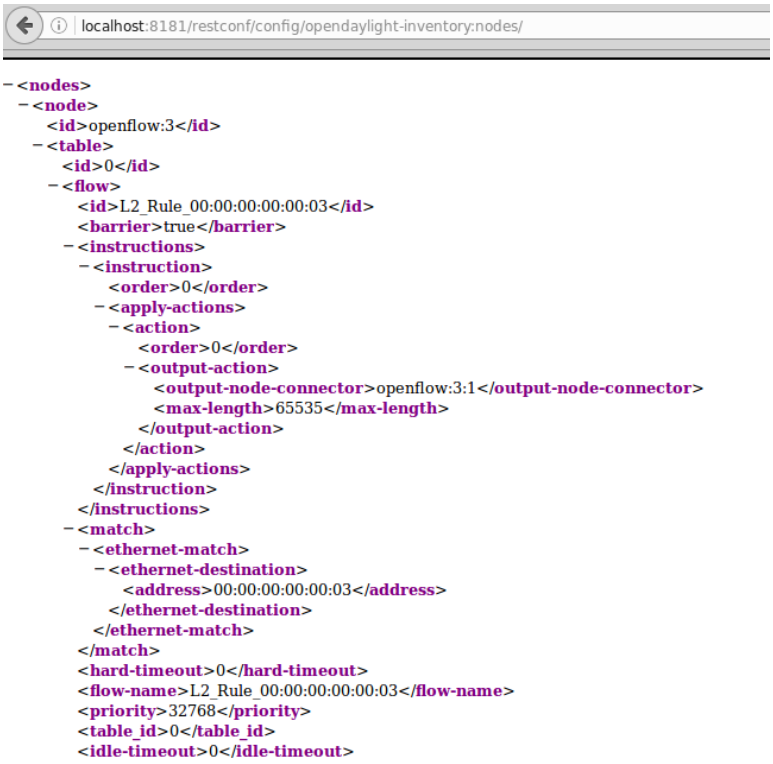

Figura 7.1: Base de datos del controlador *Publisher*

Figura 7.2: Base de datos del controlador *Subscriber*

Una vez comprobado el estado de las bases de datos, el siguiente paso, ahora sí, es hacer *ping* entre dos nodos. Como se ha comentado en capítulos anteriores de la memoria, la aplicación desarrollada esta hecha de forma que solo la primera vez que llegue un determinado tipo de paquete al controlador este programe un flujo para el *swtich*, mejorando así la ejecución del sistema. Dicho esto, al hacer *ping* se obtiene el resultado que puede verse en la Figura 7.3

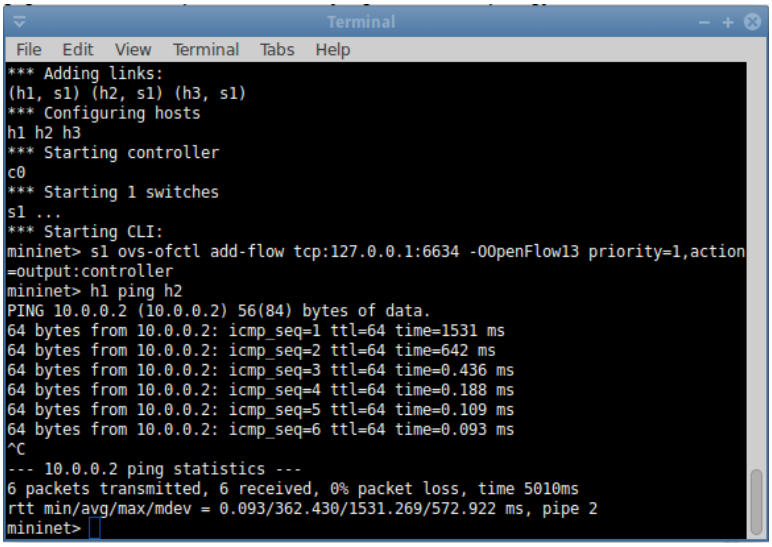

Figura 7.3: *Ping* entre dos nodos de la red.

