UNIVERSIDAD
REY JUAN CARLOS

GRADO EN INGENIERÍA DE COMPUTADORES

Curso Académico 2017/2018

Trabajo Fin de Grado

DESPLIEGUE ELÁSTICO DE TESTS DE RENDIMIENTO

Autor : Marcos Tenrero Morán

Tutor : Francisco de Asis Gortázar Bellas

Dedicado a

mis padres, a mi hermana y a mi chica, Bárbara

y a los dos más peludos de la casa, Maya y Tasser

II

Agradecimientos

Tengo que agradecer, a mis padres y a Bárbara no sólo haberme apoyado durante la realización del trabajo de fin de grado, sino a lo largo de toda mi carrera educativa. Han estado en los momentos buenos y en los no tan buenos, siempre me han alentado a seguir trabajando y seguir esforzándome para poder conseguir mis objetivos.

Sin su ayuda no hubiese sido posible alcanzar las metas que hemos logrado, y por eso digo 'hemos', porque las hemos alcanzado juntos.

También tengo que agradecer enormemente, aunque ellos nunca vayan a poder entenderlo, a mis perros, Maya y Tasser, la de veces que habrán sentido cuando me encontraba ofuscado y estresado y han venido corriendo para aportar lo mejor de ellos, que es su alegría. No han sido pocas las veces que después de un buen paseo de un par de horas he visto las cosas de otra manera, permitiéndome pensar en un nuevo enfoque.

A todos, gracias.

Resumen

Los test de rendimiento requieren de una gran potencia de cómputo y una gran capacidad de red para poder simular de manera satisfactoria cargas de gran tamaño. Para ello, es común ejecutar estos test de rendimiento apoyándose en un gran número de nodos.

No existen soluciones reales que simplifiquen la tediosa tarea configuración de cada uno de los nodos de una manera específica para la ejecución de tests.

El proyecto ofrece una alternativa, un middleware sobre Docker Swarm que permite aprovechar una infraestructura cloud existente para la orquestación de los contenedores especificados que actuarán como nodos remotos para los citados tests de rendimiento. Añadiendo una capa de abstracción no sólo a Docker y a la gestión de contenedores, sino a toda la capa de red, ya que gestiona y configura automáticamente las redes necesarias para aislar múltiples ejecuciones entre sí y entre el sistema base del host.

Se ha desarrollado en *Go*, ofreciendo así la posibilidad de ser desplegado sobre diferentes Sistemas Operativos.

Aunque se ha diseñado teniendo en cuenta los test de carga, y en concreto basándose en la herramienta Apache JMeter, no sólo puede ser empleado para dichos tests. Sino que es completamente compatible con cualquier despliegue que se base en un modelo de maestro / esclavos.

Índice general

Índice de figuras

Capítulo 1

Introducción y Motivación

La última era de la informática no ha parado de evolucionar durante las últimas décadas, han surgido y se han puesto en práctica diversas metodologías en el desarrollo del software, desde el modelo de procesos hasta las más actuales como las denominadas metodologías ágiles, formadas por una serie de técnicas y de métodos como Extreme Programming , Kanban o Scrum.

En la actualidad, el uso de las metodologías ágiles está en auge, y esto conlleva la adopción de una nueva manera de concebir la informática, dando prioridad a fases que antes no se tenían en cuenta, como es el caso de la fase de testing.

En el mundo del desarrollo del software, el testing, es quizás, una de las partes más importantes del ciclo, ya que permiten asegurar la estabilidad del producto entre las diferentes iteraciones del desarrollo.

Dentro de esta etapa, existen diversos tipos de pruebas a realizar, pero este trabajo de fin de grado se centra en las pruebas de rendimiento.

En términos de calidad de software, a la hora de realizar un test, se tiene muy en cuenta la pirámide de test, la cual pretende dar unas guías generales a tener en cuenta con respecto a la cantidad de test que se deberían desarrollar para un proyecto, teniendo en cuenta su complejidad de codificación y mantenimiento.

Normalmente se sitúan en la cima esta pirámide los tests de sistema o End-To-End (E2E) ya que el coste de éstos es muy elevado al requerir que todos los componentes de la aplicación se encuentren operativos en el momento de realización de las pruebas y al ser más frágiles. Los test de rendimiento se encuentran justo en esta cima.

El análisis del rendimiento de una aplicación es crucial para poder comprobar que ésta escala de una manera apropiada con diferentes cargas de trabajo. Normalmente, es un aspecto que se suele olvidar en las etapas iniciales del desarrollo, ya que es un factor que suele preocupar más en aplicaciones que ya tienen una carga considerable.

No sólo es una etapa relegada debido a que no suele ser un factor preocupante en las primeras iteraciones del desarrollo, sino que también resulta complicado el establecer una infraestructura para poder ejecutar, y sobre todo, mantener este tipo de tests.

Este trabajo cubre el desarrollo de una herramienta que ayude en el proceso del análisis de rendimiento de cualquier aplicación. Pretende ofrecer una manera sencilla de ejecutar estos test de manera automática y distribuida sin tener que preocuparse de la infraestructura subyacente necesaria, ya que normalmente, con los test de rendimiento se pretende simular una gran carga de trabajo sobre la aplicación y esto se traduce en muchas conexiones simultáneas a la misma, y para ello, se necesitan diversos nodos para poder distribuir esta simulación de la carga.

La parte mas tediosa de trabajar con diversos nodos es la configuración de éstos para que estén preparados para realizar su cometido.

Por ello, para una efectiva gestión, es necesario poder desplegar y ejecutar estos tests de manera sencilla, y la única manera de hacerlo ágilmente, es contemplar el paradigma de cloud computing, que aporta el concepto de contenedores y un despliegue rápido de infraestructura bajo demanda.

La herramienta que se ha desarrollado durante la realización de este trabajo de fin de grado añade una capa de abstracción a la tecnología de orquestación de contenedores por excelencia, Docker. Lo que se traduce en la posibilidad de ejecutar cualquier tipo de aplicación que requiera este modelo de despliegue, aunque como ejemplo, se ha tomado como referencia la herramienta Apache JMeter.

Por otra parte, llevo cerca de dos años trabajando en el departamento de automatización y calidad de software en una empresa multinacional de desarrollo de software, estando estos valores y premisas muy presentes en mi vida laboral. En los últimos meses, el análisis de rendimiento ha sido un objetivo muy importante en esta empresa, y mi labor se ha centrado en gran medida en esta tarea, diseñando y desplegando toda la infraestructura necesaria para poner en práctica el proyecto.

Este hecho me ha permitido probar en primera línea los avances sobre mi trabajo de fin de grado en un entorno empresarial real.

1.1. Despliegue tradicional de Apache JMeter

Apache JMeter tiene diversos modos de funcionamiento, ya que ofrece dos interfaces, una gráfica y otra por línea de comandos. Para aprovechar varios nodos y poder lanzar un test de carga de forma distribuida, es necesario ejecutar la aplicación en modo esclavo en cada uno de los nodos remotos en los que se va a ejecutar el test.

Por otro lado, estos nodos deben estar visibles entre sí dentro de la red, lo que implica una configuración manual del firewall para poder así abrir los puertos necesarios.

Un despliegue típico consistiría en los siguientes pasos:

Descarga de Apache JMeter

Bien via repositorio de paquetes *Debian* o *Red-Hat based* o vía fichero comprimido con los binarios pre-compilados por la fundación Apache.

Como ejemplo se va usar una distribución CentOS 7, basada en RedHat utilizando los binarios disponibles desde la página oficial de JMeter.

```
$ wget https://archive.apache.org/dist/jmeter/binaries/ \
apache-jmeter-3.3.zip
$ unzip jmeter-3.3.zip
```

Con estos comandos se dispondría de los binarios de JMeter preparados para ser ejecutados en prácticamente cualquier sistema UNIX.

Configuración del firewall

JMeter requiere que el firewall esté correctamente configurado para permitir la comunicación entre nodos, esto implica la apertura de diversos puertos como **1099**,**1664** entre otros, ya que depende de la configuración que se realice sobre JMeter.

Dependiendo de la distribución sobre la que realizar las configuraciones, se usará un comando u otro, los más típicos son los siguientes:

Ubuntu UFW

```
$ ufw allow <port>/tcp
```

IP Tables

```
$ iptables -A INPUT -p tcp --dport <port> -j ACCEPT
```

Firewall-cmd

```
$ firewall-cmd --zone=public --add-port=<port>/tcp --permanent
```

Tratar con la complejidad de la configuración del firewall y de la red se vuelve realmente complejo y más cuando se dispone de una gran cantidad de nodos y éstos son heterogéneos en lo que a Sistema Operativo se refiere.

Despliegue de esclavos

Apache JMeter requiere de un orden de arranque específico, primero deben arrancarse los nodos esclavos con el siguiente comando:

```
$ ./jmeter-server -Dserver.rmi.localport=4445 -Dserver_port=1099
```

Este comando debe ser ejecutado en cada uno de los nodos que vayan a ser empleados como esclavos.

Arranque del nodo Maestro

Es necesario manejar y actualizar constantemente la lista de las direcciones de los nodos a los que que conectarse a través del nodo maestro para ejecutar un test de carga distribuido ya que es necesario especificar explícitamente los nodos remotos.

Para ejecutar un test de carga distribuido, una vez que la infraestructura previa está correctamente configurada, basta con ejecutar un comando similar al siguiente:

```
$ ./jmeter -n -t $TEST_PATH -R$REMOTES -Dserver.rmi.localport=60000
```

Publicación de resultados

Es posible especificar un ***Listener*** durante la ejecución del nodo maestro junto al test para así poder exportar en tiempo real los resultados del test a un sistema externo como *Apache Kafka*, *ElasticSearch*, una base de datos entre otros.

Consideraciones adicionales: Cifrado de comunicaciones

Es común utilizar un cifrado SSL o TLS en las comunicaciones de red, y JMeter permite encriptar las comunicaciones entre los diferentes nodos con SSL[1]. Esto requiere de la generación de certificados y de un almacén de claves, lo que dificulta, aun mas, el despliegue de un clúster de forma tradicional.

[1] https://jmeter.apache.org/usermanual/remote-test.html#setup_ssl

Capítulo 2

Objetivos

El propósito general de este trabajo es permitir de una manera sencilla y agnóstica poder ejecutar tests de rendimiento de manera distribuida, con capacidad de crecimiento horizontal en cualquier entorno, incluso en una nube privada y con independencia del sistema operativo ya que en el mundo empresarial, es un requisito bastante demandado.

Existen múltiples herramientas para poder ejecutar tests de rendimiento, incluso de forma distribuida, pero la mayoría requieren una configuración previa nodo a nodo, o son soluciones SAAS, alojadas en la nube y no ofrecen ninguna alternativa on-premise. Esto fuerza a una configuración manual de las aplicaciones tradicionales y obliga a tener en cuenta la configuración de red entre ellos.

Para poder ofrecer una funcionalidad descrita en los párrafos anteriores, se ha optado por apoyarse en Docker, y en su modo de funcionamiento como clúster, Docker Swarm, lo que ofrece una nube privada creada a partir de las máquinas o nodos que el usuario considere, siendo posible la creación de un clúster heterogéneo en caso de ser necesario, permitiendo una gran flexibilidad y haciendo posible el despliegue en diversas arquitecturas.

Las funcionalidades principales que ofrece el framework son las siguientes:

- Interfaz sencilla para el usuario (Representational State Transfer (REST))
- Orquestación de contenedores, redes y volúmenes de forma transparente

- Carácter efímero y reusable de la infraestructura

Conseguir las funcionalidades descritas conlleva cumplir una serie de objetivos intermedios:

- Construir un Middleware que comunique con la Interfaz de Programación de Aplicaciones (API) que expone el demonio de Docker
- Diseñar un plan de acción para la orquestación de contenedores
- Descubrir la configuración de red apropiada para el carácter de la aplicación
- Diseñar una API REST para el consumidor
- Construir un proceso de orquestación de recursos Docker

Además he fijado una serie de requisitos metodológicos para asegurar un desarrollo con la forma mas práctica posible.

