UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

LXC platform manager

Trabajo fin de grado

Grado en Ingeniería Informática

Autor: Tomás Roig Martinez

Tutor: Ignacio Blanquer Espert
Carlos de Alfonso Laguna

Curso 2016-2017

Resumen

Actualmente, la mayoría de compañías punteras lanzan sus productos en la nube. Este concepto de nube viene dado por una aglomeración de servicios que se pueden escalar, para ajustarse a la demanda de los usuarios.

En los últimos tiempos, las plataformas en la nube se están moviendo hacia un modelo de gestión de recursos basados en contenedores. Actualmente *Docker* es prácticamente el estándar, aunque solo proporciona aplicaciones con un único proceso. *LXC* en cambio, ofrece contenedores de sistemas completos, sin el sobrecoste de la virtualización de sistemas tradicional. *LXC* es capaz de gestionar contenedores ejecutados en la misma máquina donde está instalado, mientras que *LXD* ofrece una interfaz para gestionarlos de manera remota.

En la actualidad no existe una herramienta que permita gestionar varios *host* con *LXD*de manera conjunta, por eso, en este proyecto desarrollamos un gestor de plataforma sobre *LXD* que permite controlar diferentes nodos *hardware* con esta tecnología, con el fin de facilitar el proceso de despliegue y gestión de recursos, a la comunidad en torno a la herramienta de virtualización.

Palabras clave: lxc, contenedores, sistemas, gestor,imágenes,LXD,Docker,Docker-Swarm

Resum

Actualment, la majoría de companyies punteres llancen els seus productes en el núvol. Aquest concepte de núvol ve donat per l'aglomeració de serveis que es poden escalar, per tal d'ajustarse a la demanda dels usuaris.

En els últims temps, les plataformes en el núvol están fent un moviment cap a un model de gestió de recursos basats en contenidors. Actualment *Docker* és prácticament l'estàndard, encara que sols proporciona aplicacions amb un únic proccés. *LXC*, en canvi, ofereix contenidors de sistemes complets, sense l'afegit de la virtualització de sistemes tradicional. *LXC* es capaç de gestionar contenidors executats en la mateixa màquina on està instal·lat, mentre que *LXD* ofereix una interfície per a gestionar-los de manera remota.

A l'actualitat no existeix una ferramenta que permeta gestionar diversos *host* amb *LXD* en conjunt, per això, a aquest projecte desenvoluparem un gestor de plataforma fent ús de *LXD* que permeta controlar diversos nodes hardware amb aquesta tecnología, amb la finalitat de facilitar el proccés de desplegament y gestió de recursos, a la comunitat en torn a la ferramenta de virtualització.

Paraules clau: lxc, contenidors, sistemes, gestor,imatges,LXD,Docker,Docker-Swarm

Abstract

Nowadays, the majority of high technology companies launch their products in the cloud. This concept of cloud comes from an agglomeration of services being able to be scaled, to suit the needs of users.

In the last times, platforms on the cloud are making a move towards a new resource management model based on containers. At the present time, *Docker* is practically the standard, even though it only provides services with a single process. *LXC*, instead, offers full system containers, without the overrun that often comes with traditional virtualization systems. *LXC* is able of managing local hosted containers, while *LXD* offers an interface to manage containers in remote.

Nowadays there isn't a tool allowing the management of several hosts with *LXD* . That is why, in this project, we will develop a platform manager over *LXD* that will allow control over different hardware nodes, with the final purpose of easing the process of deployment and management for the community around the virtualization tool.

Key words: lxc, containers ,systems, manager, images,LXD,Docker,Docker-Swarm

Índice general

Apéndices

Índice de figuras

CAPÍTULO 1
Introducción

La noción de computación en la nube no es nueva. Desde los años 60 se ha buscado ofrecer servicios a través de la red. La idea era ser capaces de disponer de recursos informáticos bajo demanda de forma remota como si se tratara de recursos propios, en forma de servicio. Hoy en dia, las grandes empresas tecnológicas, tanto a nivel interno como externo se valen de estas tecnologias para satisfacer la demanda de sus millones de usuarios. A nivel operativo, sus arquitecturas se basan en nodos de cómputo, conectados entre sí, pudiendo conformar un sistema mediante diferentes servicios en cada uno de estos nodos. Inicialmente estos nodos eran *hardware*, para pasar a ser simulados mediante máquinas virtuales, y llegar posteriormente a convertirse en los contenedores de hoy en dia. La finalidad de este Trabajo de Fin de Grado, es desarrollar un gestor de plataforma para *LXC* , una de las tecnologías que implementan esta abstracción de nodos de computación, con el fin de facilitar el despliegue de sistemas basados en esta tecnología.

La meta de este proyecto es posibilitar la gestión coordinada del despliegue de los recursos de un sistema de este tipo, así como promover y crear comunidad en torno a *LXC* frente a otras alternativas. Para ello, el gestor de plataforma cuenta con diferentes funcionalidades que permiten el lanzamiento automático de una infraestructura, dados sus nodos y una red. Se analizará la mejora de esta aproximación contra el despliegue a mano de los contenedores[1] y se comparará con *Docker* y *Docker-Swarm* su gestor de nodos host, la solución estándar a dia de hoy.

1.1 Motivación

A lo largo de mis estudios de grado, y con el fin de estar mejor preparado para el futuro laboral, siempre he intentado probar nuevas herramientas a la hora de desarrollar pequeños proyectos con aplicación. A menudo, requerían instalar dependencias con librerias externas, controlar versiones de lenguajes, gestionar variables de entorno, y pronto me di cuenta de que un entorno de desarrollo podia convertirse fácilmente en un caos, pudiendo hacer que lo que un sistema funcionase, en otro dejara de funcionar correctamente. Por suerte descubrí los entornos

[1] En el contexto de este TFG, nos referiremos a los nodos como containers o contenedores.

de máquinas virtuales, que hacían más fácil aislar estos problemas al encapsular un sistema operativo con toda su funcionalidad, aislando el entorno físico de desarrollo de todos los problemas que pudieran ocurrir, y pudiendose portar a otro host muy fácilmente. No obstante, estos entornos eran pesados, y hacían uso de demasiados recursos en la máquina host. Cuando empezé a trabajar en una startup, fué cuando me di cuenta de la importancia de aislar entornos, pues la aplicación debía pasar por las fases de *development* y *staging*, para finalmente ser lanzadas a producción. Fué entonces cuando descubrí Docker. Esta tecnología nos permitió encapsular los diferentes servicios en containers, y además, según su entorno, conectarse a servicios locales o en la nube. Me fascinó la rapidez que podíamos alcanzar para entregar una nueva versión de la aplicación, y lo ligeras que eran las imágenes[2] comparándolas con máquinas virtuales que emulaban un sistema operativo completo, aunque en este caso *Docker* encapsule en contenedores servicios, y no sistemas completos.

Esto fué lo que hizo que me interesara por este Trabajo Final de Grado. Me preguntaba como estarían diseñados estos sistemas y si podrían mejorarse, porque sé de buena mano lo importantes que pueden ser para desarrolladores de todo el mundo. Con *LXC*, vi la oportunidad de explorar el núcleo de un proyecto de este tipo, y aprender como diseñar una herramienta de estas características. Asimismo, me gusta probar nuevos lenguajes de programación y ver sus fortalezas y debilidades, y con este proyecto, he podido finalmente probar *Golang*, un lenguaje de programación que implementa la concurrencia con construcciones del propio lenguaje, y en el que están escritos tanto *LXC* como *Docker*.

En conclusión, el trabajar con estas tecnologías me puede abrir puertas hacía el futuro, cada vez más brillante, del cloud computing, en el que se basan más y más organizaciones. También me da la oportunidad de probar una nueva herramienta, similar en funcionalidad a *Docker* aunque con distinta implementación, y ver si con las facilidades que incorporaré, ayuda a fomentar la comunidad *LXC*, así como su popularidad.

1.2 Objetivos

El objetivo principal del proyecto es el diseño e implementación de un gestor de plataforma para la tecnología *LXC*. Esto es, crear una capa software por encima de esta tecnología, que de manera transparente, pueda gestionar un conjunto de nodos que actúan como hosts para contenedores *LXC*. Cuando hablamos de transparencia, se quiere dar a entender que la manera de interactuar con un simple nodo mediante mensajes *JSON* enviados a una *API REST*, va a ser exactamente la misma que cuando se comuniquen con el gestor de plataforma, con la finalidad de integrar un front end web existente para facilitar la gestión de los nodos. Finalmente se analizará la herramienta y se comparará con una solución similar existente para *Docker*, *Docker-swarm*.

Para cumplir este objetivo principal, va a ser necesario desglosar el objetivo principal en tareas, que son las siguientes:

[2] Versión portable de un container.

- Estudiar la implementación de la tecnología *LXC*.
- Implementar la lógica de una API equivalente, que acepte las mismas instrucciones que la original, y afecte a los nodos que correspondan.
- Hacer posible el lanzamiento de tareas asíncronas.
- Diseñar un planificador para el lanzamiento de contenedores.
- Implementar un almacén de imágenes desde las que distribuirlas para todos los hosts.
- Comparar el resultado de utilizar el gestor de plataforma contra utilizar *Docker-swarm*.
- Añadir el frontend web a la *API* implementada.

1.3 Estructura del proyecto

El primer punto de este documento trata la introducción del tema del TFG y sus objetivos, así como las subtareas que conforman el proyecto. Se explica la motivación para elegir este proyecto y su estructura.

El segundo punto trata la actualidad de la virtualización y las distintas soluciones existentes similares a *LXC*. También se habla de la comunidad en torno a las herramientas existentes y los distintos usos que se les dan. Finalmente, reflexiona acerca de la actualidad tecnológica y el por qué de su estado actual.

El tercer capítulo esta dedicado al diseño del proyecto. Primero se analiza el problema que plantea el proyecto y se explica la relación entre los distintos elementos que conforman el sistema a implementar. Después se comentan las herramientas y técnicas utilizadas para el desarrollo y finalmente se explica la planificación seguida durante el proyecto.

El cuarto capítulo trata los detalles de implementación del proyecto, los pasos que se siguieron, los requisitos y la solución a los distintos problemas que se encontraron durante la implementación.

En el quinto capítulo continúa comparando el uso de *Docker-Swarm* con el de nuestra plataforma, mostrando ejemplos de ambas plataformas.

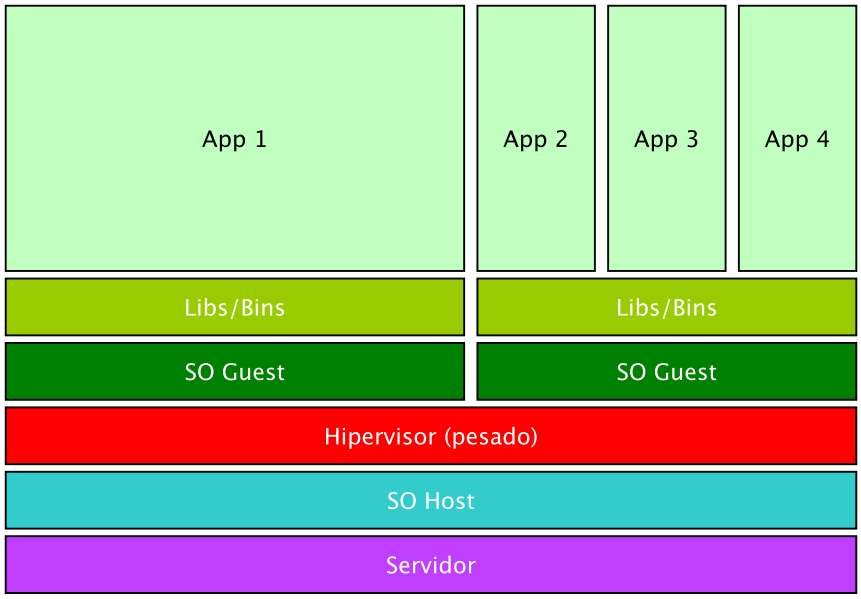
El sexto contiene las conclusiones finales del trabajo realizado, los trabajos futuros derivados del actual y los agradecimientos.

Finalmente se incluye la bibliografía utilizada para consulta de información relacionada con el proyecto, y distintos documentos para facilitar la comprensión del tema tratado.

