UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Verification and Validation Methodology for Interfaces in Network Environments

Javier Galán Sánchez
Tutor: Víctor López Álvarez
Ponente: Jorge Enrique López de Vergara Méndez

Julio 2020

Verification and Validation Methodology for Interfaces in Network Environments

AUTOR: Javier Galán Sánchez
TUTOR: Víctor López Álvarez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2019

Resumen

La verificación y validación de software es una metodología muy utilizada en la actualidad por los desarrolladores de software. Es importante que cada proyecto tecnológico siga unas pautas para validar los cambios y no es menos para los entornos de red. Este TFG desarrolla y evalúa una metodología de validación y verificación de un entorno de red emulado mostrando las principales tecnologías de virtualización y automatización de pruebas.

Concretamente, se generará un entorno de red con una topología que constará de cinco nodos: dos hosts, dos *switches* (conmutador de red) y un servidor de monitorización usando gNMI (gRPC Network Management Interface). Para probar el funcionamiento correcto de la red se generará tráfico en la misma desde un host hacia otro mientras que el servidor de monitorización obtendrá información de ambos *switches* gracias a la telemetría que posee SONiC, el sistema operativo que se introducirá en los dispositivos de red. Para la virtualización y automatización del proyecto se utilizará la tecnología de contenedores Docker, por lo que cada nodo de la red se desplegará sobre un contenedor con su imagen particular. Para la verificación y validación del entorno se hará uso del servicio de integración continua Travis CI. Finalmente se realizarán pruebas para comprobar la conectividad y el correcto funcionamiento de la monitorización de los *switches*.

De esta manera, se construirá una metodología totalmente funcional para la verificación y validación de un entorno de red.

Palabras clave

Validación, Verificación, Virtualización, Automatización, Integración Continua, Entorno de Red, gNMI, telemetría, SONiC, Docker, Travis CI.

Abstract

Software verification and validation is a methodology currently used by software developers. It is important that each technological project follow some guidelines to validate the changes and it is no less for the network environments. This Bachelor Thesis aims to develops a validation and verification methodology for an emulated network environment showing the main technologies of virtualization and test automation.

Specifically, a network environment with a simple topology consisting of five nodes will be generated: two hosts, two switches and a monitoring server with gNMI. To test the correct functioning of the network, traffic will be generated in it from one host to another, while the monitoring server will obtain information status from both switches thanks to the telemetry that SONiC has, the operating system that will be introduced in the network devices. For the virtualization and automation of the project, the Docker container technology will be used, so that each node of the network will be deployed on a container with its particular image. For verification and validation of the environment, Travis CI continuous integration service will be use. Finally, tests will be carried out to verify the connectivity and the correct functioning of the monitoring of the switches.

In this way, a fully functional methodology will be built for the verification and validation of a network environment.

Keywords

Validation, Verification, Virtualization, Automation, Continuous Integration, Network Environment, gNMI, telemetry, SONiC, Docker, Travis CI.

Agradecimientos

Dado el gran reto que ha supuesto la realización de este trabajo con el tiempo tan limitado del que he dispuesto este curso, me gustaría agradecer enormemente a aquellas personas que también han sido partícipes de ello, tanto directa como indirectamente.

En primer lugar, quiero agradecer a mi tutor Víctor López por haberme enseñado la disciplina que requiere un trabajo de este calibre, así como muchos de los conocimientos sobre redes que han permitido la elaboración del proyecto. Otra gran parte del trabajo ha sido posible gracias a la ayuda de Alejandro Aguado, quien me ha resuelto gran parte de las dudas que me iban surgiendo durante la elaboración del trabajo. Y no dejo atrás la figura de todos y cada uno de los profesores que me han hecho crecer académicamente y verme capaz de algo que tanto temía cuando empecé la titulación, el TFG.

A mis amigos, la mayoría de ellos "telecos", quienes siempre me han dado esa motivación extra para afrontar cada problema que se me ha puesto por delante durante estos seis años.

A mi novia, Laura, que seguro es la persona que más me ha apoyado y me ha hecho ver que de verdad era capaz de sacar todo adelante. No hubiera sido posible sin su ayuda incondicional desde el principio y la confianza que siempre ha depositado en mí.

Y, por último, a mi familia. Papá, Mamá y Chía. Vosotros habéis estado ahí cuando he tenido momentos difíciles y asignaturas más complicadas. Siempre estaré agradecido a mis padres por el esfuerzo que han hecho para inculcarme la educación y los valores que me han permitido llegar hasta aquí.

ÍNDICE DE CONTENIDOS

INDICE DE FIGURAS

Glosario

- **ASIC:** *Application specific integrated circuit.* Aplicaciones específicas para circuitos integrados.
- **BGP:** *Border Gateway Protocol.* Protocolo de puerta de enlace de frontera.
- **CRUD:** *Created, Read, Update, Delete.* Se usa para referirse a las funciones básicas de bases de datos.
- **DANOS:** *Disaggregated Network Operating System*. Sistema operative de red desagregado.
- **gNMI:** *Network Management Interface.* Interfaz de gestión de red.
- **gRPC:** *Remote Procedure Call.* Procedimiento Remoto de Llamada.
- **HTTP:** *Hypertext Transfer Protocol*. Protocolo de Transferencia de Hipertexto.
- **IETF:** *Internet Engineering Task Force.* Grupo de trabajo de ingeniería de Internet.
- **JSON:** *JavaScript Object Notation.* Notación de objeto de JavaScript.
- **NOS:** *Network Operating System.* Sistema operativo de red.
- **OSI:** *Open System Interconnection.* Modelo de interconexión de sistemas abiertos.
- **OVS:** *Open vSwitch.*
- **PFC:** *Priority-Based Flow Control. Gestión de flujo basado en prioridades.*
- **RFC:** *Request for comments.* Solicitud de comentarios.
- **SAI:** *Switch Abstraction Interface.* Interfaz de abstracción de switch.
- **SDK:** *Software Development Kit.* Kit de desarrollo de Software.
- **SDN:** *Software Defined Networking.* Redes definidas por software.
- **SONiC:** *Software for Opened Networking in the Cloud.* Software para redes abiertas en la nube.
- **XML:** *Extensible Markup Language.* Lenguaje de marcado extensible.
- **YANG:** *Yet Another Next Generation.*

1 Introducción

1.1 Motivación

La tecnología no cesa su crecimiento exponencial desde hace ya años. Esto requiere que los profesionales de campos como la Informática y las Telecomunicaciones no cesen de adquirir conocimientos constantemente. Ese crecimiento exponencial también provoca que los proyectos de software que se llevan a cabo hoy día tarden poco tiempo en quedarse obsoletos.

Dicho lo cual, se abre la necesidad de validar el software desarrollado cada poco tiempo para verificar que el funcionamiento sigue cumpliendo los requisitos preestablecidos, independientemente de que el entorno donde se esté ejecutando varíe las condiciones. Es decir, la mejora y el crecimiento de la tecnología deben ir unidos a los proyectos que se desarrollan, haciendo que las técnicas para llevarlos a cabo deban ser óptimas en todo momento.

Por otro lado, la infraestructura de red por la que viajan los millones de datos que se mueven diariamente en el mundo es cada vez más compleja. Por ello, los desarrolladores necesitan nuevos métodos de integración de cambios y automatización de pruebas que permitan la actualización de dichas redes para seguir soportando la cantidad ingente de datos que se transmiten.

Otro punto vital para motivar este trabajo es la importancia de los grandes proyectos *Open Source* (código abierto) hoy en día. Estos proyectos permiten que el software no deje de mejorarse gracias a la comunidad de desarrolladores que hay detrás atendiendo a las necesidades de los usuarios que hacen uso de ello.

Se está dejando a un lado la encapsulación del hardware y software de los dispositivos de red por parte de un mismo vendedor dando lugar a las corrientes *Open Hardware* y *Open Software* lo cual permite una gran flexibilidad a la hora de configurar los dispositivos, permitiendo al desarrollador una libertad que hasta hace poco no existía. Esto motiva el hecho de mezclar las evoluciones de los dos mundos para acelerar el desarrollo de los dispositivos de red.

1.2 Objetivos

El objetivo principal de este trabajo es diseñar y desarrollar una metodología de validación y verificación de interfaces en entornos de red. Para ello, se analizarán las distintas posibilidades que se presentan hoy día, comparando las prestaciones de las muchas herramientas tecnológicas que permiten la virtualización y automatización de pruebas en dichos entornos de red y finalmente realizar el diseño y desarrollo seleccionando herramientas y añadiendo código para que funcione el entorno.

El propósito del proyecto se centra en los siguientes objetivos:

1. Estudiar y analizar el marco de trabajo de los sistemas operativos de red de código abierto, sus módulos y herramientas para conocer y ampliar su funcionalidad.
2. Estudiar y analizar el estado del arte del control de *White-Boxes* usando interfaces programáticas.
3. Desarrollar una metodología de verificación y validación para entornos de red.

4. Demostrar y experimentar la metodología definida con un entorno de red emulado usando un sistema operativo de red de código abierto.

1.3 Metodología y plan de trabajo

La realización de este trabajo ha comenzado con una investigación y documentación de las diferentes plataformas que permiten la verificación y validación de software. Es decir, herramientas que permitan la integración continua en proyectos, así como la virtualización de los entornos de producción requeridos. Esto ha requerido la mayor parte del tiempo ya que, a priori, era un área totalmente desconocida y a su vez novedosa.

Una vez adquiridos los conocimientos necesarios se empezó a estructurar el diseño del trabajo que se iba a realizar. Posteriormente, se comenzó el desarrollo del entorno usando las herramientas que se habían preestablecido en el diseño y documentando a su vez las pautas seguidas para llevarlo a cabo.

Finalmente, se ha usado el sistema de pruebas para validar el entorno de red y obtener resultados favorables que verificasen el buen diseño realizado anteriormente.

En la Figura 1-1 se puede observar un diagrama de Gantt que detalla la metodología y planificación seguidas hasta el final del proyecto.

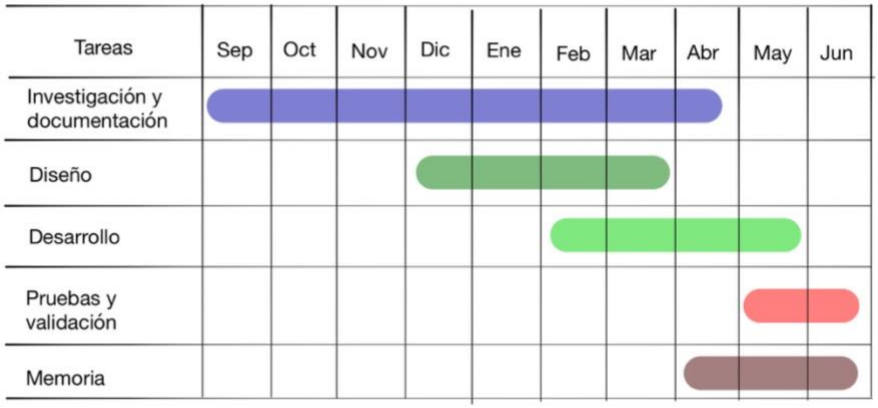

Figura 1-1: Diagrama de Gantt

1.4 Organización de la memoria

Con el objetivo de presentar el trabajo realizado y los resultados obtenidos mostrando la manera en que se ha llegado a los objetivos marcados, este documento consta de los siguientes capítulos:

- **Capítulo 2: Estado del arte.** En este capítulo se ponen de manifiesto los pilares que sustentan este proyecto. Se presentará la corriente actual *White-Boxes* indicando las ventajas que conlleva en los dispositivos de red. Se comentará el lenguaje de modelado de datos YANG, así como los protocolos de configuración de redes Netconf y Restconf. Se incluirá una detallada alusión a la corriente *Open Source* nombrando y explicando dos grandes proyectos como son DANOS y SONiC. Se explicarán las tecnologías de virtualización, incluyendo Docker diferenciándola de las tradicionales máquinas virtuales. Finalmente, se definirán las etapas del ciclo de vida del software que pondrán en contexto este trabajo.

- **Capítulo 3: Diseño.** En este capítulo se hará un minucioso esquema y explicación del entorno de red que se ha realizado en este trabajo. En primer lugar, se definirá el método de integración continua presentando los principales servicios que nos permiten llevarla a cabo. Se explicará con detalle la configuración de cada elemento de la red, así como la funcionalidad de cada uno ellos. Para concluir, se mostrarán las herramientas que se ha decidido utilizar en el entorno de red como solución al sistema operativo de red, la virtualización del entorno y la integración continua del mismo.

- **Capítulo 4: Desarrollo.** En este capítulo se explican con detalle los diferentes ficheros que se han generado para levantar todo el entorno detallado anteriormente en el diseño. En primer lugar, se comentarán las decisiones de configuración realizadas sobre SONiC. Posteriormente, se hablará de cómo se ha realizado la monitorización de los *switches* gracias a la telemetría que puede activarse con SONiC. Finalmente, se pondrá de manifiesto el código necesario para la generación de los contenedores de Docker y la integración con Travis CI.

- **Capítulo 5: Integración, pruebas y resultados.** En este capítulo, se explica el ciclo que ha de seguir el proyecto para que cumpla con una integración continua. Además, se mostrará el sistema de pruebas que verificará y validará el correcto funcionamiento de la red.

- **Capítulo 6: Conclusiones y trabajo futuro.** En este último capítulo se sintetizan las conclusiones extraídas del trabajo que se ha realizado y las asignaturas y herramientas que han sido útiles para el desarrollo del mismo. Además, se detallarán las posibilidades que brinda este trabajo de cara al futuro, comentando la posible escalabilidad del mismo en redes de mucho mayor tamaño.

