PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

DISEÑO E IMPLEMENTACIÓN DE UN EMULADOR DE REDES

Tesis para optar el Título de Ingeniero de las Telecomunicaciones, que presentan los bachilleres:

Rubén Francisco Córdova Alvarado

Antonio de Jesús Merino Gala

ASESOR: César A. Santiváñez, Ph. D.

Lima, marzo del 2017

Resumen

El trabajo desarrollado en la presente tesis consiste en el diseño e implementación de un emulador de redes de alta capacidad, como entorno de pruebas de nuevas tecnologías previo su despliegue. El emulador está conformado por módulos implementados usando el lenguaje de programación Python. A largo plazo, se desea integrar el presente emulador en un *rack* de alta capacidad (con enlaces de 10 Gbps), el cual se encuentra en el laboratorio del Grupo de Investigación en Redes Avanzadas (GIRA) de la Pontificia Universidad Católica del Perú. En dicho *rack*, se ha instalado el *software* de *Cloud Computing* OpenStack, el cual emplea diferentes servicios para la creación de las máquinas virtuales a usar en el emulador. El objetivo principal del emulador es que las pruebas realizadas en él presenten una alta fidelidad: no solo se desea capturar el comportamiento de la red al limitar la tasa de bits o introducir latencia en un enlace, sino también evitar generar fenómenos espurios –como pérdida de paquetes– debido a las limitaciones de procesamiento del *hardware* sobre el cual funciona el emulador. Por ello, el emulador incluye un proceso de calibración del *hardware* subyacente, así como un módulo de validación de recursos (p.ej. RAM, *cores*, etc.) para asegurar que el emulador puede soportar la topología de interés.

En el primer capítulo, se describe la situación actual de las redes —específicamente la de los proveedores de servicios— y se identifican los problemas que surgen con la evolución y desarrollo de nuevos servicios. Asimismo, se presentan algunas tecnologías emergentes como posibles soluciones a los problemas mencionados.

En el segundo capítulo, se presentan los tipos de entornos de prueba que se utilizan para evaluar distintos prototipos de redes; en especial, los que proponen las tecnologías emergentes. Adicionalmente, se realiza un análisis de las ventajas y desventajas de cada entorno y se determina cuál es más conveniente para los objetivos del presente trabajo. En la última parte del capítulo, se presentan los objetivos del trabajo de tesis, en base a la problemática y al entorno de prueba determinado.

El tercer capítulo está orientado a las tecnologías de virtualización. Primero, se introduce el concepto de virtualización y se presenta el modelo de *Cloud Computing*.

Luego, se desarrollan las áreas de virtualización, mostrando las diversas técnicas existentes en cada una y la necesidad de monitorear el estado de la infraestructura. Por último, se realiza una descripción de OpenStack, dado que es el *software* empleado en el presente trabajo de tesis.

En el cuarto capítulo, se detalla el diseño completo del emulador. Se inicia con las consideraciones y requerimientos del emulador; a continuación, se presenta la arquitectura del emulador, indicando sus módulos y el funcionamiento de cada uno; luego, se detallan los cambios realizados a OpenStack, seguido del módulo de validación del emulador –que lo diferencia de los demás existentes; finalmente, se muestra la interfaz de usuario para la creación de los experimentos.

En el capítulo final, se presentan los resultados obtenidos al realizar pruebas en el emulador. Estas pruebas tienen la finalidad de demostrar que el emulador funciona en base a los objetivos y requerimientos planteados. También se muestra el análisis realizado en la etapa de calibración, en donde se obtiene la cantidad de recursos requeridos por cada nodo.

Por último, se presentan las conclusiones obtenidas de la implementación y de las pruebas desarrolladas, basadas en los objetivos de la tesis y los requerimientos del emulador, así como las recomendaciones sugeridas para un mejor desempeño del mismo.

Dedicatoria

A nuestros padres y hermanas.

A nuestro asesor.

Agradecimientos

A Dios, porque sin él nada sería posible. A mis padres, por el apoyo incondicional y por su amor. A mi hermana, por siempre creer en mí. A mis amigos, por los momentos amenos durante la carrera. Al Prof. Santiváñez, por el apoyo y exigencia en todo momento. A Antonio, por el gran trabajo en equipo y la divertida experiencia de hacer una tesis.

Ruben

A Dios y a la Virgen María Auxiliadora, por haberme acompañado a través de este largo camino. A mis padres, por darme el apoyo y la fuerza necesaria para seguir adelante. A mi hermana, por los constantes consejos y por creer en mí siempre. A mis familiares, por sus palabras de aliento y consejos. A todos los amigos que conocí durante esta etapa de mi vida. Al profesor Santiváñez, por su asesoría durante el desarrollo de la tesis. Finalmente, a Rubén, por el excelente trabajo en equipo, la paciencia necesaria y por los divertidos y emocionantes momentos que acontecieron durante toda esta experiencia de realizar una tesis, fue un placer haber trabajado contigo y espero poder volverlo hacerlo en un futuro.

Antonio

Índice

Lista de Figuras

Lista de Tablas

Introducción

En la actualidad, las redes de datos han tomado un rol importante, debido a la creciente dependencia de las personas por las aplicaciones que van apareciendo. Sin embargo, las tecnologías de gestión de redes no han seguido el ritmo de la constante y rápida evolución de estos servicios. Adicionalmente, los fabricantes de equipos emplean tecnologías y *features* propietarios para su plano de control, generando una mayor complejidad en el despliegue y operación de entornos *multivendor*. Ante ello, están surgiendo nuevas arquitecturas de redes, con propuestas innovadoras, capaces de soportar los servicios emergentes. Debido a que estas nuevas propuestas no generan la confianza para implementarlas en la infraestructura de los proveedores de servicios, es necesario que las nuevas arquitecturas sean evaluadas y validadas en entornos de prueba (también llamados *testbeds*).

Entre los 3 tipos de *testbeds* existentes (físicos, emuladores y simuladores), los entornos físicos presentan un alto costo de implementación, mientras que los simuladores presentan baja fidelidad en sus resultados; los emuladores —que implementan el código usado por los equipos sobre *hardware* virtualizado— presentan un punto de equilibrio entre costo y fidelidad. No obstante, los emuladores tradicionales sufren una pérdida de fidelidad al ser sometidos a demandas muy altas de tráfico: al permitir que se saturen los recursos de los servidores sobre los cuales son ejecutados, se introducen fenómenos artificiales (p.ej. pérdida adicional de paquetes, aumento de latencia en equipos, etc.) que alteran los resultados obtenidos.

El objetivo del presente trabajo es el diseño e implementación de un emulador de alta fidelidad de redes de alta capacidad, en donde se pueda evaluar el comportamiento de nuevas tecnologías. Para el diseño, se toma como base un *rack* de servidores corriendo el *software* de OpenStack. Por otro lado, el emulador provee una interfaz de usuario para definir y cargar topologías, visualizarlas y definir las características de los enlaces (velocidad de transmisión y retardo). Además, incluye un módulo de validación para evitar que excesiva demanda de recursos disminuya la fidelidad de los resultados obtenidos.

1. Drivers de nuevas tecnologías de redes

A lo largo de los años, las redes de datos y las aplicaciones han ido evolucionando, lo que ha originado diversos problemas en la red. Estos problemas abarcan temas como la complejidad de operación de la red, inconsistencias de configuración, soluciones incluidas en *releases* de proveedores, ataques de seguridad cada vez más sofisticados, inconsistencias en las políticas de seguridad, entre otros. Para explicarlos con mayor detalle, se delimita el análisis en las redes de los proveedores de servicios y de Internet.

Las redes de los proveedores de servicios poseen equipos distribuidos en varias zonas geográficas. Dichas redes son modificadas constantemente, para mejorar el servicio brindado o para abarcar nuevas zonas. Estas modificaciones implican que se realicen cambios en la configuración de ciertos equipos. Al realizar cambios en una gran cantidad equipos, se requiere tiempo y cuidado de no introducir errores de configuración que afecten los servicios desplegados. Por ejemplo, la configuración de políticas de seguridad (*access lists*) en los equipos requiere de un cuidado

especial, ya que la omisión de reglas en las políticas de seguridad podría generar puntos vulnerables en la red. Adicionalmente, estas redes están conformados por equipos de diversas marcas (Cisco, Alcatel, Huawei) con diferentes interfaces de control y configuración (algunos de ellos propietarios).

Por otro lado, con el transcurso de los años, Internet se ha vuelto cada vez más popular y, debido al avance tecnológico, ha aparecido una gran cantidad de servicios sobre ella. Algunos de estos servicios manejan información sensible de los usurarios (p.ej. consultas de Banca por Internet). Si bien se pueden implementar mecanismos de protección de la información (como VPNs), los ataques de seguridad se vuelven cada vez más sofisticados. Además, ciertas nuevas aplicaciones requieren de mayor ancho de banda, mínima pérdida de paquetes, bajo retardo y *jitter*; las redes actuales no pueden no pueden satisfacer estas exigencias, debido a que el enrutamiento se realiza en base a direcciones IP y no al tipo de servicio.

Los problemas anteriormente mencionados se pueden clasificar en base a tres limitantes principales: operatividad, seguridad, y flexibilidad. En las siguientes secciones, se desarrollarán dichas limitantes de las redes, así como las nuevas arquitecturas de redes como posibles alternativas a la situación actual.

1.1. Límites actuales

Los problemas de la complejidad en la operación de las redes, inconsistencia en las políticas de seguridad, soluciones incluidas en *releases* de proveedores, ataques de seguridad sofisticados y servicios que requieren un trato diferenciado se pueden agrupar en base a tres limitantes: operatividad, seguridad y flexibilidad.

1.1.1. Operatividad

Las redes actuales —por ejemplo, las redes de los proveedores de servicios— cuentan con gran cantidad de equipos distribuidos e interconectados entre sí. En su mayoría, estos equipos son *routers* y tienen como principal función el reenvío de paquetes, con tráfico de usuarios, hacia un destino en base a su dirección IP.

Los *routers* cuentan con dos partes principales: *hardware* (puertos de red, procesadores, *buffers* y *switch fabric*) y *software* (sistema operativo). Los

componentes del *hardware* se encargan de procesar rutas y reenviar paquetes —principales funciones de un *router*— mientras que el sistema operativo permite administrar la memoria del equipo, controlar dispositivos de entrada y salida, entre otras funciones.

Cuando se necesita realizar un cambio en la configuración del equipo, se debe hacer uso de su interfaz de configuración. Sin embargo, al existir varios *routers* en la red, esta tarea requiere de tiempo (debido al número de equipos) y una planificación adecuada para no introducir errores de configuración. Este inconveniente nace cuando se desplegaron las primeras redes: ellas se concibieron como estáticas y, en caso se modificase su topología, el cambio era mínimo [1].

Para asistir en la labor de configurar varios nodos, existen algunos gestores de red que, además de posibilitar el monitoreo y control los recursos de los nodos de la red, ofrecen *features* de *Config Push* (Alcatel-Lucent SAM, Cisco Prime). Con estos gestores, se pueden aplicar — en base a *scripts*— la misma configuración a múltiples equipos simultáneamente. Si bien esta característica permite que la operación de la red sea menos compleja, no es posible introducir configuraciones diferentes en los equipos.

1.1.2. Seguridad

En los últimos años, han aparecido gran cantidad de servicios sobre Internet. Estos servicios están orientados a satisfacer diferentes necesidades de los usuarios; por ello, que cada servicio maneja distinto tipo de información.

Un caso en particular que se puede mencionar es la aplicación de Banca por Internet. Este tipo de aplicación mantiene una interacción directa con el usuario, a través del intercambio de información sensible. Si bien se han implementado mecanismos para proteger dicha información, los ataques de seguridad son cada vez más sofisticados. Esta situación está generando desconfianza entre los usuarios con respecto a la seguridad de los servicios.

Por otro lado, las redes de los proveedores cuentan con gran cantidad de equipos dentro de su infraestructura; dichos equipos están interconectados y distribuidos de tal manera que cubren varias zonas geográficas. Para brindar una mejor calidad de servicios y cubrir nuevas zonas, es necesario que la configuración de los equipos se

modifique constantemente. Estas modificaciones también incluyen actualizaciones en las políticas de seguridad para reducir los riesgos de filtración de información sensible. La modificación y configuración de las políticas de seguridad —las cuales contienen varias reglas— se realizan de forma manual; por esta razón, un error en la configuración de dichas políticas puede generar puntos vulnerables en la red, que podrían ser aprovechados por los *hackers*.

1.1.3. Flexibilidad

Los proveedores —que brindan diferentes tipos de servicios— cuentan con una infraestructura de gran número de equipos, algunas veces de diferentes fabricantes (Cisco, Alcatel Huawei). Esta diversidad de equipos, entre otros aspectos, se debe a los *features* adicionales que provee cada fabricante, ya que pueden ser de utilidad en diferentes circunstancias.

Todos los equipos de la red interactúan correctamente debido a que los protocolos del plano de control se rigen bajo estándares; sin embargo, los *features* adicionales son desarrollados para operar solo con el *hardware* y sistema operativo (OS) propietarios. Es por ello que, de presentarse un *bug* en los equipos, se debe esperar hasta que el fabricante ponga a disposición un *release* o parche del sistema operativo donde soluciona el inconveniente.

Asimismo, las redes de los proveedores emplean un modelo tradicional para el reenvío de paquetes, utilizando las direcciones IP destino: el enrutamiento de este modelo no opera en base al tipo de tráfico que transmite, sino envía los paquetes al siguiente salto buscando la dirección IP destino en su *forwarding table*.

No obstante, existen nuevas aplicaciones cuyos requerimientos de la red (p.ej. mayor ancho de banda, mínima pérdida de paquetes, bajo retardo y *jitter*, etc.) no pueden ser satisfechas de forma nativa por las redes actuales; estas aplicaciones requieren que el camino óptimo se elija en base al tipo de servicio transportado, no a la dirección IP destino (p.ej. el tráfico de videoconferencia vaya por un camino diferente al tráfico de correo).

1.2. Nuevas arquitecturas

Con el paso del tiempo —a pesar que las arquitecturas de las redes actuales no

fueron diseñadas para soportar las nuevas aplicaciones y servicios— sí han podido hacerlo, debido a las modificaciones realizadas en la red para cumplir sus requerimientos; empero, no es posible determinar hasta cuándo resultarán sostenibles estos métodos. Es por ello que, ante las nuevas exigencias, se ha orientado la perspectiva del cambio hacia el diseño de nuevas arquitecturas. En las siguientes secciones, se muestran diferentes modelos de redes como posibles soluciones a las limitaciones de las redes de hoy en día: *Future Internet Architecture*, *SoftwareDefined Networking* y *ESnet/ OSCARS*.

1.2.1. *Future Internet Architecture*

Future Internet Architecture (FIA) [4] es un programa lanzado por la dirección de *Computer and Information Science and Engineering* del *National Science Foundation*. Este programa busca promover la investigación, diseño y evaluación de futuras arquitecturas de Internet, para que sean innovadoras y confiables. La motivación de este programa se basa en que el Internet del futuro será diferente al que conocemos actualmente, debido a los avances tecnológicos y a las aplicaciones emergentes —incluyendo las que están por implementarse. Las propuestas no deben enfocarse en realizar cambios al Internet vigente, sino en diseñar nuevas arquitecturas de red que cumplan con los requisitos y retos, actuales y futuros.

Como se menciona en su página web [2], FIA está formado por cinco proyectos: *Named Data Networking*, *MobilityFirst*, NEBULA, *eXpressive Internet Architecture* y *ChoiceNet.*

El primer proyecto propuesto se denomina *Named Data Networking* (NDN); este consiste en una arquitectura de red que se centra en el contenido de aplicaciones, en vez de las direcciones IP de origen y destino. NDN está orientado a la seguridad de la información, en lugar de la seguridad en el canal de comunicación. Este enfoque permite crear soluciones escalables de comunicación; por ejemplo, el almacenamiento automático en caché para optimizar el ancho de banda o enviar la información por diferentes caminos hacia el destino.

En segundo lugar, se encuentra el proyecto *MobilityFirst*. Este proyecto propone una arquitectura donde la movilidad de los dispositivos pasa de ser un caso en particular, a ser el eje principal de la red. El enfoque propuesto permite la existencia servicios *context-and-location aware* de manera inherente. Por otro lado, esta arquitectura

utiliza el modelo *generalized Delay-Tolerant Networking*, el cual provee comunicaciones robustas incluso cuando se presentan interrupciones en el enlace. Adicionalmente, hace uso del sistema *self-certifying public key addresses* para brindar una red confiable y segura.

Luego está el proyecto NEBULA, el cual plantea una arquitectura donde los *data centers* son los principales repositorios de información y centros de procesamiento. Por lo tanto, deben estar conectados por *backbones* de alta velocidad, redundancia y seguridad. Este proyecto está orientado a desarrollar redes de *core*, con planos de control y datos que soporten servicios de alta disponibilidad, bajo el modelo de *Cloud Computing*.

También cuenta con el proyecto *eXpressive Internet Architecture* (XIA). Este proyecto consiste en una arquitectura que busca crear una única red, que soporte la comunicación entre las principales entidades actuales —clientes, contenido y servicios— y las futuras. Para cada entidad, se define una *Application Programming Interface* (API) y mecanismos flexibles (*intent and fallback addresses*) para el envío de información; además, provee seguridad para la información intrínseca usando *self-certifying identifiers*, lo que permite una comunicación íntegra y con autenticación.

El último proyecto propuesto es *ChoiceNet*, el cual plantea desarrollar una arquitectura en el *core* de las redes, donde se pueda innovar usando principios económicos. Esta arquitectura —basada en el concepto de innovación generada por la competencia— tiene como objetivos generar alternativas para que los usuarios puedan escoger entre diferentes servicios; que los usuarios puedan pagar por mejores servicios; y proveer mecanismos para informar sobre alternativas disponibles y sus performances.

1.2.2. *Software-Defined Networking*

Software-Defined Networking (SDN) es una arquitectura de red en la cual se separa el plano de control del plano de datos (*forwarding*), contrario al modelo actual, donde el plano de control está distribuido en los elementos de red; esta separación permite que el plano de control sea completamente programable. Adicionalmente, ofrece una abstracción de la infraestructura subyacente para las aplicaciones y servicios de redes, tratando a la red como una sola entidad lógica.

Según Kreutz et.al [3], esta arquitectura se basa en 4 principios: separación de plano de control del plano de datos; decisiones de *forwarding* en base a *flows* en vez de direcciones IP destino; lógica de control migrada a un controlador SDN; y capacidad de programar las redes a través de aplicaciones en el controlador.

Como se muestra en la figura 1-1, la arquitectura de SDN se divide en 3 niveles: infraestructura de red, control y aplicación. Estas se comunican entre sí a través de APIs, más específicamente las interfaces *southbound* y *nourthbound* [5].

La capa de infraestructura de red está compuesta por un conjunto de equipos de red (p.ej. *switches*, *routers*); estos equipos —a diferencia de los tradicionales— son nodos que se limitan a reenviar paquetes, pues no poseen un plano de control que les indique qué hacer con los paquetes. La interfaz *southbound* es el elemento que permite separar el plano de control del de datos; OpenFlow es la interfaz con mayor aceptación y despliegue para SDN. En la capa de control, reside el controlador SDN; este puede abstraer los detalles de la infraestructura subyacente y presentarla como un solo *switch* lógico a las aplicaciones de red, pues posee una visión global de la red. Con esta capacidad, los *switches* no deben entender diferentes protocolos, sino simplemente aceptar instrucciones del controlador. La interfaz *northbound* se encarga de comunicar al controlador con las aplicaciones y servicios de la red. Finalmente, la capa de aplicación se encarga de implementar la lógica de control de la red: esta lógica se traduce en comandos a instalar en el plano de control, que posteriormente decidirán el comportamiento de los equipos de *forwarding*.

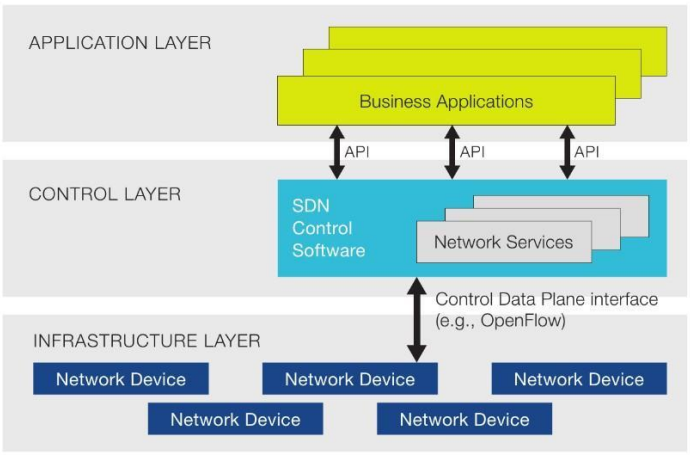


FIGURA 1-1: ESQUEMA DE LAS CAPAS DE LA ARQUITECTURA DE SDN. FUENTE: [5]

Como se mencionó anteriormente, OpenFlow es la interfaz estándar de comunicación entre las capas de control e infraestructura. Esta interfaz define a los elementos de *forwarding* como *switches* OpenFlow, los cuales poseen 3 partes principales: *Flow Table*, *Secure Channel* y el protocolo OpenFlow [6].

Una *Flow Table* es una tabla que almacena *Flow Entries* junto con una acción, indicando cómo procesar dicho *flow*; el *Secure Channel* es un canal que conecta el *switch* con el controlador, lo que permite una comunicación segura; por último, el protocolo OpenFlow se usa en la comunicación del controlador y el *switch*, donde el primero indica qué *flows* se deben instalar en la *Flow Table* del segundo, tal como se muestra en la figura 1-2.

Existen dos tipos de *switches* OpenFlow: *dedicated* OpenFlow *switches*, los cuales reenvían paquetes únicamente según lo defina el controlador (no soportan la lógica de L2/ L3 normal), y OpenFlow-*enabled switches*, que son equipos comerciales que poseen un *feature* para habilitar el protocolo de la interfaz *southbound* (OpenFlow u propietario) entre ellos y el controlador.

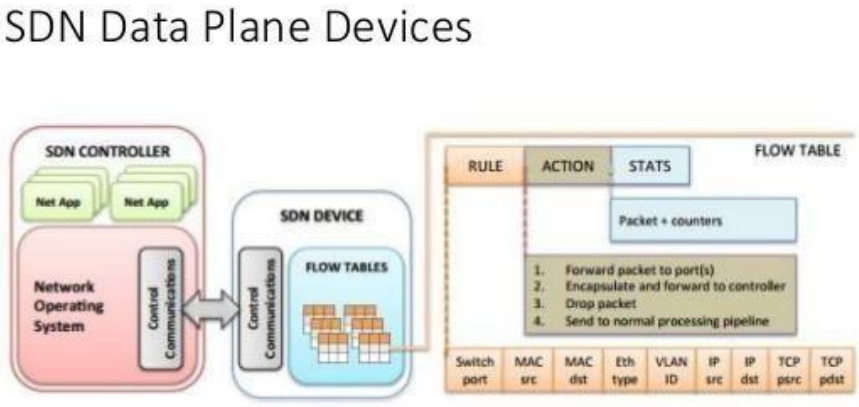


FIGURA 1-2: INTERACCIÓN ENTRE EL CONTROLADOR Y *SWITCHES* OPENFLOW, JUNTO CON LOS CAMPOS DEL *FLOW TABLE*.
FUENTE: [3]

1.2.3. ESnet/ OSCARS

Energy Sciences Network (ESnet) [7] es una red a nivel nacional de alta performance,

que tiene como función transportar datos de investigaciones científicas. Fue fundada por la Oficina de Ciencias (SC) del Departamento de Energía de los Estados Unidos (DoE) y actualmente es administrada por el Laboratorio Nacional *Lawrence Berkeley*. En la figura 1-3, se observa que ESnet brinda servicios a más de 40 sitios de investigación del DoE, entre los cuales se encuentran el *National Laboratory System,* sus instalaciones de supercomputadoras y sus principales instrumentos científicos. También se conecta con 140 redes de investigación y comerciales, lo que permite a los científicos del DoE colaborar con sus colegas alrededor del mundo.

On-demand Secure Circuits and Advance Reservation System (OSCARS) [8] es un *software* de código abierto desarrollado por ingenieros de red especializados en brindar soporte al *National Laboratory System* de los Estados Unidos. Este *software* está diseñado para reservar los recursos de la red ESnet por un intervalo de tiempo, lo que permite asegurar que la transferencia de grandes volúmenes de datos de redes científicas sea de forma rápida y sin fallas. Esto representa una ventaja para este tipo de aplicaciones, pues desplegar y configurar una red que soporte dicha magnitud de tráfico puede tardar semanas o meses. Adicionalmente, provee de forma dinámica caminos de respaldo en caso el usado presente fallas.

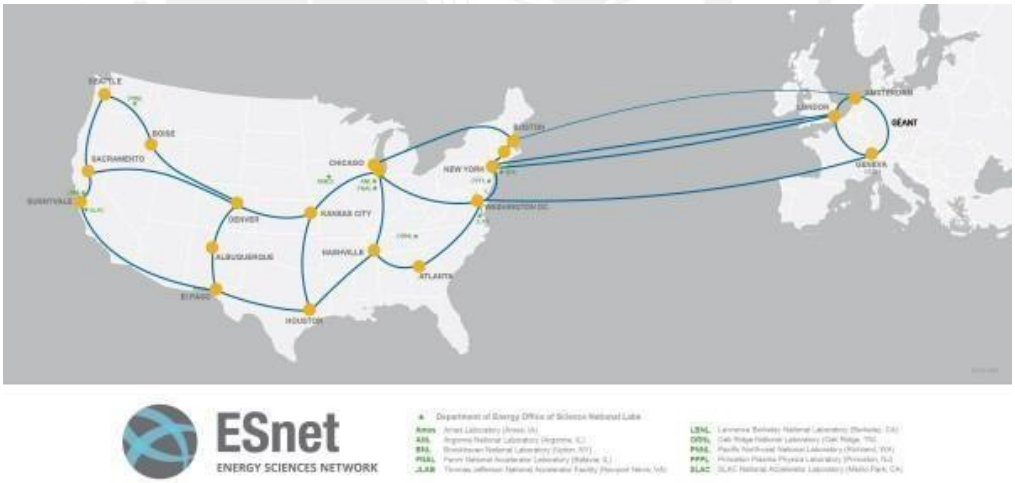


FIGURA 1-3: INTERCONEXIONES DE LA RED DE ESNET.
FUENTE: [7]

El funcionamiento de OSCARS se centra en dos procesos: reserva de recursos y enrutamiento de caminos en base a la disponibilidad de recursos y reservas vigentes. Tal como se muestra en la figura 1-4, el usuario realiza una reserva donde indique

sus requerimientos mínimos (p.ej. origen, destino, ancho de banda requerido y tiempos de inicio y fin de la reserva). Luego, OSCARS recibe la solicitud y autentica la identidad del usuario para determinar si se encuentra autorizado; de ser exitosa la autenticación, el *software* determina si existen caminos que satisfagan los requerimientos del usuario. Para hacer el procesamiento más eficiente, se simplifica la red eliminando los enlaces que no cumplan con el requerimiento de ancho de banda en el intervalo pedido; en caso se encuentre una alternativa, OSCARS actualiza su base de datos de recursos con la información de la reserva.

Si bien los inconvenientes de las redes tradicionales (p.ej. relacionados a operatividad, seguridad y flexibilidad) se resuelven con la propuesta de nuevas arquitecturas (FIA, SDN, OSCARS), para que los operadores de redes opten por implementar estos nuevos modelos, **se requieren de entornos de prueba donde se puedan analizar las ventajas y limitantes de estas nuevas arquitecturas.**

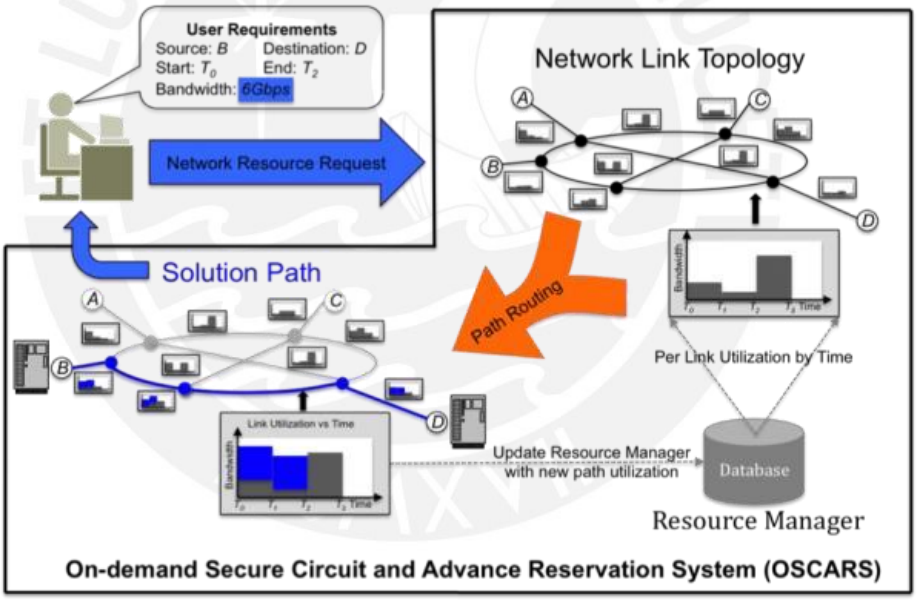


FIGURA 1-4: DIAGRAMA DEL FUNCIONAMIENTO DE LA RESERVA DE CIRCUITOS EN OSCARS.
FUENTE: [8]

2. Entornos de prueba de redes

Si bien se requiere cambiar el modelo actual de las redes y de Internet, los operadores deben evaluar el funcionamiento de las arquitecturas emergentes, antes de implementarlas dentro de su infraestructura. De acuerdo con los resultados obtenidos en las pruebas, el proveedor de servicios determinará si es viable su implementación. Por este motivo, es necesario disponer de entornos con las herramientas necesarias para analizar el desempeño de las nuevas arquitecturas.

Una característica fundamental de los entornos de prueba es la fidelidad de los resultados obtenidos, en comparación con los alcanzados en un entorno en producción. Tomando en cuenta el grado de fidelidad de los resultados conseguidos, los entornos de prueba se pueden clasificar en tres: *testbed* físicos, entornos conformados por equipos reales cuya fidelidad es la más alta; emuladores, entornos que modelan el *hardware* de equipos reales usando máquinas virtuales (VM o *virtual machine*), sobre los cuales se cargan los sistemas operativos (OS) reales de los equipos de redes; y simuladores, cuya fidelidad es la más baja debido a que se modela la interacción del OS con el *hardware* de los equipos mediante código.

De los entornos de prueba mencionados, el emulador es el punto de equilibrio entre fidelidad y costo: sus resultados son más exactos que los de un simulador y —si bien un *tesbed* físico tiene resultados aún más precisos— el costo de implementación de un emulador es mucho menor. En base al alcance de los recursos que poseen, los emuladores se pueden clasificar en tres tipos: *in-a-box*, el cual opera en el *hardware* de un solo servidor; *in-a-rack*, conformado por una red de servidores en una misma locación; e *in-a-federation of racks*, compuesto por un conjunto de *racks* distribuidos geográficamente.

2.1. Ordenados por fidelidad

Debido a que los resultados obtenidos en un entorno de prueba influyen en la decisión de implementar una nueva arquitectura, la fidelidad es un factor que debe estar presente en dichos entornos. De acuerdo al grado de fidelidad obtenido en cada uno, los entornos de prueba se pueden clasificar en *testbeds* físicos, emuladores y simuladores.

2.1.1. *Testbed* físico

Un *testbed* físico es un entorno de prueba en donde se implementan prototipos de redes utilizando equipos reales. Estos entornos permiten recrear diversos escenarios en base a la cantidad de equipos y puertos disponibles en cada uno. Si bien los *testbeds* físicos presentan una alta fidelidad en los resultados obtenidos, no resultan escalables por el elevado costo de implementar una red grande.

Las topologías de redes evaluadas en los *tesbeds* físicos varían en su alcance de acuerdo a los equipos utilizados. Existen entornos en los cuales se emplean equipos comerciales (p.ej. Arista, NEC IP8800) así como *hardware* con diseños más simples (p.ej. NetFPGA [9], Raspberry-Pi [10]). La elección de los equipos influye en el alcance del prototipo de red, pues los primeros poseen mayor cantidad de puertos (permite mayor número de conexiones) y ASICs de mayor capacidad.

La característica principal de estos entornos es la presencia de una alta fidelidad en sus resultados en comparación con los alcanzados en una red comercial. Esto es posible ya que, al utilizar el mismo *software* y *hardware* (o uno de menor capacidad),

se puede analizar de forma exacta las limitantes del equipo, la lógica implementada por el sistema operativo y la interacción de este con el *hardware*.

