Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Gestor del Laboratorio Académico de Redes

Autores: Joaquín Correa, Matías Davyt
Tutores: Eduardo Grampín, Javier Baliosián

Montevideo, Uruguay
Diciembre 2017

Facultad de Ingeniería, Universidad de la República
Instituto de Computación

Resumen

La habilidad de realizar experimentos sobre redes de computadoras y sistemas distribuidos no es una tarea sencilla debido a la alta complejidad en sus estructuras y diversidad de componentes. Los costos de mantenimiento, implantación y configuración de las infraestructuras necesarias son muy altos para ser realizados manualmente cada vez que un nuevo experimento o teoría lo requiere.

Es de interés para el ámbito académico poder realizar pruebas de manera sencilla sobre infraestructuras configurables que permitan validar y repetir dichas pruebas de manera mas o menos precisa como para obtener resultados trascendentes. Ademas de que esto es un aspecto fundamental del método científico, sirve para ámbito académico desde el punto de vista de la enseñanza.

El Laboratorio Académico de Redes (LAR), de la Facultad de Ingeniería, consiste en algunas decenas de routers domésticos y servicios virtualizados que se pueden interconectar en forma arbitraria para experimentar en diversos aspectos de las redes, desde algoritmos de enrutamiento a estudios de desempeño de aplicaciones.

Las infraestructuras dispuestas para estas pruebas son llamadas testbeds (o bancos de pruebas). Existen diversas organizaciones, generalmente universidades, que ofrecen este tipo de servicios. Ellas fueron el punto de partida para investigar el tema.

En el presente trabajo se realiza, en primer lugar, una investigación de las tecnologías más utilizadas en las testbeds existentes y la capacidad de éstas de ser utilizadas en el proyecto. La investigación sobre las testbeds existentes aportó información sobre las tecnologías utilizadas, todas de código abierto, para la realización de experimentos, pero poco se obtuvo sobre cómo se gestionan internamente los recursos.

Luego, con la decisión de qué tecnología existente se utiliza y qué se debería crear para lograr una testbed funcional, se discute el diseño de los nuevos componentes y sus interfaces con lo ya existente.

Finalmente, se implementó una primera versión de un gestor para el LAR, que incluye tanto herramientas para la configuración de topologías compuestas por redes de datos que interconectan nodos con capacidad de computo como integración con tecnologías ya existentes para el orquestado centralizado de recursos distribuidos, logrando una infraestructura que puede ser fácilmente amoldada a los requerimientos de un usuario, junto con la capacidad de dirigir los nodos de manera centralizada. El sistema implementado contiene las funcionalidades básicas necesarias para gestionar la infraestructura disponible y ejecutar experimentos. Se trata de una versión totalmente funcional, con un conjunto de funciones acotado pero suficiente para un uso interno al instituto de computación.

Palabras clave: Testbeds, Redes experimentales, Virtual LAN (VLAN), Virtualización Linux.

Índice general

1 Introducción

La habilidad de repetir experimentos a partir de un trabajo de investigación y obtener resultados similares es un aspecto fundamental del método científico. Esto es particularmente difícil de conseguir en el ámbito de la investigación en computación distribuida y redes. El avance en la tecnología y el constante crecimiento en las redes de computadoras incorporando nuevos dispositivos hace imperiosa la necesidad de crear nuevas tecnologías que permitan explotar este crecimiento.

Para que una nueva tecnología sea aceptada por la comunidad, debe ser probada en condiciones reales. Las simulaciones de redes proveen un mecanismo rápido y económico para la realización de pruebas y obtención de resultados. Si bien este mecanismo es muy útil como un primer acercamiento, las simulaciones implican un modelo del mundo simplificado, que no se corresponde con un escenario de uso realista.

Los principales problemas al momento de experimentar con una nueva tecnología en ambientes reales son la dificultad en el abastecimiento, configuración y control de los recursos que interactúan y sus dependencias. Para solucionar este problema existen las testbeds (bancos de pruebas), término utilizado generalmente en el ámbito de la computación[27].

Una testbed es una plataforma en la cual dirigir de manera rigurosa, transparente y repetible experimentos sobre teorías computacionales, herramientas y nuevas tecnologías. En el ámbito de sistemas distribuidos y redes, las testbeds constan esencialmente de nodos capaces de ejecutar código y una red que los interconecta, además de una infraestructura que gestiona, particiona y asigna los recursos a diferentes experimentadores.

Dado el amplio espectro de tecnologías sobre las que se puede implantar una red de computadoras (red cableada, WiFi, redes móviles, etc) puede ser muy costoso para una organización abordarlas a todas, por lo que éstas generalmente se concentran en un tipo de tecnología. La federación de testbeds permite que varias testbeds se comporten como una única plataforma, exponiendo los recursos disponibles de varias de ellas para su utilización por parte de los usuarios en un mismo experimento. Esto permite diversificar las tecnologías utilizadas y escalar en el tamaño y la complejidad del experimento de manera más asequible.

A diferencia de otros campos de investigación, la investigación de nuevas tecnologías de redes carece de una cultura de verificación rigurosa de resultados experimentales. Esto se debe no sólo a la dificultad de desplegar una infraestructura idéntica o similar a la utilizada originalmente en un experimento, sino a la imposibilidad de describir de forma no ambigua los recursos necesarios para llevarlo a cabo, la ejecución del mismo y las medidas a tomar, de manera que otros investigadores sean capaces de replicarlo[23].

2 Objetivos del proyecto

El grupo MINA (Network Management | Artificial Intelligence) se está enfocando fuertemente hacia la experimentación, en las áreas de robótica móvil y redes. Actualmente, en conjunto con los grupos de Investigación Operativa y Seguridad Informática del INCO, el Laboratorio de Probabilidad y Estadística del IMERL y el grupo ARTES del IIE se está construyendo el LAR (Laboratorio Académico de Redes), financiado por CSIC, y en colaboración con el grupo del Cluster de Cómputo de FING.

El LAR consiste en algunas decenas de routers domésticos que se pueden interconectar en forma arbitraria para experimentar en diversos aspectos de las redes, desde algoritmos de enrutamiento a estudios de performance de aplicaciones (ver Figura 2.1). Para la interconexión se dispone de conmutadores con soporte de VLANs, y los routers ejecutan un sistema operativo abierto (OpenWRT[15]) con herramientas de routing Quagga[18] o similar. También se cuenta con dispositivos especializados con hardware reconfigurable (NETFPGA) y potentes servidores para virtualización y simulación.

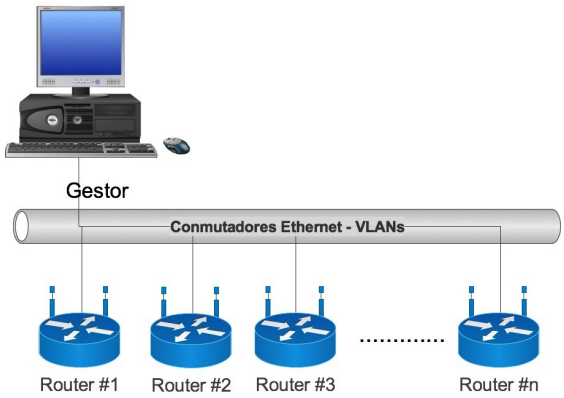


Figura 2.1: Vista general del LAR

Se busca que esta infraestructura esté disponible para investigación y docencia en el área de redes, y para esto es imprescindible contar con herramientas de gestión del laboratorio que permitan definir topologías de prueba, que eventualmente combinen dispositivos físicos y emulados, y generar configuraciones de base para hacer experimentos en dichas topologías.

Este proyecto busca generar herramientas de gestión para el LAR, que permitan definir la topología y otros aspectos básicos del experimento a realizar, y sean capaces de generar las configuraciones necesarias en los dispositivos. De esta forma es posible asignar porciones de la infraestructura para ejecutar diferentes experimentos en forma simultánea;

una vista posible del LAR con M particiones lógicas usando VLANs se ve en la Figura 2.2.

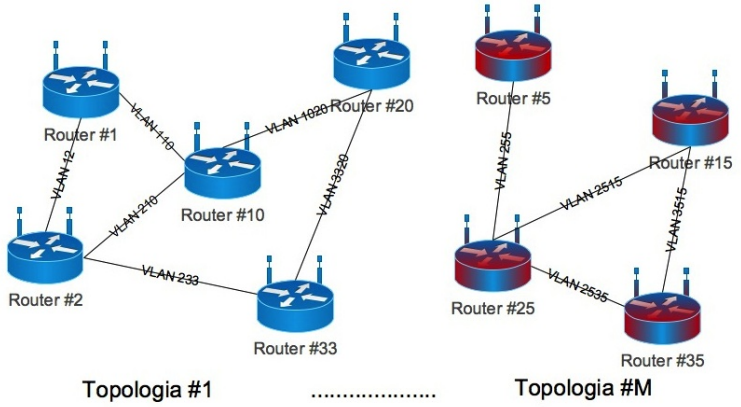


Figura 2.2: Partición lógica del LAR en M topologías

3 Estado del arte

En la actualidad existen diversas organizaciones que proveen acceso a testbeds, como Orbit[16], Fibre[4], NICTA[1], Geni[5], Nitos[13], Planetlab[17] y Fed4Fire[3]. Todas estas coinciden en el flujo de trabajo para realizar experimentos y arquitectura general que lo maneja, si bien no todas ellas están enfocadas a la misma tecnología. El hecho de que estas organizaciones coinciden en estos puntos demuestra cierta madurez en el ámbito.

La arquitectura de una testbed se comprende básicamente de los siguientes componentes:

- Recurso: es la entidad ofrecida por la testbed para realizar pruebas y experimentos. Puede ser un PC, máquina virtual, un enlace entre dos nodos de una red, una red, interfaces, aplicaciones, etc.
- Controlador de recurso: componente encargado de controlar al recurso. Esto incluye creación/destrucción, configuración. Generalmente es un programa que interactúa directamente con el recurso y responde a instrucciones del Aggregate Manager.
- Aggregate Manager (AM): este componente se encarga de la gestión, monitoreo y asignación de los recursos. Un usuario solicita recursos al AM donde los configura y asigna a una slice. Una slice puede ser vista como una entidad que agrupa recursos, es decir como un conjunto de éstos. Luego los experimentos son realizados sobre los recursos de una slice. Si la tesbed soporta federación, una slice puede abarcar varias testbeds.
- Interfaz de usuario: Se requiere una interfaz de usuario que se comunique con el AM y posibilite al usuario la solicitud y configuración inicial de los recursos. Opcionalmente se pueden ofrecer una interfaz de usuario para realizar los experimentos, de otra manera, éstos son llevados a cabo desde una interfaz de linea de comandos.

Los pasos para ejecutar un experimento se pueden resumir en los siguientes:

- Reserva de recursos: en esta etapa el experimentador solicita una serie de recursos que formarán parte del experimento. El Aggregate Manager es el responsable de asignar al usuario una slice con los recursos solicitados.
- Configuración inicial de recursos: los recursos solicitados son configurados por el usuario de acuerdo a sus necesidades. Esta etapa dependerá del tipo de recurso, pudiendo consistir en cargar la imagen de un sistema operativo, configuración de una interfaz de red, instalación de un paquete de software, etc.

- Ejecución del experimento: luego de configurados los recursos se procede a la ejecución del experimento. Existen básicamente dos enfoques. Uno en que existe una entidad principal, que recibe una descripcion de experimentos de un usuario y orquesta el experimento enviando las instrucciones a los distintos recursos. El otro, en que el usuario tiene completo acceso a todos sus recursos e interactúa con ellos directamente para instruirlos.

- Recolección de datos resultantes: esta etapa se puede realizar en simultáneo con la etapa anterior. A medida que se ejecutan las instrucciones del experimento, los datos relevantes para el usuario son almacenados de alguna forma. Puede ser mediante herramientas que automatizan este trabajo o de manera manual.

Existen tecnologías y herramientas de código abierto utilizados por las organizaciones mencionadas para llevar a cabo alguno de estos pasos. A continuación se describirán estas y otras que son pueden ser de utilidad para el proyecto.

3.1. Arquitectura general de una testbed

Como se dijo, existe cierta madurez en el diseno arquitectonico de las tesbeds. Esto ha dado lugar a la creación de un estandar que define las entidades y actores involucrados en una testbed, los componentes que los interrelacionan, tomando como base que las testbeds sean federadables, osea que puedan compartir funcioamiento entre ellas pero teniendo su propia lógica de negocio interna. El estandar es llamado Slice-based Federation Architecture y cuenta con varios refiniamientos a la fecha, siendo su última version la SFA 2.0.

El estandar ha sido adoptado por algunas testbeds y hay una iniciativa por parte del proyecto europeo Fed4FIRE[1] de ofrecer una implementación genérica que facilitaría su implantación sobre nuevas o ya existentes testbeds.

3.1.1. Slice-based Federation Architecture (SFA)

SFA es un estándar que define un conjunto mínimo de interfaces que permiten la federación de plataformas experimentales basadas en slices para su interoperabilidad.

Reconoce cuatro tipos de entidades. Por un lado se encuentran los propietarios de las plataformas, responsables del comportamiento y las políticas de uso de las mismas. Por otro lado se encuentran los operadores de las plataformas, cuya responsabilidad es gestionar la plataforma y garantizar su funcionamiento. Los experimentadores son los usuarios finales, que corren experimentos sobre las plataformas, y los identity providers garantizan la identidad y los roles de los distintos participantes.

El estandar permite a los propietarios declarar políticas de uso de sus plataformas, y proporciona mecanismos para hacer cumplir dichas políticas. Permite también a los operadores gestionar el equipamiento necesario, y a los experimentadores crear slices, asignarles recursos y utilizarlos para ejecutar experimentos.

[1] https://www.fed4fire.eu/

3.1.1.1. Definiciones

Los componentes son los bloques principales de la arquitectura. Un componente consiste en una serie de recursos, que pueden ser físicos o lógicos.

Componentes Los componentes se agrupan en agregados. Cada agregado se encuentra bajo el control de una Management Authority (MA), la cual es responsable de la estabilidad de los componentes, el cumplimiento de las políticas establecidas y la asignación de recursos. El control de cada agregado se realiza a través de un Aggregate Manager (AM), que exporta una interfaz accesible remotamente para la asignación de recursos a los usuarios. Si el agregado consiste en un único componente, puede denominarse Component Manager (CM).

Es posible compartir recursos de un componente entre múltiples usuarios. Esto puede lograrse mediante una virtualización del componente, o una partición del mismo. En cualquier caso, se dice que al usuario se le asigna un sliver del componente. Se deben proporcionar mecanismos que aseguren el correcto aislamiento entre los distintos slivers.

Slices Un slice es, desde el punto de vista del experimentador un conjunto de recursos computacionales y de red capaces de correr un experimento. Desde el punto de vista de un operador, es la abstracción principal que permite la auditoría del comportamiento del sistema, ya que los recursos son adquiridos y consumidos por los slices, y por lo tanto el comportamiento de un programa es rastreable a un slice.

Un slice se define como un conjunto de slivers, asociado a un conjunto de usuarios autorizados para su utilización. Un slice es registrado cuando se le asigna un nombre y un conjunto de usuarios. Luego es instanciado con un conjunto de componentes y recursos asignados. Por último es activado, y a partir de dicho momento ejecuta código en nombre de un usuario.

Los slices son registrados en el contexto de una Slice Authority (SA), que es la entidad responsable de uno o más slices. Estas entidades crean slices y permiten a los usuarios su utlización. Las Management Authorities tienen la autoridad para determinar qué Slice Authorities pueden crear slices en sus recursos. El registro de un slice se realiza una única vez, pero el conjunto de usuarios asociado puede variar a lo largo del tiempo.

Identificadores SFA define identificadores globales (GID) para todos los objetos que componen el sistema. Esto incluye componentes, slices, servicios, usuarios y authorities.

Los GIDs son la base de la seguridad y autenticación del sistema, ya que una entidad que posee un GID puede confirmar que el mismo ha sido emitido por una autoridad competente y no ha sido falsificado. Un GID es básicamente un certificado que contiene una clave pública, un Universally Unique Identifier (UUID) y un tiempo de vida (lifetime).

GID=(PublicKey,UUID,Lifetime)

El objeto identificado por el GID tiene en su poder la clave privada correspondiente a la PublicKey en el certificado. El UUID de un objeto es inmutable en el sentido que no cambia aunque cambie el par de claves, y absoluto, en el sentido que representa a un mismo objeto en todo el sistema.

RSpecs Las resource specifications (RSpecs) son especificaciones de los recursos que posee un componente y las condiciones de los mismos. SFA no define la forma de un RSpec, pero agrega los campos StartTime y Duration, que indican el período de tiempo (inicio y duración, respectivamente) por el cual se otorgan los recursos.

Ticket Un ticket consiste en un RSpec firmado por un Aggregate Manager, indicando la voluntad de otorgar los recursos a un usuario. Consiste en los siguientes valores

Ticket=(RSpec,GID,SeqNum)

donde GID representa la entidad a la cuál se le otorgarán los recursos, y SeqNum es un número de secuencia que impide la duplicación de tickets. El ticket es emitido por la autoridad correspondiente, y es luego reclamado por el propietario del mismo, asociándole efectivamente el conjunto de recursos solicitados. También puede dividirse, otorgando parte de los recursos a otra entidad.

Credenciales Las credenciales indican privilegios que poseen las entidades. Estos privilegios incluyen

- instantiate, que habilita a obtener tickets y slices.
- bind, que habilita a obtener tickets y prestar recursos a otro slice.
- control, que habilita a operar sobre slices existentes, pero no a soicitar nuevos tickets.
- info, que habilita operaciones de consulta de información sobre slices.
- operator, que habilita a invocar cualquier operación de la interfaz de gestión.

3.2. OpenFlow

Este estándar abierto permite a los experimentadores probar nuevas tecnologías utilizando la infraestructura operacional de una organización utilizando la noción de redes programables. OpenFlow es agregado como característica adicional a los switches ethernet y routers. Ofrece una manera estándar de modificar el comportamiento de estos dispositivos de manera uniforme sin exponer su funcionamiento interno.

Funciona delegando las decisiones de ruteo de los dispositivos a un controlador especial, el controlador OpenFlow, generalmente un servidor especializado, donde se puede definir el comportamiento del flujo de datos entre nodos.

Esta tecnología es utilizada en testbeds de redes cableadas. Tiene la ventaja de su flexibilidad ya que permite definir flujos de datos entre nodos, sobre potencialmente varios dispositivos de ruteo intermedios; y la capacidad de funcionar en infraestructura no especializada, es decir sobre redes utilizadas en el día a día por las organizaciones, lo que evita incurrir en los costos extras de despliegue de una testbed y permite experimentar sobre redes con tráfico real.

OpenFlow es soportado por cualquier dispositivo que soporte linux como sistema operativo, mediante la herramienta Open vSwitch[26]. En dispositivos switchs, que en su mayoria no corren linux, debe ser soportada de manera nativa por el fabricante. Sino, si el dispositivo permite cambio de firmware, pueden existir versiones no oficiales con soporte de esta tecnología.

En el caso de la infraestructura del LAR, contamos con algunso switchs que la soportan de manera oficial. Además contamos con routers domésticos ejecutando el sistema operativo OpenWRT, basado en linux, por lo que estos también soportan OpenFlow.

3.3. OFELIA Control Framework

Este framework provee un conjunto de herramientas de código abierto para administración de testbeds basadas en la tecnología OpenFlow. Fue desarrollado como proyecto cerrado entre 2010 y 2013 en el marco de una comisión europea de desarrollo tecnológico[14]. También se creó una testbed con el mismo nombre que interconecta varias islas de nodos en diferentes partes de europa.

OFELIA es autosuficiente en el sentido que controla el ciclo completo de experimentación y especifico para la experimentación de redes compuestas por hosts virtualizados sobre redes con OpenFlow.

Los principales componentes que incluye son una interfaz de usuario para reserva, aprovisionamiento y manejadores de recursos (que se denominan Aggregate Manager o AM) para creación y monitoreo de estos. Los recursos pueden ser máquinas virtuales o controladores OpenFlow. Cada isla de nodos, que en la práctica puede ser un servidor de virtualización en una universidad, es controlado por uno o varios AM que responden a solicitudes de un servidor central, a través del cual acceden los usuarios.

Los usuarios crean recursos en diferentes islas y los asignan a una slice, luego de configurarlos se conectan mediante SSH a estos y ejecutan los experimentos manualmente.

Este framework no provee ningún lenguaje de alto nivel que controle el experimento de manera centralizada ni medios para obtener datos resultantes.

3.4. OMF

El cOntrol and Management Framework es un conjunto de utilidades para ejecución de experimentos cuya principal característica es que provee un lenguaje de alto nivel para especificar experimentos lo que implica el control de forma centralizada de los diferentes recursos y sus acciones. Esta suite de utilidades incluye OML (OMF Monitoring Library) que permite tomar mediciones durante la ejecución de aplicaciones y la recolección de estas de forma centralizada.

La arquitectura de OMF consiste en tres planos lógicos: control, medición y gestión. El plano de control incluye las utilidades OMF, que permiten a los investigadores describir su experimento y a las entidades OMF, responsables de orquestar los recursos. El plano de medida corresponde a las utilidades responsables de tomar mediciones durante un experimento y recolectarlas. Por último, el plano de gestión, está constituido por las

entidades encargadas de configurar y aprovisionar los recursos. Éstos son provistos por testbeds y utilizados por experimentadores.

OMF comenzó en 2009 y a través de sus muchas versiones a sufrido cambios en su arquitectura fundamental. El mayor cambio fue de la versión 5.X a las 6.X, la última versión liberada al día de hoy es la 6.2.3. Si bien las versiones 5.X son las más utilizadas actualmente, se explicará el funcionamiento de la 6 porque ofrece un modelo simplificado, más flexible y algunas mejoras sobre las versiones 5. Además algunas organizaciones (por ejemplo Fibre, Fed4Fire) han anunciado la actualización a la última versión y se espera que las demás que lo utilizan hagan lo mismo. Por lo tanto la documentación siguiente refiere a las versiones 6.

3.4.1. Plano de control

El plano de control es el responsable del desarrollo y la ejecución de experimentos. El experimentador debe describir el experimento a realizar en un lenguaje específico llamado OMF Experiment Description Language (OEDL). El lenguaje permite la especificación detallada de los recursos necesarios para ejecutar un experimento, su configuración inicial y una serie de eventos antes los cuales realizar acciones. A esta descripción se la denomina Experiment Description (ED).

Una vez generada la descripción del experimento, la misma debe ser ejecutada por el Experiment Controller (EC), que es la entidad responsable de dirigir el experimento. Este controlador envía directivas a los Resource Controllers (RC), responsables de manejar cada recurso, de acuerdo a la descripción del experimento. Los Resource Controllers son los responsables de ejecutar las instrucciones del experimento en, o en nombre de, los recursos y reportar el estado de estos.

Los eventos definidos en el experimento son verificados por el EC, en cada evento el EC envía directivas a los RCs y estos envían sus respuestas. Tambíen los RCs pueden enviar información de manera asíncrona.

3.4.2. Plano de medición

El plano de medición es el responsable de recolectar y almacenar cualquier métrica que sea relevante en el contexto de un experimento particular. El mismo consiste en una librería llamada Measurement Library (OML) y un servidor llamado Measurement Collection Server (MCS). Esta librería puede ser utilizada independientemente de OMF pero éste provee una integración sencilla a través de su lenguaje de especificación de experimentos.

OML es integrada a una aplicación mediante su biblioteca, disponible para varios lenguajes, por lo que los puntos de medición son especificados por el programador. Las medidas son enviadas al servidor MCS, el cual se encarga de almacenarlas en una base de datos SQL para su posterior obtención y análisis por parte del experimentador. Las mediciones pueden enviarse en tiempo real o en modo batch, y pueden ser obtenidas mediante consultas SQL durante la ejecución del experimento, brindando al experimentador información valiosa antes de la finalización del mismo.

3.4.3. Plano de gestión

El plano de gestión es el responsable de gestionar la infraestructura necesaria para la plataforma de experimentación. Esto incluye la instalación, configuración y mantenimiento de los recursos y las redes dentro de la infraestructura.

Todos los servicios de gestión se combinan en un Aggregate Manager (AM). Este es el responsable de proveer mecanismos mediante los cuales descubrir y reservar los recursos disponibles en la testbed para los distintos experimentos. Cumple la función de nexo entre los usuarios y la infraestructura, permitiendo a los usuarios reservar recursos por un cierto tiempo, así como implementar controles de autenticación y autorización en la testbed. Permite además la federación de recursos siguiendo el estándar SFA[22].

3.4.4. Recursos

En OMF cualquier dispositivo de software o hardware puede ser un recurso. La arquitectura define un agente RC que recibe instrucciones y las ejecuta dependiendo del tipo de recurso. Los RC pueden controlar recursos de computo, caso en que ejecutarían en el recurso mismo, u otro tipo de recurso como interfaces de red, dispositivos accesibles a traves de la red, caso en que ejecutan en un nodo desde el cual tenga acceso. OMF incluye RCs para la interacción con distintos tipos de recursos como nodos, interfaces y aplicaciones, y es posible integrar otros tipos de recursos implementando los RC correspondientes.

3.4.5. Comunicación

La arquitectura de OMF require la existencia de dos tipos de redes a las cuales se conectan sus recursos. Por un lado se encuentran las redes experimentales, sobre las cuales circula el tráfico correspondiente al experimento. Por otro lado se encuentra la red de control y gestión, a través de la cual se comunican los componentes necesarios para la ejecución de un experimento. Esta comunicación se realiza a través del servidor publicación-suscripción que puede ser XMPP o AMQP, utilizando el protocolo Federated Resource Control Protocol (FRCP).

Cada recurso se identifica por un nombre y, al iniciar, crea un canal en el servidor mencionado identificado por dicho nombre. Al comenzar un experimento, EC realiza la comunicación inicial con los recursos utilizando los canles identificados por sus nombres, luego crea un canal nuevo específico del experimento en curso, a través del cual se comunicará en adelante, de manera bidireccional con los RCs.

En el protocolo existen tres tipos de entidades: requesters, components y observers. Los requesters envían mensajes a los componentes (o recursos), que pueden aceptar o no el mensaje, realizando las acciones solicitadas. Esto puede derivar en futuros mensajes enviados a los observers en consecuencia.

Si bien la especificación del protocolo no exige ningún sistema de mensajería en particular, se asume la utilización del patrón Publicador-Suscriptor, en el cual las entidades se suscriben a ciertos canales, y reciben los mensajes enviados a dicho canal por otras entidades.

El protocolo consiste en cinco tipos de mensajes: inform, configure, request, create y release.

- Un mensaje inform es enviado por un recurso, ya sea informando de alguna de sus propiedades (mensaje espontáneo) o como respuesta a una tarea previa que le fuera solicitada.
- Un mensaje configure es publicado en un canal solicitando a sus suscriptores que modifiquen cierta propiedad a un valor solicitado.
- Un mensaje request es publicado en un canal solicitando a sus suscriptores que publiquen información sobre ciertas propiedades.
- Un mensaje create es publicado en un canal solicitando a sus suscriptores la creación de un nuevo recurso. El recurso creador es denominado como padre, y el creado como hijo.
- Un mensaje release es publicado en un canal solicitando a sus suscriptores la liberación o terminación de un cierto recurso hijo.

3.5. Otros

Existen otras tecnologías y herramientas que no son específicas para testbeds, resuelven otros problemas al enfrentado en este proyecto o son menos utilizadas en la comunidad pero pueden ser utilizadas en un futuro en el LAR.

3.5.1. Nepi & nepi-ng

Nepi es una biblioteca Python que provee mecanismos para correr experimentos en varios tipos de testbeds, por ejemplo Planetlab, OMF 6.X, redes con soporte OpenFLow, simulador de redes ns3 o sobre hosts linux directamente si se tiene acceso ssh.

Nepi provee una manera simple de definir experimentos y la lógica para desplegarlos en una plataforma objetivo. También provee mecanismos para controlar los recursos utilizados en el experimento durante su ejecución y la recolección de los resultados a un repositorio central.

El proyecto fue desarrollado por un grupo de investigadores en el instituto de investigación Inria[8] en 2011, aunque se han liberado versiones hasta el año 2016. Cuenta con buena documentación y madurez comparable con OMF y OFELIA.

Nepi-ng es la nueva version de Nepi, que a la fecha solo soporta orquestar experimentos sobre hosts accesibles a través de ssh.

3.5.2. Ansible, Puppet, Chef y Salt

Estas herramientas cuya concepción es la gestión de configuración y orquestación de servidores de forma centralizada con el objetivo de facilitar implantación de configuraciones, paquetes de software y servicios a un administrador de sistemas. Pueden ser

utilizadas en el ámbito de este proyecto para la configuración de recursos de manera sencilla, por ejemplo cargar cierta imagen de un sistema operativo con los paquetes de software especificados por el experimentador. Todas coinciden en que proveen un lenguaje común de especificación que es ejecutado por un servidor especial con el objetivo de hacer cumplir ciertas instrucciones a un conjunto de nodos.

3.5.3. Hipervisor LXD

LXC (Linux Containers) es una tecnología de virtualización de sistemas Linux a nivel de sistema operativo. Utiliza la funcionalidad cgroups del kernel de Linux para proveer mecanismos de aislamiento que permitan que distintos sistemas operativos tengan sus propios recursos (CPU, memoria, redes, dispositivos de entrada/salida) sin interferir unos con otros, así como los namespaces de Linux para la virtualización de recursos.

Con estos mecanismos, LXC provee un entorno virtual para los sistemas operativos sin necesidad de crear máquinas virtuales, disminuyendo así el consumo de recursos y el espacio necesario para los sistemas virtuales.

LXD es un hipervisor desarrollado por Canonical[2] que facilita la creación y gestión de Containers LXC, proporcionando una API y una interfaz de línea de comandos para la interacción con los procesos virtuales. Permite limitar los recursos asignados a cada container, migrar sistemas en caliente entre distintos hipervisores y anidar containers dentro de otros containers, permitiendo así la simulación de ambientes de laboratorio realistas.

Utilizar containers en lugar de máquinas virtuales convencionales disminuye sustancialmente los recursos necesarios, así como los tiempos necesarios para su instalación. Como ejemplo, un container recién instalado con Ubuntu 16.04 consume unos 15MB de RAM, y reiniciarlo tarda aproximadamente dos segundos.

3.6. Tecnologías por Organización

En la práctica, las organizaciones investigadas utilizan algunas de las tecnologías mencionadas en esta sección. En el Cuadro 3.1 se encuentra un resumen con el tipo de infraestructura que cuentan y la tecnología que utilizan.

[2] https://www.canonical.com/

Organización	**Tipos de redes**	**Software**
Orbit	Wireless	OMF
Fibre	WiFi, Wired	OMF, OCF (Ofelia), OpenFlow
Nicta	Wireless	OMF
GENI	Wired	OMF, OpenFlow, Ansible
Nitos	Wired, wireless	OMF
Planetlab	Wireless	Openflow, Nepi
Fed4Fire	Wired, wireless	OMF, Nepi

Cuadro 3.1: Resumen de tecnologías por organización

4 LAR

Los recursos disponibles en el LAR que pueden formar parte de la testbed son tanto virtuales como físicos. A grandes rasgos distinguimos dos tipos básicos para el caso de ejecución de experimentos: hosts y redes. Como recursos hosts el LAR cuenta con máquinas virtuales, contenedores linux y routers domésticos. El recurso red, puede abarcar varios dispositivos físicos y virtuales, estos son: switches, interfaces de red físicas e interfaces de redes virtuales.