2 Estado del arte

2.1 Introducción

El mundo de la tecnología no cesa su crecimiento exponencial en todos los aspectos, y esto obliga a cada uno adquirir conocimientos continuamente. En esta sección se presentará la disrupción del arcaico modelo de los dispositivos de red encapsulados, provocada por la aparición de una nueva corriente mucho más flexible y permisiva para los desarrolladores, las *White-boxes*.

2.2 Entornos basados en White Boxes

A continuación, se hablará de la evolución de la arquitectura de los conmutadores de red, a los que nos referiremos de ahora en adelante con el término en inglés *switch*.

Los *switches* eran hasta hace poco dispositivos de red muy encapsulados y poco tolerantes dado que cada componente interno tanto de hardware como de software provenía del fabricante en cuestión. No obstante, hoy día se está estableciendo una nueva generación de estos dispositivos: *White-box Switches*.

2.2.1 Definición de White Boxes

En primer lugar, es de vital importancia conocer la estructura de los componentes de un *switch*. A continuación, podemos observar en la Figura 2-1 dicha estructura.

Figura 2-1: Pila de la distribución de los componentes de un *switch*

Un ASIC (circuito integrado para aplicaciones específicas) es un elemento específico de hardware destinado para una determinada tarea. En el caso de los *switches*, dicha tarea consiste en el envío paquetes por la red. El hardware incluye muchos otros componentes físicos como puertos de entrada/salida, LEDs, etc. El NOS se encarga tanto de controlar el ASIC para los propósitos de red como de facilitar el enlace entre el hardware y las aplicaciones del plano de control para poder gestionar adecuadamente la red. Por último, el plano de control facilita al usuario herramientas para la gestión del *switch* y del propio NOS que contenga.

Para comprender la diferencia entre los *switches* basados en *White-Boxes* y los *switches* tradicionales, tenemos que ver la manera en que interaccionan los componentes entre sí. Si hablamos de un NOS totalmente independiente del hardware, estamos ante un *open switch*. Por otro lado, un *switch* tradicional se entrega totalmente encapsulado y con el software pre-instalado de tal manera que no se puede modificar. Por tanto, los *open switches* permiten al usuario una libertad adicional permitiendo elegir el NOS que deseen instalar en el dispositivo.

Tras varios años desde que comenzó su uso en el mercado, aún hay confusión sobre los dispositivos de red desagregados, es decir, con la posibilidad de poder obtener el hardware de un fabricante por un lado y un NOS por el otro.

Las arquitecturas de los *switches* eran cerradas y propiedad del vendedor hasta la aparición de este nuevo modelo basado en la desagregación de los componentes, más concretamente, del hardware y software del dispositivo en cuestión como muestra la Figura 2-2. Esto genera nuevos horizontes y posibilidades a la hora de configurar cada dispositivo de red.

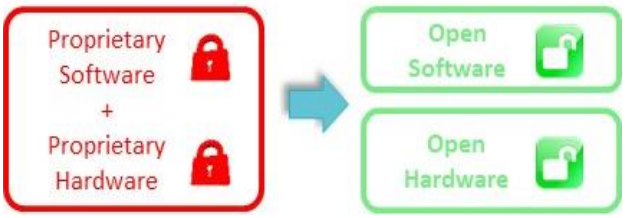

Figura 2-2: Modelo de arquitectura *White-Box* [1].

La clave de esta nueva arquitectura está en que el hardware sea abierto. A diferencia de los servidores, un *switch* no es un dispositivo estandarizado. Es decir, cada diseño es diferente y, por tanto, reúne mayores problemas a la hora de adoptar un NOS. El correcto funcionamiento del hardware de un *switch* con un determinado NOS no es tarea fácil. Es necesario disponer de drivers de los componentes internos tales como sensores, LED, memorias EEPROM, etc. Así como también es indispensable la configuración y el mapeo de los puertos físicos del propio ASIC con los puertos del NOS.

Una vez llegados a este punto, se puede decir que la industria de los dispositivos de red está provocando la creación de un ecosistema más amplio y con muchas más posibilidades que años atrás. En la Figura 2-3 se puede observar que, en un principio, las redes tradicionales estaban regidas por un set hardware-software totalmente cerrado y encapsulado por el propio fabricante. El siguiente paso fue la sustituir el ASIC propietario dando lugar a *switches* cuyo hardware seguía siendo del propietario, pero las piezas de silicio podían proceder de diferentes fabricantes. Finalmente, se está alcanzando una arquitectura totalmente desagregada y abierta que deja a un lado los dispositivos cerrados para dar lugar a una nueva generación (*White-Box Switches*) basada en entornos abiertos y la total adaptación de cada *switch* en función de las necesidades de la red. Hablamos de *Open Software* cuando el software puede ser instalado en cualquier plataforma, mientras que *Open Hardware* sería aquella plataforma que permite instalar cualquier software. En este trabajo, veremos que existen proyectos *Open Source*.

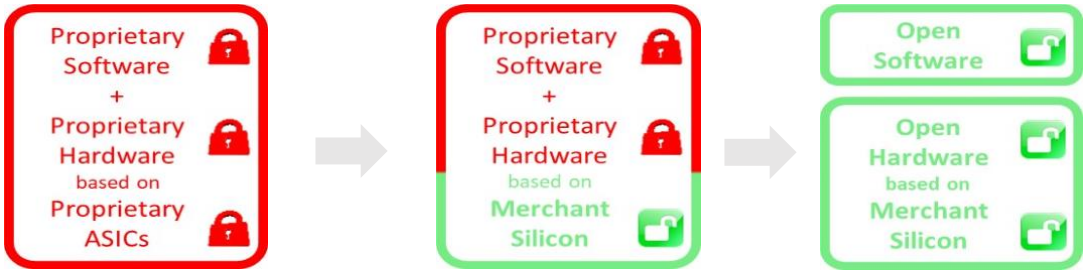

Figura 2-3: Evolución de la arquitectura interna de un *switch* [1].

Cabe mencionar que en esta introducción se ha usado el término *switch*, para dar contexto al capítulo. Sin embargo, esta evolución es aplicable a los *routers*. Muchos *switches* tienen capacidades de nivel 3 y con ello realizan funcionalidades de *router*. A día de hoy tanto los operadores en la nube, como los tradicionales, están viendo esta evolución como una arquitectura alternativa para sus redes.

2.2.1 Interfaces programáticas

YANG es un lenguaje de modelado de datos que permite la configuración de protocolos de gestión de redes como pueden ser Netconf y Restconf. Este lenguaje es desarrollado por el IETF y fue publicado como RFC 6020 en octubre de 2010.

YANG puede ser usado tanto para modelar datos de configuración como datos del estado de los dispositivos de la red. Se puede usar también para definir el formato de notificaciones de eventos emitidos por la red y permite a los modeladores de datos definir la firma de las llamadas a procedimientos remotos (RPC) que pueden ser invocados por los elementos de la red mediante Netconf.

Es un lenguaje modular que representa estructuras de datos en un formato de árbol XML. Los modelos de datos YANG usan Xpath para definir límites en los elementos del modelo. Dado que YANG nació específicamente para usarse con Netconf, este se detalla a continuación. Información más detallada en [2].

El protocolo Netconf es un protocolo de administración de red desarrollado y estandarizado por el IETF, más detallado en [3]. Define un mecanismo simple a través del cual podemos gestionar un dispositivo de red, obtener información sobre la configuración de dicho dispositivo y gestionar ficheros de configuración. YANG se encarga de definir una jerarquía sobre los mismos usando XML como se ha comentado anteriormente.

Netconf está compuesto por cuatro capas claramente diferenciadas:

- Capa de Contenido que alberga los datos de configuración y notificación que son intercambiados entre cliente y servidor.
- Capa de Operaciones que especifica un grupo de operaciones base para obtener y manipular los datos de configuración.
- Capa de mensajes encargada de los mensajes RPC y las notificaciones.
- Capa de Transporte Segura que permite una conexión cliente-servidor confiable para el intercambio de mensajes.

En la Figura 2-4 se ilustra la pila de los componentes del protocolo Netconf.

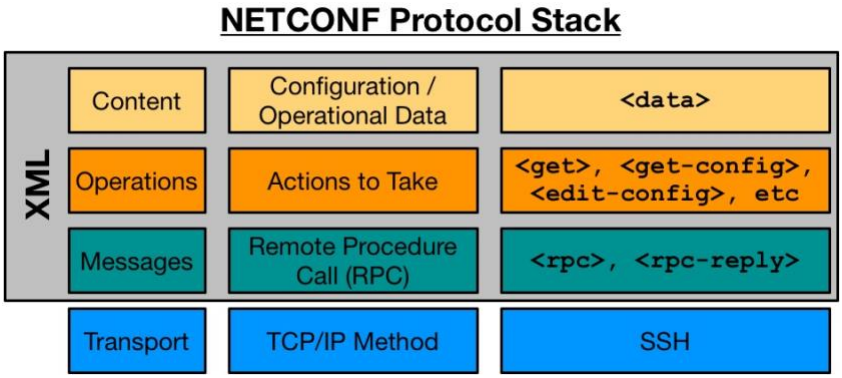

Figura 2-4: Pila del protocolo Netconf [4].

En línea con los protocolos de configuración de redes, Restconf es un protocolo basado en HTTP que también provee una interfaz programática de acceso a datos definidos, una vez más, en lenguaje YANG. De acuerdo a la información que se puede recopilar de [5], se definen varias capas en la estructura del protocolo.

Capas de la pila del protocolo Restconf:

- Capa de contenido que, a diferencia de Netconf, permite usar tanto XML como JSON para el formato de los datos.
- Capa de operaciones que alinea cada una de las operaciones con los diversos métodos HTTP, proporcionando el conjunto requerido de operaciones CRUD (*Create, Replace, Update and Delete*).
- Capa de transporte que usa HTTP como protocolo de transporte, permitiendo también el uso de HTTPs.

En la Figura 2-5 se ilustra la pila de los componentes del protocolo Restconf.

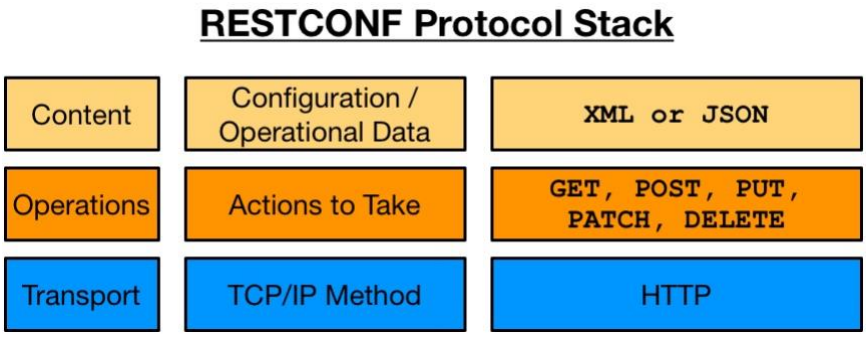

Figura 2-5: Pila del protocolo Restconf [6].

Otra alternativa de las interfaces programáticas es gNMI (*gRPC Network Management Interface*). Al igual que Netconf y Restconf, es un protocolo de administración de redes desarrollado por Google. De acuerdo a la definición dada en [7], gNMI tiene la función de instalar, manipular y borrar la configuración de los elementos de la red al igual que puede comprobar el estado operacional de los mismos. No está únicamente restringido a usar YANG.

Un término que aparece nuevo con gNMI es la telemetría. La transmisión telemétrica permite a un cliente crear una subscripción con un modo específico (*poll*, *once* o *stream*)

con el fin de recibir información del estado de los elementos de la red. Es decir, una subscripción permite a un cliente recibir actualizaciones continuamente de los cambios producidos en el árbol del modelo de datos. Para ello, el cliente necesita indicar uno o varios *paths* sobre los cuales quiere recibir información. Se puede encontrar una descripción de los diferentes modos de subscripción y maneras de hacerlo en [8].

2.2.2 Proyectos Open Source: DANOS y SONiC

Dado este cambio tan drástico que se está produciendo en los entornos de red con la idea de *White-Box* y la mejora en el despliegue de nuevos servicios mucho más ágiles y flexibles, los proyectos *open source* se están convirtiendo en una herramienta totalmente indispensable para la comunidad.

A diferencia del código cerrado y licenciado, muchos proyectos *open source* terminan dando lugar a software gratuito y totalmente libre. El mejor ejemplo es sin lugar a duda Linux, proyecto que ha dado lugar al sistema operativo con la mayor libertad hoy día y ha ayudado en gran medida al desarrollo de Android.

En lo que a entornos de red se refiere, uno de los mayores proyectos *open source* es dNOS (*Disaggregated Network Operating System*) llevado a cabo por AT&T. Tras acoger dNOS, la Fundación Linux modificó el nombre llamando al proyecto: DANOS. El objetivo de DANOS es fomentar la corriente de *White-Boxes* y por ello tiene las siguientes metas según [9]:

- Separación del sistema operativo de red (NOS) del hardware de los dispositivos de red.
- Desarrollo de APIs estándar bien definidas que proporcionen un *framework* en el sistema operativo, un plano de control y gestión y un plano de datos.
- Desarrollo de APIs estándar bien definidas que suministren una división lógica entre el plano de control (*control plane*) y el plano de datos (*data plane*), es decir, una separación entre la gestión y mantenimiento de la red y la gestión del tráfico destinado a los servicios.

En la siguiente imagen se muestra un diagrama de las capas que presenta DANOS.

