Escuela Técnica Superior de Ingeniería Informática

GRADO EN INGENIERÍA DE COMPUTADORES

CURSO ACADÉMICO 2020/2021

TRABAJO DE FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA BASADA EN CONTENEDORES PARA LA AUTOMATIZACIÓN DE LA EVALUACIÓN DE CÓDIGO

Autor: Pablo López Parrilla
Tutores: Jesús Sánchez-Oro Calvo
Isaac Lozano Osorio

Agradecimientos

Agradecer a la universidad por esta etapa de mi vida, a mis compañeros de grado, a mis profesores, a mis compañeros de trabajo del departamento de informática de Alcorcón y finalmente a los tutores que han hecho posible este TFG. Agradecer a mi familia, a mis amigos y a todas las personas que han estado para aguantar todas mis tonterías.

Siglas

HW. Componentes físicos de un sistema informático o en inglés *Hardware*.

SW. Componente lógico de un sistema informático o en inglés *Software*.

CPU. Unidad central de procesamiento o en inglés *Central Processing Unit*.

RAM. Memoria de acceso aleatorio o en inglés *Random Access Memory*.

MV. Máquina Virtual.

API. Interfaz de programación de aplicaciones o en inglés *Application Programming Interfaces*.

BIOS. Sistema básico de entrada y salida o en inglés *Basic Input Output System*.

IP. Protocolo de internet o en inglés *Internet Protocol*.

DNS. Sistemas de nombres de dominio o en inglés *Domain Name System*.

LAN. Red de área local o en inglés *Local Area Network*.

BBDD. Bases de Datos.

CLI. Interfaz de línea de comandos o en inglés *command-line interface*.

TFG. Trabajo de Fin de Grado.

TIC. Tecnología de la información y la comunicación.

CEO. Director ejecutivo o en inglés *Chief Executive Officer*.

SCTP. Protocolo de comunicación de capa de transporte o en inglés *Stream Control Transmission Protocol*.

TCP. Protocolo de control de trasmisión o en ingles *Transmission Control Protocol*.

HTTP. Protocolo de transferencia de hipertexto o en inglés *Hypertext Transfer Protocol*.

PC. Ordenador personal o en inglés *Personal Computer*.

Resumen

La realización de exámenes en los distintos centros educativos ha estado siempre ligada al uso de métodos clásicos como el papel. A día de hoy, con motivo de la modernización de la educación, se empiezan a realizar exámenes de forma digital.

Dentro del sector de TIC, tanto en la enseñanza como mas a posterior en la vida laboral, las pruebas de programación son recurrentes. La forma habitual de evaluar dichos controles es manual, pudiendo demorar en mucho tiempo los resultados. En este proyecto presentamos una forma de evaluar automáticamente códigos, formando un sistema que se podrá utilizar como repositorio de problemas permitiendo diferentes niveles de organización y controlando las soluciones propuestas por cada usuario.

Este Trabajo Trabajo de Fin de Grado(TFG) se centra en el servicio dedicado a la ejecución de código, trabajando sobre el problema de segurización de la ejecución de los códigos. Para ellos se utilizarán distintas tecnologías de virtualización, utilizando finalmente la tecnología de contenedores, específicamente Docker.

Además, se explicarán los objetivos que se pretenden alcanzar, la metodología utilizada para el desarrollo del programa, las diferentes herramientas y tecnologías que se emplean, y la arquitectura y diagramas de flujo que se han diseñado para el correcto desarrollo y escalabilidad del servicio.

Se finalizará la memoria con algunos resultados obtenidos de las diferentes tecnologías de virtualización, con las conclusiones obtenidas a la finalización del proyecto y con la proyección a futuro del juez, así como los trabajos que se están realizando.

Palabras clave

Juez automático, Docker, orquestador de contenedores, RabbitMQ, ejecución automática de código, virtualización, contenedores.

Índice general

Capítulo 1

Introducción

Existen aproximadamente 23.9 millones de personas en el mundo que se dedican profesionalmente a programar [1]. Una gran mayoría de programadores han obtenido dichos conocimientos formándose en escuelas, universidades o cursos, siendo una parte muy importante de la educación la parte práctica. Por norma general, los ejercicios y exámenes se han realizado mayoritariamente sobre papel. En el campo de la informática realizar controles utilizando las técnicas clásicas como son el papel y bolígrafo es cada vez menos común, con tendencia a modificarse y pasar a lo digital.