CAPÍTULO 2
Estado del arte

En la actualidad, existen numerosas soluciones de virtualización. Antes de enumerar las diferentes soluciones, hay que dejar claro que se trata de virtualización a nivel de sistema operativo, un método de virtualización en el que el kernel de un sistema operativo permite la existencia de multiples espacios de usuario aislados en lugar de sólo uno.

Figura 2.1: Pila de tecnologías requeridas para la virtualización tradicional. Nótese en rojo el hipervisor, cuello de botella de este tipo de virtualización. Se pueden ejecutar varias aplicaciones, pero requiere simular un sistema operativo completo, consumiendo recursos limitados por el host.



En este paradigma el coste de la virtualización es muy bajo, ya que los programas en particiones virtuales utilizan la interfaz de llamadas al sistema que provee el propio sistema host, es decir no necesita estar sujeta a emulación, o ser ejecutada en una máquina virtual intermedia, como en el caso de virtualización de sistemas completos o la paravirtualización.

También se trata de una solución menos flexible en cuanto a los sistemas que puede virtualizar, pues se reduce simplemente al sistema que sirve de host para la

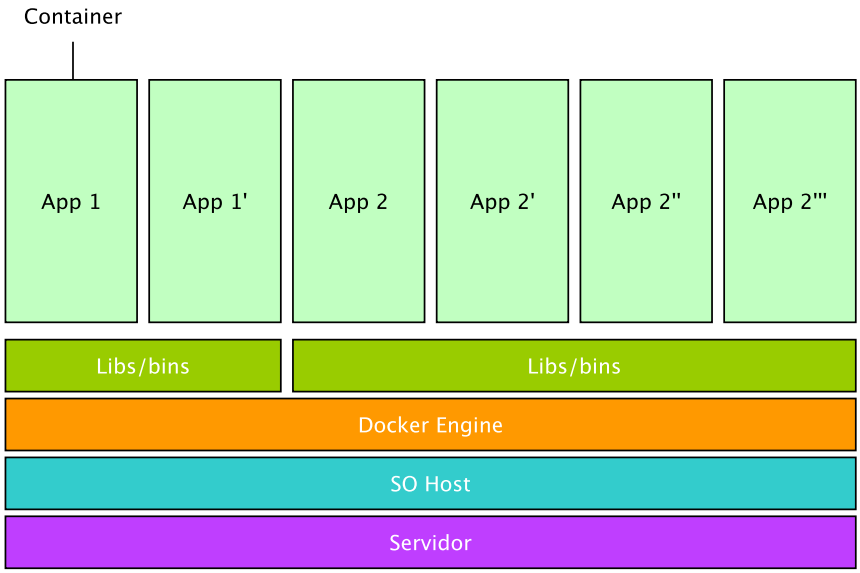
virtualización. No obstante, sigue siendo útil, pues permite aislar entornos, protegiendolos de amenazas externas, o previniendo que escapen de ese aislamiento, y por supuesto, permite asegurar que el código seguirá funcionando al desplegar la imagen del contenedor en otro host con la misma tecnología de virtualización. En la práctica, muchas grandes empresas lo utilizan como medio para aumentar su seguridad interna, reducir el tiempo de onboarding de los empleados, así como para reducir el coste temporal y operativo de lanzar a producción un servicio después de actualizarlo y la entrega de aplicaciones sin requerir la instalación evitando los problemas de dependencias.

Estas son las soluciones existentes más utilizadas actualmente:

- Docker
- LXC
- Kubernetes pods
- CoreOS rkt pods

Docker

Figura 2.2: Virtualización mediante Docker. Docker utliza su motor para gestionar los contenedores, que pueden compartir dependencias. En la imagen, cada aplicación representa a un contenedor. Si hay dos aplicaciones x1 y x2, con dependencias comunes, Docker las gestiona con su sistema de persistencia por capas para evitar duplicaciones y ganar en eficiencia.



Docker se puso a la cabeza como tecnología estándar de contenedores. Se basa en utilizar llamadas al kernel de un sistema operativo, que actualmente puede ser Windows, Linux o Mac. Los contenedores comparten el kernel del host, y aunque

se puedan crear imágenes con base en un sistema operativo, solo carga la parte del modo de usuario. Se trata de una solución que:

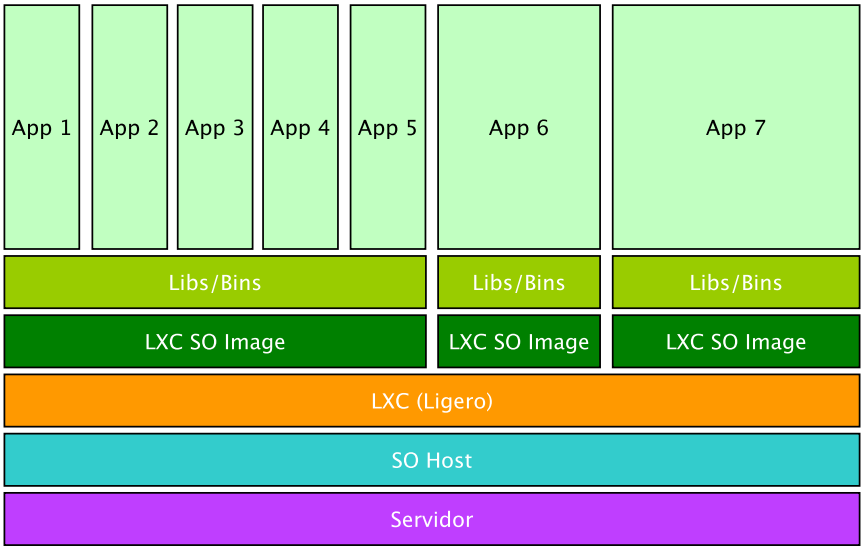
- Encapsula un contenedor en un proceso, que después puede ser gestionado con su ecosistema de herramientas
- Contiene la configuración de la aplicación para no repetir procesos manuales
- Ha conseguido crear una gran comunidad en torno a la herramienta
- Utiliza un sistema de archivos por capas, que hace las imágenes ligeras
- Tiene soluciones que se pueden incluir para dar soporte a redes, almacenamiento y autenticación
- Suele ser utilizado para entornos de desarrollo, así como aplicaciones basadas en microservicios
- Puede controlar cuotas de uso de recursos

Por contra, el hecho de que no tenga habilitado el almacenamiento persistente por defecto[1], que no tenga acceso a multiples daemons del sistema host, y que utilice la misma ip del host más un puerto como punto de entrada, pudiendo afectar a aplicaciones tradicionales, hace que la industria no acabe de adoptarlo para la mayoría de tareas.

[1] Se pueden configurar los contenedores para que compartan capas de almacenamiento con el host.

LXC

Figura 2.3: Virtualización mediante LXC. En este caso, como se aprecia en la figura, LXC actúa como hipervisor ligero, permitiendo usar imágenes de sistemas con el mismo kernel que el del host. A diferencia de Docker, un contenedor permite la ejecución de diferentes aplicaciones.



LXC, a pesar de no ser tan reciente, también ha encontrado su lugar en la industria. En lugar de estar centrado en aplicaciones, su foco de desarrollo persigue ofrecer contenedores de sistemas operativos. A diferencia de las máquinas virtuales tradicionales, esta herramienta satisface los casos de uso de estas, con la ventaja de no consumir apenas recursos físicos, ya que utilizan el kernel del sistema host.

En concreto, utilizan una mezcla de elementos de seguridad que son el espacio de nombres, control de acceso mandatorio y grupos de control. La principal diferencia con *Docker* es que no encapsula procesos, sino que son sistemas enteros. Tristemente, incluso siendo más antiguo que *Docker*, *LXC* no tiene tanta comunidad en torno, a pesar de poder hacer básicamente lo mismo, e incluso más. Esto puede deberse a la inmadurez de la herramienta inicial, y a el gran trabajo de *Docker*y su ecosistema, para facilitar y fomentar su uso por la comunidad.

Kubernetes pods

Muy similar a *Docker*, los pods de *Kubernetes* son contenedores de aplicaciones en un solo proceso, aunque al parecer, *Kubernetes* soporta más entornos de ejecución que solamente *Docker*.

CoreOS rkt pods

Es conceptualmente lo mismo que *Docker* , con la diferencia de que su daemon puede ser integrado con el arranque del sistema. En el caso de *Docker*, el lanzamiento de contenedores es gestionado por el *daemon containerd*, haciendo que el ciclo de vida de los contenedores no sea accesible por el daemon de inicio del sistema.

Existían y existen otras herramientas, pero han ido cayendo en desuso en pro de nuevas herramientas, han sido absorbidas por las enumeradas, o están fuertemente vinculadas a un nicho de sistemas operativos. Hay que destacar que actualmente, la mayoría de tecnologías en desarrollo dentro de este ámbito, se agrupan bajo la *OCI*[2], con el fin de establecer un estándar de formato de contenedores para la industria.

En conclusión, podemos decir que mientras la mayoría se esfuerza por crear contenedores de aplicaciones aisladas, *LXC* sigue un camino distinto, que es sustituir la virtualización de sistemas operativos, tradicionalmente costosa, por un método mucho más ligero, que evita la ejecución de un kernel separado y la simulación del hardware del sistema.

2.1 Crítica al estado del arte

Después de analizar las diferentes tecnologías, y experimentar con sus herramientas, el hecho de que las tecnologías de containerización de aplicaciones tengan más adeptos viene dado por su estandarización propuesta por la *OCI*, la creación de comunidades open-source, como por ejemplo su comunidad de creación y distribución de imágenes, y un rápido ritmo de desarrollo.

Pese a ser una solución muy válida, pues *LXC* es capaz de contener sistemas enteros con aplicaciones multiproceso frente los contenedores monoproceso de la *OCI* , *LXC* no cuenta con tantos seguidores. La mayoría del desarrollo es llevado a cabo por Stéphane Graber, el líder del equipo que impulsa esta tecnología, a pesar de ser open-source. Es interesante ver que su página web personal contiene más información útil en cuanto al uso de *LXC* que la propia web de la tecnología. Además, sólo existe una herramienta, *LXD*, para ayudar al usuario a gestionar los contenedores, que cumple la función de *API REST* , frente a todas las que existen para *Docker* .Sin embargo, no se debe considerar este hecho como una limitación, ya que LXC y Docker se orientan a nichos de mercado diferente. En los últimos meses están surgiendo numerosos entornos de gestión de contenedores y por tanto vemos como una oportunidad el apostar por un desarrollo que dentro de la comunidad de LXC no tiene todavía competidores.

También es interesante ver como, a nivel de imágenes disponibles, la *OCI* lleva una gran ventaja. *Docker* tiene varias formas de distribuir sus imágenes, tanto de forma abierta para la comunidad, como mediante su tienda para empresas. *LXC* por su parte reduce el foco a distribuir imágenes de sistemas operativos base, a pesar de que, cómo *Docker* hace con aplicaciones, podría ofrecer imágenes con

[2]Open Container Initiative

pilas de tecnologías listas para usar, como por ejemplo el LAMP[3], consiguiendo dos objetivos: mayor facilidad de uso para el usuario y promoción entre desarrolladores.

Así pues, señaladas las mejoras que se pueden realizar sobre esta tecnología, procederemos a diseñar nuestra herramienta para agrupar y gestionar múltiples nodos hardware con la tecnología *LXD/LXC*, con el fin, tanto de facilitar las tareas de gestión de contenedores, como de mejorar la comunidad en torno a la tecnología.

[3]Servidor web compuesto por Linux, Apache, MySQL y PHP.

CAPÍTULO 3
Diseño del sistema

En este apartado se explican los factores que se han tenido en cuenta a la hora de diseñar el gestor de plataforma para *LXC*. Se explica la fase de análisis de los requisitos del proyecto, las herramientas utilizadas, las técnicas aplicadas, la metodología de desarrollo y la planificación llevada a cabo.