Como se observa, los dos primeros paquetes ICMP tardan más tiempo en llegar al destino y ser respondidos, esto se debe a que para estos paquetes todavía no había un flujo creado en el *switch* correspondiente, por lo que el controlador es el encargado de enrutar los paquetes. Una vez que el flujo esta creado e instalado en el *switch*, el tiempo se reduce en gran medida.

Si el *ping* es exitoso significa que el flujo ha sido creado e instalado de forma correcta en el *switch* correspondiente, por lo que también se produce, según el diseño realizado, el envío de todos las muestras DDS con los datos de la topología y los flujos al controlador *Subscriber* (secundario). Por lo

tanto esto indica que si la implementación ha sido la correcta, en el controlador secundario deberían recibirse inmediatamente estas muestras DDS. Este hecho queda demostrado en la Figura 7.4 ya que la programación diseñada incluye la aparición de mensajes por pantalla cuando se reciben un tipo u otro de muestras.

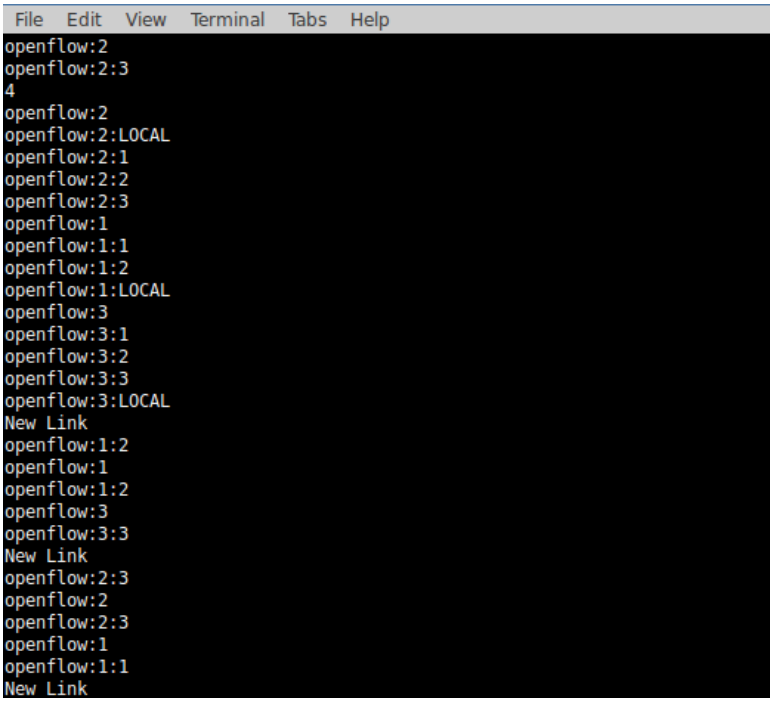

Figura 7.4: Resumen de las muestras DDS recibidas.

Una vez comprobado que las muestras se reciben de forma correcta, el siguiente paso es ver si la aplicación del controlador secundario extrae la información necesaria de estas muestras e instala los datos de forma correcta en la base de datos mencionada anteriormente. El método de comprobación más sencillo para esto es volver a entrar a la base de datos y ver si su contenido ha sido actualizado. La Figura 7.5 muestra el resultado de dicha comprobación, mostrando tanto datos pertenecientes a la topología, como una descripción de uno de los flujos alojados en la *data-store* del controlador.