Puntualmente los switches son Cisco modelo SF200-48P. Utilizan VLANs para generar distintas redes virtuales. Los routers domésticos son Linksys WRT54G2 y Tp-Link TL-WR740N con el sistema operativo OpenWRT. Como virtualizadores contamos con XEN y KVM para VMs y LXD[9] para contenedores linux.

4.1. Observaciones

Las VLANs de los switches deben ser configuradas dinámicamente por el gestor cuando un usuario crea una red involucrando hosts virtuales y físicos. El modelo de switch mencionado solo provee una interfaz web a través de la cual se realiza toda la configuración. El hecho de no proveer una API explícita causa que la interacción con el switch sea lenta y propensa a errores.

Los routers Linksys y Tp-Link son dispositivos con pocos recursos de hardware, siendo la memoria RAM y almacenamiento interno las limitaciones principales. La realización de experimentos puede necesitar de software extra ejecutando en los recursos por lo que es un problema no menor analizar la posibilidad de ejecución de estas en los routers teniendo en cuenta, además, que éstos utilizan una arquitectura no convencional y un sistema operativo mucho más básico que los linux de escritorio, por lo que estas deben ser compiladas expresamente para la arquitectura de estos dispositivos.

Los virtualizadores XEN y KVM son ampliamente conocidos hipervisores de maquinas virtuales tradicionales, mientras que LXD es de desarrollo reciente y funciona sobre contenedores linux. Para todos ellos existen APIs con una interfaz sencilla de gestion de servicios virtualizados.

Por último, se decidió utilizar como base el sistema operativo Ubuntu Server 16.04 LTS para componentes servicios virtualizados y servidores. Esta distribución de Linux tiene la ventaja de ser conocida, tener una gran cantidad de software disponible para su instalación y contar con soporte oficial hasta el año 2021.

4.2. Solución propuesta

El objetivo del grupo MINA no es que la infraestructura provea todas las capacidades vistas en la sección anterior para ejecutar experimentos, sino que facilite su ejecución lo más posible. El proyecto se enfocó en implementar algunas de estas funcionalidades y analizar su extensión a futuro.

En primer lugar se atacó el problema de la ejecución de experimentos. Responder preguntas como ¿Qué es un experimento? ¿Que se desea obtener de estos? ¿Como se puede especificar? hasta las flexibilidades y capacidad de adaptación e implantación en el LAR que las herramientas existentes proveen.

Luego el problema de automatización de las operaciones comunes sobre la infraestructura. Por ejemplo la configuración de topologías, que implica creación de vlans potencialmente involucrando varios switchs. Creación y destrucción de servicios virtualizados por ejemplo máquinas virtuales o contenedores linux. Y por último monitoreo de todos los recursos involucrados.

Estas dos problemáticas que distinguimos son en principio independientes, sin embargo la solución final debe relacionarlas ya que las dos son indispensables para realizar el ciclo completo de experimentación en el LAR. Las diversas tecnologías que fueron mencionadas en el estado del arte ofrecen una solución para la primera, la segunda o ambas cuestiones.

5 Diseño de la plataforma

Según el diccionario de la Real Academia Española, experimentar significa "probar y examinar prácticamente la virtud y propiedades de algo". En el ámbito que estamos trabajando podemos caracterizar un experimento como el procedimiento por el cual se prueba software (ejemplo aplicaciones y protocolos) en una infraestructura de redes de computadoras. La virtudes y propiedades se observan en comparación con productos de similares características o distintas variantes del mismo. Para realizar dicha comparación un experimentador debe poder medir ciertos parámetros de manera precisa.

En resumen un experimento en una testbed es software ejecutando en los nodos de una red de computadoras según los criterios de un experimentador. Dicha ejecución genera medidas de parámetros que son de interés para el susodicho. El único requerimiento fundamental para realizar un experimento es un conjunto de hosts y una red física o virtual que los interconecta. Vale la pena recalcar que en realidad son dos redes: una de gestion, utilizada para orquestar el experimento y otra para realizar la experimentación misma. Idealmente el experimento no utiliza la red de gestión. Ésta debe ser conocida por el experimentador y debe ser lo menos intrusiva posible en los recursos involucrados.

Para poder realizar un experimento repetidas veces (con variantes o no) se requiere que el software ejecutado en los hosts sea orquestado de manera precisa. Realizar esto en un ambiente distribuido es más complejo en la medida que se agranda el experimento, en el sentido de cantidad de hosts, software involucrado y la cantidad de estados por los que debe pasar un componente de software (cuya transición es activada por el experimentador). Un experimentador podría orquestar sus hosts manualmente, o utilizar algún mecanismo centralizado lo que implica una solución mucho más escalable. Este mecanismo podría hacerse cargo de la recolección de medidas lo que facilita aún más la tarea del usuario.

Asimismo, la tarea de configuración y asignación de recursos a experimentadores es indispensable en una testbed multiusuario. Los experimentadores deben poder solicitar y configurar recursos fácilmente y los administradores de la testbed deben poder monitorear y auditar el uso de estos. Un componente que realice estas acciones es prescindible para la realización de experimentos pero imprescindible si se quiere optimizar y facilitar la utilización de la infraestructura.

5.1. Evaluando las posibilidades

5.1.1. Ejecución de experimentos

Las herramientas existentes como OMF y Nepi proveen una buena solución, con madurez suficiente como para confiar en ellas en nuevas testbeds. Llevar a cabo una im-

plementación desde cero llevaría demasiado esfuerzo y un desfase con el objetivo del proyecto.

La herramienta Nepi permite ejecutar experimentos en diferentes testbeds utilizando el lenguaje Python. Soporta diversas variantes de testbeds desde las que cuentan con una herramienta o framework para experimentación hasta las compuestas por hosts únicamente. La razón por la que no la tuvimos en cuenta fue la tardanza con que nos encontramos con ella en el transcurso del proyecto. Es una buena solución si se cuenta con una infraestructura que gestiona redes y hosts linux. En ese caso tiene la ventaja frente a OMF en la ejecución de experimentos ya que no requiere un agente ejecutando en el recurso (o por el recurso) por lo que representa una solución mucho menos intrusiva y fácil de implantar. Además provee un mecanismo de recolección de datos de un experimento, que no es mediante una biblioteca con la que interactúan los programas, aunque no es tan potente como OML.

Si bien Nepi parece una buena solución no llegamos a evaluar el poder de expresión y ejecución de experimentos. Aunque, podemos suponer que es al menos tan expresivo como OMF ya que es una de las plataformas de experimentación soportadas.

Las herramientas de gestión de servidores como Ansible y similares se descartaron como ejecutores de experimentos ya que no ofrecen más capacidad que OMF o Nepi y por no ser herramientas específicas para este ámbito se estaría utilizando una solución más compleja de lo que el problema requiere.

Por otro lado pueden ser de utilidad para configurar dinámicamente los hosts de la testbed, sin embargo el proyecto no incluyó esto en el alcance.

Finalmente se optó por OMF para la ejecución de experimentos en el LAR. Este framework es el mas flexible de los disponibles en términos de adaptabilidad a los recursos del LAR. Además provee un lenguaje de especificación de experimentos que tiene una curva de aprendizaje corta y provee una capacidad de definición aceptable. Incluye una biblioteca de recolección centralizada de mediciones que facilita la tarea de experimentadores, al no tener que realizar esta tarea ellos mismos. Por último este framework ha sido utilizado en gran medida en la comunidad y ha sido implantado en diversas testbeds.

OMF incluye por defecto controladores para recursos como aplicaciones, interfaces de red ethernet y wlan, máquinas virtuales y físicas. Es fácilmente extensible, por lo que se pueden desarrollar controladores para diversos tipos de recursos utilizando una API provista por el framework. Ha sido utilizado en el ámbito durante años y su desarrollo está activo al dia de hoy. Si bien la última versión contiene bugs y errores es esperable que estos sean corregidos eventualmente.

Las herramientas que provee este framework pueden manejar casi todo el ciclo de vida del experimento. Desde creación, configuración y monitoreo de recursos hasta la experimentación y recolección de datos. Sin embargo no maneja la reserva y asignación de recursos a experimentadores.

Como desventaja principal, OMF sufre de carencias en su documentación en todo sentido: manuales de uso, instalación e implementación. Esto es aún más grave en la última versión que como se dijo sufrió cambios arquitectónicos.

Otra desventaja es la estabilidad de algunos de sus componentes, especialmente los que aparecieron luego de la reestructuración. Algunos de estos son completamente inuti-

lizables por falta de documentación y por la poca legibilidad de su código fuente. Afortunadamente los componentes que sufren de estas falencias no son imprescindibles para el funcionamiento de la testbed ni de la ejecución de experimentos.

Pese a estas desventajas, la última versión del framework (y las versiones superiores a 6.0) provee un conjunto básico de componentes con el que se pueden realizar experimentos que funcionan adecuadamente. Las versiones anteriores mantenían un alto acoplamiento entre sus partes y poca capacidad de extensión, por lo que su utilización hubiera significado ganancias por el lado de la estabilidad pero perdidas por el lado de la adaptación al LAR. Preferimos utilizar las herramientas básicas que nos permitan realizar experimentos e implementar desde cero, con tecnologías nuevas y más afines a nosotros los componentes faltantes. Esto se traduce a la integración de OMF como ejecutor de experimentos con un módulo que realiza la gestión, monitoreo, reserva y asignación de recursos.

5.1.2. Gestión de recursos

El framework OFELIA provee las herramientas necesarias para realizar una gestión adecuada de los recursos de una testbed consistente de máquinas virtuales y switches con soporte OpenFlow. Sin embargo fue descartado por desconocer en el momento de la decisión la existencia de soporte OpenFlow por parte de los switchs. El soporte de OpenFlow en los routers con OpenWRT no fue relevante para esta decisión ya que estos dispositivos no forman parte del recurso red disponible para el experimentador, sino que son nodos cuyo objetivo es ser utilizados por el experimentador a su parecer: como hosts o como routers, pero cuya configuración es un fin mismo del experimento.

La gestión de recursos de una testbed es un problema de características particulares. Es difícil idear (y no encontramos) una herramienta general que resuelva este problema, que sea extensible y adaptable a recursos específicos. Por lo tanto optamos por realizar este componente desde cero específicamente para el manejo de recursos del LAR. Realizarlo de manera general trasciende el objetivo del proyecto.

5.1.3. Virtualización

De los tres hipervisores mencionados, dos funcionan utilizando maquinas virtuales y uno sobre contenedores linux.

Las maquinas virtuales ofrecen la capacidad de virtualizar prácticamente cualquier sistema operativo y cualquier arquitectura, mientras que los contenedores linux solo ofrecen diferentes distribuciones de Linux.

En el contexto en que estamos trabajando, Linux es la opción que mayormente satisface la necesidad de los usuarios experimentadores, ya que es el sistema operativo predilecto para el estudio en redes de computadoras e infraestructuras de red. De todas maneras se entiende necesario que se pueda integrar otros sistemas operativos a un experimento. Por simplicidad, en la primera versión de la solución, se decidió utilizar virtualización unicamente sobre Linux.

Tomando en cuenta esta decisión, los contenedores poseen toda la funcionalidad de una máquina Linux convencional. Como el LAR está enfocado principalmente a la ejecución

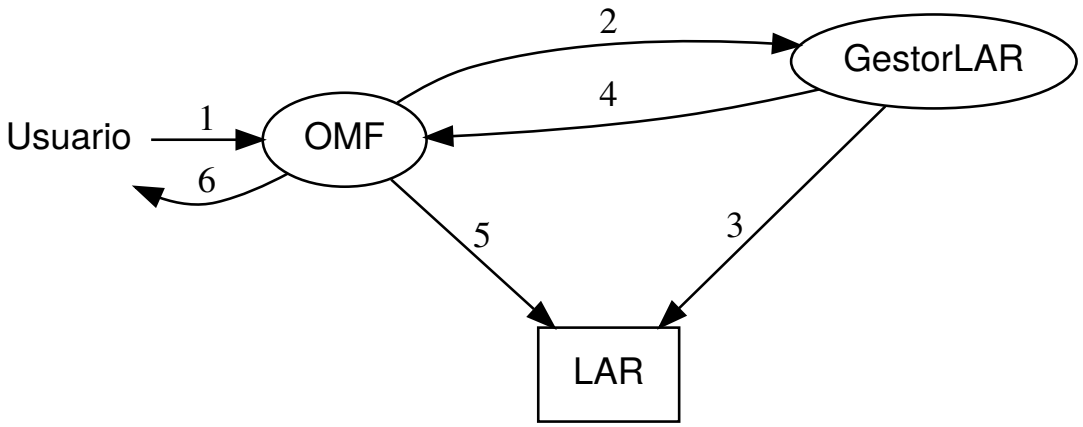


Figura 5.1: Diagrama de componentes globales.

de experimentos que requiere constante creación y destrucción de topologias, consideramos que la utilización de máquians virtuales, impactan negativamente en el desempeño general del sistema, ya que sus tiempos de creación/destrucción asi como el overhead en el consumo de recursos de hardware es demaciado alto comparado con contenedores, por lo que no se justifica su utilización.

5.2. Componentes del gestor

Continuando con la perspectiva de los componentes involucrados que venimos discutiendo, la ejecución de un experimento en la solución propuesta involucra, a grandes rasgos, los pasos descritos en la Figura 5.1.

Se distinguen cuatro entidades: un usuario (experimentador); el LAR, compuesto por host físicos, hosts virtuales y switchs que los interconectan; OMF y el GestorLAR. Los últimos dos son los componentes de software que distinguimos en las secciones anteriores, el primero encargado de la experimentación y el segundo de la gestión de los recursos.

Los pasos 1 y 6 indican la entrada y salida respectivamente del experimento. El usuario ingresa un experimento en el primer paso y obtiene como salida información de su ejecución.

Los pasos 2, 3, y 4 corresponden a la reserva y configuración de recursos. Al indicar el experimento, el usuario incluye también una topología especificando los recursos involucrados y su interconexión. La topología es tomada por OMF y enviada al GestorLAR donde se crean, configuran y asignan los recursos al usuario. Si todo sale bien, el paso 4 indica a OMF que la topología se configuró satisfactoriamente.

Por último el paso 5 representa la ejecución del experimento. Dicho paso involucra pasaje de información desde y hacia el LAR conteniendo instrucciones del experimento, respuestas a estas instrucciones y datos de mediciones recolectadas.

A continuación vamos a discutir los componentes de software. OMF es el más complejo de los dos por la cantidad de módulos que involucra y la distribución de ellos sobre el

LAR. Describiremos sólo su funcionamiento e intenciones detrás de su diseño y realizamos comentarios sobre su implantación en la infraestructura. El GestorLAR es mucho más sencillo pero ofreceremos una discusión mucho más detallada al ser implementado y diseñado por nosotros.

5.2.1. OMF

Los subcomponentes de OMF encontrados en esta sección de la documentación son los utilizados por nosotros para implantar una testbed sobre el LAR. Puede haber diferencias en la implantación realizada con otras testbeds en términos de cuáles y cómo se utilizan estos.

5.2.1.1. Arquitectura

Como se dijo anteriormente, la disposición más básica de OMF permite únicamente configurar y orquestar recursos de manera centralizada. Para lograr esto se utiliza un controlador de experimento, que llamaremos omf_ec y uno o varios controladores de recursos, llamados omf_rc. Cada instancia ejecutando de omf_rc se identifica por un nombre. Estos dos controladores se comunican bidireccionalmente a través de un servidor publicación-suscripción utilizando el protocolo FRCP.

En esta forma más básica OMF permite ejecutar experimentos a un experimentador sobre recursos predefinidos, osea nombres de instancias de omf_rc conocidas por el experimentador.

Sobre esta arquitectura podemos agregar la funcionalidad de obtener medidas de manera centralizada. Para esto vamos a necesitar un servidor OML, que recolecta las medidas en una base de datos y la creación de programas que utilicen la biblioteca de OML para recolectar medidas en puntos definidos por el desarrollador. La integración de OML con los componentes omf_ec y omf_rc es mediante su archivo de configuración respectivo, donde se especifica la dirección del servidor OML.

El controlador omf_ec provee una interfaz de línea de comandos para ejecutarse. Podemos facilitar su ejecución al experimentador ofreciendo una interfaz web donde se puedan definir, ejecutar y documentar experimentos. Esto lo logramos integrando LabWiki a la arquitectura actual. Además, como múltiples experimentadores pueden potencialmente utilizar la plataforma concurrentemente se cuenta con un planificador de experimentos, llamado job_service.

LabWiki interactúa con los otros sistemas a través de sus plugins. Utilizamos dos: uno para agendar y monitorizar un experimento a través de job_service y otro para realizar la configuración de la topología interactuando con el GestorLAR.

Para tener una idea global de la interacción entre componentes, en la figura 5.2 se describe la interacción entre ellos y el protocolo o medio de comunicación.

5.2.1.2. Objetivos del diseño

Uno de los principales objetivos en el diseño de OMF en sus versiones 6.X fue la modularidad. La arquitectura completa está compuesta por componentes con baja acoplacion

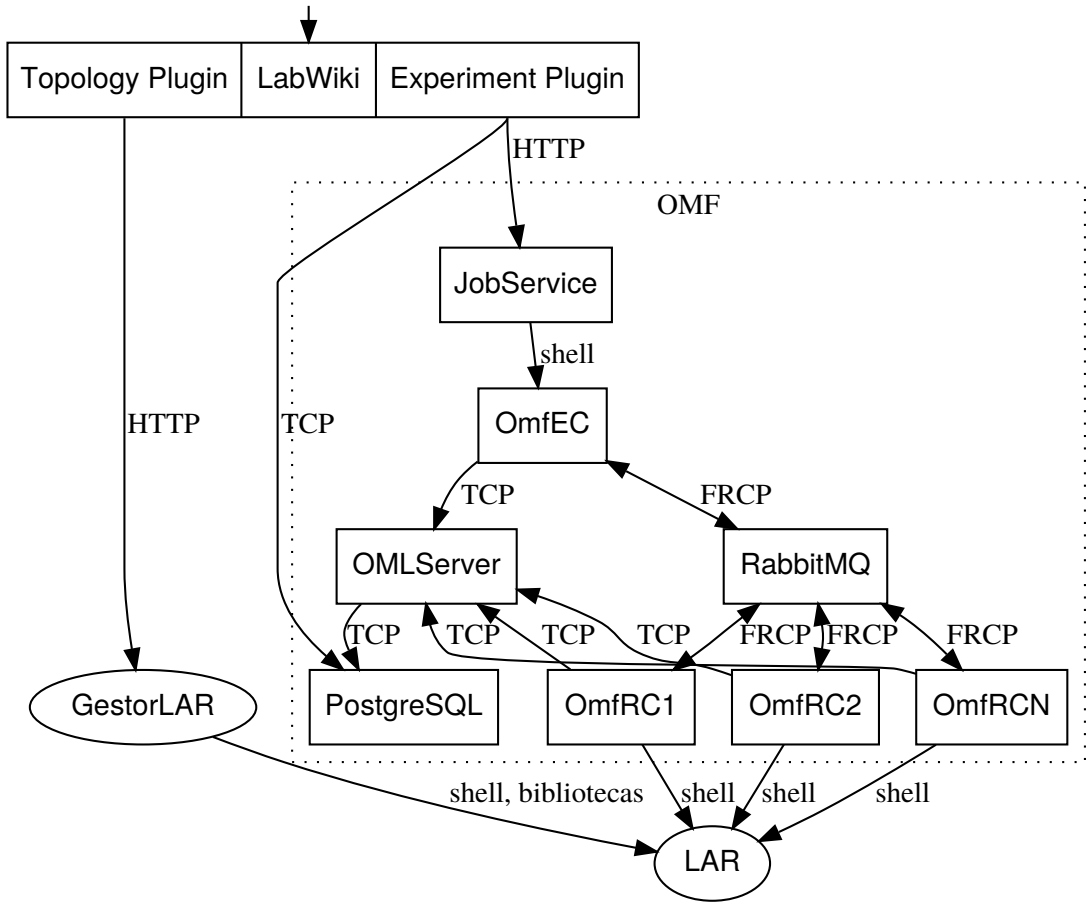


Figura 5.2: Diagrama de interacción de componentes de OMF

entre ellos donde cada uno resuelve un problema puntual.

Otro de los objetivos fue la fácil extensión de sus recursos a dispositivos no soportados en su distribución estándar. Los recursos OMF se pueden ver como un conjunto de propiedades con diferentes permisos sobre ellas: lectura o lectura-escritura. Las propiedades son accesibles al experimentador y este puede leerlas y configurarlas durante la ejecución de un experimento. Para definir un tipo de recurso basta definir las propiedades que este posee y el comportamiento ante las operaciones de lectura y escritura sobre cada una de ellas.

5.2.1.3. Implantación

Si bien la arquitectura modular de OMF es un aspecto favorable en su diseño, se volvió una desventaja al momento de instalar y configurar los componentes principalmente por

el ya mencionado defecto en la documentación.

LabWiki, job_service y OMF EC Los servicios LabWiki y job_service utilizan las mismas tecnologías subyacentes y muestra una implementación similar. Ambos están desarrollados en Ruby y utilizan rack[19] como framework de servidor web. Su instalación y ejecución es directa aunque ambos mostraron dificultades al realizar una instalación completamente funcional por sus dependencias y complejidad de configuración.

LabWiki depende de sus plugins para ser utilizable. El primero, que realiza la interacción con la parte de experimentación de OMF, ya estaba implementado y funcionando correctamente por lo que solo tuvimos que integrarlo. El otro, que provee la capacidad de crear y configurar topologías tuvo que ser adaptado, desde una implementación inconclusa, para que interactúe con nuestra implementación del GestorLAR. LabWiki por defecto no realiza autenticación de usuarios, por lo que cualquier usuario con acceso a la web puede acceder a todos sus servicios. Se debió implementar un módulo para la autenticación con objetivo básico de distinguir usuarios y mantener los datos de cada uno por separado.

El servicio job_service interactúa con omf_ec directamente a través de una shell, por lo que éstos deben ser instalados en el mismo host. La instalación de ambos se puede realizar fácilmente utilizando el gestor de paquetes de Ruby, lenguaje en el que están implementados.

OML El servidor OML y su librería para integrar el servicio a las aplicaciones está disponible en un repositorio para ubuntu hasta la versión 14.04. Para instalarlos se necesitó agregar la URL a fuentes de software de los hosts ubuntu 16.04, tanto en el servidor como en los recursos. Además se debió instalar una base de datos PostgreSQL junto con el servidor OML para el almacenamiento de las métricas.

Message Queue Como servidor publicación-suscripción OMF soporta dos tipos: XMPP y AMQP, en particular la documentación recomienda las implementaciones de OpenFire y RabbitMQ respectivamente. En la versión 6 de OMF se recomienda RabbitMQ por su fácil instalación y configuración. En ubuntu 16.04 se encuentra disponible a través de su gestor de paquetes de software.

OMF RC Por último la implantación de omf_rc fue sencilla en los host virtualizados en los que utilizamos ubuntu 16.04, ya que es provisto como paquete de Ruby. En los routers con OpenWRT, se debió utilizar el SDK de OpenWRT para generar un paquete que funcione para dicho sistema operativo y las arquitecturas de los routers. Este SDK provee herramientas para compilación cruzada[24] y empaquetado de software para su instalación con el gestor de paquetes de OpenWRT. Si bien omf_rc está escrito en ruby, un lenguaje interpretado, tiene dependencias (librerías en ruby y nativas) que deben ser compiladas para la arquitectura objetivo, por ejemplo el mismo intérprete de ruby. Para cada librería a ser compilada debe proveerse un Makefile con las instrucciones y otras dependencias necesarias para la compilación, siguiendo ciertas reglas del SDK. Generar

Ejecución	Mínimo (KB)	Máximo (KB)	Promedio (KB)
Idle	31844	-	-
1	31844	44844	40883
2	41068	45836	43129
3	41860	48888	44465
4	40648	57684	47004
5	38652	67804	48951

Cuadro 5.1: Consumo de memoria RAM de omf_rc en OpenWRT. Se ejecutó cinco veces consecutivas el mismo experimento para mostrar el consumo de memoria de una instancia ejecutando de omf_rc en un router con sistema operativo OpenWRT y cómo evoluciona con el paso del tiempo. El experimento consta en utilizar la herramienta ping para mostrar información de RTT y pérdida de paquetes en la comunicación con un host.

una versión funcional de omf_rc en OpenWRT fue trabajoso y por momentos exasperante .

Debido a los bajos recursos de hardware de estos enrutadores, omf_rc debió ser instalado en una sección del sistema de archivos alojado en memoria, por lo que debe ser instalado cada vez que se reinicia el dispositivo. Para automatizar esto se agrego un script ejecutado al inicio del sistema que lo descarga desde un servidor web y realiza la instalación.

Las pruebas realizadas indican, que una vez iniciado, el agente omf_rc consume gran parte de la memoria RAM de estos dispositivos pudiendo dejarlos sin memoria dada la gran oscilación en el consumo de ésta durante experimentos incluso básicos. En el Cuadro 5.1 se detalla este comportamiento. La implementación no está pensada para este tipo de dispositivos y por lo tanto se toma especial cuidado por el consumo de memoria.

5.2.2. GestorLAR

Este componente se encarga de la gestión y configuración de los recursos del LAR. El framework OMF provee un servicio que realiza estas acciones, llamado OMF AM (Aggregate Manager), pero su implementación es defectuosa y su documentación muy escasa. Para el diseño del GestorLAR tuvimos en cuenta la implementación de OMF aunque no estrictamente, por eso vamos a describir el diseño del segundo, para luego dar una discusión de nuestra implementación mencionando las similitudes con OMF AM.

El OMF AM está diseñado de manera que interactúa con los recursos de la tesbed directamente, mediante el protocolo FRCP, comunicandose a traves del servidor publicación-suscripción. Todos los recursos se suscriben a un canal del servidor publicación-suscripción creado por el AM para interactuar con ellos.

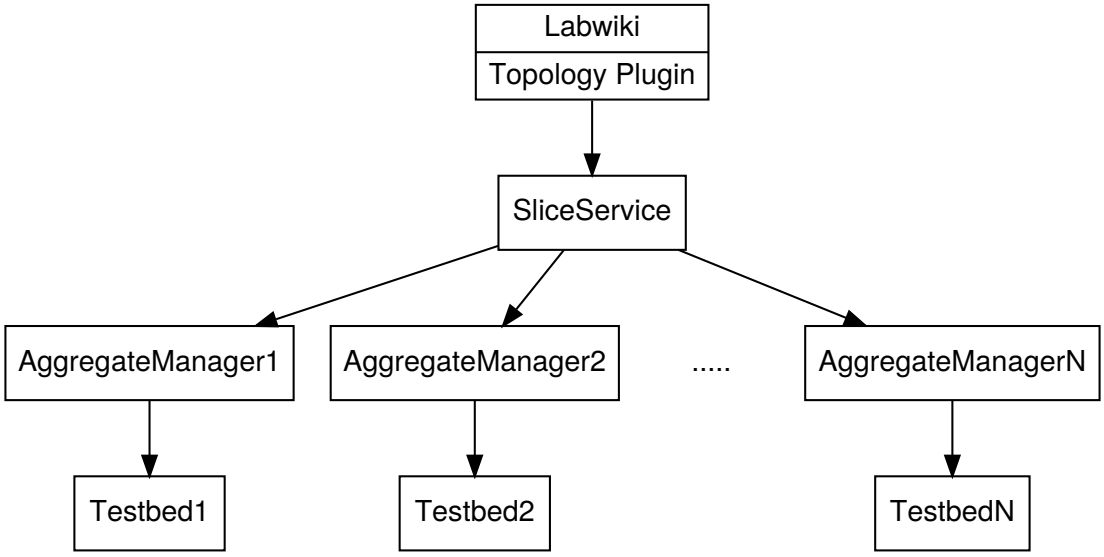


Figura 5.3: Esquema de interacción entre LabWiki, Slice Service y varios Aggregate Managers

Los recursos envían un mensaje create, al iniciar el agente omf_rc, y mensajes inform tanto sincrónica como asincrónicamente informando de su estado. Luego el AM puede configurarlos y acceder a las propiedades con mensajes configure y request. De esta manera puede descubrir, configurar y monitorear recursos de forma sencilla, con una interfase unificada y sin costos adicionales ya que utiliza la misma API que para experimentación.

Además OMF cuenta con otro componente, OMF Slice Service, quien se comunica con el topology_plugin de LabWiki para configurar los recursos a través de potencialmente varias testbeds manejadas por OMF AMs. De esta forma se conseguiría la federación de testbeds con el objetivo de integrar recursos gestionados por diferentes organizaciones en un mismo experimento. La Figura 5.3 representa gráficamente las dependencias entre componentes desde LabWiki hasta los recursos de la tesbed. Vale la pena resaltar que Slice Service provee solo la gestión centralizada de los recursos federados, las interconexiones entre testbeds necesarias para que exista conectividad entre recursos debe ser manejada y gestionada entre ellas. Lamentablemente la implementación actual está lejos de lograr su objetivo por razones idénticas al OMF AM.

La implementación del GestorLAR sigue un diseño general similar. Cuenta de dos módulos lógicamente distintos, aunque funcionan juntos, bajo el mismo servidor. Estos módulos son LAR Aggregate Manager y LAR Slice Service y su funcionamiento es igual a sus homónimos en OMF. A diferencia de estos, el GestorLAR no interactúa con los recursos a través del protocolo FRCP sino que de diferentes maneras según el tipo de recurso. La razón de esto fue la inexistencia de una biblioteca que implementa este protocolo para la tecnología utilizada, Python. De todas formas la implementación sigue un esquema de operaciones igual a FRCP por lo que de existir tal biblioteca, se podría

integrar.

Las próximas secciones describen el diseño de los dos módulos implementados por separado, comenzando por LAR Slice Service, que es el punto de entrada al GestorLAR en el ciclo de vida de un experimento.

5.2.2.1. LAR Slice Service

Este módulo se puede describir como un middleware entre LabWiki y los AMs que gestionan cada testbed. También se encarga de la gestión de slices y usuarios que las utilizan. Su función principal es recibir una descripción de topología desde LabWiki y delegar la configuración de los recursos a cada AM. La interacción entre LabWiki (a través del topology_plugin) y Slice Service se realiza mediante una API HTTP Rest que el último provee.

Como se vio, LabWiki es el punto de entrada de los experimentadores a la plataforma. Interactúa con Slice Service básicamente en cuatro situaciones:

1. Periódicamente, solicitando los AMs que son soportados por Slice Service.
2. Cuando un experimentador se autentica en LabWiki. En este caso se envían las credenciales a Slice Service para que éste las utilice a su vez para solicitar recursos. Además se solicita información sobre dicho experimentador.
3. Cuando se solicita una slice por parte de un experimentador.
4. Cuando se quiere configurar una topología y asociarla a un slice.

El primer caso es simple. Slice Service mantiene los AMs sobre los cuales se pueden solicitar recursos, por lo tanto LabWIki debe poder mostrarlos como opción a los experimentadores. Es decir, al momento de crear un recurso, se debe mostrar sobre qué AM y por lo tanto sobre que testbed, se debe crear.