Por otra parte, el costo de implementación de un *testbed* físico varía de acuerdo a la complejidad del escenario que se desee recrear: mientras más compleja sea la red, mayor es el número de equipos necesarios. Si se considera el precio de cada equipamiento, el monto total de implementación resulta alto para un escenario mediano.

Si bien la cantidad de equipos por *testbed* físico es limitada, existen proyectos internacionales que permiten interconectar diferentes *testbeds* de instalaciones en otros países, a través de redes de alta capacidad y utilizar sus recursos; no obstante, estos recursos están disponibles solo para las entidades —por lo general campus universitarios— que pertenezcan al proyecto de investigación. Un ejemplo de ello es Ofelia [11], el cual es un proyecto europeo que permite experimentar con el protocolo OpenFlow. Una de las universidades asociadas a Ofelia es la Universidad de Bristol, la cual dispone de *switches* OpenFlow, controladores SDN, equipos ópticos y servidores para la creación de máquinas virtuales.

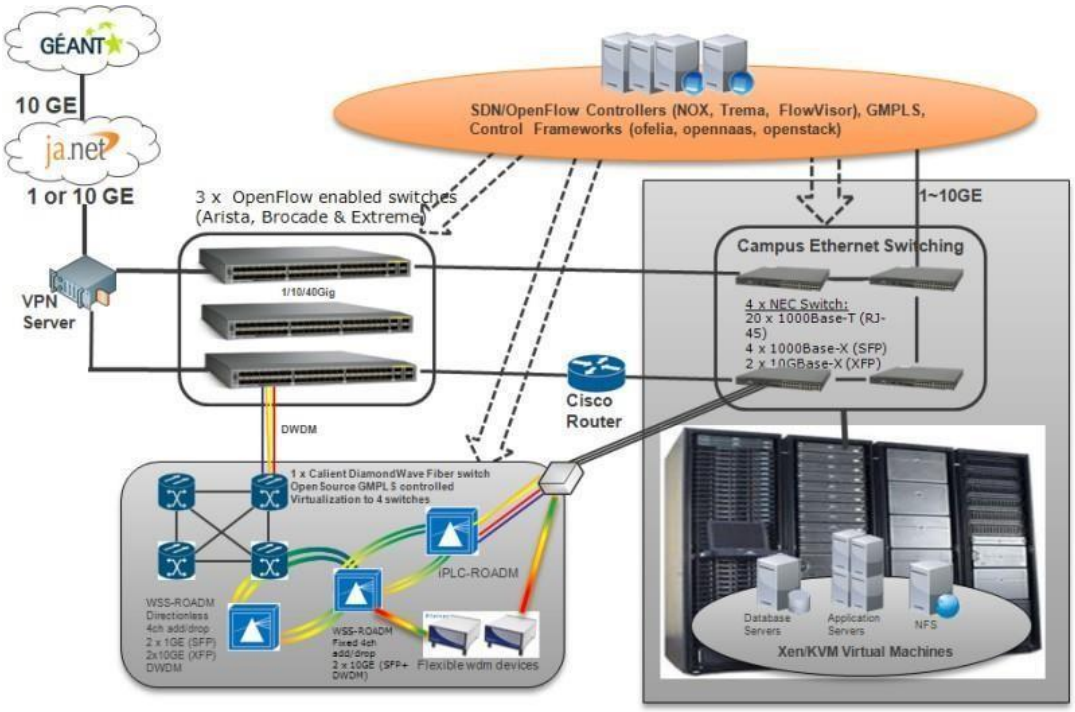


FIGURA 2-1: DISTRIBUCIÓN DE EQUIPOS DE LA UNIVERSIDAD DE BRISTOL. FUENTE: [12]

2.1.2. Emuladores

Los emuladores son programas que presentan una abstracción del *hardware* a los sistemas operativos de los equipos, utilizando el *Computer Model*. Este se define como la "representación virtual de un sistema en el cual los componentes físicos del sistema están representados por componentes virtuales" [13]. Para virtualizar el *hardware*, los emuladores utilizan *hypervisors.* Estos proporcionan máquinas virtuales sobre las cuales se cargan los sistemas operativos.

Como se menciona en [13], una propiedad fundamental de los emuladores es la ejecución en tiempo real de las instrucciones del sistema virtualizado. Esto permite que el tiempo de ejecución de las instrucciones de dichos sistemas sea idéntico al de los equipos reales. Por otra parte —a través del *hypervisor*— es posible abstraer y asignar una parte de los recursos de los servidores (CPU, RAM, almacenamiento) a cada máquina virtual.

Otra característica de los emuladores es que la alta fidelidad de los resultados alcanzados en ellos; esto se debe a que las instrucciones del sistema emulado se ejecutan de la misma manera que en los equipos físicos. Además de ello, el hecho de ejecutarse en tiempo real permite que estos resultados sean bastante semejantes con los obtenidos en entornos de producción.

Asimismo, el costo de implementación de un emulador se puede considerar como bajo-intermedio. Por un lado, puede resultar con un bajo costo de implementación pues existen emuladores (p.ej. Mininet) que pueden funcionar en computadoras personales; por otro lado, se han desarrollado emuladores (p.ej. DOT) que operan en servidores distribuidos, lo que eleva su costo de despliegue.

Uno de los entornos de emulación más empleados es GNS3. Este entorno es un emulador de redes donde principalmente se virtualizan equipos Cisco [14] utilizando el *hypervisor* llamado Dynamips, el cual abstrae el *hardware* (ASICs) donde se ejecuta el IOS de Cisco. Este emulador utiliza una arquitectura de cliente-servidor, donde la GUI del programa se encuentra en el cliente y el *hypervisor* reside en el servidor. También se puede integrar con otros *hypervisors* como QEMU o VirtualBox, en los cuales se puede emular equipos *hosts* o clientes (p.ej. con Windows), así como equipos (con su respectivo sistema operativo) de otros fabricantes (p.ej. Juniper JunOS, Alcatel TiMOS, etc.).

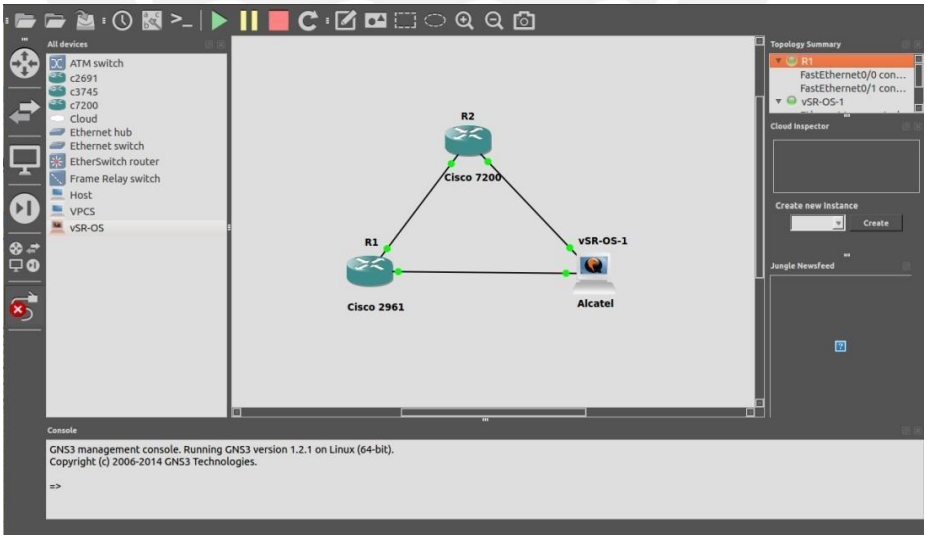


FIGURA 2-2: INTERFAZ GRÁFICA DE GNS3.
FUENTE: ELABORACIÓN PROPIA

2.1.3. Simuladores

Los simuladores son entornos de prueba que reproducen la operación e interacción de los equipos reales, utilizando líneas de código. De acuerdo con [13], los simuladores suelen emplear la técnica *discrete-event simulation* [13]. Esta consiste en representar la operación de los sistemas como una secuencia de eventos.

Una característica de los simuladores es que modelan en *software* ciertos aspectos de los equipos físicos (p.ej. transmisión de paquetes, interacción del sistema operativo con el *hardware*); debido a ello, no tienen noción del tiempo real, sino que emplean un tiempo virtual [13]. Este tiempo varía de acuerdo a la complejidad del escenario de prueba, a los algoritmos empleados para modelar las operaciones de la red y la capacidad de procesamiento del equipo.

Por otra parte, la fidelidad de los resultados del simulador es baja, pues depende de la complejidad del escenario de prueba [13] y el nivel de detalle puesto en el modelamiento de los equipos. La simulación de escenarios simples puede ser rápida, incluso más que en tiempo real; sin embargo, simular entornos más grandes y reales toma mayor tiempo de ejecución. Además, este tiempo varía si el procesador se encuentra ocupado ejecutando procesos en paralelo.

Con respecto al costo de implementación, este resulta bajo debido a que el simulador está basado en líneas de código, las que conforman un conjunto de *scripts*. Existen casos donde puede ser gratuito —incluso de código abierto; en otros casos, puede tratarse de un *software* comercial, el cual tendrá cierto costo de adquisición. En cualquiera de los casos, el costo de implementación del simulador es menor, pues puede correr en cualquier *hardware*.

Entre los diversos simuladores que existen en la actualidad podemos mencionar a Ns-3 [15]. Este *software* es una herramienta de simulación de redes, en la cual las operaciones de un nodo son implementadas compilando y relacionando un módulo desarrollado en C++ con el código de simulación. También permite crear diferentes tipos de redes a nivel de L3 (no solamente IP): en él se pueden simular escenarios de redes inalámbricas (capas 1 y 2 de Wi-Fi, WiMAX, LTE) y nuevas arquitecturas de redes IP (SDN).

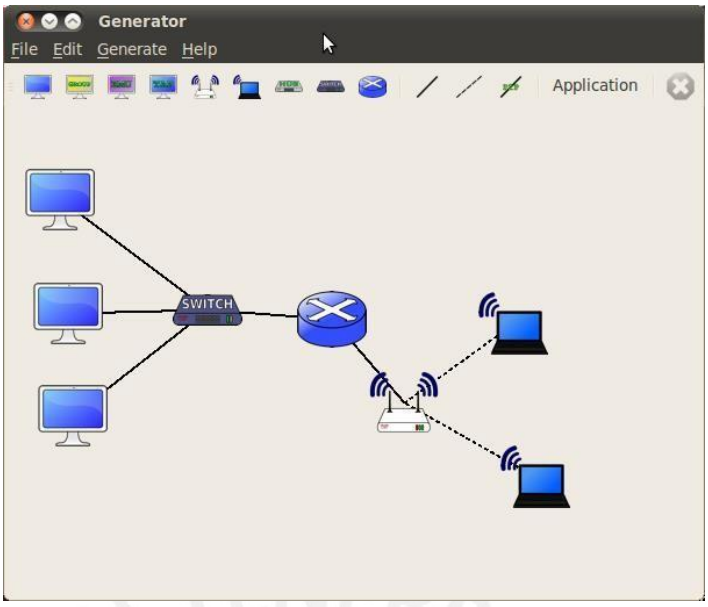


FIGURA 2-3: GENERADOR DE TOPOLOGÍAS DE NS-3.
FUENTE: [16]

De lo mencionado en las secciones anteriores, se puede determinar lo siguiente:

- Los *testbeds* físicos son los entornos de prueba que poseen mayor fidelidad en sus resultados; empero, su costo de implementación es el más elevado de las opciones.
- Los emuladores presentan un menor nivel de fidelidad en comparación con los *testbeds* físicos; no obstante, dichos resultados se asemejan a los observados en redes reales. Por otro lado, poseen un costo de implementación medio.
- Los simuladores, si bien poseen el costo de implementación más bajo, los resultados obtenidos en ellos pueden diferir de los esperados en redes reales; es por ello que presentan el menor nivel de fidelidad.

Por estas razones, se puede considerar al emulador como un entorno de pruebas que presenta un equilibrio entre fidelidad de resultados y costo de implementación. **Como parte del desarrollo del presente trabajo de tesis, se centrará el análisis en los emuladores**.

2.2. Emuladores por alcance

Los emuladores se pueden clasificar por el alcance del número de equipos que utilizan. Esta clasificación indica la cantidad de recursos disponibles para las

máquinas virtuales y en dónde se ubican dichos recursos; según esta característica, los emuladores se pueden agrupar en emuladores *in-a-box, in-a-rack* e *in-a-federation of racks*.

2.2.1. *In-a-box*

Los emuladores *in-a-box* son aquellos que residen en un solo servidor físico. Este tipo de emulador se emplea para validar el prototipo de una red de forma rápida a un bajo costo de implementación.

Una particularidad de los emuladores *in-a-box* es que no comparten una arquitectura definida; sin embargo, se pueden identificar dos etapas principales de virtualización: una de equipos de red y otra de *hosts*. En la primera, se hace uso de herramientas de virtualización (p.ej. *dynamips*, *kvm*, etc.) para emular el *hardware* de los equipos de comunicaciones; en la segunda, pueden existir *hosts* físicos o virtuales (*namespaces*, máquinas virtuales), dependiendo del emulador.

Este tipo de emulador permite evaluar redes de forma rápida, pues hace uso de herramientas como *lightweight virtualization* (p.ej. linux *namespaces,* enlaces tipo veth, etc.) o linux *disciplines* (*traffic control* y *network emulation*). Además, debido a que puede funcionar en equipos con diversas características —desde un servidor dedicado hasta una computadora portátil— su costo de implementación es bajo. Por otro lado, disponer de una cantidad fija de recursos de un único *hardware* limita la escalabilidad del emulador. Esto genera que el volumen de tráfico simulado sea bajo y se prefieran usar contenedores en lugar de máquinas virtuales.

Mininet [17] es un emulador de redes SDN basado en contenedores, que permite crear escenarios con los *switches* virtuales Open vSwitch conectados a un controlador. Mininet está diseñado para ejecutarse en sistemas operativos basados en Linux, utilizando linux *namespaces* para los Open vSwitch (root) y uno diferente por *host*. En cuanto a las conexiones entre equipos, este emulador utiliza enlaces de tipo veth (*switch-host* y *switch-switch*). También permite manipular el *bit rate* y *delay* usando linux *disciplines*; no obstante, el uso de contenedores asociados a un *kernel* en particular limita la flexibilidad en la elección de nodos y *hosts* a emular.

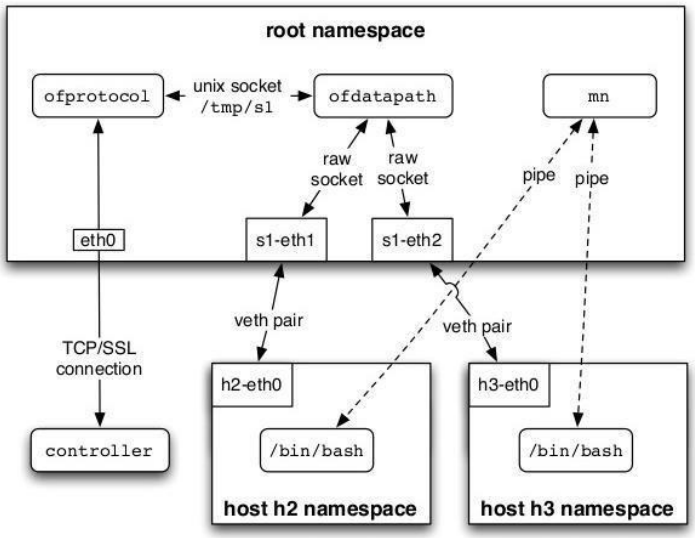


FIGURA 2-4: ARQUITECTURA DE MININET.
FUENTE: [17]

2.2.2. *In-a-rack*

Los emuladores *in-a-rack* son aquellos entornos cuyos componentes se encuentran distribuidos en equipos a lo largo de una red LAN. Estos equipos pueden ser computadoras o servidores dedicados a la emulación de nodos de red; así, la cantidad de elementos virtualizados es mayor en comparación con los emuladores *in-a-box*, y los enlaces entre máquinas virtuales (en diferentes servidores) adquieren mayor realismo por estar sobre una red física.

En la arquitectura de los emuladores de redes *in-a-rack*, se pueden diferenciar dos elementos: un *control node* (CN), donde se administran los recursos del emulador, y uno o más *worker nodes* (WNs), donde se alojan las máquinas virtuales (VMs). En el CN reside el *software* encargado de mapear los requerimientos de las redes virtuales

en reservas de recursos en los WNs. También cuenta con una base de datos, donde se almacena la información acerca de la utilización de recursos del *rack* recolectada por cada WN y el mapeo de requerimientos-recursos hecho por el CN. En los WNs, se asignan y utilizan dichos recursos para crear los elementos de la red virtual (enlaces, túneles, VMs). Adicionalmente, un *switch* virtual distribuido en cada WN se encarga de enviar el tráfico entre VMs a diferentes WNs mediante una tecnología de encapsulamiento (VLANs, L2 túneles, etc.); en cuanto a la LAN que une los CN y WNs, se necesita por lo menos un *switch* físico para la interconexión.

Este método de emulación, al tener mayor cantidad de recursos con respecto al emulador *in-a-box*, permite probar redes con una mayor cantidad de nodos; como se muestra en [18], esta limitación de recursos en los emuladores *in-a-box* —a diferencia de los emuladores *in-a-rack*— condiciona a tener un *throughput* bajo en los enlaces cuando se aproximen a la capacidad máxima de *switching* del equipo. Adicionalmente, se puede garantizar un cierto ancho de banda en los enlaces, en base a la distribución de la topología de la red emulada y el volumen de tráfico entre los *worker nodes*; sin embargo, para disponer de una mayor cantidad de recursos, se requieren más servidores en el rack, lo que aumentaría su costo de implementación. Además, tener un *software* de control implica que se añada un nivel adicional de complejidad a este enfoque.

Distributed OpenFlow *Testbed* (DOT) es un emulador de redes SDN propuesto por Arup Roy y demás [18], en el cual presentan un sistema distribuido y plantean un algoritmo heurístico para minimizar el *overhead* y el número de servidores activos por topología emulada. DOT hace uso de un *hypervisor* para crear VMs (para los *hosts*) en los WNs y —al tratarse de un emulador orientado a OpenFlow— emplea *switches* virtuales que soporten el protocolo OpenFlow. Estos *switches* virtuales no se encuentran en VMs, sino que se crean en el mismo sistema operativo de cada WN. Finalmente, para enviar el tráfico entre VMs de diferentes WNs, DOT utiliza túneles tipo GRE para encapsular y aislar el tráfico de cada enlace virtual.

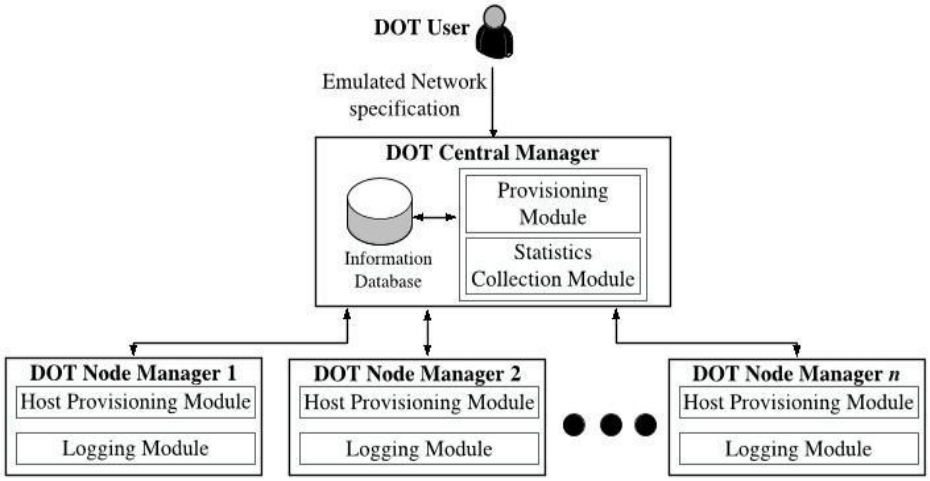


FIGURA 2-5: ARQUITECTURA DE DOT.

FUENTE: [18]

2.2.3. *In-a-federation of racks*

Network in-a-federation es un tipo de emulador en donde se hace uso de *racks* distribuidos geográficamente, unidos por una red de alta velocidad. Estos *racks* no son propiedad de una única entidad, sino que funcionan como una red de colaboración entre varios grupos de investigación. Este enfoque se utiliza cuando se desea probar una red de mayor escala, con eventos reales que afecten a la red.

Los emuladores *in-a-federation of racks* utilizan el modelo del emulador descrito anteriormente (*network-in-a-rack*) para crear las máquinas virtuales e interconectarlas en un solo *rack*. No obstante, para interconectar VMs en diferentes *racks* (o *slices*) se hace uso de redes de capa 2 de alta velocidad y un nivel más de control: *orchestrator.* Como se muestra en [19], en este *framework* de control se diferencian tres niveles: usuario, autenticación y gestión de recursos. En el primero, se encuentra la interfaz de usuario —gráfica o línea de comandos— con el cual los usuarios se autentican en el sistema y eligen los recursos para sus experimentos. El segundo, también llamado *Clearinghouse*, es el responsable de la administración de los *slices* y estado de los usuarios del *Testbed*, así como de realizar la autenticación y autorización de los usuarios. Por último, en el nivel de gestión se administran recursos (p.ej. VM, *flowspaces*) y *slices* en un *rack*.

Este modelo de emulación permite tener un mayor número de recursos disponibles para los experimentos, con lo cual se pueden generar topologías aún más complejas.

Además, debido a que los *racks* de servidores se encuentran distribuidos geográficamente, se usan redes de capa 2 de alta velocidad; esto genera que los enlaces entre máquinas virtuales tengan un mayor realismo que los enlaces en un *rack*, donde son enlaces virtuales o como máximo una LAN dedicada. Por otra parte, para pertenecer a un grupo de *Testbeds* distribuidos, se deben tener equipos con ciertas características (p.ej. servidores con determinadas características, *switches* específicos, etc.). Este requerimiento es importante, ya que estos *Testbed* pertenecen a los grupos de investigación donde se experimentan con nuevas arquitecturas de redes; ello hace que el costo de despliegue del emulador sea alto y no esté al alcance de cualquier usuario.

Un caso de emuladores de redes distribuidos geográficamente es ExoGENI [19]. Este es un *Testbed* de GENI, en el cual se utiliza el *software* de *Cloud Computing* OpenStack (en su quinta versión llamada Essex) para la creación de máquinas virtuales y *dynamic circuit fabrics* para interconectar los *racks*. Según la página del proyecto, las implementaciones de ExoGENI consisten en *racks* ubicados en campus —en EEUU, Holanda, Australia y próximamente Perú— interconectados por redes de investigación a través de *programmable exchange points* (p.ej. BEN y *StarLight*). Los *sites* de ExoGENI y el *software* de control permite operar la red de forma flexible, usando VLAN-*based switching* y OpenFlow. El *software* de control utilizado para orquestar los *cloud sites* tiene por nombre *Open Resource Control Architecture* (ORCA). Este orquestador permite recibir pedidos de recursos y asignarlos a los usuarios a través del *Aggregate Managers (AMs),* administrar los *tickets* que permiten utilizar los recursos del *rack* a diferentes usuarios mediante el *Broker* y administrar el conjunto de recursos (*slices*) a través del *Slice Manager* (SM).

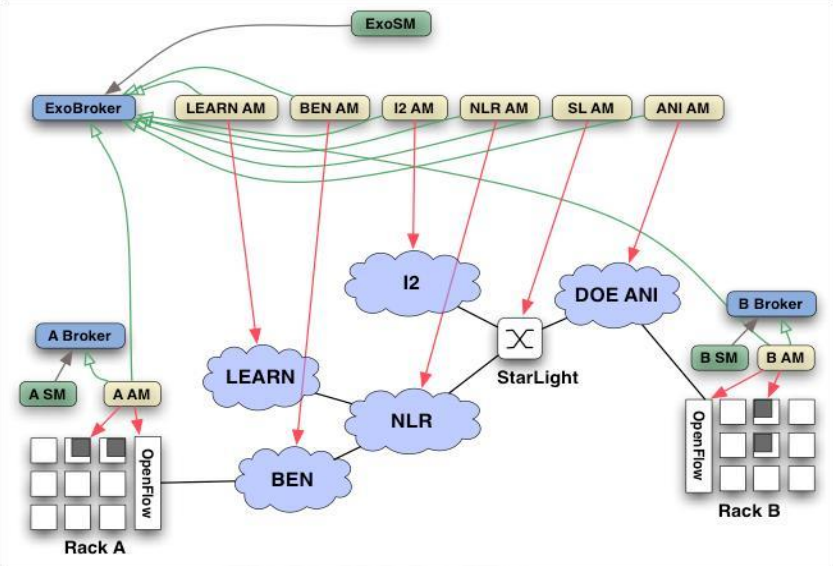


FIGURA 2-6: ARQUITECTURA DEL ORCA *FRAMEWORK*.
FUENTE: [20]

2.3. Comparación entre emuladores

En esta sección se realizará una evaluación de los emuladores que existen actualmente; en ella, se va a analizar las ventajas de cada uno y limitantes que se encontraron en cuanto a su alcance, la variedad de prototipos que se pueden emular y el acceso a dichas herramientas. Para desarrollar dicha evaluación, se tomarán como referencia los emuladores GNS3, Mininet, DOT, y el *Testbed* distribuido ExoGENI.

Como se muestra en la tabla 2-1, los *racks* de ExoGENI exhiben una mayor escalabilidad dado que está basado en un modelo de *Cloud Computing*. Por otra parte, GNS3 y Mininet están orientados a una arquitectura de red específica (*legacy* y SDN, respectivamente); por ello, no es posible evaluar el impacto de nuevos paradigmas en las redes tradicionales en ninguno de ellos. Finalmente, el acceso a los recursos de los *racks* de ExoGENI se encuentra limitado a las instituciones que posean un *rack* de estas características, pues mayormente se utiliza para realizar investigación. Por el contrario, GNS3 y Mininet son emuladores que están al alcance de cualquier usuario.

Cabe mencionar que el único emulador que permite validar la existencia de recursos disponibles para el experimento es ExoGENI (a través del *Aggregate Manager*). Este

es un *feature* fundamental para evitar que se saturen los recursos de los equipos físicos (p.ej. bus de procesador, memoria, almacenamiento, etc.) y genere resultados erróneos.

En el presente trabajo, se optó por implementar el emulador de redes en un entorno *in-a-rack*, debido a que posee mayor cantidad de recursos que un emulador *in-a-box* y se puede distribuir el consumo de recursos entre los servidores. Si bien el emulador DOT permite lo anterior, está diseñado únicamente para emular redes SDN. Además, los *switches* virtuales (Open vSwitch) son implementados en un *namespace* (*root*), por lo que no existe una reserva de recursos para cada nodo. El emulador presentado en esta tesis —al utilizar un *software* de *Cloud Computing* que contiene un *hypervisor*— permite emular diferentes tipos de *kernels*. También se incluye una sección de validación de la topología a implementar, ausente en los emuladores analizados, para asegurar que los resultados de las pruebas sean correctos y proteger al sistema.

Emulador	Ventajas	Desventajas
GNS3	• Redes con equipos de diversos fabricantes (Cisco, Alcatel, Juniper) • Integración otros *hypervisors* • Disponible en cualquier tipo de sistema (Windows, Linux, Mac)	• No permite emular *switches* Cisco • *Throughtput* total limitado por el bus del procesador • Emula redes *legacy*
Mininet	• Innovación en soluciones basadas en SDN/Openflow • Alta escalabilidad de *switches* • Control de tasa de bits y latencia de enlaces	• Disponible en distribuciones de Linux • Aplicaciones basadas solo en Linux • Capacidad de *switching* limitada por recursos del equipo
DOT	• Evaluación de arquitectura nueva como SDN • Mayor cantidad de recursos • Se puede simular enlaces con retardo y con cierta tasa de bits	• Disponible en distribuciones de Linux • No se separan recursos para *switches* virtuales
ExoGENI	• Realismo en enlaces • Uso de recursos de otros *sites* • Evaluación de arquitecturas nuevas y *legacy* • Validación de recursos a nivel de *Broker* de ORCA	• Acceso restringido a miembros de ExoGENI • Complejidad de mantenimiento

TABLA 2-1: COMPARACIÓN DE EMULADORES GNS3, MININET, DOT Y EXOGENI *RACK.*

FUENTE: ELABORACIÓN PROPIA

2.4. Objetivos de la tesis

En base al estudio realizado acerca de la problemática actual de las arquitecturas de redes y el análisis de los tipos de entorno de pruebas existentes, se establece que **el objetivo de la presente tesis es el diseño e implementación de un emulador de redes de alta fidelidad** –que el comportamiento de la red emulada sea similar al de la red implementada con equipos físicos– **y alta capacidad** –que se puedan emular redes LAN y MAN–, en donde sea posible probar el comportamiento de nuevas tecnologías. El diseño debe ser modular y tomar como base un *rack* de servidores con 160 *cores* corriendo la plataforma de OpenStack, al cual se le realizarán

modificaciones para permitir la simulación de la capa física. El emulador deberá proveer una interfaz al usuario en la cual podrá definir y visualizar topologías, cargar imágenes del *software* de equipos reales, y definir propiedades de los enlaces (retardo y velocidad de transmisión). Asimismo, deberá incluir un módulo que valide que el *hardware* no introduzca fenómenos artificiales.

El diseño e implementación del emulador requiere: (i) determinar los requerimientos del emulador y su comparación con las prestaciones de las soluciones de virtualización (IaaS) existentes, en particular con la plataforma de código abierto OpenStack; (ii) extender OpenStack, incluyendo funcionalidades de control de la velocidad de transmisión y retardo de cada enlace; (iii) desarrollar (en Python) la interfaz de usuario, los servicios *core* del emulador, el módulo de validación, y la interfaz a OpenStack; (iv) calibrar el *hardware* para obtener el consumo de recursos por elemento de red virtualizado y la cantidad disponible de estos, y (v) realizar pruebas de funcionamiento, escalabilidad, validación y aislamiento.

3. Tecnologías de virtualización

En la actualidad, los proveedores de servicios se caracterizan por poseer infraestructuras distribuidas geográficamente. Estas infraestructuras están conformadas por cientos de equipos interconectados entre sí, generando grandes inversiones tanto de implementación como de mantenimiento. Asimismo, la gestión de cada equipo se ha vuelto complicada, sobre todo si es necesario realizar algún cambio que influya en la operación de la red. Ante estos inconvenientes presentes en las redes tradicionales, se está optando por utilizar la virtualización como una solución.

La virtualización se define como la abstracción de los diversos recursos de TI. A partir de esta afirmación, existen varios enfoques donde se puede aplicar la virtualización. El enfoque de *Server Virtualization* consiste en la división de los recursos de un servidor físico para crear máquinas virtuales, utilizando un *hypervisor* en conjunto con una imagen de disco; *Network Virtualization* es un enfoque en el cual se abstrae la red física que interconecta los equipos y se genera una red virtual; *Storage Virtualization* consiste en abstraer el almacenamiento de los equipos y lo divide de

acuerdo a la capacidad requerida. Estos enfoques de virtualización utilizan un gestor de entornos virtualizados para monitorear los recursos (CPU, memoria, almacenamiento) de los servidores y evitar un consumo excesivo de estos.

Por otro lado, para crear un entorno virtualizado es necesario un administrador que permita automatizar la creación, supervisión e implementación de diversos recursos. Este administrador es denominado orquestador. Uno de los orquestadores más conocidos es OpenStack, el cual es una plataforma de *Cloud Computing* que permite crear máquinas virtuales —conectadas entre ellas y hacia redes externas— mediante la interacción de varios servicios. En los siguientes capítulos, se realizará una explicación más detallada de cada uno de los enfoques de virtualización, así como de OpenStack, plataforma sobre la cual se implementó el emulador.

3.1. Virtualización

Según [21], se define la virtualización como la emulación transparente de los recursos de TI para generar beneficios a los usuarios. Esta definición implica que dichos recursos (memoria, CPU, redes, almacenamiento) se encuentran abstraídos, de manera que los usuarios no noten que utilizan una infraestructura emulada. Estos recursos virtuales permiten expandir la memoria de los sistemas, optimizar y crear redundancia de los recursos físicos, etc.

3.1.1. Tipos de virtualización

Según [21], se pueden identificar tres tipos de virtualización en base a la forma en que se utilizan los recursos físicos: *Pooling*, *Abstraction* y *Partitioning*. A continuación, se muestra la tabla 3-1 donde se explica cada uno de ellos.

Tipo	Características
Pooling	Los recursos funcionan en forma paralela, pero se presentan como una sola entidad lógica
Abstraction	Un elemento de los recursos crea un solo elemento virtual con diferentes características respecto al elemento físico
Partitioning	Los recursos físicos se dividen en particiones y cada una forma una entidad lógica

TABLA 3-1: TIPOS DE VIRTUALIZACIÓN.