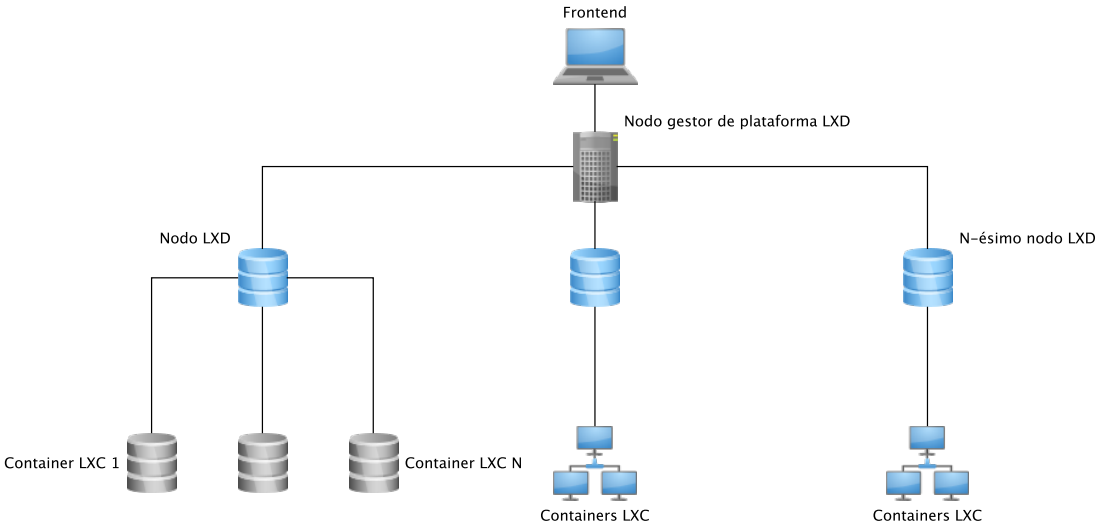
3.1 Análisis del proyecto

Para poder llevar a cabo este proyecto, hubo que estudiar el problema, y los distintos componentes que deben interactuar, y como debería comportarse el software desarrollado.

3.1.1. El problema

La necesidad principal que tiene como meta cubrir este proyecto es gestionar un conjunto de servidores con la tecnología *LXC*, a través de un sólo servidor. *LXC* por si mismo no contiene un mecanismo para comunicarse con difentes nodos a través de la red, pero una herramienta escrita por el mismo grupo de desarrolladores, *LXD*, sí que lo permite. Esta herramienta es un *daemon* que controla los contenedores LXC dentro del host en el que esta instalado y permite la gestión externa a través de una *API REST*. Por tanto, usar esta herramienta nos permitiría reducir la complejidad del problema.

Figura 3.1: El problema, representado visualmente. Se muestra la multiplicidad de los elementos que interactuarán en nuestro gestor de plataforma.



Por otra parte, tenemos la restricción de que este gestor sea manipulado por gente no necesariamente experta en el uso del shell de *Linux*, a través de un frontend web ya escrito para LXD. Nos interesa, por tanto, hacer compatible nuestro gestor con ese frontend, y gestionar todos los servidores host desde ahí, además de reducir la curva de aprendizaje.

3.2 Herramientas utilizadas

Durante el desarrollo del proyecto, se han utilizado algunas herramientas. A continuación se describirá el uso de cada una de ellas con relación al desarrollo del proyecto. En algunos casos se tratan de herramientas de gestión y en algunos casos tienen relación con el código, para facilitar el desarrollo del proyecto.

LXC Linux Containers es la tecnología subyacente que utilizamos para interactuar con el sistema. Esta tecnología implementa los contenedores, y es la que se comunica con el sistema a bajo nivel. Con el fin de hacerla disponible a más desarrolladores, tiene bindings en varios lenguajes de programación. Está escrita en lenguaje *C*.

LXD Es un *daemon* del sistema, un cliente de línea de comandos y un plugin para *Nova* de *OpenStack*. Es como un *hypervisor* que permite gestionar los contenedores a través de una *REST API* . Está escrita en *Go* y utiliza el binding de *Go* con *LXC*.

SQlite Es una base de datos muy ligera basada en el lenguaje SQL que nos permite persistir los datos asociados a los recursos gestionados.

Curl Para enviar las peticiones *HTTP*, se utiliza la herramienta *Curl* del *shell*.

Go La elección de este lenguaje para la implementación surge de algunas características interesantes que contiene dicho lenguaje, además de ser el lenguaje

de implementación de *Docker* y *LXD*. Este lenguaje hace sencillo el uso de concurrencia mediante su sintaxis, y teniendo en cuenta que se van a hacer muchas peticiones HTTP, nos interesa hacerlas lo más rápido posible.

Shell La línea de comandos es vital para el desarrollo del proyecto, pues todas la interacciones con el servidor a la hora de actualizarlo, lanzarlo o gestionarlo, pasan por aquí. Se utiliza también para el acceso a los host habilitados en el laboratorio para el desarrollo del proyecto.

Git Esta herramienta es la utilizada para gestionar las versiones del software del proyecto. Se creó un repositorio privado para el desarrollo, y se fueron creando las ramas representando las características, para después ir uniendolas en la versión estable de la herramienta.

Yed Esta herramienta es un editor de diagramas, con soporte para su exportación en pdf o svg, útil para la creación de los diagramas de este trabajo, que espero faciliten la comprensión del mismo.

3.3 Técnicas aplicadas

Para este proyecto, se han aplicado varias técnicas para enlazar los distintos elementos del problema y/o mejorar la eficiencia del sistema.

API REST Una API[1] es una capa de abstracción, generalmente un conjunto de operaciones sobre un servicio o software. Una API con este apellido, indica que debe cumplir ciertas características:

- La comunicación entre cliente y servidor debe ser sin estado. En nuestro caso, los JSON contienen toda información necesaria.
- El conjunto de operaciones definidas está basado en CRUD[2], aunque al tratarse del protocolo HTTP, estas acaban siendo POST, GET, PUT y DELETE. En algunos caso, dependiendo de la versión de *LXD*, también se añade el PATCH, que a diferencia de la operación PUT, no reemplaza el recurso completamente, sino que lo modifica.
- Una sintaxis universal para identificar los recursos. En este caso, las diferentes URLs identifican inequívocamente cada recurso.
- Requiere el uso de hipermedios. En nuestro caso, JSON es el hipermedio, representando el estado actual de un recurso.

Concurrencia En nuestra disciplina, se entiende por concurrencia la propiedad por la cual un problema puede descomponerse en subproblemas sin orden, o con un orden parcial, para poder ser procesados individualmente en paralelo sin alterar el resultado del problema principal sin paralelizar. En nuestro caso, algunas operaciones GET pueden tardar en completarse si hay demasiados nodos en la red. Para evitar esto, se utiliza la concurrencia para hacer peticiones HTTP en paralelo, con el fin de conseguir inmediatez de resultados para el usuario final.

[1] Aplication Programming Interface (Interfaz de programación de aplicaciones)
[2] Create,Read,Update y Delete.

Tareas en background En algunos casos, hay operaciones de LXC que pueden tardar en realizarse. La creación de una imágen a partir de un snapshot, el despliegue de un contenedor, la descarga de imágenes de un endpoint... Como estas operaciones son asíncronas por defecto, en el proyecto también hemos tenido que implementar este comportamiento para cuando hay que realizar este tipo de operaciones sobre un conjunto de nodos. La API devolverá un JSON con la creación de la operación, y se podrá consultar su estado de completitud consultando la URL del recurso adecuado.

Revertir operaciones En algunos casos, solo interesa que se lleven a cabo las operaciones, si y solo si se completan correctamente en todos los nodos involucrados. Para ello, en algunas operaciones, como en el caso de distribución de imágenes en todos los nodos, la operación espera en background a que todos los nodos contesten positivamente. En el caso de que uno de ellos falle, se volverá al estado anterior a ejecutar la operación.

Planificador Con el fin de automatizar aún más el despliegue de los contenedores, se utiliza la técnica del planificador de recursos, para abstraer el despliegue de los contenedores de su localización. El planificador implementa diferentes algoritmos de despligue de recursos.

- *Aleatoria* : El planificador debe distribuir los contenedores de forma aleatoria.
- *Distribuir* : El planificador debería repartir equitativamente los contenedores entre los host.
- *Concentrar* : El planificador debe maximizar la densidad de cómputo de los hosts.

3.4 Metodología de desarrollo

Para tratar este proyecto, se podrían haber seguido diferentes metodologías para tratarlo y llevarlo a termino, y el resultado final sería el mismo. No obstante, debido a ciertas características de este trabajo y las herramientas utilizadas, se ha seguido una metodología de la familia de las Ágiles que es la *Lean*. Esta metodología está basada en 7 principios:

- Eliminar los desperdicios: eliminar todo lo que no añade valor al cliente.
- Amplificar el aprendizaje: el desarrollo es un aprendizaje contínuo con ciclos cortos iterativos, en los que los desarrolladores, además de mediante pruebas y refactorización, pueden aprender del cliente integrándolo en dichas pruebas.
- Decidir lo más tarde posible: se trata de ir implementando partes menos indispensables, y retrasar las decisiones críticas a cuando los clientes tengan más claros los requisitos.
- Entregar tan rápido como sea posible: se trata de ir ofreciendo versiones tan pronto como estén listas. Aunque estas puedan cambiar en la iteración

siguiente, el cliente satisface sus requisitos de puntualidad, y se pueden preparar los cambios sin retrasos.

- Capacitar al equipo: se trata de una inversión de papeles de los estilos de gestión tradicionales. Los gestores escuchan, aprenden y dejan hacer al equipo, que son los que saben acerca del problema a resolver.
- Construir integridad intrínseca: significa que desde un primer momento, todos los elementos del sistema deben funcionar en conjunto. También, el sistema deberia ser refactorizado según se van incluyendo elementos. Finalmente, el sistema debería de ser probado para verificar que los requisitos del cliente se cumplen.
- Ver el conjunto: se refiere a tener todos los puntos anteriores en cuenta antes de que el equipo se ponga a desarrollar.

En nuestro caso particular, esta metodología nos ha permitido tener una *API REST* casi desde el inicio, que sería nuestro conjunto y mediante iteraciones, se han ido añadiendo las funcionalidades a medida que se iban realizando minireuniones para verificar el estado del proyecto. También ha resultado positivo al aplicarla teniendo en cuenta el aprendizaje y uso de un nuevo lenguaje de programación, pues debía ir aprendiendo el lenguaje, mientras entendía la tecnología y iba haciendo avances. Y por supuesto, con la meta de mantener la base de código ágil, durante el transcurso del proyecto se ha ido refactorizando el código a medida que se iban añadiendo las nuevas funcionalidades.

3.5 Planificación

Para la planificación, se procedió a desglosar el proyecto en tareas, para posteriormente puntuarlas de mayor a menor prioridad. A partir de esta puntuación, se escogían las tareas prioritarias, y se les daba una estimación de tiempo. Después, se utilizaban para formar un *sprint* de 2 semanas, intentando acabar el máximo de tareas asignadas para ese espacio de tiempo.

1er Sprint: Estudio de las tecnologías. Para poder implementar un gestor de contenedores *LXC* a través de varios nodos *LXD* , ha sido necesario estudiar los conceptos que implementan. Se estudió el pasado y estado actual de las tecnologías de virtualización, y finalmente se adquirieron los conocimientos necesarios sobre las tecnologías que finalmente se han utilizado en el proyecto. También se aprovechó para desplegar la infraestructura necesaria para las pruebas de implementación.

2do Sprint: Definición del producto mínimo viable. A partir del estudio de las tecnologías, se procedió a diseñar el sistema para satisfacer los requisitos. El sistema deberia pues, aceptar los mismos mensajes *JSON* que recibe la tecnología final, actuando de intermediario, y teniendo la capacidad de desplegar contenedores y operar sobre ellos en distintos *host*. El gestor debe permitir las mismas operaciones sobre contenedores.

3er Sprint: Estructura mínima del gestor de plataforma. Con esto, se diseñó una *API* equivalente, con los mismos *endpoints*[3] que el software a gestionar. Inicialmente se implementó el servidor que mantendría el servicio activo. Para facilitar el montaje de los diferentes nodos, en lugar de utilizar el sistema de certificados de *LXD*, se optó por utilizar *Curl* y *SSH* para enviar los comandos a los diferentes *endpoints* a través de *unix sockets*[4]. Para esto, los nodos que se quieran gestionar con este software deben tener la clave pública del nodo principal, y un usuario del sistema para poder ejecutar los comandos.

4to Sprint: Estructura mínima del gestor de plataforma. En este punto, ya se habían implementado los primeros endpoints de contenedores, y funcionaban correctamente, junto con la base de datos. Se detectó una repetición de patrones en el proceso de análisis sintáctico de las respuestas de los diferentes hosts, y en la ejecución de los comandos.