El segundo caso permite a Slice Service llevar registro de los usuarios que utilizan el servicio ya que mantiene los usuarios y slices asociados a ellos. Este caso tiene dos fines diferentes. Por un lado Slice Service mantiene almacenado información de los usuarios que han utilizado el sistema. Dicha información es utilizada por LabWiki para renderizar informacion dependiendo del usuario. Por otro lado, el envío de las credenciales, no necesariamente las de ingreso a LabWiki, tienen el fin de proveer autorización a los experimentadores a las diferentes testbeds a las que Slice Service tiene acceso, a través de sus respectivos AMs. Las credenciales deben ser verificadas contra los AMs al momento de requerir recursos para un experimentador, ya que éste debe tener permisos sobre cada testbed para poder utilizarla. Generalmente las credenciales es uno o varios tickets provistos por los administradores de las testbeds a los usuarios experimentadores.

El tercer caso se da previo a que un experimentador deseé implantar una topología, osea configurar recursos y redes para que su topología sea plasmada en la realidad. Para esto debe crear o reutilizar una slice a la cual se asociará todos los recursos y por la cual se identificará la topología.

El último caso se da al momento en que se envía la descripción de una topología para que Slice Service la configure.

Detalles de la implementación Nuestra implementación realiza los cuatro pasos descritos en la sección anterior aunque simplificando el segundo. La razón de esto son los tiempos que lleva integrar aspectos de seguridad en la solución. La plataforma experimental diseñada, tiene como único punto de entrada la interfaz web de LabWiki, para simplificar, tomamos el autenticarse en ella como la única línea de confianza. A su vez, sólo soportamos el AM del LAR, esto implica que en el paso dos no es necesario enviar las credenciales de un usuario sobre un AM, basta con su nombre de usuario para identificarlo en el sistema.

El propósito de Slice Service es primordialmente el de coordinar servicios LabWiki y el AM. Algunas de estas operaciones pueden tomar mucho tiempo, como la configuración de una topología entera. Para esto, Slice Service implementa un mecanismo de promises. Ante una petición de larga duración, el servicio retorna inmediatamente una promsie, que puede ser consultada periodicamente para ver el progreso de la tarea y su resultado una vez que finalice. Al crearse una promise, el servicio retorna una URL que puede ser consultada. Una promise cuya tarea asociada no está finalizada retornará status HTTP 504 y una vez que finalice retorna HTTP 200.

Para implementar estas operaciones asíncronas, se utilizó el framework Celery[7], que permite mantener una cola de trabajos asincrónos basado en pasaje de mensajes de manera distribuida. Es decir, la configuración de topologías se lleva a cabo por procesos esclavos que reciben trabajos desde Slice Service a través de una cola de mensajes y mantienen el estado de la tarea en tiempo real sobre la base de datos a la que tanto los procesos esclavos como Slice Service tienen acceso.

Los trabajos recibidos por los proceso esclavos son los que interactúan con el LAR AM. La interacción se realiza a través de una pequeña API Python que provee una interfaz que debe ser implementada para cada AM. Dicha API provee las funciones básicas que nuestra solución requiere y seguramente deba ser extendida si se quiere utilizar en un escenario más complejo. Debido a que LAR Slice Service y LAR AM ejecutan bajo el mismo servidor, la API se implementó con llamadas directas al funciones dentro del código del AM.

5.2.2.2. LAR Aggregate Manager

Este servicio se encarga de interactuar con los recursos del LAR para su uso por los usuarios experimentadores. También mantiene información del estado de los recursos y de su utilización en experimentos.

Cada recurso que quiere ser gestionado por el AM debe ser ingresado a la base de datos de recursos. Los tipos de recursos soportados son router, container, switch, network e interfaz. Para cada tipo de recurso se debe implementar un controlador, dentro de la solución LAR AM, que se encargue de interactuar con el recurso físico por diferentes medios según el tipo de recurso.

Los recursos pueden relacionarse entre sí, por ejemplo container y router pueden tener varias interfaces asociadas. El recurso network, que representa una red lógica entre nodos (routers y containers), está asociado a los puertos de un switch a través del valor de la VLAN y con las interfaces de los nodos a través del puerto asociado a cada interfaz. A

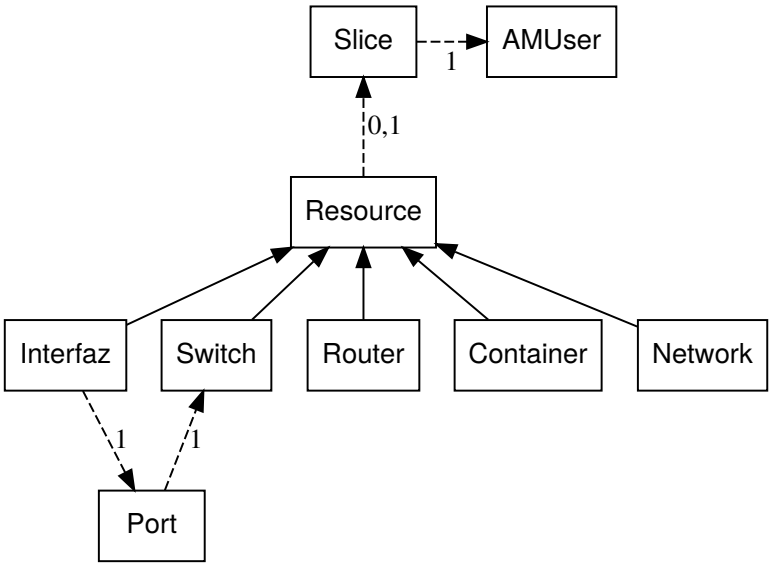


Figura 5.4: Conceptos del AM. Las flechas punteadas indican asociación, mientras que las continuas indican herencia.

su vez, existe un puerto trunk de un switch, que se conecta con el virtualizador, y al cual varias interfaces virtuales estan asociados. Los puertos del switch no representan recursos en sí mismos ya que no tiene sentido gestionarlos por separado de las interfaces de los nodos a los que están conectados.

Cada recurso puede ser dinámico o estático. Los dinámicos pueden ser creados y destruidos por el AM, los estáticos no. Algunos tipos de recursos pueden ser solicitados por un experimentador y otros no (por ejemplo un switch). Se definió también las precedencias entre recursos "creado por" y "poseído por" con el objetivo de automatizar la creacion y destruccion de recursos delegando la responsabilidad a los controladores, esto será de utilidad si se utiliza FRCP para comunicarse con los recursos. Para tener una idea general de las relaciones entre conceptos ver la figura 5.4.

El mapeo entre los recursos físicos y su representación en el LAR AM y sus características se muestran en el Cuadro [Recursos del AM].

Al igual que LAR Slice Service, los recursos utilizados por experimentadores se asignan a slices, pero a diferencia del anterior, las slices del AM pueden involucrar sólo recursos que éste gestiona.

Detalles de la implementación La implementación simplifica un poco la especificación de la sección anterior. En un escenario ideal, el AM mantiene el estado de los recursos actualizado, monitorizándolos con algún mecanismo, por ejemplo mediante el ya mencionado FRCP de OMF. La implementación del LAR AM no realiza monitorización de manera activa por lo tanto el estado de los recursos no cambia a largo de su vida. Por esta razón no se incluyeron en la base de datos propiedades de gran dinámica asociadas a los

Tipo	Representa	Estático / Dinámico	Solicitable	Creador	Dueño
router	Router ejecutando OpenWRT	Estático	Si	-	-
container	Contenedor Linux	Dinámico	Si	-	-
interfaz	Interfaz de red física o lógica, Ethernet o WLAN	Dinámico	No	-	router / container
network	Red lógica	Dinámico	Si	-	-
switch	Switch	Estático	No	-	-

Cuadro 5.2: Recursos del AM

recursos, por ejemplo direcciones ip, consumo de cpu, memoria, etc. El unico parámetro de estado que se incluye es si esta con vida, que para los recursos dinámicos implica que se encuentra desplegado y activo en el LAR. El LAR AM no interactúa con los recursos estáticos para actualizar este estado.

Otra simplificación es la creación de recursos dinámicos. Las precedencias entre recursos descritas en la sección anterior están orientada a su implementación con OMF. Por ejemplo si el recurso interfaz es creado por el recurso nodo, entonces la interfaz lógica de un contenedor linux efectivamente debería ser creada por el controlador del recurso ejecutando en el contenedor. La simplificación se da porque, dado que son pocos los tipos de recursos solicitables y que estos siempre se piden en el mismo orden dada una topología, ocurre que estos se crean en el mismo orden y por lo tanto no necesariamente por el controlador que lo tendría que crear, ya que no utilizamos FRCP para estas operaciones. Primero se reservan y crean, si es necesario, los recursos de tipo router y container que representan nodos en la red, luego se crean las interfaces lógicas de los contenedores y luego se crean las redes utilizando la información de todos los recursos anteriores sobre uno o varios switchs.

5.2.3. Interacción con los componentes de la testbed

Los servicios y dispositivos externos con los que el GestorLAR debe interactuar para configuración de recursos son switchs y contenedores LXC. La interacción con ellos se describe a continuación, ésta toma lugar dentro de la solución del LAR AM.

5.2.3.1. Switches

Los switches disponibles para el LAR no cuentan con ninguna interfaz o API para la administración remota de los mismos, únicamente se dispone de una interfaz web. Esto dificulta la automatización de la creación y asignación de VLANs a los distintos puertos.

Fue necesario desarrollar un módulo que interactúe con la interfaz web, permitiendo realizar las operaciones necesarias para la gestión del laboratorio. El módulo fue desarrollado en Python utilizando la biblioteca unirest[20] para realizar requests HTTP. Las funciones que desempeña son exclusivamente sobre VLANs. Permite obtener la lista de VLANs configuradas, crear nuevas VLANs y asignarlas a los distintos puertos.

El detalle de la implementación y su funcionamiento se encuentra en el Apéndice 1.

5.2.3.2. Containers e interfaces virtuales

Para la creación automatizada de containers se utilizó la API provista por LXD[12]. Esta API permite realizar todas las operaciones necesarias sobre los containers, luego de autenticar el cliente contra el servidor LXD.

Se desarrolló un módulo en Python utilizando la biblioteca oficial pylxd[6]. El módulo se encuentra integrado en el Aggregate Manager, siendo el responsable de la gestión del ciclo de vida de los containers, así como de mantener un inventario de los containers existentes.

Para la conexión de los containers a las distintas VLANs y creación de interfaces virtuales fue necesario interactuar directamente con el sistema operativo en el que se encuentra corriendo el hipervisor. El servidor se encuentra conectado a un enlace trunk que proporciona conectividad con las distintas redes virtuales.

Para conectar un container a una VLAN es necesario crear una interfaz virtual conectada a dicha VLAN en el servidor LXD, luego asignarla como padre de una interfaz del contenedor. Esta configuración se realiza a través de SSH, utilizando paramiko[21], una implementación en Python del protocolo SSHv2. La interfaz virtual, tambien llamada vlan-device, en el servidor LXD es creada con el comando vconfig. Luego se crea una nueva interfaz de red de tipo macvlan[11] en el contenedor utilizando la API de LXD asignando como padre la vlan-device recién creada.

El hipervisor expone la API HTTP rest de LXD, permitiendo al aggregate manager crear containers y asignarlos a las distintas redes de manera remota.

5.2.4. Interacción Aggregate Manager con Slice Service

Toda la comunicación entre Slice Service y Aggregate Manager se da al momento de configurar una topología. Cuando Slice Service recibe una topología a ser configurada sobre el LAR, éste debe asociarla a la slice que indica el experimentador para luego comenzar el proceso de reserva de recursos del LAR.

El primer paso es la autorización del usuario y creación de slices en cada AM. Para cada AM sobre el cual existe un recurso de la topología que es gestionado por él, se envía un mensaje speaks-for y request-slice-membership. El primero indica al AM que, por ejemplo, el usuario User1 que realiza la petición habla por otro usuario User2, quien tiene

permisos sobre el AM. En este paso se enviaria el ticket emitido por un administrador del AM para que el usuario User1 pueda acceder a los recursos del AM. Recordemos que esto no fue implementado por lo tanto un usuario siempre habla por sí mismo y no hay envío de tickets. El segundo mensaje solicita la membresía del usuario sobre la slice, dicha slice es creada si no existe. Aquí tenemos otra simplificación, ya que en la solución existe un único usuario puede miembro de una slice.

Luego, para cada recurso de la topología se envía un mensaje request-resource. Primero se solicitan los nodos de la topología, osea routers y contenedores. Luego se solicitan las redes. Si ocurre un error durante la solicitud de uno de estos recursos, la configuración de la topología será revertida indicando el error ocurrido y liberando los recursos solicitados hasta el error.

Finalmente, Slice Service puede liberar los recursos de una topología enviando un mensaje release-resource para cada recurso que esta utiliza. Ante la falla de la liberación de un recurso se continuará con los siguientes. El orden de liberación es el opuesto al de solicitud de recursos.

5.3. Despliegue de servicios en la infraestructura

El despliegue de los servicios de gestión se debe disponer sobre una red distinta de la utilizada para la realización de experimentos. El LAR dispone, a grandes rasgos, de máquinas virtuales y routers interconectados por switchs. Los servicios de gestión se implantaron en la misma red física que los servicios gestionados. La separación se realizó creando una VLAN específica para los servicios de gestión. Los servicios gestionados se encuentran conectados a la VLAN de gestión por una interfaz y las demás interfaces, en principio, a la VLAN por defecto. Recordemos que al crear una topología, los recursos asignados a un experimentador se aíslan mediante la creación de nuevas VLANs, asignadas especialmente para dicha topología. La representación de los servicios y recursos se encuentra en la figura 5.5.

El gestor utiliza dos máquinas virtuales en total. En la primera se instaló un hipervisor LXD, donde se crean contenedores utiliados exclusivamente para experimentos. En la segunda se instalaron todos los servicios asociados al GestorLAR y la ejecución de experimentos.

Se agregó una capa minima de seguridad a la máuina virtual que ejecuta los contenedores con los servicios del GestorLAR, configurando un firewall (con reglas de iptables) que restringen el acceso a servicios de uso interno del gestor.

5.4. Ciclo de vida de un experimento en la plataforma

A modo de resumen y para describir el funcionamiento del sistema como un todo, se explica en esta sección el ciclo de vida típico de un experimento utilizando la solución implementada desde el punto de vista del usuario experimentador e indicando el principal flujo de mensajes entre servicios y componentes.

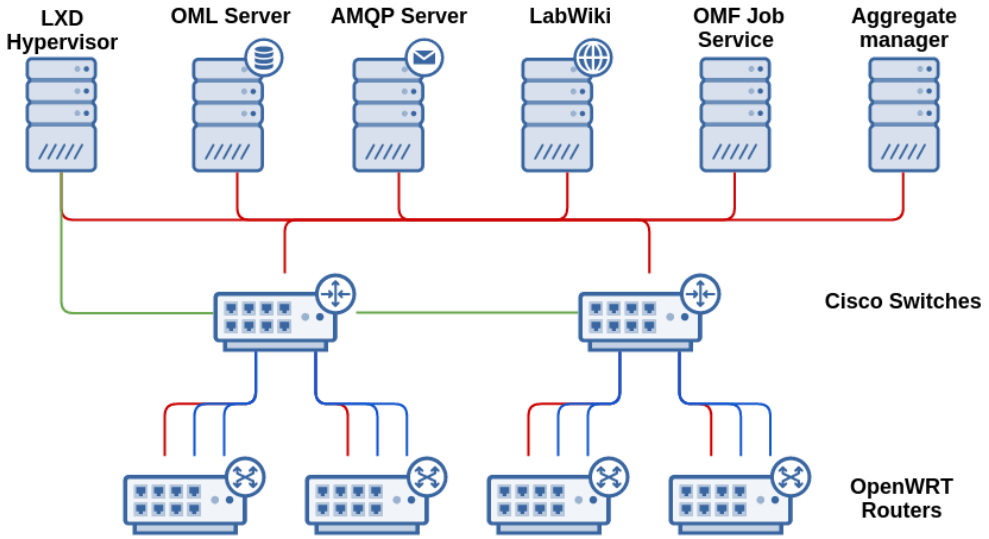


Figura 5.5: Diagrama de implantación del Gestor LAR. Se puede observar las redes de gestión (roja) y de experimentación (azul) junto con todos los recursos que forman parte de la testbed. La línea verde representa enlaces trunk, necesarios en el hipervisor LXD para conectar las interfaces virtuales de los contenedores a diferentes VLANs y entre switchs para desplegar VLANs a través de varios switchs.

1. El usuario ingresa a LabWiki e inicia sesión: el usuario ingresa a la web de LabWiki e ingresa sus credenciales: nombre de usuario y password. Hecho el ingreso, tendrá acceso a un editor de textos y un editor de grafos. El primero se utiliza para escribir experimentos en lenguaje OEDL así como documentación utilizando Markdown[25]. El segundo se utiliza para describir la topología. Los documentos creados se almacenan en LabWiki en un repositorio por usuario.

2. El usuario crea una descripción de la topología: utilizando el editor de grafos. La nomenclatura básica es: los cuadrados representan nodos y los círculos redes, los nodos son configurables en el tipo de recurso que representan: router o contenedor.

3. El usuario despliega la topología. Una vez creada y guardada la topología, debe crear una slice a la cual asociar los recursos que le serán asignados. Para esto arrastra la topología al panel de ejecución donde se listaran las slices existentes del usuario y se dara la opcion a crear una nueva solo indicando un nombre. Al dar click en “create”, LabWiki se comunica con Slice Service indicando si debe crear una slice y luego, si el paso anterior es exitoso, le envía la descripción de la topología asociada a la slice.

4. Setup de la topología en el LAR. Cuando Slice Service recibe la topología a ser configurada retorna una promise, LabWiki consulta periódicamente esa promise

consultando el estado de la tarea hasta que esté finalizada. Mientras tanto Slice Service se comunica con el LAR Aggregate Manager para configurar cada recurso, actualizando el progreso de la promise en cada paso.

5. Setup de cada recurso. Para cada recurso solicitado, el LAR Aggregate Manager debe crearlo, si el recurso es creable o tomarlo de la lista de disponibles en caso contrario.

6. Pre-ejecución de experimento. Una vez finalizada la implantación de la topología, el usuario carga o escribe un experimento en el editor de LabWIki y lo arrastra al panel de ejecución. En dicho panel podrá indicar sobre qué slice ejecutarlo y luego hacer click en "Correr".

7. Ejecución de experimento (proceso automático). LabWiki se comunica con omf_job_service, servicio que encola el experimento en su cola de trabajos. El usuario puede seguir su estado desde LabWiki. El scheduler actual de omf_job_service es FIFO, por lo que no se ejecutan experimentos concurrentemente. Una vez seleccionado para correr, el experimento se pasa a omf_ec para su ejecución. Durante la ejecución del experimento, los agentes omf_rc reciben ordenes del omf_ec y las ejecutan.

8. Ejecución de experimento. Desde el punto de vista del usuario, se pueden observar gráficas de mediciones recolectadas por OML y logs de omf_ec en el panel de ejecución de LabWIki en tiempo real.

9. Una vez finalizado, el usuario puede exportar los logs y mediciones, que se encuentran almacenados en una base de datos, desde la misma interfaz de LabWiki.

6 Experimentación

Con el fin de verificar la solución, llevamos a cabo las dos tareas fundamentales sobre la infraestructura desde el punto de vista del usuario experimentado. En primer lugar, realizamos pruebas sobre creación de topologías de diversa complejidad. Luego realizamos experimentos sobre alguna de ellas tratando de resaltar las cualidades de utilizar OMF junto con OML.

6.1. Creación de topologías

El sistema permite crear cualquier tipo de topología siempre que se cumpla que cada nodo contenga a lo sumo una interfaz conectada a cada red. Esto deja fuera el caso en que un nodo tenga varias interfaces conectadas a una red.

Una topología puede involucrar nodos físicos y virtuales. La cantidad de recursos involucrados en una topología no está acotada de manera explícita por usuario.

La cantidad de recursos utilizados por el gestor está acotada solamente por restricciones en los tipos de recursos. Los recursos routers al ser físicos están limitados obviamente en cantidad. Los recursos virtuales como contenedores, se encuentran limitados por la capacidad de hardware del hipervisor y por los siguientes parámetros:

Parámetro	**Descripción**
Tamaño de la red de gestión.	Las IPs de gestión se asignan mediante DHCP. El número máximo de nodos utilizables (accesibles una vez creados) está acotado superiormente por la cantidad de IPs asignables.

Asimismo, el número de redes está limitado por:

Parámetro	**Descripción**
Número máximo de identificadores de VLANs.	En una trama Ethernet 802.1Q se dispone de 12 bits parar el identificador de la VLAN, por lo que existen 4096 posibles VLANS de las cuales 0x000 y 0xFFF están reservadas por el protocolo. Además, se utilizó la 0x001 como la VLAN de gestión.

6.1.1. Ejemplos

Por mencionar algunos ejemplos de topologías, fue posible crear utilizando el gestor las topologías en la Figura 6.1. Estos ejemplos son algunos de los que encontramos en los distintos cursos de redes de computadoras. El de la izquierda conecta dos nodos a través de una red con varios saltos, donde existen dos posibles rutas para el tráfico entre ellos. La imagen central, muestra varios nodos conectados a la misma red. La imagen de la derecha, muestra una topología con tres islas de nodos que se encuentran conectadas a través de un nodo central, que se comportaría como un router. Vale la pena aclarar que las redes se comportan como un switch, es decir que el tráfico entre dos nodos no es visible por un tercer nodo, aunque este comportamiento se puede modificar.

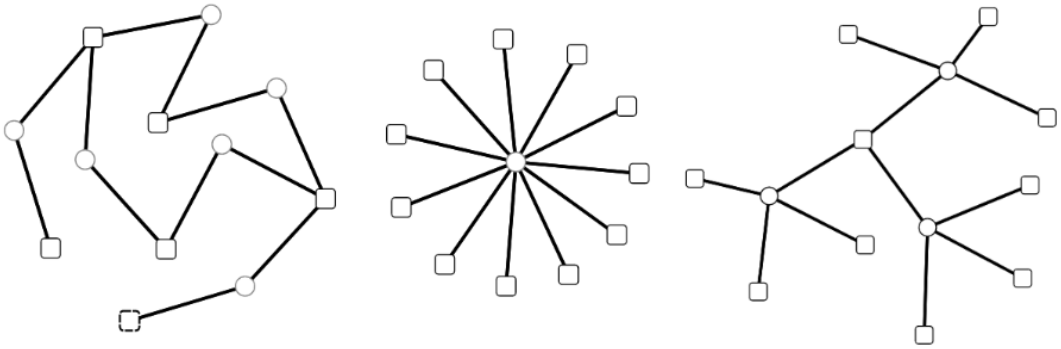


Figura 6.1: Ejemplos de topologías. En cada grafo, los nodos circulares representan redes y los cuadrados nodos. Los grafos fueron descritos utilizando LabWiki.

6.2. Ejecución de experimentos

En esta sección, mostraremos la ejecución de un experimento simple utilizando toda la capacidad de OMF junto con OML.

6.2.1. Primer experimento

Como primer experimento, planteamos la observación del tráfico entre dos nodos por un único camino a través de un nodo intermedio, cuando el nodo intermedio da de baja una de las interfaces por las que pasa el flujo de datos. El tráfico se observa en el nodo intermedio en sus dos interfaces. Para cada interfaz, obtenemos la cantidad de paquetes entrantes y salientes. El tráfico generado es UDP en un solo sentido a 10 megabits por segundo. Para ver el experimento completo dirigirse al apéndice 4.

En la Figura 6.2 vemos el escenario planteado.

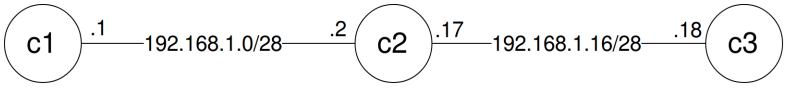


Figura 6.2: Topología del experimento 1. El nodo cliente es c1 y el servidor c3.

6.2.1.1. Software utilizado

Para generar tráfico se utiliza la herramienta iperf. Ésta permite medir el desempeño de una red mediante la generación de tráfico TCP o UDP entre nodos donde opera en modo cliente o servidor (o ambos, si el tráfico es bidireccional). También permite configurar varios parámetros de una conexión TCP o de los datagramas UDP. Contamos con una implementación que se integra con la biblioteca OML por lo que podemos obtener y almacenar información en tiempo real sobre el tráfico generado. Para medir el tráfico en el nodo intermedio se utiliza nmetrics, una herramienta provista dentro del paquete de aplicaciones integradas con OML. Permite medir la utilización de los recursos de hardware de un nodo, como son CPU, memoria e interfaces de red. Finalmente, las rutas se definieron de manera estática utilizando el comando ip.

6.2.1.2. Resultados

Al inicio del experimento se configuran las IPs de todos los nodos y las rutas. Luego se inicia el generador de trafico. 20 segundos después se borra la IP de la interfaz eth2 (conectada a la red entre el nodo intermedio y el destino del flujo) en el nodo intermedio. 20 segundos después se asigna nuevamente una ip a eth2 y finalmente luego de 120 segundos de comenzado el experimento se detiene el generador de tráfico y la aplicación de medición de la red.

En la Figura 6.3 se puede observar los gráficos de cantidad de paquetes recibidos, a la izquierda, y cantidad de paquetes enviados, a la derecha, de cada interfaz del nodo intermedio de la topología. Se puede observar claramente a la derecha el intervalo de tiempo en que la interfaz eth2 estuvo sin IP.

6.2.2. Segundo experimento

El experimento que planteamos consiste en la configuración manual de una red de múltiples nodos, luego la generación de tráfico entre dos nodos con varios saltos entre ellos y la caída de un enlace con el fin de observar el comportamiento de la red y el flujo de datos ante este evento.

La red se puede observar en la Figura 6.4. El tráfico se da entre c1 y c6 en forma de cliente servidor respectivamente. En el nodo c2, se definen rutas a c6 por los nodos c3 y c4.

6.2.2.1. Ejecución del experimento

La topología del segundo experimento implementada en LabWiki se puede observar en la figura 6.5.

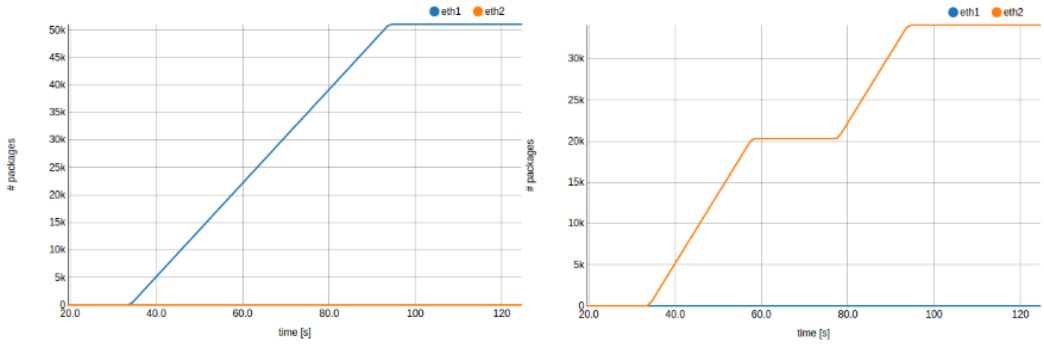


Figura 6.3: Gráfica de tráfico del primer experimento. Cantidad de paquetes en función del tiempo. El tiempo 0 corresponde al inicio de la aplicación que obtiene las mediciones. Los gráficos fueron obtenidos ejecutando el experimento en LabWiki.

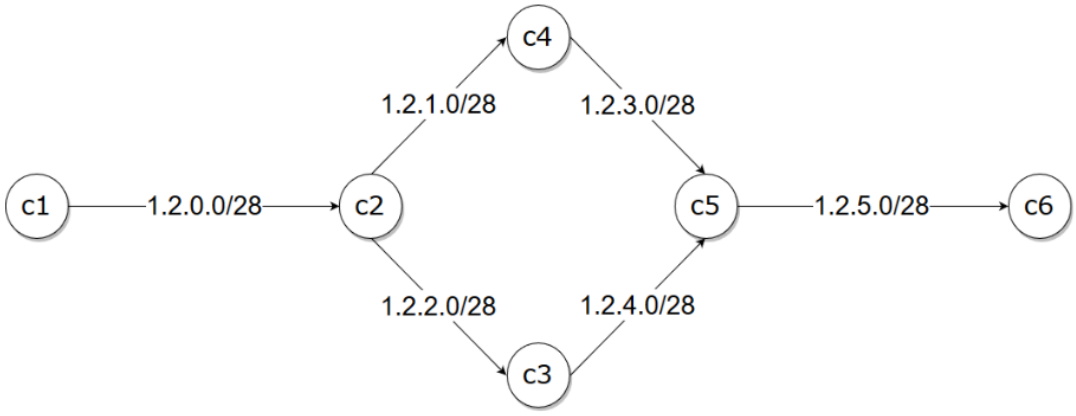


Figura 6.4: Topología del experimento 2

Para ejecutar el experimento, debemos escribirlo en lenguaje OEDL. Una vez escrito y cargado en LabWiki, lo podremos ejecutar y ver los resultados.

Dentro del experimento, primero se declaran los recursos nodos que se van a utilizar, configurando sus interfaces si corresponde y las aplicaciones que ejecutarán. Luego se describen los eventos ante los cuales se realizan acciones.

El primer evento, que da comienzo al experimento es ALL_NODES_UP, disparado cuando todos los recursos nodos aceptaron la membrecía al experimento. En primer lugar se ejecutan los comandos para configurar las rutas estáticas en los nodos. Luego de 15 segundos,se inician las aplicaciones iperf, primero el servidor y luego el cliente. Pasados los 40 segundos se tira la interfaz de menor costo entre c2 y un contenedor sucesor. Luego de 140 segundos se detienen las aplicaciones iperf, se remueven las rutas estáticas (utilizando el comando ip) y se finaliza el experimento.

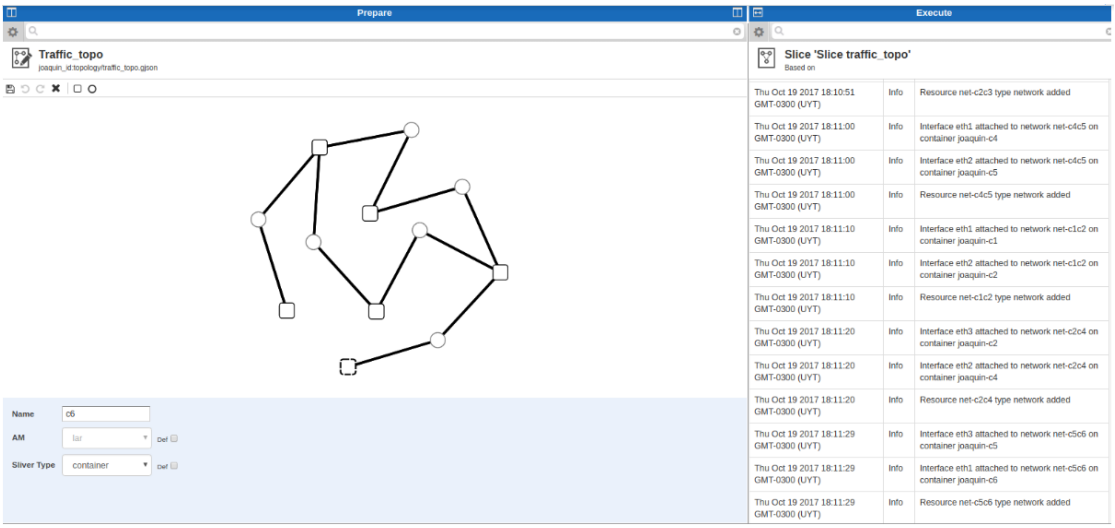


Figura 6.5: Configuración de la topología en LabWiki. A la izquierda se encuentra el editor de grafos donde se describe la topología. A la derecha el log obtenido de la implantación.