FUENTE: [21]

Asimismo, existen subclases de cada una en base al funcionamiento de los recursos, tal como se observa en la siguiente tabla.

Tipo	Subclase	Características	Ejemplo
Pooling	Homogénea	Elementos de un mismo tipo	GLBP
	Heterogénea	Elementos de diferentes tipos	Memoria virtual (RAM)
Abstraction	*Address remapping*	Solo los identificadores del elemento virtual diferentes de los físicos	HSRP
	Structural	Formato del elemento virtual diferente del físico	L2/L3 Túnel
Partitioning	*Resource allocation*	Se reservan recursos para cada partición	Máquinas virtuales (VM) de IBM
	No resource allocation	No se reservan recursos para cada partición	VM de KVM con *overcommit*

TABLA 3-2: SUBCLASES DE VIRTUALIZACIÓN.
FUENTE: [21]

3.1.2. Áreas y herramientas de virtualización

A medida que el desarrollo de las técnicas de virtualización ha ido avanzando, estas se han orientado a tres áreas principales: *storage*, *server*, y *networking* [21], tal como se muestra en la siguiente imagen.

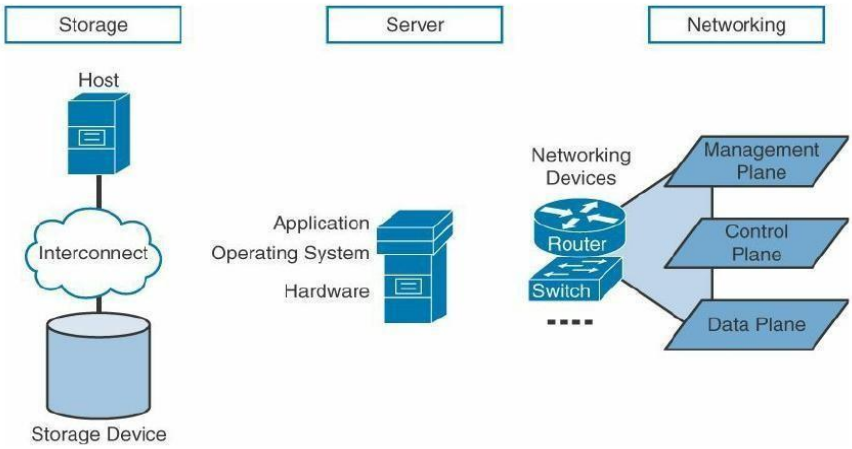


FIGURA 3-1: ÁREAS DE VIRTUALIZACIÓN: *STORAGE*, *SERVER* Y *NETWORKING.*
FUENTE: [21]

Adicionalmente, se puede realizar una clasificación más detallada —dependiendo de qué elemento se virtualiza— tal como se muestra en la tabla 3-3.

Por otro lado, es necesario contar con un sistema de monitoreo y gestión de los recursos virtuales —para tener una visión interna del estado de las máquinas virtuales y de los recursos de los servidores— y, así, realizar un *troubleshooting* más eficiente en los entornos virtualizados. Además, hace posible que los sistemas funcionen de manera óptima y que se tomen medidas correctivas con máquinas virtuales que consuman los recursos excesivamente [22].

Además, en los entornos virtualizados se introduce el concepto de orquestación de máquinas virtuales. Este sistema, usado mayormente en entornos de *Cloud Computing*, permite la automatización del aprovisionamiento de los recursos de los servidores; con ello, se pueden desplegar escenarios de servidores virtualizados con gran número de nodos, de forma rápida y en el orden correcto, de modo que no genere conflictos entre los servicios [23].

Área	Sub-área	Descripción
Storage virtualization	*Host*	Sistema que maneja y guarda la información
	Interconexión	Indica el medio (red) entre el *host* y el dispositivo de almacenamiento
	Equipo de almacenamiento	Lugar donde se guardan los datos
Server virtualization	Aplicación	Aplicación corriendo en el sistema operativo
	Sistema operativo	Capa de *software* que controla el *hardware* del servidor
	Hardware	*Hardware* del equipo
Network virtualization	*Management plane*	Contiene componentes destinados a administrar equipos (CLI, SNMP)
	Control plane	Procesa el tráfico de los equipos de red que usan para controlar el *data plane*
	Data plane	Maneja el tráfico de los usuarios entre equipos de red

TABLA 3-3: CLASIFICACIÓN DE *SERVER*, *NETWORK* Y *STORAGE VIRTUALIZATION.*
FUENTE: [21]

3.1.3. *Cloud Computing*

Cloud Computing es un modelo de computación que brinda acceso en demanda a un *pool* compartido de recursos configurables (redes, servidores, almacenamiento, aplicaciones y servicios). Este modelo posee la característica de proveer dichos recursos de forma rápida, con una mínima configuración de la plataforma [24].

De los servicios ofrecidos en las plataformas de *cloud*, se pueden identificar tres categorías principales: *Software as a Service (SaaS)*, *Platform as a Service (PaaS)* e *Infrastructure as a Service (IaaS)*. En la siguiente tabla, se muestran los tipos de *Cloud Computing* junto con una descripción del servicio que ofrecen.

Tipo de *Cloud*	Descripción	Ejemplo en el mercado
Software as a Service	Presenta aplicaciones a los usuarios a través de la Web	Google Aps, Microsoft Office online
Platform as a Service	Brinda herramientas de programación para desarrollar aplicaciones	Google App Engine, AWS Elastic Beanstalk
Infrastructure as a Service	Ofrece recursos como servidores, almacenamiento, redes y sistemas operativos	Amazon EC2, OpenStack

TABLA 3-4: DESCRIPCIÓN DE LOS TIPOS DE *CLOUD COMPUTING.*

FUENTE: [24]

3.2. *Server virtualization*

Server virtualization es un enfoque de virtualización en el cual se dividen los recursos de un servidor físico entre múltiples sistemas operativos [25]. Entre los diferentes mecanismos de virtualización, lo más frecuente es emplear máquinas virtuales. Este enfoque requiere utilizar un *hypervisor* en conjunto con una imagen de disco para desplegar una máquina virtual.

3.2.1. Métodos de server *virtualization*

Como se indica en [25], el enfoque de *server virtualization* puede utilizar tres métodos distintos para virtualizar los recursos del servidor físico: *hardware emulation*, *paravirtualization* y c*ontainer virtualization*, como se ilustra en la figura 3-2.

En primer lugar, *hardware emulation* es un método que permite crear máquinas virtuales a través de *hypervisors* (o *virtual machine monitor*). Se necesita de uno para que el sistema operativo de la VM pueda ejecutar en *user space* instrucciones que requieren permisos de administrador; la razón de ello es que —al no haber sido modificados los sistemas operativos— desconocen que no están corriendo sobre *hardware* real.

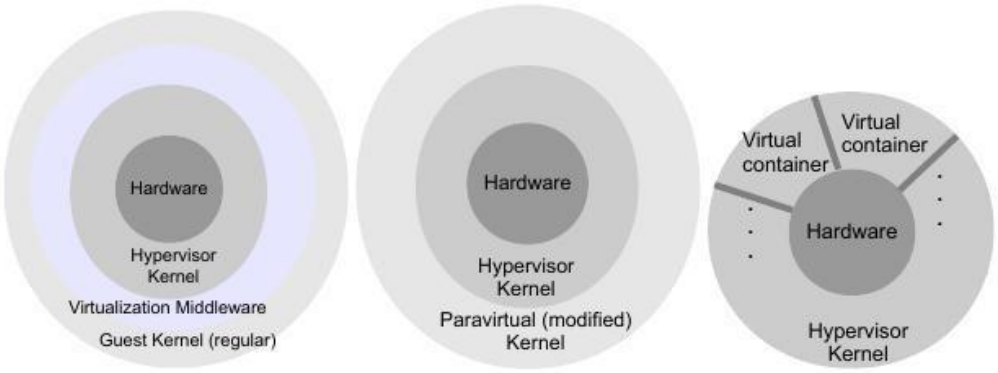


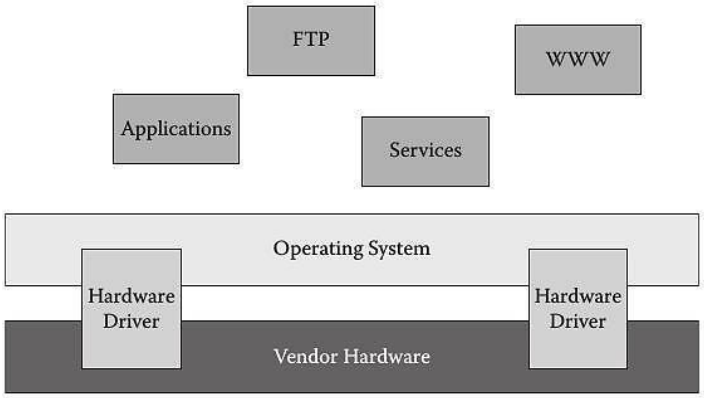
FIGURA 3-2: *HARDWARE EMULATION*, *PARAVIRTUALIZATION* Y *CONTAINER VIRTUALIZATION.*
FUENTE: [25]

El siguiente método de virtualización es la *paravirtualization*. En este método —al igual que en *hardware emulation*— se hace uso de *hypervisors*, con la diferencia de que emplea sistemas operativos con modificaciones; de esta manera, se evita utilizar instrucciones con privilegios de *root*. Es por ello que los sistemas operativos requieren modificados para saber que van a ser ejecutados desde *user space*.

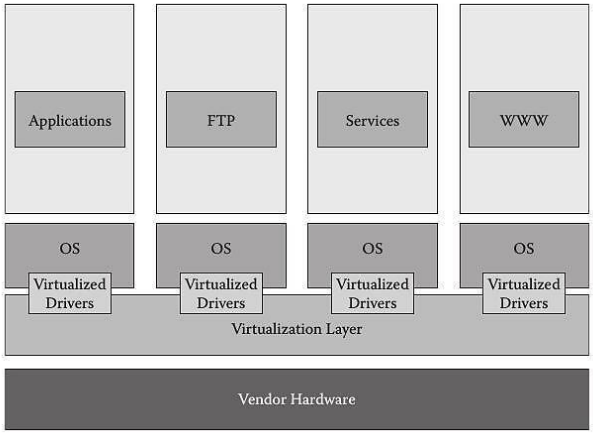
El último método de virtualización es *container virtualization*. En este, los sistemas operativos virtuales —también llamados *containers*— comparten el mismo *kernel*. Cada *container* dispone de recursos aislados y seguros ante fallas. Este método permite tener diferentes distribuciones de un sistema operativo, mas no diferentes arquitecturas, debido a que el *kernel* es el mismo para todos.

3.2.2. *Hypervisor*

Un *hypervisor* es un *software* que introduce una capa de virtualización entre el *hardware* del servidor y los diversos sistemas operativos que se ejecutarán en el entorno virtualizado [26]. Esta capa de virtualización contiene los *drivers* necesarios para controlar los recursos físicos del servidor y presentar *drivers* genéricos a los sistemas operativos de las máquinas virtuales. A continuación, se muestran imágenes de cómo interactúa la capa de virtualización con los sistemas operativos de los entornos virtuales.



(a)



(b)

FIGURA 3-3: ARQUITECTURA DE SERVIDORES: (a) ANTES Y (b) DESPUÉS DE LA VIRTUALIZACIÓN.
FUENTE: [26]

Por otra parte, existen dos tipos de *hypervisors*: *bare metal* (Tipo 1) y *OS hosted* (Tipo 2). El primero, se monta directamente en el *hardware* del equipo; el segundo, se carga en el sistema operativo que está corriendo en el *hardware* del servidor. Estos se muestran a continuación.

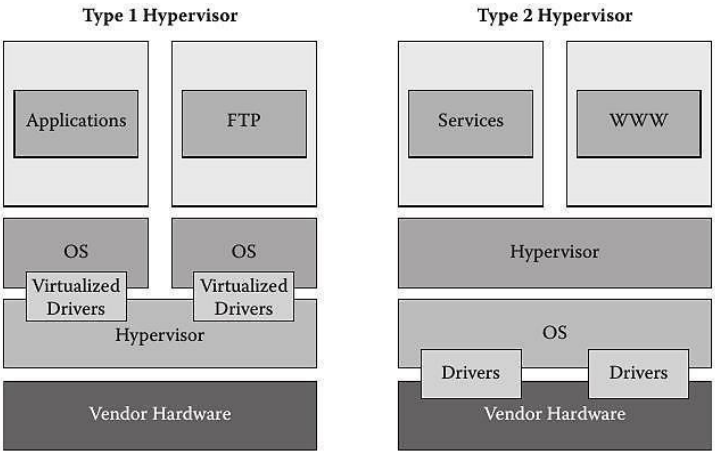


FIGURA 3-4: TIPOS DE *HYPERVISORS.*
FUENTE: [26]

En la siguiente tabla, se muestran los *hypervisors más* utilizados, indicando si son de *software* propietario y su tipo.

Hypervisor	**Propietario**	**Tipo**
VMware ESX/EXSi	Si	*Bare metal*
Microsoft's Virtual Server	Si	*OS hosted*
VMware Workstation	Si	*OS hosted*
Xen	No	*Bare metal*
Hyper-V	No	*Bare metal*
KVM	No	*OS hosted*

TABLA 3-5: *HYPERVISORS* MÁS USADOS CON SUS CARACTERÍSTICAS.
FUENTE: ELABORACIÓN PROPIA

3.2.3. Imágenes

Una imagen de disco es un archivo que contiene un disco virtual con un sistema operativo ejecutable [27], los cuales son utilizados por los *hypervisors* para poder inicializar las máquinas virtuales. Existen dos tipos de imágenes: *VM image* y *OS image*: la primera incluye el sistema operativo y todos los discos que estuvieron conectados al nodo virtual; la segunda, solo al sistema operativo [28]. Estas imágenes pueden tener diferentes formatos, los cuales se muestran en la siguiente tabla.

Formato	Descripción
Raw	Soportado por KVM y Xen
qcow2	Usado por KVM. Soportan *snapshots*
AMI/AKI/ARI	Usado por Amazon EC2. Indican la imagen de la VM, *kernel* y RAM, respectivamente
VMDK	Usado por VMware ESXi
VDI	Usado por VirtualBox
VHD	Soportado por Hyper-V
ISO	Usado para imágenes de CD y DVD

TABLA 3-6: FORMATOS DE IMÁGENES.

FUENTE: ELABORACIÓN PROPIA

3.3. *Network virtualization*

Network virtualization es otro enfoque de virtualización, en el cual se abstrae la red física que interconecta a los equipos y se monta una red virtual sobre ella. Si bien este tema incluye múltiples tecnologías (p.ej. VLAN, VRF, etc.), en las siguientes secciones se abordarán las tecnologías que permiten crear una red virtual sobre un entorno virtualizado (p.ej. *overlay networks*) y los eventos que afectan el desempeño de la red (p.ej. retardo, *jitter*, pérdida de paquetes, etc.). Estas tecnologías forman la base para la creación de los enlaces virtuales requeridos por un emulador de redes.

3.3.1. *Virtual switch*

Un *switch* virtual (vSwitch) es un programa que permite conmutar los paquetes de diversos equipos —en su mayoría máquinas virtuales— ya que este tipo de *switch* es utilizado principalmente en entornos virtualizados. También incluye funciones de VLAN *tagging*, *trunking*, *link aggregation*, entre otros. El vSwitch (ver figura 3-5) se conecta con las VMs a través de *virtual interface cards* (VIF) y puede comunicarse con equipos físicos a través de *physical virtual interface cards* (PIF).

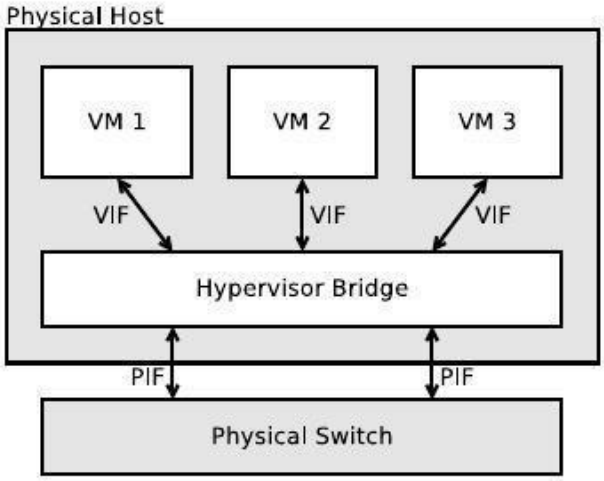


FIGURA 3-5: ESTRUCTURA DE UNA RED VIRTUAL CON UN *SWITCH* VIRTUAL.
FUENTE: [29]

Entre los vSwitches existentes para un *host* con Linux destacan *Linux Bridge*, *Macvlan Switch* y Open vSwitch.

Linux Bridge (figura 3-6) es un vSwitch que incluye funcionalidades de *Spannig Tree Protocol* (STP), L2 *firewall* a través de *ebtables*, y *traffic shaping*. Este vSwitch reside en el *kernel* de Linux, entre las NICs y el TCP/IP *Stack* del sistema operativo.

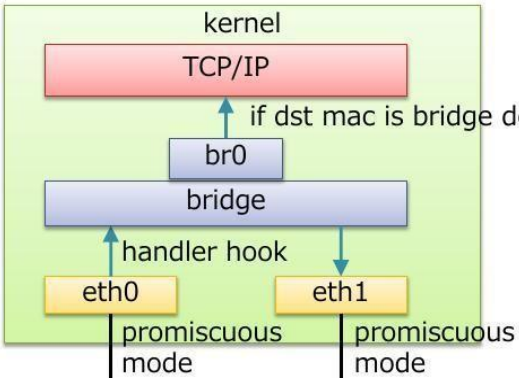


FIGURA 3-6: LINUX BRIDGE.
FUENTE: [30]

Macvlan switch (figura 3-7), un vSwitch que utiliza subinterfaces con direcciones MAC en vez de VLAN *tags*. Posee 4 modos de funcionamientos: VEPA, *Bridge*, *Private* y *Passthru*, de los cuales el modo *Bridge* es el más óptimo para el tráfico entre VMs.

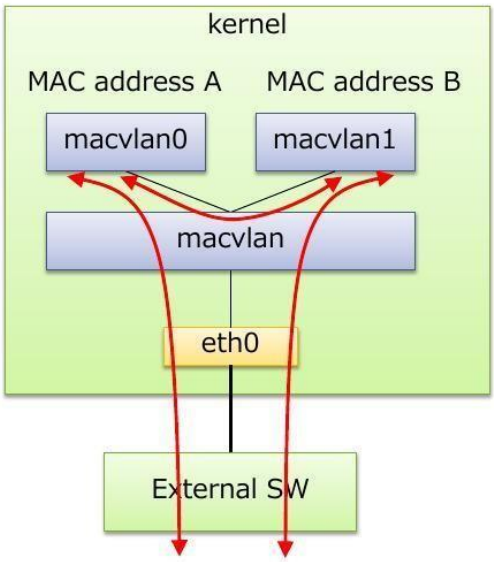


FIGURA 3-7: MACVLAN SWITCH.

FUENTE: [30]

Por último, Open vSwitch (figura 3-8) es un vSwitch que opera en base a *flows*, con el plano de control en *user space* y el plano de datos en el *kernel*. Soporta el protocolo OpenFlow y puede ser usado como un *switch* tradicional (VLAN *tagging*, túneles VXLAN y GRE, *bonding*, etc.).

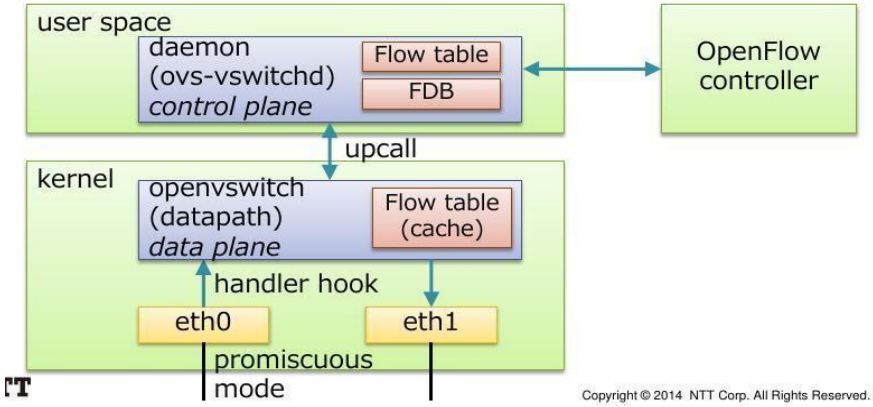


FIGURA 3-8: OPEN VSWITCH.

FUENTE: [30]

3.3.2. Túneles

Los túneles en redes son protocolos que permiten encapsular los paquetes de un protocolo en el *payload* de los paquetes de una red intermedia (*transit network*). Este mecanismo brinda una abstracción de la red subyacente, permitiendo la comunicación directa entre los extremos del túnel. Entre los túneles existentes, los más utilizados en entornos de *datacenter* son *Generic Routing Encapsulation* (GRE) y *Virtual Extensible* LAN (VXLAN).

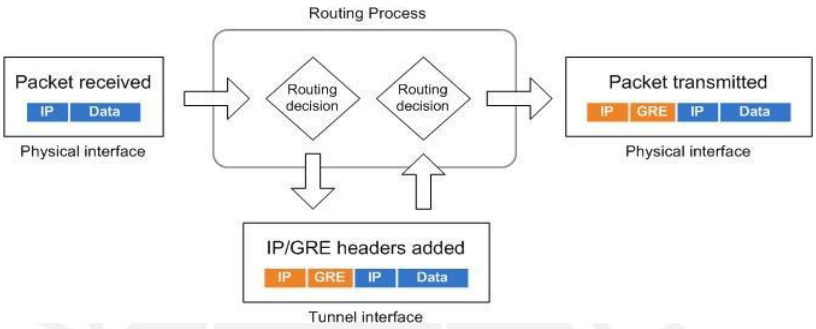


FIGURA 3-9: ESTRUCTURA DE UN TÚNEL.
FUENTE: [31]

GRE es un tipo de túnel punto a punto, en el cual se encapsula un paquete de un protocolo en otro (Figura 3-10). Se utiliza para presentar enlaces punto a punto a otras aplicaciones de red (p.ej. MPLS o Multicast) o para comunicar sitios con direcciones IP privadas [32] (IP *over* IP). También permite crear túneles L2 entre nodos (Ethernet *over* IP).

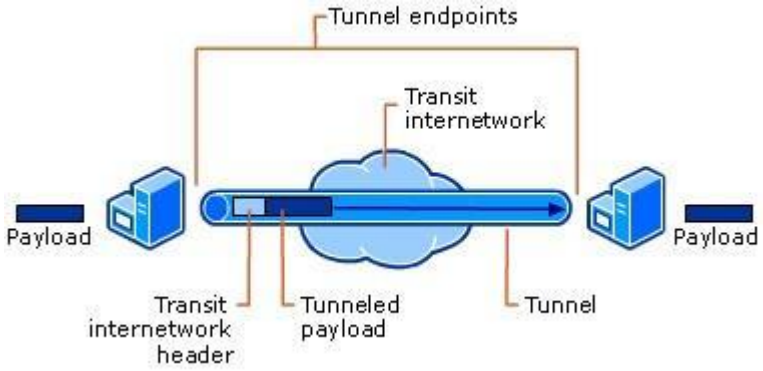


FIGURA 3-10: ENCAPSULAMIENTO DE IP SOBRE IP EN UN TÚNEL GRE.
FUENTE: [33]

Por el contrario, VXLAN es un tipo de túnel punto a multi-punto en el cual se encapsulan paquetes de capa 2 en datagramas de UDP. Esta tecnología presenta una mejora respecto a la tecnología de VLAN, pues ofrece identificadores de mayor tamaño (24 bits frente a los 12 de VLAN). Adicionalmente, permite interconectar redes de capa 2 a lo largo de redes de capa 3 utilizando *Multicast*, enviando los paquetes encapsulados hacia los *Virtual Tunnel Endpoints* (VTEP – extremos del túnel VXLAN) [34].

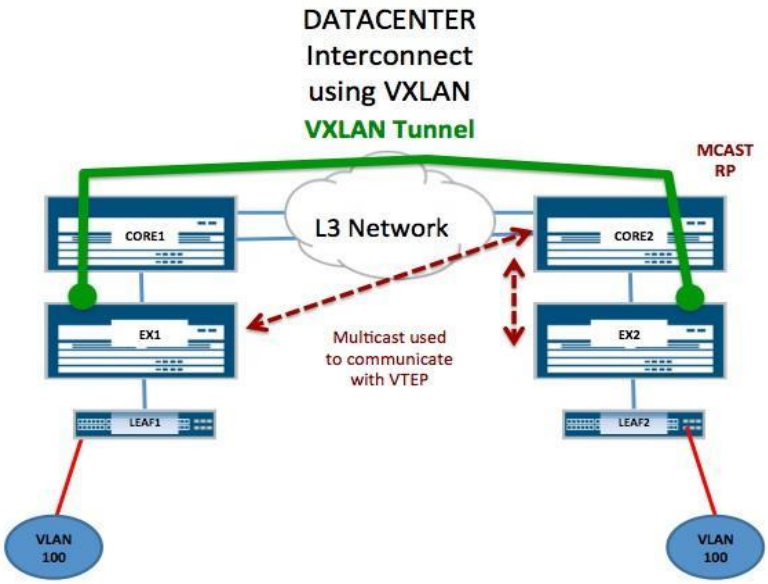


FIGURA 3-11: TÚNEL VXLAN ENTRE VTEP UTILIZANDO MULTICAST. FUENTE: [35]

3.3.3. *Overlay networks*

Overlay networking es un método de virtualización de redes en el que se crea una capa de abstracción de la red. Como se observa en la Figura 3-12, el presente método permite desplegar múltiples redes virtuales e independientes (*overlay network*) coexistiendo sobre una misma red física (*underlay network*). Esta abstracción facilita la creación de nuevos servicios y brinda un mayor control de la seguridad de la red.

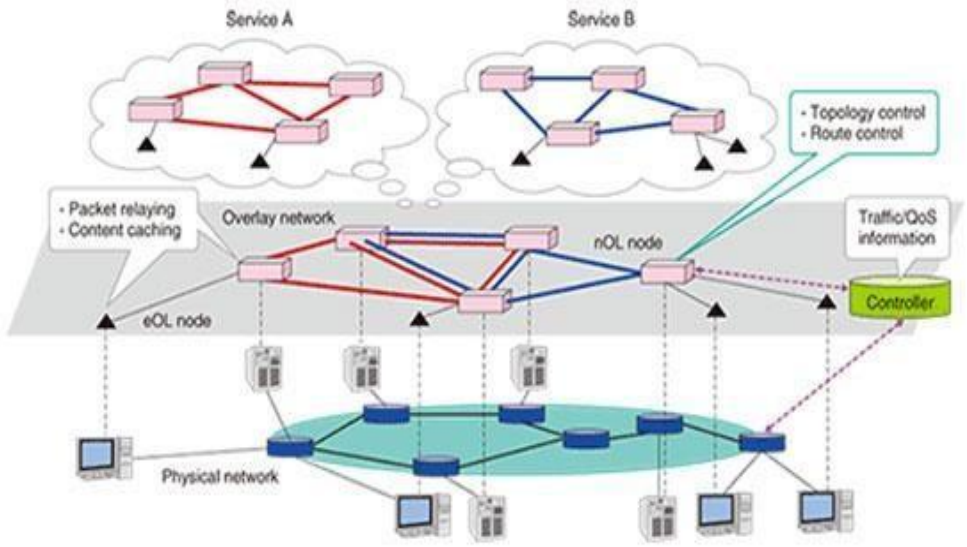


FIGURA 3-12: ESQUEMA DE UNA *OVERLAY NETWORK.*
FUENTE: [36]

Estas redes se forman a partir de protocolos que brindan una conexión virtual de dos *endpoints*. Estos *endpoints* pueden ser tanto equipos físicos (p.ej. servidores) como máquinas virtuales en un entorno de *Cloud Computing*. Los protocolos utilizados para crear las conexiones virtuales o túneles pueden ser GRE o VXLAN (basados en identificadores), así como IPSec en modo túnel (los endpoints deben autenticarse antes de usar la conexión).

3.3.4. *Link delay* y *bandwidth throttling*

En los entornos de Linux, existe un módulo del *kernel* llamado *Linux Traffic Control* (TC), el cual se encarga de proveer constantemente los paquetes desde la capa de IP hacia el *driver* de la interfaz de salida [37]. TC está compuesta por *queue disciplines* (qdisc), los cuales representan las políticas de manejo de una cola (de entrada o salida). Este puede estar implementada como FIFO (*First In First Out*), SFQ (*Stochastic Fairness Queueing*), TBF (*Token Bucket Filter*), entre otros.

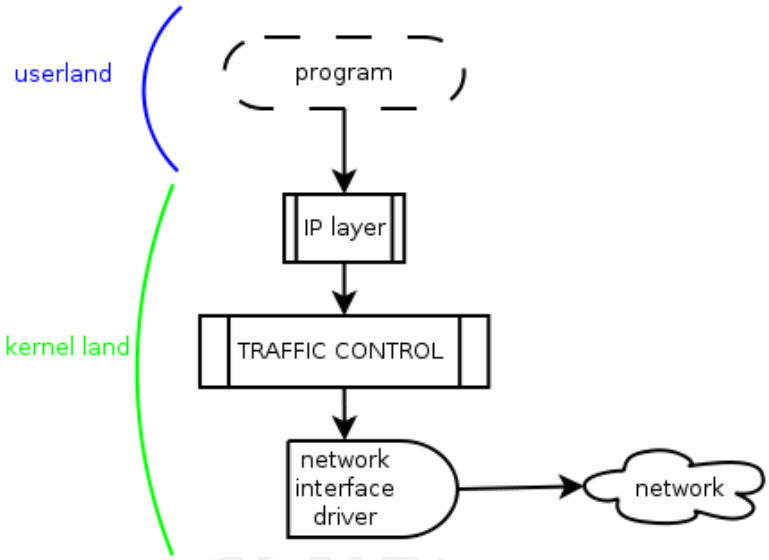


FIGURA 3-13: UBICACIÓN DEL BLOQUE DE *TRAFFIC CONTROL*.
FUENTE: [37]

Link delay o latencia es el tiempo que se tarda un paquete en llegar desde un nodo origen al destino. El retardo observado en el receptor incluye los tiempos de propagación de la onda (eléctrica o electromagnética), procesamiento del paquete y de las colas. En los entornos de Linux, se puede reproducir esta propiedad utilizando el módulo de *Network Emulation* (netem) en TC [38].

Por otra parte, *bandwidth throttling* es la reducción arbitraria de la tasa de bits en un enlace. Esta medida se emplea para regular el tráfico en un enlace y evitar la congestión del mismo. También se utiliza en escenarios donde se desea analizar el comportamiento de aplicaciones ante la reducción del ancho de banda.

En un entorno emulado, se puede replicar este comportamiento utilizando el algoritmo *Token Bucket* Filter (*TBF)* [38]. Este algoritmo autoriza el paso de paquetes de salida mediante *tokens* (ver figura 3-14). Estos *tokens* se renuevan a cierta tasa (*rate*) y se almacenan en un recipiente (*bucket*) de cierto tamaño (*size*).

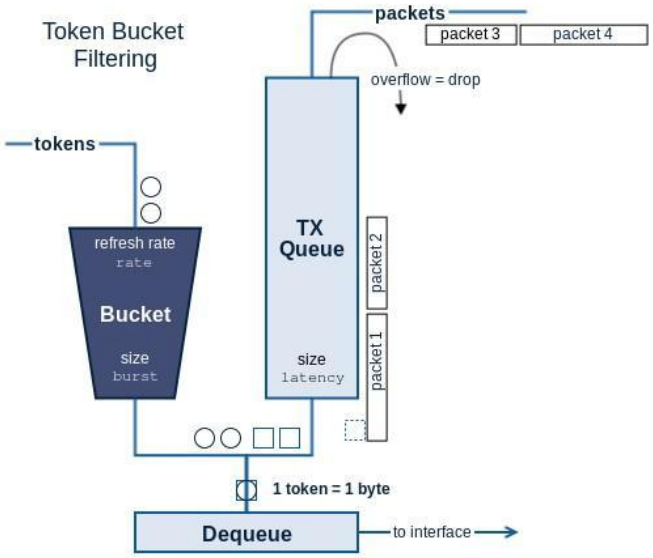


FIGURA 3-14: *TOKEN BUCKET FILTER.*

FUENTE: [38]

3.4. *Storage Virtualization*

El último enfoque de virtualización presentado es el de *storage virtualization*. Este enfoque permite abstraer los dispositivos de almacenamiento y presentar sistemas flexibles y robustos, obteniendo una mejor performance y mayor tolerancia a fallas (p.ej. RAID) [25]. Si bien existen otros tipos de *storage virtualization*, las siguientes secciones se centrarán en las arquitecturas de *data storage* basadas en su ubicación (en el mismo equipo o en *racks*) y en la forma de acceder a ellos (local o a través de una red).