La realización de exámenes y prácticas de forma digital elimina ciertas desventajas que tenía el método tradicional. Entre ellas encontramos la de la corrección de estos controles, no pudiendo comprobar su funcionalidad de manera sencilla.

Un juez de programación permite validar una posible solución ante un problema propuesto, ya sea con fines de practicar un ejercicio, evaluar unos conocimientos o medir las soluciones, buscando la de mayor eficiencia.

El uso de un juez es aplicable a muchos escenarios como podría ser una empresa que quiere saber qué candidato es más adecuado para el puesto, una asignatura que requiere de numerosos ejercicios prácticos o un torneo de programación. Algunos ejemplos de jueces de programación son los utilizados por compañías internacionales en la búsqueda de los mejores perfiles. Empresas como Amazon o Google utilizan sistemas automatizados de evaluación de código para selección de su personal [2].

Dentro de un sistema de un juez de programación nos encontramos con dos partes claramente diferenciadas:

- El sistema por el cual interactuamos con la interfaz y se procesa toda la información referente a problemas, concursos, además de el análisis de los resultados obtenidos.
- El sistema encargado de la ejecución del código propuesto.

La virtualización es la tecnología por la cual se permite crear un entorno simulado en el cual podremos asignar, de forma flexible, recursos informáticos a la máquina

sobre la que queremos trabajar. Con ello, es posible particionar recursos físicos en recursos virtuales, actuando de manera independiente, pero compartiendo el entorno físico con otras máquinas.

El uso de esta tecnología puede estar enfocado a multitud de escenarios, ya sea el despliegue de aplicaciones con múltiples microservicios [1] independientes, un escenario de pruebas donde poder testear una configuración sin la necesidad de adquirir un Hardware(HW) [2] que cumpla dichos requisitos, o como es en nuestro caso, la necesidad de segurizar la ejecución de un código dentro de un sistema con unos recursos específicos, o la de automatizar el despliegue de entornos previamente construidos con un fin. El concepto de segurizar es utilizado en este proyecto con el fin de definir la necesidad de asegurar la integridad del sistema ante los posibles ataques que se pueden producir desde dentro del propio sistema virtualizado, pudiendo afectar al "host" y resto de entornos virtualizados en este.

Mediante la virtualización podemos independizar un sistema de ficheros del propio sistema de ficheros del HW base, podremos dotar de una cantidad fija de memoria RAM y de tiempo o número de CPUs, así como prescindir de una conexión de red desde dentro del sistema virtualizado, ganando con esto una completa libertad para hacer lo que se quiera dentro de del entorno virtualizado sin afectar al resto de entornos.

En esta memoria nos centraremos en este último, proporcionando con ello una plataforma automática de ejecución de código arbitrario[3] en entornos constantes, independientes y securizados, buscando una manera eficiente, segura y justa de ejecutar un código.

La estructura de la memoria consta de 6 capítulos y 1 guía de instalación. Los capítulos comenzaran con una introducción 1 seguido de los objetivos 2 que se pretenden conseguir con el proyecto. Continuamos con el marco teórico y estado del arte 3, en el que profundizaremos en los diferentes sistemas de virtualización utilizados por jueces de programación, para continuar con la descripción informática 4, donde explicaremos detalladamente el proyecto. Finalizaremos con los resultados 5 y conclusiones 6.

[1] Estilo de arquitectura que busca la división de una aplicación en elementos más simples y pequeños

[2] Conjunto de elementos físicos que constituyen un sistema informático

[3] Código sin modificar, sin filtrar posibles funciones maliciosas

Capítulo 2

Objetivos

El objetivo principal de este TFG es desarrollar una plataforma de ejecución de código independiente, de forma segura, utilizando las tecnologías más ligeras y eficientes, y garantizando el mayor nivel de seguridad. Debido a esta funcionalidad final a la que queremos llegar, podemos definir varios objetivos secundarios que queremos conseguir con este proyecto.

- Obtener conocimientos sobre los diferentes métodos de virtualización.
- Aprender el funcionamiento de los contenedores Docker, su gestión y configuración.
- Aprender el funcionamiento de la librería Docker-Java.
- Aprender el funcionamiento de RabbitMQ.
- Ampliar los conocimientos sobre Spring.
- Aprender el desarrollo de microservicios y una arquitectura independiente mediante el paso de mensajes.
- Estudio de las diferentes metodologías de programación para su posterior implementación.
- Generar una API que trabajará con la aplicación desde una interfaz gráfica independiente.