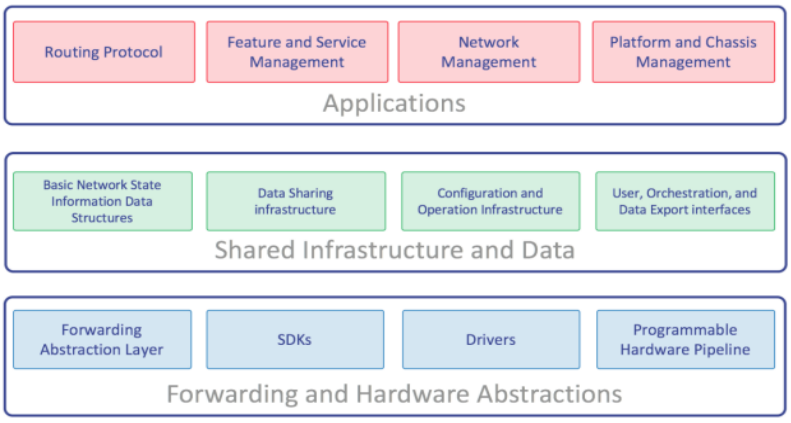

Figura 2-6: Capas y componentes de DANOS [9].

Se observa que DANOS está separado por tres capas funcionales desde el propio *hardware* hasta el nivel de aplicación y cada una de ellas con varios módulos con una función diferente.

En primer lugar, la capa de aplicaciones contiene cualquier servicio y herramienta relacionada con los protocolos de red. Por otro lado, la capa de infraestructura se encarga de almacenar todo tipo de datos e información compartida sobre el estado de los dispositivos de red o tabla de vecinos entre otros. En último lugar, se encuentra la capa que posee los módulos de *drivers* y SDKs del dispositivo en cuestión.

Además, como se ha comentado anteriormente, existe una separación lógica entre el plano de control y el plano de datos. En la Figura 2-7 se observan ambos módulos de control claramente diferenciados. El plano de control y gestión, enlazado a la base del propio sistema operativo, se encarga de orquestar los protocolos de red mientras que el plano de datos tiene la función de mantener sincronizado el bajo nivel (*hardware* y *software*) con el plano de control.

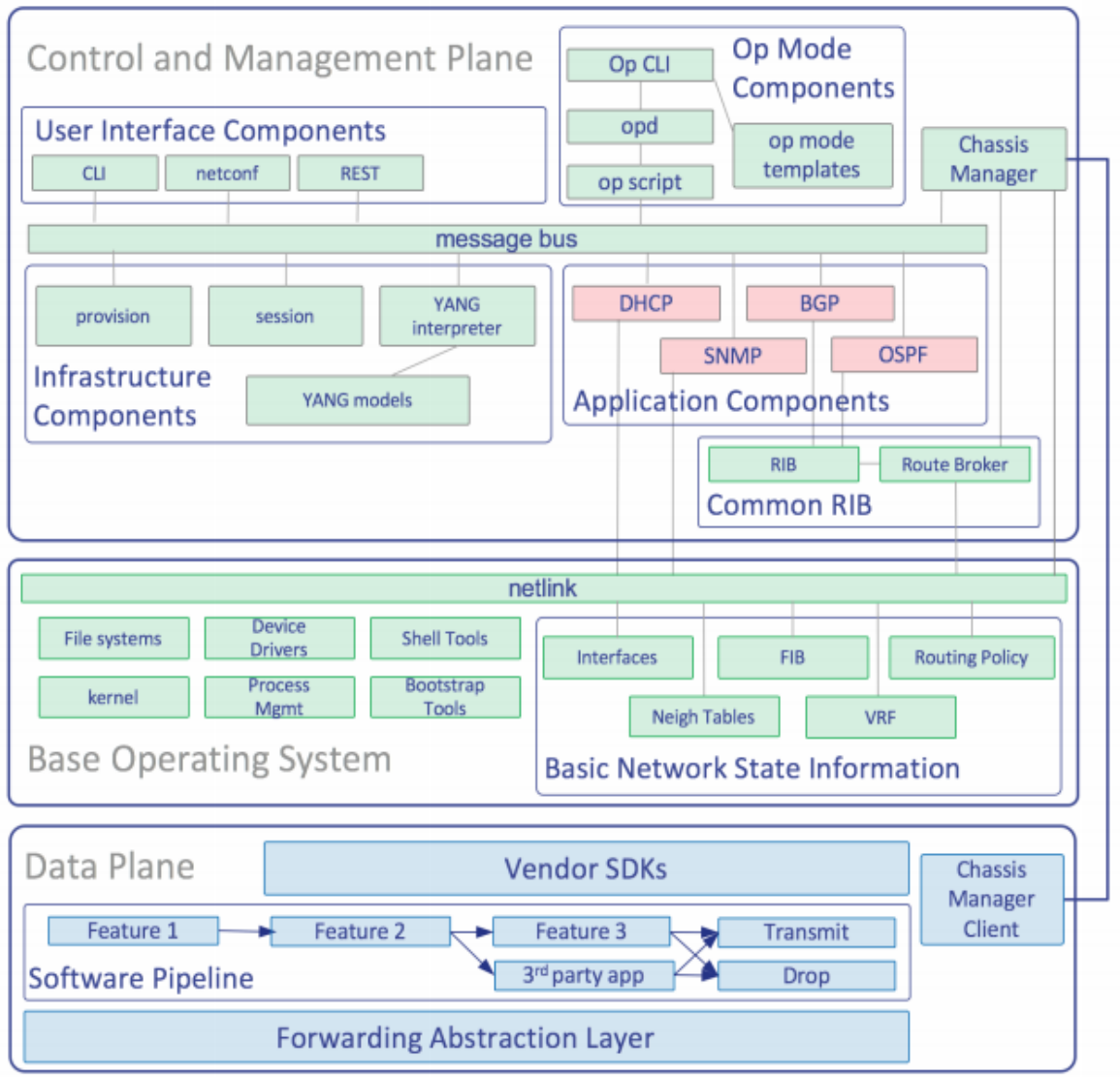

Figura 2-7: Plano de control y plano de datos de DANOS [9].

Otro de los proyectos *open-source* en cuanto a sistema operativo de redes se refiere es SONiC (*Software for Open Networking in the Cloud*). A pesar de que SONiC está basado

en Linux, es desarrollado actualmente por Microsoft con la pretensión de hacer los dispositivos de capa 3 (o *routers*) del modelo OSI completamente funcionales. En la Figura 2-8 se observa la estructura del software de SONiC.

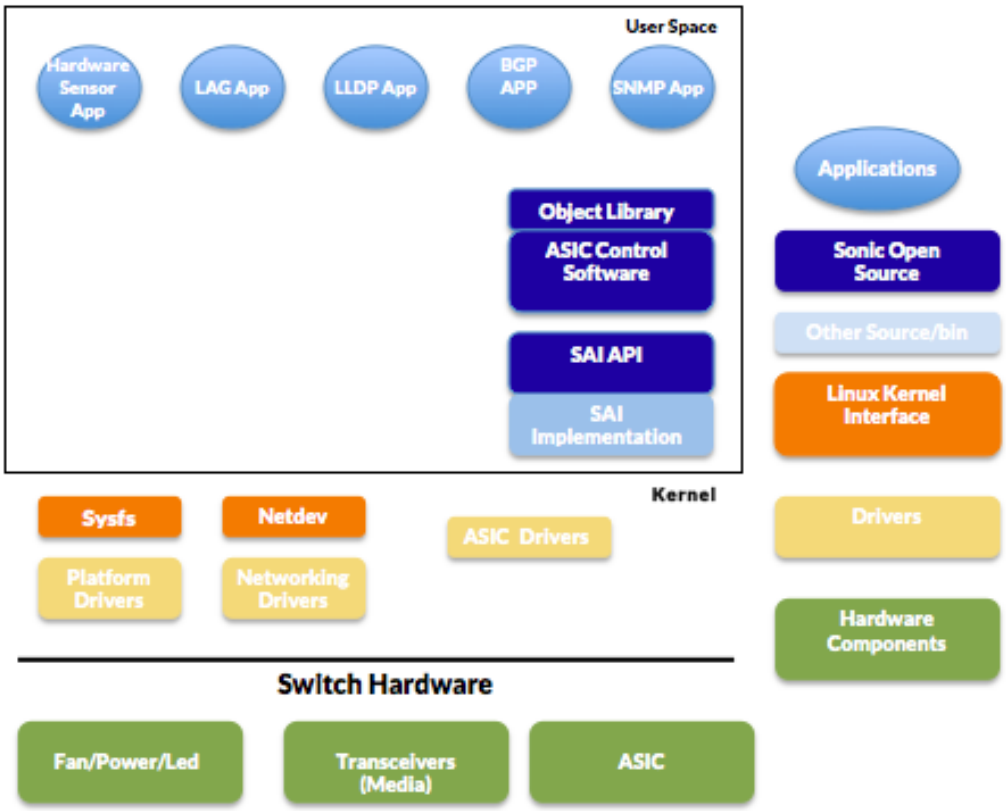

Figura 2-8: Arquitectura de SONiC.

SONiC usa SAI (*Switch Abstraction Interface*) para programar el ASIC de los dispositivos de red, lo que hace a SONiC compatible con todos los ASICs que tienen soporte para SAI. Las librerías *open-source* de SONiC [10] permiten a otras aplicaciones interaccionar entre ellas y con las propias aplicaciones que facilita SONiC.

Por otro lado, SwSS (*Switch State Service*) actúa como módulo de control del ASIC usando una base de datos como interfaz entre las aplicaciones de red que corren en el *switch* y el propio hardware del *switch*. La Figura 2-9 nos muestra, a su vez, la estructura del SwSS.

Las aplicaciones de red usan una base de datos llamada APP_DB para lectura y escritura. Los agentes de orquestación son responsables de la sincronización entre APP_DB y otra base de datos llamada ASIC_DB. Para otorgar una mayor universalidad, las bases de datos están configuradas con un almacenamiento de tipo clave-valor.

SwSS, por tanto, permite a las aplicaciones de red ser ejecutadas sobre SONiC de una manera completamente independiente del hardware del dispositivo. Esto último es un claro punto a favor de la corriente *White-Box* que se ha explicado anteriormente.

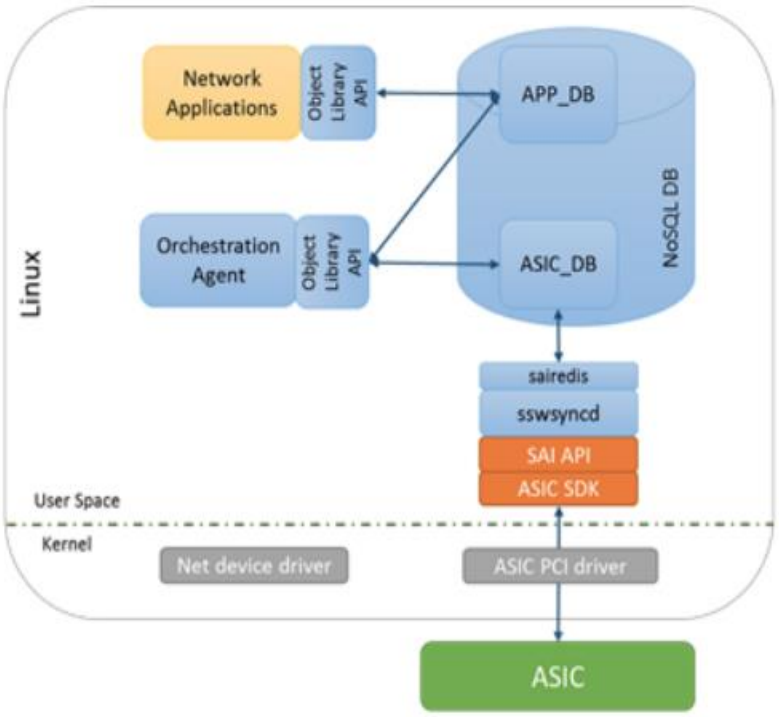

Figura 2-9: Diseño del SwSS, donde se muestra la relación entre APP_DB, ASIC_DB y los agentes de orquestación [11].

2.3 Virtualización para emulación de redes

En esta sección se describirán las diferentes herramientas disponibles para la creación de redes emuladas. En primer lugar, se hará una breve descripción de lo que es una máquina virtual para luego dar paso a dos tecnologías basadas en contenedores viendo así las diferencias entre ambos métodos de virtualización.

2.3.1 Tecnologías para emulación de redes

Una máquina virtual, según [12] es un software desarrollado con el objetivo de poder albergar un sistema operativo y comportarse igual que un equipo real. Esto crea un entorno ideal para emular otros sistemas operativos, incluidas versiones beta, acceder a datos infectados por virus, crear copias de seguridad de sistemas operativos y ejecutar software o aplicaciones en sistemas operativos para los que no se habían creado inicialmente.

Cada máquina virtual proporciona su propio hardware virtual, incluidas las CPU, memoria, unidades de disco duro, interfaces de red y otros dispositivos. El hardware virtual se asigna después al *hardware* real de la máquina física, lo que permite ahorrar costos, porque reduce la necesidad de tener sistemas de hardware físico, con los costos de mantenimiento que conllevan, y también reduce la demanda de alimentación y refrigeración.

Hay dos tipos de máquinas virtuales diferenciadas por su funcionalidad:

- Máquinas virtuales de sistemas: son aquellas que emulan un equipo al completo, es decir, se puede ejecutar un sistema operativo en su interior y además tiene su propio

disco duro, memoria, tarjeta gráfica y otros componentes de hardware, todos ellos virtuales.

- Máquinas virtuales de procesos: son aquellas que solo ejecutan un proceso en concreto, como puede ser una aplicación en su entorno de ejecución. Cada vez que se usa Java o .NET estamos en realidad ante una máquina virtual de proceso. Esto es de gran utilidad a la hora de desarrollar aplicaciones para varias plataformas, pues en vez de tener que programar específicamente para cada sistema, el entorno de ejecución (es decir, la máquina virtual) es el que se encarga de lidiar con el sistema operativo.