- Integración continua desde el comienzo del proyecto
- Gestión ágil del proyecto
- TestDrivenDevelopment (TDD) Test-Driven-Development

Capítulo 3

Metodologías

Este proyecto ha seguido la filosofía de las metodologías ágiles, es decir, un desarrollo e incremental.

Dado el carácter dinámico de la disponibilidad del tiempo, se ha optado por una planificación **Kanban**, una vez definidos los requisitos y plan de acción del proyecto.

Además ha sido vital afrontar la filosofía DevOps, aplicando desde el comienzo del proyecto la integración continua, para asegurar la salud del producto con cada nueva iteración, ya que al trabajar directamente con la API del demonio de Docker, cualquier mínimo cambio afectaba al resto del proyecto.

3.1. Extreme Programming

Extreme Programming es una práctica bastante extendida en el mundo del desarrollo de software. Obliga a definir la estructura del código e incluso los tests antes de desarrollar el producto final, de tal manera, en primer lugar se diseñan y se implementan los tests, los cuales fallarán hasta que la funcionalidad esté implementada completamente.

Esta metodología obliga al programador a pensar en el diseño en un primer lugar, dotando al software de mas calidad.

3.2. Kanban

Kanban es un modelo ágil de organización del desarrollo de software que se caracteriza en la división de las tareas en diferentes estados, típicamente, *PENDIENTE*, *EN PROGRESO* y *TERMINADO*.

Se ha utilizado la gestión de proyectos en el propio repositorio que ofrece la plataforma GitHub[1] para que estuviese disponible para la comunidad de forma sencilla. Además ofrece sincronización automática de tareas basada en *Issues* y *Pull Requests*.

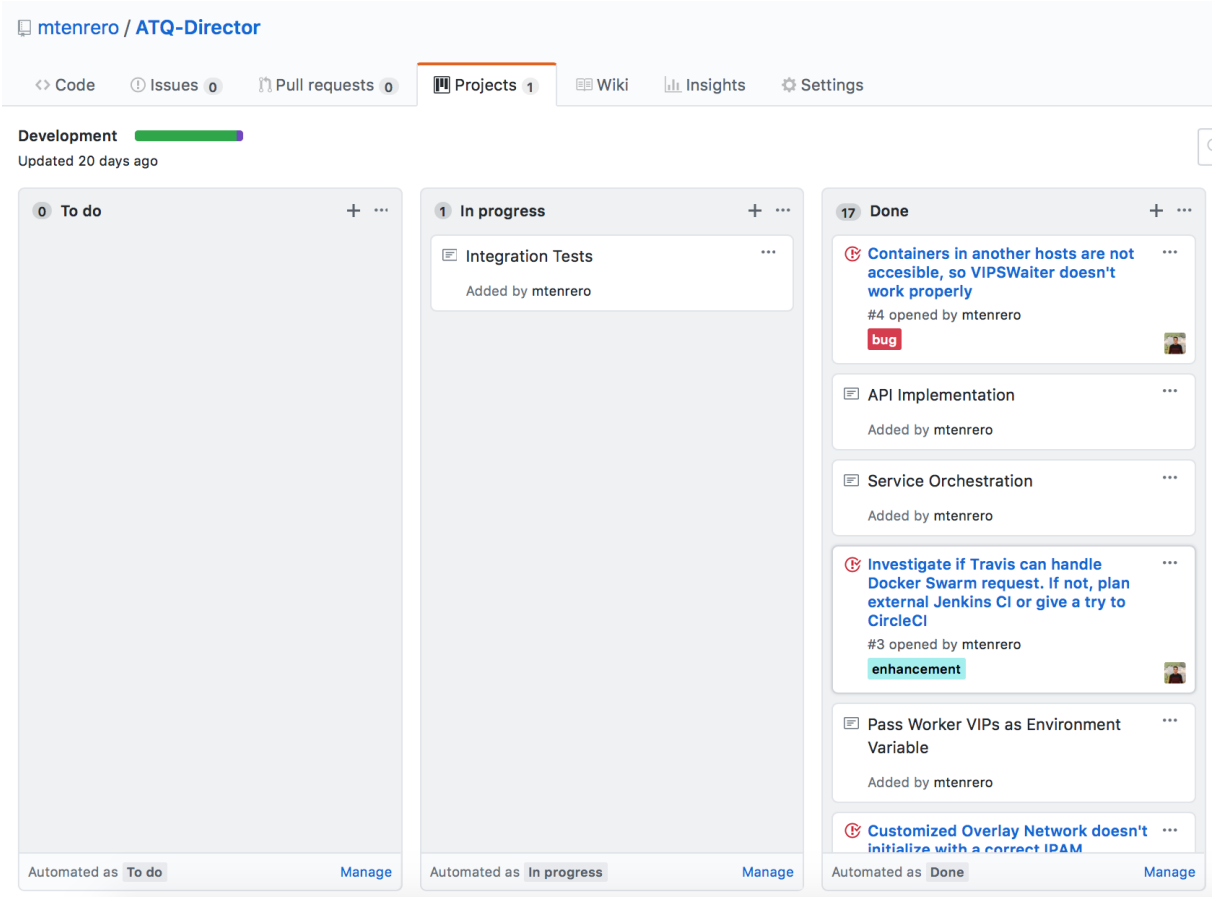

Figura 3.1: Tablero Kanban del proyecto en GitHub

[1] https://github.com/mtenrero/ATQ-Director/projects/1

Capítulo 4

Tecnologías y Herramientas

4.1. Golang

Go es un lenguaje de programación que se comenzó a desarrollar en 2007 [11] y nació con el ideal de eliminar todos los obstáculos de la programación actual [4], ya que desde hace varios años no había sido desarrollado ningún lenguaje de programación de alta importancia. Se necesitaba que estuviese diseñado por completo, teniendo en cuenta factores de la informática actual, como la concurrencia o la rapidez en la compilación y en la codificación.

Los orígenes de Go se remontan a los lenguajes Oberon 2 , C y Alef [4]. Nace como un proyecto de Google como solución para la codificación de soluciones complejas.

Como particular característica [10], Go es un lenguaje de programación con recolector de basura, para permitir así trabajar de una forma correcta con la concurrencia de las aplicaciones.

El compilador de Go se ideó de tal forma para que fuese compatible nativamente con todos los Sistemas Operativos, introduciendo el cross-compile (Compilación para otras plataformas o arquitecturas en un único Sistema Operativo) como uno de sus puntos fuertes.

4.2. Dep: Go Dependency Management

Uno de los puntos fuertes de *Go*, es la gestión de dependencias durante la compilación [10]. Sin embargo, de cara al desarrollador, por defecto, carece de un fichero a nivel global de proyecto para poder definir las dependencias del mismo y/o la versión con la que se desea trabajar, sino

que se debe especificar a través de sentencias *import* en los ficheros *.go* del código. Además, para poder usar esas dependencias, es necesario que estén presentes en el $GOPATH del sistema.

Dep nace como un experimento de *Golang*[1] , preparada para usar en producción, aunque sin llegar a ser la herramienta oficial de gestión de dependencias a nivel de proyecto/usuario. Emplea un fichero TOML en la raíz del proyecto, en él se indica la dependencia requerida, su versión e incluso la rama del control de versiones desde la cual obtener los paquetes.

Para conseguir la capacidad de gestión de las versiones de dependencias a nivel de proyecto, dep descarga todas las dependencias en el directorio local *vendor* para que estén disponibles en el propio proyecto, al estilo del directorio *node_modules* en el caso de NodeJS.

Esta distribución de ficheros evita al compilador de *Go* tener que buscar en el $GOPATH las dependencias necesarias.

4.3. Goa Design: Design-first Network framework

Goa[2] es un framework completo para construir microservicios en Go que invierte la forma de construir APIs web completamente. Posee generación automática de código y documentación.

Es un framework enfocado al diseño de la API en primer lugar, y es lo que hace a este framework único. Posee su propio lenguaje DSL (lenguaje descriptivo) en el que antes de codificar, obliga al programador a pensar en el diseño de la API, ya que esta debe ser definida en un primer lugar. Permite definir desde el endpoint, el contenido que consumirá y los parámetros que recibirá.

Dispone de un comando generador, *goagen*, que produce todo el código necesario a partir

[1] https://github.com/golang/dep

[2] https://goa.design

de la descripción proporcionada en los ficheros de definición. No está completamente integrado con *dep* y para el correcto funcionamiento del comando *goagen* se requiere que no exista la carpeta *vendor*.

4.4. OpenAPI Specification

OpenAPI es una especificación mantenida e ideada por la comunidad Open-Source en la plataforma GitHub[3], independiente de cualquier lenguaje de programación, que permite definir cualquier tipo con toda la especificación completa de una API, para que sea comprensible, tanto para personas como para ordenadores, ya que se basa en ficheros JSON y YAML.

4.5. DNS

Domain Name System es un sistema de nombrado de redes IP que se utiliza para resolver nombres en direcciones IP. El servicio de resolución de nombres puede tener diferentes tipos de registros. En la investigación del trabajo sólo se han usado registros de tipo **A**, los cuales traducen un nombre en una o varias direcciones IP, dependiendo de los registros A que tenga almacenados para un mismo nombre de dominio o subdominio.

4.5.1. DNS Round-Robin

DNS Round-Robin es un algoritmo de selección de IP, en el que con cada petición que realiza el usuario, se obtiene una lista completa de todas las direcciones IP registradas en el servidor de nombres ordenada de tal forma que nunca recibe la misma dirección, realizando, de esta manera, realiza labores de balanceador de carga.

Docker implementa este algoritmo de forma alternativa en su DNS interno, y el desarrollador puede elegir si desea operar con DNS Round Robin o con *Routing Mesh*, balanceo de carga interno, donde el nombre es resuelto con una única IP Virtual de manera no determinista.

Cuando se opta por usar el modo DNS Round-Robin, por limitaciones de diseño, resulta imposible exponer puertos hacia el host, únicamente se permiten conexiones de red con las

[3]https://github.com/OAI/OpenAPI-Specification

redes indicadas por el servicio.

4.6. Contenedores

Los contenedores son un nuevo tipo de virtualización ligera de Sistemas Operativos que no requieren la emulación de instrucciones en el procesador, reduciendo así el consumo de recursos sin prescindir del aislamiento necesario presente en la virtualización tradicional [5].

4.6.1. LXC: Linux Containers

LXC es un tipo de virtualización basada en contenedores, usa el espacio de nombres de kernel y cgroups para asegurar el aislamiento de los contenedores, aunque comparten el kernel con el Sistema Operativo del host [8].

4.6.2. Docker

Docker añade una capa para gestionar las redes de los contenedores dotándolos de una IP propia y sistema de ficheros a LXC. Permite especificar imágenes de contenedores autocontenidas y gestionar el ciclo de vida de los mismos. [5] [2]. Introduce un modelo cliente-servidor, siendo el demonio de Docker en el host el responsable de la comunicación con los contenedores [8]

Docker ofrece además un repositorio o registro donde almacenar estas imágenes de los contenedores para que estén disponibles para su uso [9] de manera pública a través de DockerHub[4], o de manera privada con un registro desplegado a demanda.

Docker usa una terminología propia:

- **Imagen:** Descripción de los contenidos de un contenedor, interoperable entre todo el ecosistema Docker.
- **Contenedor:** Es la unidad mínima en Docker, corresponde a una imagen ejecutada sobre una máquina o Swarm.

[4] https://hub.docker.com

- **Volumen:** Unidad de almacenamiento persistente en Docker, pueden ser bindados a un directorio o fichero del sistema de ficheros del host o en el propio contexto de Docker.