3.4.1. Arquitecturas de *Data Storage*

Las necesidades de las empresas por capacidades de almacenamiento y la manera de acceder a ellos varían de acuerdo a los servicios que van a brindar. En algunos casos, no se requiere de una gran capacidad de almacenamiento, por lo que una solución local es suficiente (DAS); en otros, se requiere compartir un mismo dispositivo de almacenamiento entre varios equipos, por lo que una solución en red bastaría (NAS); por último, si se requiere de una gran capacidad de almacenamiento con aplicaciones que accedan de forma rápida a los datos almacenados, se requiere de una red dedicada para el almacenamiento (SAN). A continuación, se desarrollará cada una de estas arquitecturas de almacenamiento.

3.4.1.1. *Direct-Attached Storage*

Direct-Attached Storage es una arquitectura de *storage* que se caracteriza por tener los dispositivos de almacenamiento directamente conectados a los equipos. En un inicio, se usaban interfaces paralelas que se regían bajo los estándares SCSI (*Small Computer System Interface*) y ATA (*Advanced Technology Attachement*, también conocido como IDE) [25]. Sin embargo, la necesidad de aumentar la tasa de transferencia de datos de los dispositivos de almacenamiento volvió ineficientes estos estándares. Como solución, aparecen las interfaces seriales denominadas SAS (*Serial Attached* SCSI) y SATA (*Serial* ATA), las cuales reemplazan a las interfaces paralelas SCSI y ATA, respectivamente [25].

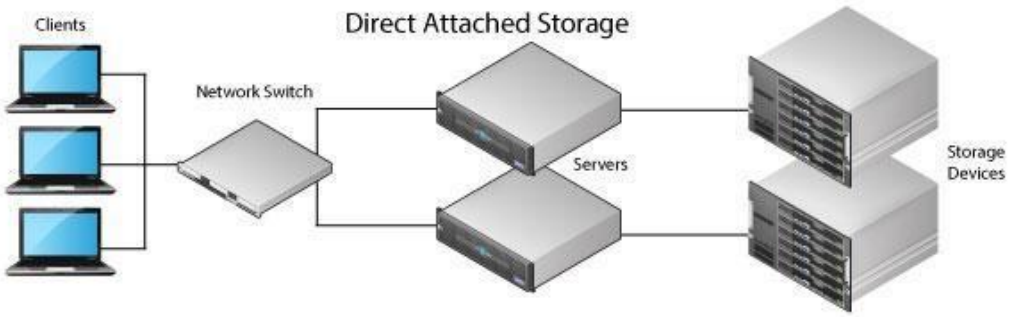


FIGURA 3-15: ARQUITECTURA *DIRECT ATTACHED STORAGE* (DAS).
FUENTE: [39]

3.4.1.2. *Network-Attached Storage (NAS)*

Network-Attached Storage es una arquitectura que se caracteriza por tener los dispositivos de almacenamiento conectados a una red LAN. Los sistemas NAS emplean un almacenamiento basado en archivos (*file-level storage*), por lo que el servidor NAS posee su propio sistema de archivos (*file system*) [25]. Otra característica de esta arquitectura es que los archivos y carpetas almacenadas pueden ser accedidos por varios clientes. Esta tecnología emplea protocolos de acceso como NFS (sistemas UNIX) y CIFS (Microsoft) [25].

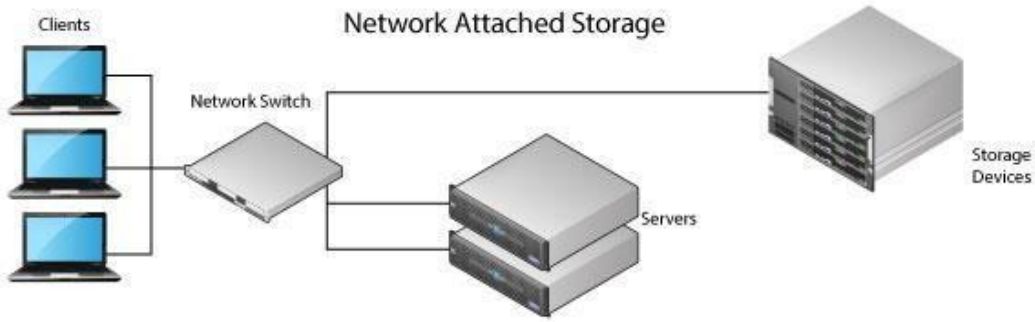


FIGURA 3-16: ARQUITECTURA *NETWORK-ATTACHED STORAGE* (NAS).
FUENTE: [39]

3.4.1.3. *Storage Area Network (SAN)*

Storage Area Network (SAN) es una arquitectura que provee un almacenamiento basado en bloques (*block-level access*). Debido a ello, en una SAN, los dispositivos de almacenamiento no requieren de un sistema de archivos (*file system*) propio [25]. Esta arquitectura presenta una red dedicada de alta velocidad para la comunicación entre los servidores y los dispositivos de almacenamiento. En esta red se emplea un protocolo (por lo general SCSI) para definir el método de transferencia de datos entre los servidores y los dispositivos de *storage*. SCSI opera sobre diferentes interfaces, siendo las más usadas *Fiber Channel* y iSCSI (ver figura 3-17).

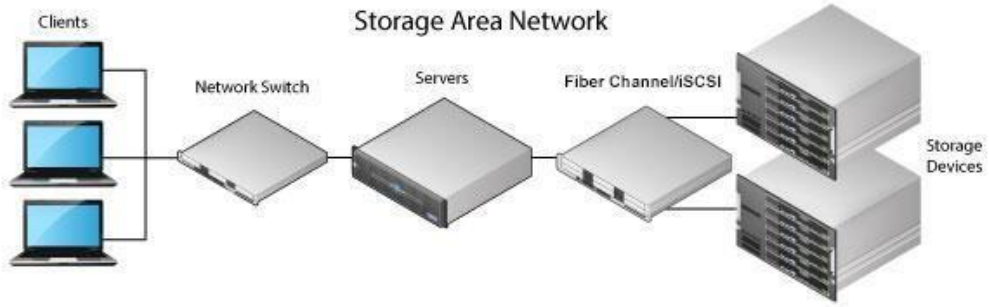


FIGURA 3-17: ARQUITECTURA *STORAGE AREA NETWORK* (SAN).
FUENTE: [39]

Fiber Channel (FC) es una tecnología de *networking* que permite implementar redes de alta velocidad (1 – 128 Gbps) y escalables, empleando su propia tecnología de red local (*switching fabric*) con ayuda de *switches Fiber Channel* [25]. Actualmente, existen dos variantes de *Fiber Channel*: FC *over* IP (FCIP) y FC *over* Ethernet (FCoE). Por un lado, FC *over* IP permite conectar más de dos FC SAN aisladas

mediante una red virtual FC sobre una red IP. Esta red IP se usa como transporte entre las FC SAN aisladas encapsulando los FC *frames* dentro de paquetes IP [25]. Por otro lado, FC *over* Ethernet (FCoE) transporta el tráfico de la FC SAN y de Ethernet en una misma infraestructura separadas lógicamente por VLAN IDs. FCoE utiliza Ethernet como red de transporte y encapsula los FC *frames* dentro de tramas Ethernet.

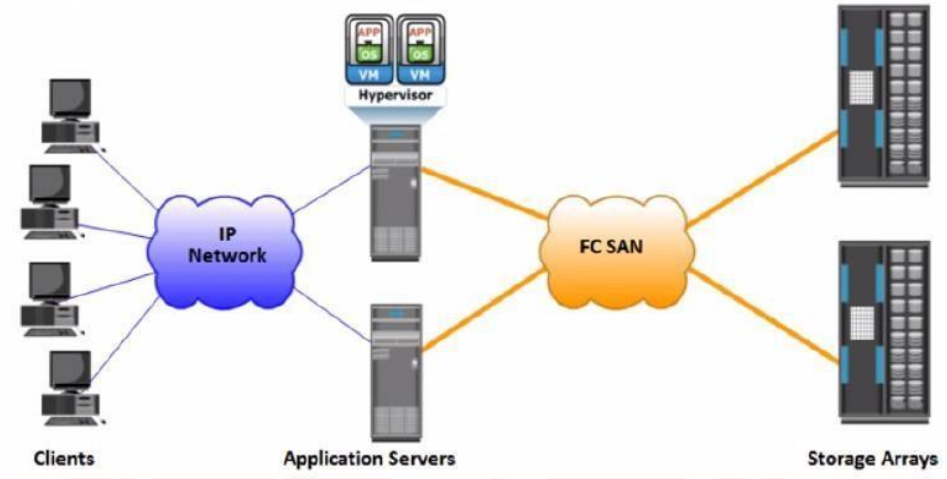


FIGURA 3-18: ESQUEMA DE FIBER CHANNEL EN SAN.
FUENTE: [25]

Internet Small Computer System Interface (iSCSI) es otra tecnología de red usada en SANs. Este protocolo utiliza una cabecera IP para encapsular los mensajes de SCSI sin pasar por las capas superiores de FC (ver figura 3.19).

Como se aprecia en la figura 3-19, se puede emplear una red IP como transporte entre los servidores y los arreglos de almacenamiento o se puede usar un *gateway* iSCSI en caso se esté usando el puerto FC en un *storage array*. La principal ventaja de usar iSCSI es que se puede reutilizar una red IP existente, lo cual reduce los costos de implementación, además de utilizar una tecnología madura y robusta como IP [25].

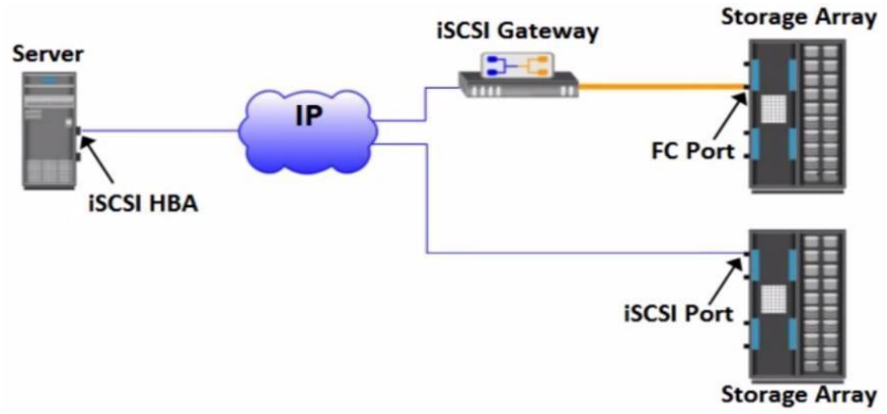


FIGURA 3-19: ESQUEMA DE ISCSI EN SAN.

FUENTE: [25]

En la figura 3-20 se muestra la relación entre iSCSI y otras diferentes tecnologías de transporte (FC, FCIP, FCoE).

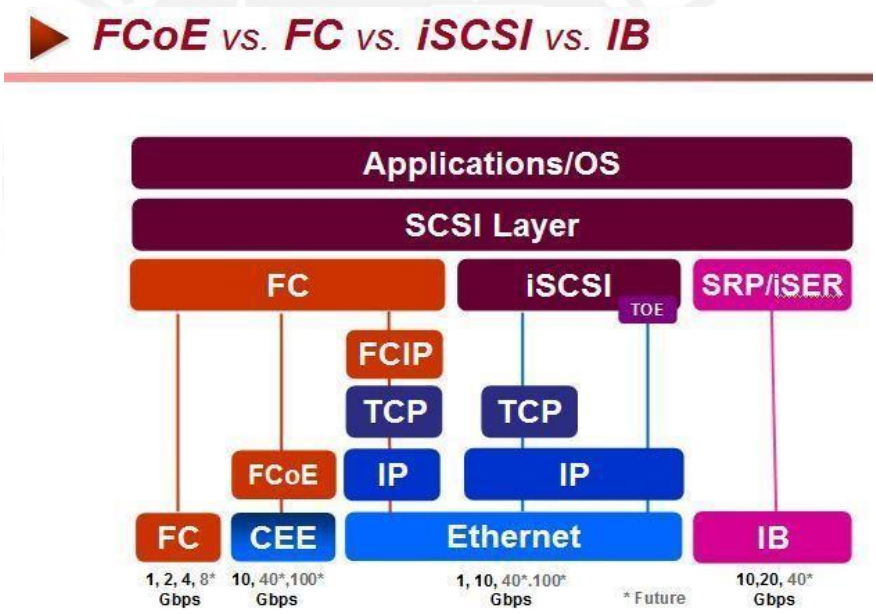


FIGURA 3-20: RELACIÓN ENTRE FC, FCIP, FCoE, e iSCSI.

FUENTE: [40]

3.5. Monitoreo de Infraestructura

El monitoreo de una red se realiza mediante el uso de equipos gestores, los cuales forman parte de la infraestructura de red y permiten tener una visualización completa

del estado de los equipos y los recursos de red. Esta visualización ayuda en la detección de fallas de equipos, sobrecarga de procesadores y memoria, congestiones de enlaces, entre otros. La gestión de redes utiliza un modelo denominado FCAPS (*Fault, Configuration, Accounting, Performance, Security*), el cual define las tareas de la gestión de redes. En la tabla 3-7 se explica con más detalle este modelo.

Modelo FCAPS		
Tarea	**Descripción**	**Funciones**
Fault Management	Permite localizar e identificar errores de manera eficiente para minimizar el tiempo de corte.	• Generación, comprobación, filtrado, manejo y detección de alarmas. • Pruebas de diagnóstico. • Manejo, estadísticas y registro de errores.
Configuration Management	Permite automatizar las configuraciones de los equipos de manera que entren en funcionamiento lo más rápido posible.	• Recursos de inicialización. • Copia de seguridad y restauración. • Automatización de distribución de *software.*
Accounting Management	Permite almacenar información sobre la utilización de recursos en base a diversos parámetros.	• Costo de los servicios. • Contabilidad límite. • Uso de las cuotas. • Combinación de los costos de múltiples recursos.
Performance Management	Consiste en el monitoreo de manera remota de las estadísticas de las alarmas, capacidad y rendimiento.	• Nivel consistente de performance. • Realización de análisis de datos. • Reporte de problemas. • Generación de informes de rendimiento.
Security Management	Consiste en la autenticación e identificación para acceder a la red.	• Registros de acceso. • Protección de datos y acceso restrictivo a los recursos. • Control de los derechos de acceso de usuario, seguridad de alarma.

TABLA 3-7: MODELO FCAPS.

FUENTE: [41], [42]

3.5.1. Monitoreo de Infraestructura Virtual

El monitoreo de una infraestructura virtual es similar al de una infraestructura real. Su función principal es asegurar que las máquinas virtuales tengan una alta disponibilidad y una máxima performance. Para ello, existen diversos *softwares* que permiten monitorear el estado de las VMs, identificando qué recursos se están utilizando (CPU, memoria, almacenamiento, red), así como qué VMs los están empleando. También permiten visualizar los recursos consumidos mediante gráficos y, en algunos casos, identificar el componente de *hardware* defectuoso que pueda estar afectando el rendimiento de las máquinas virtuales.

3.5.2. Herramientas de Medición

Una vez implementada cualquier red, es necesario entender y analizar su funcionamiento; en otras palabras, entender cómo se caracterizan sus enlaces, el tipo de tráfico que está circulando, etc. Existen ciertas herramientas de medición para obtener toda esta información, entre las cuales resaltan iperf y tcpdump. A continuación, se realizará una explicación más a fondo de cada una.

3.5.2.1. Iperf

Iperf es un programa definido por el esquema cliente-servidor y se utiliza para caracterizar un camino (conjunto de enlaces) entre 2 nodos. Este programa permite generar 2 tipos de tráfico: *Transmission Control Protocol* (TCP) y *User Datagram Protocol* (UDP). Con tráfico TCP se puede estimar la tasa de bits máxima permitida en el enlace entre el cliente y el servidor basados en el tamaño de la ventana TCP. Por otro lado, con tráfico UDP se puede estimar el porcentaje de pérdida de datagramas, el *jitter* presente en el camino y la cantidad de datagramas recibidos fuera de orden; todo esto basado en el tamaño del datagrama.

3.5.2.2. Tcpdump

Tcpdump es un programa empleado como *sniffer* de paquetes de red: se utiliza para analizar los paquetes que circulan por la red donde se encuentra conectado el equipo. Este programa funciona en la mayor parte de sistemas operativos basados en UNIX. Con esta herramienta, el usuario puede capturar y observar los paquetes transmitidos y recibidos por la tarjeta de red de su dispositivo, para posteriormente analizarlos y

determinar si existe algún error en la comunicación. Tcpdump utiliza *libpcap*, una librería portable basada en C/C++ para realizar la captura de paquetes.

3.6. Orquestador: OpenStack KILO

Como se indica en [43], OpenStack es una plataforma de *Cloud Computing* que provee una solución de *Infrastructure-as-a-Service* (IaaS) a través de servicios (también llamados proyectos) complementarios. Cada uno de estos proyectos presenta una *application programming interface* (API) que facilita la integración entre proyectos. Esta plataforma puede ser desplegada en la mayoría de distribuciones de Linux, trabaja bajo la licencia de código abierto Apache 2.0 y es administrado por OpenStack Foundation.

La 11va versión de OpenStack, también llamada Kilo, fue lanzada el 30 de abril del 2015. Esta versión ofrece una mayor estabilidad y escalabilidad entre los servicios *core*, así como el *release* del *bare metal service* llamado Ironic [44]. Este servicio permite proveer *bare metal machines* en vez de máquinas virtuales. También permite desplegar entornos de máquinas virtuales desde una plantilla utilizando el servicio de Heat, manejar direcciones tanto IPv4 como IPv6, mover volúmenes de almacenamiento entre proyectos, encriptarlos y extender su tamaño.

3.6.1. Arquitectura

La arquitectura escogida para implementar OpenStack necesita previamente de los requerimientos de los usuarios de *cloud*, de esta forma se podrá determinar la configuración más adecuada para el entorno: se debe estimar la cantidad de servidores necesarios y sus características (RAM, procesador, *storage*), así como los servicios de OpenStack requeridos para su funcionamiento. Una vez que se tiene dicha información, se pueden plantear diferentes arquitecturas a implementar: se puede proponer una arquitectura mínima con los *core services* de OpenStack, así como una arquitectura que presente todos los servicios (principales y opcionales) con redundancia de servidores por servicio, entre otras.

3.6.1.1. Arquitectura de *Hardware*

Como se mencionó anteriormente, se debe determinar la cantidad de servidores que tendrá el sistema y qué servicios de OpenStack se ejecutarán en él antes de

desplegar el *rack*. Como se observa en la figura 3-21, se debe considerar cuáles serán los nodos de control, *network, compute* y los nodos opcionales de *block* y *object storage*.

En el nodo de control, como su nombre lo indica, reside el *software* encargado de controlar el funcionamiento de todo el sistema de OpenStack; el nodo de *network* se encarga de proveer servicios de red (p.ej. DHCP, NAT, *firewall*, enrutamiento dentro y hacia fuera del *rack*) a las máquinas virtuales; en el nodo de *compute* se encuentra el *hypervisor* que permite emular las máquinas virtuales; y en los nodos de *storage*, se almacena tanto datos estructurados (*block*) como no estructurados (*object*).

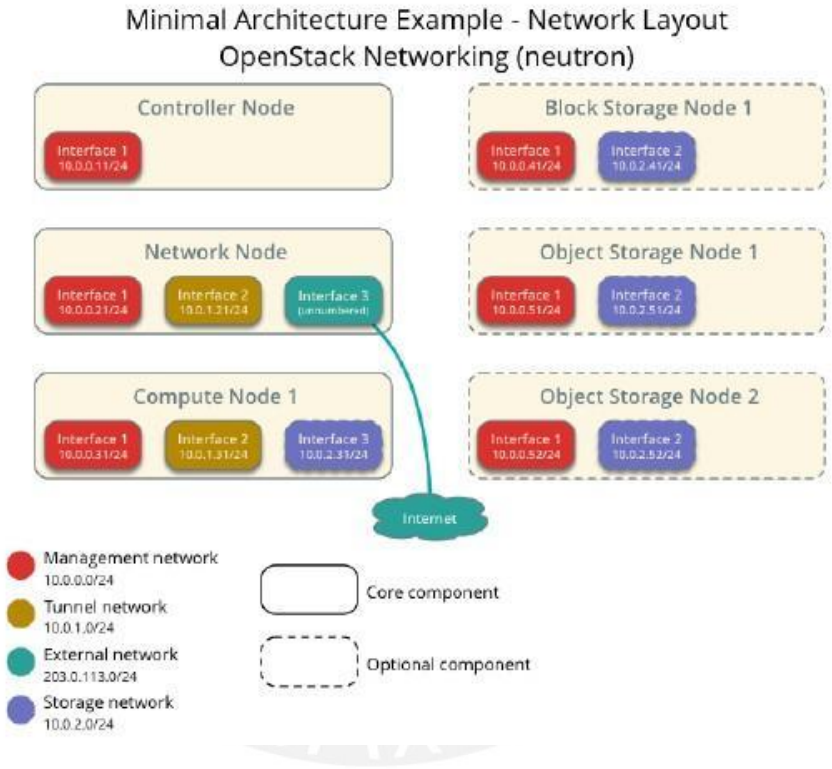


FIGURA 3-21: ARQUITECTURA DE *HARDWARE* DE OPENSTACK.
FUENTE: [43]

En cuanto a las interconexiones, existen 4 tipos de redes: administración, datos, externa y SAN. La red de administración (*management network*) transporta el tráfico de control del sistema entre todos los nodos; la red de datos (*tunnel network*) se encarga de transportar el tráfico de usuario de las máquinas virtuales; la red externa (*external network*) brinda conectividad a las máquinas virtuales con redes externas al *rack* de OpenStack (p.ej. internet), y la red SAN (*storage network*) permite conectar

las máquinas virtuales con los dispositivos de almacenamiento.

3.6.1.2. Arquitectura de Servicios

La plataforma de *Cloud Computing* OpenStack presenta la una arquitectura basada en diversos servicios, los cuales están implementados en el lenguaje de programación Python.

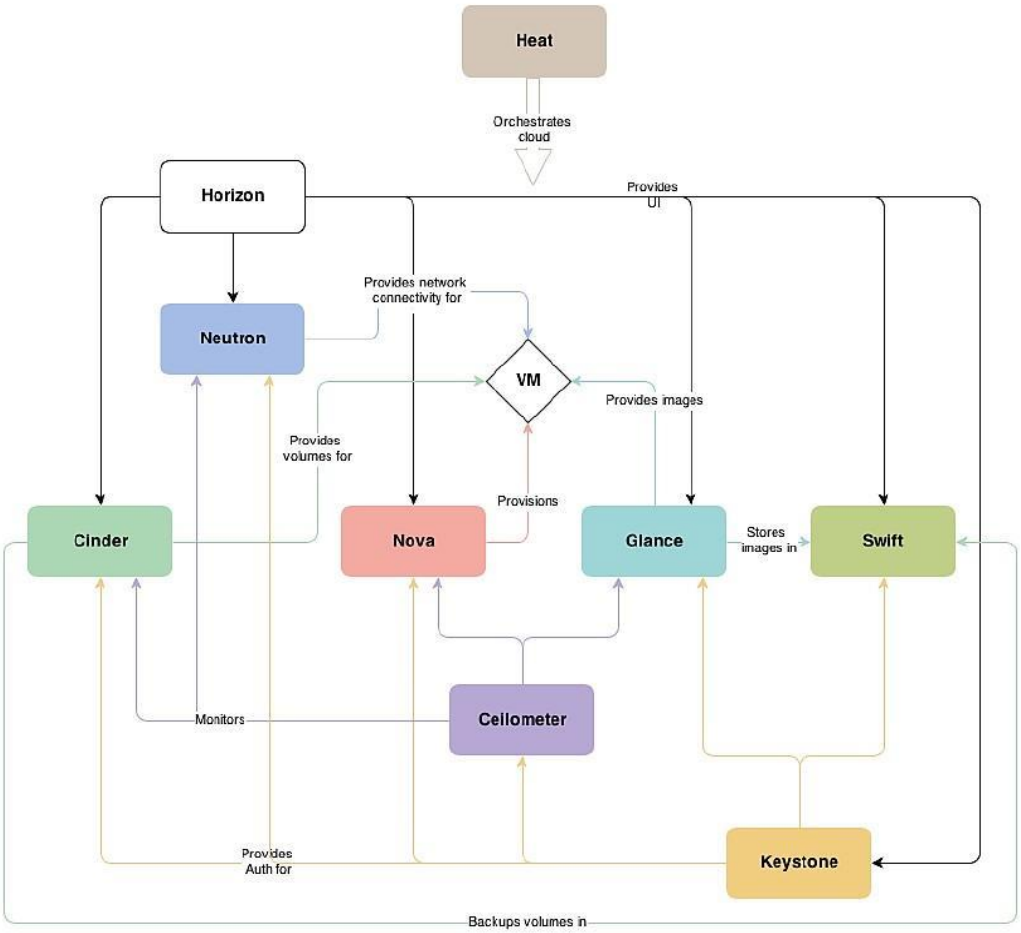


FIGURA 3-22: ARQUITECTURA DE *SOFTWARE* DE OPENSTACK.
FUENTE: [46]

Como se muestra en la figura 3-22, todos los servicios están relacionados e interactúan entre sí para lograr la creación de una máquina virtual: Horizon provee una interfaz web que permite manipular los recursos de cómputo, red y almacenamiento de forma gráfica; Neutron provee la abstracción de una red a las máquinas virtuales y les brinda conectividad con otros equipos; Cinder provee

volúmenes de almacenamiento para las VM; Nova brinda recursos de cómputo para emular las máquinas virtuales; Glance permite crear y cargar imágenes de discos que pueden utilizar las VMs; Swift permite almacenar los *backups* de volúmenes de almacenamiento, así como las imágenes de disco cargadas en OpenStack; Ceilometer permite monitorear los recursos disponibles y utilizados para realizar una tarificación;y Keystone provee autenticación y autorización a todos los servicios.

3.6.2. Descripción servicios

Como se mostró en la imagen anterior, el *software* OpenStack consiste en la interacción de diferentes servicios cuya finalidad es la creación de máquinas virtuales, con conexión a una red y disponibilidad de almacenamiento. A continuación, se presenta una tabla resumen que detalla todos los servicios ofrecidos en OpenStack.

Adicionalmente, existen servicios que no son propios de OpenStack, pero son complementarios a los explicados anteriormente. Entre ellos se encuentran los servicios de *Message Queue*, *Database* y *Network Time Protocol*.

3.6.2.1. *Message Queue*

El servicio de *message queue* permite la comunicación y notificación entre proyectos de OpenStack, facilitando las funciones de comando y control en las diferentes arquitecturas desplegadas [45]. Existen 2 tipos de *back-ends* para este servicio: *Advanced Message Queuing Protocol* (AMPQ), el cual provee colas de mensajes para la comunicación punto a punto entre servicios (p.ej. RabbitMQ y Qpid); y la clase de ZeroMQ, el cual brinda una comunicación punto a punto directa entre servicios a través de *sockets* TCP.

3.6.2.2. *Database*

El uso de base de datos para almacenar información del sistema es parte fundamental de toda la arquitectura. En el caso de OpenStack, el tipo de base de datos empleada es SQL (por lo general MariaDB o MySQL). En ella se almacena información acerca de los *tenants*, usuarios, máquinas virtuales, entre otros. Por otro lado, es importante que este servicio sea a prueba de fallas y se dimensione correctamente la cantidad de conexiones permitidas y activas en la base de datos.

3.6.2.3. *Network Time Protocol*

El protocolo NTP permite mantener los servicios de OpenStack sincronizados; para lograr ello, se recomienda configurar un servidor NTP en el nodo de control (sincronizado con un servidor NTP externo o de la organización de menor nivel) y luego sincronizar los demás nodos (*compute*, *network*, *storage*) con el nodo de control. Es necesario que los todos nodos estén sincronizados para evitar diferentes tipos de errores: en el *scheduling* de máquinas virtuales, en el almacenamiento de datos en *object storage*, al agregar *timestamps* en el *debugging* [46], etc.

Servicio	Nombre de Proyecto	Descripción
Dashboard	Horizon	Provee una interfaz web que permite gestionar, de forma gráfica, los recursos disponibles en OpenStack.
Compute	Nova	Se encarga la creación, eliminación y monitoreo de las máquinas virtuales presentes en el sistema.
Networking	Neutron	Se encarga de brindar los recursos de red (puertos, subredes, etc.) a las máquinas virtuales
Storage		
Object Storage	Swift	Su función es almacenar y recuperar *data* no estructurada. Posee un mecanismo de réplica de información que lo hace a prueba de fallas.
Block Storage	Cinder	Este servicio provee un *block-storage* dedicado para las VMs. Asimismo, permite la creación y gestión de los dispositivos de almacenamiento.
Shared Services		
Identity Service	Keystone	Su función principal es ofrecer autenticación y autorización para todos los servicios que tiene OpenStack.
Image Service	Glance	Se encarga de almacenar y recuperar las imágenes de los discos que utilizan las máquinas virtuales.

TABLA 3-8: SERVICIOS DE OPENSTACK CON UNA BREVE DESCRIPCIÓN.

FUENTE: [43]

4. Diseño del emulador

En la sección anterior, se habló acerca de la virtualización y las tecnologías que permiten crear recursos virtuales a partir de servidores, redes y dispositivos de almacenamiento. Luego, se mencionó al *software* de *Cloud Computing* OpenStack, el cual permite crear servicios de IaaS; en base a estas herramientas, es posible crear un emulador de redes que permita probar tanto arquitecturas *legacy* como futuras que vayan apareciendo.

Antes de empezar con el diseño del emulador, es necesario plantear los requerimientos del mismo; estos se basan tanto en las limitantes encontradas como en funcionalidades rescatables de los entornos de prueba que existen en la actualidad. En base a los objetivos (establecidos al final del capítulo 2) y requerimientos definidos, se presenta la arquitectura y el diseño del emulador. La arquitectura exhibe una estructura modular, donde se muestran las diferentes etapas del emulador y el flujo e interacción de cada *feature* desde el punto de vista del usuario. Adicionalmente, se explican las modificaciones hechas a los servicios de

computing y *networking* de la plataforma de *Cloud Computing* (versión Kilo) para adaptarlos a nuestros requerimientos.

En las últimas secciones, se menciona el módulo de validación. Este módulo proporciona una nueva funcionalidad en el emulador, que sirve como mecanismo de protección para evitar saturar el bus de procesamiento y la memoria de los servidores empleados. Asimismo, en la sección de la interfaz de usuario, se explicará el funcionamiento completo del emulador. Esto incluye la definición de topologías (ya sean pre-definidas o personalizadas), visualización de topologías, el funcionamiento de los *features* de SDN *control plane* y simulación de enlaces punto a punto.

4.1. Consideraciones y requerimientos

En el segundo capítulo, se realizó un análisis entre los tres tipos de emuladores; en este, se encontró que los principales problemas de los emuladores eran la cantidad de recursos disponibles para el experimento, las arquitecturas que podían ser probadas y el acceso a dicho emulador. Del primer problema, se desprenden los tres requerimientos del emulador: que sea escalable, que los resultados obtenidos sean similares a los que se consiguen en redes físicas (alta fidelidad) y que el sistema sea robusto ante fallas; de los dos siguientes, se formularon las consideraciones del emulador: soportar diferentes arquitecturas y ser implementado en plataformas de OpenStack (por ser *software* libre). En las siguientes secciones, se explicarán dichos requerimientos y consideraciones del emulador de redes.

4.1.1. Escalabilidad

Para que el sistema diseñado se considere escalable, se establecen dos condiciones: emular diferentes tipos y versiones de *kernels* (p.ej. Windows, Ubuntu, CentOS); y emular una red metropolitana de aproximadamente 50 nodos con enlaces hasta 1Gbps. Por tal motivo es necesario un entorno que disponga de una gran cantidad de recursos para poder escalar en el número de nodos.

De los diferentes entornos que existen, los modelos de *Cloud Computing* son los que cumplen con dichas características. Estos modelos utilizan servidores dedicados a la emulación y *storage*, además de una red de alta velocidad para el plano de datos. Actualmente, la Pontificia Universidad Católica del Perú cuenta con un *rack* de 8 servidores con 20 *cores* cada uno, interconectados por una red de 10 Gbps en el

Grupo de Investigación de Redes Avanzadas (GIRA). Dado que se cuenta con un *rack* de dichas características, se requiere que el emulador pueda funcionar sobre dicha plataforma.