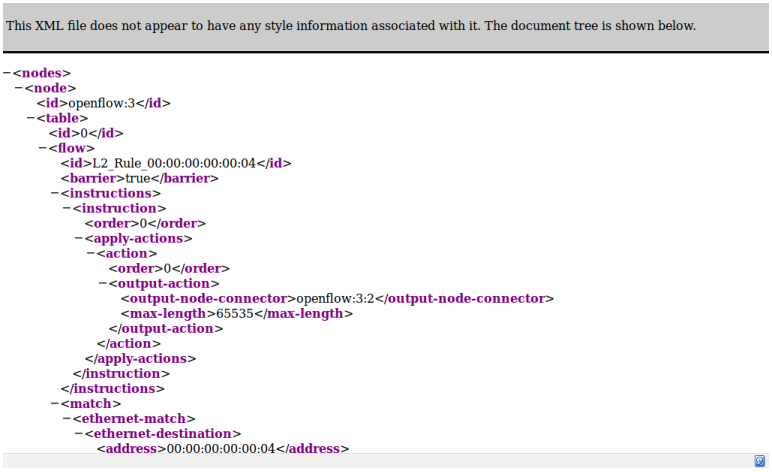

Figura 7.5: *Operational Data-Store después de recibir las muestras DDS.*

La última prueba de esta evaluación cualitativa, una vez comprobado todo lo anterior, consiste en provocar un fallo intencionado en el controlador primario, para que el controlador secundario pase a ser primario. En este caso se ha optado por eliminar el enlace que unía el *switch* principal con el controlador primario. Una vez hecho esto, el cambio al otro controlador es inmediato, por lo que debería seguir habiendo conectividad entre los nodos para los que ya había flujos instalados y para los que no. Este hecho puede comprobarse en la Figura 7.6.

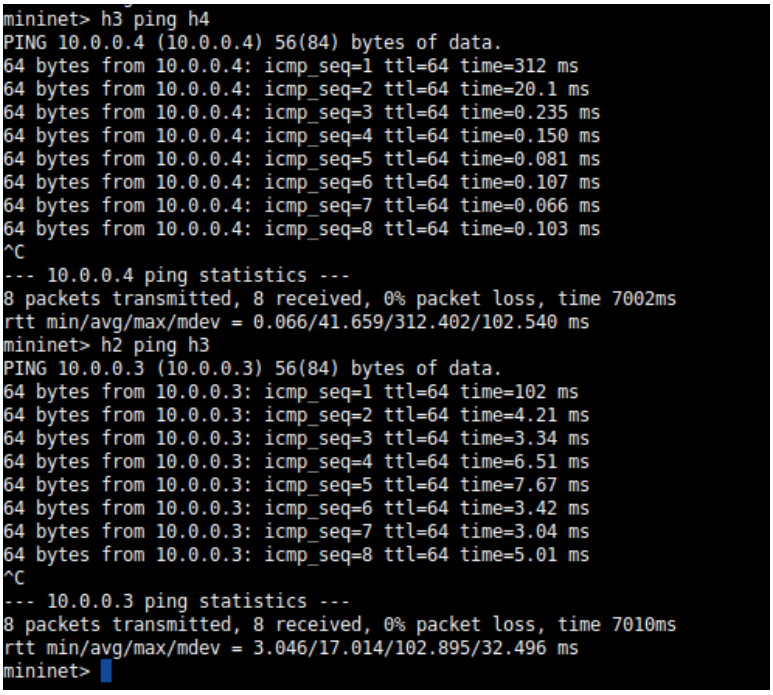

Figura 7.6: Ping despues de conectar con el controlador *Subscriber*.

Con los resultados cualitativos ya analizados, es hora de analizar los resultados cuantitativos, es decir, cuanto más bueno es este sistema con respecto a usar un solo controlador en el despliegue de la topología.

7.2. Evaluación cuantitativa

En el mundo actual, y más en el mundo de la tecnología, cuando se quiere reforzar la creencia de que la solución que se está ofreciendo es la mejor, se suelen aportar datos y estudios que avalen dicha solución. En el ámbito de las redes, los números que suelen estudiarse a fin de ver si una solución es mejor que otra son el retardo, el tiempo de recuperación frente a fallos, la cantidad de tráfico generado al implementar la solución, etc.