5to Sprint: Refactorización de la estructura mínima. Después de verificar que se podía mejorar la calidad del código, se procedió a refactorizar la parte de proceso de las respuestas y ejecución de los comandos en los hosts remotos, para poder reutilizarlos en otras partes del código en operaciones con varias fases. Se allanó el camino para los endpoints restantes, ofreciendo una referencia para su implementación.

6to Sprint: Implementación de otros endpoints. Después de los contenedores se decidió implementar los endpoints de *imágenes* y de *perfiles de ejecución*, por considerarlos más importantes. De estos, no toda la funcionalidad está implementada, por no considerarlos oportuno para la primera versión. Al principio, se especificaba el host en el que los contenedores debían operar, pero se cambió este comportamiento mediante un planificador, con su propio endpoint, para evitar la incompatibilidad con la *API*subyacente. Este planificador se encarga de las diferentes estrategias de lanzamiento de contenedores.

7mo Sprint: Validación. Previo a documentar el proyecto, se procedió a validar todas las operaciones para comprobar que fueran correctas en la *API* implementada. Para ello se definieron pasos para crear, destruir, o modificar recursos en cada endpoint.

8vo Sprint: Documentación. Finalmente se documentó todo lo desarrollado en el proyecto, para prepararlo para futuras extensiones y mejoras.

[3] Puntos de acceso a los recursos.
[4] Una conexión entre programas bajo el estándar POSIX.

CAPÍTULO 4
Implementación del sistema

En este capítulo se explica como se resolvieron los retos que se encontraron al implementar el proyecto en base a los requisitos, y se muestran las funciones de los distintos elementos del gestor.

4.1 Requisitos

Para la validez de este proyecto, se deben cumplir ciertas condiciones, que son las siguientes:

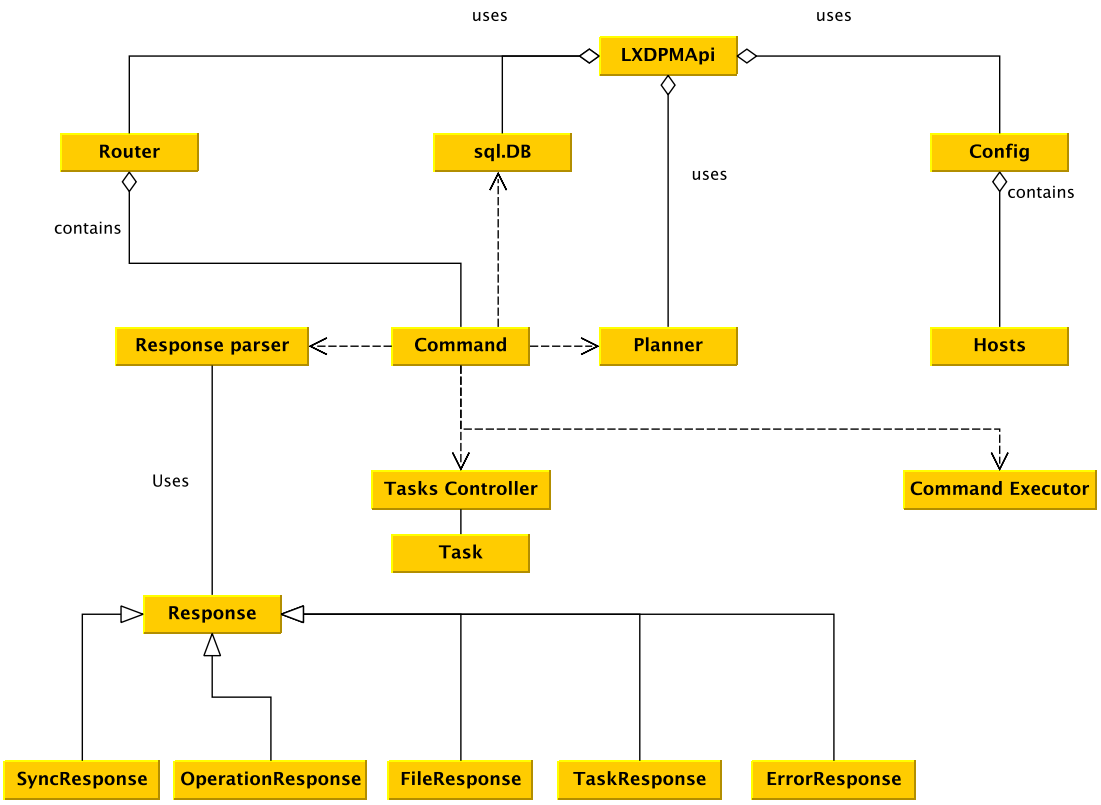
- La *API* debe ser equivalente a la de *LXD* para mantener compatibilidad con otros posibles software de terceros.
- La *API* debe ser compatible con un proyecto externo front-end.
- La *API* debe gestionar varios nodos *LXD*.
- Debe ser capaz de gestionar contenedores y sus subrecursos, y los recursos de imágenes de contenedores y perfiles de ejecución.
- El planificador debe gestionar el lanzamiento de los contenedores.

Estos requisitos afectarán a las decisiones tomadas en la resolución de las tareas derivadas del objetivo final.

4.2 Diagrama UML

Este es el diagrama UML del software gestor de plataforma implementado. Se puede observar que el servicio *LXDPMApi* está formado por un *Router*, la base de datos, el *Planner* y la configuración. El *Router* a su vez está formado por *Commands* , que implementan las diferentes rutas de los recursos del servidor. Estos *Comandos* se valen de la base de datos, el *Command Executor* , el *Response Parser* y el *Planner*, para satisfacer las peticiones, procesarlas, dirigirlas a los hosts indicados y devolver una respuesta al usuario.

Figura 4.1: El diagrama UML del proyecto



4.3 Agregación de recursos

Para poder controlar *hosts*, primero hay que crearlos y ver como agregarlos a nuestra solución. Para ello se creó un entorno de pruebas formado por tres nodos mononúcleo con sistemas *Ubuntu* de iguales características, donde se instalarían los distintos *daemons LXD* a agregar.

Después de estudiar la tecnología *LXD* se observó que se podía implementar la conexión entre nodos con certificados *SSL* mediante conexiones *HTTPS*, o bien mediante *sockets UNIX* , sin necesidad de certificados. La conexión directa entre nodos *LXD* requiere un proceso de emparejamiento. Como el objetivo de este proyecto es controlar un conjunto de nodos, y la parte de emparejamiento con certificados podía tener un coste alto de implementación, se optó por una opción más sencilla a corto plazo, que sirve igualmente a nuestro propósito. La manera de agregar host consiste en copiar la clave pública de el host principal en los hosts con acceso al host a controlar, y crear un usuario que tenga permisos para ejecutar comandos en Ubuntu.

De este modo, el *host* principal tiene acceso a los host controlados, y se pueden enviar los datos de la aplicación mediante *sockets UNIX*, siendo capaces de operar sobre el *daemon* del host y por tanto, de controlarlo a través de nuestra aplicación, evitando el coste de implementar el sistema de emparejamiento a bajo nivel.

4.4 Procesado de los comandos

Para poder cubrir los requisitos de ser equivalentes con la *API* de *LXD*, el diseño del gestor de plataforma se ha basado en el propio diseño de *LXD*, y se utilizan sus estructuras de datos importadas en nuestro proyecto, para asegurar la máxima correspondencia entre su manera de tratar los datos y la nuestra. Como se puede observar en la figura 4.1, la estructura *LXDPMApi* es un servidor que se vale de un *Router* , una conexión a una base de datos *SQlite*, un planificador y una configuración con *hosts* a controlar, para poder recibir las peticiones a gestionar. El *Router* es el elemento que suscribe todos los puntos de acceso a los recursos mediante rutas *URL*.

Estos puntos de acceso están representados por *Command*, que son estructuras que implementan los diferentes métodos *HTTP*[1] para procesar y ejecutar las operaciones recibidas. Así pues, el servidor recibe la petición en una de las rutas *URL*, suscritas al router. Si la ruta y los métodos están implementados, la *API* sigue con el procesado de la petición, y en caso contrario, informa de que ese punto de acceso no está implementado. En el caso de estar implementado, el *Command* continúa, desempaquetando el mensaje *JSON* que recibe. Este mensaje es entonces enviado al método adecuado que implementa la petición *HTTP* correspondiente, para lanzar la orden en el servidor *host* que haya sido solicitado transparentemente.

A partir de aquí, el servidor devuelve una respuesta, que puede ser de 5 tipos:

Síncronas Este tipo de respuestas se utilizan para respuestas inmediatas.

Asíncronas Este tipo de respuestas se utilizan cuando hay que hacer tareas en segundo plano.

Operación Este tipo de respuesta es una asíncrona con información sobre la tarea que esta siendo llevada a cabo en segundo plano.

Archivo Este tipo de respuesta es la que envia el servidor al devolver un archivo.

Error Este tipo de respuesta es la que se envia al producirse un error en el host.

Se analiza el tipo de respuesta, y nuestra *API* devuelve la respuesta adecuada, además de añadir, borrar o modificar, si corresponde, la referencia al recurso en la base de datos.

4.5 Lanzamiento de tareas en segundo plano

En algunas operaciones, el tiempo de espera de la respuesta puede no ser aceptable para una herramienta gestionada por un usuario que espera resultados inmediatos. Para devolver una respuesta adecuada, se diseñó una estructura de comportamiento casi idéntico a la de *operations* en el proyecto *LXD*. En su caso, al lanzar una orden sobre su *API*, devuelven una respuesta operación con un

[1] GET, PUT, POST, DELETE y PATCH

identificador que es añadida a un conjunto de operaciones que se irán ejecutando asíncronamente, y a medida que vayan acabando, dejarán los resultados en la ruta de operaciones. El identificador es necesario para consultar el resultado real de la operación requerida. Es posible también esperar a que el resultado de la operación esté disponible, añadiendo */wait* al final de la *URL* del recurso.

En nuestro caso, y para evitar confusiones con el resto de recursos, y por si finalmente se implementa el recurso operaciones, la estructura se ha llamado *Task*. Esta estructura es capaz de lo mismo que son capaces las operaciones en el proyecto original. En este caso al hacer una petición a un recurso que requiera ser gestionado por una *Task* o tarea, se añadirá a una cola de tareas y devolverá una *TaskResponse*, muy similar a una operation response. Esta cola, gestionada por un *mutex*[2], que limita o permite el acceso a las *goroutines* [3] dependiendo del estado de bloqueo, es la que contiene todas las *Task* que se han lanzado. Los diferentes hilos irán completando su ejecución, hasta completarla o fallar.

4.5.1. Operaciones lanzadas en segundo plano

Existen dos operaciones que se manejan con tareas, y las dos ocurren en el *endpoint* de *Images*, el encargado de gestionar las imágenes a utilizar por el contenedor a desplegar. En concreto estas dos operaciones corresponden a los métodos:

POST en /1.0/images En este caso, en la *API* de *LXD* se da la orden para que una imagen de contenedor sea descargada por el *daemon*, haciéndola disponible en los recursos de imágenes. En nuestro gestor de plataforma, debe ser por tanto capaz de conectar con los diferentes servidores, y mandarles la orden de añadir la imagen seleccionada a sus recursos. Esto podía dejar en estado inconsistente nuestro sistema en el caso de que una de las imágenes no llegase a ser añadida a su *host* correspondiente. Por esa razón, aprovechando que los host *LXD* pueden esperar al resultado de las *Operations*, esta orden se lanza en background, envía la operación a todos los host afectados, y espera los resultados. Si en alguno de ellos falla, en los que ha sido desplegado correctamente, la imagen se borra. Si todo ha sido positivo, entonces la tarea acaba satisfactoriamente.

DELETE en /1.0/images/<fingerprint> La *API* original recibe una orden de borrado, y devuelve una operación en segundo plano. Nuestra *API* debe devolver una tarea en segundo plano, y debe borrar la imagen de todos los hosts. En el caso de que no se pueda borrar, devuelve el error y el *host* en el que ha ocurrido.

De esta forma, se consigue que la *API* no se bloquee y pueda seguir atendiendo peticiones, mientras va ejecutando tareas definidas en segundo plano.

[2]Un mutex es un objeto software que implementa exclusión mutua, esto es, bloquea el acceso a un recurso compartido por varios hilos o *threads*.

[3]Hilos o *threads* en Golang.

4.6 Diseño e implementación del planificador

En el contexto de despliegue de recursos, un planificador o *scheduler* es la pieza de software que decide, en base a unas funciones de puntuación del host, donde se van a desplegar los recursos solicitados. En nuestro caso, estos recursos van a ser los contenedores.