Una vez desplegado el emulador, crear redes de gran tamaño puede resultar una tarea larga y tediosa de realizar: añadir nodos, interconectarlos de cierta manera —o incluso borrarlos— puede consumir tiempo a los usuarios, además de añadir cierta complejidad para definir la topología final. Puesto que el objetivo del emulador es probar nuevas arquitecturas de redes, se deben incluir *features* que permitan crearlas de forma rápida sin importar el tipo de red (proveedores de servicios, *data centers*, plano de control separados del plano de datos, etc.).

4.1.2. Fidelidad

Otro de los requerimientos del emulador es que los resultados de los experimentos que se realicen en este entorno sean numéricamente similares de los obtenidos en redes con equipos físicos. Con resultados precisos en la etapa de emulación, es posible comprobar el funcionamiento de la red y detectar errores en ella. Esta característica es esencial para que el emulador sirva como una etapa de pruebas de los prototipos de redes, antes de desplegarlas.

Para ello, el emulador debe reproducir equipos de capa 2 del modelo OSI (L2) y superiores. Entre los equipos consignados para las pruebas, se encuentran *switches* y *routers*, ambos basados en código abierto (Linux) y de otros fabricantes (p.ej. Cumulus). Asimismo, debe ser posible reproducir eventos como retardo o límite de *bit rate* en los enlaces entre equipos. El realismo y los resultados de las pruebas dependen no sólo del funcionamiento de los equipos, sino de las condiciones en las que operan; por lo tanto, al introducir un control sobre el retardo y *bit rate*, se puede evaluar tanto el desempeño de la red como las aplicaciones sobre ella.

Por otro lado, la precisión de los resultados del emulador depende de la utilización de los recursos de los servidores: el emulador no debe permitir que se creen topologías cuyos nodos (en total) sobrepasen la cantidad de memoria, CPU y almacenamiento disponibles en los servidores. Es por ello que un paso previo al funcionamiento del emulador es la calibración del mismo, con respecto al *hardware*. Esto permite hallar los valores máximos de recursos que puede utilizar el emulador, los cuales se emplearán en la etapa de validación.

4.1.3. Robustez

El último requerimiento del emulador es que pueda seguir funcionando ante la presencia de posibles fallas en las máquinas virtuales. Esto permite que, ante alguna eventualidad, los usuarios puedan continuar con el experimento de forma correcta. Para ello, se requiere de un enfoque proactivo: se busca prevenir las fallas en vez de lidiar con ellas una vez que sucedan. De esta forma, se reduce la posibilidad de encontrar errores de funcionamiento del emulador.

Como se explicó en el punto de fidelidad, antes de ejecutar el emulador se debe realizar una calibración en base a los recursos disponibles de los servidores. Con esta calibración, se pueden modificar las cuotas de OpenStack y los límites del validador (posteriormente explicado) para que el sistema no exceda los recursos disponibles. Cabe resaltar que este paso depende de la capacidad de cada *rack* donde resida el emulador, por lo que no es un requerimiento realizarlo de forma automática.

Una vez definidas las cuotas de OpenStack, se debe incluir una sección de validación de los recursos en la lógica del emulador. Esta sección emplea los recursos disponibles de los servidores y las condiciones de la topología de prueba, para indicar si es posible emularla. Si bien con la calibración del emulador respecto al *hardware* se definen los límites de recursos del proyecto, es mandatorio asegurar que el emulador funcione correctamente en todo momento. La sección de validación debe asegurar dos cosas: el ingreso de valores válidos y —principalmente— la asignación de recursos necesarios para un correcto funcionamiento en el emulador.

Tomando en consideración que un entorno de pruebas de redes debe ser escalable, robusto y con resultados fieles a los obtenidos con equipos reales, en las siguientes secciones se va a presentar el emulador realizado. Este emulador cumple con los requerimientos y consideraciones mencionados para evaluar diferentes arquitecturas de red.

4.2. *High Level Design*

Con una visión más concreta de los requerimientos y objetivos del emulador, se procede a presentar su arquitectura. Esta se encuentra definida por tres niveles: Usuario, Código e *Hypervisor*. En las siguientes secciones se analiza la estructura

de cada nivel, el funcionamiento desde el punto de vista del usuario, así como se indica la interacción entre los diferentes módulos que posee y en qué momentos se activan. Asimismo, se explican las diferentes herramientas que se usaron para implementarlo.

4.2.1. Arquitectura

Para estructurar la arquitectura del emulador, se decidió agrupar los diferentes módulos dentro de bloques específicos. Cada bloque indica uno de los tres niveles principales: Nivel de Usuario, Código e *Hypervisor*; como se muestra en la figura 4-1, estos niveles permiten organizar los módulos del emulador de acuerdo a la interacción del código con los usuarios y OpenStack.

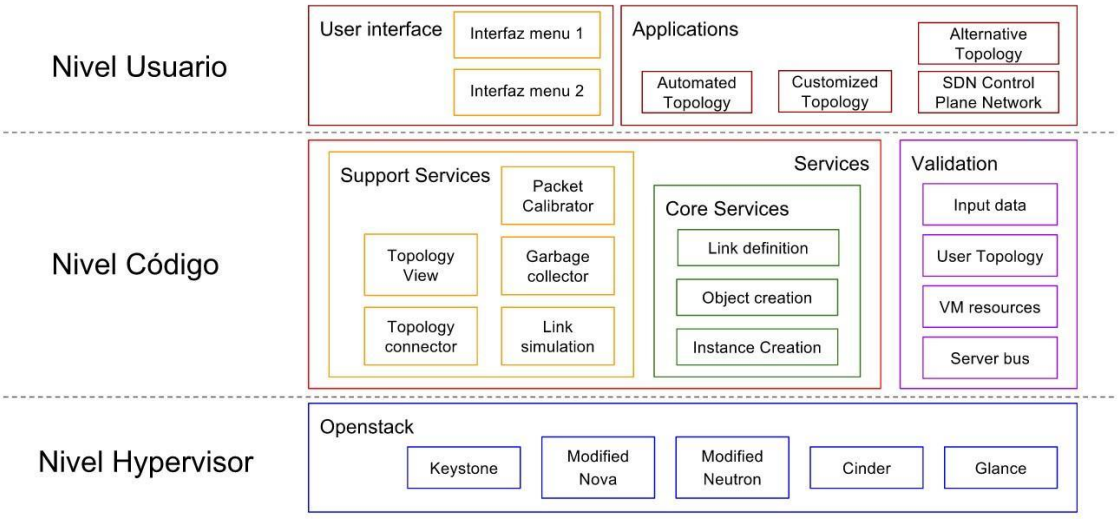


FIGURA 4-1: ARQUITECTURA DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

En la base de la arquitectura se encuentra el nivel *Hypervisor*, el cual está conformado por los servicios de OpenStack (figura 4-2). Como se explicó en el capítulo anterior, los servicios mostrados (Keystone, Nova, Neutron, Cinder y Glance) son mandatorios para la creación y funcionamiento de máquinas virtuales. Por la forma cómo están implementados los servicios de Nova y Neutron, se presentaron dificultades en el desarrollo el emulador. Por esta razón, se realizaron modificaciones a dichos servicios, aprovechando que esta plataforma es *open source*. Estas modificaciones se detallarán más adelante.

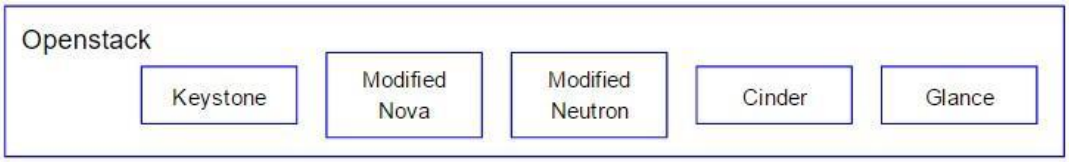


FIGURA 4-2: NIVEL DE *HYPERVISOR*.
FUENTE: ELABORACIÓN PROPIA

El siguiente nivel de la arquitectura es el de Código (figura 4-3), el cual se caracteriza por estar desarrollado usando el lenguaje de programación Python. Este nivel cuenta con dos grupos de módulos: Servicios y Validador.

El grupo de servicios está conformado por módulos principales y soporte. Los servicios principales definen el funcionamiento del emulador mediante tres módulos, los cuales realizan la lógica de creación de topologías en Python e interacción con el nivel *Hypervisor* para implementarlas en OpenStack. Por otro lado, los servicios secundarios proporcionan módulos de apoyo para utilizar de mejor manera las topologías y borrar de la memoria los nodos eliminados.

Se debe mencionar que los módulos *Topology view* y *Link simulation* utilizan herramientas externas a Python: para el primero, se reutilizó el servidor Apache —utilizado por OpenStack— para su interfaz web. Se añadió un archivo html en donde se pueden visualizar las topologías creadas; para el segundo, se cambió la cola por defecto de Linux (pfifo_fast) por TBF para limitar el *bit rate* de los enlaces. Adicionalmente, se utilizó la cola de *Netem* para introducir retardo en los mismos.

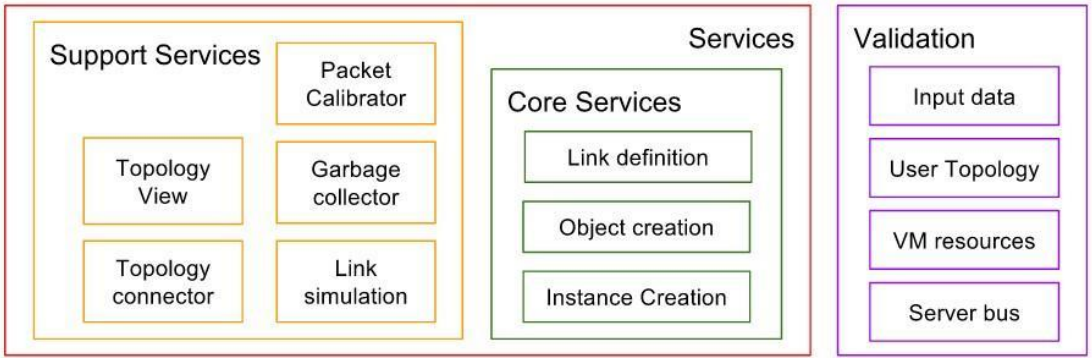


FIGURA 4-3: NIVEL DE CÓDIGO.
FUENTE: ELABORACIÓN PROPIA

El otro grupo de módulos en este nivel conforman la Validación, la cual proporciona mecanismos de protección al emulador. Estos módulos corroboran los datos usados para definir las topologías (datos de entrada) y evitan que el proyecto supere los recursos delimitados tanto por OpenStack como por el mismo *hardware* sobre el cual está funcionando el emulador.

El último nivel en la arquitectura es el de Usuario (figura 4-4), en el cual se encuentran las interfaces con las que interactúa el usuario y las aplicaciones que pueden utilizar para crear topologías.

El grupo de aplicaciones está formado por cuatro módulos: Topologías Predefinidas, (concede al usuario métodos para crear topologías de forma rápida); Topologías Personalizadas (permite que el usuario cargue un archivo de texto con la topología que desea implementar); SDN *control plane*, brinda las facilidades para crear una red control —pre-definida o personalizada— para los *switches* OpenFlow); y Topología Alternativa (indica la cantidad de recursos que se deben reducir en caso de superar los recursos de OpenStack y del *hardware*).

Por el lado de las interfaces de usuario, se cuenta con un menú para definir las topologías e interconexiones a implementar en OpenStack y con otro para configurar los parámetros de los enlaces entre máquinas virtuales.

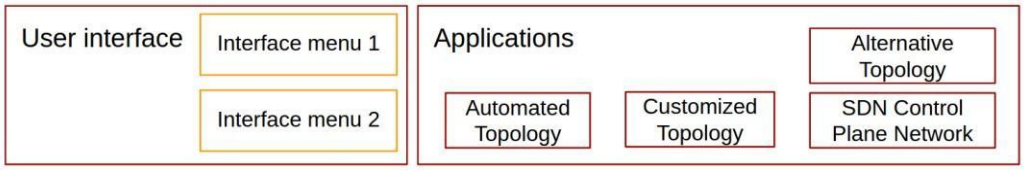


FIGURA 4-4: NIVEL DE USUARIO.
FUENTE: ELABORACIÓN PROPIA

El desarrollo de los diferentes módulos que conforman el emulador se encuentra en el Anexo 1, en el cual se detalla la lógica utilizada para cada módulo y la explicación de cada algoritmo utilizado.

4.2.2. Diagrama de bloques

En la sección anterior, se presentaron la arquitectura del emulador, los módulos implementados y una breve descripción de ellos, dando a conocer la función que cumple cada uno. En la presente sección, se describe el funcionamiento del emulador en base a los tres módulos principales: Definir Enlaces, Definir Objetos e Implementación en OpenStack.

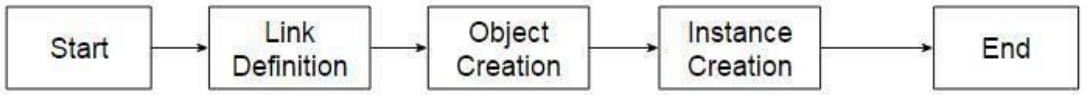


FIGURA 4-5: DIAGRAMA DE BLOQUES DEL FLUJO DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Al iniciar el programa del emulador, se crea un objeto de la clase Proyecto, el cual incluye tres parámetros: un diccionario con las topologías creadas en el proyecto, un arreglo de los enlaces entre topologías y un arreglo con los enlaces de la red de *control plane* (en caso hubiera una red SDN). A medida que se vayan creando y modificando topologías, el proyecto obtendrá los datos necesarios para definir la red de prueba final.

En el siguiente bloque, se define la secuencia de enlaces que representan las interconexiones de nodos en la red. Utilizando los módulos de crear o cargar topología, se especifica la topología: se definen los nodos que serán *switches* y *routers*, si la topología incluirá *hosts* y cómo serán las conexiones entre los nodos.

Con toda esta información, se forma el arreglo de enlaces y se envía al siguiente bloque.

Una vez que el segundo bloque del *core* recibe el arreglo de enlaces, se empiezan a crear los objetos en Python. Para ello, primero se crean los objetos de tipo Red en base a los enlaces; luego, se construyen los objetos de Instancias (*Switch*, *Router* y *Host,* imágenes disponibles en una librería); por último, se crean dos puertos por red y se asigna cada uno a las Instancias interconectadas por esa red. Estos objetos de Puerto se almacenan en un arreglo y se crea una Topología, la cual se agrega al diccionario de topologías en el Proyecto (con identificador el nombre de la topología).

En el siguiente bloque, se crean las instancias en OpenStack siempre y cuando se haya validado que se disponen de los recursos necesarios. En este bloque, se mapean los objetos Red, Puerto e Instancia por red L2, puertos en dicha red y máquinas virtuales con los puertos anteriores. Una vez creados, se pueden manipular los parámetros de las interfaces TAP (mapeados en puertos de VMs) para simular retardos y tasas de bits específicas.

En el último bloque —al finalizar el programa del emulador— se borran todos los elementos creados para representar la red de prueba: elementos de OpenStack (instancias, puertos y redes) y objetos de Python (Instancias, Puertos, Redes, Topologías y proyecto).

4.2.3. Herramientas utilizadas

En el punto anterior, se describió completamente la arquitectura de emulador, el flujo que sigue su funcionamiento y la interacción de los módulos entre sí. Para el desarrollo del emulador, se emplearon diferentes herramientas, entre las cuales se encuentran Python, NetworkX, OpenStack, Linux *traffic control*, *intermediate functional block* y *network emulation*.

Python es un lenguaje programación orientada a objetos, simple y flexible en el desarrollo del código: la curva de aprendizaje de este lenguaje es menos marcada en comparación con otros lenguajes (p.ej. Java, Perl, etc.); además, los servicios de OpenStack también están basados en Python. Es por estas razones que se usó como lenguaje base para la implementación de los módulos del emulador.

Para este lenguaje de programación, existen diferentes tipos de librearías, tanto incorporadas —que pueden ser importadas y usadas— como librerías para instalar e importar. Un ejemplo de este último tipo de librerías es NetworkX, la cual se emplea en el módulo de visualización de la siguiente manera: primero, se crea un grafo; luego, se añaden los nodos y enlaces (llamados *vertex* y *edges*, respectivamente); finalmente, se asigna un *layout* de acuerdo al tipo de topología que se vaya a mostrar para evitar que se superpongan nodos o se crucen los enlaces. Una vez que se ha creado el gráfico, se muestra en una página web, que se actualiza cada 3 segundos, para que el usuario observe en tiempo real todo lo que va elaborando con el emulador.

En cuanto a OpenStack, se instaló la versión Kilo en dos servidores Dell PowerEdge R730: un servidor sirve como nodo de Control, *Compute* y Almacenamiento, mientras que el otro sirve como *Networking*, *Compute*, y Almacenamiento. Para emular las máquinas virtuales en Nova, se utiliza el *hypervisor* KVM. Ambos servidores cuentan con una partición de 512 GB para ser utilizados por el servicio de Cinder y se comunican a través de una conexión directa usando iSCSI. Por el lado de la conexión entre servidores, se cuenta con dos redes separadas: *management* y datos. En la red de datos, Neutron emplea túneles GRE entre servidores para encapsular las tramas de las máquinas virtuales, lo que permite que exista una red L2 entre máquinas virtuales.

Para simular los enlaces entre nodos de red, se incluyen los programas *linux traffic control* y *network emulation*. Estos programas permiten reproducir los comportamientos de los enlaces de las redes, como la latencia por la distancia entre equipos y el tiempo de procesamiento de cada paquete, y una tasa de bits limitada por la tecnología de transporte de los enlaces (p.ej. SDH, microondas, satelital). Linux *traffic control* se utiliza en conjunto con el algoritmo *Token Bucket Filter* para limitar el *bit rate* de los enlaces, mientras que *traffic emulation* (netem) se emplea para generar retardos en las redes entre instancias.

Inicialmente en el diseño, estos programas se aplicaban directamente a las interfaces TAP donde las máquinas virtuales se conectaban; sin embargo, esta implementación era incorrecta. Desde el punto de vista del sistema operativo, el tráfico enviado desde una máquina virtual **entra** y el tráfico hacia una VM **sale**; por otro lado, las políticas de *shaping* se aplican al tráfico **entrante**. Si bien se observaba que el *throughput*

end-to-end era similar al esperado, se limitaba el tráfico después de recorrer el sistema, lo cual consumía recursos innecesarios.

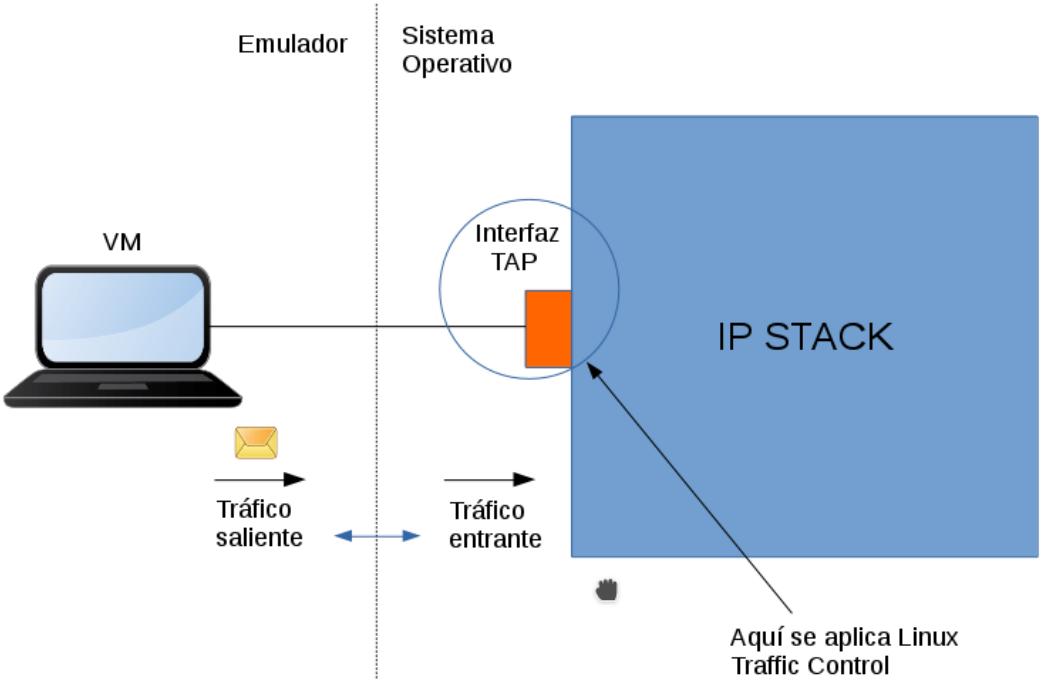


FIGURA 4-6: TRÁFICO DE VM VISTO DESDE LA PERSPECTIVA DEL OS.
FUENTE: ELABORACIÓN PROPIA

Por esta razón, se incluyeron las pseudo-interfaces *intermediate functional block* (IFB) en el diseño del módulo *link simulation*. Estas se ubican entre las interfaces TAP y el IP *stack* de Linux, generando que el tráfico entrante a la interfaz TAP salga por la interfaz IFB y entre al sistema operativo; con esta configuración, se puede limitar el *bit rate* de un enlace antes que se envíe el tráfico por todo el sistema.

4.3. Modificaciones a OpenStack

OpenStack es un sistema operativo de *cloud* que permite ofrecer servicios de IaaS. Al ser Kilo su versión número once (11), incluye diversas funcionalidades para proteger el sistema ante fallas en la operación y ataques desde el exterior (p.ej. L3 *high availability* y *antispoofing*). Si bien estas herramientas son de utilidad para entornos que presentan servidores virtualizados, no lo son para entornos de NFV. Estos últimos requieren múltiples conexiones L2 entre máquinas virtuales para implementar topologías de redes comerciales. Debido a que en esta versión de

OpenStack no estaban desarrolladas las funcionalidades de NFV, se realizaron algunas modificaciones realizadas al sistema, las cuales se detallan en las siguientes secciones.

4.3.1. *Computing*

En OpenStack Kilo, el servicio encargado de desplegar e iniciar las máquinas virtuales se llama Nova. Dicho servicio utiliza a Glance y Cinder para obtener la imagen de disco y los volúmenes de almacenamiento de las instancias (nombre que emplea OpenStack para las máquinas virtuales). Adicionalmente, requiere de Neutron para brindar conectividad a las máquinas virtuales; en otras palabras, necesariamente se deben crear las imágenes de disco, volúmenes y redes antes de iniciar una instancia. En el caso de las interconexiones entre VMs, se debe crear una red (conexiones L2) con una subred (L3) donde se indiquen las direcciones que se emplearán en dicha red.

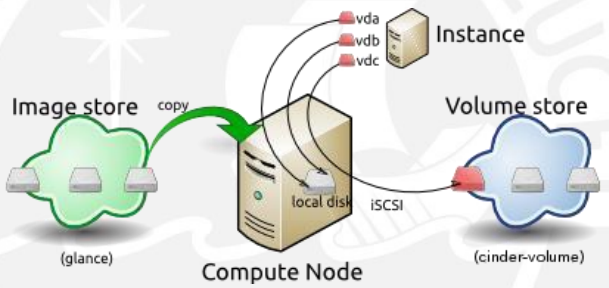


FIGURA 4-7: CREACIÓN DE INSTANCIA EN OPENSTACK.
FUENTE: [47]

La funcionalidad de brindar redes L3 a las instancias permite a OpenStack evitar el la configuración de direcciones IP en las VMs (ideal para una granja de servidores virtualizados); sin embargo, el emulador requiere únicamente de una red de capa 2 entre las máquinas virtuales. Esta red debe constar de dos puertos, pues representan enlaces entre dos nodos.

En la tabla 4-1 se muestran los cuatro métodos planteados para la creación de las máquinas virtuales sin una subred (IP) asociada; de estos cuatro, los métodos 3 y 4 resultaron ser los más adecuados, tomando en consideración el consumo de recursos y el tiempo empleado para crear cada instancia. Más aún, con el método 4 se puede tener control de los puertos creados; empero, la limitante de Nova se presenta al momento de desplegar instancias unidas por redes sin subredes,

mostrando una excepción que no permite iniciar las máquinas virtuales.
Debido a esta limitante, fue necesario modificar dos archivos en el código fuente de OpenStack: excepciones de Nova y API de Neutron. En el primero (exception.py), se comentaron las líneas de código donde definen que un puerto requiere una dirección IP para ser usada en una instancia; en el segundo (api.py), se comentaron las líneas que invocan a la excepción del primer archivo. Con estas modificaciones, fue posible crear instancias con puertos de redes que no presentan subredes. Las líneas comentadas se observan en las figuras 4-7 y 4-8.

# de método	Secuencia	Resultados
1	1. Se crea máquina virtual (VM) sin red 2. Se crea red con subred 3. Se apaga VM, se asocia red a VM y se prende VM	Inicia la instancia, pero no es óptimo, pues prender y apagar VM consume recursos y tiempo de ejecución.
2	1. Se crea máquina virtual (VM) sin red 2. Se crea red con subred 3. Se crea puerto de la red 4. Se apaga VM, se asocia puerto a VM y se prende VM	Inicia la instancia, pero no es óptimo, pues prender y apagar VM consume recursos y tiempo de ejecución.
3	1. Se crea red sin subred 2. Se crea VM asociando la red	No se inicia instancia. Excepción de red sin subred.
4	1. Se crea red sin subred 2. Se crea puerto de la red 3. Se crea VM asociando al puerto	No se inicia instancia. Excepción de puerto sin dirección IP.

TABLA 4-1: PRUEBAS REALIZADAS PARA CREAR INSTANCIAS DE FORMA ÓPTIMA.
FUENTE: ELABORACIÓN PROPIA

4.3.2. *Networking*

El servicio de Neutron es el encargado de crear las redes, subredes y puertos, brindando conectividad a las máquinas virtuales (entre ellas o hacia el exterior). En el primer caso, emplea enlaces veth entre ovs y linux bridge, interfaces TAP conectadas a procesos (en este caso VMs), linux *bridge* junto con IPTABLES para filtrar paquetes y realizar NAT; y Open vSwitch para diferenciar las redes de las máquinas virtuales utilizando VLANs; en el segundo caso, se incorporan dos

procesos a través de *linux namespaces*: DHCP (dnsmasq), para brindar direcciones IP de forma dinámica; y enrutamiento (IP *forwarding*), que les permite tener conectividad L3. Como se observa en la figura 4-10, si dos máquinas virtuales (*overlay network*) se desean comunicar, deben pasar por un *linux bridge* y dos Open vSwitch (*underlay network*); si se encuentran en diferentes servidores, se emplea un túnel GRE o VXLAN.

La limitante de esta red L2 está dada por las tramas con direcciones *multicast*, debido a que utilizan direcciones MAC reservadas y estas no pueden ser transmitidas. Este comportamiento condiciona el funcionamiento del emulador, pues bloquea los mensajes LLDP (*Link Layer Discovery Protocol*) empleados en redes SDN para que el controlador descubra los enlaces entre *switches*. Durante las pruebas de funcionamiento se observó que tanto el *linux bridge* (recuadros morados en figura 4-10) como los Open vSwitch (recuadros rojos en figura 4-10) —partes de la red de OpenStack— descartaban los paquetes LLDP.

```
748 class DatastoreNotFound(NotFound):
749     msg_fmt = _("Could not find the datastore reference(s) which the VM uses.")
750
751
752 class PortInUse(Invalid):
753     msg_fmt = _("Port %(port_id)s is still in use.")
754
755
756 #class PortRequiresFixedIP(Invalid):
757 #    msg_fmt = _("Port %(port_id)s requires a FixedIP in order to be used.")
758
759
760 class PortNotUsable(Invalid):
761     msg_fmt = _("Port %(port_id)s not usable for instance %(instance)s.")
762
763
764 class PortNotFree(Invalid):
765     msg_fmt = _("No free port available for instance %(instance)s.")
766
```

FIGURA 4-8: MODIFICACIÓN DEL CÓDIGO DEL ARCHIVO EXCEPTION.PY. FUENTE: ELABORACIÓN PROPIA

```
                port = neutron.show_port(request.port_id).get('port')
            except neutron_client_exc.NeutronClientException as e:
                if e.status_code == 404:
                    port = None
                else:
                    with excutils.save_and_reraise_exception():
                        LOG.exception(_LE("Failed to access port %s"),
                                      request.port_id)
            if not port:
                raise exception.PortNotFound(port_id=request.port_id)
            if port.get('device_id', None):
                raise exception.PortInUse(port_id=request.port_id)
            #if not port.get('fixed_ips'):
            #    raise exception.PortRequiresFixedIP(
            #        port_id=request.port_id)
            request.network_id = port['network_id']
        else:
            ports_needed_per_instance += 1
            net_ids_requested.append(request.network_id)
```

FIGURA 4-9: MODIFICACIÓN DEL CÓDIGO DEL ARCHIVO API.PY.
FUENTE: ELABORACIÓN PROPIA

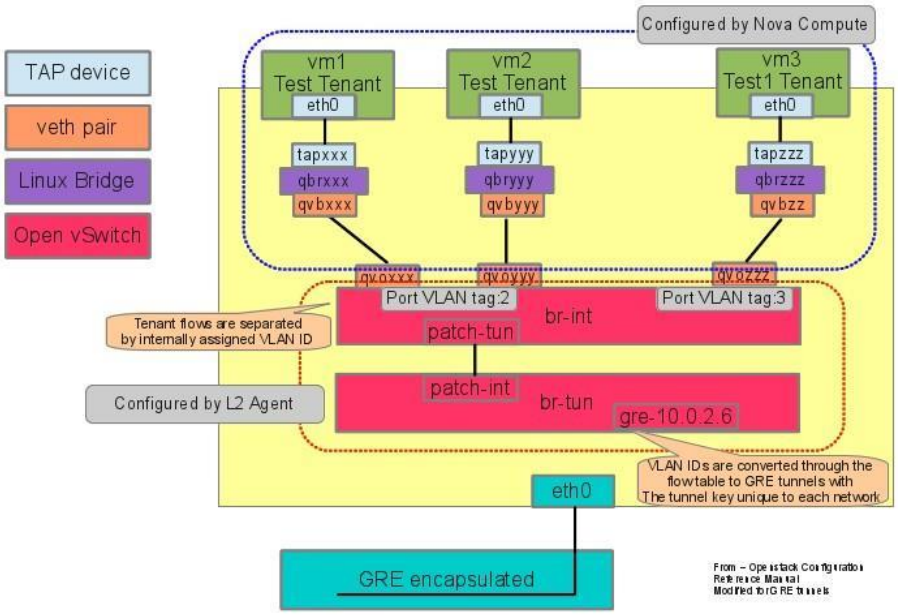


FIGURA 4-10: DIAGRAMA DE REDES DE OPENSTACK.
FUENTE: [21]

Tal como se explica en [48], los *linux bridge* descartan los mensajes LLDP (*Link Layer Discovery Protocol*) debido al diseño que presentan. Los mensajes LLDP emplean la dirección MAC *multicast* destino 01-80-C2-00-00-0E, la cual se encuentra dentro del rango de direcciones L2 que bloquean los *linux bridge* (IEEE Std 802.1D-2004). Por otro lado, los Open vSwitch presentan un comportamiento similar al no permitir el paso de dichas direcciones, aunque es posible cambiar este comportamiento sin

modificar su código fuente. Como posibles soluciones se puede introducir *flows* o habilitar el *feature forward-BPDU* para permitir el paso estos mensajes.

Por estas razones, fue necesario realizar dos cambios en la configuración de Neutron. Primero se deshabilitó la creación de *linux bridges* para cada instancia. Esto se logró cambiando la configuración del *firewall driver* en la sección [securitygroup] del archivo ml2_conf.ini.

```
88 [securitygroup]
89 # Controls if neutron security group is enabled or not.
90 # It should be false when you use nova security group.
91 enable_security_group = False
92
93 # Use ipset to speed-up the iptables security groups. Enabling ipset support
94 # requires that ipset is installed on L2 agent node.
95 enable_ipset = True
96
97 firewall_driver = neutron.agent.firewall.NoopFirewallDriver
98 #firewall_driver = neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver
99
```

FIGURA 4-11: CONFIGURACIÓN DEL ARCHIVO ML2_CONF.INI.
FUENTE: ELABORACIÓN PROPIA

Una vez que se quitaron los *linux bridge* de la red L2, se dejó de utilizar los enlaces tipo *veth* y se conectaron las interfaces TAP directamente a los Open vSwitch. Luego, se permitió el paso de tramas con direcciones reservadas por el Open Vswitch. De las dos soluciones existentes, se optó por habilitar el *feature* de bpdu_forward en vez de instalar *flows* manualmente para cada red (VLAN) distinta. Esto se realizó usando el comando "ovs-vsctl set bridge [nombre_bridge] other-config:forwardbpdu=true" en los *br-int* y *br-tun* de todos los servidores de Nova.