Aunque en este trabajo, como se ha visto en el apartado anterior, con los resultados obtenidos de la evaluación cualitativa, los objetivos que se propusieron en los inicios de este trabajo quedan más que logrados, sin embargo, a fin de reforzar la idea de que la federación de controladores utilizando el estándar DDS es el mejor escenario en la implementación de las SDN, se han realizado algunas pruebas, como medir el tiempo de recuperación frente a fallos y la cantidad de tráfico generada para recuperar el estado de la red sobre tres escenarios que quedan descritos a continuación:

- **Escenario 1.** Este escenario es el más simple, ya que en el despliegue solo se usa un controlador Opendaylight. Cuando se produce un fallo en el mismo controlador, este se resetea completamente.
- **Escenario 2.** Escenario un poco más completo, en el cual ya existen dos controladores en el despliegue de la red. Sin embargo estos controladores no están federados ni comparten ninguna información. Cuando el controlador primario falla, el secundario toma automáticamente el control de la red, pero no dispone de ninguna clase de información de la misma, por lo que requiere cierto tiempo el recuperar el estado que se tenía antes del fallo del controlador primario.
- **Escenario 3.** Este es el escenario sobre el que se ha trabajado durante todo el proyecto. Despliegue con dos controladores federados entre sí, que comparten la misma imagen de la red, permitiendo que el cambio de un controlador a otro sea automático sin afectar prácticamente nada al correcto funcionamiento de la red.

El primer aspecto que se analiza es el tiempo de recuperación frente a un fallo del controlador, hasta que la red vuelva a estar completamente operativa de nuevo. Obviamente estos tiempos son aproximados y dependen de la potencia del ordenador usado como controlador, pero en este caso, tal y como se muestra en la Figura 7.7 la diferencia es bastante clara.

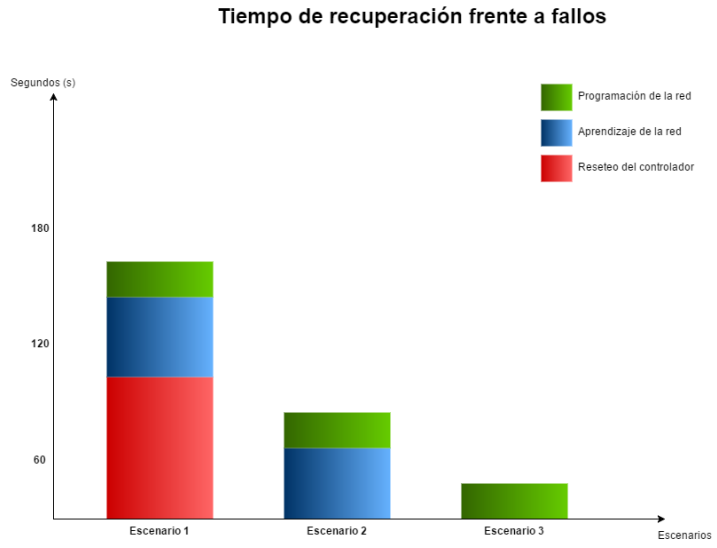

Figura 7.7: Gráfica del tiempo de recuperación.

Analizando brevemente la Figura 7.7, se puede ver como el Escenario 1 es el menos eficaz y más lento, ya que conlleva el total reseteo del controlador, el

nuevo aprendizaje de la red, y la obligada configuración para el enrutamiento de paquetes. El Escenario 2 es un poco más rápido, ya que elimina la parte del reseteo. Por último, el Escenario 3 es el más eficaz puesto que solo hay que ajustar la configuración para el enrutamiento de paquetes.

Otro aspecto importante a analizar, es la cantidad de tráfico necesario y los mensajes que tienen que ser intercambiados para volver a un estado óptimo de funcionamiento cuando se produce un fallo en la red. En la Figura 7.8 se puede ver un análisis de todos estos mensajes generados durante el tiempo de inactividad en cada uno de los escenarios propuestos, con el fin de volver a estar completamente operativos.