- **Servicio:** Encapsulación de una imagen que se puede configurar de manera concreta. Permitiendo configurar la cantidad de replicas a desplegar, el tipo de red o los volúmenes con los que trabajar.

- **Red:** Red dentro del ecosistema Docker, se encarga de interconectar Servicios y/o Contenedores

Raft Consensus

Es un algoritmo para asegurar tolerancia a fallos. Para su correcto funcionamiento, se tiene que establecer un *Quorum*, un conjunto de nodos elegidos para administración. El tamaño del *Quorum* se define por la siguiente ecuación $2f + 1$, siendo f la cantidad de fallos a tolerar. Por lo tanto, para formar *Quroum*, serán necesarios un mínimo de 3 nodos [6] .

Docker Swarm

Docker Swarm es un modo de funcionamiento de Docker que permite utilizar diversas máquinas para formar un clúster de Docker en el que desplegar contenedores, donde el demonio de Docker gestiona la red, el sistema de ficheros y la comunicación entre servicios.

Para desplegar diversos Servicios en un Swarm, Docker introduce el concepto de **Stack**, una agrupación de Servicios con su configuración de red particular.

Los nodos de Docker Swarm pueden tener dos roles:

- **Manager:** Gestionan el clúster, como mínimo, debe haber 3 Managers para que se puede aplicar Raft-Consensus. Tienen una visión completa del cluster. A su vez, uno de ellos se elegirá como **Líder**, el cual será responsable de la gestión del clúster.

- **Worker:** Son nodos en los cuales sólo se despliegan contenedores o servicios, sólo tienen visión de los contenedores o servicios que están desplegados en el propio nodo.

4.7. Apache JMeter

Apache JMeter es un software de código abierto desarrollado en Java con el objetivo de realizar test de rendimiento automáticos. Tiene una baja curva de aprendizaje y permite ejecutar las pruebas de manera distribuida y una API pública lo que permite extender el funcionamiento de la herramienta a demanda.

Está dotado de interfaz de usuario gráfica desde la cual se pueden diseñar y ejecutar tests, pero también ofrece una versión de ejecución por línea de comandos que permite automatizar completamente el lanzamiento de las pruebas.

Ejecución distribuida

JMeter se desarrolló pensando en un esquema de ejecución distribuida **Maestro / Esclavos**. Los esclavos deben estar accesibles antes de la ejecución del test a través del nodo maestro.

El lanzamiento del nodo Maestro requiere que se especifique la lista de los nodos remotos que utilizar para lanzar el test de rendimiento.

4.8. ElasticSearch

ElasticSearch forma parte del Stack ELK (*ElasticSearch*, *Logstash*, *Kibana*) y es una base de datos y motor de búsqueda distribuido y enfocado a la alta disponibilidad. Está desarrollado en Java bajo una licencia open source y basado en Lucene [3], una libreria open source de búsqueda de texto completo.

4.9. Grafana

Grafana[5] es un panel UI que sirve para visualizar datos de diferentes orígenes de datos, como bases de datos relacionales, ElasticSearch, Graphite o CloudWatch.

[5]https://grafana.com

Se conecta directamente al origen de datos y no requiere de ninguna importación importación previa para poder trabajar y consultar los datos.

4.10. Sistema de ficheros distribuido

Un sistema de ficheros distribuido permite acceder a unos directorios o ficheros determinados desde diversos ordenadores. Cada Sistema Operativo implementa su propio protocolo de ficheros distribuidos. El sistema Operativo Windows utiliza Samba.

4.10.1. Samba

Samba es el protocolo open source por defecto de Windows que comparte directorios e impresoras como estándar. Su carácter open source permite que sea posible acceder a recursos compartidos Windows desde otros Sistemas Operativos[7].

4.10.2. GlusterFS

GlusterFS[6] es un sistema de ficheros distribuido open source a través de la red que puede replicar los contenidos entre los nodos que lo tengan configurado. Las características principales de Gluster son las comunes a cualquier sistema distribuido, tolerancia a fallos y alta escalabilidad [1].

4.11. Amazon Web Services

Amazon Web Services es una solución Platform-as-a-Service (PAAS) que ofrece diversos productos, entre ellos:

- **EC2:** Recursos de cómputo bajo demanda. Permitiendo desplegar máquinas virtuales en pocos minutos con la imagen de sistema deseada, ya sea Windows o Linux.
- **ElasticIP:** IP pública bajo demanda

Se caracteriza por ofrecer un modelo de pago por uso y por la capacidad de interconexión entre sus servicios.

[6] http://gluster.org

4.12. Sistemas de Integración Continua

Un Sistema de Integración Continua, permite ejecutar con cada commit. sobre el Control de Versiones, ejecutar una serie de programas definidos en un pipeline, como puede ser la ejecución de tests unitarios para asegurarse de la regresión de la aplicación, es decir, que no se ha roto ninguna funcionalidad previo con los nuevos cambios introducidos.

4.12.1. TravisCI

TravisCI[7] es un servicio de Integración Continua ofrecido de manera gratuita para proyectos open-source de GitHub. Está pre-configurado para usarse de manera sencilla para una gran cantidad de lenguajes de programación así como para despliegues en servicios PAAS como es el caso de Heroku[8]. Usa ficheros YAML para su configuración y todos los comandos y procesos indicados en los ficheros de configuración se ejecutan sobre contenedores completamente aislados. Ofrecen Docker-In-Docker de manera limitada, únicamente para construir imágenes y publicarlas en un registro.

4.12.2. Jenkins

Jenkins es una aplicación open-source desarrollada en Java enfocada a Integración Continua y Despliegue Continuo (CI/CD), al igual que TravisCI. Es completamente configurable y versátil, aunque requiere una instalación y configuración previa. Además, aporta la flexibilidad de poder instalar o desarrollar plugins de terceros.

Permite configurar los pipelines de ejecución mediante una interfaz visual y de manera declarativa o procedural a través de código en el propio repositorio.

[7] https://travis-ci.com

[8] https://www.heroku.com

Capítulo 5

Descripción informática

5.1. Requisitos

Teniendo en cuenta las carencias observadas durante el despliegue manual y tradicional de Apache JMeter se observaron diferentes puntos a mejorar:

- Configuración manual nodo a nodo
- Obligatoriedad de administración y mantenimiento de la infraestructura
- Seguridad en las comunicaciones de red
- Crecimiento horizontal real

Tras un análisis del proyecto a desarrollar, se llegó a la conclusión de que había que seguir un enfoque top-down, o de arriba hacia abajo. Era necesario que durante toda la etapa del desarrollo, se tuviese muy claro a dónde se pretendía llegar. Siendo los objetivos principales los siguientes:

- **Capacidad de desplegar los servicios Docker necesarios para una tarea específica** de manera transparente y automática
- **Networking automático y transparente** que cifre o aísle las comunicaciones entre todos los contenedores.
- **Interfaz sencilla para el usuario** para crear una capa de abstracción de la administración de sistemas tradicional.

- **Integración completa con Apache JMeter** ya que es la herramienta en la que se basa el proyecto, sin olvidar que el middleware o framework a desarrollar debería permitir ser usado con cualquier otra herramienta.

Así se mostró en el tablero Kanban del proyecto, proporcionado por GitHub para el repositorio concreto[1].

5.2. Arquitectura y Análisis

Golang fue el lenguaje de programación elegido debido a su alta simplicidad en cuanto a concurrencia se refiere, por permitir ser compilado y generar binarios para los mayores Sistemas Operativos y debido a la existencia de un paquete distribuido por Docker para comunicarse directamente con la API que ofrece el demonio de Docker.

La mayor complejidad presente en este proyecto ha sido obtener las direcciones virtuales de cada contenedor desplegado sobre el cluster Swarm, ya que son necesarias para poder comunicarse con las instancias directamente. Docker, por defecto, cuando crea una red entre servicios, puede invocar directamente al nombre del servicio para acceder a él, pero en el caso de que se despligue el servicio en modo replicado y éste haya sido configurado para tener más de una replica, por defecto, Docker actúa como un balanceador de carga y sólo responde con una única dirección IP correspondiente a un único contenedor. Esto ha sido un duro handicap, ya que se requería conocer todas las direcciones de los contenedores desplegados.

Durante el análisis de cómo afrontar esta problemática, surgieron dos enfoques totalmente diferentes

5.2.1. Primer enfoque: Starter en cada contenedor a desplegar

Este enfoque[2] surge tomando como referencia la mayoría de herramientas para service discovery usadas en Docker y en Kubernetes como Consul.io[3].

[1] https://github.com/mtenrero/ATQ-Director

[2] https://github.com/mtenrero/AutomationTestQueue

[3] https://www.consul.io/

Cada contenedor a desplegar, tiene una imagen modificada, la cual, antes de ejecutar el entrypoint predefinido en la imagen, ejecuta una pequeña aplicación, y obtiene la IP virtual del contenedor y se la comunica al agente controlador, el cual lleva un listado de todas los contenedores descubiertos.

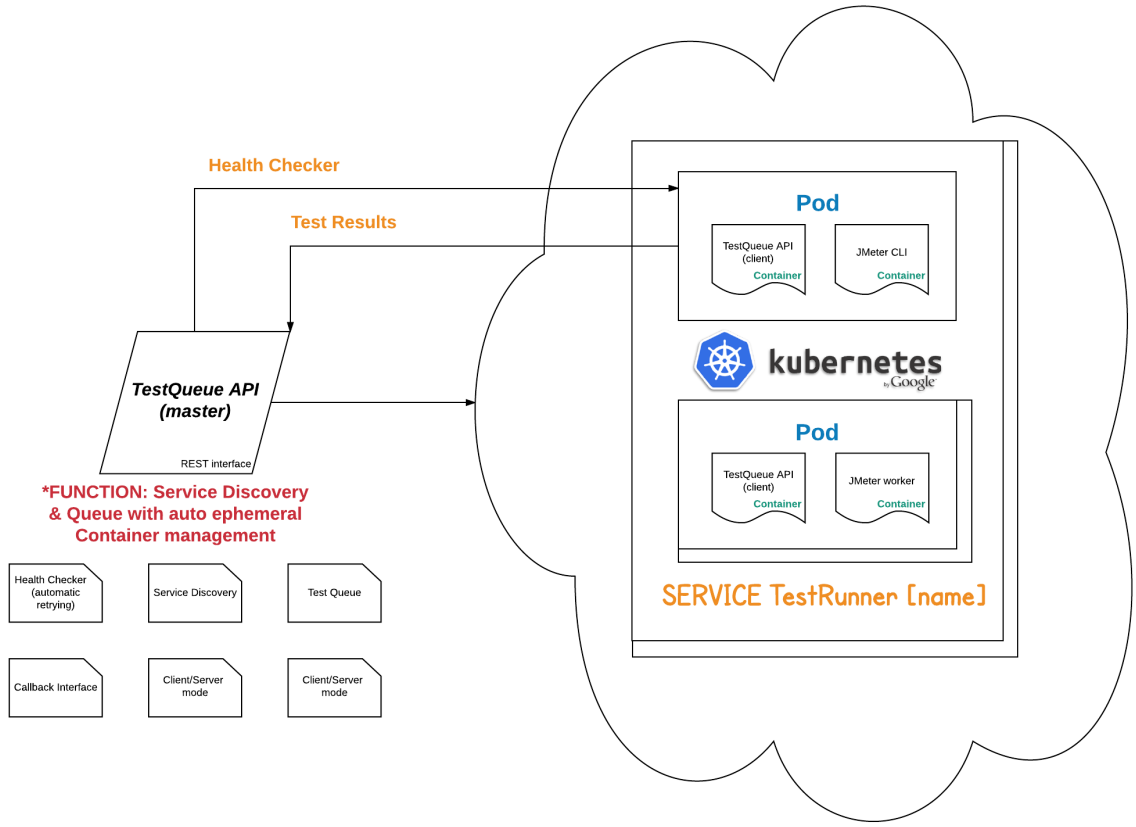

Figura 5.1: Arquitectura 0 Starter en cada contenedor

Este enfoque, implicaba modificar todas las posibles imágenes a usar con la aplicación, con lo que se reducía drásticamente la facilidad de uso con otras aplicaciones al requerir que en caso de no existir la imagen modificada deseada a desplegar con Automation Test Queue, obligaría al usuario a modificarla por él mismo.