El exponencial crecimiento que hoy día está sufriendo la tecnología en general es buen motivo por el cual las máquinas virtuales se están quedando un poco obsoletas con la aparición de la tecnología de contenedores.

Los contenedores funcionan de una manera diferente a las máquinas virtuales ya que no necesitan alojar el sistema operativo al completo, sino que sólo alojan la aplicación y las librerías necesarias compartiendo los recursos del propio sistema operativo sobre el que se ejecutan. Es por ello que, la principal diferencia y, a su vez, ventaja de los contenedores frente a las máquinas virtuales es que los primeros necesitan mucho menos espacio y sobrecargan poco, con lo que los tiempos de ejecución son óptimos.

Veamos la principal diferencia entre máquinas virtuales y contenedores en la Figura 2-10 y la Figura 2-11.

Figura 2-11: Capas de un equipo con tres máquinas virtuales.

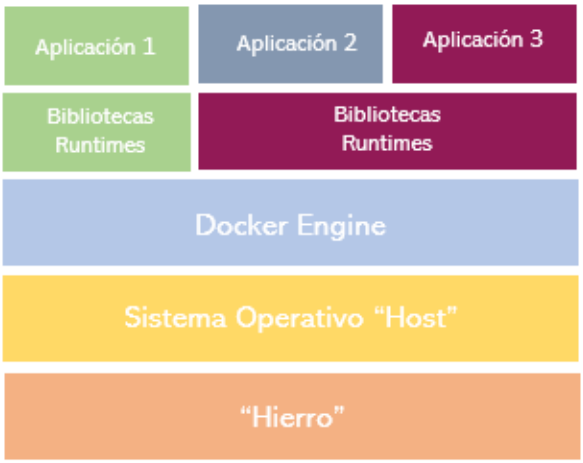

Figura 2-10: Capas de un equipo con dos contenedores.

Como se puede observar, la capa del sistema operativo se elimina por completo haciendo que los contenedores sean mucho más ligeros.

A continuación, se va a describir dos de los proyectos más importantes en cuanto a tecnología de contenedores: Kubernetes y Docker.

Kubernetes es una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores. Kubernetes agrupa los contenedores que conforman una aplicación en unidades lógicas para una fácil administración y descubrimiento.

Este orquestador de contenedores fue diseñado inicialmente por Google, quien después lo donó a la *Cloud Native Computing Foundation*. Está escrito en el lenguaje de programación Go y se puede desplegar en múltiples entornos *cloud* o en *bare-metal* y soporta múltiples *runtimes* de contenedores como Docker o rkt entre otros.
Por otro lado, Docker es una herramienta de un nivel por debajo de Kubernetes. Es el software que nos permite la creación propia de contenedores en un equipo. Dado que Docker permite la creación de contenedores con las librerías y recursos necesarios para ejecutar la aplicación que se desee, le hace ser un software idóneo para testing. Es decir, Docker permite a los equipos de desarrollo tener un entorno de pruebas específico y de manera ligera y portable.

La estructura de Docker está compuesta principalmente por tres elementos:

- Contenedores: Son como un directorio, contienen todo lo necesario para que una aplicación pueda funcionar y ejecutarse sin necesidad de acceder a un repositorio externo al contenedor.

- Imágenes: Una imagen de Docker se podría entender como un sistema operativo con aplicaciones previamente instaladas. Dichas imágenes son customizables, de tal manera que es posible añadir aplicaciones que se vayan a necesitar en otro equipo donde se tenga intención de usar la imagen. En adición, Docker presenta una forma intuitiva de actualizar las imágenes, así como un método muy eficaz para crearlas (Dockerfile).

- Repositorios: También conocidos como registros, contienen imágenes creadas por otros usuarios y están completamente disponibles a todo el público (Docker Hub). Se pueden encontrar repositorios públicos y gratuitos donde conseguir las imágenes que se requieran. Estás imágenes las podemos ver como plantillas que permiten al desarrollador ahorrar tiempo en la creación e implementación de aplicaciones o sistemas.

2.3.2 Docker: Comandos básicos

En primer lugar, se hará una descripción del proceso de creación de una imagen con Dockerfile. Esta es una herramienta que permite al usuario crear imágenes a medida mediante una serie de instrucciones ubicadas dentro de un fichero. Es decir, gracias a un fichero Dockerfile es posible la creación de imágenes de forma automática. Dentro de este fichero se puede hacer uso de varios comandos para crear instrucciones. Entre los más importantes se encuentran:

- FROM: (Obligatorio como primera instrucción del archivo) Especifica la imagen base desde la que se quiere crear el contenedor Docker.
- ENV: Configura las variables de entorno.

- ADD: Se encarga de copiar ficheros y directorios desde una ubicación específica y agregarlos al sistema de ficheros del contenedor.
- EXPOSE (obligatorio): Muestra los puertos que se exponen en el contenedor. (Para que los puertos sean accesibles desde el host es necesario añadir la opción *–p* en el comando *docker run*).
- WORKDIR: Indica el directorio por defecto donde se ejecutan las acciones.
- ARG: Permite añadir parámetros al Dockerfile lo cual puede ser de gran utilidad en numerosas ocasiones.
- RUN: Especifica uno o más comandos que instalan paquetes y configuran la aplicación dentro de la imagen. Por ejemplo, se podría utilizar para descargar la librería de los comandos para configuración de interfaces de red de Linux de la siguiente manera: *RUN sudo apt-get install net-tools*.

Ahora bien, una vez se diseña una imagen a medida gracias a un fichero Dockerfile es necesario generarla y posteriormente saber cómo se puede integrar en un contenedor para así hacer uso de ella. Por ello, se describirán a continuación los comandos más básicos de Docker que permiten al usuario interaccionar tanto con las imágenes como los contenedores.

Comandos básicos de Docker:

- *docker build*: Permite crear una imagen a partir del fichero Dockerfile.
- *docker pull NOMBREIMAGEN*: Permite descargar una imagen de los repositorios públicos de Docker.
- *docker images*: Muestra las imágenes descargadas localmente en el equipo desde Docker Hub.
- *docker rmi IMAGE_ID*: Borra una imagen por su ID.
- *docker ps*: Lista todos los contenedores creados con información sobre la imagen que contienen, fecha de creación y su ID entre otros.
- *docker stop CONTAINER_ID*: Permite parar un contenedor por su ID.
- *docker start CONTAINER_ID*: Permite arrancar un contenedor por su ID.
- *docker rm CONTAINER_ID*: Borra un contenedor dada su ID.
- *docker run*: Es el comando básico para crear y arrancar un contenedor con una imagen dada. Por ejemplo, si se ejecutara "*docker run –it ubuntu:14.04 bash"* entonces se generaría un contenedor con una imagen de ubuntu:14.04 indicando, con *–it* que se quiere dejar un terminal abierto para trabajar de forma interactiva con el contenedor.

2.4 Ciclo de vida del software

La ingeniería del software es una rama principal de la informática. El estudio de las diferentes etapas que debe seguir un proyecto de software permite una mejor organización para poder generar un producto de la manera más óptima posible.

La razón básica por la que se requiere disponer de un proceso de desarrollo es mejorar la seguridad de trabajo eliminando riesgos innecesarios y conseguir un producto de la máxima calidad. Además, la escalabilidad es una propiedad importante de un proceso, ya que las dimensiones de los proyectos de software son muy variables. Una de las propiedades que deben ser exigidas a un proceso de desarrollo de aplicaciones software es la escalabilidad, lo que hace posible que sea aplicable tanto a sistemas complejos como a sistemas sencillos.

El modelo clásico del proceso de desarrollo de software es el modelo en cascada. Consiste en una secuencia de actividades: análisis de requisitos, diseño, desarrollo, integración y pruebas. No se comienza una fase hasta que no se ha terminado la anterior.

Véase a continuación una descripción detallada de cada etapa en el ciclo de vida del software:

- **Captura de requisitos:** Esta etapa comprende aquellas tareas que determinen las necesidades o condiciones que debe satisfacer el software. Puede haber requisitos funcionales, los cuales describen los servicios que se esperan del sistema en cuestión, y los no funcionales, que son restricciones sobre los requisitos funcionales.
- **Diseño de software:** Etapa que requiere un análisis que detalle los cimientos bajo los cuales se va a desarrollar el software. Es decir, se debe establecer la estructura de los datos y la arquitectura general del software.
- **Desarrollo y codificación:** Esta etapa tiene el objetivo de traducir el diseño en una forma legible por una máquina. Es decir, se ha de elaborar el código que permita dar los servicios que los requisitos han establecido basándose en el diseño realizado anteriormente.
- **Integración:** Etapa en la cual se ha de probar que los diferentes módulos que forman el producto encajen perfectamente en el entorno de producción. La integración continua es un método eficaz para validar esta etapa de manera óptima.
- **Pruebas para verificación y validación:** Etapa donde se comprueba que el producto desarrollado está acorde a los requisitos y satisface las necesidades del cliente. Verificar el software es percatarse de que se cumplen los requisitos obtenidos en la primera etapa, mientras que validarlo es ver que cumple las expectativas del cliente.
- **Mantenimiento:** Esta etapa es la que requiere mayor tiempo. Es imposible asegurar que un producto de software está perfectamente desarrollado. Por ello, se necesita una fase de mantenimiento constante una vez el producto se entrega al cliente.

2.5 Conclusión

Tras conocer las bases que asientan este proyecto, se puede comprobar que la aparición de la corriente *White-Boxes* está permitiendo una gran flexibilidad en los dispositivos de red permitiendo introducir Open Software aumentando así las prestaciones de la red. Se ha comprobado la funcionalidad de YANG como lenguaje de modelado de datos para poder definir estructuras de datos que permitan mayor interoperabilidad entre los diferentes protocolos de configuración de redes. También se han adquirido conocimientos sobre los proyectos *Open Source* y las herramientas de virtualización actuales. Por último, se ha detallado la importancia de las fases del ciclo de vida del software.

3 Diseño

3.1 Introducción

En esta sección se explicará las decisiones tomadas para diseñar el entorno que se tenía como objetivo. En primer lugar, se hará una breve descripción de la integración continua de software. Para ello se comenzará con el conocido sistema de control de versiones Git. A continuación, se presentarán tres de los principales gestores de integración continua: Ansible, Jenkins y Travis CI. Finalmente, se introducirán las tecnologías y herramientas que se ha decidido utilizar para llevar a cabo el desarrollo del trabajo.

3.2 Integración continua de software

La integración continua de software es un método de desarrollo de software basado en la existencia de un repositorio central donde el trabajo y código implementado por los desarrolladores se va combinando periódicamente. Por tanto, un servicio de integración continua genera un entorno específico para testear el código de un proyecto y verificar que los cambios que se hayan realizado anteriormente no han provocado fallos en el mismo.

La integración continua presenta tres claros beneficios en el ciclo de vida del software:

- Mejora la productividad de desarrollo. La integración continua mejora la productividad del equipo al liberar a los desarrolladores de las tediosas tareas manuales y fomentar comportamientos que ayudan a disminuir el número de errores en el código.
- Se encuentran y arreglan los errores con mayor rapidez. Gracias a la realización de pruebas más frecuentes, el equipo puede descubrir y arreglar los errores con mayor antelación.
- Entregas y actualizaciones en el menor tiempo posible. La integración continua le permite a un equipo de desarrollo de software entregar actualizaciones a los clientes con mayor rapidez y frecuencia.

Antes de ver algunas de las herramientas que permiten la integración continua de software, se expondrá la base de este tipo de proyectos: Git. Posteriormente, se describirán las principales características de los gestores de integración continua actuales, incluyendo una descripción de tres de ellos: Ansible, Jenkins y Travis CI.

3.2.1 Git

Git es un sistema de control de versiones creado en 2005 por Linus Torvalds para gestionar el desarrollo de código del núcleo de Linux. Está diseñado para trabajar de forma concurrente en diferentes partes de código mejorando así la eficiencia del mantenimiento de versiones de un software.

Su principal objetivo es coordinar el trabajo que llevan a cabo los programadores sobre un mismo archivo de manera inteligente y optimizada. Cada proyecto desarrollado sobre Git se compone de una rama maestra o principal la cual se encuentra publicada en producción y por ello debe estar estable. Desde ella se pueden generar paralelamente otras ramas secundarias que permiten a los desarrolladores implementar módulos sin modificar la rama principal para, en un futuro, enlazar de nuevo con la rama maestra cuando se cercioren de

que no hay errores. De esta manera, Git ejerce un control sobre cada versión que se sube al repositorio haciendo que cada miembro del proyecto pueda ver el estado del mismo.

3.2.2 Gestores de integración continua

Los gestores de integración continua son servicios distribuidos que permiten a los desarrolladores de software automatizar el proceso de integración de código. Con la integración continua, los desarrolladores envían los cambios de forma periódica a un repositorio compartido con un sistema de control de versiones como Git. Un servicio de integración continua crea y ejecuta automáticamente pruebas de unidad en los nuevos cambios realizados en el código para identificar inmediatamente cualquier error.

3.2.2.1 Ansible

Ansible es un software que automatiza el aprovisionamiento de software, la gestión de configuraciones y el despliegue de aplicaciones en varios sistemas. Está categorizado comúnmente como una herramienta de orquestación y disponible tanto en Linux como Mac, pero no para Windows.

La principal característica de Ansible es que gestiona sus diferentes nodos a través de SSH y únicamente requiere Python en el servidor remoto en el que se vaya a ejecutar para poder utilizarlo. Usa el formato YAML para describir acciones a realizar y las configuraciones que se deben propagar a los diferentes nodos. Como veremos a continuación en un esquema de Ansible (Figura 3-1), el nodo principal se nutre de dos archivos: *Inventory* y *Playbook*. *Inventory* contiene las direcciones IP de los nodos que van a ser gestionados mientras que *Playbook*, el cual sigue el formato YANG, posee las acciones que tendrán lugar en cada uno de los nodos declarados previamente en el fichero *Inventory*.