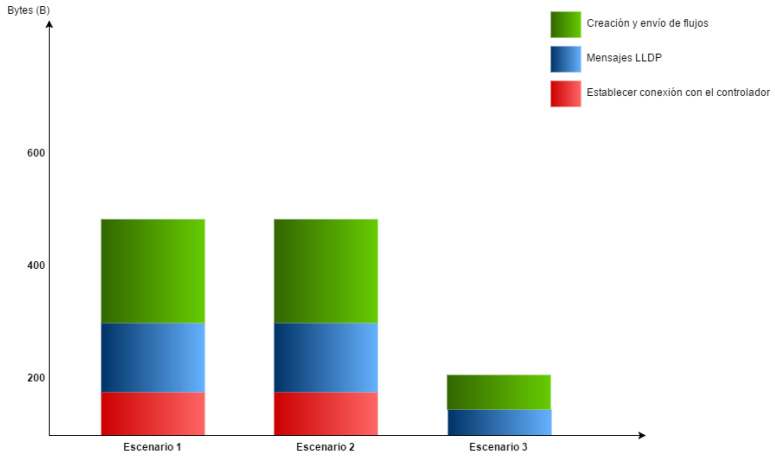

Figura 7.8: Gráfica del tráfico generado durante la recuperación.

Se ve claramente como en el caso de un solo controlador (Escenario 1), la cantidad de mensajes intercambiados, y por lo tanto el tráfico generado es mayor. En este escenario, tras un fallo, se debe volver a establecer la conexión, descubrir de nuevo la topología de toda la red y crear todas las tablas de los *switches* en el controlador.

En el Escenario 2, los mensajes necesarios son los mismos, la única ventaja con el Escenario 1 es que no hay que esperar a que el controlador se resetea para volver a establecer la conexión.

El Escenario 3 es el que más ventajas ofrece, ya que aunque la conexión se debe establecer con los dos controladores, en caso de fallo de la red los mensajes LLDP para reconstruir la topología no deben volver a enviarse, al igual que los flujos no deben ser creados de nuevo, puesto que ambos controladores comparten la misma información.

Con el análisis realizado de todos los aspectos que resultan fundamentales para el funcionamiento de la red, queda ampliamente demostrado que el

escenario elegido para la implementación de la solución propuesta es el más acertado de los tres que se han planteado. Esto demuestra por lo tanto, que la federación de controladores SDN mediante DDS permite, aparte de mejorar la escalabilidad y fiabilidad de la red, reducir de manera drástica el tiempo de inoperatividad cuando se produce un fallo en el controlador primario de la misma.

El siguiente capítulo resume algunas de las ideas que quedarían para trabajo futuro, con el fin de mejorar las prestaciones de los controladores y de la misma red, así como las conclusiones del desarrollo de este proyecto.

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

El uso de las *Sofware-Defined Networks* como tecnología de *core* en las redes actuales, es un paradigma que está ganando mucha fuerza durante los últimos años, ya que la arquitectura de red más utilizada actualmente sufre ya problemas de escalabilidad y sobrecarga de tráfico, sin mencionar la inminente llegada del tráfico procedente del IoT. Por este motivo, además de las múltiples ventajas que ofrecen las SDN frente a las redes convecionales, como por ejemplo la separación del plano de control y del plano de datos o la reducción del tiempo para el enrutamiento de paquetes, colocan a las SDN como principal modelo de red en un futuro cercano.

En este proyecto se ha perseguido la realización de una labor de investigación y desarrollo de un método de federación de controladores SDN mediante el uso del estándar DDS, más concretamente, mediante el uso del *software* RTI Connext DDS. Esto permite, en otras cosas, el intercambio de información entre varios controladores Opendaylight en tiempo real, siendo útil para mejorar la escalabilidad y resistencia a fallos de la red.