Cuando hubo que implementar la parte de gestión de contenedores, se encontraron varios problemas. Cómo se puede indicar en qué host se va desplegar el contenedor? Cómo se puede saber en que host estaba alojado un container? Cambios en la estructura de los mensajes *JSON* devueltos por nuestra *API* podrían hacer que los mensajes dejaran de ser equivalentes a los de *LXD*.

Se propuso entonces, que el encargado de decidir dónde desplegar un contenedor fuese una pieza de software, liberando de la responsabilidad de la toma de decisión del despliegue de recursos al usuario de nuestra *API* y evitando romper la equivalencia del servicio. El comportamiento por defecto del planificador es tomar de forma aleatoria un host y asignarlo cuando se cree el contenedor. No obstante, también se quería dar opción a la optimización del despliegue de los recursos, y se añadieron las estrategias *Gather* y *Scatter*[4].

4.6.1. Función y estrategias de selección

Se han diseñado tres estrategias de selección de host, para que, sin proveer el nombre del *host* a utilizar, la API fuese capaz de elegir el destino del contenedor que va a ser desplegado. Para poder tomar esa decisión, se necesita información sobre los hosts, y para ello, la función de selección del host se basa en los procesadores y la memoria libre del host. De igual modo que ocurre en *Command* , se implementaron métodos para consultar el estado de la memoria del host, y de su número de procesadores.

La función de selección calcula una puntuación de host como:

$$f(x,y,z) = (x/y + (z/y) * (x/y)) \tag{4.1}$$

$$z >= 1 \tag{4.2}$$

$$y >= 1 \tag{4.3}$$

donde $x = memorialibre$, $y =$ contenedores desplegados y $z =$ número de procesadores en el sistema.

Se trata de una función heurística que relaciona los recursos de memoria y capacidad de cómputo. A mayor cantidad de recursos por contenedor, mayor disponibilidad de recursos en el servidor. De esta manera, al maximizar esta función, conseguimos implementar la estrategia de *scatter* o distribuir, ya que nos dará el host con mayor disponibilidad de recursos, repartiendo la carga, y al minimizar, la estrategia de *gather* o concentrar, que nos dará el *host* con menos recursos, para poder concentrar el despliegue de contenedores en un único host.

Estas son las tres estrategias de selección de *host*:

[4]Concentrar y distribuir

Aleatoria Disponible desde el principio, esta estrategia selecciona un *host* de forma aleatoria en el que desplegar el contenedor.

Scatter o Distribuir Selecciona el servidor con mayor cantidad de recursos por contenedor.

Gather o Concentrar Selecciona el servidor con menos recursos libres por contenedor, para poder concentrar el lanzamiento de contenedores en ese *host*.

4.6.2. Interfaz del planificador

De igual modo que ocurre con el resto de recursos en la *API*, el acceso al planificador se ofrece a través de una *URL*, y se puede consultar y manipular con los distintos métodos *HTTP*.

GET /1.0/planner Este *endpoint* se utiliza para consultar la estrategia actual en el planificador. Devuelve una respuesta síncrona con la estrategia del planificador en el campo *metadata*.

POST /1.0/planner Inicialmente se utilizaba para inicializar el estado del planificador. Finalmente por diseño se decidió inicializar internamente la estrategia en modo aleatorio al iniciar la aplicación. Aceptaba un mensaje *JSON* con el campo *State* y los valores, *random*, *scatter* y *gather*. Devuelve una respuesta síncrona vacía, o un error.

PUT /1.0/planner Se utiliza para actualizar el estado de la estrategia de decisión. Acepta un mensaje *JSON* con el campo *State* y los valores, *random*, *scatter* y *gather*. Devuelve una respuesta síncrona vacía, o un error.

4.7 Operaciones soportadas

Todas la rutas en común entre el gestor de plataforma y *LXD* son capaces de recibir exactamente el mismo tipo de mensajes que procesa el segundo, ya que se decidió incorporar una dependencia con las estructuras que se usan en el proyecto. Así se asegura la máxima compatibilidad entre ambas herramientas. Para ver la especificación de los mensajes, ver [4] o consultar el glosario de ejemplos de comandos.

Estos son los puntos de acceso a los recursos, soportados por el gestor de plataforma:

/1.0/containers

GET Es la operación que devuelve el conjunto de contenedores disponibles en todos los host.

POST Es la operación que permite alojar un nuevo contenedor en uno de los *host*. La elección del *host* depende de la estrategia seguida por el planificador.

/1.0/containers/<container_name>

GET Esta operación obtiene información sobre el contenedor que se consulte.

POST Esta operación se utiliza para cambiar el nombre del contenedor.

PUT Esta es la operación que modifica la información del contenedor.

DELETE Esta es la operación que borra el contenedor seleccionado.

/1.0/containers/<container_name>/exec

POST Ejecuta un commando de *shell* en el contenedor seleccionado.

/1.0/containers/<container_name>/files

GET Devuelve el archivo consultado en el contenedor.

POST Envia un archivo al contenedor.

DELETE Elimina un archivo del contenedor.

/1.0/containers/<container_name>/snapshots

GET Devuelve la lista de *snapshots* en el contenedor.

POST Crea un nuevo snapshot.

/1.0/containers/<container_name>/snapshots/<snapshot_name>

GET Devuelve la información del *snapshot* seleccionado.

POST Cambia el nombre del *snapshot*.

DELETE Borra el snapshot.

/1.0/containers/<container_name>/state

GET Devuelve el estado actual del contenedor.

PUT Modifica el estado del contenedor.

/1.0/images

GET Muestra las imágenes disponibles.

POST Envia la imagen a todos los hosts, y en el caso de fallo en algunos de los *hosts*, se devuelve a los *hosts* a su estado anterior.

/1.0/images/<fingerprint>

GET Muestra la información de las imágenes

DELETE Borra la imagen seleccionada. Para guardar la coherencia, se elimina la imagen de todos los *host*.

[/1.0/images/<fingerprint>/export

GET Descarga la imagen en el servidor principal.

/1.0/profiles

GET Devuelve los perfiles de ejecución disponibles.

POST Crea un nuevo perfil de ejecución.

/1.0/profiles/<profile_name>

GET Devuelve la configuración del perfil de ejecución.

POST Cambia el nombre del perfil.

DELETE Borra el perfil seleccionado.

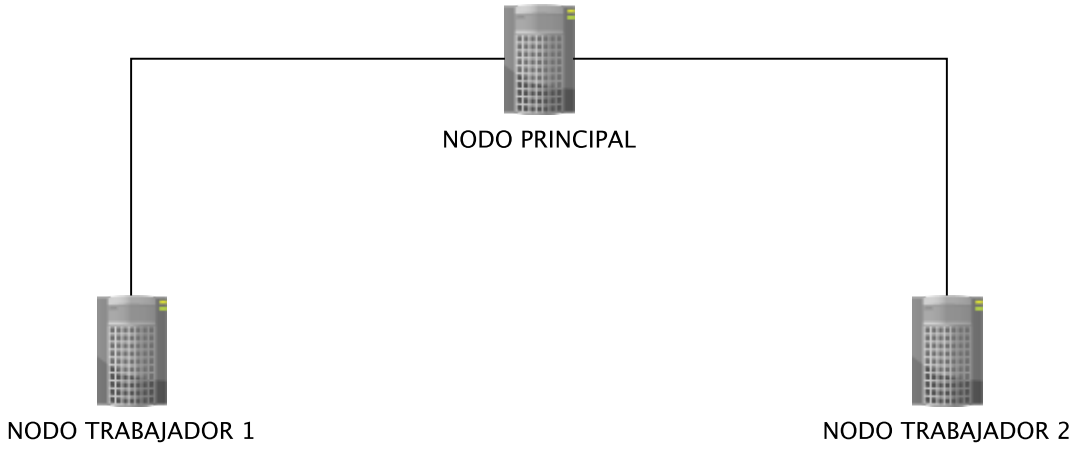
CAPÍTULO 5
Validaciones y pruebas

En este capítulo se mostrarán las pruebas que se hicieron para validar el software y se comparará con Docker-Swarm.

5.1 Comparación con Docker-Swarm

Hay que dejar claro que el enfoque de nuestro gestor de plataforma difiere de *Docker-Swarm* en cuanto a su uso final. Nuestro gestor está orientado a facilitar la gestión de infraestructura de contenedores de sistemas completos mientras que *Docker-Swarm* facilita la gestión de la orquestación de servicios contenidos. Sin embargo, su funcionamiento interno es muy similar, pues ambos deben controlar recursos sobre un conjunto de *hosts*. Para las pruebas se utilizaron tres host virtuales con la misma capacidad de cómputo y memoria.

Figura 5.1: Configuración del entorno



En esta configuración el *NODO PRINCIPAL* actúa de nodo maestro para controlar un conjunto de nodos *worker*. En *LXD/LXC* con nuestro gestor, el nodo principal contiene la *API* del gestor, junto con un daemon *LXD*, y los nodos *worker* contienen sólo el daemon *LXD*, que son los que manejan los contenedores *LXC* en cada host. En el caso de *Docker-Swarm*, todos los nodos contienen el mismo

software, que es *Docker Engine*, la tecnología de contenedores, que contiene las funciones para gestionar un cluster de nodos. Hay que destacar en lugar de tener una *API* como en nuestro caso, *Docker* y *Docker-Swarm* se gestionan mediante clientes de línea de comandos.

5.1.1. Agregación de host

En nuestro caso, en el apartado de implementación se explicaba como era el proceso de añadir *host* para ser controlados. En el de *Docker-Swarm* ocurre de la siguiente manera.

1. Se configuran los *host* con la versión correspondiente de *Docker Engine* , con una configuración de los nodos que les permita comunicarse mediante una red.

2. Se lanza el nodo principal mediante el cliente de *Docker* en modo *swarm*[1]. Para añadir un nodo al enjambre, hay que ejecutar el comando que resulta de lanzar el contenedor. Se trata de un modo de emparejamiento basado en tokens, por el cual un *host* queda añadido al cluster de nodos.

3. Se añaden tantos host como se quieran controlar desde el nodo principal.

Cómo en nuestro caso es una primera versión, la *API* gestora de nodos *LXD* no es totalmente compatible con el protocolo que utiliza *LXD* para emparejar nodos. Esto provoca que sea más complicado añadir host actualmente, aunque se mejorará en versiones posteriores. No obstante, a pesar de esto, los *endpoints* de recursos ofrecidos a través de la *API* son funcionales, como se ha demostrado en las pruebas realizadas, y es capaz de gestionar los *host* suscritos.

5.1.2. Despliegue de recursos

Una vez se ha configurado la infraestructura necesaria, ambas soluciones están listas para recibir peticiones y procesarlas. Recordamos que *Docker* despliega servicios y *LXD* despliega sistemas operativos completos. En el caso de *Docker-Swarm* ocurre lo siguiente:

1. Una vez listos los nodos, se pueden lanzar servicios especificando el nombre del contenedor, el número de replicas y la imagen virtual del servicio que se quiera lanzar. En este caso, *Docker* obtiene las imágenes de su servicio *DockerHub*, un repositorio de imágenes de servicios propiedad de *Docker*.

2. Lanzado el contenedor, se puede consultar el estado, la metainformación, y realizar operaciones sobre de los contenedores disponibles en el clúster de *Docker-Swarm*.

En el caso de nuestro gestor:

[1] Enjambre.

1. Se lanza el servidor, y a partir de ese momento se puede interactuar con él. Se pueden lanzar todas las operaciones definidas en el apartado 4.7.

2. A partir de disponer de recursos en el gestor, se puede consultar su estado con el método correspondiente, o realizar la operación que corresponda.

Figura 5.2: De este modo se inicia el funcionamiento del gestor de plataforma LXD

5.1.3. Gestión de recursos

Ambos proyectos son muy similares en cuanto a funcionalidad sobre sus respectivos recursos, pues ambos pueden:

- Consultar los recursos.

Figura 5.3: Ejemplo de obtención de contenedores en todos los *host*.

Figura 5.4: Ejemplo de obtención de nodos *host* en *Docker*.