5.2.2. Segundo enfoque: Single Daemon y DNS Round Robin

Con una configuración de red entre servicios Docker configurada para operar con el algoritmo DNS Round Robin, se consigue disponer de la lista de todas las direcciones IP Virtuales de

los contenedores de un mismo servicio.[4]

Disponiendo de esta información, dejaría de ser necesario un descubrimiento de servicios como el expuesto en el anterior punto, dotando a la aplicación de la sencillez de uso deseada debido a que el usuario ya no tendría que modificar la imagen a utilizar.

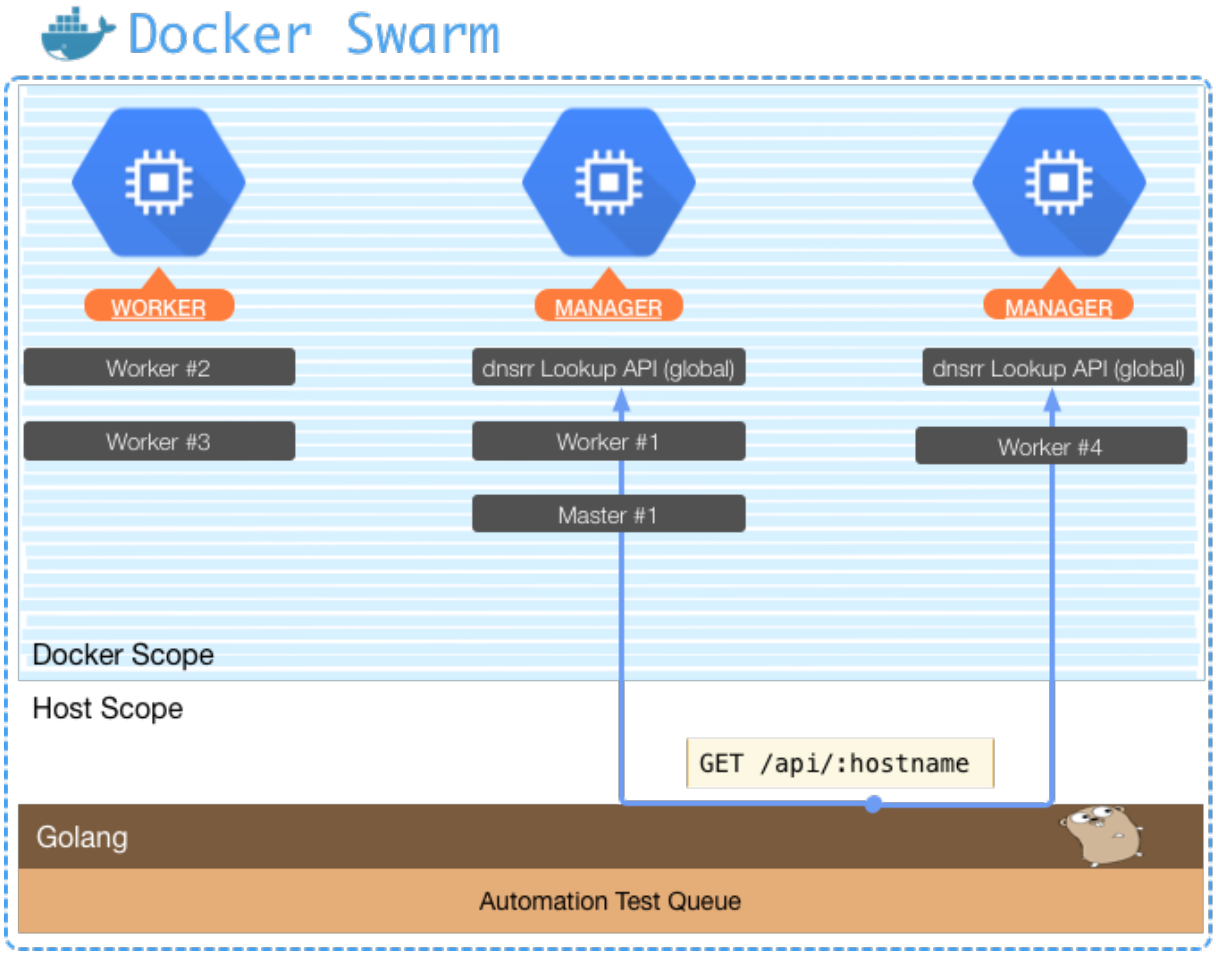

Figura 5.2: Arquitectura 1 DNS RR con Auto Discovery

De esta manera, el middleware ***Automation Test Queue*** se comunicaría directamente con el demonio de Docker a través del SDK proporcionado por la comunidad y se encargaría de toda la orquestación necesaria para el despliegue automático de test de carga en el clúster.

[4]https://github.com/mtenrero/ATQ-Director

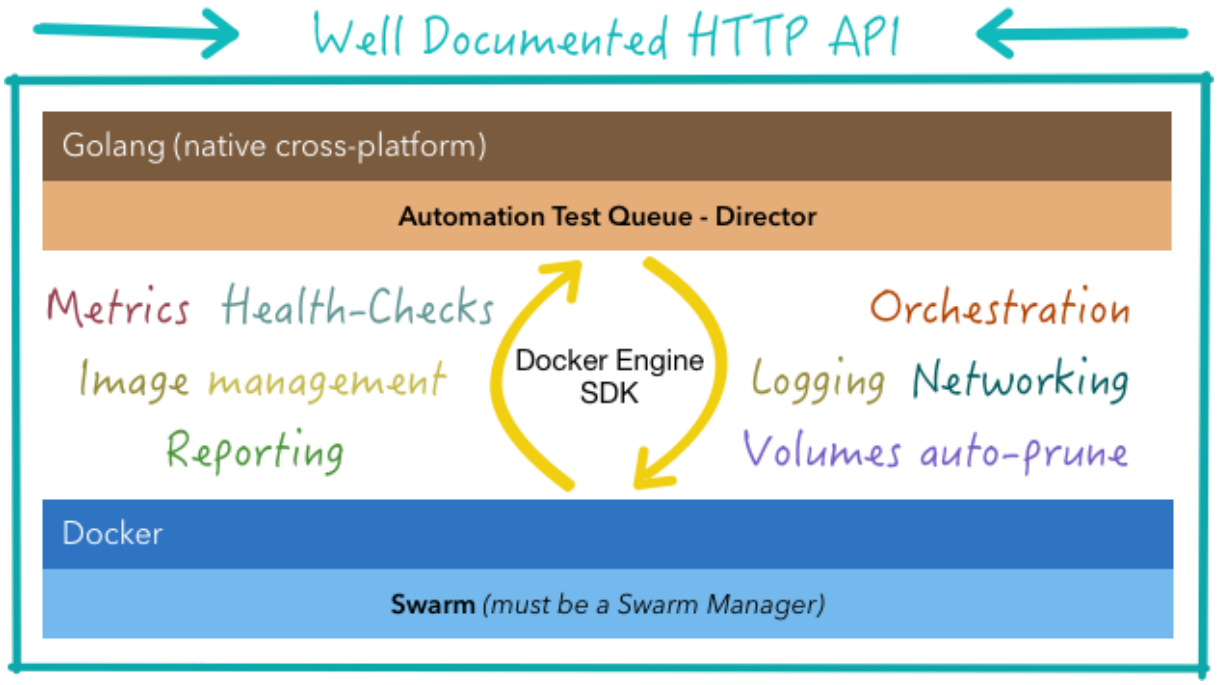

Figura 5.3: ATQ low level architecture

5.2.3. Estructura del proyecto

El proyecto está íntegramente codificado en *Go*. Sigue una estructura típica de este lenguaje, siempre teniendo en cuenta que se mencionan rutas relativas a la ubicación del repositorio (/github.com/mtenrero/ATQ-Director) bajo el $GOPATH definido en el sistema.

```
app Código autogenerado por Goa
client Implementación de un cliente en Go, autogenerado
configLoader Implementación de la configuración externalizada
dnsdiscovery Implementación discovery DNS
dockerMiddleware Middleware Docker
http
  design Diseño API REST con el DSL de Goa
persistence Implementación persistencia datos
swagger Especificación OpenAPI autogenerada
tool Helper autogenerado para trabajar con la API
types Mapeado tipos Goa/Middleware
monitoring.go Controlador Monitoring
swarm.go Controlador Swarm
task.go Controlador Task
databind.go Controlador Databind
```

En el esquema anteriormente presentado, se puede apreciar que la implementación ha sido

fragmentada en grandes paquetes para cumplir el estándar de *Go* y de los principios *KISS* y *SOLID*.

En la raíz del directorio del proyecto se pueden apreciar unos ficheros con extensión *.go*, contienen los controladores para cada endpoint definido con el DSL de *Goa* en el paquete *http/design*. Se genera la estructura básica en la primera ejecución de *goagen* y a partir de ahí se integra la implementación propia de cada endpoint.

El paquete **persistence** contiene la implementación de un almacén de datos clave/valor muy básico que se utiliza para el almacenamiento de tareas planificadas y las características definidas por el usuario.

Para gestionar la configuración externalizada a través de ficheros YAML, se ha creado el paquete **configLoader**. Es responsable de la carga de parámetros desde el fichero especificado.

Los paquetes **dockerMiddleware** y **dnsdiscovery** contienen la implementación mas significativa para el framework y se detallarán más adelante.

5.2.4. Apache JMeter dockerizado

Existen multitud de imágenes de JMeter disponibles en DockerHub[5], todas creadas por la comunidad. Al no existir ninguna imagen oficial, para no dar nada por sentado, se construirá una imagen Docker de Apache JMeter desde el sistema operativo básico para así poder controlar completamente el funcionamiento de la herramienta.

Además se debe ofrecer un modo de configuración a través de variables de entorno para que sea fácilmente ejecutable por línea de comandos para garantizar una integración completa con la automatización.

Las variables de entorno que definirán el modo de operación del contenedor son las siguientes:

[5] https://hub.docker.com/

- **MODE**

  master: Nodo principal y coordinador del test

  node: Nodo esclavo para el test de carga distribuido

- **TEST_PATH**: Ruta del fichero .jmx con el test a ejecutar

- **REMOTES**: Lista separada por comas con las direcciones de los nodos esclavos a usar en el test distribuido

5.3. Diseño e Implementación

Aunque, la planificación de la herramienta se ha realizado con un enfoque bottom-up, el diseño de la misma va a ser expuesto utilizando el enfoque contrario, top-down, es decir, desde lo mas abstracto a lo más específico.

5.3.1. API REST y modelo de datos

El diseño de la interfaz con la que trabajará el usuario que utilice esta aplicación ha sido diseñada meticulosamente utilizando el framework de Go *GoaDesign*, el cual, obliga a definir el diseño de la API con un lenguaje declarativo propio antes de la implementación.

Se valoraron otros frameworks de red para Go, pero ninguno de ellos se enfocaba en primero el diseño y posteriormente la implementación, siendo *Goa* el único que si lo ofrecía. Además, una parte fuerte de este framework es la generación automática de Documentación, ya que genera ficheros OpenAPI compatibles con Swagger[6].

La definición del modelo de datos con el que trabaja la API REST está fuertemente ligado a la fase de definición de la API con *Goa*, ya que este framework obliga a definir las estructuras de datos en su propio lenguaje DSL, y es el propio framework el que genera las *structs* de *Go* cuando se genera el código con el comando *goagen* .

[6] http://swagger.io

Tarea

Se definió el concepto **Tarea** como la definición de las pruebas a ejecutar usando el framework Automation Test Queue.