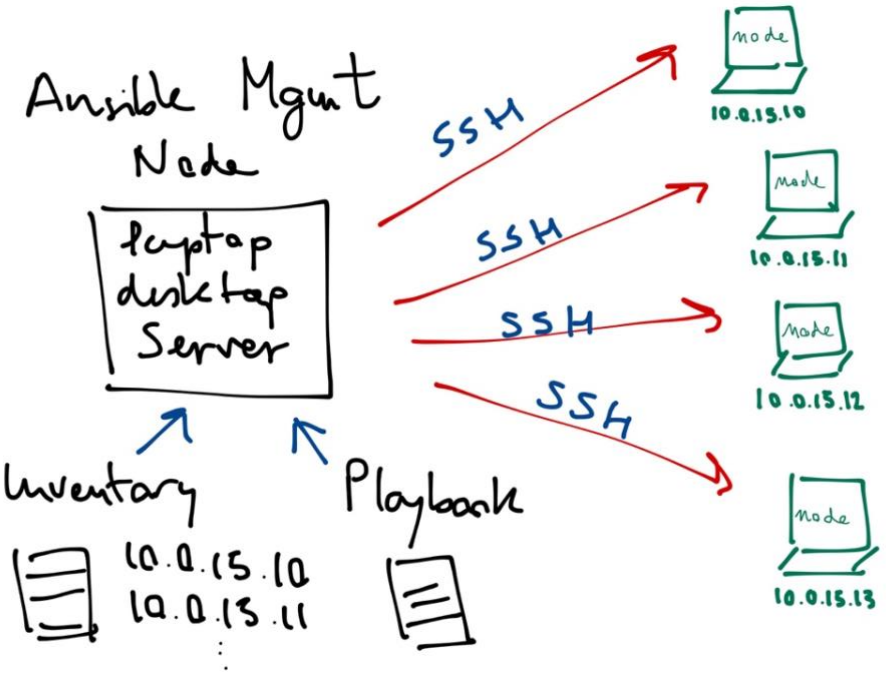

Figura 3-1: Esquema de un entorno con Ansible.

3.2.2.2 Jenkins

Jenkins es un servidor de integración continua *open-source* escrito en Java y completamente gratuito. Jenkins necesita ser corrido en un servidor y soporta herramientas de control de versiones como Git o Subversion. Puede ejecutar proyectos basados en Ant y Maven, así como scripts programados en Bash.

Jenkins es una excelente herramienta para grandes proyectos donde se necesita una gran cantidad de personalizaciones y configuraciones gracias a su sistema de *plugins*. El amplio conjunto de *plugins* hace que Jenkins sea flexible y permita construir, implementar y automatizar en varias plataformas.

Cuando un desarrollador sube su código al control de versiones, Jenkins se encarga de construir ese software (hacer *build*) y, si todo ha ido bien, de ejecutar otras tareas que se le haya indicado, como pueden ser la instalación de esa versión de software en un determinado entorno, y la ejecución de una serie de pruebas que nos indicarán que la nueva versión de software funciona correctamente.

3.2.2.3 Travis CI

Travis CI es otro entorno de integración continua, gratuito para proyectos *Open Source* y de pago para proyectos privados. Trabaja conjuntamente con *GitHub* con el objetivo de testear y construir (hacer *build*) aplicaciones escritas en lenguajes de programación como Java, Go o Ruby entre otros.

Para utilizar Travis desde un repositorio GitHub es necesario que la raíz del proyecto posea un archivo llamado travis.yml. Este archivo contiene las directrices que seguirá Travis para montar el entorno deseado y de esta manera poder ejecutar el *build* que permitirá testear el proyecto en cuestión. Este formato hace que Travis sea menos personalizable que Jenkins, ya que este último permite realizar tareas mediante *jobs*. Además, Travis no posee un sistema de *plugins*. No obstante, mientras que Jenkins necesita un servidor dedicado, Travis está desplegado completamente en la nube, lo que supone una gran ventaja.

3.3 Diseño de la metodología de verificación y validación para entornos de red

A continuación, se hará una descripción del diseño del entorno de red en el que se ha trabajado, así como de las diferentes decisiones tomadas para el desarrollo del mismo.

En primer lugar, se mostrará un esquema (Figura 3-2) del entorno para situar cada uno de los componentes de la red sobre la que se va a trabajar. Se puede observar que la topología consta de cinco elementos: Host1, Host2, Switch1, Switch2, gNMI client.

Cada uno de ellos se desplegará sobre un contenedor de Docker con una imagen personalizada. Tanto Host1 como Host2 correrán sobre un Ubuntu 14.04. Por otro lado, los *switches* tendrán una imagen de SONiC lo cual permitirá realizar telemetría sobre ambos desde un contenedor que ejercerá como un servidor de monitorización.

Este es, por tanto, el diseño de la topología de red llevado a cabo en este trabajo y sobre el que se tratará de ahora en adelante.

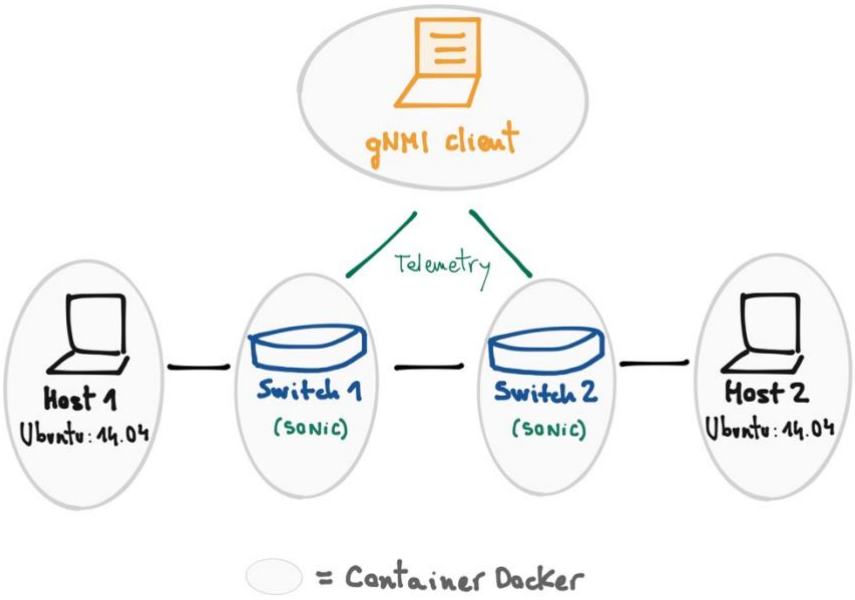

Figura 3-2: Topología del entorno de red.

3.3.1 Sistema operativo de red Open Source

Como bien se ha comentado en las secciones anteriores, los *switches White-box* permiten al usuario introducir el NOS que desee. Como decisión de diseño, SONiC ha sido el NOS elegido al ser completamente *Open Source* y permitir realizar telemetría.

Para ello, la imagen de SONiC seleccionada ha sido la siguiente [13]. Dado que se quieren generar dos *switches* virtuales, se ha decidido elegir SONiC-P4 como imagen de Docker, lo que nos permite emular el comportamiento a su vez del SAI del ASIC de un *switch* real.

3.3.2 Solución de virtualización

Con el objetivo de virtualizar todo el entorno de red se ha decidido usar la tecnología de Docker. Cada elemento de la red será un contenedor con una imagen preestablecida. Para generar las imágenes se ha usado en algún caso el archivo Dockerfile. Este archivo, como se ha comentado en secciones anteriores, permite indicar varias acciones para personalizar el contenedor según las necesidades del desarrollador.

3.3.3 Solución de integración continua

Por último, con la premisa de llevar a cabo una integración continua, Travis CI permitirá construir el entorno de red y testear el código de forma que siempre que algo se modifique, el desarrollador se cerciore de que toda la funcionalidad previa sigue intacta y realmente se ha avanzado en el proyecto.

Para ello se ha necesitado crear un usuario de Travis CI como un usuario de GitHub. Una vez que se dispone de ambas cuentas, el código se sube a un repositorio de GitHub cuya raíz contenga un archivo travis.yml. Este archivo, como muestra la Figura 3-3, contiene varias acciones y directrices que personalizan la construcción del entorno que se desea montar.

22 lines (16 sloc) | 288 Bytes — Raw | Blame | History

```
language: bash
os: linux
sudo: required

services:
  - docker

install:
  - sudo apt-get install -y net-tools
  - sudo apt-get install -y bridge-utils
  - ./install_docker_ovs.sh
  - ./load_image.sh

before_script:
  - ./start.sh

script:
  - ./test.sh

after_script:
  - ./stop.sh
```

Figura 3-3: Fichero travis.yml

A continuación, se explica cada acción del archivo:

- *sudo*: Esta etiqueta indica el lenguaje que se va a utilizar (bash), la distribución (en este caso Linux) y que se requerirá ser superusuario.

- *services*: Esta etiqueta indica qué servicios de Travis se van a utilizar. Los servicios de Docker serán necesarios para la integración y construcción del entorno por lo que se indica en el archivo.

- *install*: En esta etiqueta se indica lo que se ha de instalar para que los test posteriores puedan ejecutarse correctamente y tengan a su disponibilidad las herramientas necesarias para ello. Son necesarias las librerías *net-tools* y *bridge*-utils para configurar la red. Además, se instala tanto Docker como OVS (*Open vSwitch*). Dado que en un principio este entorno se construía localmente, era necesario la instalación de Docker. Pero como se acaba de ver, el propio gestor de integración Travis nos facilita este servicio luego se podría prescindir de ello. *Open vSwitch* es un software *Open Source* necesario para la virtualización de los dispositivos de red utilizados.

- *before_script*: Esta etiqueta contiene aquellos scripts que se ejecutaran previamente a los test a modo de preparación. En este caso, start.sh monta el entorno de red creando los contenedores y las subredes necesarias para su conexión.

- *script*: En esta sección se indican los archivos ejecutables que van a testear el proyecto.

- *after_script*: Finalmente con esta etiqueta se indica todo aquello que es necesario tras la ejecución de los test. En este trabajo es necesario eliminar los contenedores creados y los bridges generados para su conexión con el archivo stop.sh.

3.4 Conclusión

En este capítulo se ha puesto de manifiesto el concepto de integración continua y los principales gestores que dan esta metodología como servicio. Con ello, se ha podido realizar tanto el diseño de la topología de red como decidir qué tecnología de virtualización e integración continua se utilizarán en el desarrollo del trabajo.

4 Desarrollo

4.1 Introducción

Esta sección explicará cómo se ha desarrollado el entorno de red virtual. Primero, se explicará la configuración del SONiC de ambos *switches*. Se hará una descripción del proceso de monitorización que se lleva a cabo usando la telemetría que proporciona SONiC. Posteriormente, se comentará la estructura de cada contenedor de Docker generado para cada elemento de la red. Por último, se verá cómo Travis CI permite crear el entorno en la nube descargando las dependencias necesarias y hacer pruebas sobre él.

4.2 Configuración de SONiC

Partiendo del diseño, el cual se ha comentado en la sección anterior, ambos *switches* deben tener un archivo de configuración para especificar los parámetros de red necesarios para que la red tenga una coherencia. Por ello, tanto *switch1* como *switch2* poseen un archivo con nombre *vlan_config.json.* Este archivo, como se muestra en Figura 4-1 y Figura 4-2, contiene una serie de configuraciones que son necesarias para generar las *vlan* necesarias que permitirán la conectividad de la red: *host1 – switch1 – switch2 – host2*.

Figura 4-1: vlan_config.json (switch1)

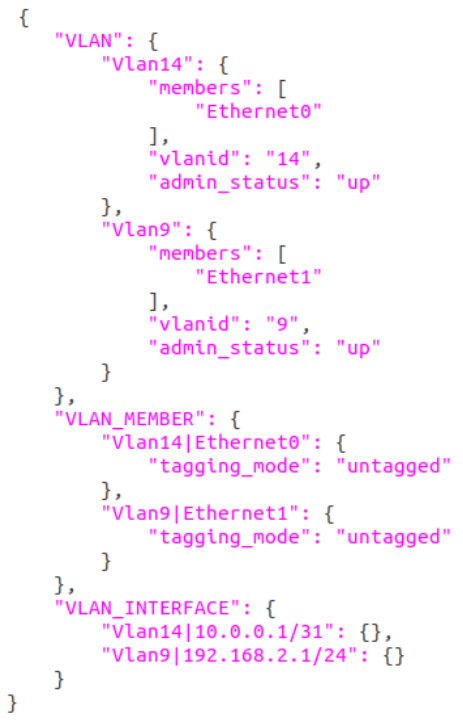

Figura 4-2: vlan_config.json (switch2)

Se observa que el switch1 genera la *vlan15* y *vlan10* para la conectividad con el *switch2* y *host2* respectivamente, mientras que el *switch2*, a su vez, crea la *vlan14* para la conectividad con el *switch1* y la *vlan9* para la conectividad con el *host2*. Además, se ha de indicar qué interfaces (Ethernet0 o Ethernet1) se utilizarán para cada *vlan*.

Una vez se establece esta previa configuración, SONiC obtiene estos ficheros y los introduce en su base de datos redisDB. Esto tiene lugar en el momento en que se crea el contenedor de

Docker con la imagen de SONiC. Además, es importante añadir que SONiC organiza la información con una jerarquía tabla, clave, campo y valor. Más adelante se comentará la estructura de redisDB en SONiC. Siguiendo las pautas indicadas por el proyecto SONiC y añadiendo un servidor de monitorización se llega al siguiente entorno.

