UNIVERSIDAD
REY JUAN CARLOS

GRADO EN INGENIERÍA DE COMPUTADORES

Curso Académico 2017/2018

Trabajo Fin de Grado

TÍTULO DEL TRABAJO EN MAYÚSCULAS

Autor : Marcos Tenrero Morán

Tutor : Francisco Gortázar

Trabajo Fin de Grado

Título del Trabajo con Letras Capitales para Sustantivos y Adjetivos

Autor : Marcos Tenrero Morán

Tutor : Francisco Gortázar

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

Dedicado a

mi familia / mi abuelo / mi abuela

II

Agradecimientos

Resumen

Aquí viene un resumen del proyecto. Ha de constar de tres o cuatro párrafos, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es su objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el proyecto? ¿Es un proyecto dentro de un marco general?

Lo mejor es escribir el resumen al final.

Índice general

1. Introducción y Motivación **1**

2. Objetivos **5**

3. Metodologías **7**

3.1. Extreme Programming 7

3.2. Kanban 8

4. Tecnologías y Herramientas **9**

4.1. Golang 9

4.2. Dep: Go Dependency Management 9

4.3. Goa Design: Design-first Network framework 10

4.4. OpenAPI Specification 10

4.5. DNS 11

4.5.1. DNS Round-Robin 11

4.6. Contenedores 11

4.6.1. LXC: Linux Containers 11

4.6.2. Docker 12

4.7. Apache JMeter 13

4.8. ElasticSearch 14

4.9. Sistema de ficheros distribuido 14

4.9.1. Samba 14

4.9.2. GlusterFS 14

4.10. AWS: Amazon Web Services 15

4.11. Sistemas de Integración Continua 15

4.11.1. TravisCI 15
4.11.2. Jenkins 16

5. Descripción informática **17**
5.1. Requisitos 17
5.2. Arquitectura y Análisis 17
5.2.1. Primer enfoque: Starter en cada imagen a desplegar 18
5.2.2. Segundo enfoque: Single Daemon y DNS Round Robin 18
5.2.3. Estructura del proyecto 19
5.3. Diseño e Implementación 20
5.3.1. API REST y modelo de datos 20
5.3.2. Docker Middleware 22
5.3.3. Fase de orquestación 24
5.3.4. Imágenes Docker 24
5.3.5. Service Discovery 24
5.4. Integración Continua 24
5.5. Pruebas 25

6. Conclusiones y trabajos futuros **27**

Bibliografía **29**

A. Manual de usuario **31**

B. Fragmentos de código **33**

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 el testeo y comparación del 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 que ofrece 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 considerable carga de trabajo.

No solo 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 la realización de una herramienta de ayuda en el proceso del análisis de rendimiento de cualquier aplicación. Pretende ofrecer una manera sencilla de ejecutar estos test de manera automática sin tener que preocuparse de la infraestructura subyacente necesaria.

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 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 poder ejecutar cualquier tipo de aplicación que requiera este modelo de despliegue.

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 aplicando 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.

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, lo que obliga a una configuración manual de éstos y 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. 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
- Orquestación de contenedores, redes y volúmenes de forma transparente
- Carácter efímero de la infraestructura

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

- Construir un Middleware que comunique con la 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
- 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 integrando 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*.

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 salido 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 raiz 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.

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.

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.

[1] https://github.com/golang/dep
[2] https://goa.design
[3] https://github.com/OAI/OpenAPI-Specification

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/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 IP, realizando, de esta manera, 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 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 los contenedores 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 el sistema de ficheros a LXC. Permite especificar imágenes de contenedores autocontenidas y gestionar el ciclo de vida de ellos. [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.
- **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] .

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

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 especique 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. 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.

4.9.1. 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].

4.9.2. GlusterFS

GlusterFS[5] 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].

[5] http://gluster.org

4.10. AWS: 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.

4.11. 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 asegurarase de la regresión de la aplicación, es decir, que no se ha roto ninguna funcionalidad previo con los nuevos cambios introducidos.

4.11.1. TravisCI

TravisCI[6] es un servicio de Integración Continua ofrecido de manera gratuita para proyectos open-source de GitHub. Está preconfigurado 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[7]. 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, únicamante para construir imágenes y publicarlas en un registro.

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

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

4.11.2. 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.

Capítulo 5

Descripción informática

5.1. Requisitos

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
- Networking automático y transparente
- Interfaz sencilla para el usuario
- Integración completa con Apache JMeter

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

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

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 imagen a desplegar

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

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

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

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

los contenedores de un mismo servicio.

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.

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
persistance 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 raiz 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 **persistance** 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.

5.3. 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[3].

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.

[3] 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 despliegues 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. Existen una variable reservadas 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
- **Tty** Parámetro que fuerta la consola interactiva en el servicio.

Endpoint API

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 tuvo en cuenta todos los tipos de respuesta que puede 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 uno 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[4].

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**

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 estos paquetes.

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

5.3.3. Fase de orquestación

5.3.4. Imágenes Docker

5.3.5. Service Discovery

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[5] 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* [6]. 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

Hablar de Cobertura de Test, como se ha organizado por paquetes, cómo está integrado con el Pipeline de IC de Jenkins, los triggers que tiene configurado...

Explicar cómo se diseñaron primero los tests básicos, luego la implementación, y después se amplió la cobertura de tests

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

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

Capítulo 6

Conclusiones y trabajos futuros

Me gustaría hablar de la integración completa con OS Windows, otros tipos de despliegue como en AWS de manera nativa, otros modelos diferentes a la arquitectura Master/Workers...

Quizás de una interfaz web completa con Angular o similar...

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.

Apéndice A

Manual de usuario

Apéndiceb

Apéndice B

Fragmentos de código

Listing B.1: Jenkisfile CI Docker-in-Docker

```
pipeline {

    agent {
        docker {
            image 'tenrero/golang-dep-alpine:1.10.2'
            reuseNode true
            args '-it -v /var/run/docker.sock:/var/run/docker.sock
               -v $WORKSPACE:/tmp/app'
        }
    }

    stages {
        stage('Prepare Environment') {
            steps {
                sh 'echo $GOPATH'
                sh 'mkdir -p /go/src/github.com/mtenrero/'
                sh 'ln -s /tmp/app
                   /go/src/github.com/mtenrero/ATQ-Director'
                sh 'ls -a /go/src/github.com/mtenrero/ATQ-Director'
                sh 'go get -u github.com/golang/dep/cmd/dep'
                sh 'go get -u github.com/golang/lint/golint'
```

```
                sh 'go get -u github.com/tebeka/go2xunit'
                sh 'go get -u golang.org/x/tools/cmd/cover'
                sh 'go get -u github.com/mattn/goveralls'
            }
        }

        stage('Download Vendor') {
            steps {
                sh 'cd /go/src/github.com/mtenrero/ATQ-Director &&
                    dep ensure'
            }
        }

        stage('Test') {
            steps {
                sh 'cd /go/src/github.com/mtenrero/ATQ-Director &&
                    go test ./... -race -coverprofile=coverage.txt
                    -covermode=atomic'
            }
        }

        stage('Build') {
            steps {
                sh 'cd /go/src/github.com/mtenrero/ATQ-Director &&
                    ./build.sh'
            }
        }
    }

}
```