El experimento consiste en la siguiente secuencia de pasos:

- Se crean seis contenedores. Se crean en cada contenedor la cantidad de interfaces que corresponden, según se muestra en la figura anterior.
- Se configuran las direcciones IP y máscaras de red para cada interfaz.
- Se configuran de manera estática las rutas desde todos los contenedores hacia la red que conecta los contenedores 5 y 6, de manera que el costo de enviar el tráfico a través de c4 es menor que a través de c3.
- Se mide en los nodos c3, c4 y c6 el tràfico recibido.
- Se ejecuta en c1 una aplicación que genera tráfico hacia c6 a través de c4.
- Luego de 20 segundos, se baja el enlace entre c2 y c4.
- Se observan las medidas obtenidas por los contenedores.

6.2.2.2. Resultados

Durante la ejecución de este experimento, OMF y varios de sus componentes de su entorno de ejecución mostraron un alto consumo de CPU. Tanto el controlador de experimento como el controlador de recurso, omf_ec y om_rc respectivamente, son los más problemáticos en este sentido, ya que no cuentan con una implementación concurrente que pueda sacar provecho de múltiples CPUs.

Otro de los componentes con alto uso de CPU fue el servidor OML. Este puede ser un cuello de botella durante la ejecución del experimento ya que recibe datos constantemente tanto el controlador de experimento como los controladores de recursos. La implementación de este servidor también es mono-proceso pero es posible aumentar el desempeño si se ejecutan varias instancias del servidor detrás de un balanceador de carga TCP.

Debido a estas limitaciones, no fue posible ejecutar este experimento de manera apropiada, ni experimentos de mayor porte, ya que la saturación de la CPU causa que los mensajes del controlador del experimento hacia los controladores de los distintos recursos involucrados no fueran enviados en tiempo y forma.

7 Trabajo a futuro y puesta en producción

La solución propuesta puede ser vista como una prueba de concepto. Aún queda trabajo por delante para lograr una testbed que satisfaga de mejor manera los requerimientos de usuarios ya sea con fines de investigación o de enseñanza.

Por un lado en el trabajo a futuro indicamos requerimientos que nos parecen necesarios de incluir en la solución. Por otro lado, en la puesta en producción detallamos requerimientos y condiciones que nos parecen necesarias para que la solución escale y sea utilizada apropiadamente. Los puntos en común de ambos temas son descritos desde la perspectiva de cada uno.

7.1. Trabajo a futuro

En esta sección detallaremos los principales aspectos que no fueron implementados por no considerarse críticos para una primera versión del gestor, pero que aportarían capacidades adicionales al mismo y podrían ser implementados en un futuro.

7.1.1. Mayor acceso a los recursos

Se reconoce que es de interés que los usuarios puedan acceder a una consola en los recursos de cómputo que se les han asignado para mayor control y configuración de éstos. Sin embargo, dar un acceso más profundo dentro de LAR requiere más controles y seguridad sobre todos los servicios expuestos internamente y sobre los recursos de otros usuarios, por lo que este requerimiento depende a su vez del de seguridad.

7.1.2. Heterogeneidad de servicios virtualizados

Al inicio del proyecto se contempló la idea de utilizar KVM o XEN para manejo de hosts virtualizados en LAR, ya que son los virtualizadores utilizados por el grupo de investigación. Por una cuestión de complejidad y rendimiento se optó finalmente por la utilización de contenedores Linux. Una razón para utilizar maquinas virtuales es la carga y exportación de imágenes de sistemas operativos por parte del usuarios al LAR, si bien esto puede ser llevado a cabo con LXD, es conveniente soportar varios formatos.

7.1.3. Corrección de bugs de OMF

Debido a la poca madurez de la última versión de OMF existen numerosos bugs en su implementación. Si bien generalmente la corrección de estos se espera que sea realizada

por la comunidad o mantenedores del código fuente también puede ser realizada por nosotros, en particular los bugs que más afectan el uso en el LAR.

7.1.4. Gestión y monitorización de recursos en el AM

Actualmente, los recursos gestionados por el Aggregate Manager no están limitados en cuanto a los recursos de hardware que pueden utilizar. En el caso de los routers, al ser dispositivos dedicados, esto no representa mayores inconvenientes. Sin embargo, en el caso de los contenedores, ejecutan sobre un único virtualizador por lo que los recursos de hardware consumidos por uno pueden impactar negativamente el rendimiento de los otros. LXD proporciona excelentes mecanismos de gestión de recursos de hardware para los contenedores[10]. Además de los límites de CPU y memoria RAM presentes en todos los virtualizadores tradicionales, es posible limitar la velocidad de acceso a disco o el tráfico de red. Esto permite simular redes con distintas tecnologías (Wi-Fi, Ethernet, Fibra óptica, etc.), o evaluar el rendimiento de aplicaciones con distintos tipos de almacenamiento. Estos límites pueden, además, ser fijados de manera dinámica mientras el contenedor se encuentra en ejecución, permitiendo emular congestiones de red o saturación de recursos.

7.1.5. Federación

Uno de los objetivos a futuro del proyecto es la federación del laboratorio con el proyecto FIBRE[4]. FIBRE es una red de testbeds disponible para la ejecución de experimentos por parte de estudiantes e investigadores. Actualmente cuenta con 12 testbeds en Brasil, así como interconexiones con Estados Unidos y Europa.

Para formar parte de una red como FIBRE es necesario implementar el estándar SFA descrito anteriormente.

7.1.6. Seguridad

El gestor del LAR está pensado para ser utilizado, al menos inicialmente, únicamente desde la red del INCO por docentes del mismo. Por esta razón, la seguridad del sistema implementado no fue una de las prioridades para este proyecto. Si bien se tomaron en cuenta ciertas consideraciones de seguridad, como el firewall configurado en la máquina central del gestor, hay muchos aspectos en los cuales es necesario trabajar, especialmente si se desea federar la testbed en un futuro.

Por otra parte, OML no utiliza ningún tipo de autenticación y es por lo tanto considerado inseguro[2]. Debe ser utilizado únicamente en redes privadas, detrás de un firewall. En el caso del LAR, el servidor OML se encuentra en un container dentro del servidor principal, protegido por lo tanto por el firewall del mismo.

7.1.7. Concurrencia

El planificador utilizado actualmente por el gestor es el provisto por el proyecto omf_job_service no permite la ejecución concurrente de experimentos. El mismo utiliza un algoritmo de planificación FIFO (First In, First Out), lo que hace que sea necesario

finalizar un experimento antes de comenzar otro. Para un conjunto de usuarios reducidos este algoritmo es suficiente, pero en caso de ser utilizado por más usuarios o federar la plataforma será necesario implementar un algoritmo distinto que permita la ejecución de experimentos de manera concurrente.

7.1.8. Utilización de DNS

Son conocidas las ventajas de utilizar DNS. Debe cambiarse la utilización de IPs fijas por nombres de dominio en la red de gestión. También es recomendable la asignación de nombres de dominio a los recursos hosts dinámicos y estáticos.

7.2. Puesta en producción

Para que la solución sea desplegada en un entorno de producción habría que realizar algunos agregados y modificaciones que permitan por un lado mayor interacción de los usuarios con la testbed, y por otro, escalar en cantidad de usuarios y por lo tanto en recursos utilizados sin degradar significativamente el servicio.

En primer lugar, describiremos los casos de uso que escapan a los contemplados en este proyecto pero que son deseables en un entorno de producción. Luego se detallan las modificaciones necesarias en diferentes aspectos.

7.2.1. Otros casos de uso

7.2.1.1. Creación y configuración de recursos

La creación y asignación de recursos se debe realizar por los usuarios sobre el AggregateManager que maneja los recursos. Esto permite un mejor manejo de las cuotas de usuario y sobre todo, permitirá una configuración más específica de los recursos solicitados. Además, debe ser posible para un usuario acceder a los recursos hosts que le son asignados para configurar software y el entorno sobre el que va a experimentar. La solución actual es demasiado rígida en este sentido.

7.2.1.2. Ejecución de experimentos

Sin bien es necesario ofrecer una interfaz común para ejecutar experimentos, también debe ser posible ejecutarlos a demanda a través de la CLI de omf_ec, ya sea desde un host particular de la testbed o desde la PC del usuario.

7.2.1.3. Acceso a base de datos de experimentos

Cada experimento ejecutado crea una nueva base de datos en el servidor PostgreSQL. Si bien es posible exportarla desde LabWiki, si el experimento es ejecutado desde ahí, puede ser de interés poder acceder a la base de datos con el objetivo de hacer consultas SQL sobre los datos recolectados o exportar los datos de un experimento ejecutado por otro medio.

7.2.2. Seguridad

Para proveer los servicios descritos en los casos de uso hay que agregar y reforzar la seguridad en el sistema. La creación de recursos desde el AggregateManager implica que éste tiene que poder gestionar usuarios. El AggregateManager debe, idealmente, gestionar claves publico privadas por usuario que autentiquen el acceso de éstos a los recursos. Por un lado, es un mecanismo seguro de autenticación al acceder por ssh a los host. Por otro lado es un mecanismo soportado por OMF para autenticar una ejecución de omf_ec con los omf_rc utilizados. Recordemos que en la versión actual no existe autorización en esta comunicación y basta con conocer el nombre de los recursos para comunicarse con ellos a través de OMF.

El hecho de que los usuarios puedan ejecutar experimentos desde su PC implica que deben tener acceso a la red de gestión o parte de ella, al menos al host que ejecuta RabbitMQ que es el medio de comunicación entre omf_ec y omf_rc. Por lo tanto es necesario proveer acceso a dicha red de manera segura, mediante VPN.

7.2.3. Optimización de OMF

Los componentes imprescindibles para la ejecución de recursos deben ser optimizados. El omf_ec, como ya se dijo, tiene alto consumo de CPU ejecutando experimentos complejos. Además, si múltiples usuarios ejecutan experimentos mediante LabWiki, potencialmente varias instancias de omf_ec ejecuten concurrentemente por lo que se hace imperiosa la necesidad de no solo optimizarlo, sino proveer una implementación concurrente.

El omf_rc sufre de los mismos problemas, por seguir el mismo diseño y compartir gran parte del código con el primero, aunque es menos susceptible a consumir tanto CPU ya que generalmente esta menos activo durante los experimentos. Respecto a su ejecución en dispositivos de bajos recursos, creemos que es factible lograrlo si se quitan muchas de las dependencias que son poco utilizadas y pueden ser prescindibles en estos dispositivos. La implementación actual de RC apunta a un conjunto heterogéneo de sistemas operativos tipo unix, una simplificación específica para OpenWRT debería hacer más eficiente su ejecución.

7.2.4. Implantación

La implantación de los servicios en un entorno de producción debe ser sobre hardware muy potente en particular para el servidor que ejecuta omf_ec y el servidor OML. La implantación realizada en este proyecto provee modularidad suficiente como para que los servicios sean distribuidos sobre tantos servidores como se dispongan. Ya que los servicios se encuentran instalados en contenedores LXC su migración entre servidores es muy sencilla.

8 Cronología

Si bien el proyecto transcurrió según se detalla en este documento, hubo trabajo que no generó resultados inmediatos o tangibles, pero que fue realizado ya sea para tomar una decisión de diseño o porque se creyó lo mejor en ese momento, principalmente al inicio del mismo. En esta sección mostramos el cronograma de las tareas realizadas y detalles sobre estas. El periodo de tiempo considerado es de abril 2016 a noviembre 2017.

Las tareas las agrupamos en cuatro tipos de actividades.

1. Análisis y diseño
2. Implementación e implantación de solución final
3. Verificación y validación
4. Documentación

Estas etapas no son secuenciales. El progreso de las tareas en cada una se puede observar en la figura 8.1.

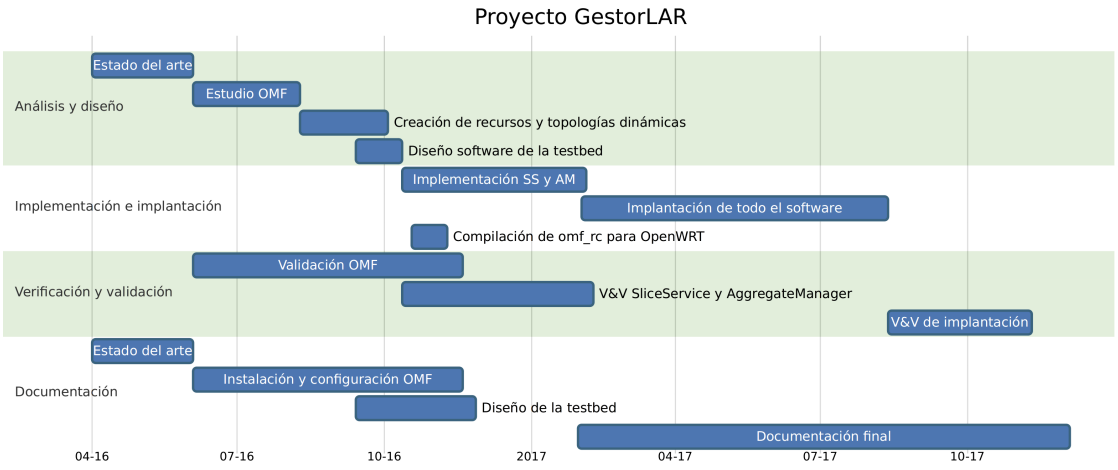


Figura 8.1: Diagrama de Gantt de tareas realizadas. En el eje Y se indica mes y año. Las tareas son autodescriptivas. El caso de la implementación de SliceService y AggregateManager tambien incluye la interacción con el Switch y virtualizador LXD.

En el análisis y diseño hubo trabajo que no se ve reflejado en la solución final ya que en esta etapa se realizaron pruebas de software existente y desarrollo de conceptos

que finalmente no se utilizaron. La implementación e implantación fue desempeñada con fluidez al comienzo y de manera más obstaculizada hacia el final. Esto se explica por la dificultad de la puesta en funcionamiento de manera óptima del software que compone la solución en su conjunto. Las actividades de verificación y validación, y la de documentación abarcan casi toda la duración del proyecto intensificando en el final. Este proceder nos parece natural en relación a la documentación. Respecto a verificación y validación, se desarrolló así por ser un sistema compuesto por múltiples componentes de software, de los cuales la mayoría estaban ya desarrollados. Esto permite que el proceso de verificación y validación se realice de manera iterativa e incremental sobre los componentes que forman parte de la solución.

A continuación se detalla en profundidad el trabajo en el análisis y diseño, implementación e implantación, y verificación y validación. En particular, se detalla el trabajo que no fué documentado anteriormente.

8.1. Análisis y diseño

Para lograr una mejor comprensión de los problemas que afrontamos se dedicó tiempo, al inicio del proyecto, para realizar pruebas de los componentes de software existentes que se podríamos utilizar. Esto también nos permitió obtener una estimación del alcance final del proyecto y determinar los nuevos componentes que hubo que implementar.

En concreto, las pruebas tienen como fin encontrar la mejor manera de integrar OMF, como ejecutor de experimentos, con una infraestructura que ofrezca todo lo que se plantea en el objetivo del proyecto.

El framework OMF ofrece muchos componentes de software utilizados a lo largo de la vida de un experimento. En primer lugar, debimos dedicar tiempo en comprender la arquitectura, la función de cada componente y su esencialidad para el conjunto. Vale la pena aclarar, que no encontramos manuales ni documentación que especifique la integración entre ellos, sino únicamente el código fuente alojado en varios repositorios bajo el grupo *mytestbed* en github y el paper con el diseño original. Luego evaluamos el estado de cada uno, como la capacidad de configuración, documentación y estabilidad en su funcionamiento.

Comenzando por el conjunto mínimo de componentes para realizar un experimento, OMF demostró buena documentación y una puesta en funcionamiento inmediata. Sin embargo, al intentar integrar nuevas funciones surgieron los problemas de falta de documentación sobre la arquitectura, sobre la configuración, y en algunos casos sobre el rol específico del software.

8.1.1. Tareas realizadas

Luego de haber probado exitosamente la ejecución de experimentos con los componentes mínimos, es decir omf_ec y omf_rc (de tipo nodo o host), seguimos integrando los componentes LabWiki y JobService que actúan de frontend web y planificador de experimentos respectivamente. Esto nos permitió tener una primera versión de una infraestructura de ejecución de experimentos multiusuario.

A continuación, analizamos diferentes enfoques para la creación de hosts virtuales y redes dinámicas. A grandes rasgos distinguimos dos: implementar recursos OMF, que se encarguen de crear estas entidades dinámicas, o realizar una implementación desde cero con una arquitectura nueva. La ventaja del primer enfoque es que el software que utiliza la infraestructura está unificado, y la interfaz de interacción con los recursos está unificada para todos, ya sean accesibles para experimentadores o de uso interno de la testbed. El segundo enfoque es simplemente el complemento del primero.

Siguiendo el enfoque ya provisto por OMF para máquinas virtuales, decidimos implementar un recurso ContainerFactory, encargado de crear recursos Container que representan contenedores LXC. El agente omf_rc que implementa ambos recursos ejecuta en el mismo host que el hipervisor LXD e interactúa con éste directamente.

El siguiente paso fue determinar la manera de permitir a un experimentador crear estos recursos dinámicos. La primer idea fue utilizar el mismo omf_ec para crear los nodos virtuales que para llevar a cabo el experimento. Implementamos métodos y eventos para facilitar la creación y configuración de contenedores a través del recurso Container. Luego hicimos monkey patch de las clases que manejan el lenguaje OEDL para que nuestros métodos estén disponibles en dicho lenguaje. El resultado fue el esperado y la creación de contenedores utilizando omf_ec funcionaba en tanto funcione la implementación del recurso Container. Si bien este enfoque funcionó, no nos pareció ideal ya que no se logra una definición sencilla de las topologías ni se puede visualizar adecuadamente como grafos.

El descubrimiento (o mejor dicho comprensión del funcionamiento) del plugin para creación de topologías de LabWiki, el componente con el que interactúa (omf_slice_service), y quien gestiona los recursos mediante FRCP (omf_sfa), pareció lo que faltaba para satisfacer el requerimiento. Sin embargo, nunca logramos hacer funcionar estos últimos y pese a haber dedicado mucho tiempo en buscar documentación y lectura del código fuente, no logramos comprender del todo su funcionamiento. Dado el tiempo invertido en este último paso y los pocos resultados obtenidos decidimos implementar ambos componentes nosotros.

En general, el trato con OMF fue dificultoso. Una arquitectura modular de software que se correspondía y muchas veces parecía no hacerlo, posiblemente por falta de comprensión por nuestra parte así como ocurrencia de bugs en ellos. No es un trabajo agradable modificar o arreglar software del que no tenemos documentación.

8.2. Implementación e implantación

Durante el periodo de implementación se desarrollaron los modulos SliceService y AggregateManager del GestorLAR que interactúan con LabWiki a través del TopologyPlugin, como se detalla en la sección 5.

Al momento de la implantación, habíamos comprobado la integración de los componentes con sus adyacentes, pero todavía no los habíamos integrado en una misma instalación. Una vez hecha la instalación completa, comenzó un proceso iterativo sobre cada uno para lograr una configuración del mismo y de su entorno que logre el comportamiento y

desempeño deseado o aceptable. Por ejemplo, encontramos problemas de desempeño que en algunos casos pudimos arreglar, ya que a la par realizamos pruebas cada vez más complejas sobre la infraestructura.

8.3. Verificación y validación

A lo largo de todo el proyecto, validamos el software utilizado, con el objetivo de ratificar nuestro entendimiento de la documentación y comprender mejor el funcionamiento de los componentes. Estos, en general, fueron validados ya integrados, menos el caso de omf_rc al que se realizaron pruebas modulares, al ser provistas herramientas de interacción con éste y considerando que realizamos una implementación de recursos OMF.

Para los componentes SliceService y AggregateManager realizamos pruebas unitarias automáticas y pruebas de integración entre ellos y el plugin de LabWiki, que es su única interacción externa.

Vale la pena aclarar que las pruebas de validación sobre OMF se realizaron siempre desde una interfaz de usuario, ya sea un interfaz de línea de comandos (CLI) o una web. Por lo tanto los casos de uso seguidos fueron generalmente los típicos ya que el objetivo no fue la validación del software sino de nuestros conocimientos.

9 Conclusiones

Se implementó una primera versión del gestor para el Laboratorio Académico de Redes (LAR) del Instituto de computación de la Facultad de Ingeniería, que permite definir redes lógicas y ejecutar experimentos sobre ellas, utilizando los recursos disponibles en el laboratorio.

El gestor puede ser utilizado para la ejecución de experimentos que permitan probar nuevas tecnologías de redes utilizando un escenario realista. Los experimentos son especificados de manera no ambigua utilizando un lenguaje bien definido, lo que permite la replicación del experimento repetidas veces y por lo tanto la verificación de los resultados.

El gestor será útil también para los distintos cursos de la carrera de Ingeniería en Computación para los cuales se requiere particionar la infraestructura disponible en un conjunto de topologías independientes, asignando una porción de los recursos disponibles a cada estudiante para la realización de trabajos de laboratorio. Actualmente esta partición es realizada de forma manual por los docentes del curso, por lo que el gestor permitirá realizar dichas tareas de forma más rápida y flexible. Se dispone de una interfaz gráfica sencilla para la definición de la topología de red deseada, lo que permite que el sistema sea utilizado por usuarios sin necesidad de poseer experiencia en administración de redes.

El sistema implementado contiene las funcionalidades básicas necesarias para gestionar la infraestructura disponible y ejecutar experimentos. Se trata de una versión totalmente funcional, con un conjunto de funciones acotado pero suficiente para un uso interno al instituto de computación.

9.1. Análisis retrospectivo

Podemos concluir de la experiencia algunas ideas sobre el enfoque y la estrategia de resolución de los problemas planteados, de las decisiones tomadas y del proyecto en general.

En primer lugar el proyecto consta de dos problemas distintos pero relacionados que distinguimos y propusimos soluciones aunque incompletas o imperfectas.

Sobre la realización de experimentos, la solución es particularmente imperfecta ya que el framework que se decidió utilizar no es completamente estable y sufre de numerosos bugs. Debido a la falta de documentación del framework, y en general sobre este tema, gran parte del tiempo invertido en el planteo de esta solución se la llevó el entender las herramientas existentes que podrían ayudarnos a resolverlo. OMF fue el candidato desde un principio por su renombre en el ámbito, su utilización por parte de la mayoría de las organizaciones que ofrecen testbeds y su flexibilidad al momento de su implantación. Sin

embargo en la práctica demostró los fallos que tiene y que pese a que su diseño está bien logrado no podemos decir que sea un claro ganador sobre Nepi.

El módulo GestorLAR, encargado de gestión de recursos, es una solución incompleta ya que, como se vio en la sección de diseño, realiza las acciones mínimas de gestión para llevar a cabo un experimento. Esto es: mantener consistentes los datos sobre asignación de recursos a usuarios y la interacción mínima con estos para configurar una topología. Pese a ser incompleta, esta solución está bien lograda y puede ser extendida apropiadamente.

La decisión de utilizar OMF influyó negativamente en el alcance final de lo implementado en el GestorLAR. El costo de implantación de los servicios de OMF resultó muy alto para ser software ya implementado y con años de uso. Además, subestimamos el costo de instalar omf_rc en los routers del LAR que utilizan el sistema operativo OpenWRT, la razón de esto fue por no realizar una investigación profunda sobre las dependencias del agente y la disponibilidad de software para dicho sistema operativo; y subestimar las diferencias con los sistemas operativos linux convencionales para los que omf_rc fue implementado.

En segundo lugar, afrontamos los dos problemas de manera secuencial y el orden en que los afrontamos determinó el resultado final: primero el problema de ejecución de experimentos, que nos dio otra perspectiva del objetivo del segundo problema, conocimientos sobre cómo hacerle frente y simplificar su puesta en marcha. La razón de hacerlo en ese orden fue el desconocimiento e incertidumbre sobre el primero.

Por último, dados los resultados obtenidos, concluimos que aunque no todos los requerimientos del proyecto fueron realizados completamente, construimos una base a partir de la cual se puede continuar con cada problema de manera independiente.

Bibliografía

[1] Nicta, 2016. https://www.nicta.com.au/contact/.

[2] Oml2 server, 2016. http://oml.mytestbed.net/doc/oml/latest/oml2-server.1.html.

[3] Fed4fire+ - federation for fire plus, 2017. https://www.fed4fire.eu/.

[4] Fibre - future internet brazilian environment for experimentation, 2017. http://fibre.org.br/.

[5] Geni, 2017. http://www.geni.net/.

[6] Github - lxc/pylxd: Python module for lxd, 2017. https://github.com/lxc/pylxd.

[7] Homepage - celery: Distributed task queue, 2017. http://www.celeryproject.org/.

[8] Inria - inventeurs du monde numerique, 2017. https://www.inria.fr/.

[9] Linux containers - lxd - introduction, 2017. https://linuxcontainers.org/lxd/.

[10] Lxd 2.0: Resource control [4/12] | ubuntu insights, 2017. https://insights.ubuntu.com/2016/03/30/lxd-2-0-resource-control-412/.

[11] lxd/containers.md at master lxc/lxd - github, 2017. https://github.com/lxc/lxd/blob/master/doc/containers.md.

[12] lxd/rest-api.md at master Â· lxc/lxd Â· github, 2017. https://github.com/lxc/lxd/blob/master/doc/rest-api.md.

[13] Nitos - nitlab - network implementation testbed laboratory, 2017. http://nitlab.inf.uth.gr/NITlab/nitos.

[14] Ofelia - about ofelia, 2017. http://www.fp7-ofelia.eu/about-ofelia/.

[15] Openwrt, 2017. https://openwrt.org/.

[16] Orbit, 2017. http://www.orbit-lab.org/.

[17] Planetlab - an open platform for developing, deploying, and accessing planetary-scale services, 2017. https://www.planet-lab.org/.

[18] Quagga software routing suite, 2017. http://www.quagga.net/.

[19] Rack: a ruby webserver interface, 2017. http://rack.github.io/.

[20] Unirest for python - simplified, lightweight http request library, 2017. http://unirest.io/python.html.

[21] Welcome to paramiko! - paramiko documentation, 2017. http://www.paramiko.org/.

[22] Vinicius Goncalves Braga. Avaliacao das funcionalidades do broker e realizacao de testes fim-a-fim em recursos openflow e sem fio com o omf 6. Technical report, Universidade Federal de Goias, Instituto de Informatica, 2016.

[23] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. Omf: A control and management framework for networking testbeds. *SIGOPS Oper. Syst. Rev.*, 43(4):54–59, jan 2010.

[24] Wikipedia. Cross compiler — wikipedia, the free encyclopedia, 2017. https://en.wikipedia.org/w/index.php?title=Cross_compiler.

[25] Wikipedia. Markdown — wikipedia, la enciclopedia libre, 2017. https://es.wikipedia.org/w/index.php?title=Markdown.

[26] Wikipedia. Open vswitch, 2017. http://openvswitch.org/.

[27] Wikipedia. Testbed — wikipedia, the free encyclopedia, 2017. https://en.wikipedia.org/w/index.php?title=Testbed.

Apéndice 1: Documentación técnica

GestorLAR

Llamamos GestorLAR a la solución implementada que gestiona los recursos del Laboratorio Académico de Redes. Esta solución se integra con OMF (cOntrol and Management Framework) para permitir la realización de experimentos y gestión de topologías utilizando este framework.

La solución esta implementada en el lenguaje Python utilizando el framework para aplicaciones web Django. Básicamente consta de una API HTTP rest con la que interactúa OMF. Utiliza una base de datos SQL para almacenar datos de recursos y su estado, y provee una interfaz web con visualización básica sobre los recursos.

La solución esta compuesta en dos módulos: SliceService y AggregateManager. El primero expone la API con la que interactúa OMF y utiliza las funciones expuestas por el AggregateManager para satisfacer las solicitudes de la API sobre los recursos. Además almacena información de slices a las que se asocian recursos, que en esta fase involucran a un único AggregateManager pero a futuro pueden involucrar más. El segundo, es el encargado de interactuar con los recursos y mantener su estado. Éste utiliza dos submódulos independientes para interactuar con los recursos del LAR, estos son el SwitchController y LXDController para gestión de switchs y contenedores LXD respectivamente.

Slice Service

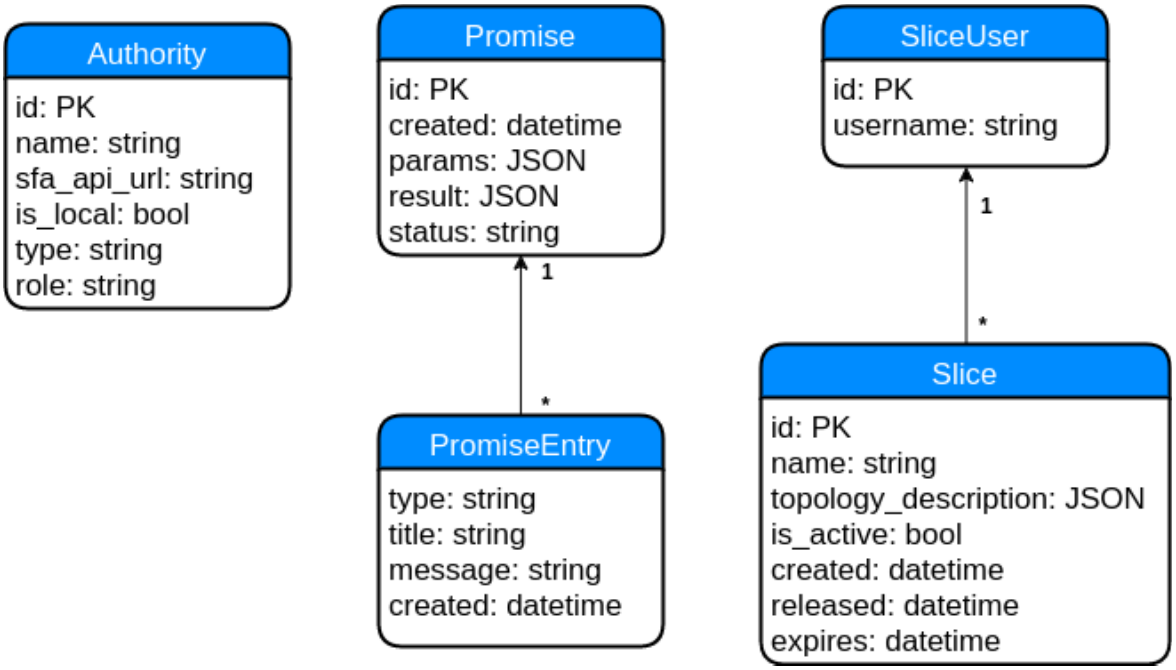
La API HTTP Rest que expone este módulo consta de métodos para creación y configuración básica de topologias. Los recursos involucrados en una topología se agrupan en una slice identificada por un nombre. Cada slice pertenece a un usuario.