Capítulo 3

Marco teórico y estado del arte

En este capítulo explicaremos en profundidad en qué consiste la virtualización y los antecedentes de esta tecnología. Continuaremos con otras tecnologías derivadas de la virtualización como son las máquinas virtuales o los contenedores, los cuales explicaremos más detenidamente.

3.1. Virtualización

3.1.1. Definición

Como bien hemos introducido en el apartado anterior la virtualización se define como el proceso de ejecutar instancias virtuales de un entorno abstrayéndolo de la capa física del mismo [3]. Cada entorno será independiente del resto y tendrá la capacidad de ejecutar su propio Sistema Operativo. Dentro del entorno virtualizado, todo se comporta como si fuera un ordenador independiente, siendo transparente esto al sistema operativo.

3.1.2. Tipos de virtualización

Existen varios tipos de virtualización entre los cuales podemos destacar [4, 5]:

- Virtualización de sistemas operativos: se trata de la abstracción de un HW con el fin de ejecutar un Sistema Operativo en él. Este tipo de virtualización es el más utilizado con el fin de poder ejecutar varias instancias de un sistema operativo en una misma máquina física. Dentro de esta categoría encontramos la virtualización de servidores.

- Virtualización de almacenamiento: se trata de la abstracción de los datos en almacenamiento. Suele implicar la consolidación de múltiples sistemas y

recursos de almacenamiento físicos en uno o varios virtuales. Se pueden generar sistemas de tipo RAID, SAN, NAS, etc.

- Virtualización de red: se trata de la abstracción de la red. Se utiliza con VLANs, VPNs o infraestructura con direcciones IP virtuales.

En nuestro caso nos centraremos en la virtualización de sistemas operativos.

3.1.3. Virtualización de sistemas operativos

La virtualización de sistemas operativos o virtualización de servidores, es la parte que se dedica a la generación de entornos virtualizados con el fin de instalar sistemas operativos estándares en sistemas con características específicas. Además, permite la ejecución de múltiples de estos entornos dentro del mismo *host* pudiendo maximizar la utilización de los recursos del mismo. El funcionamiento dentro del entorno virtualizado será transparente al sistema operativo comportándose de la misma manera que si lo hiciera directamente instalado sobre un *host*.

Dicha capacidad de simular y separar un entorno de los componentes físicos es lo que vamos a utilizar para garantizar la igualdad de características entre entornos y la seguridad de ellos mismos con respecto a otros entornos virtuales, además de la seguridad del sistema sobre el que se ejecutan dichos entornos.

3.1.4. Antecedentes

Para encontrar los primeros trabajos sobre virtualización tenemos que volver atrás hasta los años 60. La investigación sobre la virtualización se dio a causa de la necesidad de mejorar los tiempos en los que los ordenadores se encontraban desocupados debido a que se estaban introduciendo nuevos parámetros para la siguiente ejecución. Con esto se inventó el concepto de compartición de tiempo, implicando que varios usuarios podrán compartir los recursos compartidos de una misma máquina [6].

Fue en 1968 cuando IBM creaba el primer sistema operativo con posibilidad de virtualización [6]. Se basaba en el uso de un hipervisor.

En 1994 Java presentó su sistema de máquinas virtuales que pudieran ejecutar código compilado en Java bytecode. Se desarrolló con el fin de aumentar la portabilidad de los programas entre las diferentes máquinas. Consistía en la creación de una pequeña máquina virtual dentro del sistema operativo del *host* [7]. No se trata de una maquina virtual al uso, ni entraría en el tipo de Virtualización de sistemas operativos, esta diseñada para ejecutar código dentro de un entorno estándar virtualizado de tal forma que se gana en portabilidad. Entraría en el grupo de *Process Virtual Machines*, este termino se explicara más tarde en el punto 3.2.

El desarrollo de los procesadores basados en la arquitectura x86 y su mejora en rendimiento a comienzos de los años dos mil se expandirían las herramientas de

virtualización para equipos domésticos. En 1999 VMware presentó su sistema para procesadores de este tipo [8].

3.1.5. Hipervisor

En toda virtualización contamos con una capa de abstracción que separa el HW de los entornos virtualizados. Este SW se llama hipervisor. Por definición, un hipervisor es un SW encargado de crear y ejecutar máquinas virtuales. El sistema sobre el que se instala el hipervisor lo llamamos *host machine* y las máquinas virtuales que se ejecutan en el *guest machines*.