Una tarea tiene una serie de propiedades tales como:

- **Delay** Segundos a esperar entre el despliegue del servicio de workers y el servicio maestro.
- **Name** Nombre de la tarea a desplegar, es su identificador único en el clúster.
- **WaitCommand** Comando a ejecutar en un contenedor aislado, para asegurar que se han desplegado los contenedores del servicio worker correctamente.
- **Master** Definición del servicio Maestro
- **Worker** Definición del servicio Worker

Definición de Servicio

Los servicios Maestro y Worker tienen los mismos parámetros de definición, y a nivel de estructura no los diferencia ninguna característica.

En el contexto del framework Automation Test Queue, un servicio es la definición de la imagen Docker a desplegar con una serie de características de despliegue ligadas al propio framework.

- **Alias** Identificador del servicio
- **Args** Argumentos que se ejecutarán en cada contenedor del servicio
- **Environment** Lista de variables de entorno que se configurarán en cada contenedor del servicio. Existe una variable reservada para uso interno del framework, **WORKER_CSV_VIPS**, cuya funcionalidad se detallará en los próximos capítulos.
- **FileID** Identificador de un fichero subido previamente a través del endpoint */databind/upload* que se montará en el servicio para que su contenido esté disponible en los contenedores del servicio.

- **Image** Nombre de la imagen Docker a utilizar.
- **Replicas** Cantidad de contenedores a desplegar para el servicio, que se corresponde a la cantidad de nodos esclavos a usar en el test de carga.
- **Tty** Parámetro que fuerza la consola interactiva en el servicio.

API Endpoint

Se ofrecen diferentes acciones descritas a continuación

- **/databind** Operaciones relacionadas con la gestión de archivos

  GET /list Devuelve una lista de los ficheros disponibles en el orquestador

  POST /upload Subir un fichero .zip con contenidos para que esté disponible para una futura tarea

- **/monitoring** Monitorización del framework

  GET /ping Devuelve un HTTP 200OK si el orquestador está operativo

- **/swarm** Estado del cluster Swarm

  GET / Devuelve los detalles del Cluster Swarm

- **/task** Operaciones con las tareas a lanzar con el framework

  PUT /task Crea una nueva tarea

  DELETE /task/$\{id\}$ Elimina una tarea ya planificada

  GET /task/$\{id\}$ Inspecciona una tarea planificada

Durante el diseño de la API, se tuvieron en cuenta todos los tipos de respuesta que podía ofrecer y el contenido de cada mensaje. Se puede consultar en la definición completa de la API en los anexos

5.3.2. Docker Middleware

El Middleware de Docker fue una de las primeras tareas realizadas durante el desarrollo del framework. Su función es crítica, se encarga de comunicarse con el demonio de Docker a través

del SDK proporcionado por Docker para el lenguaje Golang[7].

Se comenzó el proyecto[8] con la primera implementación de Docker SDK[9]. Esta librería pronto quedó deprecada en favor de *go-docker*, librería especificada en el párrafo anterior.

Como la librería aún estaba indicada como *WORK-IN-PROGRESS*, se decidió crear un nuevo repositorio por si fuese necesario consultar o volver a los ficheros fuente anteriores. Además, en el primer repositorio, se empezó a trabajar sobre el primer enfoque, basado en el auto registro de servicios.

Ofrece una interfaz para las tareas más sencillas ofrecidas por la API de Docker como:

- **Gestión Volúmenes**

  Bindado

  Eliminación

- **Gestión Redes Overlay**

  Creación

  Agregación

  Eliminación

- **Gestión Servicios**

  Creación

  Eliminación

- **Tareas auxiliares**

- **Mapeo de configuraciones**

[7] https://github.com/docker/go-docker

[8] https://github.com/mtenrero/AutomationTestQueue

[9] https://godoc.org/github.com/docker/docker/client

Además, ofrece multitud de abstracciones a la estructura definida por el paquete oficial de Docker preconfiguradas para los propósitos del framework.

Este paquete también contiene la implementación de la orquestación propia del framework Automation Test Queue debido a la alta dependencia de las funciones implementadas en estos paquetes.

5.3.3. Fase de orquestación

El proceso de orquestación está implementado en el paquete dockerMiddleware. El proceso parte de una definición de una tarea que se recibe vía API HTTP. Se comienza con el despliegue de un servicio global auxiliar, creado específicamente para el proyecto.

Despliegue del descubridor de servicios

Este servicio contiene otra aplicación *Go* muy básica creada específicamente para este proyecto. El servicio expone otra API REST cuyo principal cometido, es devolver una lista con las direcciones IP encontradas para un determinado nombre de dominio. Para el correcto funcionamiento de este servicio es necesario que el servidor DNS al que se dirige la petición responda con una lista completa de todos los registros **A** disponibles para el dominio consultado, y ésto, en el contexto de Docker, sólo es posible si el el servicio identificado por el nombre de dominio proporcionado está desplegado con una configuración de red de DNS Round-Robin.

Se utiliza el paquete *net* proporcionado por defecto por el lenguaje para realizar la resolución de nombres DNS. El único endpoint que expone la API es el siguiente:

- *GET /api/:hostname* Devuelve las IPs encontradas en el DNS para el *hostname* indicado

Las únicas respuestas posibles por esta API son:

- **HTTP 200:OK** Lista con las direcciones encontradas.
- **HTTP 204: No Content** No se han resuelto nombres para el dominio especificado.

La aplicación se ha dockerizado y publicado tanto en GitHub[10] como en DockerHub[11] con una construcción automatizada, de tal manera, que con cada actualización sobre el repositorio, se construye automáticamente una nueva imagen y se publica en DockerHub para que esté disponible de forma automática para toda la comunidad.

Desde un punto de vista arquitectónico, el servicio Docker se despliega de modo global, exponiendo el mismo puerto para que sea accesible en todos los nodos controladores del clúster y asegurarse la accesibilidad desde el framework Automation Test Queue.

El carácter de este servicio es efímero, una vez que ha cumplido su cometido, al finalizar la orquestación de la tarea, el servicio es eliminado.

Despliegue de Workers

Una vez que el servicio de descubrimiento de direcciones está correctamente desplegado, se comienza con el despliegue de los contenedores del servicio Worker. El despliegue se realiza con un servicio configurado con una red DNS Round-Robin para que puedan ser descubiertas todas las direcciones de los contenedores, como se ha explicado con anterioridad, y con la cantidad de réplicas definida en la tarea especificada por el usuario.

Para asegurarse de que todos los nodos están disponibles, de manera genérica, mediante una espera activa, por medio de canales de *Go*, se espera hasta que el descubrimiento de servicios devuelve una cantidad de direcciones igual a la cantidad de réplicas especificadas en la tarea por el usuario. Una vez que se ha detectado la cantidad correcta de contenedores, si se ha especificado en la tarea, se realizan comprobaciones de salud sobre todos los contenedores.

El comando de comprobación de salud es ejecutado en un nuevo contenedor que comparte la misma red de los servicios desplegados para garantizar una seguridad mínima en el sistema. Por un lado, se aísla el sistema host, ya que se ejecuta en un contenedor aislado, sin capacidad de conectarse al socket de Docker, por otro, al solo disponer acceso a la red interna de la tarea, dispone de un acceso muy limitado al resto de recursos de red, lo que limita el riesgo en los

[10] https://github.com/mtenrero/dnsrr-discovery-api

[11] https://hub.docker.com/r/tenrero/dnsrr-discovery-api/

posibles comandos a ejecutar por parte del usuario.

Una vez que se ha verificado la salud de todos los contenedores, se procede a la orquestación del servicio Master.

Las direcciones descubiertas se guardan para posteriormente inyectárselas al servicio Master.

Despliegue de Maestros

El despliegue del servicio maestro es muy similar al despliegue del servicio Worker, pero con unas particularidades. El modo de red es configurado como replicado y VirtualIP,en lugar de DNS Round-Robin, con una única replica a desplegar, de esta manera se permite que el contenedor exponga puertos hacia el exterior en caso de que sea necesario.

Una vez que se ha finalizado la orquestación del servicio Master, se elimina el servicio Discovery desplegado al inicio de la orquestación de la tarea.

Visión general

En el siguiente diagrama se expone un esquema del proceso general de orquestación de tareas del middleware Automation Test Queue.

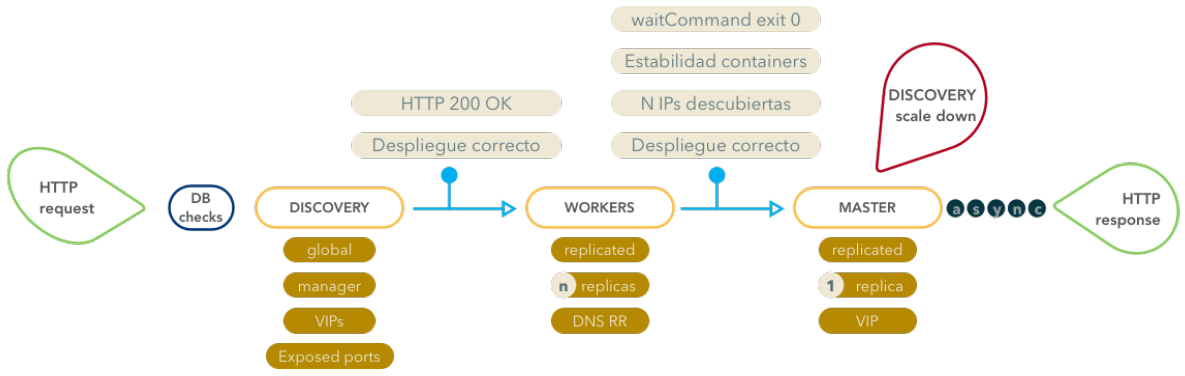

Figura 5.4: Diagrama del proceso de orquestación de tareas

La documentación de la implementación del proyecto se puede consultar en *GoDoc*[12]

[12]https://godoc.org/github.com/mtenrero/ATQ-Director/dockerMiddleware

5.3.4. Almacenamiento de datos

Se ha utilizado por simplicidad el paquete *buntdb*[13] para el almacenamiento de datos del framework. Se basa en un esquema clave/valor, siendo compatible con otras soluciones de almacenamiento de datos distribuidos similares, como etcd[14].

Todas las operaciones relacionadas con el almacenamiento de datos se encuentran en el paquete **Persistence**. En el lanzamiento del framework, durante las primeras instrucciones, se realiza una llamada al método *InitPersistence()*, el cual devuelve un struct del tipo **Persistence* que contiene la instancia inicializada de *BuntDB* y ésta es inyectada sobre los controladores que hacen uso de ella, como es el caso del controlador **Task**.

Debido a la naturaleza del lenguaje *Go* y la implementación del paquete *BuntDB*, se ha creado un wrapper para el almacenamiento y la obtención de datos para gestionar las llamadas de una manera más simple y comprensible.

Se encuentra en el fichero *wrapper.go* y ofrece los métodos

- **store(key, value string) error** Almacena una nueva entrada.
- **read(key string) (string, error)** Obtiene el valor de una clave dada.
- **delete(key string) error** Elimina una entrada del almacenamiento de datos.
- **iterateStringString(index string) (*map[string]string, error)** Devuelve una colección de entradas dado un patrón de indexado existente.

Para conseguir un funcionamiento similar a objetos con múltiples propiedades, *BuntDB* ofrece una solución abstracta, la definición de índices mediante wildcards.

Por ello, se ha configurado un índice para permitir obtener múltiples propiedades indicando un identificador de tarea, definiendo el patrón de almacenamiento de datos *task:*:*

[13] https://github.com/tidwall/buntdb

[14] https://coreos.com/etcd/

El uso de un almacenamiento de datos es necesario para almacenar el estado actual de la tarea y los identificadores de los directorios bindados a cada servicio. Como restricción se ha implementado un control de duplicidades, para asegurarse de que no se desplieguen dos tareas con el mismo identificador.