En la primera parte de esta proyecto, se ha llevado a cabo una descripción de las limitaciones de las redes actuales y como las SDN pueden resolver estos problemas. También se ha explicado en profundidad el funcionamiento de las *Software-Defined Networks* y el protocolo sobre el que se apoyan para gestionar la red, el protocolo OpenFlow, además del funcionamiento del estándar DDS y el *software* utilizado para implementarlo. Otro de los elementos de la red, vital para el desarrollo de la solución propuesta, ha sido el controlador Opendaylight, el cual aglutina toda la lógica de la red y es el encargado de garantizar el correcto funcionamiento de la misma junto con el

protocolo comentado anteriormente. Por último, también se han desarrollado las alternativas existentes a la federación de controladores SDN, pero como se vio en el capítulo correspondiente (Capítulo 3) estas alternativas son escasas y la mayoría son publicaciones en revistas científicas de las cuales no se puede obtener apenas información.

Al ser esta parte del proyecto de investigación, ha supuesto muchas horas de lectura de bibliografía, así como de otros proyectos relacionados que han permitido sacar toda la información necesaria. Llegado este punto es imporante decir que en la parte teórica del proyecto, la información es abundante y fácil de conseguir. Todo lo contrario ha ocurrido en la parte que se expone a continuación, la parte práctica. Ha sido un auténtico calvario la búsqueda de simples tutoriales o ejemplos para el desarrollo de aplicaciones para el controlador Opendaylight, además de la inexistencias de foros en los que consultar los numerosos problemas que han ido surgiendo durante todo el proyecto. Esto ha llevado al proyecto a un punto de dificultad por encima de lo previsto en un principio, ya que las etapas de diseño e implementación han consistido en múltiples iteraciones de ensayo y error hasta conseguir el principal logro de este proyecto; la integración del *software* RTI Connext DDS, que implementa el estándar DDS, dentro del controlador Opendaylight. Esto, según toda la bibliografía consultada, no se había realizado antes, por lo que ha supuesto el eje sobre el que ha girado la construcción de este proyecto.

En la segunda parte del proyecto se ha llevado a cabo el diseño e implementación de la solución propuesta para satisfacer los objetivos planteados al principio de la memoria.

En el Capítulo 5, se ha diseñado un escenario de red formado por varios *switches* y dos controladores. Para cada uno de estos controladores se ha diseñado también una aplicación para ser integrada en su arquitectura. Esta aplicación permite, por una parte, el correcto enrutamiento de paquetes entre todos los equipos de la red, y por otra parte, el intercambio de información sobre la topología de la red o los flujos creados para el enrutamiento de paquetes anteriormente comentado.

En el Capítulo 6, se ha llevado a cabo la implementación del diseño comentado antes sobre una red simulada con el programa Mininet. Solo con que ambos controladores consiguieran establecer conexión entre sí e intercambiar información ya se cumplen la mayoría de los objetivos básicos propuestos para este proyecto. Sin embargo, en el Capítulo 7 se ha llevado a cabo una evaluación más exhaustiva. En esta evaluación se ha demostrado como ambos controladores intercambian la información necesaria para que ambos mantengan la imagen de la red en tiempo real, de ahí el uso del estándar DDS. Además, también se ha llevado una comparativa entre posibles escenarios de implementación en el despliegue de redes SDN, demostrando que el esquema desarrollado es el más rápido y eficaz para reducir el tiempo de

recuperación frente a fallos de la red, así como para solucionar problemas de escalabilidad en la misma.

Para terminar esta sección,se va a hacer un análisis de la consecución de los objetivos propuestos en el Capítulo 1 de esta memoria.

Objetivo. Estudiar el problema de la federación de controladores SDN para que comparten una imagen única y común de un conjunto de *switches*.

- Se ha diseñado un escenario en el cual se despliega una red SDN con dos controladores Opendaylight. Estos controladores comparten y manejan los mismos datos de la topología de la red, además de todos los flujos necesarios para el enrutamiento de paquetes en dicha red.

Objetivo. Conocer y comprender la tecnología SDN y DDS.