Esta capa permite realizar las distintas configuraciones de recursos que podemos dar a un entorno virtualizado, en cual podremos asignarle un número de CPUs, una cantidad de memoria RAM, podremos crearle diferentes tipos de tarjetas de red, tarjetas de sonido, tarjetas de vídeo, además de poder crear su sistema de almacenamiento completamente independiente del almacenamiento del equipo físico. Este almacenamiento que se le otorga al entorno será uno o múltiples ficheros que pueden ser movidos a otros hipervisores en otro HW y arrancarlos de la misma manera aun cambiando las características físicas del HW.

Históricamente se diferencian 2 tipos de hipervisores [9][10]:

- Tipo 1:

  Se trata de los hipervisores que hacen función de sistema operativo del *host*, es decir se instala directamente sobre el HW. Suele ser el utilizado en servidores por su mejor eficiencia respecto a los de tipo 2. Algunos ejemplos son: VMware ESXi, Microsoft Hyper-V

- Tipo 2

  Se trata de los hipervisores que se instalan encima de un sistema operativo convencional. Trabaja con el sistema operativo del *host* para obtener una capa de abstracción entre los entornos virtuales y el sistema operativo del *host*. Suelen ser los más utilizados en PCs personales. Algunos ejemplos son: VMware Workstation Player o VirtualBox.

3.2. Máquinas Virtuales

3.2.1. Definición

Una máquina virtual se define como la emulación de un sistema informático mediante SW [11]. El sistema sobre el que se ejecuta una MV se le llama *host* y la MV que se ejecuta sobre un sistema físico se le llama *guest*. Un *host* puede ejecutar

varios sistemas *guest* paralelamente. Según Gerald J. Popek en un artículo que escribió en 1974 decía que las Máquinas Virtuales debía tener 3 características [12]:

- Una máquina virtual genera un entorno en el cual los programas y aplicaciones funcionarán de forma idéntica al de una máquina física.
- Las aplicaciones y programas que se ejecuten en una máquina virtual tendrán un rendimiento similar o un poco menor que una máquina física con las mismas características.
- La máquina virtual tendrá completo control sobre los recursos del sistema, así como memoria, periféricos, disco, tiempo de procesador...

3.2.2. Tipos de Máquinas Virtuales

Existen dos tipos diferenciados de Máquinas Virtuales [13]:

- *Full-System virtual machines* Se trata de MV que generan entornos completos sobre un hipervisor y en el cual se permite instalar sistemas operativos completos y emular un sistema físico tradicional. Estos están aislados entre ellos y a su vez con el sistema operativo del hipervisor. Un ejemplo es una MV de VirtualBox.
- *Process Virtual Machines* Se trata de MV que generan entornos en tiempo de ejecución, generándose en el momento en el que se comienza el proceso y destruyéndose en el que termina el mismo. No están aisladas de otros procesos o MV ni siquiera del sistema operativo del *host*. Su ventaja principal es la portabilidad de procesos entre máquinas. Un ejemplo es la MV de Java.

3.3. Contenedores

3.3.1. Definición

Un contenedor es un entorno de ejecución virtualizado que se ejecuta sobre un el kernel de un Sistema Operativo [14]. A diferencia de las máquinas virtuales, un contenedor intenta emular un Sistema Operativo, pero no un Sistema HW como haría una maquina virtual.

Los *hosts*, usan el propio kernel de su Sistema Operativo con las operaciones nativas para permitir generar entornos virtuales sin necesidad de un medio adicional como es el hipervisor.

La tecnología que se usa para gestionar dichos contenedores se le llama *container engine*. Un *container engine* es un entorno de gestión que tiene como objetivo desplegar una aplicación basada en e uso de contenedores [14]. Dicho motor asigna

recursos como pueden ser de CPU o memoria al contenedor, además de asegurar el aislamiento entre contenedores, la seguridad y la escalabilidad de estos.

En la tecnología de los contenedores existe una organización que tiene como fin el crear un estándar para la tecnología de contenedores. La *Open Container Iniciatibe* (OCI) establece desde 2015 los estándares de desarrollo y funcionalidad de la tecnología de contenedores [15]. Está formado por la mayoría de las empresas que se dedican a esto e incluye dos partes:

- Especificación en tiempo de ejecución: se trata de cómo ha de ejecutar el sistema de ficheros en los contenedores.
- Especificación de imágenes: se trata de la configuración de las imágenes las cuales son una combinación de diferentes características como la configuración de la imagen o el sistema de serialización del sistema de ficheros.