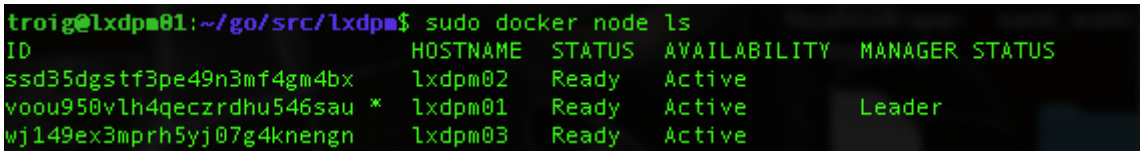
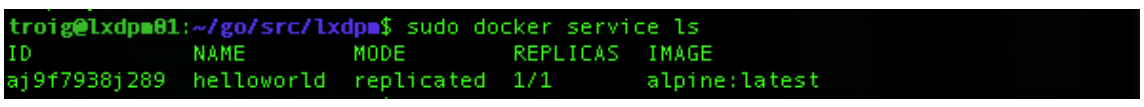


Figura 5.5: Ejemplo de obtención de servicios en Docker.



- Crear recursos.

Figura 5.6: Ejemplo de creación de contenedores en algún *host* determinado por el planificador.

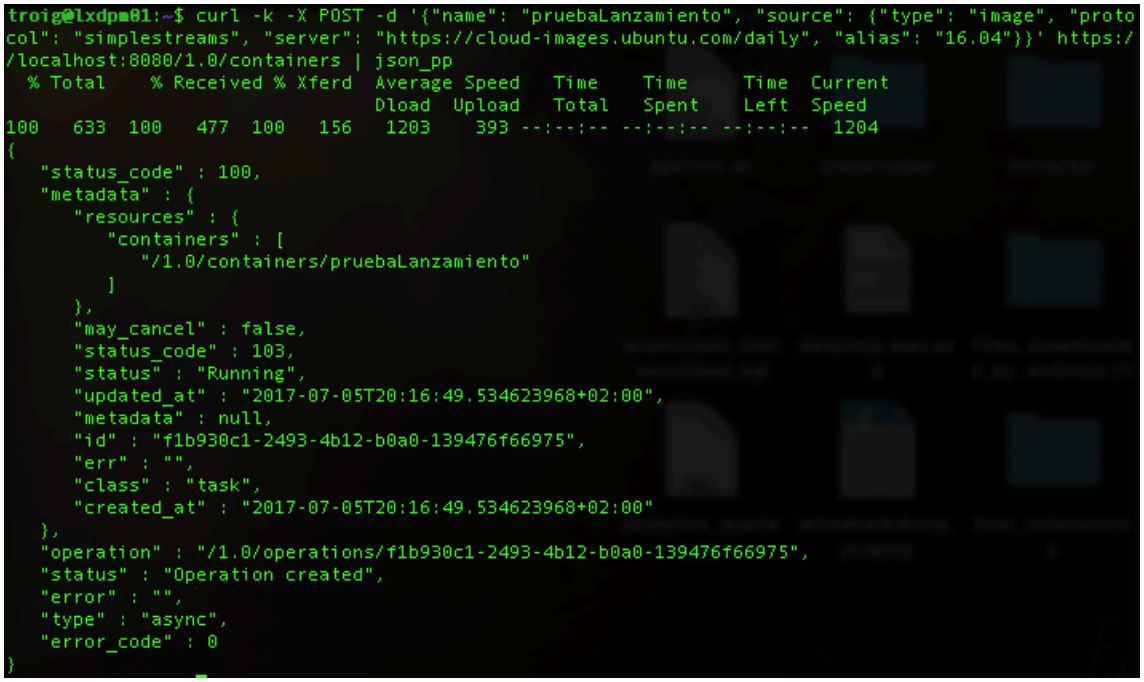
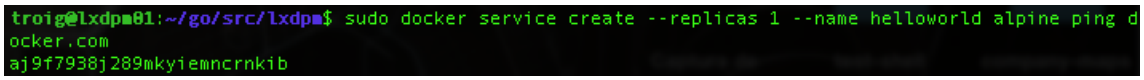


Figura 5.7: Ejemplo de creación de servicios en algún *host* del clúster de *Docker-Swarm*.



- Modificar los recursos.

Figura 5.8: Ejemplo de modificación del nombre de un contenedor.

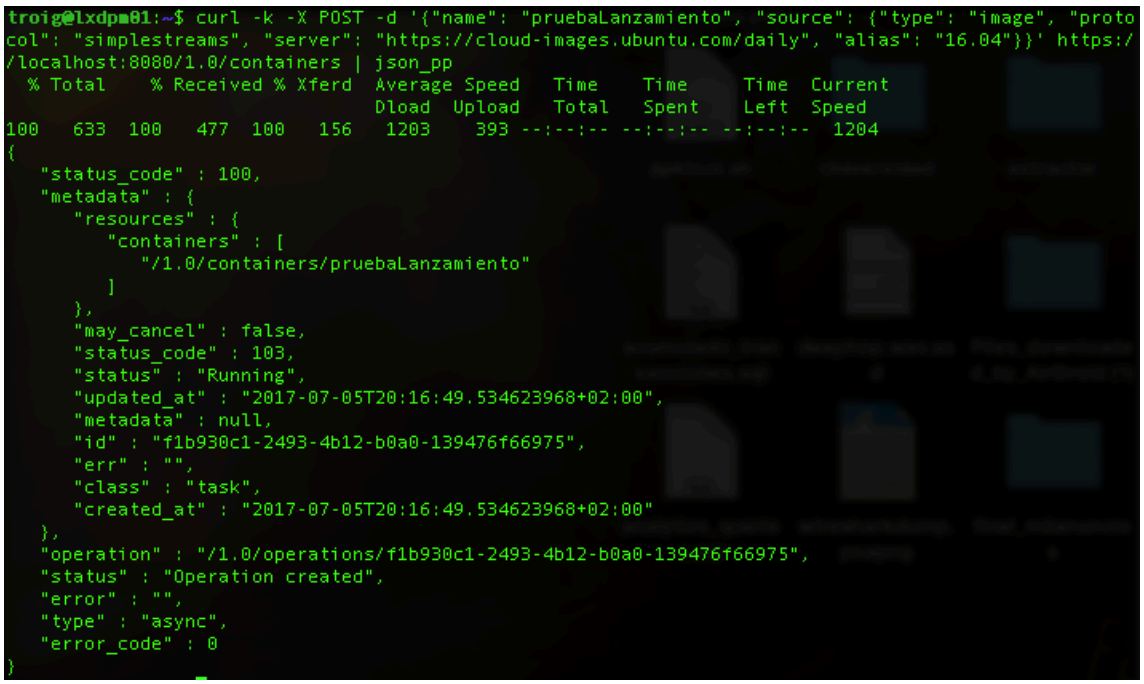


Figura 5.9: Ejemplo de modificación de los argumentos de un contenedor.

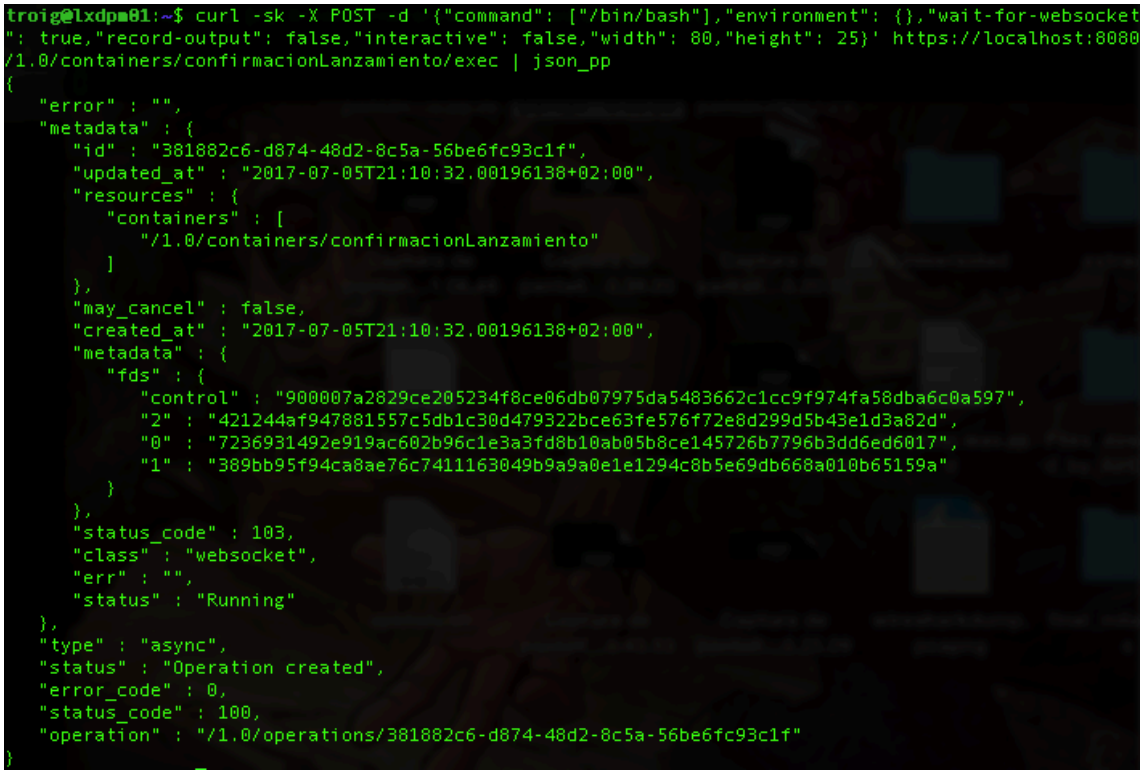
```
troig@lxdpm01:~/go/src/lxdpm$ sudo docker service update --args "echo 'Hello'" helloworld
helloworld
```

Figura 5.10: Resultado de la modificación de los argumentos de un contenedor. Véase el campo *args*.


```
troig@lxdpm01:~/go/src/lxdpm$ sudo docker service inspect helloworld
[
    {
        "ID": "aj9f7938j289mkyiemncrnkib",
        "Version": {
            "Index": 61
        },
        "CreatedAt": "2017-07-05T22:10:45.209649851Z",
        "UpdatedAt": "2017-07-05T23:04:09.026374545Z",
        "Spec": {
            "Name": "helloworld",
            "TaskTemplate": {
                "ContainerSpec": {
                    "Image": "alpine:latest@sha256:1072e499f3f655a032e88542330cf75b02e7bdf673278f701d
7ba61629ee3ebe",
                    "Args": [
                        "echo",
                        "Hello"
                    ],
                    "DNSConfig": {}
                },
                "Resources": {
                    "Limits": {},
                    "Reservations": {}
                },
                "RestartPolicy": {
                    "Condition": "any",
                    "MaxAttempts": 0
                },
                "Placement": {},
                "ForceUpdate": 0
            },
            "Mode": {
                "Replicated": {
                    "Replicas": 1
                }
            },
            "UpdateConfig": {
                "Parallelism": 1,
                "FailureAction": "pause",
                "MaxFailureRatio": 0
            },
            "EndpointSpec": {
                "Mode": "vip"
            }
        },
        "PreviousSpec": {
            "Name": "helloworld",
            "TaskTemplate": {
                "ContainerSpec": {
                    "Image": "alpine:latest@sha256:1072e499f3f655a032e88542330cf75b02e7bdf673278f701d
7ba61629ee3ebe",
                    "Args": [
                        "ping",
                        "docker.com"
                    ],
                    "DNSConfig": {}
                },
```


- Ejecutar comandos en los recursos a traves de la línea de comandos.

Figura 5.11: Ejemplo de ejecución de un comando en el contenedor *LXD*.



- Hacer copias del estado actual del recurso.

Figura 5.12: Ejemplo de creación de una imagen a partir del contenedor.