La documentación del paquete se puede encontrar fácilmente en *GoDoc*[15]

5.3.5. Imágenes Docker

Para todas las herramientas y el middleware desarrollado se proporcionan los ficheros *Dockerfile* para que sea posible construir las imágenes de las mismas y poder así desplegarlas de forma nativa en Docker.

No obstante, se han enlazado los repositorios de GitHub con Docker Hub mediante la configuración de webhooks, para que con cada commit o Pull-Request se construyan las imágenes de forma automática y sean publicadas a Docker Hub y esté disponible para la comunidad la versión más reciente de la utilidad en el registro público de manera completamente automatizada.

Por otra parte, para el middleware, también se proporciona un fichero *docker-compose* (Apéndice B.1 y B.2) para que pueda ser desplegado en una infraestructura existente de Docker Swarm como un nuevo Stack.

5.3.6. Service Discovery

Para el descubrimiento de servicios, como se ha avanzado en algún párrafo anterior, se ha desarrollado una mínima aplicación *Go* que expone una API REST sencilla. Esta aplicación aprovecha el enrutado interno de los contenedores de los servicios docker que estén configurados con el algoritmo DNS RR.

Una vez que se ha realizado la consulta al servidor de nombres, se parsea la información y

[15] https://godoc.org/github.com/mtenrero/ATQ-Director/persistance

se devuelve en formato *CSV*, es decir, en valores separados por comas, ya que es el formato que JMeter requiere para especificar los nodos remotos y es muy fácilmente iterable por cualquier script para así poder emplear los datos con futuras herramientas.

El descubrimiento de servicios también es empleado en el algoritmo interno de orquestación del middleware, ya que éste para asegurarse de que los contenedores han sido correctamente inicializados, realiza peticiones asíncronamente con una espera activa hasta que se recibe la cantidad deseada de contenedores a ser desplegados. Y una vez que se puede acceder a dichos contenedores, si el usuario lo ha especificado, se procede a ejecutar el comando que se asegura de la disponibilidad de los contenedores desplegados a través de peticiones tcp o udp.

5.4. Integración Continua

Inicialmente, el proyecto, al estar alojado en el servicio de control de versiones de GitHub, en un repositorio de código abierto, se ofrecían diversas opciones SAAS de Integración Continua, entre ellas TravisCI o CircleCI.

Al comenzar el desarrollo, las características que ofrecía TravisCI encajaban perfectamente con los requisitos de Automation Test Queue:

- Pipeline-as-code
- Enfocado a Contenedores
- Integración con GitHub

Se utilizó durante la mitad de desarrollo del proyecto y ofreció muy buenos resultados. Pero llegó un momento en el que era necesario que el host donde se ejecutaban los tests cada vez que realizaba un commit formase parte de un cluster de Docker Swarm, y esto no se podía conseguir con la solución que ofrecía TravisCI.

La limitación de TravisCI era muy importante ya que rompía la línea de Integración Continua, por lo que no quedó mas remedio que optar por una solución mas configurable, Jenkins.

No se optó desde un principio por Jenkins debido a la necesidad de disponer de una máquina con acceso a la red las 24 horas al día y la configuración del mismo.

El despliegue de las máquinas destinadas a la línea de integración continua se realizó sobre máquinas EC2 de Amazon Web Services, todas ellas bajo el mismo grupo de disponibilidad y con un servicio de IP elástica, permitiendo de esta manera que la dirección de acceso a Jenkins fuese estática. Además se incluyó un registro A en el DNS de mi dominio particular para poder acceder con una dirección[16] mas fácil de recordar aún.

Una parte vital para el correcto funcionamiento de la línea de integración continua fue configurar webhooks en GitHub para apuntar a la instancia de Jenkins además de habilitar la integración propia de Jenkins. Esto permitió que con cada evento en el repositorio, como *commits* o *pull-requests* se enviase una notificación a Jenkins para así poder ejecutar el job especificado y lanzar los tests unitarios sobre todas las ramas del proyecto para asegurar la regresión y un buen funcionamiento del mismo.

Hace apenas dos años, Cloudbees, la organización que es oficialmente responsable del desarrollo de Jenkins, lanzó una característica que llamaron *Jenkins Declarative Pipelines* [17]. Estos pipelines, permiten escribir, de una forma desriptiva, las tareas y procesos a realizar cada vez que el job es ejecutado.

Esto ha permitido describir el workflow de ejecución de manera totalmente agnóstica y compatible con cualquier Jenkins desplegado, siempre que tenga Docker instalado, ya que hace uso del denominado Docker-in-Docker, que permite utilizar el demonio de Docker del host dentro de un contenedor ya existente.

5.5. Pruebas

Desde el comienzo del proyecto, la parte de las pruebas sobre la aplicación ha sido muy importante debido a la aplicación de las metodologías ágiles y TDD, ademas del uso de la librería Goa con la que se ha realizado la interfaz REST del Middleware.

[16] http://atq.mtenrero.com:8080

[17] https://jenkins.io/blog/2017/02/03/declarative-pipeline-ga/

Se ha organizado el proyecto siguiendo el estándar de *Go*, es decir, dentro de cada paquete se han alojado sus tests unitarios. Los tests se diferencian del código en el nombre de fichero, ya que por convenio, los ficheros test en Go deben finalizar con la palabra *_test.go*.

Es común encontrar diferentes ficheros de test en cada paquete ya que se ha seguido un estilo de distribución de ficheros por funcionalidad para así ofrecer un mayor facilidad para comprender el código.

Tomando como ejemplo el paquete ***persistence*** se han realizado un total de seis ficheros de test:

- **databind_test.go** Contiene los tests relacionados con las operaciones sobre la entidad *databind*, encargada de las operaciones con los ficheros que proporciona el usuario para sus tests.
- **filesystem_test.go** Contiene los tests de almacenamiento de la base de datos clave/valor sobre el sistema de ficheros del sistema.
- **persistence_test.go** Comprueba la escritura y la lectura básica utilizando la implementación de almacenamiento.
- **task_test.go** Contiene los tests sobre todas las posibles operaciones relacionadas con la entidad Task del Middleware.
- **timestamper_test.go** Comprueba el funcionamiento correcto del generador de UUIDs basado en Timestamp.
- **wrapper_test.go** Tests relacionados sobre el wrapper que encapsula y simplifica el uso de la librería BuntDB.

La mayoría de los ficheros anteriormente indicados, se corresponden conceptualmente con un fichero con el mismo nombre sin la terminación _test. Esos ficheros contienen la implementación de la funcionalidad descrita.

De esta manera, se ofrece a un futuro desarrollador, además de unas pruebas que aseguren una regresión, un ejemplo de cómo usar las implementaciones ofrecidas, ya que se han realizado pruebas para prácticamente la totalidad de la implementación realizada salvo una mínima excepción.

5.5.1. Tecnologías utilizadas

Para realizar un seguimiento sobre la cobertura de tests se ha utilizado la plataforma Coveralls.io[18]. Permite visualizar gráficamente la evolución de la cobertura entre los diferentes commits realizados al control de versiones.

Para el correcto uso de esta plataforma ha sido necesario apoyarse en la librería *Goveralls*[19]. Esta librería envía los resultados de las pruebas a la plataforma Coveralls.io. Durante el cambio a Jenkins como sistema de Integración Continua, Goveralls no ofrecía soporte completo para este sistema de CI y fue necesario introducir algunos cambios en el repositorio[20] mediante un Pull Request[21] para mejorar la compatibilidad con Jenkins.

Se ha utilizado el paquete de testing nativo de *Go* y el paquete *Testify*[22] que proporciona una serie de funciones y utilidades para simplificar el proceso de testeo, como comparaciones, expected conditions, etc...

5.5.2. Cobertura de tests

En un primer lugar se realizaron tests básicos sobre los casos de uso necesarios y posteriormente, estos tests se fueron ampliando para disponer de una regresión completa en la mayoría de paquetes.

[18] https://coveralls.io/github/mtenrero/ATQ-Director
[19] https://github.com/mattn/goveralls
[20] https://github.com/mtenrero/goveralls
[21] https://github.com/mattn/goveralls/pull/116
[22] github.com/stretchr/testify/assert

Docker Middleware

El paquete dockerMiddleware dispone de una cobertura de test del 42,02 %, y esto es debido a que las pruebas a realizar sobre este paquete en realidad pertenecen mas a pruebas de integración ya que requieren comunicarse con el demonio de Docker del sistema y requieren una serie de pre-comprobaciones. No obstante, se han implementado la mayor parte de pruebas posibles, sobre todo para las operaciones más básicas como crear y eliminar redes, contenedores y volúmenes.

Cargador de configuraciones

El paquete configLoader dispone de una cobertura de tests del 100 %, ya que es el encargado de la lectura y propagación de configuraciones desde un fichero YAML.

Persistence

El paquete de gestión de la persistencia de datos dispone de una cobertura de test del 87,56 %, cubriendo casi la totalidad de la funcionalidad. Quedando fuera de la regresión ciertos casos muy difíciles de conseguir al estar relacionados propiamente con la librería, pero como se tienen en cuenta en los test el valor esperado por la llamada a dichas funciones, se puede afirmar casi con total seguridad que se podría detectar prácticamente cualquier fallo.

Capítulo 6

Conclusiones y trabajos futuros

La utilización del middleware ATQ reduce drásticamente el tiempo de setup de la infraestructura necesaria para ejecutar tests de manera elástica. Ya que con unos sencillos comandos es posible desplegarla de una manera rápida, transparente y eficiente.

En la siguiente sección se describe un posible uso real del middleware junto con algunas herramientas de apoyo que sirven de ayuda para conocer la salud de una aplicación web basándose en la cantidad de conexiones concurrentes a la misma. Y es aquí donde ATQ toma protagonismo. Porque cuando se tiene una determinada carga de trabajo en una aplicación en producción con un gran número de usuarios, es complicado conseguir reproducir o superar esta carga de trabajo en un entorno de pruebas, debido a los requisitos de hardware necesario.

Debido a las especificaciones técnicas de Docker Swarm, con una limitada configuración de las políticas de seguridad de una nube pública como Amazon Web Services o Google Cloud Platform se podría conformar un clúster híbrido, público o privado bajo demanda para adaptarse realmente al crecimiento de una aplicación en producción. Ya que el despliegue del middleware ATQ es realmente sencillo tanto en Docker Swarm por medio de Stacks como en Kubernetes, gracias a las funciones experimentales de compatibilidad con ficheros *docker-compose*, nativos de Docker Swarm.

6.1. Resultados en tiempo real de lanzamiento con ATQ

Para visualizar de forma más clara los resultados de los tests de carga, se puede configurar un *Backend Listener* en el *Test Plan* de Apache JMeter para que retransmita los resultados a un sistema externo como *ElasticSearch* o *InfluxDB*[1].

Yo he decidido emplear ElasticSearch, y para ello, hay que añadir un plugin[2] de JMeter al directorio *./lib/ext* de la instalación de JMeter. (La imagen de JMeter creada para el uso con ATQ ya lo incluye)

Además, también es necesario configurar una serie de parámetros, como la url donde se expone la API de ElasticSearch o el índice donde guardar los resultados.