4.4. Validador

El validador del emulador está conformado por cuatro módulos, cuya función principal es prevenir errores en el funcionamiento del programa. Estos cuatro módulos validan datos de entrada, archivo de la topología personalizada, recursos de cada máquina virtual y del bus del procesador.

Tanto la validación de los datos como del archivo que se carga con la topología personalizada se especifica en la tabla 4-2.

Módulos	Restricciones
Menús 1 y 2	• La opción elegida debe ser un número dentro del rango presentado. • Borrar topología funcionará si existe al menos una creada. Además, valida que el nombre sea de una topología existente. • Unir topología funcionará si existen dos o más. • Implementar topología se aplicará si existe por lo menos una. Asimismo, valida que la respuesta de crear una red SDN sea correcta.
Definir enlaces	• El número de nodos para una topología estrella debe ser mayor a 3; el de clique, mayor a 2. • El nivel, cantidad de filas y columnas deben ser un número. • Se escoge el tipo de nodo (*routers:* "r" o "R"; *switches:* "S" o "s") y cuáles serán de cada tipo (p.ej. S, 0, 1,2 o r, 2, 8,5). • Para elegir todos los nodos se indica "a" o "A" (p.ej. S,a o R,A) El número de *hosts* por nodo debe ser un número entero.
Unir topologías	• El nombre de los nodos debe tener el formato correcto (p.ej. R0-t0, S1t5) • Los nodos deben estar creados. • El nombre de la topología debe ser de una existente. • Los nodos deben pertenecer a las topologías indicadas. • Los nombres de los nodos deben ser diferentes.
Cargar archivo	• El archivo debe existir y no puede estar vacío. • Los nodos se definen especificando el tipo de elemento (*router*: "r" o "R"; *switch*: "S" o "s"; y *host*: "h" o "H") y un identificador único (p.ej. S0, R2, H3). • Un enlace se define como el nombre de dos nodos separados por una coma (p.ej. S0, R2). • Un enlace no debe tener más de dos nodos. • No deben existir espacios en blanco en la definición de enlaces. • No deben presentarse líneas sin enlaces.

TABLA 4-2: RESTRICCIONES DE MÓDULOS.
FUENTE: ELABORACIÓN PROPIA

Por el lado de la validación de recursos y bus de procesamiento, ambas dependen de dos calibraciones previas: con respecto al *hardware* donde está funcionando y a un factor llamado Tamaño por Paquete (TPP).

La calibración respecto al *hardware* consiste en establecer los valores máximos (cuotas) de RAM, vCPUs y cantidad de instancias totales que puede tener el proyecto del emulador en OpenStack. Para ello, primero, se debe hallar la cantidad de

recursos (RAM, vCPUs y capacidad del enlace entre servidores) disponibles por servidor. Luego, se suman dichas cantidades para obtener los valores necesarios para las cuotas de OpenStack. En cuanto a la cantidad máxima de instancias, se consideró que cada una (*host, switch* y *router*) tiene un *flavor* (conjunto de recursos) mínimo (m1.small).

Por otro lado, para la calibración respecto a los paquetes, cada usuario tiene la posibilidad de especificar cuál es el tamaño promedio de los paquetes con el que trabajará su red de prueba. Este parámetro —llamado Tamaño por Paquete o TPP— indica la cantidad de *bytes* estimada que tendrán los paquetes que viajen por las instancias. Un valor pequeño de este parámetro implica que una máquina virtual va a tener que procesar más paquetes por segundo a una determinada tasa de bits, por lo que requerirá mayor número de *cores.*

En base a ambas calibraciones, se construyen las condiciones que se utilizaran en el validador, de tal forma que no exceda las capacidades de los servidores. Estas son:

1. La capacidad de todos los enlaces virtuales existentes sobre un enlace real (p.ej. enlace entre un puerto de un servidor y un puerto del *swicth* del *rack*) no debe exceder la capacidad de dicho enlace real.

$$\sum Enlaces\ entre\ VMs\ en\ distintos\ servidores < Capacidad\ INTX\ servidores \quad (1)$$

2. La capacidad de todos los enlaces de las máquinas virtuales en un solo servidor no debe exceder la capacidad del bus de procesamiento del servidor.

$$\sum Enlaces\ entre\ VMs\ en\ un\ servidor < Bus\ de\ \ procesamiento\ del\ servidor \quad (2)$$

3. La cantidad de *cores* de todas las máquinas virtuales en un solo servidor debe ser menor a la cantidad de *cores* disponibles en el mismo.

$$\sum Cores\ de\ VMs\ en\ un\ servidor < Cantidad\ de\ cores\ en\ un\ servidor \quad (3)$$

4. La cantidad de RAM de todas las máquinas virtuales en un solo servidor debe ser menor a la cantidad de RAM disponibles en el mismo.

$$\sum RAM\ de\ VMs\ en\ un\ servidor < Cantidad\ de\ RAM\ en\ un\ servidor \quad (4)$$

Para hallar las constantes de las inecuaciones, se utilizaron los valores de los recursos disponibles en cada servidor. El emulador se encuentra instalado en servidores con un procesador Intel® Xeon® Processor E5-2640 v3 [26], el cual posee 32 *cores*, 64 GB de RAM y un ancho de banda máximo de memoria igual a 59 GB/s. Adicionalmente, el enlace entre los dos servidores tiene una capacidad de 1Gb/s, lo cual genera un cuello de botella en la conexión entre ambos servidores.

Cabe mencionar que, para la protección ante errores de funcionamiento del programa, específicamente en las instancias, KVM utiliza instrucciones específicas del procesador para asegurar el aislamiento tanto entre el *hypervisor* y las máquinas virtuales como entre ellas [27]. En [27] también se menciona que los procesadores Intel y AMD incluyen *features* (p.ej. VMX y SVM) que añaden un nivel adicional de protección al ejecutar las máquinas virtuales en un modo restringido. Debido a que estas instrucciones son nativas de KVM, no fue necesario implementarlas dentro del módulo validador, sino que se utilizan como un complemento.

Ancho de Banda enlace real entre servidores (Mbps)	Bus de procesamiento del servidor (Gbps)	Cantidad de *cores* en servidor Control	Cantidad de *cores* en servidor Nova	Cantidad de RAM en servidor Control (GB)	Cantidad de RAM en servidor Nova (GB)
950	40	27	29	40	58

TABLA 4-3: CONSTANTES USADAS PARA LAS CONDICIONES DEL VALIDADOR.
FUENTE: ELABORACIÓN PROPIA

4.5. Interfaz de usuario y creación de topologías

El nivel de Usuario está divido en dos módulos: Interfaz de usuario y aplicaciones. El primero, permite interactuar con el emulador mediante un menú; el segundo, presenta las diversas funcionalidades que el emulador proporciona al usuario para la creación rápida de topologías.

4.5.1. Interfaz de usuario

En el nivel de usuario, el emulador cuenta con dos menús. El primero se muestra en la figura 4-12. Este proporciona, al usuario, diversas opciones para facilitar la implementación de la red de prueba deseada.

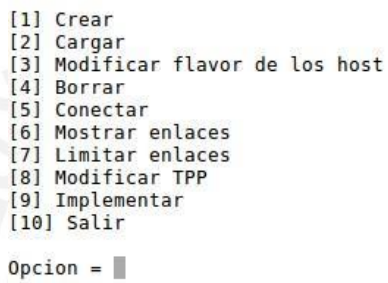


FIGURA 4-12: PRIMER MENÚ DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Como se observa en la imagen anterior, el menú presenta opciones como crear topología, cargar topología, modificar el *flavor* de los *hosts*, borrar topología, conectar topologías, mostrar los enlaces de todo el proyecto, limitar enlaces en ancho de banda y *delay*, modificar el valor del tamaño por paquete (TPP), implementar las topologías en el proyecto y salir del programa. Estas opciones se detallan a continuación:

- **Crear topología:** Proporciona seis topologías predefinidas para que el usuario implemente su red de forma rápida y sencilla. Entre las topologías ofrecidas se encuentran: Estrella, Árbol, Lineal, Anillo, Malla y *Full Mesh* (todos contra todos). Además, permite determinar qué elemento será cada nodo (*Switch* o *Router)*. Asimismo, incluye la opción de añadir *hosts*, ya sea a todos o algunos nodos específicos.

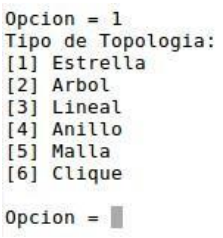


FIGURA 4-13: OPCIÓN PARA CREAR TOPOLOGÍA.
FUENTE: ELABORACIÓN PROPIA

- **Cargar Topología:** Permite cargar un archivo de texto que contiene los enlaces de la topología a implementar. Estos enlaces deben mostrar los nodos de la topología (*host* – H, *switch* –S, *router* –R y un número que identifique a cada nodo) separados por comas. Para ello, se debe especificar la ruta donde se encuentra almacenado dicho archivo.

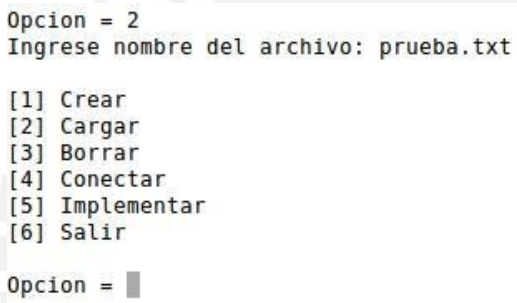


FIGURA 4-14: OPCIÓN PARA CARGAR TOPOLOGÍA.
FUENTE: ELABORACIÓN PROPIA

- **Modificar *flavor* de los *hosts*:** Cambia el *flavor* (conjunto de recursos) de los *hosts* creados a nivel de Python; por defecto, todos los *hosts* tienen asignado un *flavor* con recursos mínimos (m1.small). Esta opción se habilita cuando por lo menos existe una topología creada.

```
Opcion = 3
+------------------------+------------------------+--------+
| Nombre de la topologia | Nombre de la Isntancia | Flavor |
+------------------------+------------------------+--------+
|           t0           |         H1-t0          |   1    |
|           t0           |         S1-t0          |   1    |
|           t0           |         R1-t0          |   1    |
|           t0           |         S2-t0          |   1    |
|           t0           |         H2-t0          |   1    |
+------------------------+------------------------+--------+
Nombre del nodo : H1-t0
+---------------+--------------------+----------+-------+
| ID del flavor | Nombre del flavor | RAM (MB) | VCPUs |
+---------------+--------------------+----------+-------+
|       0       |      m1.tiny       |   512    |   1   |
|       1       |      m1.small      |   2048   |   1   |
|       2       |     m1.medium      |   4096   |   2   |
|       3       |      m1.large      |   8192   |   4   |
|       4       |     m1.xlarge      |  16384   |   8   |
+---------------+--------------------+----------+-------+
Ingrese ID del flavor: 2
Se actualizo el flavor
```

FIGURA 4-15: OPCIÓN PARA MODIFICAR EL *FLAVOR* DE LOS *HOSTS*.
FUENTE: ELABORACIÓN PROPIA

- **Borrar topología:** Su función es eliminar una topología creada dentro del proyecto a nivel de Python; para ello, se debe especificar el nombre de la topología mostrada en la gráfica de redes. Esta opción está disponible siempre y cuando exista como mínimo una topología.

```
Opcion = 4

Nombre de topologia a borrar: t0
puerto32-t0 borrado
puerto31-t0 borrado
red16-t0 borrado
puerto30-t0 borrado
puerto29-t0 borrado
red15-t0 borrado
puerto28-t0 borrado
puerto27-t0 borrado
```

FIGURA 4-16: OPCIÓN PARA BORRAR TOPOLOGÍA.
FUENTE: ELABORACIÓN PROPIA

- **Conectar topologías:** Permite unir dos topologías del proyecto con un enlace punto a punto; para ello, debe indicar un nodo de cada topología. Esta opción estará disponible cuando estén creadas, por lo menos, dos topologías.

```
Opcion = 5
Nodo 1 a conectar: R1-t1
Nodo 2 a conectar: R1-t2
```

FIGURA 4-17: OPCIÓN PARA UNIR TOPOLOGÍAS.
FUENTE: ELABORACIÓN PROPIA

- **Mostrar enlaces:** Imprime una tabla mostrando todos los enlaces creados, la topología a la que pertenecen, cuáles son los nodos que unen y, principalmente, el ancho de banda y *delay* de cada enlace.

```
Opcion = 6
+-----------------------+-------------------+-----------------+-----------------------+-------+-------------------------+
| Nombre de la topologia | Nombre del Puerto | Nombre de la Red | Ancho de Banda (Mbps) | Delay | Nombre de la Isntancia |
+-----------------------+-------------------+-----------------+-----------------------+-------+-------------------------+
|          t0           |     puerto1-t0    |     red01-t0     |          100M         |   0   |         H1-t0          |
|          t0           |     puerto2-t0    |     red01-t0     |          100M         |   0   |         S1-t0          |
|          t0           |     puerto3-t0    |     red02-t0     |          100M         |   0   |         R1-t0          |
|          t0           |     puerto4-t0    |     red02-t0     |          100M         |   0   |         S1-t0          |
|          t0           |     puerto5-t0    |     red03-t0     |          100M         |   0   |         S2-t0          |
|          t0           |     puerto6-t0    |     red03-t0     |          100M         |   0   |         R1-t0          |
|          t0           |     puerto7-t0    |     red04-t0     |          100M         |   0   |         S2-t0          |
|          t0           |     puerto8-t0    |     red04-t0     |          100M         |   0   |         H2-t0          |
|          t0           |     puerto9-t0    |     red05-t0     |          100M         |   0   |         S1-t0          |
|          t0           |    puerto10-t0    |     red05-t0     |          100M         |   0   |         S2-t0          |
+-----------------------+-------------------+-----------------+-----------------------+-------+-------------------------+
```

FIGURA 4-18: OPCIÓN PARA MOSTRAR ENLACES.
FUENTE: ELABORACIÓN PROPIA

- **Modificar enlaces:** Permite cambiar el valor del ancho de banda (Mbps) y el *delay* de un enlace determinado. Esta opción está disponible si, por lo menos, hay una topología creada.

```
Opcion = 7
Ingrese nombre de red: red01-t0
Ingrese nuevo BW (Mbps): 200
Ingrese delay para los enlaces: 50
Se actualizo el valor de BW y delay
```

FIGURA 4-19: OPCIÓN PARA MODIFICAR EL BW Y DELAY DE LOS ENLACES.
FUENTE: ELABORACIÓN PROPIA

- **Modificar TPP:** Esta opción permite modificar el valor del tamaño por paquete (TPP) con el cual va a trabajar el emulador. Por defecto, el valor asignado es 400 B. Este valor determina la cantidad de *cores* que requerirá una máquina virtual.

```
Opcion = 8
Actualmente el valor de TPP es 400 Bytes. Ingrese nuevo valor: 1500
Se actualizo el valor de TPP: 1500
```

FIGURA 4-20: OPCIÓN PARA MODIFICAR EL TPP DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

- **Implementar:** Confirmación para crear la topología del usuario a nivel de OpenStack. En este punto se mapean los objetos de clases Instancias, Puertos y Redes en máquinas virtuales, puertos y redes de OpenStack, respectivamente. En el caso que exista una topología del proyecto que contenga únicamente *hosts* y *switches*, se habilitará la opción de crear redes SDN.

```
Opcion =  9

Posibles topologias SDN

[t0]

Topologias SDN [Y/N]: Y
Se va a implementar redes SDN
Topologias implementadas en Openstack
red01-t0
red02-t0
red03-t0
```

FIGURA 4-21: OPCIÓN PARA IMPLEMENTAR TOPOLOGÍAS EN OPENSTACK.
FUENTE: ELABORACIÓN PROPIA

- **Salir**: Termina el flujo del emulador. Al salir, se borran todos los objetos creados en Python (instancias, puertos, redes, topologías y el proyecto).

```
Opcion =  10
proyecto borrado
puerto32-t0 borrado
puerto31-t0 borrado
red16-t0 borrado
```

FIGURA 4-22: OPCIÓN PARA SALIR DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Una vez implementada la red de prueba en OpenStack, se muestra el segundo menú. En este menú ya no es posible crear topologías, sino se brinda la posibilidad de interactuar con las VMs creadas.

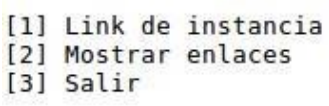


FIGURA 4-23: SEGUNDO MENÚ DEL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

En la figura 4-23 se aprecian las opciones del segundo menú. Las tres opciones disponibles son: mostrar los enlaces y características entre instancias, mostrar la URL de las instancias y salir. Estos tres de detallarán a continuación:

- **Link de instancias:** Muestra la URL de la instancia indicada. Las interfaces de las máquinas virtuales se visualizan a través del protocolo VNC. OpenStack incluye al cliente noVNC para acceder a dicha consola a través de HTML.

- **Mostrar enlaces:** Permite visualizar los enlaces entre máquinas virtuales y sus características. La visualización de realiza a través de tablas con los siguientes valores: nombre de la red, instancias de la red, *bit rate* y *delay*.

- **Salir**: Termina el flujo del emulador. Además de borrar todos los objetos creados en Python, se eliminan las máquinas virtuales a nivel de OpenStack.

4.5.2. Topologías pre-definidas

Analizando la infraestructura de los proveedores de servicios, se puede apreciar que su red está conformada por un conjunto de topologías interconectadas entre sí: *full mesh* (todos contra todos) en el *core*, anillos a nivel metropolitano, árbol en los extremos de la red, etc. Es por ello que se optó por implementar un *feature* de creación de topologías predefinidas en el emulador. Entre dichas topologías se encuentran: Estrella, Árbol, Lineal o Bus, Anillo, Malla y *Full Mesh*.

Como se explicó anteriormente, una topología se modela como un arreglo de pares de objetos de clase Puerto, y cada elemento del arreglo hace referencia a un enlace entre nodos. Es por esta razón que se plantearon algoritmos para indicar la manera de interconectar los nodos. Como dato de entrada, se ingresa un número "n" cuyo significado varía de acuerdo a la topología.

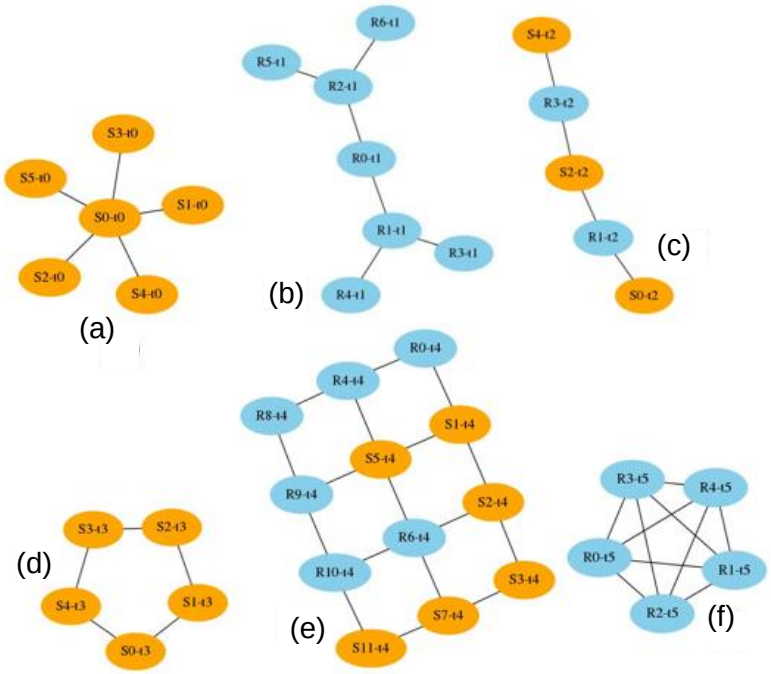


FIGURA 4-24: EJEMPLOS DE TOPOLOGÍAS PREDEFINIDAS: (a) ESTRELLA, (b) ÁRBOL, (c) LINEAL, (d) ANILLO, (e) MALLA, (f) *FULL MESH*.
FUENTE: ELABORACIÓN PROPIA

Para el caso de una topología Estrella, el número ingresado hace referencia a la cantidad de nodos alrededor del nodo central. El algoritmo para formar los enlaces consiste en mantener al nodo central como abscisa e ir agregando los demás nodos como ordenada, formando pares ordenados dentro del arreglo.

$$Seq = [(a_0, a_1), (a_0, a_2), (a_0, a_3), (a_0, a_4), \dots, (a_0, a_n)] \quad (5)$$

En el caso de una topología Lineal, el número solicitado hace referencia a la cantidad de nodos que van uno a continuación del otro. El algoritmo se basa en mantener el primer nodo como abscisa del primer par y el último como ordenada del par final. Los nodos intermedios van alternándose entre abscisa u ordenada en cada par ordenado.

$$Seq = [(a_0, a_1), (a_1, a_2), (a_2, a_3), \dots, (a_{n-2}, a_{n-1}), (a_{n-1}, a_n)] \quad (6)$$

Para una topología anillo se forman los enlaces de manera similar a la topología lineal, con la diferencia de que se agrega un enlace adicional. Este enlace corresponde a la unión entre el primer y último nodo.

$$Seq = [(a_0, a_1), (a_1, a_2), (a_2, a_3), \dots, (a_{n-2}, a_{n-1}), (a_{n-1}, a_n)(a_n, a_0)] \quad (7)$$

Con respecto a una topología Árbol, usando el valor del nivel (N) se obtiene el número de nodos como:$\#_{NODOS} = \sum_{k=1}^{N} 2^{k-1}$, donde $\#_{NODOS}$ = número de nodos. Para determinar los enlaces, se debe recorrer cada nivel; a excepción del nivel = N, cada nodo "x" tiene dos enlaces: hacia la izquierda $2 * (x + 1)$ y hacia la derecha $2 * (x + 2)$.

$$Seq = [(a_0, a_1), (a_0, a_2), \dots, (a_x, a_{2*(x+1)}), (a_x, a_{2*(x+2)}), \dots, (a_{(n-2)/2}, a_{n-1}), (a_{(n-2)/2}, a_n)] \quad (8)$$

donde $n = \#_{NODOS} - 2^{(N-1)}$

En el caso de una topología Malla, el número de filas (N_F) y columnas (N_C) determinan el número de nodos ($\#_{NODOS} = N_F x N_C$). Los enlaces se hallan recorriendo cada fila. Asumiendo que se tienen N_F filas (de 0 a $N_F - 1$) y N_C columnas (de 0 a $N_C - 1$).

$$Seq = \begin{cases} A_{(i,j)}, A_{(i,j+1)} \begin{cases} 0 \le i \le N_C - 1 \\ 0 \le j < N_C - 1 \end{cases} \\ A_{(i,j)}, A_{(i+1,j)} \begin{cases} 0 \le i < N_F - 1 \\ 0 \le i \le N_F - 1 \end{cases} \end{cases}$$

Para el último caso, *Full Mesh*, el valor ingresado (N) indica la cantidad de nodos. Asumiendo que $0 \le k$, los enlaces se generan uniendo un elemento a_k con otro elemento a_{k+1} tal que $k \le (N - 1)$.

$$Seq = [(a_0, a_1), (a_0, a_2), \dots, (a_0, a_N), (a_1, a_2), (a_1, a_3), \dots, (a_1, a_N), \dots, (a_{N-2}, a_N), (a_{N-1}, a_N)] \quad (9)$$

4.5.3. Topologías personalizadas

Si bien el usuario puede implementar su red de prueba con topologías prediseñadas, puede darse el caso donde el usuario desee implementar y probar su propio diseño de red. Es por ello que se incorporó una opción para la creación de topologías personalizadas. Esta opción le proporciona flexibilidad al usuario para crear sus propios diseños de red, indicando los enlaces entre los diferentes dispositivos bajo cierto formato (ejemplos en el Anexo 2) y guardándolo en un archivo de texto. Para cargar este archivo en el emulador, el usuario debe especificar la ruta donde se encuentra almacenado.

Una vez que el proyecto cuente con más de una topología creada (predefinida o personalizada), se puede utilizar la opción de "Conectar Topología". Como se mencionó en el punto anterior, esta opción permite crear enlaces punto a punto entre dos topologías diferentes indicando los nodos a conectar.

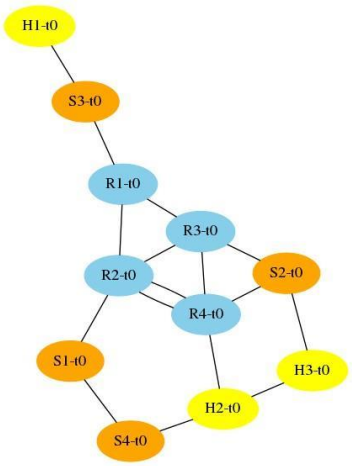


FIGURA 4-25: GRÁFICA DE LA TOPOLOGÍA PERSONALIZADA.
FUENTE: ELABORACIÓN PROPIA

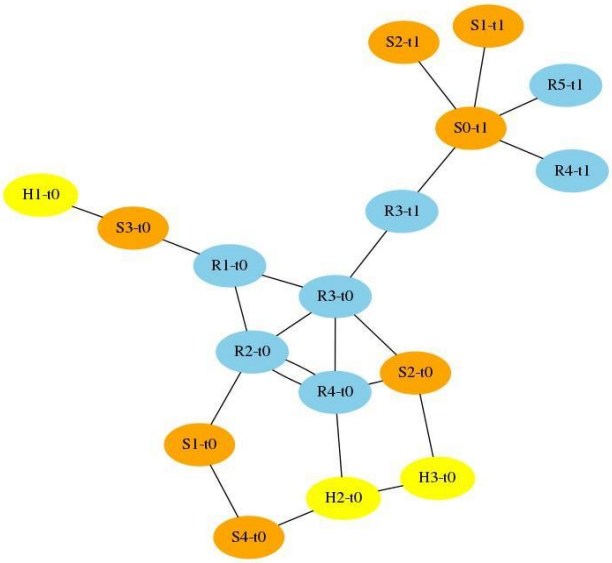


FIGURA 4-26: GRAFICA DE DOS TOPOLOGÍAS UNIDAS A TRAVÉS DE R3-T0 Y R3-T1.
FUENTE: ELABORACIÓN PROPIA

Una vez que se ha explicado el diseño completo del emulador, es preciso señalar las consideraciones que se tuvieron presentes. Estas son:

- OpenStack es una plataforma *Multitenant*, el cual distribuye los recursos de los servidores entre cada *tenant* creado. Para el caso del emulador, se consideró que el *tenant* utilizado es el un único creado y está designado exclusivamente para el emulador. Por ello, dispone de la totalidad de recursos de los servidores.

- OpenStack crea de manera aleatoria las máquinas virtuales (VMs) en los servidores. Ya que no se puede predecir en qué servidor estará cada VM, se implementó un módulo que realiza la asignación de VMs en los servidores de forma alternada. Sin embargo, debido a que el emulador está diseñado para funcionar sobre un entorno de mayor capacidad (p.ej. ExoGENI), se requiere de un módulo que determine en qué servidor debería implementarse cada VM. La asignación debe darse en base a la cantidad de recursos que requiere cada máquina virtual y los disponibles en cada servidor.

- Es fundamental cuantificar la cantidad de *cores* y RAM que consumen los procesos rutinarios en los servidores para no considerarlos en el número de recursos disponibles para las VM. Para determinar estos valores, se procedió a medir los recursos que consumen dichas rutinas por un periodo de 30 segundos. Con las mediciones se obtuvo que, en promedio, el servidor Nova utiliza 0,75% del total de *cores*, mientras que Control emplea el 7,57% de ellos.
  El número de *cores* equivalentes a estos porcentajes se halla de la siguiente manera.

$$total\ de\ CPUs\ Nova\ =\ 32x0{,}75\% \sim 1\ core$$
$$total\ de\ CPUs\ Control\ =\ 32x7{,}57\% \sim 3\ cores$$

  Asumiendo una guarda de 2 *cores*, la cantidad de estos disponibles en Nova es 29 y en Control 27. Asimismo, la cantidad de RAM que se consume en Nova es 3 GB y en Control 19 GB. Se establece una guarda de 3 GB de RAM para nova y 5 GB para Control; de esta manera, se cuenta con 58 GB en Nova y 40 GB en Control.

- Para obtener el ancho de banda interno disponible en cada servidor, se tomó como referencia el valor teórico (59 Gbps) indicado en su *datasheet*. Como no se encontró un método para calcular el ancho de banda real, se realizaron

pruebas de iperf con tráfico UDP para estimarlo. Se utilizaron dos topologías tipo estrella: una con 6 y otra con 8 *hosts* alrededor de un *switch* central cargado con la imagen de un Open vSwitch, todos en un solo servidor. La mitad de *hosts* hacían de clientes y el resto como servidores de iperf. Adicionalmente, se mantuvo habilitado el *feature* TCO para evitar la fragmentación de los paquetes y así obtener el mayor ancho de banda en las interfaces. Los resultados de cada prueba se muestran en las tablas 4-4 (a) y 4-4 (b), respectivamente. A valores mayores de "BW pedido por cliente", se empiezan a observar pérdida significativa de paquetes. Por tanto, los resultados representan el máximo ancho de banda interno del servidor. Como se observa, la máxima cantidad de ancho de banda en un servidor fluctúa entre 39,62 y 41,8 Gbps. Por tal motivo, se obtuvo el promedio de estas dos cantidades determinando que el ancho de banda disponible es 40 Gbps. Esta cantidad estará disponible para el uso del emulador.

Bw pedido (Gbps)	**14**			
Cliente – Servidor	**Cliente 1 – Servidor 1**	**Cliente 2 – Servidor 2**	**Cliente 3 – Servidor 3**	**TOTAL**
***Throughput* (Gbps)**	13,9	13,9	14	41,8

(a)

Bw pedido (Gbps)	**10**				
Cliente – Servidor	**Cliente 1 – Servidor 1**	**Cliente 2 – Servidor 2**	**Cliente 3 – Servidor 3**	**Cliente 4 – Servidor 4**	**TOTAL**
***Throughput* (Gbps)**	9,86	9,79	9,98	9,99	39,62

(b)

TABLA 4-4: MEDICION DEL *THROUGHPUT* PARA DETERMINAR EL PROCESAMIENTO INTERNO.
FUENTE: ELABORACIÓN PROPIA

5. Pruebas y análisis

Una vez que se ha diseñado e implementado el emulador, la última parte del trabajo consiste en probar el funcionamiento de los *features* y validadores implementados. Las pruebas del emulador constan de cinco etapas: funcionamiento o concepto, calibración, fidelidad, escalabilidad y aislamiento entre máquinas virtuales.

Las pruebas de funcionamiento se realizaron a nivel de aplicación; es decir, se buscó comprobar que, si bien se puede emular una red, también se pueden analizar las aplicaciones realizadas en una red real. Para ello, se planteó una red L3 lineal con dos *hosts* a los extremos que realicen una transferencia de archivos utilizando los protocolos FTP y HTTP.

La segunda prueba presentada es la calibración del emulador. En esta parte, se muestran los resultados de las pruebas de iperf y los cálculos necesarios para hallar la cantidad de paquetes por segundo (pps) que un *core* puede procesar. Además, se establece el tamaño óptimo de cada paquete con el que iniciará el emulador. En base a este valor y la capacidad de cada enlace entre las instancias, se determina el número de *cores* que necesita cada máquina virtual.

Lo siguiente que se evaluará será la fidelidad del emulador. El objetivo principal es comprobar que los resultados obtenidos en una red emulada sean similares a los que se obtuvieron en una red real. Para ello, se va a comparar una red SDN emulada que consta de una topología anillo simple, con tres *switches* interconectados y un *host* conectado a cada uno de los tres nodos. Todos los *switches* se conectarán con un controlador Floodlight mediante una red de *control plane*. Por otro lado, la red física se implementará con equipos reales que se encuentran dentro del Grupo de Investigación de Redes Avanzadas (GIRA) de la universidad.

Por el lado de las pruebas de escalabilidad, se busca verificar que el número de máquinas virtuales está relacionado con las condiciones del experimento. Asimismo, el funcionamiento del validador y la emulación de diferentes *kernels* son puntos a evaluar con estas pruebas. En ellas se utilizaron 2 topologías Estrella: una con 3 y otra con 8 *hosts* alrededor de un *switch* central.

La última prueba que se realizó es la de aislamiento. Esta prueba sirve para verificar que, ante una falla dentro de una máquina virtual, el funcionamiento del sistema no se altera. Para esto, se va a inducir errores en una máquina virtual, tales como uso excesivo de CPUs, memoria y ancho de banda. Para esta prueba, se va a utilizar una topología anillo con *hosts* en diferentes nodos.