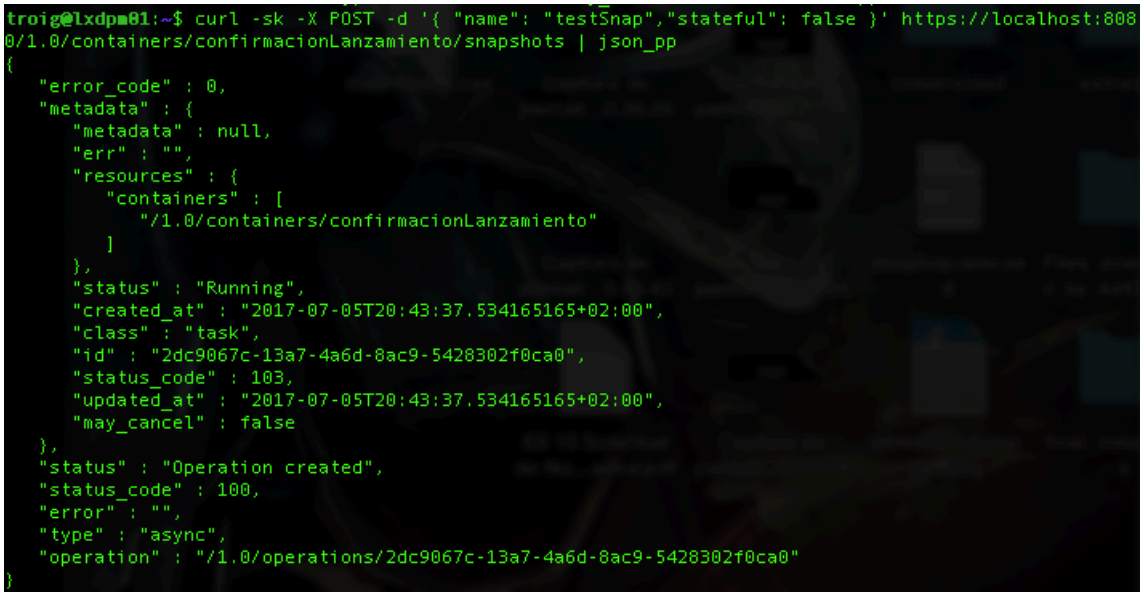


Figura 5.13: Ejemplo de consulta de los *snapshots* que se han tomado del contenedor.

```
troig@lxdpm01:~$ curl -sk -X GET https://localhost:8080/1.0/containers/confirmacionLanzamiento/snapsh
ots | json_pp
{
   "error" : "",
   "status" : "Success",
   "operation" : "",
   "status_code" : 200,
   "metadata" : [
      "/1.0/containers/confirmacionLanzamiento/snapshots/testSnap"
   ],
   "type" : "sync",
   "error_code" : 0
}
```

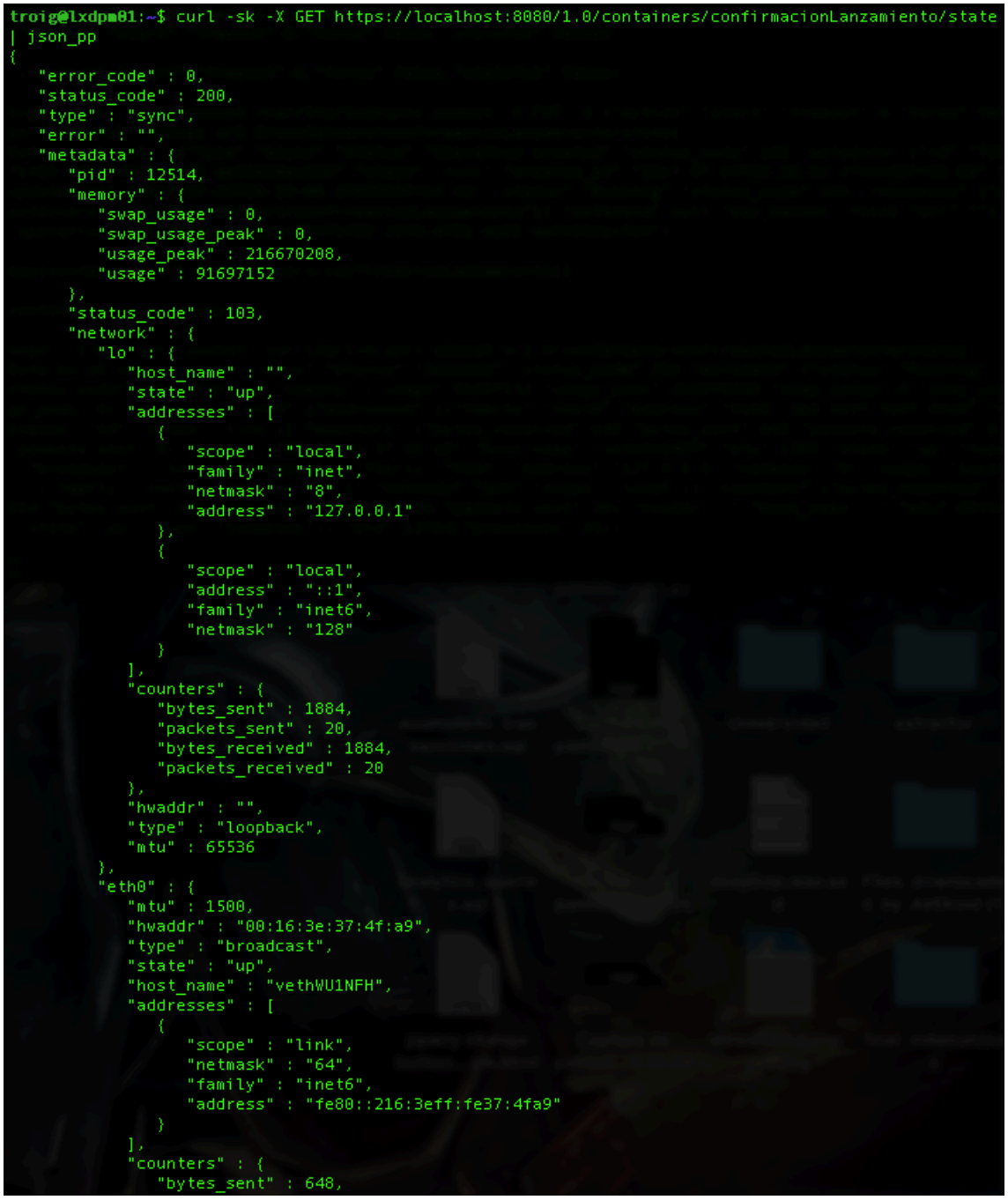
Figura 5.14: Ejemplo de consulta de un *snapshot* del contenedor.

```
troig@lxdpm01:~$ curl -sk -X GET https://localhost:8080/1.0/containers/confirmacionLanzamiento/snapsh
ots/testSnap | json_pp
{
   "status" : "Success",
   "error" : "",
   "operation" : "",
   "metadata" : {
      "created_at" : "2017-07-05T20:43:16+02:00",
      "name" : "confirmacionLanzamiento/testSnap",
      "ephemeral" : false,
      "config" : {
         "volatile.last_state.idmap" : "[{\"Isuid\":true,\"Isgid\":false,\"Hostid\":100000,\"Nsid\":0
,\"Maprange\":65536},{\"Isuid\":false,\"Isgid\":true,\"Hostid\":100000,\"Nsid\":0,\"Maprange\":65536}
]",
         "volatile.eth0.hwaddr" : "00:16:3e:37:4f:a9",
         "volatile.idmap.next" : "[{\"Isuid\":true,\"Isgid\":false,\"Hostid\":100000,\"Nsid\":0,\"Map
range\":65536},{\"Isuid\":false,\"Isgid\":true,\"Hostid\":100000,\"Nsid\":0,\"Maprange\":65536}]",
         "volatile.base_image" : "8690a59a6d2c561da504b96182d2661fd4a79a9b3ba594cba102e66fa87c4998",
         "volatile.last_state.power" : "RUNNING",
         "volatile.idmap.base" : "0"
      },
      "devices" : {
         "root" : {
            "type" : "disk",
            "path" : "/"
         }
      },
      "profiles" : [
         "default"
      ],
      "architecture" : "x86_64",
      "expanded_config" : {
         "volatile.base_image" : "8690a59a6d2c561da504b96182d2661fd4a79a9b3ba594cba102e66fa87c4998",
         "environment.http_proxy" : "http://[fe80::1%eth0]:13128",
         "volatile.last_state.power" : "RUNNING",
         "user.network_mode" : "link-local",
         "volatile.eth0.hwaddr" : "00:16:3e:37:4f:a9",
         "volatile.last_state.idmap" : "[{\"Isuid\":true,\"Isgid\":false,\"Hostid\":100000,\"Nsid\":0
,\"Maprange\":65536},{\"Isuid\":false,\"Isgid\":true,\"Hostid\":100000,\"Nsid\":0,\"Maprange\":65536}
]",
         "volatile.idmap.next" : "[{\"Isuid\":true,\"Isgid\":false,\"Hostid\":100000,\"Nsid\":0,\"Map
range\":65536},{\"Isuid\":false,\"Isgid\":true,\"Hostid\":100000,\"Nsid\":0,\"Maprange\":65536}]",
         "volatile.idmap.base" : "0"
      },
      "expanded_devices" : {
         "eth0" : {
            "parent" : "lxdbr0",
            "type" : "nic",
            "nictype" : "bridged",
            "name" : "eth0"
         },
         "root" : {
            "path" : "/",
            "type" : "disk"
         }
      },
      "stateful" : false
   },
   "error_code" : 0,
   "type" : "sync",
```

- Crear contenedores a partir de imágenes. Ver 5.6 y 5.7.

- Ver cuotas de uso de los recursos.

Figura 5.15: Ejemplo de consulta del estado del contenedor.



- Tienen un repositorio de imágenes descargadas para su uso.

Figura 5.16: Ejemplo de consulta de las imágenes disponibles en *LXD*

```
troig@lxdpm01:~$ curl -sk -X GET https://localhost:8080/1.0/images | json_pp
{
   "status" : "Success",
   "operation" : "",
   "error" : "",
   "metadata" : [
      "/1.0/images/74186c79ca2f22c1a566ebaf99f67f52313814429d110b16677b7d74b9e6e26b",
      "/1.0/images/8690a59a6d2c561da504b96182d2661fd4a79a9b3ba594cba102e66fa87c4998",
      "/1.0/images/74186c79ca2f22c1a566ebaf99f67f52313814429d110b16677b7d74b9e6e26b",
      "/1.0/images/74186c79ca2f22c1a566ebaf99f67f52313814429d110b16677b7d74b9e6e26b",
      "/1.0/images/8690a59a6d2c561da504b96182d2661fd4a79a9b3ba594cba102e66fa87c4998"
   ],
   "status_code" : 200,
   "type" : "sync",
   "error_code" : 0
}
```

Figura 5.17: Ejemplo de consulta de las imágenes disponibles en *Docker-Swarm*

```
troig@lxdpm01:~/go/src/lxdpm$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              1815c82652c0        3 weeks ago         1.84 kB
```

5.1.4. Conclusiones

Como se ha podido observar, el uso de ambas herramientas es bastante similar, aunque sirven a propósitos distintos.

La manera de agregar *host* de *Docker-Swarm* es más simple de gestionar que en el caso del gestor de plataforma implementado, si bien es verdad que *Docker-Swarm* funciona con una red *overlay*, lo que permite más flexibilidad a la hora de manejar los nodos en la red. También parece ser que el modo *Swarm* elimina alguna de las funciones más directas de *Docker*, como iniciar sesiones interactivas de línea de comando, creación de snapshots de servicios, aunque se pueden realizar buscando el host que almacena el contenedor. También vemos una diferencia en cuanto a funcionamiento, pues con *Docker* se interactúa mediante un cliente mediante la línea de comandos mientras que el gestor funciona como una *API*. Este comportamiento podría cambiar si se implementara el emparejamiento con certificados. De este modo podríamos tratar la *API* como si fuera un nodo *LXD* más, mediante su propio cliente de línea de comandos.

Finalmente, se ha podido comprobar que aunque tengan usos diferentes, ambas tecnologías facilitan a su modo el despliegue de recursos de infraestructura.

5.2 Integración con lxd-webgui

Al principio del proyecto se propuso hacerlo compatible con una aplicación web[2]que serviria de interfaz al usuario gestor de *LXD*. Aunque el estado de la

[2]Ver en [7]

API es consistente con la *API* original de LXD, la configuración de esta interfaz obliga a nuestro gestor a implementar el mecanismo de emparejamiento, para que sea tratado igual que un nodo *LXD*. En el estado actual de el proyecto, no ha sido posible por tanto, hacer esta integración.

CAPÍTULO 6
Conclusiones

En este capítulo analizaremos si se ha cumplido con los objetivos y se propondrán mejoras a realizar.

Inicialmente se pedía implementar un software con la capacidad de gestionar varios contenedores *LXC*. Hemos podido ver, en las pruebas, que el software implementado es capaz de gestionar contenedores, imágenes y perfiles de ejecución. *LXD* gestiona más recursos además de estos, aunque fueron los más prioritarios para dotarlo de funcionalidad.