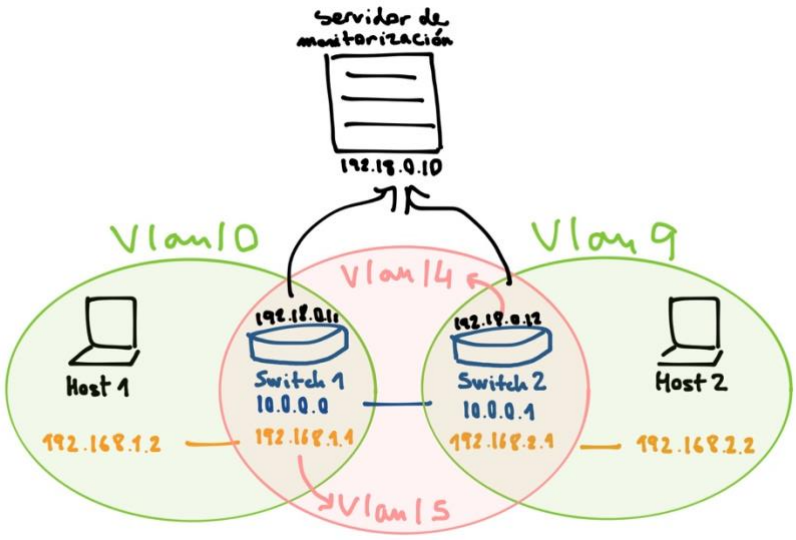

Figura 4-3: Esquema con direcciones IP de los elementos del entorno de red.

Con todo esto, se tendría configurado cada switch para permitir el tráfico de red desde un extremo hacia el otro con el futuro objetivo de poder monitorizar cierta información usando la telemetría de SONiC.

4.3 Configuración del sistema de monitorización

Como bien se ha comentado anteriormente, con SONiC se dispone de un sistema operativo de red *Open Source* que presenta diversos servicios, entre ellos, la telemetría. Antes de nada, es importante conocer la manera en que SONiC almacena la información.

Todo el almacenamiento de datos e información necesaria para el dispositivo es almacenado en el motor de base de datos Redis. Redis está basado en el almacenamiento de información en tablas hashes (clave/valor) y, además, es *Open Source*. La denominada redisDB de SONiC está dividida en siete bases de datos, cada una con una función específica detallada en el Anexo A.

La telemetría que se ha realizado en este trabajo está relacionada con la base de datos COUNTERS_DB. Esta tabla almacena información del tráfico que fluye por los diferentes puertos de un SONiC. Entre los modos de telemetría que SONiC permite, se ha decidido usar gNMI ya que la opción *get* de este modo permite realizar pruebas contra el servidor de telemetría de manera mucho más analítica haciendo las peticiones mucho más específicas usando *paths*. Otros modos como la subscripción, funcionan de tal manera que cuando hay cambios en redisDB se notifica al servidor gNMI automáticamente. De esta manera, conforme fluya tráfico de paquetes entre el *host1* y el *host2*, el servidor de monitorización obtendrá información de ambos *switches* que tendrán activada la telemetría.

Durante las primeras pruebas de este trabajo, se ha observado que la imagen de SONiC utilizada no soportaba la funcionalidad de telemetría dado que la tabla COUNTERS_DB no se actualizaba con el paso delo tiempo. Es por ello que, tras hacer un análisis de la situación se ha decidido añadir a redisDB una estructura de dos niveles que permitiese la telemetría. Es decir, dentro de COUNTERS_DB se han creado dos tablas hash para las dos interfaces operativas de cada *switch*: Ethernet0 y Ethernet1. Esto ha hecho necesario comprender el funcionamiento interno de redisDB en SONiC, así como los comandos que permiten la creación de nuevas tablas.

Ambas tablas han sido modeladas usando la herramienta YANG Suite de Cisco. Veamos en la siguiente imagen la estructura que se le ha dado.

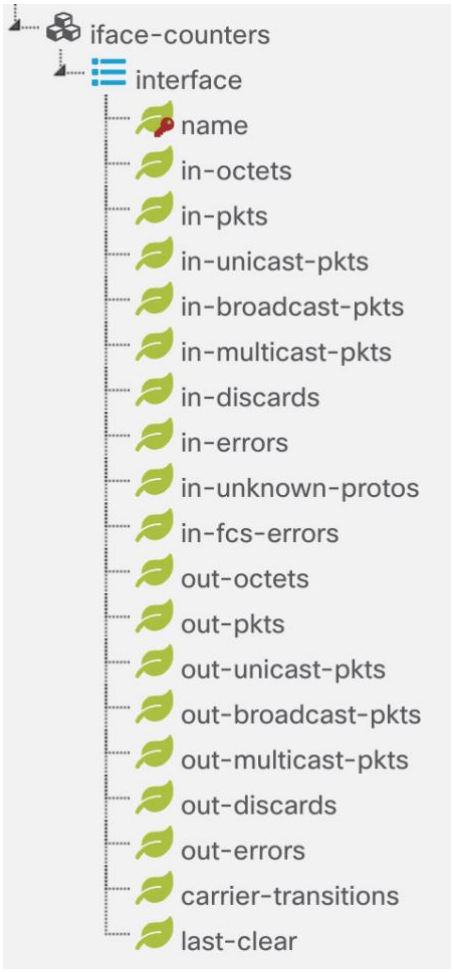

Figura 4-4: Modelo YANG de la tabla de cada interfaz para redisDB.

Se ha decidido que esta sería una estructura ideal para almacenar la información recogida del tráfico de datos por cada una de las interfaces de los *switches*, ya sea Ethernet0 o Ethernet1.

Por tanto, para paliar el imprevisto que supuso la no actualización de COUNTERS_DB por parte de SONiC, ha sido necesario generar un script adicional llamado *update_redisDB.sh* y añadirlo al sistema de ficheros de cada *switch*. Este script, el cual se ejecuta cada diez segundos en ambos *switches*, obtiene en primer lugar las estadísticas generadas por el

tráfico de red que se almacenan en la ruta */sys/class/net/Ethernet*/statistics* como se puede observar en la Figura 4-5.

```
#Getting Ethernet0 statistics (packet-in):
        in_octets_Eth0=$(cat /sys/class/net/Ethernet0/statistics/rx_bytes)
        in_pkts_Eth0=$(cat /sys/class/net/Ethernet0/statistics/rx_packets)
        in_errors_Eth0=$(cat /sys/class/net/Ethernet0/statistics/rx_errors)

#Getting Ethernet0 statistics (packet-out):
        out_octets_Eth0=$(cat /sys/class/net/Ethernet0/statistics/tx_bytes)
        out_pkts_Eth0=$(cat /sys/class/net/Ethernet0/statistics/tx_packets)
        out_errors_Eth0=$(cat /sys/class/net/Ethernet0/statistics/tx_errors)

#Getting Ethernet1 statistics (packet-in):
        in_octets_Eth1=$(cat /sys/class/net/Ethernet1/statistics/rx_bytes)
        in_pkts_Eth1=$(cat /sys/class/net/Ethernet1/statistics/rx_packets)
        in_errors_Eth1=$(cat /sys/class/net/Ethernet1/statistics/rx_errors)

#Getting Ethernet1 statistics (packet-out):
        out_octets_Eth1=$(cat /sys/class/net/Ethernet1/statistics/tx_bytes)
        out_pkts_Eth1=$(cat /sys/class/net/Ethernet1/statistics/tx_packets)
        out_errors_Eth1=$(cat /sys/class/net/Ethernet1/statistics/tx_errors)
```

Figura 4-5: Obtención de las estadísticas de red manualmente.

Se obtienen tres valores, tanto del tráfico que sale como del que llega, para ambas interfaces: *octets*, *pkts* y *errors*. Una vez que se dispone de estos valores, se actualizan las tablas hash de ambas interfaces con el comando *hset* de Redis como se comprueba en la Figura 4-6.

```
#Updating Ethernet0 in redis:
        redis-cli -n 2 hset interface:Ethernet0 in-octets $in_octets_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-pkts $in_pkts_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-unicast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-broadcast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-multicast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-discards 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-unknown-protos 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 in-fcs-errors $in_errors_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-octets $out_octets_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-pkts $out_pkts_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-unicast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-broadcast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-multicast-pkts 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-discards 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 out-errors $out_errors_Eth0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 carrier-transitions 0 > /dev/null 2>&1
        redis-cli -n 2 hset interface:Ethernet0 last-clear 0 > /dev/null 2>&1
```

Figura 4-6: Actualización de la tabla de Ethernet0 con la información de tráfico.

Por otro lado, el sistema de monitorización se conecta al servidor que se genera al activar la telemetría en cada uno de los *switches*. En la Figura 4-6 se puede ver una petición desde el nodo con el servidor de monitorización al *switch2* que tiene activada la telemetría como se puede ver con el comando de la terminal de la izquierda. Para activar la telemetría se ha seguido la descripción del proyecto SONiC [14].

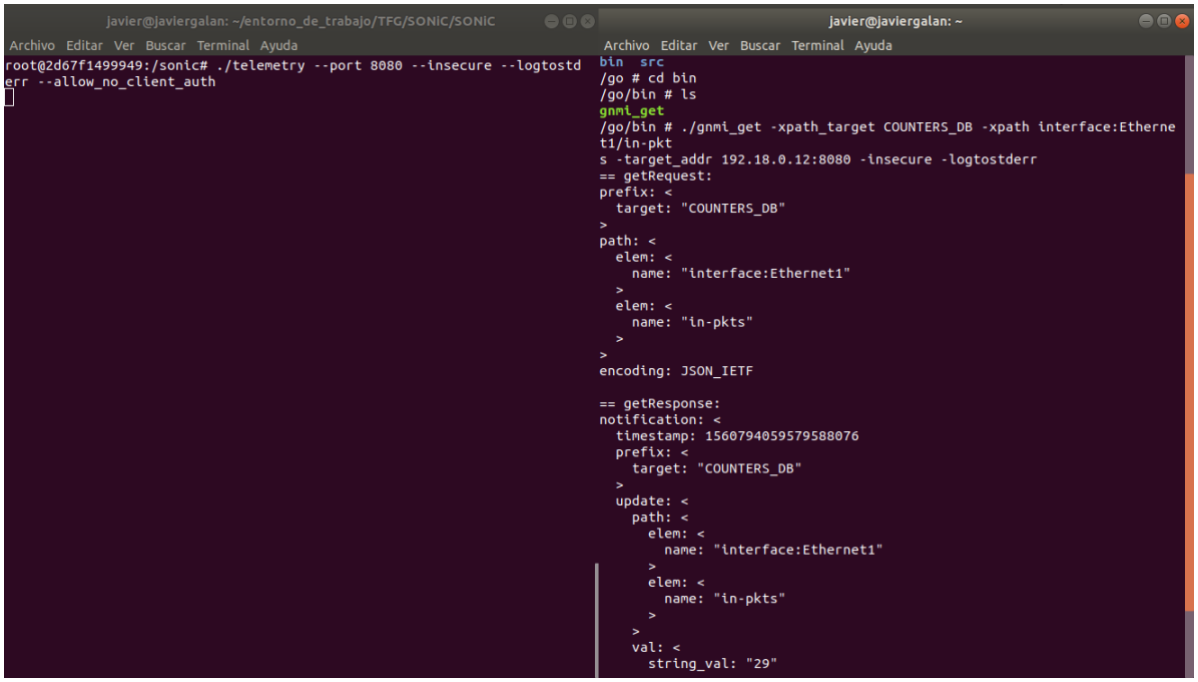

Figura 4-7: Petición para la obtención de datos por telemetría.

Analizando esta petición, vemos que desde el servidor de monitorización se debe añadir varios parámetros:

- *xpath_target*: Indica la base de datos de redisDB de SONiC a la cual se quiere acceder. En este caso, COUNTERS_DB.
- *xpath*: Indica el path en el cual se encuentra el campo que queremos resolver. En este caso, interface:Ethernet1/in-pkts. Esta petición nos devolverá los paquetes recibidos por la interfaz Ethernet1.
- *target_addr*: Indica la dirección IP del servidor gRPC junto con el puerto.
- *insecure*: Indica que no es necesario proporcionar certificados de seguridad.
- *logtostder*r: Indica que cualquier log aparecerá por la salida estándar.

Se puede observar que el valor obtenido es 29, es decir, el número de paquetes que ha recibido la interfaz Ethernet1 del *switch2*.

Con ello, se tiene finalmente un entorno de red virtual que nos permite monitorizar el tráfico de los *switches* mediante otro nodo que actúa como servidor de monitorización. Es decir, al activar la telemetría en los SONiC, se crea un servidor gRPC que tiene detrás la base de datos de SONiC redisDB. De esta manera, el usuario puede monitorizar todo cuanto contenga la base de datos de SONiC haciendo peticiones a la misma. En este trabajo, solo se monitorizarán los bytes, paquetes y errores tanto recibidos como enviados de cada *switch*.

4.4 Dockers

Cada componente del entorno de red virtual de este trabajo se encuentra en un contenedor Docker. Por ello, es necesario que cada contenedor instale en su interior las imágenes con las herramientas necesarias para la funcionalidad del nodo en cuestión.

Tanto la imagen de los *switches* como del servidor de monitorización se construyen a partir de un archivo Dockerfile, mientras que los nodos *host1* y *host2* poseen una imagen ubuntu:14.04 descargada directamente de Docker Hub. La Figura 4-8 y la Figura 4-9 muestran el contenido del fichero Dockerfile de cada *switch* y del servidor de monitorización respectivamente.

```
FROM docker-sonic-p4:latest
RUN apt-get update && apt-get install -y net-tools
```

Figura 4-8: Fichero Dockerfile_sonic.

```
FROM golang:alpine

RUN apk add git
RUN go get github.com/jipanyang/gnxi/gnmi_get
RUN go install github.com/jipanyang/gnxi/gnmi_get

EXPOSE 8080
```

Figura 4-9: Fichero Dockerfile_golang.