Debido a que las operaciones internas que desempeña el modulo ante una llamada a la API pueden tardar mucho tiempo, se implemento un mecanismo de promises. Ante una de estas llamadas, la API responde inmediatamente, con código HTTP 504 junto con una URL donde se podrá consultar el estado de la operación. Al efectuar un GET sobre esa URL, se obtendrá status HTTP 504 mientras la operación no haya finalizado y HTTP 200 una vez finalizada. Además se obtiene información sobre el progreso de la operación en cada llamada.

Las operaciones de larga duración involucran llamadas al AggregateManager, que en esta versión se encuentra dentro de la misma solución pero podría eventualmente estar separada. Esta arquitectura esta pensada para soportar varios AggregateManagers, todos accesibles a través del mismo SliceService.

Modelo de Datos

En el siguiente diagrama se muestra el modelo de datos utilizado en SliceService y sus relaciones.



API

Como se dijo anteriormente, la SliceService integra OMF con el LAR, a través de un plugin de LabWiki (La interfaz web de OMF).

La API con la que se integra LabWiki se describe a continuación. En esta versión, las llamadas a la API no cuentan con autenticación ni autorización. Todas las llamadas devuelven contenido en formato JSON.

- `GET /authorities` Lista los authority (AggregateManagers) para los cuales se pueden solicitar recursos en este SliceService.
- `GET /authorities/{id}` Retorna información detallada sobre un authority, por ejemplo sobre los recursos que puede crear.
- `GET /promise/{id}` Retorna información sobre el estado de un trabajo asociado a la promsise.
- `PUT /users/{username}/slice/{slice_id}` Configura una topología para una slice de un usuario. Devuelve la URL de una promise.
- `GET /users/{username}/slice/{slice_id}/topology/` Devuelve la topologia asociada a la slice de un usuario.
- `GET /users/{username}/slice_membership/` Retorna las slices asociadas a un usuario.
- `POST /users/{username}/slice_membership/` Solicita la creación de una slice para un usaurio.

Comunicación con AggregateManager

SliceService cuenta con un conjunto básico de operaciones que efectúa sobre un AggregateManager en la vida de un experimento. Estas operaciones se definen a través de una interfaz que debe ser implementada para cada AggregateManager.

Las operaciones son:

- `speaks_for(user1, user2, ticket)`: Avisa al AggregateManager que el usuario *user1* habla por el usuario *user2*. Este método proviene del protocolo SFA para contemplar el caso donde existen diferentes usuarios en el SliceService y AggregateManager. El usuario *user1* envía un ticket expedido por el usuario *user2* en el AggregateManager correspondiente, que valida esta asociación. Este método es llamado previamente a la interacción del usuario *user1* con los recursos disponibles en alguno de los AggregateManagers que ofrezca el SliceService para habilitar futuras llamadas a la API en nómbre del usuario *user1*.
- `request_slice_membership(user, slice)`: Solicita al AggregateManager la membrecía de *user* a *slice*. Esta membrecía puede ser manejada de diferentes formas dependiendo de la lógica del AggregateManager. De cualquier manera, debe retornarse si la membrecía fue concedida o no, de forma de poder solicitar futuros recursos para dicha slice por el usuario y ejecutar experimentos sobre ésta.
- `request_resource(user, slice, resource_data)`: Solicita la asociación de un recurso a la slice. Los datos del recurso son, en principio, tipo y nombre, el segundo lo identifica unívocamente. De ser exitosa ésta llamada, se entenderá que el recurso esta asociado a la slice *slice* y podrá ser utilizado a partir del momento en que retorna esta la función.
- `release_resource(user, slice, resource_data)`: Libera un recurso de la slice *slice*. Se entiende que el recurso ya no va a estar disponible para su utilización sobre la slice. En la practica cada AggregateManager decidirá en base a su lógica la destrucción o no de éste.
- `list_resource_types()`: Lista los recursos disponibles en el AggregataManager.

Todas estas llamadas pueden tardar tiempo arbitrario en finalizar, dependiendo del AggregateManager, por lo que son ejecutadas asincrónicamente. Su estado y progreso es almacenado en la promise correspondiente y puede ser consultado como se menciona mas arriba.

Ejecución asincrónica

Para la ejecución asincrónica de operaciones se utilizó el framework Celery http://www.celeryproject.org/ que provee un mecanismo sencillo de ejecución de tareas distribuidas por medio de una cola de tareas. En la práctica, se ejecutan varios procesos esclavos (Celery Workers) en el mismo servidor que SliceService. Estos escuchan tareas enviadas a través una cola de mensajes por el SliceService y actualizan el progreso en la base de datos, a la que, tanto SliceService como los proceso esclavos tienen acceso.

Las tareas que reciben los procesos esclavos cuentan con un único parámetro, la clave primaria de la promise. Luego obtienen de la base de datos la promise y consultan sobre ésta el resto de los parámetros de la tarea. Durante la ejecución almacenan mensajes del progreso y los asocian a dicha promise. Al finalizar almacenan el resultado en la misma promise.

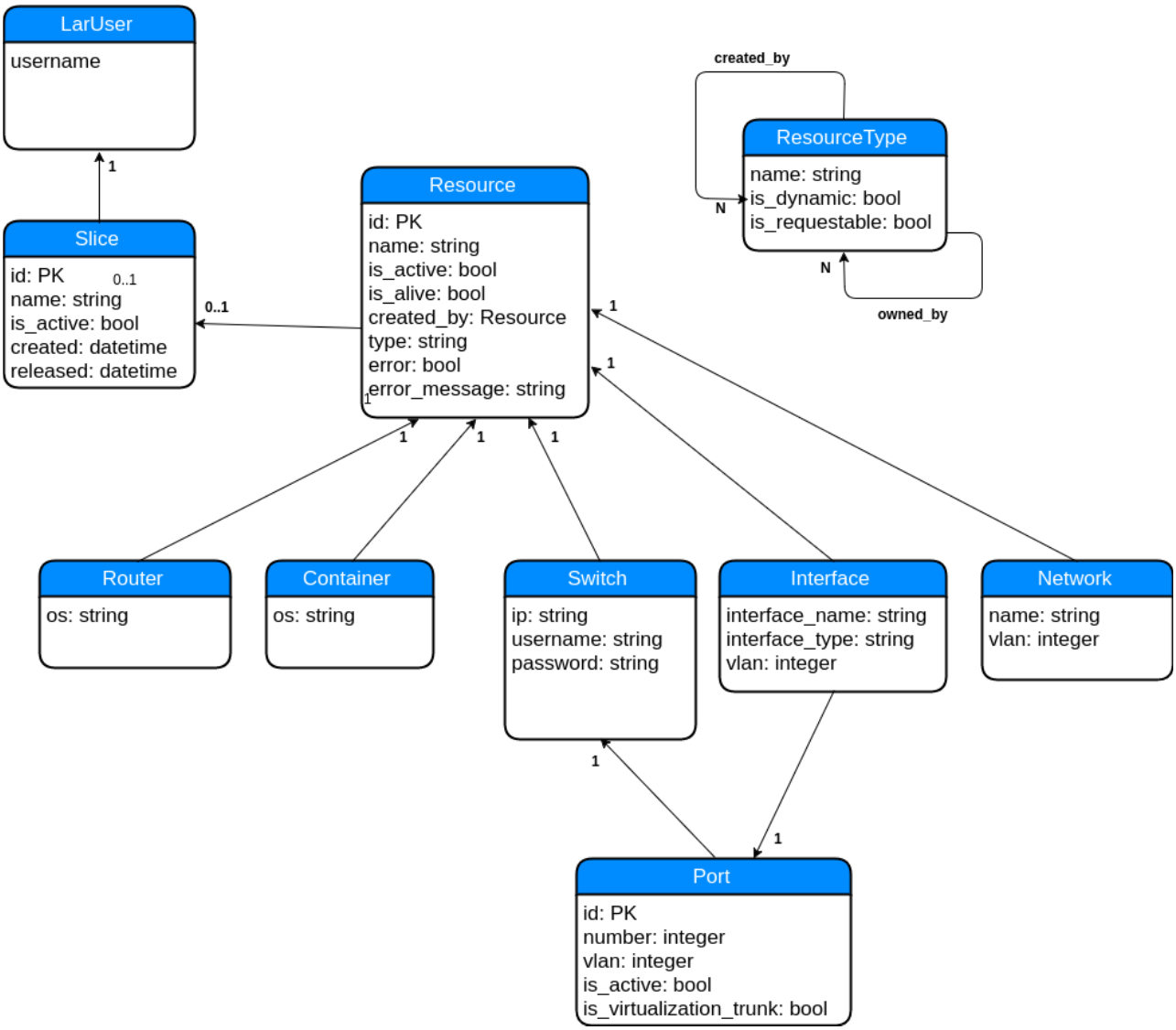
Se utilizó la misma cola de mensajes RabbitMQ que la utilizada por OMF. Para evitar posibles colisiones, RabbitMQ provee hosts virtuales que permiten crear grupos lógicamente distintos de entidades sobre el mismo servidor de mensajería.

Aggregate Manager

El AggregateManager es un módulo encargado de la gestión de los recursos de una testbed. Éste interactúa con SliceService para proveer y configurar recursos a experimentadores. La interacción se efectúa a través de una API bien definida. Como en esta etapa el SliceService y AggregateManager ejecutan bajo el mismo servidor las llamadas a la API son llamadas a funciones Python directamente. Si se encontraran en diferentes servidores, tendría que exponer una API a través de la red.

Modelo de datos

A continuación se muestra el modelo de datos utilizado en el AggregateManager y sus relaciones.



Interaz con SliceService

El AggregateManager implementa la interfaz del SliceService de la siguiente manera:

- `speaks_for(user1, user2, ticket)`: Debido a que este AggregateManager no cuenta con manejo de usuarios, lo único que se realiza en este operación es almacenar el usuario *user1*.
- `request_slice_membership(user, slice)`: Se crea, si no existe, la slice *slice* y se asocia al usuario *user*. Si la slice ya está activa o asociada a otro usuario entonces la operación falla.
- `request_resource(user, slice, resource_data)`: Si el recurso solicitado es dinámico se crea uno, sino, se verifica su existencia. Luego se asigna dicho recurso a la slice *slice*, quedando el recurso como activo. La operación falla si ya existe otro recurso con el mismo nombre y si el recurso solicitado ya está activo o, en el caso que el tipo de recurso no sea dinámico, si no existe.
- `release_resource(user, slice, resource_data)`: Libera un recurso de la slice *slice*. Si el

recurso es dinámico, se destruye, sino simplemente queda como inactivo.

- `list_resource_types()`: Lista los recursos disponibles solicitables en el AggregateManager. Estos son container, router y network. Los recursos container y network son dinámicos mientras que router es estático.

Interacción con recursos

Las llamadas `request_resource` y `release_resource` de la API con SliceService implican la interacción con los recursos del LAR. Para unificar esta interacción sobre un conjunto heterogéneo de recursos se definió una API estándar que es implementada por cada tipo de recurso y donde se tienen en cuenta las particularidades del tipo de recurso. Primero veremos los métodos que define la API y luego la explicación de los casos de uso. A cada implementación la llamamos manejador de recurso (o *resource handler*).

API para interacción con recursos

Cada tipo de recurso debe implementar los siguientes métodos:

- `create(resource, params)`: Crea un recurso en la infraestructura y actualiza la base de datos dependiendo del resultado de la creación.
- `membership(resource, slice)`: Asigna el recurso *resource* a la slice *slice*. Esta operación normalmente no requiere interacción con los recursos fisicos.
- `release(resource, slice)`: Libera el recurso *resource* de la slice *slice*. Si el recurso es dinámico se puede destruir.
- `request(resource, property_name)`: Solicita una propiedad al recurso. El valor de dicha propiedad puede estar almacenado en la base de datos o puede ser solicitado directamente al recurso físico.
- `configure(resource, property_name, value)`: Configura la propiedad de un recurso.

Las operaciones `create()`, `membership()` y `release()` realizan modificaciones necesariamente sobre la base de datos, ya que agregan o destruyen recursos, o modifican su disponibilidad. El código que realiza dichas actualizaciones es siempre el mismo a menos del modelo (o tabla en la base de datos) donde se realiza la alta/baja/modificación. Por este motivo se debe especificar también el modelo de manera que todas estas operaciones están ya implementadas en la clase abstracta de la que heredan todos los demás recursos. Para garantizar la consistencia de los datos, la actualización de los datos de recursos se realizan en transacciones SQL.

Utilización de la API

Al recibir una petición de *request* o *release* sobre un recurso, se verifica la tabla *ResourceType* (ver modelo de datos) para determinar como proceder. Dicha tabla define los tipos de recurso, si es dinámico y solicitable por los usuarios. Al recibir un *request* se verifica si el tipo de

recurso pedido es dinámico, de ser asi, se debe llamar al método `create()`. Luego se asigna el recurso a la slice con `membership()` donde se marca como activo. Al recibir un *release*, se llama al `release()` del manejador de recurso. Si el recurso es dinámico, se destruye y se marca como no disponible. Finalmente se desasocia de la slice y se marca como inactivo.

Implementacion API

A continuación se describirá la implementación de la interfaz de recurso para cada tipo de recurso solicitable. Que sea solicitable quiere decir que es posible su reserva y utilización por parte de un usuario.

La implementación de la operación `membership()` es la por defecto en todos los recursos.

> En esta primera etapa del sistema, no se interactua con los recursos una vez creados por lo que las operaciones de `request()` y `configure()` no esta implementadas ni se utilizan.

Router

Este recurso es de tipo estático ya que no es posible crearlo ni destruirlo. Representa un router físico, ejecutando OpenWRT. Por lo tanto la implementación de la interfaz de recurso solo actualiza los datos sobre la base de datos.

Container

Los *containers* son recursos dinámicos. Un recurso container representa contenedor LXC, con cierta imagen cargada. En esta versión del sistema la imagen es fija, definida por un administrador.

Ante un mensaje `create()`, se creará y pondrá en funcionamiento un contenedor con el nombre que se especifique como parámetro de la función. Como este nombre luego se asigna al *hostname* del contenedor creado, los nombres de los recursos deben ser *hostnames* validos.

La operación `release()` detiene el contenedor. Por defecto luego se destruye aunque este comportamiento puede ser fácilmente modificable a futuro.

Network

Un recurso de red representa la interconección entre varios nodos ya sean routers o containers. Para crear un recurso red ya deben estar creados todos los nodos involucrados en dicha red.

Al crear una red, se recibe ademas del nombre, que a los efectos prácticos es simbólico, una

lista de recursos que pertenecen a la red. La creación de la red implica la creación de una VLAN y la conexion de todos los nodos a dicha VLAN. La secuencia de pasos es la siguiente:

1. Obtención de un numero de VLAN disponible. Se selecciona un número al azar entre 2 y 4094 excluyendo las VLANs ya utilizadas. Para prevenir problemas de concurrencia el numero de VLAN no debe repetirse entre redes activas. De ocurrir una colicion en el valor sorteado se intentará obtener otro. Se asume que el switch es de uso exclusivo del sistema por lo que las VLANs ya utilizadas son las reflejadas en la base de datos del AggregateManager.
2. Se configuran las interfaces de los containers. Para cada contenedor de la lista de nodos, se debe crear una nueva interfaz y luego se debe conectar a la VLAN de la red. La creación de las interfaces se realiza a través del manegador del recurso interfaz. Una vez creada se debe conectar la interfaz a la VLAN, en la práctica esto se traduce a conectar la interfaz virtual recientemente creada a la interfaz trunk del host donde ejecutan los contenedores (Hypervisor LXD). Observar que las interfaces de los contenedores son dinámicas y estos pueden estar conectados a un numero arbitrarios de redes.
3. Configurar interfaces de routers. Para cada router de la lista de nodos, se obtiene una de sus interfaces existentes inactivas y se guarda el numero de puerto del switch al que está conectada físicamente y a qué switch. Luego para cada switch, se configuran todos los puertos de la iteración anterior a la VLAN de la red. Por limitaciones en la forma de interactuar con el switch no es posible la configuración unitaria de sus puertos.

Los pasos 2 y 3 se deben efectuar como uno solo y de manera transaccional. Es decir, si no ocurren errores la red debe quedar configurada, sino se deben deshacer todas las configuraciones intermedias previas al error. Para lograr esto, se almacena el estado del avance de la configuración (interfaces conectadas agrupadas por nodo), de manera que si ocurre un error se recurre a dicho estado para realizar el desmontaje de la parcialmente establecida red.

Otras Consideraciones

Limitaciones en la interacción con el Switch

Los switchs utilizados en la infraestructura LAR no soportan otro modo de interacción que a través de su interfaz web. Esto es una limitación desde el punto de vista de automatizar las operaciones de creación y asignación de VLANs ya que no es API. Además, la web de estos utiliza tecnologías deprecadas y un obsoleto cifrado del lado del cliente que solamente dificultó aún más la implementación de un módulo que interactúe de forma programable con ellos. Mas detalles se encuentra en la sección *Switch Controller*.

Información del estado de los recursos

El AggregateManager también ofrece una web donde se puede visualizar los recursos

existentes, su estado, el usuario al que están asignados y la slice. Dicha web es pública y provee información de solo lectura.

Switch Controller

Este controlador consiste en un módulo desarrollado en Python, que interactúa con la interfaz web expuesta por el switch Cisco modelo SF200-48P para la configuración de VLANs y la asignación de las mismas a los distintos puertos. Se utiliza la biblioteca unirest para la realización de requests HTTP, y PyCrypto para las operaciones criptográficas relacionadas con la autenticación.

Parámetros

Éste controlador necesita de los siguientes parámetros de configuración:

Nombre	Descripción
SWITCH_USER	Usuario para ingresar al switch a través de la interfaz web.
SWITCH_PASSWORD	Contraseña del usuario web.
SWITCH_IP	Dirección IP en la que el switch expone su interfaz web.

Estos parámetros son obtenidos de la base de datos, de la tabla correspondiente al recurso Switch.

Operaciones

- `get_url()`

  El primer paso necesario es obtener la URL detrás de la cual se encuentra el panel de administración del switch. En las pruebas realizadas esta URL siempre fue `/cs97b62215/`, sin embargo desconocemos si este valor puede cambiar entre un dispositivo y otro, por lo que optamos por obtener la URL de forma dinámica.

  Al realizar un request a la raíz del servidor web, se realiza un doble redirect como se muestra a continuación:

```
GET / HTTP/1.1
Host: 192.168.0.1

HTTP/1.1 302 Redirect
Server: GoAhead-Web
Date: Tue Nov 03 04:30:01 2015
Connection: close
Pragma: no-cache
Cache-Control: no-cache
```

```
Content-Type: text/html
Location: http://192.168.0.1/cs97b62215/
```

```
GET /cs97b62215/ HTTP/1.1
Host: 192.168.0.1

HTTP/1.1 302 Redirect
Server: GoAhead-Web
Date: Tue Nov 03 04:30:10 2015
Connection: close
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Location: http://192.168.0.1/cs97b62215/config/log_off_page.htm
```

Como la biblioteca unirest sigue automáticamente los redirects, no es posible obtener directamente la URL intermedia. Sin embargo es simple obtenerla parseando la URL final.

- `login(user, password)`

Una vez obtenida la URL se debe resolver el problema de la autenticación. A diferencia de un formulario de login tradicional, en el que se envían el usuario y la contraseña directamente al servidor, en estos dispositivos éstos valores son cifrados con RSA utilizando PKCS#1v1.5.

Esta operación:

1. Obtiene la clave pública RSA del switch, disponible en el recurso `/config/device/wcd?{EncryptionSetting}`
2. Calcula el hash SHA1 del texto a cifrar
3. Cifra el texto junto con el hash calculado, de acuerdo a lo indicado en el estándar
4. Codifica el texto cifrado en hexadecimal
5. Envía el resultado al servidor
6. Obtiene una cookie con la cual realizar el resto de las operaciones.

La función retorna la cookie obtenida.

- `get_vlan_list()`

Esta operación obtiene la lista de VLANs configuradas. Para esto realiza un request a `/wcd?{VLANGlobal}`. La respuesta consiste en un XML dentro del cual se incluye la lista de VLANs.

La lista se compone de valores separados por comas, donde cada valor es o bien un número de VLAN o un rango de valores.

La función retorna la lista de VLANs obtenidas.

- `create_vlan(vlan, name)`

  Esta operación crea una VLAN con nombre `name` y número de VLAN `vlan`. La forma de lograrlo es enviando un POST con estos parámetros al servidor, así como un conjunto de parámetros fijos. El orden en que se envían los parámetros es relevante, por lo que se utiliza un diccionario ordenado para almacenarlos.

- `config_ports(vlan, port_list)`

  Configura los puertos especificados en `port_list` con la VLAN `vlan`. Para esto configura cada puerto individualmente utilizando la función auxiliar `config_port(port, vlan)`, y luego actualiza la lista de puertos asociados a dicha VLAN.

  Para esto último se envía al servidor un array de 1008 bits, donde un 1 en la posición i de dicho array indica que el puerto i pertenece a la VLAN configurada. Este array se envía como un string de 252 caracteres hexadecimales.

LXD Controller

Este controlador consiste en un módulo desarrollado en Python, que interactúa con la API de LXD para la creación de containers, y se conecta por ssh al servidor para configurar las interfaces a nivel de sistema operativo.

Parámetros

Los siguientes parámetros deben ser configurados para utilizar el controlador LXD. Ellos son configurados en el archivo de configuración de la solución GestorLAR, menos el valor de SERVER_TRUNK_IFACE, que es configurado en las configuraciones almacenadas en la base de datos.

Nombre	Descripción
SERVER_URL	Dirección IP, protocolo y puerto en el que escucha la API de LXD. Ej. https://192.168.20.15:8443.
SERVER_PASS	Contraseña configurada en el hipervisor, necesaria para establecer una relación de confianza entre cliente y servidor.
SERVER_CRT	Certificado utilizado para la autenticación luego de establecida la relación de confianza.
SERVER_KEY	Clave privada correspondiente al certificado.
SERVER_SSH_IP	Dirección IP en la que corre el servicio SSH.
SERVER_SSH_USER	Usuario utilizado para la configuración de interfaces virtuales en el servidor. Este usuario debería tener los mínimos permisos

Nombre	Descripción
	necesarios, como se especifica en el manual de instalación en el Apéndice 3.
SERVER_SSH_PASS	Contraseña del usuario SSH.
SERVER_TRUNK_IFACE	Nombre de la interfaz de red conectada al enlace trunk. Sobre esta interfaz se crean las interfaces virtuales correspondientes a cada VLAN.

Operaciones

- `connect()`

  Establece una conexión con la API de LXD, utilizando autenticación por clave pública en caso de existir una relación de confianza, o estableciendo una relación de confianza utilizando la contraseña en caso contrario.

  Obtiene la lista de VLANs ya configuradas en el servidor, parseando las interfaces de nombre interface.vlan-id, donde interface es la interfaz conectada al enlace trunk.

  Por último, obtiene la lista de containers creados utilizando la API de LXD.

- `create(name, image)`

  Crea un container de nombre `name`, utilizando como base la imagen especificada en `image`. Esta imagen debe estar importada en el repositorio local del hipervisor.

  Retorna el objeto creado.

- `delete(name)`

  Elimina el container de nombre `name` en caso de existir.

  Retorna True en caso de éxito, o False en caso contrario.

- `start(name)`

  Inicia el container de nombre `name` en caso de existir y encontrarse detenido.

  Retorna True en caso de éxito, o False en caso contrario.

- `stop(name)`

  Detiene el container de nombre `name` en caso de existir y encontrarse iniciado.

  Retorna True en caso de éxito, o False en caso contrario.

- `create_vlan(vlan_id)`

  Crea en el servidor una interfaz virtual asociada a la VLAN `vlan_id`. Para esto ejecuta en el servidor los siguientes comandos, donde `trunk_interface` es la interfaz conectada al enlace trunk:

  ```
  sudo vconfig add trunk_interface vlan_id
  sudo ip link set trunk_interface.vlan_id up
  ```

- `connect_to_vlan(name, interface, vlan_id)`

  Conecta la interfaz interfaz del container de nombre `name` a la interfaz virtual del servidor configurada con la VLAN `vlan_id`. En caso de no existir la interfaz en el servidor, se crea utilizando la operación `create_vlan`.

Notas de desarrollo

Todos los módulos desarrollados fueron escritos en lenguaje Python por su versatilidad y amplia disponibilidad de bibliotecas. Ademas de ser un lenguaje con el que los participantes del proyecto nos sentimos cómodos.

Diseño general de modulos y patrones de diseño

La mayor parte del código fuente se encuentra dentro de la soluciones AggregateManager y SliceService. Cada una de estas cumple ampliamente con las recomendaciones de diseño del framework Django lo que nos asegura que el diseño es fuerte, al ser Django un framework largamente utilizado y documentado, y además una fácil iniciación por parte de un tercero en el código.

El patrón de diseño mas utilizado fue el de Interfaz junto con fabrica. Se utilizaron interfaces para acceso uniforme y bien definido tanto para recursos como para acceder potencialmente varios AggregateManager. Ademas de acceso uniforme a agentes heterogéneos, una interfaz permite la fácil integración de nuevos agentes. En el caso de los recursos, basta agregar una implementación y su modelo correspondiente en la base de datos. En el caso de un AggregateManager basta solo con la implementación de dicha interfaz e ingresar los datos del AggregateManager a una tabla. Luego de implementada, cada interfaz llama a un método que la registra con un nombre único. A partir de entonces, se tendrá acceso a dicha interfaz por su nombre, a través de una fábrica. En caso de los recursos el nombre es tipo de recurso. En el caso de los AggregateManagers el nombre es que se ingrese en la respectiva tabla de AggregateManager dentro del SliceService.

Otros modulos desarrollados

DbSettings

La manera tradicional de configurar una solución en Django es mediante un archivo de configuración. Esto tiene la desventaja de que hay que reiniciar la aplicación web al cambiar un valor. Como solución se desarrolló un modulo (como aplicación de Django) que permite el almacenado de valores en la base de datos con el objetivo de ser utilizados como parámetros de configuración, que pueden cambiar sin reiniciar la aplicación web.

Cada configuración se identifica con clave de tipo cadena de caracteres y almacena un valor de tipo primitivo: entero, flotante, booleano o string. Para acceder a dicho valor dentro desde el código basta utilizar la función: `dbsettings.get(key)`.

Messages

La ejecución asincrónica de tareas con el mecanismo de promises permite ejecutar una tarea y consultar su estado en tiempo real a través de la API del SliceService. Un problema que surgió fue la imposibilidad de reportar el progreso cuando la tarea ejecutando llama a un API de terceros, como el AggregateManager.

Se creo el módulo *messages* que funciona como contenedor de mensajes global por hilo en espacio de usuario. Recordar que la implementación de CPython no utiliza threads del sistema operativo. Este módulo se utilizó en el AggregateManager para loguear el progreso realizado ante una llamada a la API de éste con SliceService.

La API de este modulo funciona de la siguiente manera:

1. Un método o función de Python se decora con el decorador `@messages.with_log_messages` lo que permite que se cree el contenedor de mensajes previamente a la llamada a dicha función y se destruya luego.
2. Durante la ejecución de dicha función y en cualquier otra función que ejecute por un llamado directo o indirecto se puede agregar mensajes al contenedor utilizando las llamadas `messages.{debug(), info(), warning(), error()}`.
3. Al finalizar, la función del punto 1. se debe llamar a método `messages.get_messages()` para obtener todos los mensajes serializados y retornarlos.

También se provee un decorador adicional, `@messages.message_transaction` que permite que los mensajes agregados en la ejecución de una función se hagan en una transacción. Osea, si ocurre una excepción en la ejecución, ningún mensaje agregado en dicha función quedará en el contenedor.

El nivel por defecto es INFO y puede cambiarse en tiempo de ejecución por medio de `messages.set_level()` o cambiar el nivel por defecto en el archivo de configuración.

Apéndice 2: Manual de uso

Se presenta en este apéndice los manuales de uso del Gestor LAR para usuarios de la testbed y administradores.

Manual de usuario

El sistema permite la creación de topologias que involucran redes que interconectan nodos y posterior ejecución de experimentos utilizando OMF. El punto de entrada es una interfaz web, llamada LabWiki, desde donde se realizan estas acciones.

Mediante LabWiki se pueden crear y configurar topologías así como ejecutar y observar los resultados de un experimento.

Un experimento OMF actúa enviando instrucciones a los hosts cuando se quiere que estos cambien su estado. Las instrucciones pueden ser: ejecutar aplicaciones, configurar interfaces o rutas, y ejecución de comandos. Las aplicaciones que utilicen OML y que sean configuradas dentro del experimento para tomar mediciones, envían datos periódicamente a un servidor de recolección. Estos datos se despliegan en LabWiki en tiempo real y luego pueden ser exportados por el experimentador.

Detalles de la testbed

Las topologías se crean a partir de recursos. Los recursos pueden ser dinámicos o estáticos. Los dinámicos son creados al momento en que un usuario solicita la topología, mientras que los estáticos ya se encuentran creados y solamente son agregados o quitados de una topología.

Una topología esta compuesta por recursos de tipo host y red. Los recursos red son dinámicos así como los hosts de subtipo *container*. Los únicos recursos estáticos son hosts de subitpo *router*. Todos los recursos se identifican por su nombre, por lo tanto para utilizar un recurso estático es necesario conocer dicho valor. Los recursos disponibles y su tipo se encuentra publicado en una web de acceso público, llamada *GestorLAR*, a la que se puede acceder para conocer la disponibilidad y nombre de todos los recursos existentes en el LAR.

Recursos

Red

Este tipo de recurso representa una red a la cual se conectan interfaces de los hosts. Este recurso no se utiliza explícitamente durante la ejecución de un experimento con OMF. La razón por la que es necesario que tenga un nombre es para identificar a qué redes se

conectaron las interfaces de los hosts al momento de configurar la topología.

Host

Este recuro representa un host dentro de la red, aunque no necesariamente debe estar conectado una. Los dos subtipos existentes *router* (OpenWRT) y *container* (Ubuntu 16.04) son entidades reales y virtuales respectivamente. No necesariamente un router está encargado de routear trafico ni un contenedor está necesariamente en los extremos de la red. Ambos tipos pueden jugar ambos roles y se distinguen por sus características propias, como se muestra en la siguiente tabla.

Característica	**Router**	**Container**
Sistema operativo	OpenWRT	Ubuntu 16.04
RAM	64MB	*
CPU	1CPU ~100Mhz	*
Disk	8MB	*
Interfaz gestión	eth0	mgmt

* Estos valores dependen de la carga puntual del hipervisor. Pueden considerarse suficientes para la ejecución de cualquier experimento con OMF y muy superiores a las caracteristacas de un router.

Red de gestión

Los hosts siempre se encuentran conectados a la red gestión: 10.0.0/23. Dicha red no debe ser utilizada durante los experimentos ni mucho menos modificar las interfaces conectadas a ella, ya que a través de esta se envían instrucciones para configuración y experimentación.

¿Qué se necesita para comenzar?