5.1. Pruebas de funcionamiento

Para la primera etapa de pruebas, se va a analizar una red L3 *legacy*, en la cual se utilizará el protocolo OSPF para el enrutamiento y se realizará la transferencia de un archivo —usando FTP y HTTP. La red que se empleará será una topología lineal con 2 *switches* L3 Cumulus VX, cada uno con un *host* con SO Linux Server. La finalidad de esta prueba es observar el intercambio correcto de los mensajes del protocolo de enrutamiento, así como de las aplicaciones de empleadas para la transferencia de archivos.

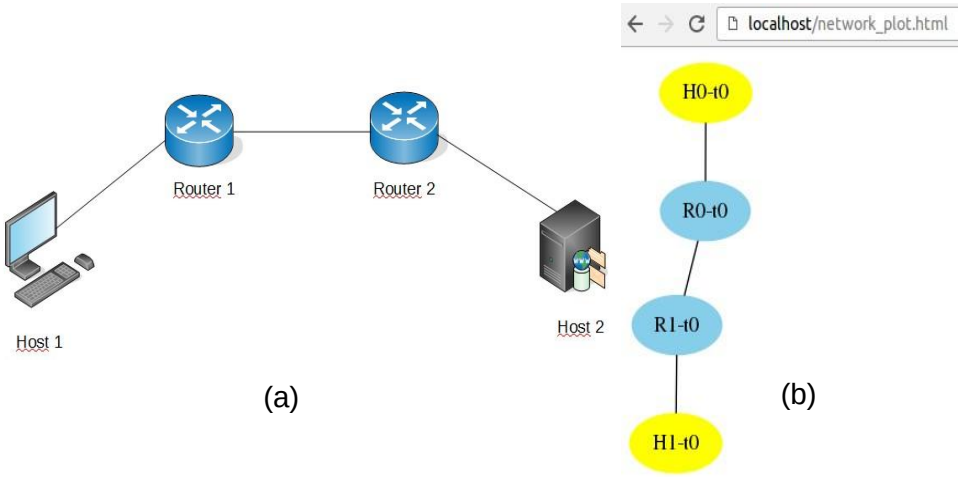


FIGURA 5-1: RED PARA PRUEBAS DE FUNCIONAMIENTO: (a) ESQUEMÁTICO Y (b) EN EL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

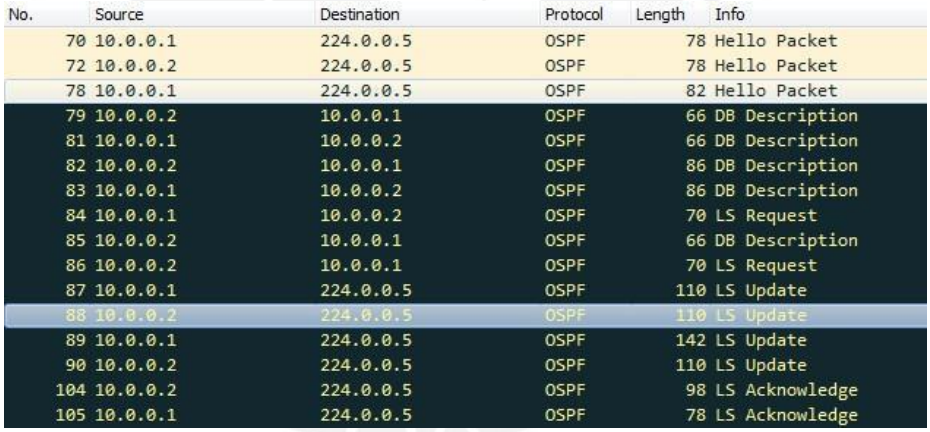
En la imagen anterior se observa la topología que se va emular (izquierda) y la creada en el emulador (derecha). Los *hosts* 1 y 2 corresponden a H0-t0 y H1-t0; los *routers* 1 y 2 corresponden a R0-t0 y R1-t0, respectivamente. Cabe mencionar que, para visualizar la topología en el emulador, se hizo una conexión SSH hacia el servidor utilizando *port forwarding* de los puertos 80 (web) y 6080 (VNC). De esta manera, cuando uno se conecta a los puertos 80 y 6080 locales, la conexión se traduce a los mismos puertos del servidor.

Una vez que se crearon las máquinas virtuales, se procedió a configurar las siguientes direcciones IP en las mismas.

Instancia	SO	interfaz	Dirección IP	Máscara	Gateway
H0-t0	Ubuntu	eth0	192.168.0.10	255.255.255.0	192.168.0.1
R0-t0		swp1	10.0.0.1	255.255.255.252	-
		swp2	192.168.0.1	255.255.255.0	-
R1-t0		swp1	10.0.0.2	255.255.255.252	-
		swp2	172.31.0.1	255.255.255.0	-
H1-t0	Ubuntu	eth0	172.31.0.10	255.255.255.0	172.31.0.1

TABLA 5-1: DIRECCIONAMIENTO DE INSTANCIAS.
FUENTE: ELABORACIÓN PROPIA

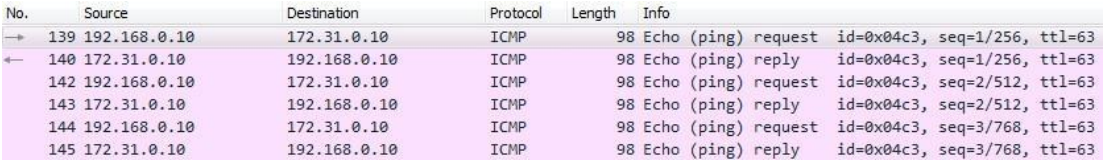
En este punto, se comenzó a capturar tramas (con el programa tcpdump) en la interfaz swp1 de R0-t0; luego, se configuró el protocolo de enrutamiento OSPFv2. En la siguiente imagen se observan los mensajes iniciales que OSPF utiliza para generar adyacencias e intercambiar redes (LSA tipo 1).



No.	Source	Destination	Protocol	Length	Info
70	10.0.0.1	224.0.0.5	OSPF	78	Hello Packet
72	10.0.0.2	224.0.0.5	OSPF	78	Hello Packet
78	10.0.0.1	224.0.0.5	OSPF	82	Hello Packet
79	10.0.0.2	10.0.0.1	OSPF	66	DB Description
81	10.0.0.1	10.0.0.2	OSPF	66	DB Description
82	10.0.0.2	10.0.0.1	OSPF	86	DB Description
83	10.0.0.1	10.0.0.2	OSPF	86	DB Description
84	10.0.0.1	10.0.0.2	OSPF	70	LS Request
85	10.0.0.2	10.0.0.1	OSPF	66	DB Description
86	10.0.0.2	10.0.0.1	OSPF	70	LS Request
87	10.0.0.1	224.0.0.5	OSPF	110	LS Update
88	10.0.0.2	224.0.0.5	OSPF	110	LS Update
89	10.0.0.1	224.0.0.5	OSPF	142	LS Update
90	10.0.0.2	224.0.0.5	OSPF	110	LS Update
104	10.0.0.2	224.0.0.5	OSPF	98	LS Acknowledge
105	10.0.0.1	224.0.0.5	OSPF	78	LS Acknowledge

FIGURA 5-2: MENSAJES DE CONVERGENCIA DE OSPFV2.
FUENTE: ELABORACIÓN PROPIA

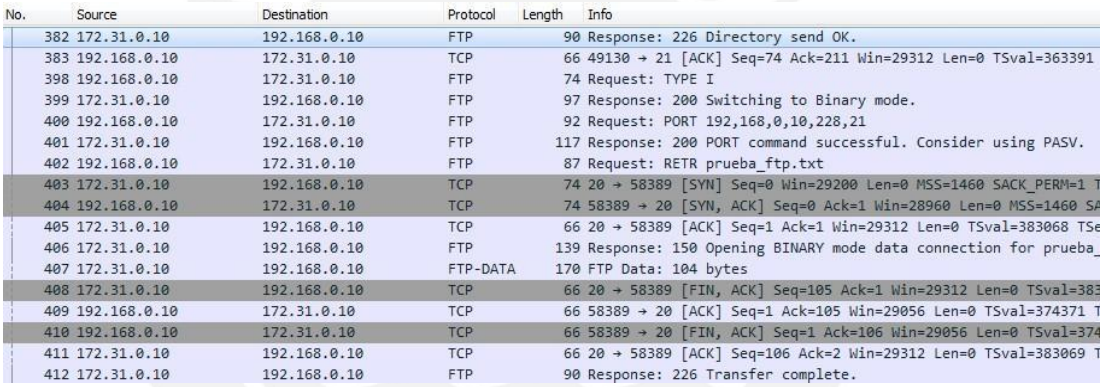
Después, se realizó una prueba de conectividad para confirmar que exista comunicación entre ambos *hosts*.



No.	Source	Destination	Protocol	Length	Info
139	192.168.0.10	172.31.0.10	ICMP	98	Echo (ping) request id=0x04c3, seq=1/256, ttl=63
140	172.31.0.10	192.168.0.10	ICMP	98	Echo (ping) reply id=0x04c3, seq=1/256, ttl=63
142	192.168.0.10	172.31.0.10	ICMP	98	Echo (ping) request id=0x04c3, seq=2/512, ttl=63
143	172.31.0.10	192.168.0.10	ICMP	98	Echo (ping) reply id=0x04c3, seq=2/512, ttl=63
144	192.168.0.10	172.31.0.10	ICMP	98	Echo (ping) request id=0x04c3, seq=3/768, ttl=63
145	172.31.0.10	192.168.0.10	ICMP	98	Echo (ping) reply id=0x04c3, seq=3/768, ttl=63

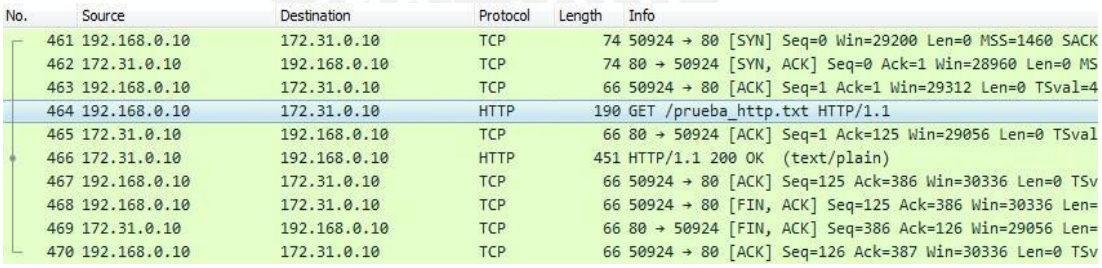
FIGURA 5-3: VERIFICACIÓN DE CONECTIVIDAD ENTRE *HOSTS*.
FUENTE: ELABORACIÓN PROPIA

Finalmente, se procede a ejecutar dos transferencias de archivos, usando los protocolos FTP y HTTP; en ambos casos, H1-t0 hace de servidor y H0-t0 de cliente. Se observa que, para la transferencia del archivo usando FTP, se crean dos conexiones TCP: hacia el puerto 21 del servidor —para la información control— y desde el puerto 20 del servidor, para enviar el archivo. En cambio, en la transferencia usando HTTP, este protocolo utiliza un mensaje HTTP GET para indicar el archivo que se va a transferir.



No.	Source	Destination	Protocol	Length	Info
382	172.31.0.10	192.168.0.10	FTP	90	Response: 226 Directory send OK.
383	192.168.0.10	172.31.0.10	TCP	66	49130 → 21 [ACK] Seq=74 Ack=211 Win=29312 Len=0 TSval=363391
398	192.168.0.10	172.31.0.10	FTP	74	Request: TYPE I
399	172.31.0.10	192.168.0.10	FTP	97	Response: 200 Switching to Binary mode.
400	192.168.0.10	172.31.0.10	FTP	92	Request: PORT 192,168,0,10,228,21
401	172.31.0.10	192.168.0.10	FTP	117	Response: 200 PORT command successful. Consider using PASV.
402	192.168.0.10	172.31.0.10	FTP	87	Request: RETR prueba_ftp.txt
403	172.31.0.10	192.168.0.10	TCP	74	20 → 58389 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T
404	192.168.0.10	172.31.0.10	TCP	74	58389 → 20 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SA
405	172.31.0.10	192.168.0.10	TCP	66	20 → 58389 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=383068 TSe
406	172.31.0.10	192.168.0.10	FTP	139	Response: 150 Opening BINARY mode data connection for prueba_
407	172.31.0.10	192.168.0.10	FTP-DATA	170	FTP Data: 104 bytes
408	172.31.0.10	192.168.0.10	TCP	66	20 → 58389 [FIN, ACK] Seq=105 Ack=1 Win=29312 Len=0 TSval=383
409	192.168.0.10	172.31.0.10	TCP	66	58389 → 20 [ACK] Seq=1 Ack=105 Win=29056 Len=0 TSval=374371 T
410	192.168.0.10	172.31.0.10	TCP	66	58389 → 20 [FIN, ACK] Seq=1 Ack=106 Win=29056 Len=0 TSval=374
411	172.31.0.10	192.168.0.10	TCP	66	20 → 58389 [ACK] Seq=106 Ack=2 Win=29312 Len=0 TSval=383069 T
412	172.31.0.10	192.168.0.10	FTP	90	Response: 226 Transfer complete.

(a)



No.	Source	Destination	Protocol	Length	Info
461	192.168.0.10	172.31.0.10	TCP	74	50924 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK
462	172.31.0.10	192.168.0.10	TCP	74	80 → 50924 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MS
463	192.168.0.10	172.31.0.10	TCP	66	50924 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=4
464	192.168.0.10	172.31.0.10	HTTP	190	GET /prueba_http.txt HTTP/1.1
465	172.31.0.10	192.168.0.10	TCP	66	80 → 50924 [ACK] Seq=1 Ack=125 Win=29056 Len=0 TSval
466	172.31.0.10	192.168.0.10	HTTP	451	HTTP/1.1 200 OK (text/plain)
467	192.168.0.10	172.31.0.10	TCP	66	50924 → 80 [ACK] Seq=125 Ack=386 Win=30336 Len=0 TSv
468	192.168.0.10	172.31.0.10	TCP	66	50924 → 80 [FIN, ACK] Seq=125 Ack=386 Win=30336 Len=
469	172.31.0.10	192.168.0.10	TCP	66	80 → 50924 [FIN, ACK] Seq=386 Ack=126 Win=29056 Len=
470	192.168.0.10	172.31.0.10	TCP	66	50924 → 80 [ACK] Seq=126 Ack=387 Win=30336 Len=0 TSv

(b)

FIGURA 5-4: MENSAJES DE (a) FTP Y (b) HTTP INTERCAMBIADOS.
FUENTE: ELABORACIÓN PROPIA

5.2. Pruebas de calibración

En esta etapa de pruebas, se va realizar la calibración. Esta consiste en hallar la cantidad de paquetes por segundo (pps) que puede procesar un *core* del servidor. En base al pps, al tamaño por paquete (TPP) indicado por el usuario, la cantidad de enlaces y capacidad de cada uno, se puede hallar el número de *cores* necesarios para un nodo (*switch* o *router*).

En pruebas anteriores, se utilizó el programa Iperf para determinar que la cantidad de tráfico que podría enviar una interfaz —sin descartar más de 1% de paquetes (valor máximo permitido para VoIP con códec G.729 [49]) — era aproximadamente 3.5 Gbps. Se eligió utilizar iperf para caracterizar el enlace entre máquinas virtuales, donde la mitad de *hosts* cumplían el rol de clientes transmitiendo en simultáneo a los demás *hosts* (servidores). Para calcular el ancho de banda total, se optó por el tráfico de UDP sobre el de TCP, pues este último lo calcula en base al TCP *Window Size* y *Round Trip Time*.

El primer paso realizado fue hallar la cantidad de pps. Para ello, se plantearon 2 topologías estrellas: uno con 6 y otro con 14 *hosts*; ambas con un solo *switch* central. La prueba consistía en mandar tráfico por varías interfaces del *switch* en simultáneo y sumar el *throughput* de cada una. Esto nos permitió hallar la capacidad de tráfico que puede procesar cada una a un determinado MTU. Los resultados que se obtuvieron son los siguientes.

Cores* por *server	**1**			
Tráfico pedido (Gbps)	***Server* 1**	***Server* 2**	***Server* 3**	**Total (Gbps)**
2.8	2.78	2.79	2.77	8.34
3	2.85	2.9	2.87	8.62
3.5	2.75	2.87	2.78	8.4
			prom.	**8.453333**

(a)

Cores* por *server	**2**							
Tráfico pedido (Gbps)	***Server* 1**	***Server* 2**	***Server* 3**	***Server* 4**	***Server* 5**	***Server* 6**	***Server* 7**	**Total (Gbps)**
2,8	2,19	1,99	2,72	2,8	2,8	2,72	2,21	17,43
3	2,15	2,17	1,45	2,72	2,98	2,97	2,74	17,18
3,5	2,75	2,87	2,78	-	-	-	-	-
							prom.	**17,305**

(b)

TABLA 5-2: TRÁFICO TOTAL EN *SWITCHES* CON: (a) 1 *CORE,* (b) 2 *CORES.*
FUENTE: ELABORACIÓN PROPIA

En el caso de tener dos *cores* por *server*, el tráfico por *core* es 17.305 Gbps. Esto significa que para un solo *core* el tráfico es 8.6525 Gbps, similar a los *throughputs* mostrados en la tabla 5-2 (a). Por lo tanto, se decidió escoger como *throughput* el valor promedio que se muestra en la primera tabla, es decir, 8.4 Gbps.

Para estas pruebas se trabajó con tramas utilizando un MTU máximo; por ello, se procede a hallar la cantidad de bytes que contiene.

Unidad	**B**						
Interframe Gap	**Preámbulo**	**Ethertype**	**MAC Origen**	**MAC Destino**	**MTU**	**CRC**	**Tamaño paquete**
12	8	2	6	6	1500	4	**1538**

TABLA 5-3: TAMAÑO EN BYTES DE PAQUETE CON MTU MÁXIMO.
FUENTE: ELABORACIÓN PROPIA

La cantidad de paquetes por segundo se halla de la siguiente manera:

$$\frac{8.4\ x10^9\ b/s}{1538\ B\ x\ \frac{8\ b}{1\ B}} = 682704{,}81 \sim 682705\ pps$$

El siguiente paso consiste en estimar el tamaño de los paquetes (TPP) que podrían fluir a través de los *switches* que se van a emular. Para elegir el valor inicial de este parámetro, se van a considerar 3 casos: utilizando el MTU mínimo, el promedio y el máximo.

En el primer caso, se emplea el MTU mínimo de Ethernet (46 B). A dicho valor se le agregan las demás cabeceras L2 y L1 para formar el paquete completo. Por lo tanto, el tamaño mínimo de cada paquete ($TPP_{mínimo}$) obtenido es 84 B.

Unidad	B						
Interframe Gap	**Preámbulo**	**Ethertype**	**MAC Origen**	**MAC Destino**	**MTU**	**CRC**	**Tamaño paquete**
12	8	2	6	6	46	4	**84**

TABLA 5-4: TAMAÑO EN BYTES DE PAQUETE CON MTU MÍNIMO.
FUENTE: ELABORACIÓN PROPIA

En el segundo caso, el valor promedio de cada paquete se determina con el IMIX (Internet MIX). Este hace referencia al tráfico de Internet que generalmente circula por la red. Tomando como referencia la distribución de los paquetes (en porcentaje) mostrado en el *Simple* IMIX de un *firewall*, el *payload* promedio de Ethernet se determina de la siguiente forma:

$$40x58{,}3\% + 576x33{,}3\% + 1500x8{,}3\% = 340\ Bytes \sim 341\ Bytes$$

Al añadir las cabeceras de L2 y L1 se consigue un tamaño de cada paquete de 379 B, aproximadamente. Para fines prácticos, se aproxima el tamaño promedio de los paquetes a 400 B.

Unidad	B						
Interframe Gap	**Preámbulo**	**Ethertype**	**MAC Origen**	**MAC Destino**	**MTU**	**CRC**	**Tamaño paquete**
12	8	2	6	6	341	4	**379**

TABLA 5-5: TAMAÑO EN BYTES DE PAQUETE CON *PAYLOAD* PROMEDIO.
FUENTE: ELABORACIÓN PROPIA

En el último caso, se emplea el tamaño máximo de cada paquete, el cual es igual a 1538 B según lo visto en la tabla 5-3.

Asumiendo que se desean emular *switches* con 24 puertos de 1 Gbps cada uno (24 x 1 Gbps), se halla la cantidad de tráfico que debe procesar cada uno (también llamado *switching capacity*).

$$switching\ capacity = 24x10^9 \frac{b}{s}/sw$$

Considerando el peor de los casos —todos los paquetes procesados son de voz (TPP = 84 B) — la cantidad de pps que debería procesar el switch se determina de la siguiente manera:

$$\frac{24x10^9\ \frac{b}{s}/sw}{84\ B/packet} x \frac{1\ B}{8\ b} = 35\ 714\ 285{,}71 \sim 35\ 714\ 286\ pps/sw$$

Luego de obtener la cantidad de paquetes por segundo para el *switch* y el valor de pps por *core*, se determina la cantidad de *cores* que se necesita para emular el *switch*.

$$\frac{35\ 714\ 286\ pps/sw}{682\ 705\ pps/core} = 52{,}31 \sim 53\ cores/sw$$

El resultado anterior muestra que se requieren 53 *cores* para emular el *switch* antes mencionado; sin embargo, los servidores solo cuentan con 32 *cores* cada uno, por lo cual no es posible emularlo.

Dado que al utilizar el tamaño mínimo de los paquetes se excede la cantidad de *cores* que poseen los servidores, ahora se asume un tamaño promedio de cada paquete, es decir, 400 B.

$$\frac{24x10^9\ \frac{b}{s}/sw}{84\ B/packet} x \frac{1\ B}{8\ b} = 7{,}5x10^6\ pps/sw$$

$$\frac{7{,}5x10^6\ pps/sw}{682\ 705\ pps/core} = 10{,}98 \sim 11\ cores/sw$$

Considerando que la cantidad de *cores* que se asignan a una máquina real es potencia de 2, para un *switch* de estas características se deben asignar 16 *cores*. Si bien el número de *cores* requeridos está dentro de lo permitido, solo se podría emular un *switch* de ese tipo en cada servidor.

Ahora bien, si se asume un *switch* con 22 puertos de 100 Mbps más 2 puertos de 1 Gbps (22 x 100 Mbps + 2 x 1 Gbps) que maneje paquetes de 400 B, la cantidad de *cores* que necesita seria:

$$switching\ capacity = (22x100 + 2x1000)x10^6 \frac{b}{s} = 4{,}2x10^9 \frac{b}{s}/sw$$

$$\frac{4{,}2x10^9 \frac{b}{s}/sw}{400\ B/packet} x \frac{1\ B}{8\ b} = 1\ 312\ 500\ pps/sw$$

$$\frac{1\ 312\ 500\ pps/sw}{682\ 705\ pps/cores} = 1{,}92 \sim 2\ cores/sw$$

En caso el tamaño de todos los paquetes fuese 1538 B, la cantidad de *cores* que necesita sería:

$$switching\ capacity = (22x100 + 2x1000)x10^6 \frac{b}{s} = 4{,}2x10^9 \frac{b}{s}/sw$$

$$\frac{4{,}2x10^9 \frac{b}{s}/sw}{1539\ B/packet} x \frac{1\ B}{8\ b} = 341\ 353\ pps/sw$$

$$\frac{341\ 353\ pps/sw}{682\ 705\ pps/cores} = 0{,}5 \sim 1\ cores/sw$$

Después de realizar todos estos cálculos, se puede apreciar que para paquetes de 1538 B se obtiene la menor cantidad de *cores* necesarios para emular un *switch*; no obstante, se estaría asumiendo que el nodo no procesa paquetes de menor tamaño; por este motivo, se elige un TPP de 400 Bytes como el más óptimo para iniciar el proyecto.

5.3. Pruebas de fidelidad

En cuanto a las pruebas de fidelidad, estas consisten en evaluar y comparar las características de una topología, tanto en el emulador como en equipos reales. El objetivo de dichas pruebas es analizar qué tan similares son los resultados obtenidos en ambos escenarios, y así calificar la fidelidad del emulador. La topología implementada para estas pruebas es una red SDN OpenFlow con una topología de anillo simple (1 enlace entre nodos) con 3 *switches*, donde cada uno se encuentra conectado a un *host*. También se tiene una red de *control plane* fuera de banda (no gestionada) para conectar los nodos con el controlador SDN. Todos los enlaces de la topología serán de 100 Mbps y no se les añadirá retardo.

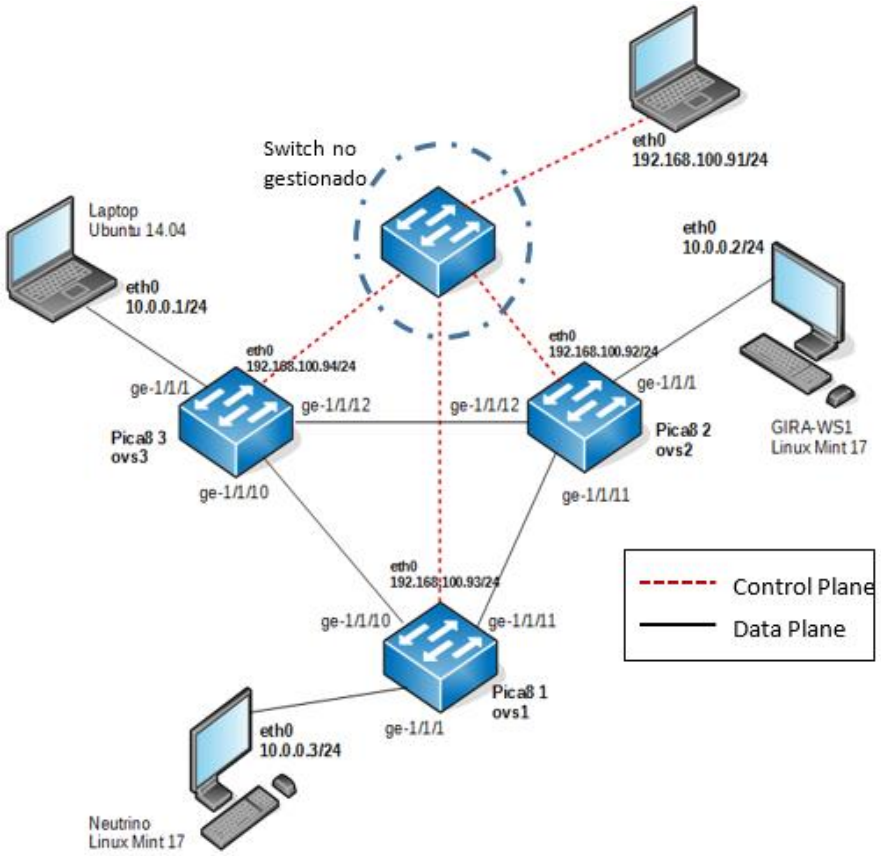


FIGURA 5-5: TOPOLOGÍA PARA LAS PRUEBAS DE FIDELIDAD.
FUENTE: ELABORACIÓN PROPIA

Como parte de las pruebas, se van a utilizar los programas ping e iperf. El primero se utilizará para averiguar el retardo que existe entre los *hosts*; el segundo, para obtener los parámetros de *throughput* en el enlace (con tráfico UDP), porcentaje de pérdida de paquetes y *jitter* del enlace.

Primero, se realizaron las pruebas de fidelidad con equipos físicos. Actualmente, en el laboratorio del GIRA se cuenta con 4 *switches* Pica8 (3 *access switches* P-3297 y 1 *core switch* P-3922) y 2 *workstations*. Ellos fueron utilizados para estas pruebas; específicamente, se usaron los 3 *switches* Pica 8 de acceso, las 2 *workstations* como clientes y 2 *laptops* como cliente y controlador SDN (Floodlight, basado en java), respectivamente.

La prueba de conectividad utilizando ping muestra un comportamiento singular de las redes SDN: el primer paquete tiene un alto retardo, mientras que los subsiguientes presentan un bajo retardo (menor a 1 ms). Esto se debe a que en las redes SDN, cuando un primer paquete llega a un *switch* OpenFlow y este no tiene una entrada en su tabla de *flows* para dicho paquete, se produce un *flow missmatch*; luego, el *switch* envía dicho paquete al controlador SDN, donde el paquete es procesado y el controlador le indica qué hacer con él (p.ej. enviarlo por interfaz gi0/0 o descartar el paquete); después, el *switch* lo envía hacia el siguiente nodo, donde puede o no ocurrir un nuevo *flow missmatch* (y repetirse el proceso anterior); finalmente, luego de seguir el camino establecido por el controlador, el paquete llega al destino.

```
ruben@RUBEN:~/Centauri$ ping 10.0.0.2 -c 10
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=140 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.670 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.684 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.735 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.727 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.698 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.698 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.717 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.700 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.645 ms
```

FIGURA 5-6: PRUEBA DE CONECTIVIDAD ENTRE UNA WORKSTATION Y UNA LAPTOP.
FUENTE: ELABORACIÓN PROPIA

También se realizaron pruebas empleando el programa iperf para caracterizar el camino entre ambos hosts en lo que respecta a capacidad del enlace, pérdida de paquetes y *jitter*. En la tabla 5.6 se muestran los valores obtenidos, en los cuales se observan que el *throughput* del camino entre la laptop con Ubuntu y la *workstation* GIRA-WS1 es similar a la tasa de bits solicitada; además, se observa que el jitter en el camino es bajo (menor a 1 ms), no existen paquetes perdidos y solamente un datagrama llegó fuera de orden.

Host	Tasa de bits solicitada (Mbps)	Tiempo (s)	*Throughput* (Mbps)	*jitter*	% pérdida de paquetes	Datagramas fuera de orden
Laptop Ubuntu	100	10	95,7	0,122	0	1

TABLA 5-6: RESULTADOS DE PRUEBA DE IPERF CON EQUIPOS REALES.
FUENTE: ELABORACIÓN PROPIA

Luego, se implementó la red de prueba en el emulador. Para ello, se utilizaron máquinas virtuales con sistema operativo Linux *server* y a Open vSwitch como *switch* virtual; los *hosts* y el controlador también se emularon como máquinas virtuales.

```
telestack@telestack:~$ ping 10.0.0.2 -c 10
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=99.0 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=7.22 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.23 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=1.03 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=1.02 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.973 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=1.44 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.886 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=1.02 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=1.02 ms
```

FIGURA 5-7: PRUEBA DE CONECTIVIDAD ENTRE MÁQUINAS VIRTUALES.
FUENTE: ELABORACIÓN PROPIA

Como se observa en la figura 5.7, la red emulada presenta un comportamiento similar al de la red real. El primer paquete presenta un alto retardo, mientras que los siguientes paquetes —sin considerar el 2do paquete— presentan un retardo menor a 1,5 ms. En cuanto a la prueba con iperf, se observa que, al igual que con los equipos reales, el *throughput* del enlace es cercano a la tasa de bits solicitada y el *jitter* es muy bajo; sin embargo, se observa una ligera diferencia en relación a la pérdida de paquetes y a la cantidad de datagramas fuera de orden. Esto se puede haber generado debido a que el bus de procesamiento de los servidores se encontraba ocupado en el momento de las pruebas o por la cantidad de Open vSwitches de la red *underlay* de OpenStack en un enlace punto a punto que atraviesan los paquetes.

Host	Tasa de bits solicitada (Mbps)	Tiempo (s)	*Throughput* (Mbps)	*jitter*	% pérdida de paquetes	Datagramas fuera de orden
Laptop Ubuntu	100	10	96,8	0,005	0,027	24

TABLA 5-7: RESULTADOS DE PRUEBA DE IPERF CON MÁQUINAS VIRTUALES.
FUENTE: ELABORACIÓN PROPIA

5.4. Pruebas de escalabilidad

El siguiente conjunto de pruebas que es el de escalabilidad. El objetivo de estas pruebas es comprobar que se pueden emular diferentes tipos de *kernels* y que la cantidad de máquinas virtuales depende de las condiciones de experimento (TPP y total de ancho de banda en instancia).