Una vez que se ha configurado correctamente el *Backend Listener* y se ejecuta el test, es posible consultar directamente a ElasticSearch por el estado del nuevo índice especificado

```
>HTTP GET elasticsearch:9200/test
```

O también se pueden consultar los contenidos volcados por JMeter sobre el índice:

```
>HTTP GET elasticsearch:9200/test/_search
```

Lo cual devolverá todos los registros de los que disponga en ese índice en formato JSON. (Apéndice B.3)

Para visualizarlo de una manera más cómoda, se puede conectar con un panel visualizador como *Grafana*[3] o *Kibana*[4]. Para los ejemplos, he utilizado Grafana conectado a ElasticSearch:

6.2. Ampliación del Middleware

[1] https://www.influxdata.com
[2] https://github.com/delirius325/jmeter-elasticsearch-backend-listener
[3] https://grafana.com
[4] https://www.elastic.co/products/kibana

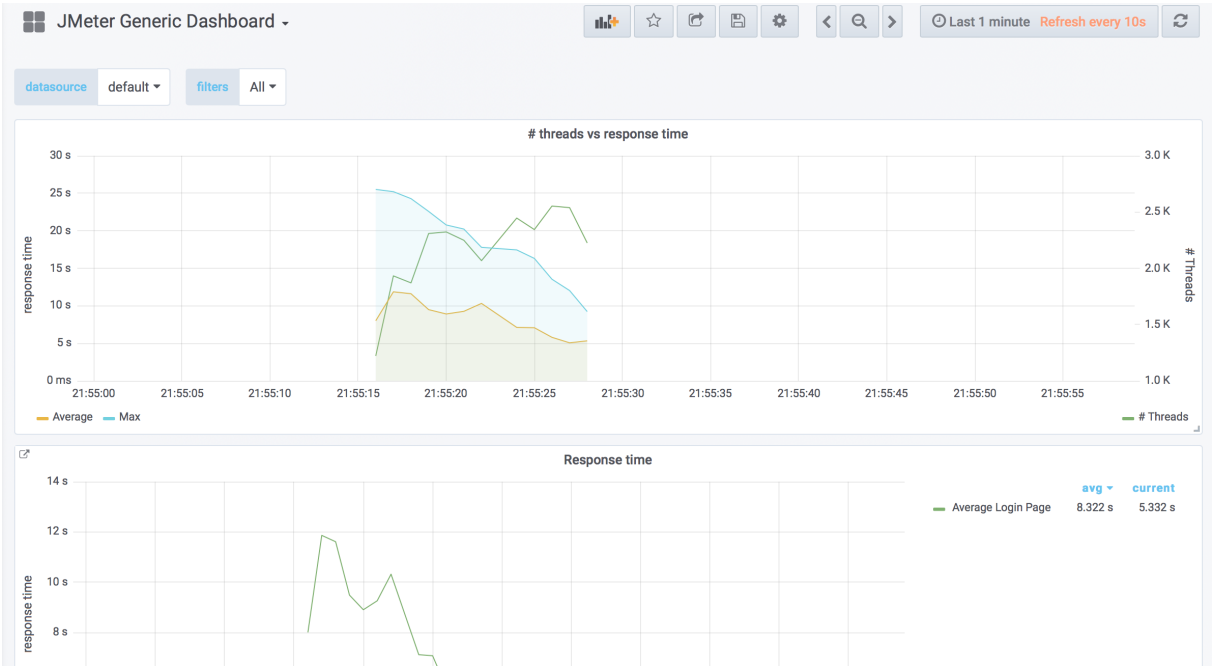

Figura 6.1: Snapshot Grafana Elastic JMeter 1

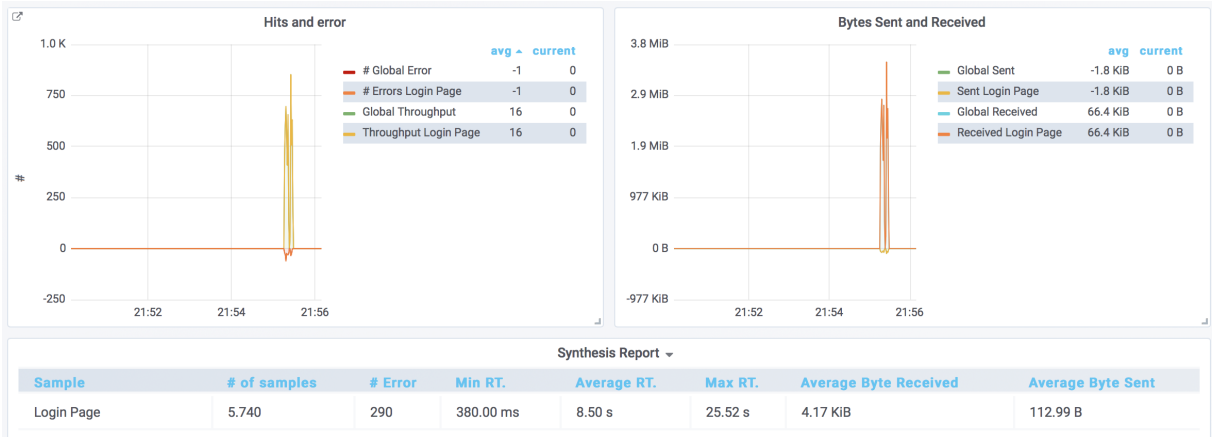

Figura 6.2: Snapshot Grafana Elastic JMeter 2

6.2.1. Sistemas Operativos Windows

La integración completa con Sistemas Operativos Windows aún no es posible, ya que las imágenes de ATQ para un despliegue basado en stacks de Swarm y del descubrimiento de servicios, están basados en imágenes linux, como CentOS 7[5] o Alpine Linux[6]. Esto implica que no se pueda desplegar ATQ en sistemas Windows que utilicen contenedores Windows nativos, aunque sí es posible desplegar ATQ sobre sistemas Windows que operen con contenedores Linux.

Para una integración completa, sería necesario crear imágenes específicas con los binarios de ATQ y del descubridor de servicios con una imagen de contenedor basada en Windows como *nanoserver*[7].

6.2.2. Integración con Jenkins

Por otro lado, se podría ampliar el proyecto creando un plugin específico para Jenkins para poder así definir de una manera declarativa sencilla los parámetros para una ejecución elástica de test de carga usando el middleware ATQ.

6.2.3. Sistema de ficheros distribuidos interno

En la actualidad, es necesario que exista un sistema de compartición de ficheros subyacente como Samba o GlusterFS para que los ficheros de test proporcionados por el usuario estén disponibles en todos los nodos que conforman el clúster de Docker Swarm.

Sería interesante extender el middleware para que expongan un socket en cada nodo y poder replicar el fichero proporcionado del usuario a través de éstos sockets antes de enviarle al usuario la respuesta a su petición.

De esta manera se reducirían prácticamente a cero los requisitos para poder utilizar Automation Test Queue y se prescindiría del sistema externo de ficheros distribuidos.

[5] https://centos.org

[6] https://alpinelinux.org

[7] https://hub.docker.com/r/microsoft/nanoserver/

6.2.4. Solución comercial

Una solución empresarial sería posible creando un portal web con una serie de características como autenticación con Kerberos o LDAP y multi-tenancy de usuarios para poder así ofrecer una solución SAAS de la herramienta y ofrecerla a través de algún tipo de suscripción.

Para ello sería necesario desplegar ATQ sobre una infraestructura cloud como Amazon Web Services[8], Azure[9] o Google Cloud Platform[10] usando el orquestador Kubernetes[11] con el objetivo de poder escalar de una manera sencilla la infraestructura en base a la demanda de los usuarios.

[8] https://aws.amazon.com
[9] https://azure.microsoft.com
[10] https://cloud.google.com
[11] https://kubernetes.io

Una posible arquitectura sería la siguiente:

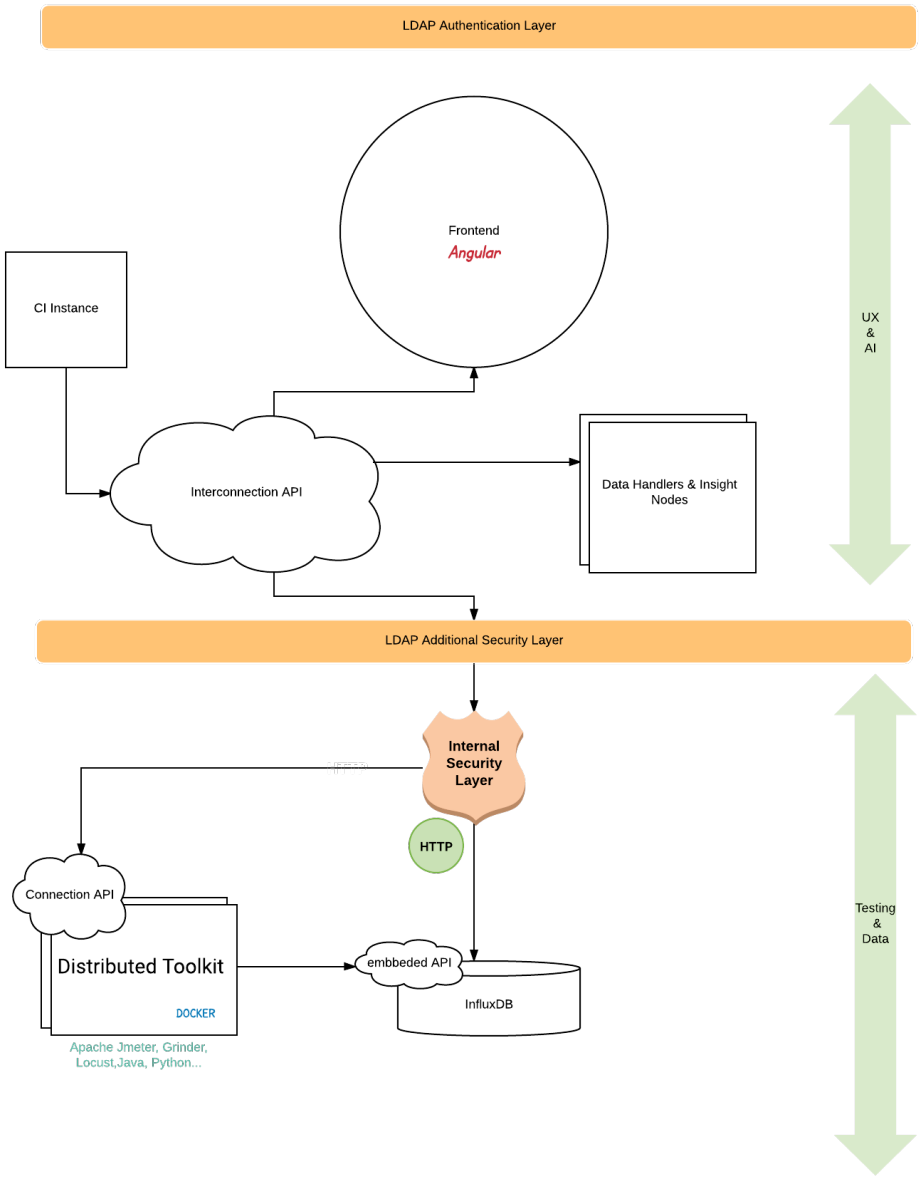

Figura 6.3: Arquitectura posible para una solución comercial SAAS

Bibliografía

[1] A. Beloglazov, S. F. Piraghaj, M. Alrokayan, and R. Buyya. Deploying openstack on centos using the kvm hypervisor and glusterfs distributed file system. *University of Melbourne*, 2012.

[2] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

[3] M. S. Divya and S. K. Goyal. Elasticsearch: An advanced and quick search technique to handle voluminous data. *Compusoft*, 2(6):171, 2013.

[4] A. A. Donovan and B. W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[5] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614, March 2014.

[6] H. Howard. ARC: Analysis of Raft Consensus. Technical Report UCAM-CL-TR-857, University of Cambridge, Computer Laboratory, July 2014.

[7] Z. Huili. Realization of files sharing between linux and windows based on samba. In *2008 International Seminar on Future BioMedical Information Engineering*, pages 418–420, Dec 2008.

[8] Á. Kovács. Comparison of different linux containers. In *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, pages 47–51, July 2017.

[9] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[10] R. Pike. The go programming language. *Talk given at Google's Tech Talks*, 2009.

[11] R. Pike. Go at google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 5–6, New York, NY, USA, 2012. ACM.