Se puede ver que el comando FROM indica la imagen del sistema operativo que se quiere instalar en el contenedor.

En cuanto al primer Dockerfile, se destaca la necesidad de añadir la librería *net-tools* de Linux para la configuración de la red que se detalla en el fichero start.h.

En el Dockerfile del servidor de monitorización cabe destacar que necesita una distribución *golang: alpine* que el propio Docker Hub facilita para poder ejecutar archivos ejecutables en lenguaje Go. El código del servidor de monitorización está escrito en este lenguaje como se puede comprobar en [15]. Por otro lado, se requiere la instalación el comando Git para poder descargar desde GitHub dicho código.

Una vez se definen correctamente los archivos Dockerfile, se procede a la construcción de las imágenes que irán en cada uno de los contenedores posteriormente. Esto se especifica en el fichero load_image.sh. La Figura 4-10 muestra el contenido del mismo.

```
wget https://sonic-jenkins.westus2.cloudapp.azure.com/job/p4/job/buildimage-p4-all/613/artifact/target/docker-sonic-p4.gz
sudo docker load < docker-sonic-p4.gz
sudo docker pull ubuntu:14.04

#Contruimos las imagenes atendiendo al Dockerfile
sudo docker build -f Dockerfile_sonic .
sudo docker build -f Dockerfile_golang1 . -t gnmi_client
sudo docker images
```

Figura 4-10: Fichero load_image.sh.

Primero se observa que se descarga la imagen de SONiC para los *switches* virtuales. Luego se hace un pull de Docker Hub para obtener Ubuntu:14.04, dedicado para *host1* y *host2*. Una vez se han descargado las imágenes base, se procede a construir las dos imágenes que especificadas en los Dockerfile anteriores.

Una vez llegado a este punto, se dispone de todo lo necesario para crear los contenedores de Docker. Por tanto, entra en acción el fichero start.sh. En él, se crean los cinco contenedores con sus respectivas imágenes además de configurar la red haciendo uso de *Open vSwitch*. En la Figura 4-11 aparece una parte del contenido de start.sh que permitirá aclarar la configuración de la red con Docker.

```
#!/bin/bash

#Bridge de gestion de contenedores
sudo docker network create \
    --driver bridge \
    --subnet=192.18.0.0/24 \
    --gateway=192.18.0.1 \
    --opt "com.docker.network.bridge.name"="gestion" \
    gestion

#Creacion de contenedores DOCKER
sudo docker run --net=none --privileged --entrypoint /bin/bash --name switch1 -it -d -v $PWD/switch1:/sonic docker-sonic-p4:latest
sudo docker run --net=none --privileged --entrypoint /bin/bash --name switch2 -it -d -v $PWD/switch2:/sonic docker-sonic-p4:latest
sudo docker run --net=none --privileged --entrypoint /bin/bash --name host1 -it -d ubuntu:14.04
sudo docker run --net=none --privileged --entrypoint /bin/bash --name host2 -it -d ubuntu:14.04

#Creacion de contenedor con gnmi_get
sudo docker run --privileged --entrypoint /bin/sh --name gnmicli -it -d gnmi_client

sudo iftobridge add-link mgmt1 switch1 gestion --sip="192.18.0.11/24"
sudo iftobridge add-link mgmt1 switch2 gestion --sip="192.18.0.12/24"
sudo iftobridge add-link mgmt1 gnmicli gestion --sip="192.18.0.10/24"

#Creamos los puentes que conectan los dispositivos virtuales (hosts y switches):
#Creamos un bridge nuevo
sudo ovs-vsctl add-br switch1_switch2
#Anadimos los puertos correspondientes y conectamos los containers con OVS bridge
sudo ovs-docker add-port switch1_switch2 sw_port0 switch1
sudo ovs-docker add-port switch1_switch2 sw_port0 switch2

#Creamos un bridge nuevo
sudo ovs-vsctl add-br host1_switch1
#Anadimos los puertos correspondientes y conectamos los containers con OVS bridge
sudo ovs-docker add-port host1_switch1 sw_port1 switch1
sudo ovs-docker add-port host1_switch1 eth1 host1
```

Figura 4-11: Fichero start.sh

En primer lugar, se crea un nuevo bridge de Docker para conseguir la conectividad entre el servidor de monitorización y los dos *switches*. A continuación, se crean los cinco contenedores con el comando *docker run.* Y el resto del archivo, usando *Open vSwitch* construye la red creando bridges entre unos nodos y otros y conectando las interfaces correspondientes.

Por tanto, es importante destacar que el uso de Docker permite la automatización de la creación del entorno. A continuación, veremos cómo puede encajar todo el trabajo en un gestor de integración continua.

4.5 Travis CI

Una vez se ha explicado con detalle el desarrollo del entorno de red virtual que se ha llevado a cabo en este proyecto, se procede a su integración con Travis CI para llevar un desarrollo continuo del mismo y permitir el desarrollo de test que validen el correcto funcionamiento del código.

Para comenzar, se recuerda que es necesario crear una cuenta tanto en Travis CI como en GitHub, ya que Travis CI tiene la ventaja de que es capaz de trabajar directamente con repositorios públicos de GitHub de manera gratuita.

Una vez que se tiene el repositorio en GitHub, podemos generar *builds* desde Travis CI sí y solo si se dispone del fichero *travis.yml.* Este fichero contiene, como se ha explicado en la sección de diseño, las directrices a seguir para generar (*build*) el contexto deseado sobre el que realizar las pruebas al proyecto en cuestión.

La figura 4-12 muestra la interfaz principal de Travis CI.

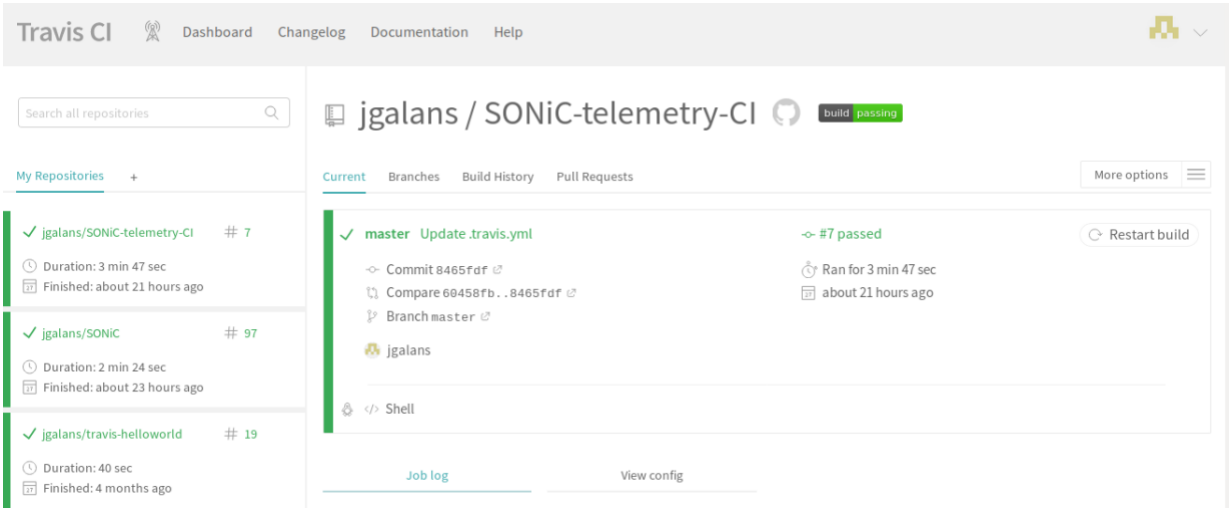

Figura 4-12: Interfaz de usuario de Travis CI.

En este caso, se puede observar que a la izquierda tenemos los repositorios de GitHub abiertos en Travis. En el caso de tener un color rojo indicarían que el último *build* no fue exitoso. En este caso, los último *builds* tuvieron éxito por lo que se muestran de color verde. Por otro lado, en la imagen nos encontramos dentro del *dashboard* del repositorio en el cual está guardado este trabajo: SONIC-telemetry-CI. Ahí se pueden ver valores como la última ejecución o la rama del repositorio sobre la cual estamos generando el entorno entre otros.

Para ejecutar un *build* se pulsa el botón *Restart build* de arriba a la derecha y estará correcto si el valor de salida es 0.

4.6 Conclusión

En este capítulo se ha podido ver las decisiones tomadas para la implementación del diseño marcado por el capítulo anterior. Además, es importante destacar el problema obtenido con la base de datos COUNTERS_DB de SONiC, la cual no se actualizaba cuando se generaba tráfico en la red. La solución tomada para paliar el problema ha sido el desarrollo de un proceso que actualiza la información de redisDB en base a la información del sistema, que ha permitido que el funcionamiento de la telemetría sea correcto. No obstante, se deja para trabajos futuros la integración del código con el proyecto SONiC para ver cómo resolverlo de forma estable. El desarrollo de los nodos en contenedores ha sido bastante exitoso y se ha comprobado que Docker es una herramienta adecuada y a la vez ligera para la virtualización de entornos. Finalmente, se ha explicado claramente la integración entre Travis CI y GitHub que optimiza enormemente las labores de integración continua de proyectos alojados en repositorios de esta plataforma como es el caso de este trabajo. Con ello, se ha desarrollado un sistema de verificación y validación de un entorno de red.

5 Integración, pruebas y resultados

5.1 Introducción

En esta sección se comentará la integración del trabajo realizado en Travis CI de una manera más detallada, así como las diferentes pruebas realizadas al código y los resultados obtenidos para la validación del proyecto y demostración de la metodología de integración continua.

5.2 Demostración del entorno de verificación y validación para redes

Desde el principio, se tiene la premisa de seguir un desarrollo continuo que permita la verificación y validación de un entorno de red virtual el cual sea posible monitorizar gracias a las prestaciones del sistema operativo de red SONiC.

Para conseguir este objetivo se ha hecho uso del motor que facilita Travis CI para generar pruebas en los entornos que el usuario le indique. Por tanto, Travis CI debe ser quien tiene que asegurar si los principales módulos del proyecto funcionan correctamente o no. La siguiente imagen muestra un gráfico de representación del ciclo que debe llevar un proyecto para seguir las pautas de la integración continua de software haciendo uso de Travis CI.

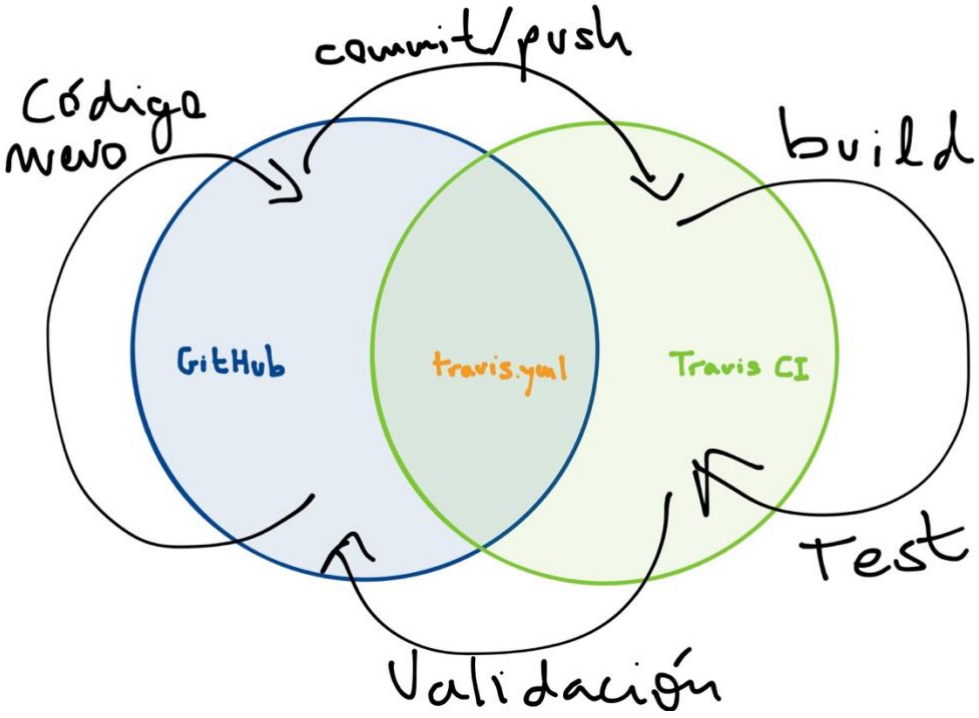

Figura 5-1: Ciclo de la integración continua de software con Travis CI.

Se observa que para que el ciclo tenga valor, la parte de pruebas es de vital importancia. Es decir, realizar las pruebas adecuadas sobre el código de un proyecto, como se comentaba en el final de la sección de desarrollo, es una parte esencial de la integración continua.

Además, en aquellos proyectos que deban funcionar multiplataforma o en entornos con sistemas operativos diferentes, la parte de *build* es muy importante. Es el momento en el cual el producto sale del entorno de desarrollo y se pone a prueba con las condiciones del entorno de producción.

Por ejemplo, un cliente pide a una empresa que desarrolle una aplicación que deba funcionar en todas las plataformas móviles. Los desarrolladores de software, por tanto, deben generar entornos idénticos a los de producción, con herramientas como Travis CI, para poder verificar que el proyecto no tiene errores y puede funcionar independientemente de la plataforma sobre la que se utilice la aplicación.

5.1 Evaluación de la conectividad en el entorno de pruebas

Para ver que el entorno desarrollado tiene conectividad y se genera tráfico entre los nodos existentes se realizará una prueba que lo valide. El siguiente código genera una prueba que demuestra la conectividad extremo a extremo entre los distintos nodos de la topología de red.

```
#!/bin/bash
#Ping desde host2 a switch2
sudo docker exec -it host2 ping 192.168.2.1 -c10
sleep 2

#Ping desde host1 a host2
sudo docker exec -it host1 ping 192.168.2.2 -c10
```

Figura 5-2: Fichero test.sh.