El usuario interactúa con la tesbed únicamente a través de la web. Existen diferentes servicios al que el usuario debe tener acceso para hacer un uso pleno de la infraestructura.

Servicio	**Función**
LabWiki	Experimentación y configuración de topologías
GestorLAR	Listado de recursos activos/disponibles.
FileServer	Servidor donde se pueden descargar archivos varios.

Las URLs de éstos deben ser provistos por un administrador o encargado del sistema.

LabWiki

LabWiki es una aplicación web donde es posible planear, ejecutar, observar y analizar series de experimentos. Permite realizar todos los pasos del ciclo de vida de un experimento, documentando cada paso transcurrido.

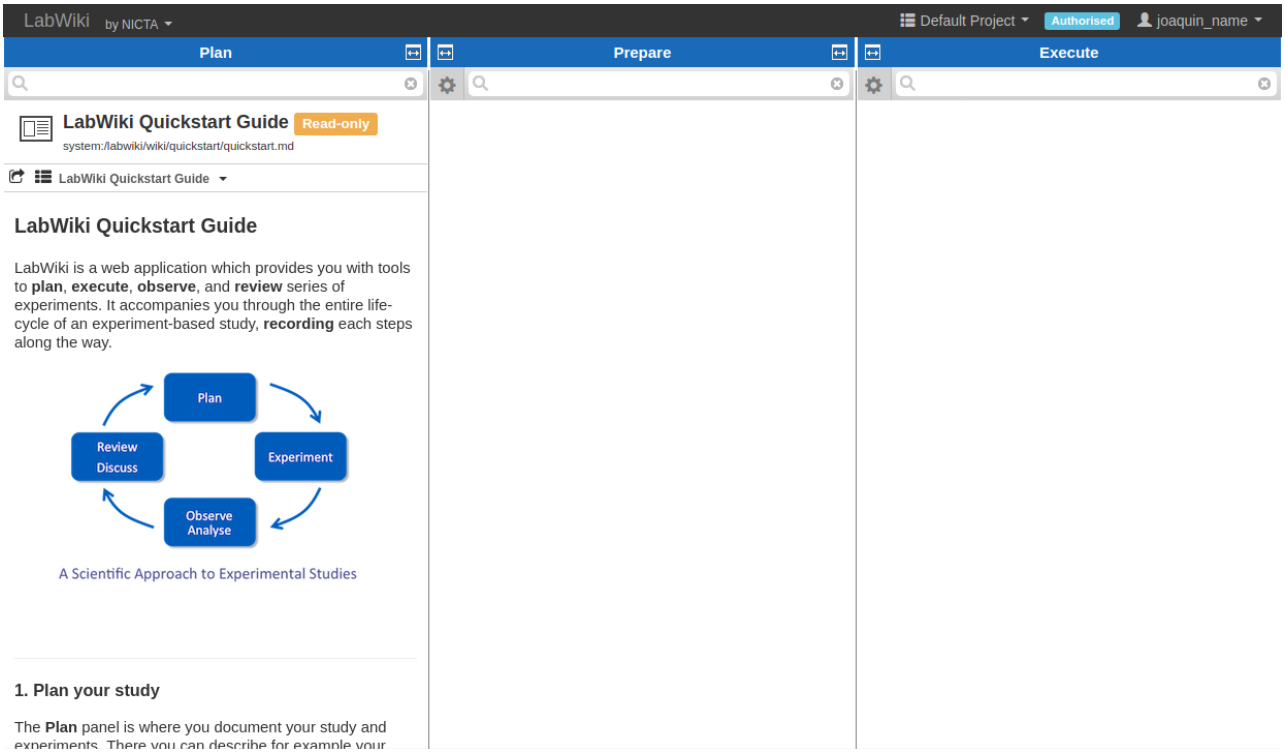
Iniciar sesión

Una vez que se ingresa a la URL de la interfaz, se debe iniciar sesión ingresando usuario y contraseña.

> En la versión actual la autenticación no se da efectivamente contra ningún *backend* y todos los usuarios son aceptados. La contraseña es completamente ignorada. Los usuarios se identifican únicamente por su *username*.

¿Que se puede hacer?

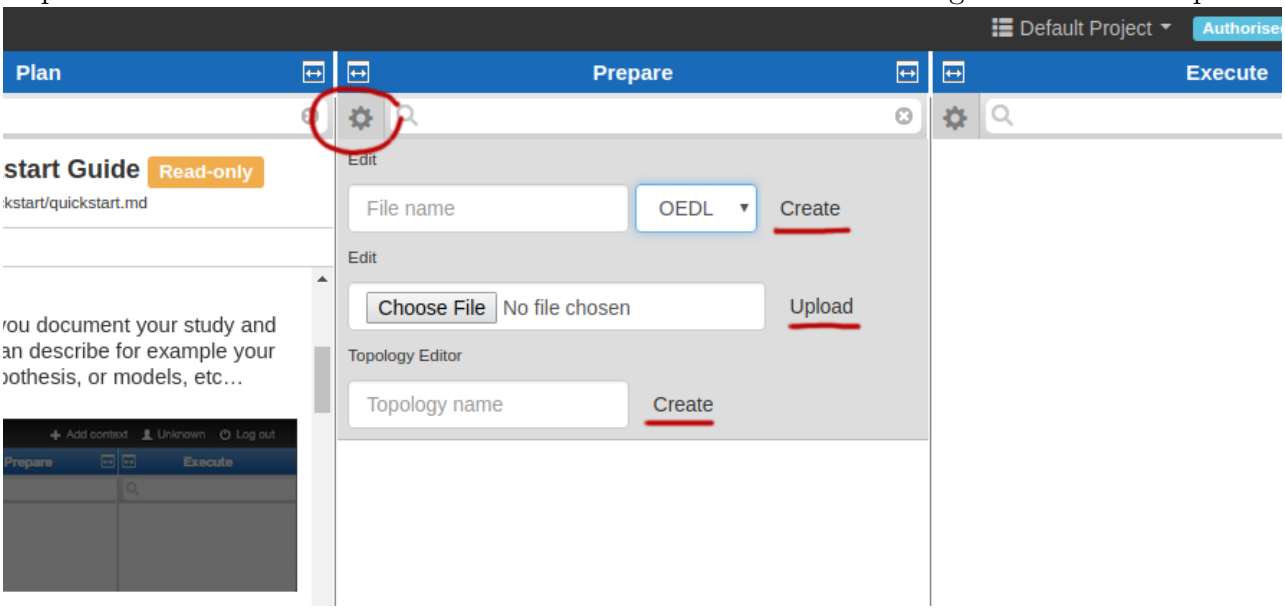
Dentro de LabWiki se despliegan tres columnas. A la izquierda la de planificación donde se podrá desplegar documentación, escrita en formato Markdown. Al centro, la columna de preparación, donde se crean y editan los archivos, tanto de documentación como los que representan acciones sobre la testbed. Finalmente, la columna de la derecha es la de ejecución, donde se ejecutan las instrucciones de los archivos de la columna central.



Creación de documentos

LabWiki es un repositorio de documentos. La creación de estos se lleva a cabo en el panel

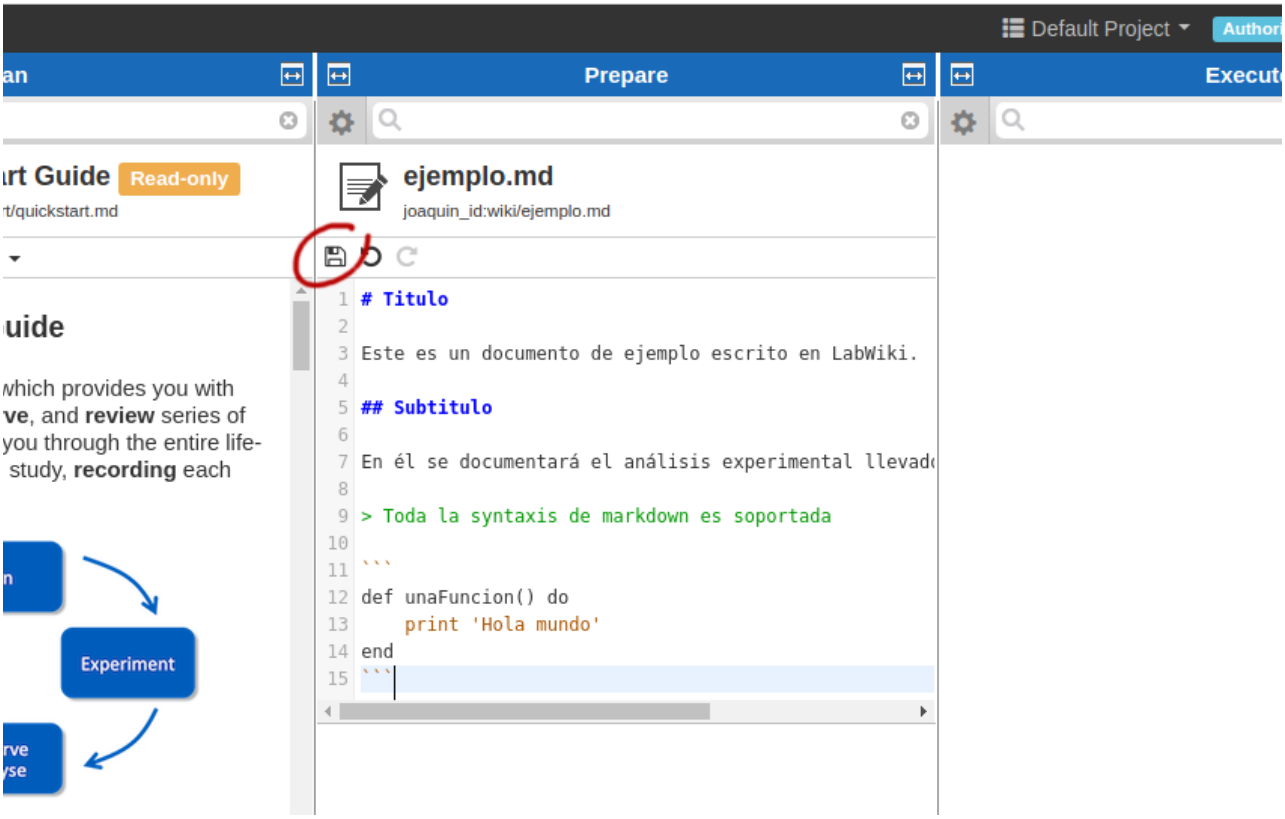
central. Dando click en el engranaje superior, se abrirá un panel donde se puede crear o cargar un archivo. Los archivos que se pueden crear son de tres tipos: *OEDL* para especificar experimentos, *Wiki* para documentación y *topology* para especificar una topología. También se puede subir un archivo desde la PC del usuario. El panel de creación y las acciones disponibles se muestra en la siguiente captura.



Los documentos creados se almacenan en un repositorio interno por usuario. Luego de creados no pueden ser destruidos y pueden ser accedidos en cualquier momento por el mismo usuario que los creó. Mientras estén en el panel central, los documentos no implican futuras acciones sobre la testbed por lo que pueden ser manipulados libremente.

Documento Wiki

Al crear un archivo de tipo Wiki, se podrá escribir documentación en la columna central en formato Markdown, con el correspondiente resaltado.



> Luego de cada edición, los cambios deben ser manualmente guardados.

Documento OEDL

Un documento de tipo OMF Experiment Description Language especifica el comportamiento de los recursos OMF en un experimento. Dicho lenguaje está basado en el lenguaje Ruby, con una API para interactuar con los recursos. El editor de este tipo de archivo tiene el resaltado para el lenguaje Ruby y su creación/edición y guardado es igual al tipo Wiki.

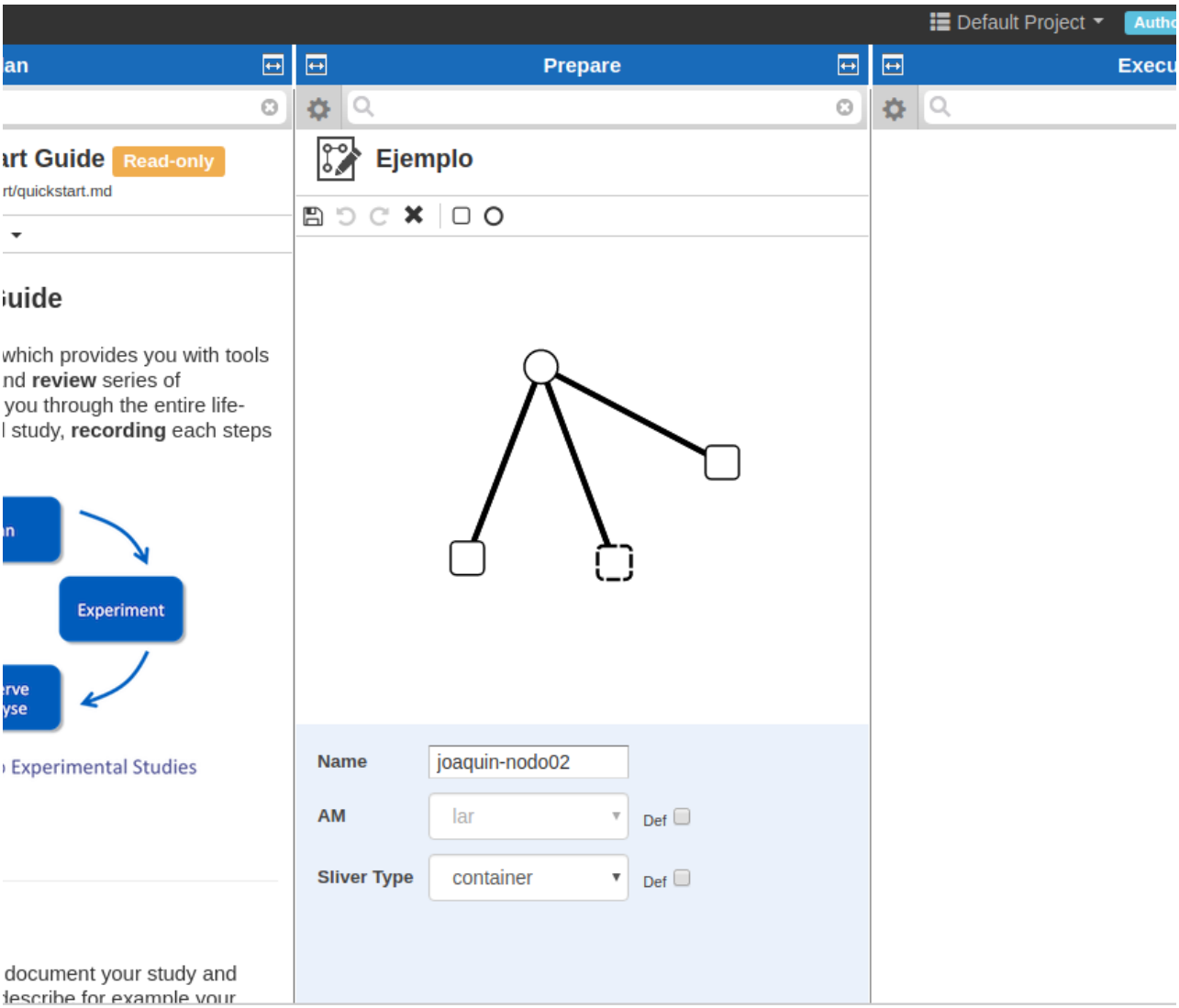
Para ver la especificación de este lenguaje dirigirse a la sección *Escribiendo experimentos con OEDL* más adelante en este documento.

> Atención: el editor no detecta errores de sintaxis ni posibles errores de programación. De existir, estos se revelarán al momento de ejecutar el experimento.

Topología

Un documento de especificación de una topología es un simple archivo de texto, pero se renderiza como un grafo en el editor de LabWiki. La manera de creación y almacenado de éste es igual a los anteriores pero cambia su forma de edición, como se ve en la siguiente

imagen.



Existen dos entidades en este tipo de topologías: hosts y redes, representados por los cuadrados y círculos respectivamente. Las aristas del grafo representan la conexión de hosts a redes, por lo que no pueden existir aristas entre dos nodos del mismo tipo.

Al seleccionar un nodo, se podrán editar sus propiedades en la parte inferior del panel. Todos los nodos deben tener un nombre único que los identifica en la tesbed. Dicho nombre debe estar en conformidad con la especificación de nombres de domino (DNS), ya que son utilizados como *hostname* en los recursos de tipo host.

Además del nombre, los recursos tiene otras propiedades que dependen de su tipo. Para los hosts se debe seleccionar el AggregateManager (AM) que los gestiona (actualmente existe uno solo y es el LAR) y el Sliver Type que representa un subtipo. Los subtipos disponibles son *router* y *container*. El primero representa un router doméstico ejecutando el sistema operativo OpenWRT y el segundo un contenedor Linux con sistema operativo Ubuntu 16.04.

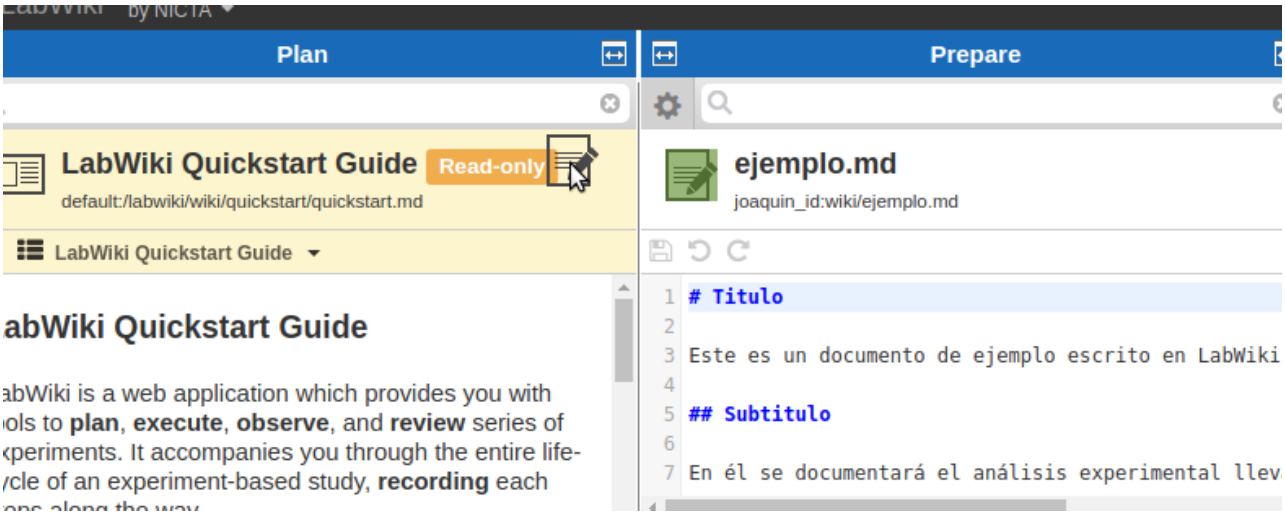
Los parámetros de los nodos de tipo red no son obligatorios. De no ingresar un nombre se

autogenerará uno al momento de solicitar la topología. El parámetro netmask es ignorado en esta versión del sistema.

> Existe también la posibilidad de seleccionar en el grafo las interfaces de los hosts y editar sus propiedades. Sin embargo en la versión actual no se toma en cuenta esta información, solamente el hecho de exista un enlace entre el host y la red. Ademas la ip de las interfaces puede ser configurada fácilmente durante la ejecución del experimento.

Acciones sobre documentos

Una vez editado un documento sobre la columna de preparación, se podrá ver el resultado de su interpretación arrastrando y soltando el icono del documento ubicado dentro de la columna central a la columna de su izquierda o derecha dependiendo de su tipo, como se muestra en la siguiente imagen.



El caso mas sencillo es el del tipo de documento *wiki* ya que no requiere futuras acciones. Este es el único tipo de documento que se interpreta en la columna izquierda (de Planificación). Las acciones a partir de documentos *oedl* y topologías se describen en detalle mas adelante.

Búsqueda

En la parte superior de cada columna se ubica un buscador de documentos donde se podrá buscar tanto documentos creados como los provistos por el LAR, en modo solo lectura, y las acciones sobre estos como experimentos o configuración de topologías.

Dependiendo de la columna en que se busque, puede ocurrir que se desplieguen resultados de archivos que no fueron creados por el usuario. Estos son archivos de solo lectura, utilizados

para dar acceso a información que puede ser útil al experimentador.

Estos son algunos de los archivos de solo lectura:

Panel	Archivo	Función
Planificación	quickstart.md	Quickstart de LabWiki
Preparación	*_oml2.rb	Definición de aplicaciones en OEDL que utilizan OML, para ser utilizadas por experimentadores.

Ejecución

Al arrastrar un documento de tipo *oedl* al panel de Ejecución se desplegará la pantalla de configuración del experimento, donde se podrán seleccionar su nombre, propiedades, proyecto y la slice (topología).

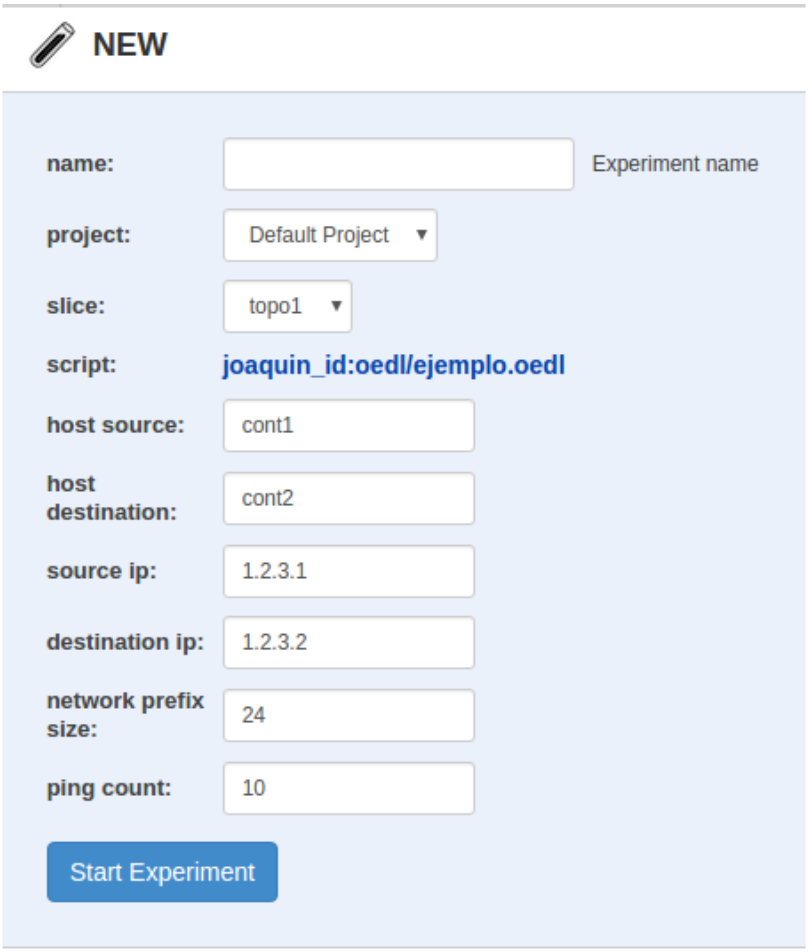
> En esta versión del sistema, el campo slice y proyecto son ignorados. Al ejecutar un experimento, el controlador de experimentos asume que los recursos están disponibles, de no estarlo el experimento finaliza por *timeout*.

Configuración del experimento

Ningún campo del formulario de configuración es requerido. Al nombre ingresado, si se ingresa, se le agrega como sufijo un timestamp, aunque es recomendable ingresar algo representativo para posterior búsqueda de el resultado de su ejecución.

Un experimento se puede parametrizar durante su escritura, para que al configurarlo se desplieguen los parámetros (llamados propiedades en LabWiki) configurables. De esta manera se logra la obvia ventaja de evitar rescribirlo para sucesivas ejecuciones con cambios en sus parámetros.

En la siguiente imagen se muestra la configuración de un experimento que obtiene datos del RTT (Round Trip Time) entre dos hosts utilizando la herramienta ping.



Ejecución del experimento

Luego de configurado, el experimento puede ser agendado para su ejecución clickeando el botón **Start Experiment**. El experimento es enviado a un planificador que lo ejecutará en algún momento. La pantalla de ejecución cambiará a la vista *ejecución de experimento* donde se podrá observar el estado actual y, cuando comience su ejecución, los mensajes y gráficas con los datos que se van generando, en tiempo real.

> Nota: En la versión actual de LabWiki la actualización de datos en tiempo real funciona por un tiempo limitado, generalmente los primeros segundos de ejecución, luego hay que refrescar la página para observar las actualizaciones.

Dentro de la vista de *ejecución de experimento*, se tendrá la posibilidad de abortar el corriente en cualquier momento, ya sea durante su ejecución o mientras esta pendiente.

Exportar datos

Una vez finalizada la ejecución, los datos generados de logs y de aplicaciones que utilicen OML podrán ser exportados en formato CSV, clickeando el botón de **Dump** en la parte

superior del panel de ejecución, dentro de la vista *ejecución de experimento*.

Consideraciones

1. Debido a que el planificador de la versión actual es FIFO, los experimentos tienen un tiempo de ejecución máximo de 6 minutos.
2. Luego de la ejecución de un experimento, no se realiza un *rollback* al estado inicial de los hosts. Sino que mantienen el estado como al final de la ejecución. La única manera de volver a un estado inicial es recreando la slice.

Solicitud de topología

Al arrastrar un documento de tipo *topology* al panel de ejecución, se mostrará la vista de configuración de una slice. Solo se requiere el nombre de la slice. Se puede crear una slice nueva o asignar la topología a una slice existente, pero inactiva.

> Nota: Si se selecciona una slice activa dentro de las existentes, no se destruirán los recursos como muestra la advertencia en LabWiki. Sino que al intentar reutilizarla se mostrará un mensaje de error. Al momento, no es posible distinguir las slices activas dentro de la lista de slices existentes.

El proceso de *set up* de una topología consta de dos partes. Primero la solicitud de la *slice*, y luego la configuración de los recursos.

La primer parte, consiste en solicitar una slice que debe ser de nombre único, para todos los usuarios. Este paso falla en el caso de que ya exista una slice con dicho nombre en estado activo o la slice existente esté asociada a otro usuario.

El pasaje al segundo paso es automático. La configuración de los recursos puede demorar varios minutos dependiendo de la complejidad de la topología. A medida que estos se configuran en la testbed, se actualiza la información dentro del panel de ejecución de LabWiki, que ahora se encuentra en la vista *monitoreo de topología*.

Se debe prestar particular atención al log de monitoreo de la topología ya que ahí se informa la asociación de interfaces de hosts a redes.

GestorLAR

GestorLAR es el servicio encargado del mantenimiento de los recursos del LAR. Este expone una interfaz web donde se listan todos los recursos existentes en la testbed e información básica sobre ellos.

FileServer

El servidor de archivos provee acceso de solo lectura a diversos archivos que pueden ser utilizados por los usuarios. Expone una interfaz web con listado de archivos por directorio.

Escribiendo experimentos con OEDL

OEDL es un lenguaje de dominio especifico para la descripción de la ejecución de un experimento. Está basado en el lenguaje Ruby, y provee su propio conjunto de comandos y declaraciones orientados a la ejecución de experimentos. Como nuevo usuario, no es neceario conocer Ruby para utilizarlo, pero si es recomendable tener conocimientos básicos de programación.

Una descripción de experimento en OEDL esta compuesta de dos partes:

1. La declaración de los recursos que se utilizarán, como hosts, y alguna configuración que se desea aplicar sobre estos recursos.
2. La definición de eventos que deseamos distinguir, y sobre los cuales queremos ejecutar ciertas tareas cuando ellos ocurran.

Sintaxis

- Todos los identificadores especificados por el usuario, deben ser identificadores válidos de Ruby. Es decir, ser verificados por la siguiente expresión regular: `/[\$\@\w][_\w]+[\!_=\?]?/i`, o más fácilmente recordable, letras mayúsculas, minúsculas y guión bajo.
- Los comentarios comienzan con #.

defProperty & property & ensureProperty

Las propiedades son los parámetros configurables de un experimento. Estas propiedades se definen y acceden con las siguientes sentencias.

- `defProperty(property_name, default_value, description)`: Esta función define una nueva propiedad, de nombre `property_name`, quien debe ser un identificador de Ruby válido.
- `ensureProperty(property_name, default_value, description)`: Verifica que exista la propiedad `property_name`. Si no existe actúa como defProperty.
- `property.{property_name}`: La variable global `property` se utiliza para acceder las propiedades previamente definidas.

Sintaxis

```
defProperty(name, default_value, description)
property.name
```

```
ensureProperty(name, default_value, description)

- name: nombre de la propiedad.
- default_value: valor por defecto.
- description: descripción.
```

Ejemplos

```
defProperty('rate', 300, 'Bits per second sent from sender')
defProperty('packetSize', 1024, 'Size of packets sent from sender, in Byte')

some_variable = property.packetSize
```

Las propiedades numéricas pueden ser utilizadas con operadores aritméticos (ej. +, -, *, /), o de concatenación de *strings* (+) ya que conservan el tipo con el que fueron definidas.

loadOEDL

Esta función obtiene un archivo OEDL y lo carga en el punto en que se llama.

Sintaxis

```
loadOEDL(location, optional_properties)

- location: a URI which references the OEDL script to load
    Los esquemas URI soportados son:
    - system:///foo/bar.rb , que carga el archivo ubicado en 'foo/bar.rb' desde
      el path de ruby por defecto para el EC.
    - file:///foo/bar.rb , que carga el archivo '/foo/bar.rb' en el FS local.
    - http://foo.com/bar.rb , que carga el archivo ubicado en URL 'http://foo.com/bar.rb'
    Nota: los esquemas 'file://' y 'system://' no soportan otro hosts que localhost,
    por lo que la parte del host de estos esquemas debe quedar vacía.
- optional_properties: un diccionario con valores sobre propiedades del OEDL a cargar
```

> Nota: desde LabWiki sólo se puede utilizar el esquema http, ya que al ejecutar un experimento éste se envía a otro servidor para su ejecución, por lo que los archivos cargados deben ser accesibles a través de la red.

Esta función puede ser utilizada para cargar las aplicaciones ya escritas en OEDL que son provistas desde el FileServer, y pueden ser accedidas desde el buscador de LabWiki para ver su definición. Ver sección *Aplicaciones precargadas*.

Ejemplos

```
loadOEDL('http://some.web.server.com/the_script.rb',  { property_name1: 'value1',
                                                        property_name2: 123 })
```

defApplication

Para ejecutar programas en los host hay que definir aplicaciones dentro del experimento. Luego de que éstas son registradas en los hosts pueden ejecutarse en el transcurso del experimento.

> Nota: existe un atajo para ejecutar comandos simples. Ej. `group('name').exec('command')`. Mas información en la sección sobre instrucciones OEDL.

Luego de que es definida, puede ser utilizada por cualquier grupo (concepto que agrupa hosts), si la aplicación fue registrada en el grupo. Ver sección de *defGroup*.

Sintaxis

```
defApplication(app_name) do |app|
  app.declaration1
  app.declaration2
  ...
end

- app_name: nombre de la aplicación, en el contexto del experimento únicamente.
- declarationX: otras declaraciones.
```

Hay tres tipos de declaraciones que se pueden hacer dentro del bloque de definición de la aplicación.