Siglas

API Interfaz de Programación de Aplicaciones. 9, 11, 14, 30, 31

ATQ Automation Test Queue. 41, 42, 44, 45

CI Continuous Integration. 39

CSV Valores separados por comas. 36

DNS Domain Name System. 15, 23, 31, 32, 37

DSL Lenguaje de Dominio Específico. 14

E2E End to End. 1

EC2 Instancia virtual de Amazon Web Services. 37

ELK Elastic + Logstash + Kibana. 18

HTTP Hypertext Transfer Protocol. 31

LDAP Lightweight Directory Access Protocol. 45

PAAS Platform As A Service. 19

REST Representational State Transfer. 7, 9, 31, 37

SAAS Software As A Service. 45

SDK Software Development Kit. 24

SSL Secure Sockets Layer. 5

TDD TestDrivenDevelopment. 9, 37

TLS Transport Layer Security. 5

UUID Identificador único. 38

Apéndice A

Manual de usuario

A.1. Despliegue de ATQ

A.1.1. Binarios

Ejecutar el binario apropiado para la plataforma:

Windows

```
> ./atq-director-win_64.exe
```

Unix

```
$ chmod +x atq-director-linux_64
$ ./atq-director-linux_64
```

A.1.2. Docker

Swarm Stack

Ejecutar el siguiente comando en un directorio que contenga el fichero *docker-compose.yml* del Apéndice B.2

```
$ docker-compose up
```

En caso de necesitar desplegarlo como servicio distribuido, usar el comando:

```
$ docker stack deploy --compose-file docker-compose-yml atq
```

A.2. Configuración de arranque

Durante el arranque del middleware ATQ, busca el fichero *controller-config.yml* en el que se especifican una serie de configuraciones críticas para el middleware.

Es necesario que esté especificada la clave **port**, que define el puerto por el cual exponer la API REST y la clave **glusterpath**, que corresponde a la ruta del directorio compartido entre todos los nodos que conforman el cluster de Docker Swarm donde desplegar el middleware.

En caso de no existir dicho fichero, el middleware **no arrancará**

A.3. Subida de ficheros de test

Los ficheros de test han de subirse **antes** de lanzar una tarea sobre el middleware.

La API REST está diseñada para que únicamente acepte peticiones **POST** con *multipart/-form* en *Request Header* que incluyan un parámetro *file* de tipo *File* y éste incluya un fichero con la extensión *ZIP*.

En caso de no cumplir alguno de los requisitos anteriores, se devolverá un error de acuerdo a la especificación de la API que se puede consultar en los ficheros OpenAPI de Swagger.

Una petición de ejemplo tendría este aspecto:

```
POST /api/databind/upload HTTP/1.1
Host: localhost:8080
Accept: application/atq.databind.upload+json
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWx
```

```
Cache-Control: no-cache
Postman-Token: c1a523b2-6a04-486c-b133-e8db47d6b3d1

----------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="file"; filename="filename.zip"
Content-Type:


----------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

A.4. Creación de tarea de Test vía API

Una vez que se dispone de un fichero de test correctamente subido al middleware y se ha anotado su identificador, se puede proceder a crear una tarea de test.

Body

```
{
  "delay": 2,
  "master": {
    "alias": "Coordinator",
    "image": "tenrero/jmeterdistributed",
    "fileid": "1530553166",
    "environment":[
                "MODE=master",
                "TEST_PATH=/atq/data/abvetmadrid.jmx"
        ],
    "replicas": 1,
    "tty": true
  },
  "name": "jm11",
  "waitCommand": {
```

```
    "command": "sleep 4",
    "expectedResult": "",
    "timeout": 2
  },
  "worker": {
    "alias": "Minions",
    "image": "tenrero/jmeterdistributed",
    "replicas": 3,
    "environment":[
                "MODE=node"
        ],
    "tty": true
  }
}
```

Apéndice B

Imágenes Docker

Listing B.1: ATQ docker-compose.yaml

```
version: 3.3

services:
    atq:
        image: tenrero/atq
        tty: true
        volumes:
            - gluster:/gluster:rw
            - /var/run/docker.sock:/var/run/docker.sock
        ports:
            - "8080:8080"
        deployment:
            mode: global
            placement:
                constraints: [node.role == manager]
volumes:
    gluster:
```

Listing B.2: ATQ Dockerfile

```
FROM centos

LABEL MAINTAINER="Marcos Tenrero"

COPY ./releases/atq-director-linux-amd64 /atq/atq-amd64
COPY controller-config.docker.yaml /controller-config.yaml

RUN mkdir -p /gluster/storage

EXPOSE 8080

RUN cd /atq

ENTRYPOINT [ "/atq/atq-amd64" ]
```

Listing B.3: ElasticSearch JMeter extract


```
{
        "_index":"test",
        "_type":"SampleResult",
        "_id":"OSuhZmQBgvemDY44wMwy",
        "_score":1.0,
        "_source": {
        "EndTimeInMs":1530730232652,
        "ContentType":"text/html; charset\u003dutf-8",
        "EndTime":"2018-07-04T20:50:32.652+0200",
        "IdleTime":0,"ElapsedTime":"Jan 4, 2018 12:00:02 AM",
        "ErrorCount":0,
        "Timestamp":"2018-07-04T20:50:32.076+0200",
        "URL":"http://abvetmadrid.com/",
        "Bytes":4371,
        "SentBytes":119,
```

```
"ElapsedTimeInMs":638,
"AllThreads":337,
"DataType":"text",
"ResponseTime":576,
"SampleCount":1,
"ConnectTime":446,
"StartTimeInMs":1530730232076,
"ResponseCode":"200",
"StartTime":"2018-07-04T20:50:32.076+0200",
"AssertionResults":[],
"Latency":575,
"GrpThreads":336,
"BuildNumber":0,
"BodySize":3982,
"ThreadName":"Grupo de Hilos 1-12","SampleLabel":"Login
   Page"}}
```

Apéndice C

Documentación Swagger OpenAPI

Warning Open

ATQ Test Orchestration Director

REST Interface for Test queuing and orchestration

Version

Paths

/databind/list

GET /databind/list

databind

Summary

list databind

Description

List of uploaded and available files

Responses

Code	Description	Schema
200	OK	⇄ ▼**Mediatype identifier: application/atq.databind.upload+json; type=collection; view=default[** *AtqDatabindUploadCollection is the media type for an array of AtqDatabindUpload (default view)* ▶**Mediatype identifier: application/atq.databind.upload+json; view=default {** **}**]
204	No Content	

Try this operation

/databind/upload

POST /databind/upload

databind

Summary

upload databind

Description

Upload new zipped file for later usage with a Task

Parameters

Name	Located in	Required	Schema
payload	body	Yes	⇄ ▼ **UploadPayload {** file: ▶ undefined * }

Responses

Code	Description	Schema
200	The file was uploaded succesfully	▼ **Mediatype identifier: application/atq.databind.upload+json; view=default {** ⇄ *User upload files response (default view)* id: ▶ string }
415	The file doesn't have a valid extension	▼ **Mediatype identifier: application/atq.databind.upload+json; view=error {** ⇄ *User upload files response (error view)* error: ▶ string }
500	Response when there are an error uploading the file	▼ **Mediatype identifier: application/atq.databind.upload+json; view=error {** ⇄ *User upload files response (error view)* error: ▶ string }

Try this operation

/monitoring/ping

GET /monitoring/ping

monitoring

Summary

ping monitoring

Description

Endpoint for pinging and healthcheck purposes

Responses

Code	Description
200	Pong

Try this operation

/swarm/

GET /swarm/

swarm

Summary

status swarm

Description

Response with the details of the swarm

Responses

Code	Description	Schema
200	Details of the Docker Swarm cluster	▼ **Mediatype identifier: application/atq.swarm+json; view=default** { ⇄ *Swarm Details (default view)* joinTokens: ▶ **JoinTokens { }** }
503	Docker Swarm context Error Message	▼ **Mediatype identifier: application/atq.swarm+json; view=error** { ⇄ *Swarm Details (error view)* error: ▶ string }

Try this operation

/task/

PUT /task/

task

Summary

create task

Description

Creates a new Task in the Swarm according with the config provided in the JSON body

Parameters

Name	Located in	Required	Schema
payload	body	Yes	⇄ ▼ **TaskPayload** { delay: integer, master: ▶ **ServicePayload** { }, name: ▶ string *, waitCommand: ▶ **WaitCommand** { }, worker: ▶ **ServicePayload** { } }

Responses

Code	Description	Schema
200	Task creation in progress	⇄ ▼ **Mediatype identifier: application/atq.task+json; view=default** { *Task description (default view)* id: ▶ string, status: ▶ string }
417	The Task definition has errors or it's not complete	⇄ ▼ **Mediatype identifier: application/atq.task+json; view=default** { *Task description (default view)* id: ▶ string, status: ▶ string }

Try this operation

/task/{id}

DELETE /task/{id}

task

Summary

delete task

Description

Deletes the Task specified and its components

Parameters

Name	Located in	Required	Schema
id	path	Yes	⇄ string

Responses

Code	Description
204	Successfuly deleted
404	The given ID doesn't not exist
500	Docker Engine error deleting the Task generated container infrastructure

Try this operation

GET /task/{id}

task

Summary

inspect task

Description

Get Task's details

Parameters

Name	Located in	Description	Required	Schema
id	path	Task's UUID	Yes	⇄ string

Responses

Code	Description	Schema
200	Successful response containing Task data in JSON format	▼ **Mediatype identifier: application/atq.task+json; view=default** { ⇄ *Task description (default view)* id: ▶ string status: ▶ string }
404	The given ID doesn't not exist	
500	Response when the Task has not been created correctly	▼ **Mediatype identifier: application/atq.task+json; view=default** { ⇄ *Task description (default view)* id: ▶ string status: ▶ string }

Try this operation

Models

AtqDatabindUpload

▼ **Mediatype identifier: application/atq.databind.upload+json; view=default**
{
⇄ *User upload files response (default view)*
id: ▶ string
}

AtqDatabindUploadCollection

▼ **Mediatype identifier: application/atq.databind.upload+json; type=collection; view=default[**

⇄ *AtqDatabindUploadCollection is the media type for an array of AtqDatabindUpload (default view)*

▶ **Mediatype identifier: application/atq.databind.upload+json; view=default { }**

]

AtqDatabindUploadError

▼ **Mediatype identifier: application/atq.databind.upload+json; view=error {**

⇄ *User upload files response (error view)*

error: ▶ string

}

AtqSwarm

▼ **Mediatype identifier: application/atq.swarm+json; view=default {**

⇄ *Swarm Details (default view)*

joinTokens: ▶ **JoinTokens { }**

}

AtqSwarmError

▼ **Mediatype identifier: application/atq.swarm+json; view=error {**

⇄ *Swarm Details (error view)*

error: ▶ string

}

AtqTask

▼ **Mediatype identifier: application/atq.task+json; view=default {**

⇄ *Task description (default view)*

id: ▶ string

status: ▶ string

}

JoinTokens

▼ **JoinTokens {**

⇄ *Docker Swarm Join Tokens*

manager: string

worker: string

}

ServicePayload

▼ **ServicePayload {**

alias: ▶ string *

args: ▶ **[]**

⇄ fileid: ▶ string

image: ▶ string *

```
    replicas: ▶ integer
    tty:      ▶ boolean
}
```

TaskPayload

```
▼ TaskPayload {
    delay:       integer
    master:      ▶ ServicePayload { }
⇄   name:        ▶ string *
    waitCommand: ▶ WaitCommand { }
    worker:      ▶ ServicePayload { }
}
```

UploadPayload

```
▼ UploadPayload {
⇄   file: ▶ undefined *
}
```

WaitCommand

```
▼ WaitCommand {
  Definition of a command to be executed
    command:        ▶ string
⇄
    expectedResult: ▶ string
    timeout:        ▶ integer
}
```