Esta prueba consiste en realizar un ping desde el *switch2* al *host2* y luego un ping desde el host1 hasta el *host2*. Como se muestra en la Figura 5-3 el resultado mostrado por Travis CI afirma que el funcionamiento de la red es adecuado y, por tanto, hay conectividad.

```
548 64 bytes from 192.168.2.1: icmp_seq=7 ttl=64 time=2.64 ms
549 64 bytes from 192.168.2.1: icmp_seq=8 ttl=64 time=5.19 ms
550 64 bytes from 192.168.2.1: icmp_seq=9 ttl=64 time=4.05 ms
551 64 bytes from 192.168.2.1: icmp_seq=10 ttl=64 time=3.76 ms
552
553 --- 192.168.2.1 ping statistics ---
554 10 packets transmitted, 10 received, 0% packet loss, time 9011ms
555 rtt min/avg/max/mdev = 2.522/5.759/19.279/4.611 ms
556 PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
557 64 bytes from 192.168.2.2: icmp_seq=1 ttl=62 time=19.6 ms
558 64 bytes from 192.168.2.2: icmp_seq=2 ttl=62 time=9.59 ms
559 64 bytes from 192.168.2.2: icmp_seq=3 ttl=62 time=7.98 ms
560 64 bytes from 192.168.2.2: icmp_seq=4 ttl=62 time=8.27 ms
561 64 bytes from 192.168.2.2: icmp_seq=5 ttl=62 time=8.12 ms
562 64 bytes from 192.168.2.2: icmp_seq=6 ttl=62 time=10.4 ms
563 64 bytes from 192.168.2.2: icmp_seq=7 ttl=62 time=11.2 ms
564 64 bytes from 192.168.2.2: icmp_seq=8 ttl=62 time=9.59 ms
565 64 bytes from 192.168.2.2: icmp_seq=9 ttl=62 time=10.4 ms
566 64 bytes from 192.168.2.2: icmp_seq=10 ttl=62 time=8.52 ms
567
568 --- 192.168.2.2 ping statistics ---
569 10 packets transmitted, 10 received, 0% packet loss, time 9010ms
570 rtt min/avg/max/mdev = 7.986/10.385/19.698/3.280 ms
571 The command "./test.sh" exited with 0.
572
573
574 $ ./stop.sh                                                    after_script  3.07s
584
585 Done. Your build exited with 0.
```

Figura 5-3: Resultado de Travis CI.

Por esta razón, la integración continua es un método que permite cerciorar al desarrollador de un proyecto que las nuevas modificaciones en el código no han provocado fallos en la funcionalidad de ningún módulo. Por ello, una buena calidad en las pruebas de un proyecto puede evitar la aparición de errores en un futuro.

5.2 Evaluación de la telemetría en el entorno de pruebas

A continuación, se realizarán varias pruebas en uno para validar el sistema de monitorización de los *switches* del entorno de red. En primer lugar, se muestra el código del fichero de pruebas que recibe Travis CI para, una vez desplegado el entorno, probar la funcionalidad del módulo de telemetría de ambos *switches*.

Se realizan pings entre *host* y *switch* de la misma subred, es decir, *host1* con *switch1* y *host2* con *switch2*. Lo que se desea comprobar es que el servidor de monitorización sea capaz de obtener resultados coherentes al tráfico que ha tenido lugar.

```
#!/bin/bash

sudo docker exec -it host2 ping 192.168.2.1 -c30

sudo docker exec -it host1 ping 192.168.1.1 -c31

sleep 20

#Monitorización switch1
sudo echo "Paquetes recibidos por Ethernet 1 en switch1"
sudo docker exec -it gnmicli ./bin/gnmi_get -xpath_target COUNTERS_DB -xpath interface:Ethernet1/in-pkts -target_addr 192.18.0.11:8080 -insecure -
logtostderr

#Monitorización switch2
sudo echo "Paquetes recibidos por Ethernet 1 en switch2"
sudo docker exec -it gnmicli ./bin/gnmi_get -xpath_target COUNTERS_DB -xpath interface:Ethernet1/in-pkts -target_addr 192.18.0.12:8080 -insecure -
logtostderr
```

Figura 5-4: Test para verificar la monitorización de los *switches*.

Por tanto, se debería obtener que el *switch1* ha recibido treinta paquetes mientras que el *switch2* ha recibido treinta y uno.

```
Paquetes recibidos por Ethernet 1 en switch1
== getRequest:
prefix: <
  target: "COUNTERS_DB"
>
path: <
  elem: <
    name: "interface:Ethernet1"
  >
  elem: <
    name: "in-pkts"
  >
>
encoding: JSON_IETF

== getResponse:
notification: <
  timestamp: 1560979350769765599
  prefix: <
    target: "COUNTERS_DB"
  >
  update: <
    path: <
      elem: <
        name: "interface:Ethernet1"
      >
      elem: <
        name: "in-pkts"
      >
    >
    val: <
      string_val: "30"
    >
  >
>
```

Figura 5-5: Resultado de la monitorización por telemetría del switch1.

```
Paquetes recibidos por Ethernet 1 en switch2
== getRequest:
prefix: <
  target: "COUNTERS_DB"
>
path: <
  elem: <
    name: "interface:Ethernet1"
  >
  elem: <
    name: "in-pkts"
  >
>
encoding: JSON_IETF

== getResponse:
notification: <
  timestamp: 1560979351009456433
  prefix: <
    target: "COUNTERS_DB"
  >
  update: <
    path: <
      elem: <
        name: "interface:Ethernet1"
      >
      elem: <
        name: "in-pkts"
      >
    >
    val: <
      string_val: "31"
    >
  >
>
```

Figura 5-6: Resultado de la monitorización por telemetría del switch2

Se puede comprobar que el sistema de monitorización extrae datos, en este caso, paquetes entrantes, de forma adecuada lo que verifica la funcionalidad del mismo.

Finalmente, concluimos que se ha realizado una prueba satisfactoria que nos valida la adecuada monitorización de ambos *switches* permitiendo extraer estadísticas de los mismos.

5.3 Conclusión

En este último capítulo se ha podido comprobar que una buena herramienta de integración continua como es Travis CI permite desplegar entornos específicos para realizar pruebas al código del proyecto. Además, este capítulo también pone de manifiesto la importancia de las tres últimas fases del ciclo de vida del software detalladas al final del capítulo primero. Tanto la integración como la verificación y validación del producto juega un papel importante para conseguir los objetivos marcados en la fase de obtención de requisitos. Por ello, las pruebas que se realicen al código en estas fases que se han comentado son determinantes a la hora de valorar la calidad del desarrollo.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Llegado a este punto, atendiendo a los objetivos planteados al principio del trabajo, se comprobará el cumplimiento de cada uno de ellos extrayendo las conclusiones oportunas. Los objetivos del trabajo eran:

1. Estudiar y analizar el marco de trabajo de los sistemas operativos de red de código abierto, sus módulos y herramientas para conocer y ampliar su funcionalidad.
2. Estudiar y analizar el estado del arte del control de *White-Boxes* usando interfaces programáticas.
3. Desarrollar una metodología de verificación y validación para entornos de red.
4. Demostrar y experimentar la metodología definida con un entorno de red emulado usando un sistema operativo de red de código abierto.

En primer lugar, se ha estudiado el marco actual de los sistemas operativos de red *Open Source* comprendiendo los diferentes módulos que les permiten establecerse en cualquier dispositivo *Open Hardware*. Se ha analizado el estado del arte del control de *White-Boxes* que han facilitado el posterior diseño y desarrollo, permitiendo interiorizar conceptos importantes como la separación lógica del plano de datos (software) y el plano de control (hardware). Se ha conseguido una metodología de verificación y validación para entornos de red basada en contenedores de Docker y el software de gestión de integración continua Travis CI. Finalmente ha sido posible experimentar esta metodología a un entorno de red emulado usando un sistema operativo de red *Open Source* como es SONiC.

Más allá de haberse cumplido los objetivos planteados al principio de este trabajo, cabe desatacar la importancia de ciertas asignaturas de la carrera que han permitido trabajar de manera más eficiente durante el proyecto. Entre ellas, se destacan Ingeniería del Software, la cual ha facilitado los encuentros con el tutor para obtener de manera eficaz tanto los requisitos funcionales como los no funcionales. Además, no hubiera sido posible, en ciertas ocasiones, entender algunos conceptos relacionados con la topología del entorno de red sin haber cursado y adquirido adecuadamente los conocimientos de Redes I y Redes II. Por otro lado, y no menos importante, Sistemas Operativos ha sido una asignatura decisiva para comprender el funcionamiento de los contenedores de Docker, los cuales podían tener distribuciones diferentes. También es importante destacar que la capacidad de trabajo y esfuerzo adquirida durante las prácticas de cada una de las asignaturas de la carrera ha facilitado enormemente la realización de este trabajo.

Finalmente, y a modo personal, este trabajo ha sido un gran reto dado que muchos conocimientos técnicos sobre redes eran bastante complejos y es por ello que tanto la ayuda del tutor como la constante investigación han permitido también que el trabajo pudiese salir adelante.

6.2 Trabajo futuro

La integración continua llevada a cabo en este proyecto permitiendo la validación de un entorno de red virtual y la monitorización del tráfico del mismo tiene gran escalabilidad. Este proyecto puede extenderse a redes mucho más grandes facilitando el trabajo de la validación y verificación de los cambios que se vayan produciendo en ellas. El crecimiento

exponencial de la tecnología hoy en día no permite al desarrollador anclarse en sus conocimientos, sino que motiva a seguir creciendo e innovando. Para los entornos de red no es menos, y por tanto la integración continua se puede implantar como un modelo casi imprescindible que permitirá validar los proyectos ante los nuevos cambios tecnológicos que aparezcan de una manera eficaz y óptima.

En un futuro se podría retomar este proyecto para probar las prestaciones que ofrecen tanto otros sistemas operativos de red *Open Source* como otros gestores de integración continua, viendo las ventajas y desventajas en la práctica.

Referencias

[1] V. López, O. González, J.P. Fernández-Palacios: ***Whitebox Flavors in Carrier Networks,*** in Optical Fiber Conference (OFC), Marzo 2019.

[2] M. Bjorklund, ***The YANG 1.1 Data Modeling Language***, Dirección web: https://tools.ietf.org/html/rfc7950, agosto 2016.

[3] R. Enns, M. Bjorklund, J. Schoenwaelder, A. Bierman, ***Network Configuration Protocol***, Dirección web: https://tools.ietf.org/html/rfc6241

[4] A médium corporation, ***Network automation and the Rise of NETCONF***, Dirección web: https://medium.com/@k.okasha/network-automation-and-the-rise-of-netconf-e96cc33fe28, Último acceso: Mayo 2019

[5] A. Bierman, M. Bjorklund, K. Watsen, ***RESTCONF protocol,*** Dirección web: https://tools.ietf.org/html/rfc8040***,*** enero 2017.

[6] Cisco Systems ***Linux on Network Switch and Management,*** Dirección web: https://slideplayer.com/slide/13872033/. Último acceso: Mayo 2019

[7] Cisco, ***gNMI Protocol,*** Dirección web: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/prog/configuration/169/b_169_programmability_cg/gnmi_protocol.pdf. Último acceso: Junio 2019

[8] Paul Borman, Marcus Hines, Carl Lebsack, Chris Morrow, Anees Shaikh, Rob Shakir, ***Grpc Network Management Interface (gNMI)***, Dirección web: https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md#35-subscribing-to-telemetry-updates , enero 2018

[9] AT&T, ***Towards an Open, Disaggregated Network Operating System***. Dirección web: https://about.att.com/ecms/dam/innovationblogdocs/att-routing-nos-open-architecture_FINAL%20whitepaper.pdf. Último acceso: Junio 2019

[10] Microsoft Azure, Dirección web: http://github.com/Azure/SONiC/wiki. Último acceso: Junio 2019

[11] Microsoft Azure, ***Architecture***, Dirección web: https://github.com/Azure/SONiC/wiki/Architecture. Último acceso: Junio 2019

[12] Microsoft Azure, ***¿Qué es una máquina virtual?***, Dirección web: https://azure.microsoft.com/es-es/overview/what-is-a-virtual-machine/. Último acceso: Junio 2019

[13] Git Hub, ***Azzure Sonic***, Dirección web: https://sonic-jenkins.westus2.cloudapp.azure.com/job/p4/job/buildimage-p4-all/613/. Último acceso: Junio 2019

[14] Git Hub, ***SONiC Grpc data telemetry***, Dirección web: https://github.com/Azure/sonic-telemetry/blob/master/doc/grpc_telemetry.md. Último acceso: Junio 2019

[15] Git Hub, ***Gnmi Get***, Dirección web: https://github.com/jipanyang/gnxi/tree/master/gnmi_get. Último acceso: junio 2019

Anexos

A Estructura de redisDB en SONiC.

Esta tabla describe la estructura de la base de datos que orquesta un dispositivo de red SONiC. Es una base de datos redis la cual está formada por tablas hash siguiendo el modelo clave valor. Se puede ver con más detalle en el repositorio GitHub del proyecto SONiC [14].

Nombre	Número	Descripción
APPL_DB	0	Datos de aplicación
ASIC_DB	1	Datos de estado y configuración de ASIC
COUNTERS_DB	2	Contadores de datos para puertos, colas, lag
LOGLEVEL_DB	3	Log para de los diferentes módulos
CONFIG_DB	4	Núcleo de configuración
FLEX_COUNTER_DB	5	Contadores PFC y otras extensiones
STATE_DB	6	Configuración del estado para objetos en CONFIG_DB