1. Declaraciones genéricas:

   - app.description: String. Una descripción sobre la aplicación.
   - app.binary_path: String. PATH absoluto del binario que se ejecuta en el host.
   - app.map_err_to_out: Booleano. Si se debe redirigir el error estándar a la salida estándar.
   - app.pkg_tarball: String. URL de un *tarball* que sera extraído en la raiz "/" dentro del hosts.
   - app.pkg_ubuntu: String. Nombre del paquete ubuntu que sera instalado con el gestor de paquetes apt de ubuntu.

2. Declaraciones de propiedades:

Son declaraciones sobre los parámetros que soporta esta aplicación. Estos pueden ser de invocación o dinámicos osea enviados a la entrada estándar durante la ejecución. Para esto se utiliza la función `app.defProperty()` como sigue:

```
app.defProperty(prop_name, description, command_line, options)

- prop_name: nombre del parámetro
- description: descripción del parámetro
- command_line: prefijo que es utilizado para pasar el parámetro al iniciar la
  aplicación en el host. Puede ser nil o vacío cuando no se necesita prefijo.
- options: diccionario con un conjunto de opciones sobre el parámetro. La lista
  de opciones válidas es:
     order: (Fixnum) orden al invocar la aplicación en la linea de comandos, por
            defecto FIFO.
     dynamic: (Boolean) si el parámetro puede cambiar dinamicamente.
     type: (Numeric|String|Boolean) tipo del parámetro.
     default: valor por defecto
     value: valor del parámetro
     mandatory: (Boolean) si el parámetro es requerido, por defecto false.

Ejemplos:

  app.defProperty('target', 'Address to ping', nil, {:type => :string,
                  :mandatory => true, :default => 'localhost'})
  app.defProperty('count', 'Number of times to ping', '-c', {:type => :integer})
```

1. Declaraciones de medidas.

Estas declaraciones pueden ser utilizadas únicamente en aplicaciones cuyos ejecutables fueron instrumentadas con la biblioteca de OML. Estas aplicaciones envían un flujo de datos de ciertos puntos de medida (MP) durante la ejecución. Podemos declarar la existencia de estos puntos de medidas y selectivamente habilitar los que nos interesan en una aplicación OEDL.

> Los MP están explicitamente programados dentro del binario que ejecuta en el host, por lo que deben ser conocidos por el usuario si se quiere definir aplicaciones con MP. Por esta razón, son provistas definiciones de aplicaciones para los binarios instrumentados con OML que se encuentran instalados en los hosts. Ver sección *Aplicaciones precargadas*.

```
app.Measurement(mp_name) do |mp|
  mp.defMetric(metric_name, metric_type)
  ...
end
```

```
- mp_name: el nombre del MP, como fue definido en la aplicación.
- metric_name: el nombre de la medida dentro del MP, como fue definida en la aplicación.
- metric_type: el tipo de la medida, como fue definida en la aplicación,
  ej. :string, :int32, :uint32, :int64, :uint64, :double, :blob

Ejemplos:

  app.defMeasurement('probe_statistic') do |m|
    m.defMetric('dest_addr', :string)
    m.defMetric('ttl', :uint32)
    m.defMetric('rtt', :double)
    m.defMetric('rtt_unit', :string)
  end

  app.defMeasurement('video_stream_statistic') do |m|
    m.defMetric('frame_number', :uint64)
    m.defMetric('drop_rate', :uint32)
    m.defMetric('codec_name', :string)
    m.defMetric('bitrate', :unit32)
  end
```

defGroup

Esta función define un nuevo grupo de recursos que va a participar en el experimento. Un grupo es considerado un recurso por lo que pueden ser anidados. La entidad básica que debe ser agregada es un host, indicándolo por su nombre, así como se ingreso al crear la topología.

Dentro del bloque de definición del grupo, se pueden asociar un conjunto de configuraciones o aplicaciones.

Sintaxis

```
defGroup(group_name, resource1_name, resource2_name, ...)

# con bloque de definición opcional:
defGroup(group_name, resource1_name, resource2_name, ...) do |g|
  ...
end

- group_name: nombre del grupo.
- resourceN_name: nombre del recurso N-ésimo del grupo
```

Dentro del bloque opcional, se puede configurar interfaces:

```
defGroup('server_host', property.on_of_my_hosts) do |g|
  g.net.e1.ip = "192.168.0.2/24"
```

```
end
```

Las interfaces pueden ser e0, e1, e2, etc. Que se traducen a eth0, eth1, eth2 en el host.

También se pueden agregar aplicaciones que estén previamente definidas en el OEDL e indicar los puntos de medición y configurar sus parámetros:

```
defGroup(group_name, resource1_name) do |g|
  g.addApplication(app_definition_name, app_definition_location) do |app|
    app.setProperty(prop_name, "some_value")
    ...
    app.measure(mp_name, :samples => 1)
    ...
  end
end

- app_definition_name: nombre de la aplicación como fue ingresado previamente con
  defApplication.
- prop_name: nombre del parámetro de la aplicación a configurar con some_value.
- mp_name: nombre del MP. Ademas se puede indicar la frecuencia con que se obtiene
  cada medición:
  samples: X - Cada X medidas reportadas
  interval: X - Cada X segundos (puede repetir la misma medición)

Ejemplos:

defGroup('Two_Resource', 'one_of_my_resource', 'another_of_my_resource')

defGroup('Pinger', 'yet_another_of_my_resource') do |g|
  g.addApplication("ping") do |app|
    app.setProperty('target', 'mytestbed.net')
    app.setProperty('count', 3)
    app.measure('probe_statistic', :samples => 1)
  end
en
```

onEvent

OEDL adopta un flujo basado en eventos para determinar las acciones a desempeñar en el experimento.

Hay un conjunto básico de eventos predefinidos y los usuarios pueden definir los propios.

Los eventos predefinidos son:

Evento	Descripción
:ALL_NODES_UP, :ALL_UP	Disparado cuando todos los recursos hosts confirmaron

Evento	Descripción
	su membrecía a los grupos.
:ALL_UP_AND_INSTALLED, :ALL_APPS_UP	Disparado cuando se cumple :ALL_NODES*UP y las aplicaciones han sido instaladas en los hosts (solo si las propiedades pkg** fueron especificadas).
:ALL_APPS_DONE	Disparado cuando los recursos de tipo aplicación han finalizado su ejecución.
:ALL_INTERFACE_UP	Disparado cuando todas las interfaces especificadas en los grupos se han configurado
:ALL_RESOURCE_UP	Disparado cuando :ALL_UP y :ALL_INTERFACE_UP y :ALL_APPS_UP

El conjunto de acciones a ejecutar cuando se dispara el evento se especifica en el bloque de declaración de `onEvent`, como se muestra a continuación:

```
onEvent(:event_name, consume_event) do
  ...
  instructions
end

- event_name: nombre del evento. (ej. :ALL_NODES_UP)
- consume_event: opcional,por defecto true. Si es false, el evento no es consumido
 luego de disparado, por lo que puede dispararse otras veces en el futuro.
- instructions: instrucciones a ejecutar ante dicho evento.

Ejemplos:

onEvent(:ALL_RESOURCE_UP)
  allGroups.startApplications
end
```

Las llamadas a onEvent pueden ser anidadas. En el caso de hacerlo, hay que asegurarse de que los eventos que se esperan ocurran en dicho orden. Si un evento ya ocurrió (y consumió), los manejadores de eventos para dicho evento cargados luego de consumido, no van a ejecutarse. Esto ocurre porque que el código OEDL del experimento es leído a medida que los eventos son disparados.

Observar que las llamadas a onEvent son no bloqueantes. El código especificado como manejador del evento en realidad es una *callback*.

Instrucciones

Dentro del bloque de manejo de un evento, se indican las instrucciones que se pueden ejecutar. Los siguientes tipos de instrucciones existen:

- Imprimir información en la salida del experimento.

```
info "Un mensaje particular"
```

- Comenzar/detener todas las aplicaciones de todos los grupos.

```
allGroups.startApplications
allGroups.stopApplications
```

- Comenzar/detener todas las aplicaciones de un grupo.

```
group('group_name').startApplications
group('group_name').stopApplications
```

- Ejecutar un bloque en X segundos.

```
group('Actor').startApplications
after 10.seconds do
  group('Actor').stopApplications
end
```

- Ejecutar un comando en los hosts de un grupo.

```
group('Actor').exec('/bin/hostname')
```

- Explicitamente solicita a los recursos que abandonen este experimento.

```
Experiment.leave_memberships
```

- Explicitamente termina la ejecución del experimento. Esto solicita a todas las aplicaciones iniciadas que se detengan.

```
Experiment.done
```

- Desconecta los recursos del experimento actual. Los recursos siguen realizando las tareas que fueron asignadas hasta el momento de realizar la desconexión. Esto implica que las instrucciones no enviadas no serán ejecutadas en los recursos.

```
Experiment.disconnect
```

defEvent

Para definir eventos propios se debe utilizar la función defEvent.

Sintaxis

```
defEvent(event_name) do |state|
  # Aquí se verifican las condiciones para determinar si disparar el evento.
end

# Con chequeo periódico cada period_seconds:
defEvent(event_name, every: period_seconds) do |state|
...
end

- event_name: nombre del evento. Se recomienda utilizar símbolos en mayúsculas. Ej. :MY_EVENT
- every: intervalo en segundos en que la condición del evento es verificada.
```

Condiciones

Las condiciones para disparar un evento pueden ser de varios tipos:

1. El estado de uno o mas recursos involucrados en el experimento. Ej. cuando cierta aplicacion finalizó.
2. El valor de una medida recolectada como parte del experimento. Ej. Cuando el RTT sea menor a 20ms.
3. Cualquier condición temporal. Ej. 7am, en 10 segundos, etc.

Estas condiciones son verificadas:

- Por defecto, cuando un mensaje de un recurso es recibido.
- Cada X segundos. Sólo si 'every' es indicado como parámetro de defEvent.

El bloque de código que verifica la condición debe:

1. Retornar el booleano `true`, si las condiciones del evento se han cumplido.
2. Retornar `false` sino.

Ejemplos

- Evento basado en el estado de ciertos recursos.

El controlador de experimento (EC) mantiene un estado de todos los recursos involucrados en el experimento. El estado es mantenido en una lista de diccionarios, donde cada diccionario mantiene los datos de un recurso. Observar que los recursos pueden ser aplicaciones, interfaces o hosts.

Cada vez que el EC recibe un mensaje de un recurso, actualiza dicho estado. Por lo tanto es posible que un experimentador consulte dicho estado y dispare un evento basado en él.

El estado esta disponible dentro del bloque de verificación del evento en la variable `state`.

En el siguiente ejemplo, se define un evento que se dispara al finalizar una aplicación.

```
defEvent :APP_EXITED do |state|
  triggered = false
  state.each do |resource|
      triggered = true if (resource.type == 'application') && (resource.state == 'stopped')
  end
  triggered
end

onEvent :APP_EXITED, consume_event = false do
    info "An application has just finished... should we do something about it?"
end
```

- Evento basado en las medidas recolectadas.

Cuando el experimento inicia una aplicación instrumentada con OML, ésta envía mediciones que están disponibles al experimentador dentro del bloque de defEvent.

Las mediciones son almacenadas en una base de datos SQL. Para consultar dicha BD se puedene scribir consultas en SQL plano, o utilizando la sintaxis de la biblioteca de Ruby Sequel. En ambos casos el resultado será una lista de diccionarios. Cada diccionario es una fila del resultado de la consulta.

A continuación se muestra un ejemplo que consulta la tabla del MP my_measurement_point y dispara el evento si el último valor es mayor a 0.99.

```
defEvent(:VALUE_ABOVE_THRESHOLD, every: 1) do
  # Query for some measurements...
  # returns an array where each element is a hash representing a row from the DB
  query = ms('my_measurement_point').select { [ :oml_ts_client, :value ] }
  data = defQuery(query)

  # Alternatively the above line could also be:
  # data = defQuery('select oml_ts_client, value from my_measurement_point_table')
  #
  # Also if you want to rename 'oml_ts_client' to 'ts'
  # query = ms('my_measurement_point').select { [ oml_ts_client.as(:ts), :value ] }
  # data = defQuery('select oml_ts_client as ts, value from my_measurement_point_table')

  triggered = false
```

```
  if !data.nil? && !(last_row = data.pop).nil? # Make sure we have some data
    triggered = true if last_row[:value].abs > 0.99
  end
  triggered
end

onEvent(:SINE_ABOVE_THRESHOLD) do
    info "The value just went above the threshold... should we do something about it?"
end
```

- Evento basado en condiciones temporales

Dispara el evento si, en el reloj del EC, la hora es mayor a 12:30.

```
defEvent(:AFTER_1230, every: 1) do
  now = Time.now
  true if now.hour>12 && now.min > 30
end

onEvent :AFTER_1230 do
  info "Do something after 12:30..."
end
```

defGraph

Esta función es utilizada para describir gráficas que LabWiki muestra y actualiza durante la ejecución del experimento.

Las gráficas pueden tomar los datos únicamente de los recolectados a través de OML.

Los tipos de gráficas soportados son: gráfica de línea, circular e histograma.

Sintaxis

```
defGraph 'graph_name' do |g|
  g.ms('mp_name').select {[ :field_1, :field_2, ... ]}
  g.caption 'some_caption'
  g.type 'type_of_graph'
  ... graph_specitific_options ...
end
- graph_name: nombre de la gráfica.
- mp_name: nombre del MP de la aplicación que genera los datos que queremos graficar.
- field_i: nombre de los campos del MP en los cuales estamos interesados.
- some_caption: subtitulo de la gráfica.
- type_of_graph: tipo de la gráfica, puede ser line_chart3, pie_chart2, histogram2
```

Observaciones

Cada MP es almacenados en una tabla distinta de la base de datos SQL. Además de las columnas correspondiente a los valores del MP, existen varias columnas comunes a todos los MP:

- oml_sender_id: id de la fuente que genero la medida.
- oml_ts_client: tiemstamp en el origen cuando la medida fue generada.
- oml_ts_server: timestamp en el servidor de recolección, cuando fue recibida.

Configuración de cada tipo de gráfica

- Línea

```
g.mapping :x_axis => :field_1, :y_axis => :field_2, :group_by => :field_3
g.xaxis :legend => 'some_legend_for_x_axis'
g.yaxis :legend => 'some_legend_for_x_axis', :ticks => {:format => 's'}
```

La configuración de arriba dibuja:

- Una linea de `field_2` en funcion de `field_1`
- Opcionalmente, si `field_2` contiene datos de varios orígenes, diferencias por `field_3`, entonces :group_by dibuja una linea separada por origen.
- La leyenda de los ejes X e Y, así como el formato de los ticks.

- Circular

  ```
  g.mapping :value => :field_1, :label => :field_2
  ```

  Esta configuración dibuja una gráfica circular donde los valores de una pieza son tomados de `field_1` y la etiqueta de dicha pieza es tomada de `field_2`.

- Histograma

```
g.mapping :value => :field_1, :group_by => :field_2
g.yaxis :legend => 'some_legend_for_y_axis'
g.xaxis :legend => 'some_legend_for_x_axis', :ticks => {:format => ',.2f'}
```

El histograma obtenido:

- Utiliza los valores de field_1
- Si field_1 contiene valores de varios orígenes, diferenciados por field_2, entonces la opción :group_by dibuja una barra separada por cada origen.

Ejemplos

La siguiente definición gráfica el RTT entre dos hosts, utilizando los MP de la aplicación ping, del OEDL ping_oml2.rb.

```
defGraph 'RTT1' do |g|
  g.ms('ping').select {[ :oml_sender_id, :oml_ts_client, :oml_ts_server, :rtt ]}
  g.caption "Round Trip Time (RTT) reported by each resource"
  g.type 'line_chart3'
  g.mapping :x_axis => :oml_ts_client, :y_axis => :rtt, :group_by => :oml_sender_id
  g.xaxis :legend => 'time [s]'
  g.yaxis :legend => 'RTT [ms]', :ticks => {:format => 's'}
end

# Draw a pie chart of the average RTT values other an entire ping run, using the sources
# that generated the measurement as labels for the pieces of the pie
#
defGraph 'RTT2' do |g|
  g.ms('rtt_stats').select {[ :oml_sender_id, :avg ]}
  g.caption "RTT Comparison Between Resources [ms]"
  g.type 'pie_chart2'
  g.mapping :value => :avg, :label => :oml_sender_id
end

# Draw a histogram chart for RTT values, if many source generated these values, then
# draw adjacent bars for each source for a given bin
#
defGraph 'RTT3' do |g|
  g.ms('ping').select {[ :oml_sender_id, :oml_ts_client, :oml_ts_server, :rtt ]}
          .where("rtt < 1")
  g.caption "Histogram of RTT counts [ms]"
  g.type 'histogram2'
  g.mapping :value => :rtt, :group_by => :oml_sender_id
  g.yaxis :legend => 'Count'
  g.xaxis :legend => ' ', :ticks => {:format => ',.2f'}
end
```

Notas sobre Ruby

- Los paréntesis son opcionales en los llamados a funciones.
- EL carecter ":" previo a un identificador lo convierte en un símbolo. Estos pueden ser considerados como *strings* en el contexto de OEDL. Se recomienda su uso sobre las *strings*.
- La última linea ejecutada en una función es su valor retornado, por lo que es común encontrar una variable sola en la última linea de un bloque de código.

Aplicaciones precargadas

La imagen utilizada al crear los hosts de tipo contenedor traen instaladas varias aplicaciones que utilizan OML. Para facilitar su utilización, se proveen definiciones de éstas en OEDL. Cada definición se encuentra dentro de un archivo, disponible en el servidor de archivos (FileServer).

Las aplicaciones disponibles son:

1. iperf_oml2: medir el rendimiento de una red, de manera activa creando flujos TCP y UPD y permitiendo ajustar algunos de sus parametros.
2. nmetrics_oml2: permite obtener información sobre el host sobre el cual ejecuta, como utilización de CPU, memoria utilizada.
3. otg_oml2: Utilizada para crear tráfico en modo cliente servidor. Cliente.
4. otr_oml2: Utilizada para crear tráfico en modo cliente servidor. Servidor.
5. ping_oml2: Simple herramienta ping.
6. trace_oml2: Captura paquetes utilizando libtrace.

y pueden ser cargadas en el experimento como sigue, suponiendo que la IP del servidor de archivos es 10.0.0.7:

```
loadOEDL('http://10.0.0.7/oml-apps/ping_oml2.rb')
```

Ademas, los usuarios pueden desplegar los archivos de las aplicaciones mencionadas buscándolos en el panel de preparación de LabWiki, aunque no pueden utilizar `loadOEDL` con los archivos locales. Estos archivos se encuentran en modo solo lectura.

Para ver los MP de cada aplicación se recomienda ver el código de definición en LabWiki o descargar el archivo correspondiente desde el servidor de archivos.

Troubleshooting

El experimento queda bloqueado en "TOTAL resources: X. Events check interval: 1".

Esto ocurre cuando el controlador de experimento envía solicitud de membrecia a los recursos pero no todos ellos están ejecutando omf_rc. Las causas del problema pueden ser:

1. Error al ingresar el nombre de algún recurso, al agregarlo a un grupo.
2. Si la topología fue configurada justo antes de ejecutar el experimento, puede ser que las instancias omf_rc no estén completamente inicializadas todavía.

Para asegurarse de que los recursos están levantados y asignados a mi slice, se puede ingresar a la web de GestorLAR donde se encuentra la lista de recursos activos e inactivos.

Known Bugs LabWiki

La información no se actualiza al crear una topoligía o iniciar un experimento

Hay un error que evita que el log del panel de ejecución se actualice mas de los primeros mensajes. Para ver sucesivos mensajes se debe refrescar la página.

No puedo guardar un documento

Si el icono de guardar documento desapareció e hicimos cambios sobre el documento, no hay otra opción que copiar el contenido del archivo de LabWiki a un lugar temporal dentro de nuestra PC, luego refrescar, sustituir el contenido viejo por el nuevo e intentar guardar nuevamente.

Otros

Pueden ocurrir otro tipo de errores en caso de que la conexión sea lente, por ejemplo:

- Al clickear guardar, no se guardan los cambios sobre el archivo.
- La página web no carga correctamente.
- No se puede exportar el dump de un experimento.

Si el problema es en la red de la infraestructura, lo mejor es reportar el problema al administrador de la testbed.

Manual de Administrador

El administrador de la testbed es capaz de acceder a información del estado de los diferentes servicios internos. A continuación se describe la interacción de diferentes componentes de software durante los principales casos de uso de los usuarios de la testbed, creación de topologías y ejecución de experimentos, y la información que puede obtenerse de ellos.

Un usuario crea una topología primero creando su descripción y luego creando una slice para esa topología, ambos pasos desde LabWiki. Al crear una slice, LabWiki envia una solicitud de creación (o activación, si ya estaba creada) de la slice al servicio GestorLAR, submodulo SliceService. Si la creación/activación es exitosa, LabWiki luego envía la descripción de la topología a SliceService para que éste la implante en la tesbed. SliceService lee la topologia, y para cada recurso host (de subtipo router o container) o red envía solicitudes de creación y membrecía del recurso a la slice, al AggregateManager, que también es un subcomponente de GestorLAR. Si todos los recursos se asignan a la slice correctamente entonces la topología fue creada exitosamente, sino, ningún recurso queda asociado a la slice y la creación falló.

Luego, un usuario crea un experimento primero escribiéndolo y luego ejecutándolo, ambos pasos desde LabWiki. En el segundo paso, el experimento es enviado un planificador de experimentos, JobService, donde es encolado para futura ejecución. Una vez que comienza el

experimento, se inicia una instancia de omf_ec con el experimento del usuario como parámetro. El omf_ec (controlador del experimento) es quien envía instrucciones a los recursos ejecutando omf_rc, controlador de recurso. Ambos controladores se comunican bidireccionalmente por medio de una cola de mensajes RabbitMQ. Durante la ejecución del experimento, los hosts ejecutando omf_rc y el propio omf_ec envían datos de mediciones sobre el experimento a un servidor de recolección de datos: OML Server. Éste ultimo, almacena los datos recibidos en una base de datos SQL, más específicamente PostgreSQL. Para cada experimento se crea una base de datos nueva y las tablas dentro de cada BD dependen de las mediciones tomadas por el experimento. Una vez que el experimento finaliza (o es abortado por el usuario), finaliza la ejecución de omf_ec y por lo tanto las instancias de omf_rc dejan de recibir instrucciones, pudiendo quedar ejecutando instrucciones o no, dependiendo de como haya finalizado el usuario el experimento.

Para acceder a información de los componentes mencionados, es necesario conocer la URL o IP donde escuchan los servicios. Como resumen, los servicios son:

Servicio	**Descripción**
LabWiki	Aplicación web. Punto de entrada para usuarios de la testbed. Toda su interacción debería ser a través de esta web.
JobService	Planificador de experimentos. Recibe experimentos desde LabWiki.
SliceService	Api HTTP rest. Crea slices, configura topologías. Recibe solicitudes únicamente desde LabWiki.
AggregateManager	Api Python. Crea y configura recursos. Los asigna a slices internas por usuario (no son las slices de SliceService).
RabbitMQ	Servidor de mensajería AMQP. Sirve de pasaje de mensajes entre omf_ec y omf_rc. Ademas es utilizado para tareas asíncronicas en SliceService.
OML Server	Recolecta medidas de experimentos y almacena en base de datos PostgreSQL.
PostgreSQL	Base de datos de OML Server.

Para acceder a información de cada componente:

- JobService: Accediendo con su IP o hostname a través de la web, es posible ver la cola de experimentos e información detallada de estos.
- GestorLAR (SliceService y AggregateManager): Se puede acceder a una interfaz web de la base de datos con el estado de los recursos y datos internos. Entrando a `/admin` con usuario y contraseña. Una vez ingresado, se pueden editar los datos internos o realizar acciones sobre ellos. El caso mas útil es dar de baja una topología. Entrando a SliceService y Luego a Slices veremos todas las Slices. Seleccionamos al menos una, y en acciones seleccionaimos "Release Slices". Esto creará un proceso asíncrono que liberará las Slices de a una, pudiendo tardar varios minutos dependiendo de la cantidad de recursos a liberar.

- RabbitMQ: Accediendo al host que lo ejecuta, se puede utilizar el comando `rabbitmqctl` para visualizar información del estado, como usuarios conectados, etc. El omf_ec ejecuta con el usuario `econtroller` mientras que los omf_rc con el usuario `resource`. Al listar usuarios conectados, podremos ver la cantidad de recursos conectados, sin embargo no podremos identificar que recurso esta conectado y cual no.
- OML Server: Sobre este servicio no hay un método para desplegar información corriente, ademas de los logs.
- PostgreSQL: Basta con conectarse a la base de datos con un cliente de PostgreSQL.

Apéndice 3: Manual de instalación y configuración

A menos que se indique lo contrario, en todos los casos se utilizará un equipo con una instalación base de Ubuntu 16.04.

Servidor Principal

Para el LAR se utilizó una máquina virtual sobre la que corren como contenedores LXC los servicios de Interfaz web, *Aggregate Manager*, *Experiment controller*, Servidor de mensajería y Servidor OML. Dichos servicios pueden correr en equipos independientes, por lo que esta sección es opcional.

El primer paso es instalar el *hipervisor* LXD para correr los servicios del gestor. Para esto ejecutar: `sudo apt install lxd`

Se recomienda utilizar ZFS como sistema de archivos para el almacenamiento de los contenedores. Para esto es necesario ejecutar: `sudo apt install zfsutils-linux`

Una vez instalados estos paquetes, ejecutar: `sudo lxd init`

En este paso se elegirá el sistema de archivos a utilizar (se recomienda ZFS), se habilitará opcionalmente la API para administración remota (no es necesaria en este caso) y se configurará una subred IPv4 para los contenedores. Por más detalles sobre la instalación, ver la guía en Ubuntu Insights.

Para crear un nuevo contenedor, ejecutar: `lxc launch ubuntu:16.04 container_name`

Crear un contenedor por cada servicio, y asignar a cada uno una dirección IP fija.

Se utilizará un servicio con reglas de *firewall* que realicen la redirección NAT a los puertos del gestor, reenviando el tráfico a los servidores que correspondan. Se realizará un respaldo previo a la activación del servicio, volviendo a cargar las reglas de LXD al detenerlo.

Se asumirán las siguientes direcciones IP para cada servicio:

Servicio	**IP**
Aggregate Manager	10.0.0.2
AMQP Server	10.0.0.3
Experiment Controller	10.0.0.4
OML Server	10.0.0.5
Web UI	10.0.0.6

Crear un directorio `/usr/lib/firewall`, y dentro del mismo un archivo de nombre firewall con el siguiente contenido, asignando correctamente SERVER_IP con la dirección IP asignada al equipo.

```
#!/bin/bash

SERVER_IP="X.X.X.X"

start() {

    # Save current rules
    iptables-save > /usr/lib/firewall/iptables.rules

    # Flush everything
    iptables -F
    iptables -t nat -F

    # Drop by default
    iptables -P INPUT DROP
    iptables -P OUTPUT DROP
    iptables -P FORWARD DROP

    # Allow NAT from containers
    iptables -A FORWARD -i lxdbr0 -j ACCEPT

    # Allow established connections
    iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
    iptables -A OUTPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
    iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT

    # Allow SSH to server
    iptables -A INPUT -d $SERVER_IP -p tcp --dport 22 -j ACCEPT

    # Allow DNS from server
```

```
    iptables -A OUTPUT -p udp --dport 53 -j ACCEPT

    # Allow DNS from containers
    iptables -A INPUT -s 10.0.0.0/24 -i lxdbr0 -p udp --dport 53 -j ACCEPT
    iptables -A INPUT -s 10.0.0.0/24 -i lxdbr0 -p tcp --dport 53 -j ACCEPT

    # Allow NAT to containers
    # Mark the packets
    iptables -t nat -A PREROUTING -d $SERVER_IP -p TCP --dport 80 -j MARK \
        --set-mark 1
    iptables -t nat -A PREROUTING -d $SERVER_IP -p TCP --dport 5672 -j MARK \
        --set-mark 1
    # Set final destination
    iptables -t nat -A PREROUTING -d $SERVER_IP -p TCP --dport 80 -j DNAT \
        --to-destination 10.0.0.6:80
    iptables -t nat -A PREROUTING -d $SERVER_IP -p TCP --dport 5672 -j DNAT \
        --to-destination 10.0.0.3:5672

    # Allow marked packets in the forward chain
    iptables -A FORWARD -m mark --mark 1 -j ACCEPT

    # Masquerade
    iptables -t nat -A POSTROUTING -o ens3 -j MASQUERADE

    exit 0

}

stop() {

    # Delete all rules
    iptables -F
    iptables -t nat -F

    # Restore previous rules
    iptables-restore < /usr/lib/firewall/iptables.rules

    exit 0

}


case "${1}" in
```

```
    start)
        start
    ;;

    stop)
        stop
    ;;

    restart|reload|force-reload)
        ${0} stop
        ${0} start
    ;;

    *)
    echo "Usage: ${0} {start|stop|restart|reload|force-reload}"
    exit 2

esac
```

Crear un archivo en `/lib/systemd/system/firewall.service` con el siguiente contenido:

```
[Unit]
Description=Reglas de Firewall para el gestor del LAR
After=lxd.service
Requires=lxd.service

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/lib/firewall/firewall start
ExecStop=/usr/lib/firewall/firewall stop

[Install]
WantedBy=multi-user.target
```

Para habilitar, iniciar y detener el servicio se utilizan los comandos:

```
systemctl enable firewall.service
systemctl start firewall.service
```

y

```
systemctl stop firewall.service
```

Web UI

Dentro del directorio donde se albergará el código, clonar los repositorios necesarios:

```
git clone https://github.com/mytestbed/labwiki.git
git clone https://github.com/mytestbed/omf_web.git
git clone https://github.com/mytestbed/omf_oml.git
```

Instalar dependencias:

```
apt install ruby sqlite3 libsqlite3-dev libicu-dev cmake libpq-dev pkg-config \
    libpq-dev ruby-dev
gem install pg bundler
```

Instalar *LabWiki*:

```
cd labwiki
export LABWIKI_TOP=`pwd`
bundle install --path vendor
rake post-install
```

Instalar los plugins de *LabWiki* utilizados:

```
./install_plugin https://github.com/mytestbed/labwiki_experiment_plugin.git
./install_plugin https://gitlab.fing.edu.uy/GestorLAR/labwiki_topology_plugin
```

> En la ultima versión de LabWiki al mes de Agosto de 2017 hay un bug fácilmente corregible si se agregaga la linea 'require omf_oml/table' al principio del archivo labwiki/lib/labwiki /plugin_manager.rb.