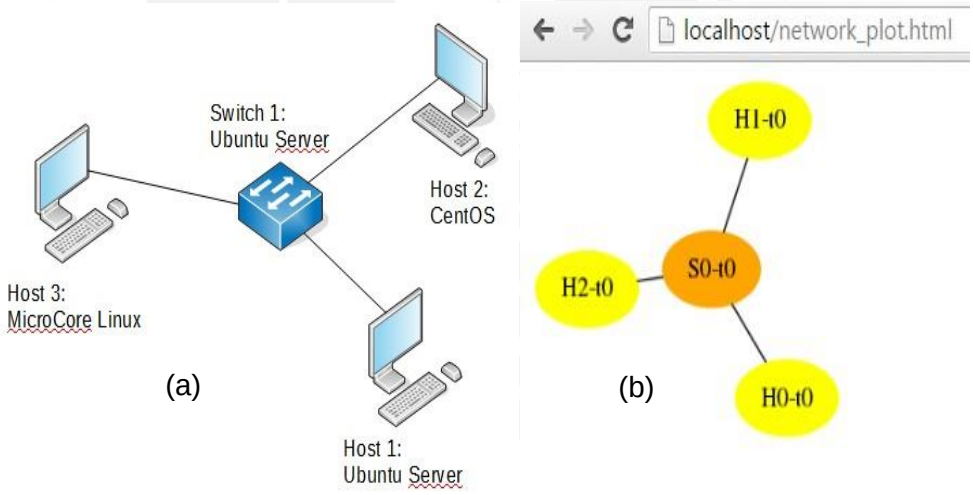


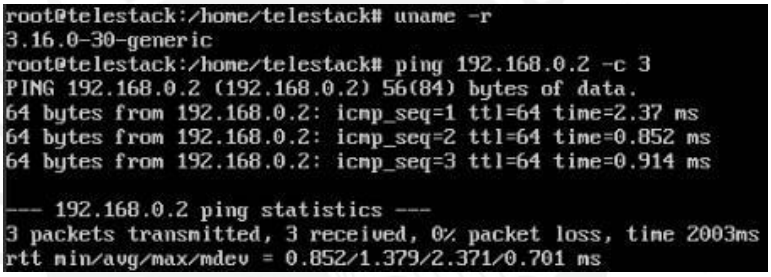
FIGURA 5-8: RED PARA PRIMERAS PRUEBAS DE ESCALABILIDAD: (a) ESQUEMÁTICO Y (b) EN EL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Como se muestra en la figura anterior, se va a utilizar una topología tipo Estrella que tenga un *switch* central y 3 *hosts* con diferentes sistemas operativos (con diferentes versiones de *kernels*). En la siguiente tabla se muestran las direcciones IP de cada equipo y los detalles de los *kernels* que usan.

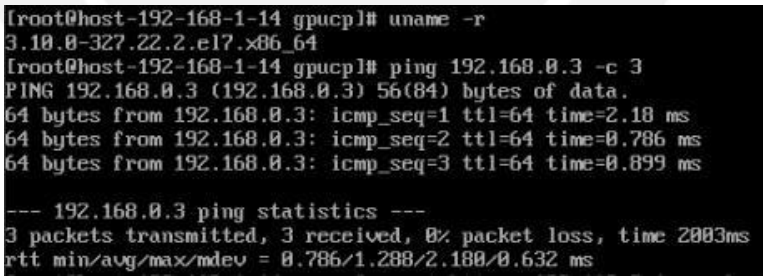
Instancia	interfaz	Dirección IP	Máscara	Sistema Operativo	*Kernel*
H0-t0	eth0	192.168.0.1	255.255.255.248	Ubuntu Server	3.16.0-30generic
H1-t0	eth0	192.168.0.2	255.255.255.248	CentOS	3.10.0-327.22.2.el7.x86_64
H2-t0	eth0	192.168.0.3	255.255.255.248	TinyCORE	3.16.6.-tinycore

TABLA 5-8: DIRECCIONAMIENTO Y *KERNELS* DE INSTANCIAS.
FUENTE: ELABORACIÓN PROPIA

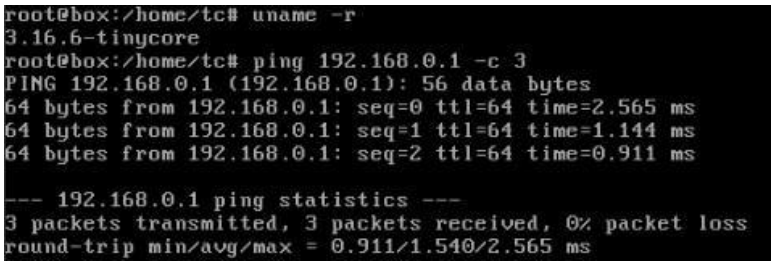
Para verificar que la red emulada funcione correctamente (con máquinas virtuales de diferentes *kernels*), se realiza una prueba de conectividad entre las instancias.



(a)



(b)



(c)

FIGURA 5-9: PRUEBAS DE CONECTIVIDAD: (a) UBUNTU SERVER, (b) CENTOS Y (c) TINY CORE.
FUENTE: ELABORACIÓN PROPIA

Al observar que se obtiene respuesta de todos los *hosts* en las pruebas de conectividad, se procede a transferir un archivo desde H0-t0 hacia H1-t0 utilizando la aplicación wget a través de HTTP.

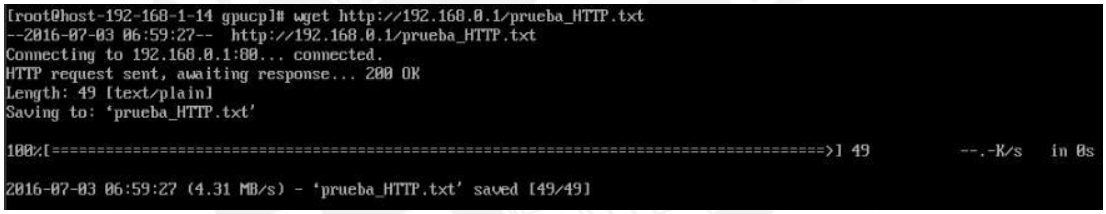


FIGURA 5-10: TRANSFERENCIA DE ARCHIVO DE H0-T0 A H1-T1.
FUENTE: ELABORACIÓN PROPIA

En la imagen anterior, se aprecia que la transferencia del archivo se realiza de forma correcta.

Como segunda parte de esta etapa de pruebas, se va a demostrar que la cantidad de recursos que requiere una topología varía en función a la cantidad de enlaces, la capacidad de cada uno y al tamaño por paquete estimado; para ello, se va a utilizar una topología Estrella con un *switch* central y 8 *hosts* alrededor.

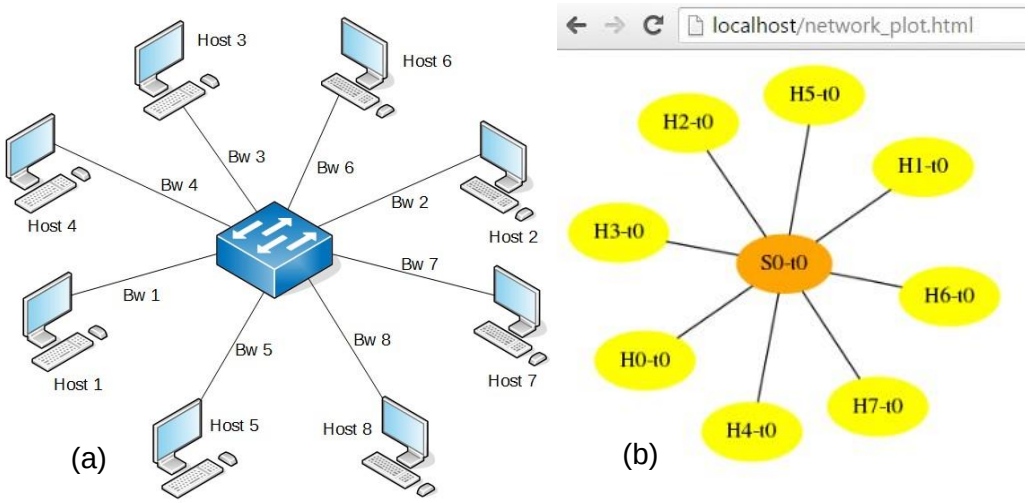


FIGURA 5-11: RED PARA SEGUNDAS PRUEBAS DE ESCALABILIDAD: (a) ESQUEMÁTICO Y (b) EN EL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Se comienza estableciendo los valores de tasa de bits y recursos de los *hosts* con un valor de TPP = 84 B, tal como se detalla en la tabla siguiente. La cantidad de recursos que necesita S0-t0 se halla utilizando el algoritmo explicado en el Anexo 3.

			TOTAL
Característica	**Enlace**	***Bandwidth* (Mbps)**	***Bandwidth* (Mbps)**
Inter-servidor	Bw1	200	800
	Bw3	200	
	Bw5	200	
	Bw7	200	
Intra-servidor	Bw2	500	2000
	Bw4	500	
	Bw6	500	
	Bw8	500	

TABLA 5-9: CAPACIDAD DE ENLACES Y CONSUMO DE ANCHO DE BANDA.
FUENTE: ELABORACIÓN PROPIA

				TOTAL	
Servidor	**Instancia**	**RAM (GB)**	***Cores***	**RAM (GB)**	***Cores***
Nova	H1-t0	8	4	64	32
	H3-t0	8	4		
	H5-t0	16	8		
	H7-t0	16	8		
	S0-t0	16	8		
Control	H0-t0	8	4	32	16
	H2-t0	8	4		
	H4-t0	8	4		
	H6-t0	8	4		

TABLA 5-10: RECURSOS DE CADA INSTANCIA Y CONSUMO DE RECURSOS.
FUENTE: ELABORACIÓN PROPIA

Adicionalmente, se presentan dos nuevas tablas indicando los valores de RAM, *cores* y *bit rate* que requiere la topología que se va a crear y los valores máximos que permite cada servidor.

	Totales		**Máximos**	
Servidor	**RAM (GB)**	***Cores***	**RAM (GB)**	***Cores***
Nova	64	32	58	29
Control	32	16	40	27

TABLA 5-11: RECURSOS REQUERIDOS CON TPP = 84 B Y RECURSOS DISPONIBLES.
FUENTE: ELABORACIÓN PROPIA

	Totales		Máximos	
Servidor	***Bandwidth* Intra (Mbps)**	***Bandwidth* Inter (Mbps)**	***Bandwidth* Intra (Mbps)**	***Bandwidth* Inter (Mbps)**
Nova	2000	800	40000	1000
Control	0		40000	

TABLA 5-12: TASA DE BITS REQUERIDA Y PERMITIDA.

FUENTE: ELABORACIÓN PROPIA

Como se observa en las tablas 5-9 y 5-10, al considerar un valor de TTP = 84 B, la cantidad de recursos que consume S0-t0 superan la cantidad de RAM y *cores* permitidos en el servidor Nova. Al querer implementar la topología, se muestra el siguiente mensaje:

```
[8] Modificar TPP
[9] Implementar
[10] Salir

Opcion = 9

Las topologias de tu proyecto pueden ser implementarse como SDN

Topologias SDN [Y/N]: n

ERROR: Deberia reducir la cantidad de cores del servidor Nova en 3
ERROR: Deberia reducir la cantidad de RAM del servidor Nova en 6
ERROR: No se implemento ninguna topologia. Por favor borrar la(s)
topologia(s) y volver a crear
```

FIGURA 5-12: MENSAJE DE ERROR DEL VALIDADOR.

FUENTE: ELABORACIÓN PROPIA

Sin embargo, si se aumenta el valor de TPP a 200 B, se observa que la cantidad de recursos que consume S0-t0 es igual a 8 GB de RAM y 4 *cores*, y se procede a hacer la comparación con los recursos disponibles.

	Totales		Máximos	
Servidor	**RAM (GB)**	***Cores***	**RAM (GB)**	***Cores***
Nova	56	28	58	29
Control	32	16	40	27

TABLA 5-13: RECURSOS REQUERIDOS CON TPP = 200 Y RECURSOS DISPONIBLES.

FUENTE: ELABORACIÓN PROPIA

En la tabla anterior se puede observar que al aumentar el valor del TPP, la cantidad de recursos que necesita la topología es menor a los valores máximos.

5.5. Aislamiento entre máquinas virtuales

En esta última etapa, se probará el aislamiento de máquinas virtuales ante alguna falla; en otras palabras, se comprobará que la falla de una máquina virtual no afecta el funcionamiento de los servidores ni el de las instancias. Los errores que se van a inducir en la máquina virtual son los siguientes: uso de ancho de banda, vCPU y memoria mayor a los asignados a cada una.

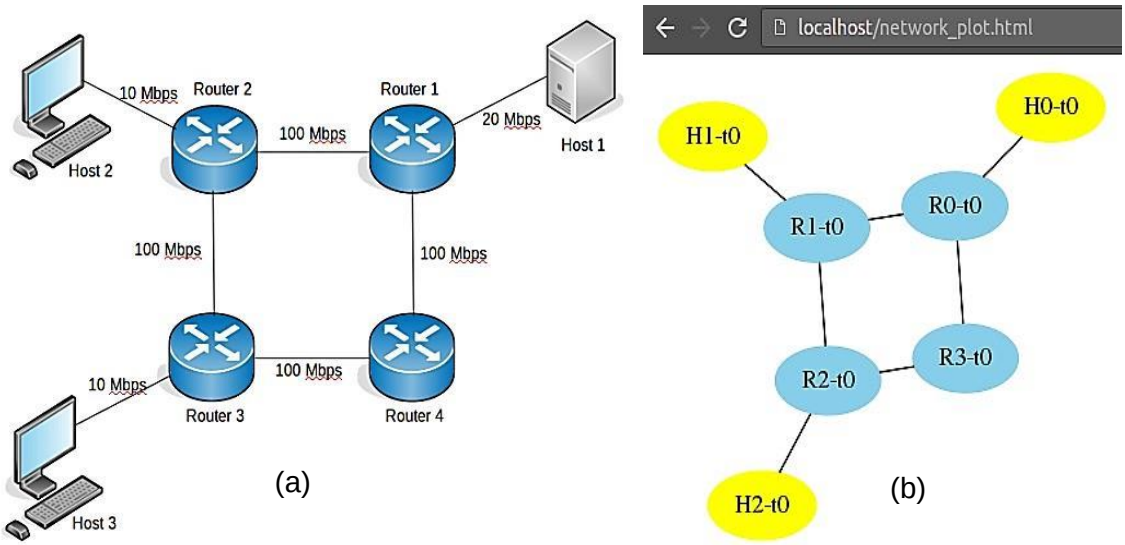


FIGURA 5-13: RED PARA PRUEBAS DE AISLAMIENTO: (a) ESQUEMÁTICO Y (b) EN EL EMULADOR.
FUENTE: ELABORACIÓN PROPIA

Instancia	SO	Interfaz	Dirección IP	Máscara	*Gateway*
R0-t0	Cumulus	swp1	10.0.0.1	255.255.255.252	-
		swp2	10.0.0.14	255.255.255.252	-
		swp3	192.168.0.1	255.255.255.0	-
R1-t0	Cumulus	swp1	10.0.0.2	255.255.255.252	-
		swp2	10.0.0.5	255.255.255.252	-
		swp3	172.19.0.1	255.255.255.0	-
R2-t0	Cumulus	swp1	10.0.0.6	255.255.255.252	-
		swp2	10.0.0.9	255.255.255.252	-
		swp3	10.0.10.1	255.255.255.0	-
R3-t0	Cumulus	swp1	10.0.0.10	255.255.255.252	-
		swp2	10.0.0.13	255.255.255.252	-
H0-t0	Ubuntu	eth0	192.168.0.100	255.255.255.0	192.168.0.1
H1-t0	Ubuntu	eth0	172.19.0.10	255.255.255.0	172.19.0.1
H2-t0	Ubuntu	eth0	10.0.10.10	255.255.255.0	10.0.10.1

TABLA 5-14: DIRECCIONAMIENTO DE INSTANCIAS.
FUENTE: ELABORACIÓN PROPIA

La topología que se va a emplear es un anillo de 4 nodos, en los cuales se está ejecutando el protocolo OSPFv2. Se determina el camino preferido (R2-t0 - R1-t0 - R0-t0) definiendo los costos de las interfaces del camino R2-t0 – R3-t0 – R0-t0 a 65535. Adicionalmente, se utiliza el valor por defecto de TPP y se emplean enlaces de 10 Mbps en los *hosts* y 100 Mbps entre *routers*. Los *hosts* H1-t0 y H2-t0 serán clientes de iperf, mientras que H0-t0 será servidor. Todas las pruebas se realizarán con tráfico UDP.

Primero se realiza una prueba de iperf entre los clientes y el servidor, utilizando un script que permite realizar las generar tráfico al mismo tiempo (ver Anexo 4). Se solicita un tráfico con una tasa de bits cercana a la capacidad nominal del enlace, que sirva como punto de comparación en las pruebas.

Reporte del servidor						
Host	**Tasa de bits solicitada (Mbps)**	**Tiempo (s)**	**Bytes generados (MB)**	***Throughput* (Mbps)**	***jitter* (ms)**	**% pérdida paquetes**
H1-t0	9,7	30	34,8	9,69	0,142	0
H2-t0	9,7		34,8	9,70	0,325	0

TABLA 5-15: PRUEBAS DE IPERF CON TASA DE BITS MÁXIMA.
FUENTE: ELABORACIÓN PROPIA

Luego, se indica que H1-t1 genere tráfico con una mayor tasa de bits, para verificar que el *throughput* entre dicha instancia y el servidor no supere la capacidad del enlace (10 Mbps) y no afecte la conexión de H2-t1 con el servidor (el tráfico de ambos pasa por R1-t0).

Reporte del servidor						
Host	**Ancho de banda solicitado (Mbps)**	**Tiempo (s)**	**Bytes generados (MB)**	***Throughput* (Mbps)**	***jitter* (ms)**	**% pérdida paquetes**
H0-t0	9,7	30	34,8	9,68	0,213	0
H1-t0	100		34,7	9,34	122,159	31

TABLA 5-16: PRUEBAS DE IPERF CON TASA DE BITS SUPERIOR A CAPACIDAD DEL ENLACE.
FUENTE: ELABORACIÓN PROPIA

A pesar de que una máquina virtual intente de generar tráfico con una tasa de bits mayor a la capacidad del enlace, no podrá superarla. Esto se observa en la interfaz swp3 del *router* R2-t0, en donde la diferencia de bytes recibidos (38,7 MB) es similar al tráfico generado por H1-t0 (34,7 MB).

Interfaz swp3 de R2-t0			
Bytes Rx antes (MiB)	**Bytes Rx después (MiB)**	**Diferencia (MiB)**	**Diferencia (MB)**
355,6	392,1	36,5	38,3

TABLA 5-17: PRUEBAS DE IPERF CON TASA DE BITS SUPERIOR A CAPACIDAD DEL ENLACE.
FUENTE: ELABORACIÓN PROPIA

Como segunda parte de esta etapa, se va a realizar el aislamiento de la memoria y procesador de cada máquina virtual. En esta parte, un solo cliente (H2-t0 en este caso) envía tráfico UDP al servidor, utilizando iperf, por 1 minuto. Después, se vuelve a realizar la misma prueba, pero con la diferencia de que en el *host* H1-t0 se ejecutará un script (*Fork bomb* [50], ver Anexo 4) para saturar su CPU. Como H1-t0 se encuentra en el mismo servidor que R2-t0 (por donde pasa el tráfico), se desea verificar que este evento no se refleje en la prueba de iperf.

Reporte del servidor						
Condición de H1-t0	**Ancho de banda solicitado (Mbps)**	**Tiempo (s)**	**Bytes generados (MB)**	***Throughput* (Mbps)**	***jitter* (ms)**	**% pérdida paquetes**
Sin *Fork bomb*	9,7	300	347	9,7	0,071	0
Con *Fork bomb*	9,7		347	9,7	0,117	0

TABLA 5-18: PRUEBAS DE IPERF SIN Y CON FORK BOMB EN PARALELO.
FUENTE: ELABORACIÓN PROPIA

La tabla anterior muestra que, ante la saturación de los recursos de la instancia, la prueba de iperf no se vio afectada en los valores hallados.

Adicionalmente, se realizaron mediciones de los recursos consumidos en el servidor donde falla la máquina virtual, antes y durante las pruebas de iperf, con y sin el *Fork bomb*.

Condición de H1-t0	Condición de H0-t0	CPU (%)	RAM (MB)	Diferencia de CPU (%)	Diferencia de CPU (*cores*)	Diferencia de RAM (MB)
Ejecutando iperf	Sin *Fork bomb*	0,8	3254	3	0,96	1921
	Con *Fork bomb*	3,8	5175			
Sin Ejecutar iperf	Sin *Fork bomb*	0,7	2981	2,8	0,9	2181
	Con *Fork bomb*	3,5	5162			
				Promedio	**0,93**	**2051**
				***Flavor* de la instancia**	**1**	**2048**

TABLA 5-19: RECURSOS CONSUMIDOS CON IPERF Y FORK BOMB.
FUENTE: ELABORACIÓN PROPIA

La tabla 5-19 muestra los resultados de las pruebas antes mencionadas. Se puede apreciar que existe una diferencia de, aproximadamente, 3% del consumo de CPU (0,93 *core*) y de 2051 MB de RAM. Estos valores son muy cercanos a los asignados en el *flavor* de OpenStack: 1 *core* y 2048 MB de RAM.

Conclusiones y Recomendaciones

- Conclusiones
    - Se logró cumplir con el objetivo principal de la tesis, pues se realizó el diseño e implementación de un emulador de redes sobre OpenStack, el cual permite introducir limites en la tasa de bits y retardos en los enlaces. También se evidencia que el emulador presenta una alta fidelidad en sus resultados (se observan fenómenos que ocurren en redes físicas), así como una capacidad media-alta en el número de nodos (si bien en el *rack* actual se pueden emular más de 50 máquinas virtuales, cada una tendría 1GB de RAM y 1 *core*).
    - Se observa que la cantidad de servidores utilizados para OpenStack no influye en el funcionamiento del emulador, pero sí en la escalabilidad debido a la cantidad de recursos disponibles. Al utilizar un entorno de 2 servidores conectados por un enlace de 1 Gbps en vez del ExoGENI *rack*, se evidencia que el funcionamiento del emulador es el esperado a pesar de no estar en un *rack* con gran cantidad de recursos.
    - El validador cumple su objetivo de proteger al sistema de quedarse sin recursos e introducir mediciones erróneas en la evaluación de prototipos. En las pruebas de escalabilidad, se muestra cómo el emulador evita estas situaciones, además de la operación del módulo de topología alternativa que indica qué recursos reducir.
    - Se concluye que las fallas que se presenten en una máquina virtual no influyen en el funcionamiento del emulador. Esto se evidencia en las pruebas de aislamiento, donde al saturar los *cores*, RAM y enlace de la máquina virtual no se afectan las pruebas en paralelo ni el desempeño del servidor.
- Recomendaciones
    - Los datos necesarios tanto para la calibración como para el validador (p.ej. la velocidad de los enlaces entre servidores, cantidad de enlaces, cantidad de *cores*, etc.) dependen de las características de los servidores. Por tal motivo, es necesario que estos valores sean lo más

precisos posible, ya que podría realizarse una mala calibración con respecto al *hardware*. Esto último podría generar resultados incorrectos de la emulación.

- La cantidad máxima de equipos que se pueden emular varía de acuerdo al número de servidores que utilice OpenStack y la cantidad de recursos que posea cada uno. Es por ello que se recomienda que el emulador se ejecute sobre entornos grandes para emular redes de mediana y gran escala, así como establecer mayor cantidad de condiciones respecto al TPP y la capacidad de los enlaces.

- El emulador se desarrolló en base a la versión 2.7 de Python y a la versión Kilo de OpenStack. Se recomienda utilizar entornos con dicha versión de Python para evitar problemas de compatibilidad con las funciones y librerías importadas. Por el lado de OpenStack, es deseable que se implemente en *racks* con la misma versión; no obstante, es posible desplegarlo en otras versiones, siempre y cuando se realicen las modificaciones en el módulo de Instance Creation y en el código de los servicios de Nova y Neutron.

Bibliografía

[1] MORREALE, Patricia y James ANDERSON

2015 *Software Defined Networking: Design and Deployment.*

Florida: CRC Press, pág. 28, 29

[2] NATIONAL SCIENCE FOUNDATION

NSF Future Internet Architecture Project. Consulta: 10 de mayo de 2016

http://www.nets-fia.net/

[3] GÖRANSSON, Paul y Chuck BLACK

2014 *Software Defined Networks: A Comprehensive Approach.*

Massachusetts: Morgan Kaufmann

[4] NATIONAL SCIENCE FOUNDATION

Future Internet Architecture: Program Solicitation. Consulta: 10 de mayo de 2016

http://www.nsf.gov/pubs/2010/nsf10528/nsf10528.htm#pgm_intr_txt

[5] OPEN NETWORKING FOUNDATION

2012 “Software-Defined Networking: The New Norm for Networks”.

California. Consulta: 10 de mayo de 2016.

https://www.opennetworking.org/images/stories/downloads/sdnresources/white-papers/wp-sdn-newnorm.pdf

[6] MCKEOWN, Nick y otros

2008 "OpenFlow: Enabling Innovation in Campus Networks". *ACM SIGCOMM Computer Communication Review*. Volumen 38, número 2, pp. 69-74. Consulta: 10 de mayo de 2016.

[7] ESNET ENERGY SCIENCES NETWORK

The Network: A Nationwide Platform for Science Discovery. Consulta: 10 de mayo de 2016.

https://www.es.net/engineering-services/the-network/

[8] ESNET ENERGY SCIENCES NETWORK

OSCARS: On-Demand Secure Circuits and Advance Reservation System. Consulta: 10 de mayo de 2016.

https://www.es.net/engineering-services/oscars/

[9] NAOUS, Jad y otros

2008 "NetFPGA: Reusable Router Architecture for Experimental Research". Ponencia presentada en *P RESTO '08.* SIGCOMM. Seattle, 22 de agosto.

[10] KIM, Hyunmin y otros

2014 "Developing a Cost-Effective OpenFlow Testbed for Small-Scale Software Defined Networking". Ponencia presentada en *ICACT2014*. IEEE-Advancing Technology for Humanity. Pyeongchang, 16-19 de febrero.

[11] OFELIA

OpenFlow in Europe: Linking Infrastructure and Applications. Consulta: 03 de junio de 2016.

http://www.fp7-ofelia.eu/about-ofelia/

[12] FEDERATION FOR FUTURE INTERNET RESEARCH AND EXPERIMENTATION

Ubristol Ofelia Island Properties. Consulta: 03 de junio de 2016.

http://www.fed4fire.eu/ubristol-ofelia-island-properties/

[13] BEURAN, Razvan

2013 *Introduction to network emulation*. Danvers: Pan Stanford Publishing, pp. 11, 30 – 32.

[14] NEUMANN, Jason

2015 *The book of GNS3: build virtual network labs using Cisco, Juniper, and more*. San Francisco: No Starch Press, Inc.

[15] NS-3

What is NS-3. Consulta: 03 de junio de 2016.

https://www.nsnam.org/overview/what-is-ns-3/

[16] NS-3

Topology Generator. Consulta: 03 de junio de 2016.

https://www.nsnam.org/wiki/Topology_Generator

[17] LANTZ, Bob y otros

2010 "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". Ponencia presentada en HotNets-IX. SIGCOMM. Monterey, 21-22 de octubre.

[18] ROY, Arup y otros

2014 "Design and Management of DOT: A Distributed OpenFlow Testbed". IEEE-Advancing Technology for Humanity.

[19] EXOGENI

Welcome to ExoGENI. Consulta: 03 de junio de 2016.

http://www.exogeni.net/

[20] BALDINE, Ilia y otros

2012 "ExoGENI: A Multi-Domain Infrastructure-as-a-Service Testbed". Ponencia presentada en la 8va Conferencia Internacional ICST. TidentCom. Thessaloniki, junio.

[21] SANTANA, Gustavo

2014 *Data Center Virtualization Fundamentals.* Indianapolis:Cisco Press, pp. 47 – 52.

[22] MANAGE ENGINE

Virtualization Monitoring. Consulta: 29 de junio de 2016.

https://www.manageengine.com/products/applications_manager/virtualization-monitoring.html

[23] IBM DEVELOPER WORKS

Orchestrating the cloud to simplify and accelerate service delivery. Consulta: 29 de junio de 2016.

https://www.ibm.com/developerworks/community/blogs/9e696bfa-94af-4f5aab50c955cca76fd0/entry/orchestrating_the_cloud_to_simplify_and_accelerate_s ervice_delivery1?lang=en

[24] DIVERSITY LIMITED

2011 CLOUD U: UNDERSTANDING The Cloud Computing Stack SaaS, PaaS, IaaS.

http://broadcast.rackspace.com/hosting_knowledge/whitepapers/Understanding-the-Cloud-Computing-Stack.pdf

[25] MUÑOZ, Jose

s/f *Datacenter Technologies.* pp. 109 – 110.

[26] MORREALE, Patricia y otros

2015 *Software Defined Networking Design and Deployment*. Boca Raton: CRC Press, pp. 9 – 12.

[27] OPENSTACK FUNDATION

2015 *OpenStack Virtual Machine Image Guide.*

[28] MICROSOFT AZURE

About images for virtual machines. Consulta: 29 de junio de 2016.

https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-classic-about-images/

[29] PETTIT, Justin y otros

s/f "Virtual Switching in an Era of Advanced Edges". Consulta: 29 de junio de 2016.

http://benpfaff.org/papers/adv-edge.pdf

[30] MAKITA, Toshiaki

2014 *Virtual switching technologies and Linux bridge* [diapositiva]. Consulta: 29 de junio de 2016. http://events.linuxfoundation.org/sites/events/files/slides/LinuxConJapan2014_makita_0.pdf

[31] TECHNET

What is VPN?. Consulta: 29 de junio de 2016.

https://technet.microsoft.com/en-us/library/cc739294(v=ws.10).aspx

[32] OCAMPO, Antonio

s/f "Multicast". Material del curso de Banda Ancha. Lima: Pontifica Universidad Católica del Perú.

[33] PACKETLIFE.NET

Visualizing tunnels. Consulta: 29 de junio de 2016.

http://packetlife.net/blog/2008/jul/11/visualizing-tunnels/

[34] DISCOPOSSE.COM

Why VXLAN is awesome, and why it may not be. Consulta: 29 de junio de 2016.

http://discoposse.com/2012/10/27/why-vxlan-is-awesome-and-why-it-maynot-be/

[35] THE NETWORK ARBORIST

VXLAN for Layer 2 strech over L3 network. Consulta: 29 de junio de 2016.

http://network-arborist.blogspot.pe/2014/08/vxlan-for-layer-2-stretch-overl3.html

[36] SDX CENTRAL

What are Network Overlays?. Consulta: 29 de junio de 2016.

https://www.sdxcentral.com/sdn/network-virtualization/resources/get-on-topof-network-overlays/

[37] LINUXWALL'S WIKI

Journey to the Center of the Linux Kernel: Traffic Control, Shaping and QoS. Consulta: 29 de diciembre de 2016.
http://wiki.linuxwall.info/doku.php/en:ressources:dossiers:networking:traffic_control

[38] EXCENTIS

Use Linux Traffic Control as impairment node in a test environment (part 2). Consulta: 29 de junio de 2016.

https://www.excentis.com/blog/use-linux-traffic-control-impairment-nodetest-environment-part-2

[39] CODERO HOSTING ON DEMAND

How to Choose the Right Type of Storage Solution for Your Needs? Consulta: 29 de junio de 2016.

http://www.codero.com/blog/how-to-choose-storage-solution/

[40] FCOE: Networks & Storage Fabrics Convergence

Drivers for the NextGen Datacenter Infrastructure. Consulta: 04 de julio del 2016.

[41] SELESTA NETWORKS

Aperto Networks. Consulta: 29 de junio de 2016.

http://selestanetworks.com/aperto.html

[42] SLIDESHARE

2010 FCAPS [diapositiva]. Consulta: 29 de junio de 2016.

http://es.slideshare.net/telematica12/fcaps

[43] OPENSTACK FUNDATION

2015 *OpenStack Installation Guide for Ubuntu 14.04.*

[44] OPENSTACK

OpenStack The 11th release of OpenStack delivers stable core of compute, storage and networking services to foster an ecosystem of innovation. Consulta: 26 de febrero de 2016.

https://www.openstack.org/software/kilo/press-release/

[45] OPENSTACK

Message queuing. Consulta: 26 de febrero del 2016

http://docs.openstack.org/security-guide/messaging.html

[46] OPENSTACK FUNDATIONS

2014 *OpenStack Operations Guide.*

[47] RED HAT CUSTOMES PORTAL

Images and Instances. Consulta: 29 de junio de 2016.

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_OpenStack_Platform/5/html/Cloud_Administrator_Guide/section_compute-images-and-instances.html

[48] THE NETWORK WAY – NIR YECHIEL'S BLOG

LLDP traffic and Linux bridges. Consulta: 29 de junio de 2016.

https://thenetworkway.wordpress.com/2016/01/04/lldp-traffic-and-linuxbridges/

[49] CISCO

2001 Quality of Service for Voice IP.

http://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/qos_solutions/QoSVoIP/QoSVoIP.pdf

[50] NIXCRAFT

Understanding Bash fork() Bomb ~ :(){ :|:& };:. Consulta: 03 de julio de 2016.

http://www.cyberciti.biz/faq/understanding-bash-fork-bomb/

http://imexresearch.com/newsletters/Feb09/fcoe.html