- Se ha llevado a cabo una profunda descripción de ambas tecnologías en el Capítulo 2, así como su aplicación y método de uso en este proyecto.

Objetivo. Revisión de estado del arte en cuanto a implementaciones.

- El Capítulo 3 recoge algunas de las implementaciones más relevantes relacionadas con el tema desarrollado y la solución propuesta.

Objetivo. Instalación y pruebas de la implementación de un escenario con varios switches SDN controlados por varias instancias colaborativas interconectadas mediante DDS.

- Se ha implementado un prototipo de solución de federación de controladores mediante la tecnología DDS, en la cual ambos controladores comparten exactamente la misma información sobre los *switches* de la topología desplegada.

Objetivo. Pruebas y evaluación de la implementación realizada.

- Se ha llevado a cabo una evaluación tanto cualitativa como cuantitativa de la implementación realizada.

8.2. Trabajo Futuro

Como trabajo futuro a este proyecto, se proponen dos líneas de investigación. Una línea de investigación consiste en aumentar la complejidad y estructura de la información intercambiada entre controladores. La otra línea de investigación esta relacionada con el desarrollo de un elemento de supervisión de toda la red, que implemente también DDS.

La primera línea de investigación hace referencia a la posibilidad de aumentar la cantidad de información que los controladores de la red comparten, es decir, que además de datos referentes a topología y flujos, compartan información sobre las estadísticas de la red o aspectos de calidad de servicio, así como políticas de seguridad. Esto permitiría un mejor funcionamiento de toda la red.

La segunda línea de investigación va orientada al desarrollo de un sistema de monitorización, similares a los existentes en las redes actuales, que permita a una persona monitorizar el estado de redes SDN mucho más complejas que las aquí implementadas. Para ello, por supuesto, este sistema debe implementar el estándar DDS, para tener acceso a la información compartida entre controladores.

Bibliografía

[1] Aumento del tráfico cloud. http://www.panoramaaudiovisual.com/2014/11/05/el-cloud-supondra-el-76-por-ciento-de-todo-el-trafico-data-center-global-en-2018/.

[2] Aumento del tráfico ip 2013-2018. http://www.tendencias21.net/El-trafico-de-Internet-se-triplicara-en-cuatro-anos_a34636.html.

[3] Informe hp. http://h17007.www1.hpe.com/hk/en/solutions/technology/openflow/index.aspx.

[4] Inter sdn controller communication. http://events.linuxfoundation.org/sites/events/files/slides/ODL-SDNi_0.pdf.

[5] The nox controller. https://github.com/noxrepo/nox.

[6] Opendaylight application developer's tutorial. http://sdnhub.org/tutorials/opendaylight/.

[7] The opendaylight platform. https://www.opendaylight.org/.

[8] OpenDayLight, «Wiki OpenDayLight: OpenDaylight Controller:MD-SAL:FAQ,». https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ. [Online; Última visita: 10/05/2016].

[9] Openflow: Proactive vs reactive. http://networkstatic.net/openflow-proactive-vs-reactive-flows/.

[10] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[11] SDN Tutorials, «Difference Between AD-SAL/MD-SAL,». http://sdntutorials.com/difference-between-ad-sal-and-md-sal/. [Online; Última visita: 20/05/2016].

[12] Sdni:developer guide. https://wiki.opendaylight.org/view/ODL-SDNiApp:Developer_Guide.

[13] Software defined networks. http://docplayer.es/1626116-Software-defined-networks.html.

[14] Open Data Center Alliance. Open data center alliance: Software-defined networking rev. 2.0. *Open Data Center Alliance, Tech. Rep*, 2014.

[15] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[16] Edjard Mota Paulo Fonseca, Ricardo Bennesby and Alexandre Passito. A replication component for resilient openflow-based networking. *Networks Operations, IEEE*, 2012.

[17] RTI. *Users Manual 5.2.0*. Real Time Innovations, 2015.

[18] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. 2010.