También se pedía que los mensajes que se enviaran fueran exactamente los mismos que aceptaba *LXD* mediante el envío de mensajes *JSON*. Hemos podido comprobar, que efectivamente, el gestor de plataforma acepta las mismas estructuras de mensajes. Para poder lograr esto, se implementó un planificador, y diferentes estrategias de despliegue, para gestionar en qué *host* desplegar los recursos sin necesidad de tener que incluirlos en los mismos mensajes *JSON* de la *API* original. También se ofrece la posibilidad de consultar el *host* dónde ha sido desplegado el recurso, de nuevo, sin aceptar a la estructura *JSON*, pues se envia en una cabecera propia en la respuesta a la petición *HTTP*.

Se pedía realizar una integración con una interfaz gráfica via aplicación web para controlar los contenedores a través de nuestra *API*, pero no ha sido posible integrarla, por el hecho de que necesita un punto de acceso con las mismas características de emparejamiento que el daemon *LXD*.

Podemos decir por tanto, que nos hemos acercado bastante a los requisitos, y los que no han podido ser llevados a cabo, se pueden mejorar en las siguientes versiones de la aplicación.

6.1 Trabajos futuros

Siendo una primera versión, aún existen muchas mejoras por hacer a partir de lo implementado y todo lo aprendido. Se proponen las siguientes mejoras:

Convertir el gestor en un nodo LXD Si se implementa esta funcionalidad, se podría integrar con otras herramientas que soporten *LXD*, y mejoraría la gestión de los nodos. Algunas de las herramientas que serían útiles son el clien-

te de línea de comandos de *LXD* y el frontend web, para manejarlo de manera más gráfica.

Utilización de un Dialer tcp En lugar de recurrir a la herramienta *Curl*, las conexiones con los otros nodos se podrían llevar a cabo con un *Dialer* sobre *tcp*. Esto permitiría reducir las dependencias del proyecto con esta herramienta, y probablemente sería más eficiente, al eliminar capas de la arquitectura del sistema.

Red overlay Se podría desarrollar un método para poner en contacto los diferentes nodos a través de una red *overlay*. Está red nos permitiría unir las diferentes redes que crea *LXD* en cada host, y por tanto aumentar el número de operaciones que se pueden realizar sobre los nodos, como en el caso de migración de recursos.

Crear un repositorio de imágenes Durante el desarrollo de las operaciones sobre imágenes, se observó que se podía suministrar la imagen a partir de una especie de repositorio. Si implementásemos un servidor web con el mismo protocolo que las imágenes de muestra, seríamos capaces de enviar las imágenes desde nuestro repositorio, y por tanto aumentar el control sobre el sistema.

6.2 Agradecimientos

Quiero dar las gracias a los tutores de este proyecto por su paciencia y atención a la hora de consultar dudas, y a la integración con el resto de personas del grupo de investigación. A mi familia y mi pareja, por su paciencia y apoyo día a día. Y a todos los profesores, compañeros de carrera y de trabajo, pues gracias a ellos he aprendido mucho y se lo debo.

Bibliografía

[1] Página web principal de la Open Container Initiative. Consultado en https://www.opencontainers.org/.

[2] Documentación acerca de como utilizar LXD. Consultado en https://www.stgraber.org/.

[3] Repositorio de Docker-Swarm. Consultado en https://github.com/docker/swarm

[4] Página web de la especificación de la API de LXD en su repositorio Github. Consultado en https://github.com/lxc/lxd/blob/master/doc/rest-api.md

[5] Página web del repositorio de LXD en Github. Consultado en https://github.com/lxc/lxd

[6] Página web en Wikipedia sobre metodología Lean. Consultado en https://es.wikipedia.org/wiki/Lean_software_development

[7] Repositorio en *Github* del proyecto GUI de *LXD*. Consultado en https://github.com/dobin/lxd-webgui

[8] Información sobre Docker. Consultado en https://www.docker.com/what-docker

[9] Información sobre pods de Kubernetes. Consultado en https://kubernetes.io/docs/concepts/workloads/pods/pod/

[10] Documentación sobre el lenguaje *Go*. Consultado en https://golang.org/pkg/

.1 Glosario de términos

API *Aplication Programming Interface* o Interfaz de programación de aplicaciones, es una capa de abstracción que aglomera métodos y funciones, para ser utilizados por software de terceros.

Clúster Agrupación de elementos, en este caso, agrupación de recursos o contenedores.

Contenedor En este caso, un contenedor hace referencia a recursos contenidos, es decir, servicios o sistemas, que contienen todo lo necesario para funcionar por si mismos.

Daemon Es un servicio en segundo plano que funciona autónomamente, en lugar de ser operado directamente por el usuario.

Dialer En *Golang*, un dialer es un software que ofrece opciones para crear una conexión, que puede ser de diferentes tipos.

Gather Hace referencia a una estrategia de despliegue de recursos que consiste en agrupar y concentrar el cómputo.

Host Cada uno de los computadores en red que contienen los recursos software, sobre los que en este caso se instalan *LXD* y *Docker*.

Imagen Es una versión portátil del contenedor, o que define como desplegar un contenedor. En el caso de *Docker*, las imágenes son servicios, mientras que en el caso de *LXD* son sistemas completos.

JSON *JavaScript Object Notation*es un formato de texto ligero para el intercambio de datos. Inicialmente se usaba en *JavaScript*pero actualmente se usa en multitud de lenguajes de programación.

Planner/planificador En el contexto de este proyecto, un planificador es un elemento software que toma decisiones en cuanto al despliegue de recursos en los *host*disponibles.

Perfil de ejecución Un perfil de ejecución es una configuración que limita el uso de recursos de memoria y/o cómputo en un contenedor.

Recursos Se hace referencia a todos los elementos que se pueden utilizar dentro de un sistema. En este caso decimos que todos los elementos que se pueden gestionar en la *API*son recursos, pero también lo son la memoria y capacidad de computo de un *host*.

Red overlay Se trata de una red de computadores, creada sobre otra red, o redes, que abstrae los enlaces entre recursos.

Repositorio Una estructura de datos que representa un conjunto de recursos disponibles para su uso, normalmente acompañados de metainformación.

Router Se trata de un elemento software que enruta peticiones a sus correspondientes puntos de acceso o *endpoints*.

Scatter Hace referencia a una estrategia de despliegue de recursos que busca esparcir el coste computacional entre la infraestructura.

Servicios En este caso se hace referencia a aquel software que cumple una función, y que se expone a terceros. También hace referencia a las imágenes de *Docker*.

Socket UNIX Se trata de un punto de acceso para la comunicación entre procesos dentro de un sistema *UNIX*.

Snapshot Es una captura del estado del contenedor en un instante.

URL *Uniform Resource Locator*, o *Localizador de recursos uniforme*, es una dirección que identifica un recurso.

Worker En una configuración maestro-esclavo, es el que desempeña el papel de esclavo, realizando las tareas que el maestro le indica.

.2 Ejemplos de comandos

Consultar todos los contenedores

```
curl -sk -X GET https://localhost:8080/1.0/containers |
json_pp
```

Crear un contenedor

```
curl -sk -X POST -d '{"name": "containerName", "source": {"type":
    "image", "protocol": "simplestreams", "server": "https://cloud
    -images.ubuntu.com/daily", "alias": "16.04"}}' https://
    localhost:8080/1.0/containers | json_pp
```

Ver la condiguración del contenedor

```
curl -sk -X GET https://localhost:8080/1.0/containers/
    containerName | json_pp
```

Cambiar el nombre del contenedor

```
curl -sk -X POST -d '{"name": "newName"}' https://localhost
    :8080/1.0/containers/containerName | json_pp
```

Cambiar la configuración del contenedor

```
curl -sk -X PUT -d '{"config":{"limits.cpu":"3","volatile.
    apply_template":"create","volatile.base_image":"41
    fecb6b51a28d886ac58aba29bf83efb300ba7015179104a62577fa27e4a789
    ","volatile.eth0.hwaddr":"00:16:3e:d6:2c:23"}}' https://
    localhost:8080/1.0/containers/containerName | json_pp
```

Borrar un contenedor

```
curl -sk -X DELETE https://localhost:8080/1.0/containers/
    containerName | json_pp
```

Consultar el estado del contenedor

```
curl -sk -X GET https://localhost:8080/1.0/containers/
    containerName/state | json_pp
```

Arrancar o parar un contenedor

```
curl -sk -X PUT -d '{"action": "start/stop"}' https://localhost
    :8080/1.0/containers/containerName/state | json_pp
```

Consultar un archivo del servidor

```
curl -sk -X GET https://localhost:8080/1.0/containers/
    containerName/files?path=/route/to/file
```

Subir un archivo al contenedor

```
curl -k -X POST -H 'Content-Type: application/octet-stream' -d '
    Content here' https://localhost:8080/1.0/containers/
    containerName/files?path=/route/to/file
```

```
curl -k -X POST -H 'Content-Type: application/octet-stream' --data
    -binary @filename https://localhost:8080/1.0/containers/
    containerName/files?path=/route/to/file
```

Consultar los snapshots del contenedor

```
curl -sk -X GET https://localhost:8080/1.0/containers/
    containerName/snapshots
```

Crear un snapshot a partir del contenedor

```
curl -sk -X POST -d '{ "name": "snapshotName","stateful": false/
    true }' https://localhost:8080/1.0/containers/containerName/
    snapshots | json_pp
```

Ver la configuración del snapshot de un contenedor

```
curl -sk -X GET https://localhost:8080/1.0/containers/
    containerName/snapshots/snapshotName | json_pp
```

Cambiar el nombre del snapshot

```
curl -sk -X POST -d '{ "name": "newName" }' https://localhost
    :8080/1.0/containers/containerName/snapshots/snapshotName |
    json_pp
```

Borrar un snapshot del contenedor

```
curl -sk -X DELETE https://localhost:8080/1.0/containers/
    containerName/snapshots/snapshotName | json_pp
```

Ver los perfiles de todos los host

```
curl -sk -X GET https://localhost:8080/1.0/profiles | json_pp
```

Crear un perfil de ejecución en uno de los host

```
curl -k -X POST -d '{"hostname":"hostName", "profilesPost": {"name
    ": "profileName","description": "Some description string","
    config": {"limits.memory": "2GB" },"devices": {"kvm": {"type":
     "unix-char","path": "/dev/kvm"}}}}' https://localhost
    :8080/1.0/profiles | json_pp
```

Consultar la información de un perfil de ejecución

```
curl -sk -X GET https://localhost:8080/1.0/profiles/profileName
```

Modificar un perfil de ejecución

```
curl -sk -X PUT -d '{"config": {"limits.memory": "3GB"},"
    description": "Just a profile test.","devices": {"kvm": {"path
    ": "/dev/kvm","type": "unix-char"}}}' https://localhost
    :8080/1.0/profiles/profileName
```

Borrar un perfil

```
curl -sk -X DELETE https://localhost:8080/1.0/profiles/profileName
```

Cambiar nombre del perfil

```
curl -k -X POST -d '{"name": "3RAM"}' https://localhost:8080/1.0/
    profiles/lowRAM | json_pp
```

Consultar las imágenes disponibles

```
curl -sk -X GET https://localhost:8080/1.0/images | json_pp
```

Enviar una imagen a todos los host

```
curl -sk -H 'X-LXD-filename: fileName' -H 'X-LXD-public: true/
    false' -H 'X-LXD-properties: os=Ubuntu&alias=image-alias' -X
    POST -d '{"filename": "fileName","public": true,"properties"
    : {"os": "Ubuntu"},"aliases": [{"name": "test-headers", "
    description": "Image description."}],"source": {"type": "url",
    "url": "https://dl.stgraber.org/lxd"}}' https://localhost
    :8080/1.0/images | json_pp
```

Consultar información de la imagen con fingerprint

```
curl -sk -X GET https://localhost:8080/1.0/images/fingerprint |
    json_pp
```

Ejecutar un comando en el contenedor

```
curl -sk -X POST -d '{"command": ["echo"],"environment": {},"wait-
    for-websocket": true,"record-output": false,"interactive":
    false,"width": 80,"height": 25}' https://localhost:8080/1.0/
    containers/containerName/exec | json_pp
```

Obtener el estado del planificador

```
curl -sk -X GET https://localhost:8080/1.0/planner | json_pp
```

Modificar el estado del planificador

```
curl -sk -X POST -d '{"state": "random"}' https://localhost
    :8080/1.0/planner
```