Crear el archivo de configuración de *LabWiki* `etc/labwiki/labwiki.yaml` (la ruta no es absoluta) con el siguiente contenido:

```
labwiki:
  environment: production
  session:
    authentication:
        type: lar
    repositories:
      - name: system
        type: file
        read_only: true
        top_dir: /labwiki/repo

        # Keep user repositories separated
      - name: <%= OMF::Web::SessionStore[:id, :user] %>
        type: file
        read_only: false
        create_if_not_exists: true
        top_dir: /labwiki/repo_users/<%= OMF::Web::SessionStore[:id, :user] %>

    default_plugins: # Create these plugins the first time a user logs in
      - column: plan
        plugin: 'wiki'
        action: "on_get_content"
        url: 'system:/labwiki/wiki/quickstart/quickstart.md'


  plugins:
    experiment:
      plugin_dir: labwiki_experiment_plugin
      job_service:
        host: 10.0.0.4
        port: 8002
      oml:
        host: 10.0.0.5
        port: 3003

    topology:
      plugin_dir: labwiki_topology_plugin
      slice_service:
        url: http://10.0.0.2
```

A continuación agregamos el código necesario para simular autenticación en *LabWiki* ya que esta por defecto viene sin control de acceso. Crear el archivo `lib/labwiki/authentication/lar_warden.rb` con el siguiente contenido:

```
Warden::Strategies.add(:lar) do


  def valid?
    params['username'] || params['password']
  end



  def authenticate!
    # u = User.authenticate(params['username'], params['password'])
    # u.nil? ? fail!("Could not log in") : success!(u)
    success!(params)
  end
end
```

Luego crear el archivo `lib/labwiki/authentication/lar.rb` con el siguiente contenido:

```
require 'labwiki/authentication/lar_warden'


module LabWiki
  class Authentication
    class Lar < Authentication
      register_type :lar



      def parse_user(user)
        # User is a hash with 'username' and 'password'
        OMF::Web::SessionStore[:id, :user] = "#{user['username']}_id"
        OMF::Web::SessionStore[:name, :user] = "#{user['username']}_name"
        OMF::Web::SessionStore[:urn, :user] = "#{user['username']}_urn"
        OMF::Web::SessionStore[:projects, :user] = [
        {'name': 'Default Project', 'uuid': 0 }
    ]#"#{user['username']}_project"]
      end



      # Here is the form to input authentication data.
      def login_content
        Erector.inline do
          div class: 'row' do
            div class: 'col col-sm-6 col-sm-offset-3' do
```

```
    form :class => 'form-signin', :method => 'post' ,
      :action => '/' do
        h2 class: "form-signin-heading" do
          text 'Please Sign In'
        end
        p do text 'For now every username/password is valid' end
        div class: 'alert', style: 'display: none' do
          p id: 'error_msg'
        end
        div class: 'form-group' do
          input type: 'text', name: 'username',
                class: "form-control",
                placeholder: "Username"
        end
        div class: 'form-group' do
          input type: 'password', name: 'password',
                class: "form-control",
                placeholder: "Password"
        end
        button class: "btn btn-large btn-primary sign-in",
              type: "submit" do text 'Sing in' end
    end
  end
end


javascript <<-JS
  function getParameterByName(name) {
    name = name.replace(/[\[]/, "\\[").replace(/[\]]/, "\\]");
    var regexS = "[\\?&]" + name + "=([^&#]*)";
    var regex = new RegExp(regexS);
    var results = regex.exec(window.location.search);
    if(results == null)
      return null;
    else
      return decodeURIComponent(results[1].replace(/\+/g, " "));
  }


  var msg = getParameterByName("msg");
  if (msg != null) {
    $('#error_msg').text(msg);
    $('.alert').show();
  }
```

```
          JS
        end
      end
    end
  end
end
```

Crear el script de inicio `/root/start-labwiki.sh`:

```
#!/bin/bash

/root/labwiki/bin/labwiki --lw-config /root/labwiki/etc/labwiki/labwiki.yaml -l \
    /var/log/labwiki.log -p 80 start
```

Crear los links y archivos necesarios:

```
ln -s /root/labwiki /labwiki
mkdir /labwiki/repo
mkdir -p /labwiki/wiki/quickstart/
touch /labwiki/wiki/quickstart/quickstart.md
```

Luego instalamos supervisor para mantener *labwiki* siempre ejecutando, para instalarlo ejecutamos:

```
apt install supervisor
```

Creamos el archivo `/etc/supervisor/conf.d/labwiki.conf` con el siguiente contenido:

```
[program:labwiki]
command=/root/start-labwiki.sh
stopasgroup=true
```

Y por último reiniciamos supervisor:

```
supervisorctl reload
```

El servicio Labwiki ejecutara al inicio del sistema por defecto y el proceso será re-ejecutado en caso de que se interrumpa su ejecución.

Aggregate Manager y Slice Service

La instalación de estos dos componentes se realiza de manera conjunta ya que se encuentran implementados en la misma solución y ejecutan bajo el mismo servicio web.

Una vez clonado el código fuente desde el repositorio, seguimos los siguientes pasos:

- Instalamos dependencias:

```
apt-get install python python-virtualenv build-essential python-dev \
      libssl-dev vlan nginx uwsgi
```

- Crear un virtualenv de Python donde se instalarán las dependencias de este lenguaje:

```
virtualenv venv -p python2
source venv/bin/activate
pip install -r requirements.txt
```

- Configurar base de datos.

  Se puede utilizar SqLite o PostgreSQL como gestor de base de datos, recomendada es la segunda opción. Si se utiliza la primera, no requiere configuración adicional. Si se utiliza la segunda, se debe configurar la IP, puerto, nombre de base de datos, usuario y contraseña para acceder al servidor de base de datos. Dicha configuración se encuentra en el archivo gestor_lar/settings.py.

  Por defecto, la ip es 127.0.0.1, puerto 5432 y se utiliza la base de datos "gestor_lar". El usuario se obtiene de la variable de ambiente `GESTORLAR_DATABASE_USER` y la contraseña de `GESTORLAR_DATABASE_PASSWORD`. Se utiliza PostgreSQL si la variable de ambiente `GESTORLAR_ENVIRONMENT` es igual a "production".

  Una vez configurada la base de datos, ejecutar el siguiente comando para crear las tablas:

```
python manage.py migrate
```

Luego se pueden cargar datos iniciales con:

```
python manage.py loaddata data
```

- Configuración de archivos estáticos javascript, css, imágenes, etc:

```
python manage.py collectstatic --clear
```

- Ejecución utilizando el servidor web Apache. Necesitamos instalar Apache y el módulo wsgi. Para esto ejecutamos:

```
apt-get install apache2 libapache2-mod-wsgi
```

Luego, creamos el archivo de configuración de Apache, bajo `/etc/apache2/sites-enabled` y agregamos la siguiente configuración, suponiendo que clonamos el repositorio bajo el directorio `/opt`:

```
<VirtualHost *:80>
    Alias /static /opt/gestor_lar/static
    <Directory /opt/gestor_lar/static>
        Require all granted
    </Directory>

    <Directory /opt/gestor_lar/gestor_lar>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>

    WSGIDaemonProcess myproject python-path=/opt/gestor_lar \
            python-home=/opt/gestor_lar/venv
    WSGIProcessGroup myproject
    WSGIScriptAlias / /opt/gestor_lar/gestor_lar/wsgi.py

</VirtualHost>
```

Por último reiniciamos el servicio *apache2* y el servidor queda funcionando.

- Configuración de cola de mensajes para trabajos asincrónicos.

  Para iniciar los procesos que ejecutan trabajos asincrónicos de nuestro servidor debemos instalar el software supervisor. Luego crearemos un Shell script que mantendrá los comandos necesarios para iniciar dichos procesos. Por último crearemos un archivo de configuración de supervisor para que éste se encargue de levantar dichos procesos si estos se caen por alguna razón. Primero debemos configurar la URL de la cola de mensajes *rabbit-mq*. Podemos utilizar una nueva o la que se utiliza para la infraestructura de OMF. La configuración se encuentra en `gestor_lar/gestor_lar/settings.py` bajo la variable `CELERY_BROKER_URL`.

  Para instalar supervisor ejecutamos: `apt-get install supervisor`

  Luego creamos un archivo `gestor-lar-worker.sh` donde agregamos las siguientes lineas, suponiendo que el código del proyecto se encuentra en `/opt/gestor_lar`:

  ```
  #!/bin/bash

  PROJECT_HOME=/opt/gestor_lar
  VENV_DIR=$PROJECT_HOME/venv

  source $VENV_DIR/bin/activate
  cd $PROJECT_HOME && celery worker -A gestor_lar -l info -c2
  ```

  Luego añadimos el archivo de configuración de supervisor. Bajo `/etc/supervisor/conf.d` creamos el archivo `celeryworker.conf` y añadimos, suponiendo que el Shell script creado anteriormente está en `/opt/gestor-lar-worker.sh`:

  ```
  [program:celeryworker]
  command=/opt/gesotr-lar-worker.sh
  stopasgroup=true
  ```

  Y por último reiniciamos supervisor: `supervisorctl reload`.

- Configuración de integración con servicios de la infraestructura. El módulo gestor_lar interactúa con un hipervisor LXD para creación de contenedores y con un switch cisco SF200-48P para configuración de VLANs. Con el primero lo hace a través de su API REST. Se requiere haber configurado previamente este servicio, para esto ver la guía de instalación y configuración de LXD.

Además se crean puentes entre las interfaces virtuales de los contenedores y la VLAN asignada para el experimento, para esto el gestor_lar se conecta por ssh.

Dentro del archivo `gestor_lar/settings.py` se deben configurar las variables:

```
- LXD_SERVER_URL: url http donde ubicar la API REST de LXD.
- LXD_SERVER_CRT: Ubicación del archivo que contiene el certificado del
  servidor LXD.
- LXD_SERVER_KEY: Archivo que contiene la clave del certificado
- LXD_SERVER_PASS: Contraseña de LXD.
- LXD_SERVER_SSH_HOSTNAME: Hostname en que el servidor LXD escucha ssh.
- LXD_SERVER_SSH_PORT: Puerto en el que el servidor LXD escucha ssh.
- LXD_SERVER_SSH_USER: Usuario de linux con el que conectarse por ssh.
- LXD_SERVER_SSH_PASS: Contraseña del usuario de linux.
- LXD_SERVER_TRUNK_IFACE: Nombre de la interfaz que está conectada al switch
  en modo trunk.
```

Los archivos server key y server crt de lxd son generados automáticamente por este y se pueden encontrar en el directorio /var/log/lxd.

Para comunicarse con los switch el gestor_lar necesita que estos sean agregados a la base de datos, donde se configurará su dirección IP, usuario y contraseña de la interfaz web. Además se deberán ingresar los puertos y su asociación con la interfaz de los routers, si es que existe tal conexión. Solo uno de los switchs gestionados está conectado al gestor_lar, el puerto al que corresponde dicha conexión debe ingresarse a la base de datos con el valor `is_virtualization_trunk=True`.

Experiment Controller

El Experiment Controller de OMF es ejecutado por OMF Job Service. Para instalarlo, primero se deben instalar las dependencias:

```
sudo apt install ruby ruby-dev sqlite3 libpq-dev libsqlite3-dev libxml2 \
    zlib1g-dev liblzma-dev libxslt1-dev
gem install bundler
```

Luego descargar el código de OMF Job Service e instalar sus dependencias de Ruby:

```
git clone <https://github.com/mytestbed/omf_job_service.git>
cd omf_job_service
bundle config build.eventmachine --with-cflags=-O2 -pipe -march=native -w
bundle install --path vendor
rake post-install
```

> Observación: Si estamos utilizando rvm, el ultimo paso (rake post-install) puede dar error. Una solución es cambiar la linea 53 del archivo Rakefile por rvm_bin_path = false.

Luego instalar omf_ec localmente a omf_job_service:

```
cd omf_job_service/omf_ec
bundle install --path vendor
export FRCP_URL=amqp://10.0.0.3/lar
rake post-install
```

El archivo de configuración se encuentra en `omf_ec.yml`. Se debe configurar el usuario y contraseña de AMQP. Por ejemplo:

```
communication:
  url: amqp://10.0.0.3/lar
  user: econtroller
  pass: econtroller
```

Instalar el cliente PostgreSQL para conectarse con la base de datos:

```
sudo apt install postgresql-client
```

Luego editar el archivo `etc/omf_job_service.yaml` que configura job_service y establecer la dirección del servidor OML.

```
oml_server: 'tcp:10.0.0.5:3003'
db_server: 'postgres://oml:oml@10.0.0.5'
```

Por último, crear un servicio para iniciar automáticamente el Job Service. Para esto instalamos supervisor y creamos un archivo con su configuración. Por ejemplo

`/etc/supervisor/conf.d/jobservice.conf` con el siguiente contenido:

```
[program:jobservice]
command=/root/start-jobservice.sh
stopasgroup=true
```

Crear el script de inicio `/root/start-jobservice.sh`:

```
#!/bin/bash

/root/omf_job_service/bin/omf_job_service start
```

Luego ejecutamos `supervisorctl reload`.

Servidor de Mensajeria

Para el servidor de mensajería se optó por RabbitMQ, la implementación de AMQP recomendada por OMF.

Para instalar el servidor, ejecutar:

```
sudo apt install rabbitmq-server
```

Se utilizarán tres usuarios, uno para los recursos, otro para el AM y otro para el controlador de experimentos. Para esto ejecutar los siguientes comandos eligiendo una contraseña para cada usuario.

```
rabbitmqctl add_user econtroller <password>
rabbitmqctl add_user resource <password>
rabbitmqctl add_user am <password>
```

Se debe crear un virtual host, el cual será utilizado para la comunicación entre el controlador y los recursos.

```
rabbitmqctl add_vhost lar
```

Por último, dar permisos a los usuarios sobre el virtual host creado.

```
rabbitmqctl set_permissions -p lar econtroller ".*" ".*" ".*"
rabbitmqctl set_permissions -p lar resource ".*" ".*" ".*"
rabbitmqctl set_permissions -p lar am ".*" ".*" ".*"
```

Servidor OML

Para instalar OML en Ubuntu es necesario agregar un repositorio. Para esto crear un archivo `/etc/apt/sources.list.d/oml2.list` con el siguiente contenido:

```
deb http://download.opensuse.org/repositories/devel:/tools:/mytestbed:/stable/xUbuntu_1
```

Luego instalamos el servidor OML desde el gestor de paquetes. Puede utilizar tres backends para almacenar la información: SQLite, archivo plano o PostgreSQL. Utilizaremos PostgreSQL ya que es la opción más potente de las tres.

```
apt-get update
apt-get install oml2 postgresql
```

Configurar PostgreSQL para escuchar en todas las interfaces, editando el archivo `/etc/postgresql/9.5/main/postgresql.conf`:

```
listen_addresses = '*'
```

Agregar al final del archivo `/etc/postgresql/9.5/main/pg_hba.conf` la siguiente línea:

```
host    all         all         10.0.0.4/32    trust
```

La configuración de OML se realiza desde el archivo `/etc/default/oml2-server`. Las opciones básicas para utilizar PostgreSQL son las siguientes:

```
OPTS="-l 3003 --logfile=/var/log/oml2-server.log --user=oml2 --group=oml2"
OPTS="$OPTS --backend postgresql --pg-host=localhost --pg-port=5432
      --pg-user=<user> --pg-pass=<password>"
```

Por último iniciamos el servicio:

```
update-rc.d oml2-server enable
```

```
service oml2-server start
```

> Se recomienda editar el archivo /etc/init.d/oml2-server de manera de corregir el orden relativo en que los servicios oml2-server y postgresql son iniciados. Para corregir la dependencia, agregar 'postgresql' a 'Required-Start' en el archivo mencionado. Luego ejecutar 'update-rc.d oml2-server remove; update-rc.d oml2-server defaults'.

Servidor OML con balanceador de carga

Para que la infraestructura escale mas apropiadamente es recomendado ejecutar varias instancias de oml2-server detrás de un *load balancer* TCP. Dos proyectos abiertos que ofrecen esta capacidad son nginx y haproxy. En este manual se detalla la configuración del primero.

Tanto el balanceador como los servidores OML ejecutan en la misma instancia. Su desacople es inmediato. Aqui se muestra la configuración para tres instancias de oml2-server.

Las instancias de oml2-server ejecutan utilizando supervisor. La configuración de este es:

- /etc/supervisor/conf.d/oml2-server.conf

```
[program:oml2-server]
command=/root/oml2-server-lar -l 30%(process_num)02d \
                        --logfile=/var/log/oml2-server%(process_num)02d.log
stopasgroup=true
process_name=oml2-server%(process_num)02d
numprocs=3
numprocs_start=4
```

Luego creamos el archivo shell `oml2-server-lar` que ejecuta oml2-server con la mayoría de configuración del servidor OML menos las dos opciones que varían: el puerto en el que escuchan y el archivo de configuración.

- /root/oml2-server-lar

```
#!/bin/bash

source /etc/default/oml2-server
/usr/bin/oml2-server $OPTS $*
```

Luego, modificamos las opciones por defecto en el archivo /etc/default/oml2-server.

```
OPTS="--user=oml2 --group=oml2"
# Use a PostgreSQL backend (default values given explicitely)
OPTS="$OPTS --backend postgresql --pg-host=10.0.0.5 --pg-port=5432
      --pg-user=oml --pg-pass=oml"
```

Por ultimo instalamos nginx y agregamos el siguiente archivo de configuración:

- /etc/sites-enabled/oml2.conf

```
stream {
 upstream stream-oml2 {
   least_conn;

   server localhost:3004;
   server localhost:3005;
   server localhost:3006;
 }
 server {
   listen        3003;
   proxy_pass    stream-oml2;
 }
}
```

> Observación: En la configuración por defecto de nginx este archivo es cargado dentro del bloque http. Sin embargo el bloque stream no debe tener un bloque padre para funcionar como balanceador de carga.

Resource controller

Instalar las dependencias necesarias para el Resource Controller:

```
apt install ruby build-essential ruby-dev libssl-dev
```

Por último, instalar el Resource Controller:

```
gem install omf_rc --no-ri --no-rdoc
install_omf_rc -i -c
```

Editar el archivo `/etc/omf_rc/config.yml`, configurando los siguientes parámetros. Sustituir *rabbitmq* por la dirección IP del servidor de mensajería, y *username* y *password* por el usuario y contraseña configurados en el mismo para los Resource Controllers.

```
debug: false

environment: production

communication:
    url: amqp://<rabbitmq>/lar
    user: <username>
    pass: <password>

resources:
  - type: node
    uid: <%= Socket.gethostname %>
```

Para iniciar automáticamente omf_rc hay que reemplazar systemd por upstart como gestor de programas de inicio. Al instalar upstart automáticamente se elimina systemd:

```
apt install upstart-sysv
update-initramfs -u
```

Reiniciar el equipo con el comando reboot. Luego se podrá iniciar y detener el Resource Controller ejecutando `start omf_rc` y `stop omf_rc`.

Hypervisor LXD

La instalación del hypervisor LXD que maneja contenedores de experimentos es idéntica a la del hipervisor que gestiona los servicios del gestor.

Configuración de red

Los containers creados por defecto incluyen una interfaz de nombre eth0, conectada a la interfaz principal del servidor. Dado que este nombre de interfaz es utilizado comúnmente, se utilizará un nombre distinto para la interfaz de gestión.

Para editar el perfil por defecto, ejecutar `lxc profile edit default` y cambiar el nombre de la interfaz por defecto:

```
name: default
config: {}
description: Default LXD profile
devices:
  mgmt:
    name: mgmt
    nictype: bridged
    parent: lxdbr0
    type: nic
```

Para poder utilizar imágenes de LXD a través de la API, es necesario copiarlas al repositorio de imágenes local. Para esto ejecutar `lxc image copy ubuntu:16.04 local: --alias ubuntu16`.

De esta forma se dispondrá de una imagen de Ubuntu 16.04 para utilizar de manera local.

Crear una imagen de recurso base

Para crear un container para utilizar como base, ejecutar `lxc launch ubuntu16 recurso-base`.

Luego ingresar al container con el comando `lxc exec recurso-base bash`.

Agregar la siguiente línea al archivo `/etc/rc.local`, para cambiar el nombre de la interfaz de red al iniciar el container.

```
sed -i 's/eth0/mgmt/' /etc/network/interfaces.d/50-cloud-init.cfg
service networking restart
```

Esto es necesario ya que, incluso al modificar el archivo manualmente, al crear un

container utilizando este como base se revierten los cambios en la configuración de red.

Instalación de omf_rc

Instalar en el container creado el componente omf_rc, como se describe en la sección Resource Controller del Apéndice 3.

Generación del template

Para publicar el container como un template, ejecutar en el host del hypervisor LXD `lxc publish recurso-base --alias omf_resource_ubuntu16`

Configurar acceso remoto a hypervisor LXD

Además de la API de LXD, se necesitará crear un usuario a nivel de sistema operativo para acceder por ssh. Este usuario se utiliza para la configuración de VLANs para la conexión de los containers.

Para crear el usuario y configurarle una contraseña, ejecutar como root `useradd gestor` y `passwd gestor`.

El usuario creado debe poder ejecutar los comandos ip y vconfig con permisos de root sin introducir la password. Para otorgar estos permisos, editar el archivo sudoers con el comando visudo, agregando la siguiente línea:

```
gestor ALL=NOPASSWD:/sbin/vconfig,/sbin/ip
```

Apéndice 4: Códigos de ejemplo

En este apéndice se encuentran descritos archivos con código fuente mencionados en el documento principal.

Análisis experimental: Experimento 1

El siguiente archivo corresponde al primer experimento descrito en la sección análisis experimental. El experimento consiste en la medición del tráfico entre dos nodos a través de un nodo intermedio.

```
# Carga de definición de aplicaciones externas
loadOEDL('http://10.0.0.7/oml-apps/nmetrics_oml2.rb')
loadOEDL('http://10.0.0.7/oml-apps/iperf_oml2.rb')

# Definición de propiedades.
# Definir propiedades permite que estos valores puedan ser editados previamente
# a iniciar el experimento sin modificar el código de éste.
defProperty('source', 'contenedor-source')
defProperty('destination', 'contenedor-destination')
defProperty('router', 'contenedor-router')

defProperty('net_source', '192.168.1.0')
defProperty('net_destination', '192.168.1.16')
defProperty('network_prefix_size', 28)

defProperty('source_ip', '192.168.1.1')
defProperty('router_ip_netsource', '192.168.1.2')
defProperty('router_ip_netdestination', '192.168.1.17')
defProperty('destination_ip', '192.168.1.18')

# Definición de grupos: los grupos representan los actores del experimento.
# Un grupo puede involucrar uno o mas recursos de tipo nodo.
defGroup('source', property.source) do |g|
  # Add client
  g.addApplication("iperf") do |app|
    app.setProperty('client', property.destination_ip)
    app.setProperty('port', 8080)
    app.setProperty('bandwidth', '10mbit/s')
```

```
    app.setProperty('time', 60)
  end

  g.e1.ip = "#{property.source_ip}/#{property.network_prefix_size}"
end

router_ip_netdestination = \
    "#{property.router_ip_netdestination}/#{property.network_prefix_size}"

defGroup('router', property.router) do |g|

  # Measure traffic
  g.addApplication("nmetrics") do |app|
    app.setProperty('interface', 'eth1')
    app.measure('network', :interval => 1)
  end
  g.addApplication("nmetrics") do |app|
    app.setProperty('interface', 'eth2')
    app.measure('network', :interval => 1)
  end

  g.e1.ip = "#{property.router_ip_netsource}/#{property.network_prefix_size}"
  g.e2.ip = router_ip_netdestination
end

defGroup('destination', property.destination) do |g|
   # Add server
  g.addApplication("iperf") do |app|
    app.setProperty('server', true)
    app.setProperty('port', 8080)
  end

  g.e1.ip = "#{property.destination_ip}/#{property.network_prefix_size}"
end

net_destination = "#{property.net_destination}/#{property.network_prefix_size}"
net_source = "#{property.net_source}/#{property.network_prefix_size}"


# Comienzo de la lógica del experimento. Se definen acciones a realizar ante
# eventos.
onEvent :ALL_UP do
  after 5 do
    group('source')\
```

```
    .exec("ip route add #{net_destination} via #{property.router_ip_netsource}")

    group('destination')\
    .exec("ip route add #{net_destination} via #{property.router_ip_netdestination}")

    group('destination').startApplications
    group('router').startApplications
  end

  after 30 do
    group('source').startApplications

    after 20 do
      group('router').exec('ip addr flush dev eth2')
    end

    after 40 do
       group('router').exec("ip addr add #{router_ip_netdestination} dev eth2")
    end
  end

  after 120 do
    info 'Stopping traffic generator'
    allGroups.stopApplications
  end

  after 140 do
    Experiment.done
  end
end


# Definición de gráficas. Las gráficas son utilizadas únicamente al ejecutar
# experimentos a través de LabWiki. Muestran el valor de ciertos parámetros
# en el correr del experimento.
defGraph 'Packages received router by interface' do |g|
   g.ms('network').select {
     [ :oml_sender_id, :oml_ts_client, :oml_ts_server, :name, :rx_packets_avg ]
   }
   g.caption "Packages received router"
   g.type 'line_chart3'
   g.mapping :x_axis => :oml_ts_client, :y_axis => :rx_packets_avg, \
    :group_by => :name
   g.xaxis :legend => 'time [s]'
```

```
  g.yaxis :legend => '# packages', :ticks => {:format => 's'}
end


defGraph 'Packages sent router by interface' do |g|
   g.ms('network').select {
     [ :oml_sender_id, :oml_ts_client, :oml_ts_server, :name, :tx_packets_avg ]
   }
   g.caption "Packages sent router"
   g.type 'line_chart3'
   g.mapping :x_axis => :oml_ts_client, :y_axis => :tx_packets_avg,\
    :group_by => :name
   g.xaxis :legend => 'time [s]'
  g.yaxis :legend => '# packages', :ticks => {:format => 's'}
end
```

Análisis experimental: Experimento 2

El siguiente archivo corresponde al segundo experimento descrito en la sección análisis experimental. El experimento consiste en la medición del tráfico entre dos nodos a través de un red de nodos intermedios con un total de dos caminos posibles entre ambos.

```
loadOEDL('http://10.0.0.7/oml-apps/nmetrics_oml2.rb')
loadOEDL('http://10.0.0.7/oml-apps/iperf_oml2.rb')

# Hosts
defProperty('c1', 'contenedor-c1')
defProperty('c2', 'contenedor-c2')
defProperty('c3', 'contenedor-c3')
defProperty('c4', 'contenedor-c4')
defProperty('c5', 'contenedor-c5')
defProperty('c6', 'contenedor-c6')

# Network first ip
defProperty('c1c2', '1.2.0.0')
defProperty('c2c3', '1.2.2.0')
defProperty('c2c4', '1.2.1.0')
defProperty('c3c5', '1.2.4.0')
defProperty('c4c5', '1.2.3.0')
defProperty('c5c6', '1.2.5.0')
```

```
# Network c1c2 interface conf
defProperty('c1_netc1c2', '1.2.0.1')
defProperty('c2_netc1c2', '1.2.0.2')

# Network c2c3 interface conf
defProperty('c2_netc2c3', '1.2.2.1')
defProperty('c3_netc2c3', '1.2.2.2')

# Network c2c4 interface conf
defProperty('c2_netc2c4', '1.2.1.1')
defProperty('c4_netc2c4', '1.2.1.2')

# Network c4c5 interface conf
defProperty('c4_netc4c5', '1.2.3.1')
defProperty('c5_netc4c5', '1.2.3.2')

# Network c3c5 interface conf
defProperty('c3_netc3c5', '1.2.4.1')
defProperty('c5_netc3c5', '1.2.4.2')

# Network c5c6 interface conf
defProperty('c5_netc5c6', '1.2.5.1')
defProperty('c6_netc5c6', '1.2.5.2')

# Networks size
defProperty('network_prefix_size', '28')

def masked_ip(ip)
  "#{ip}/#{property.network_prefix_size}"
end

defGroup('c1', property.c1) do |g|
  g.e1.ip = masked_ip property.c1_netc1c2
  # Add client
  g.addApplication("iperf") do |app|
    app.setProperty('client', property.c6_netc5c6)
    app.setProperty('port', 8080)
    app.setProperty('bandwidth', '10mbit/s')
    app.setProperty('time', 60)
  end
end

defGroup('c2', property.c2) do |g|
```

```
  g.e1.ip = masked_ip property.c2_netc2c3
  g.e2.ip = masked_ip property.c2_netc1c2
  g.e3.ip = masked_ip property.c2_netc2c4
end

defGroup('c3', property.c3) do |g|
  g.e1.ip = masked_ip property.c3_netc3c5
  g.e2.ip = masked_ip property.c3_netc2c3

  # Measure traffic
  g.addApplication("nmetrics") do |app|
    app.setProperty('interface', 'eth2')
    app.measure('network', :interval => 1)
  end
end

defGroup('c4', property.c4) do |g|
  g.e1.ip = masked_ip property.c4_netc4c5
  g.e2.ip = masked_ip property.c4_netc2c4oedl

  # Measure traffic
  g.addApplication("nmetrics") do |app|
    app.setProperty('interface', 'eth2')
    app.measure('network', :interval => 1)
  end
end

defGroup('c5', property.c5) do |g|
  g.e1.ip = masked_ip property.c5_netc3c5
  g.e2.ip = masked_ip property.c5_netc4c5
  g.e3.ip = masked_ip property.c5_netc5c6
end

defGroup('c6', property.c6) do |g|
  g.e1.ip = masked_ip property.c6_netc5c6

  # Measure network traffic
  g.addApplication("nmetrics") do |app|
    app.setProperty('interface', 'eth1')
    app.measure('network', :interval => 1)
  end

  # Add tcp server
  g.addApplication("iperf") do |app|
```

```
    app.setProperty('server', true)
    app.setProperty('port', 8080)
  end
end

# Networks with mask
netc1c2 = masked_ip property.c1c2
netc2c3 = masked_ip property.c2c3
netc2c4 = masked_ip property.c2c4
netc3c5 = masked_ip property.c3c5
netc4c5 = masked_ip property.c4c5
netc5c6 = masked_ip property.c5c6

defEvent :ROUTES_CONFIGURED do |state|
  count = 0

  state.each do |resource|
    if (resource.type == 'application') && (resource.state == 'stopped')
      count = count + 1
    end
  end
  count == 5
end

onEvent :ALL_UP do

  after 5 do
    group('c1').exec("ip route add #{netc5c6} via #{property.c2_netc1c2}")

    group('c2').exec("ip route add #{netc5c6} via #{property.c4_netc2c4} preference 1")
    group('c2').exec("ip route add #{netc5c6} via #{property.c3_netc2c3} preference 2")

    group('c3').exec("ip route add #{netc5c6} via #{property.c5_netc3c5}")
    group('c4').exec("ip route add #{netc5c6} via #{property.c5_netc4c5}")
  end


  after 30 do

    group('c3').startApplications
    group('c4').startApplications
    group('c6').startApplications

    after 15 do
```

```
      group('c1').startApplications
    end

    after 35 do
      # drop link
      group('c2').exec('ip addr flush dev eth3')
    end

    after 200 do
      # Stop all the Applications associated to all the Groups
      allGroups.stopApplications
      # Tell the Experiment Controller to terminate the experiment now
      Experiment.done
    end
  end
end


defGraph 'Received packages' do |g|
   g.ms('network').select {[ :oml_sender_id, :oml_ts_client, :oml_ts_server, :rx_packet
   g.caption "Received packages"
   g.type 'line_chart3'
   g.mapping :x_axis => :oml_ts_client, :y_axis => :rx_packets_avg, :group_by => :oml_s
   g.xaxis :legend => 'time [s]'
  g.yaxis :legend => '# packages', :ticks => {:format => 's'}
end
```