3.3.2. *Kernel Namespaces* y funcionamiento de los contenedores

Se trata de uno de los módulos más importantes dentro de la tecnología de contenedores. Un *namespace* es una herramienta que se encarga de separar los diferentes puntos de montaje de cada contenedor, así como redes, identificador de usuario, tiempo de procesador, etc [16].

Esta tecnología es la que permite la existencia de los contenedores. A grandes rasgos, cuando desde una terminal[1] Linux ejecutamos un comando, este hace una petición al kernel para crear un proceso usando una llamada al sistema tipo *exec()*. Sin embargo, cuando ejecutamos un comando en la terminal dentro de un contenedor, este envía la peticiona al *container engine*, este a su vez le pide al kernel de la máquina crear un proceso contenido usando una llamada al sistema de tipo *clone()*. Este tipo de llamadas son las que permiten crear procesos con sus puntos de montaje, sus usuarios, sus tarjetas de red, etc [16].

3.3.3. Diferencias entre Máquinas Virtuales y Contenedores

Gerald J. Popek definía que una de las características más importantes de las MVs es que mediante el hipervisor se genere un entorno virtualizado completo en el cual se fijen recursos y el sistema operativo de la máquina *guest* lo interprete como si se tratase de un sistema físico normal [12].

[1] Interprete de comandos y ordenes que hacen de interfaz entre el usuario y el sistema operativo [17]

Los contenedores sin embargo, no necesitan generar un entorno con unas características delimitadas sino más bien se ejecutan sobre las propias características físicas de la máquina *host*.

Estas son algunas de las diferencias que existen entre MV y contenedores:

- Características: Las MV usan esencialmente sistemas operativos iguales a los de un entorno físico normal, esto implica que ocupará la misma cantidad de disco, necesitará reservar la misma cantidad de memoria RAM, y sus procesos pasaran a través del hipervisor antes de llegar a los recursos físicos reales de la máquina *host*. La utilización de un hipervisor como intermediario entre las instrucciones de la MV y los recursos físicos es proceso muy exigente en el que se pierde rendimiento. Sin embargo, un contenedor al utilizar el kernel del propio *host*, no requiere de unos recursos definidos ni por ello va a reservar los mismos en su creación, lo hará de forma dinámica según sea necesario para la ejecución del contenedor. Un contenedor por lo tanto es un proceso mucho más ligero que una MV [14].

- Rendimiento: Las MV necesitan emular un HW mediante el hipervisor, así es necesario de generar una BIOS[2] y un entorno preparado para un Sistema Operativo completo. Por contra, los contenedores al estar directamente conectados con el sistema físico mediante el kernel, tienen un rendimiento mucho mejor que un sistema virtualizado mediante hipervisor, pero sin llegar al rendimiento de una instalación directa contra un entorno físico.

- Sistema Operativo: Una MV ejecuta un Sistema Operativo completo incluyendo kernel. Por contra, un contenedor contiene las librerías y binarios propios a los que se les llama imágenes y utiliza el kernel del *host*. Esto significa que el tamaño que ocupa en disco es muy inferior al que ocupa el sistema operativo de una MV [19].

- Velocidad de arranque: El tiempo que pasa desde que un entorno virtual es creado hasta que esta arrancado y completamente funcional es mucho menor en el caso de los contenedores que en el caso de las MV. Esto es debido a no necesitar el arranque de una BIOS y el arranque del sistema operativo estándar. En nuestro caso, esta característica juega un papel muy importante en la realización de este proyecto.

- Almacenamiento: El comportamiento de ambos sistemas esta preparado para ser transparente al usuario, de tal manera que no identifique si esta en una MV, un contenedor o un *host* físico. Sin embargo, en el caso de los contenedores, por

[2]Firmware instalado en una memoria ROM que permite el arranque de un PC. Funciona como puente entre el HW y el SW [18]

defecto cuando se apaga un contenedor se borra la memoria en disco utilizada. Para persistir el almacenamiento en contenedor es necesario el uso de volúmenes. Por otra parte el almacenamiento en MVs no tiene ese comportamiento y es principalmente persistente.

- Interfaces gráficas: Los contenedores no están diseñados para ser utilizados como interfaces gráficas lo cual limita el uso de los mismos y no permite utilizarlo con un entorno gráfico como escritorio.

- Seguridad: El gran problema de los contenedores es lo que rodea a la seguridad de los mismos. Un contenedor al hacer uso directamente del kernel instalado en el *host* tiene acceso directo a las instrucciones del ordenador. De esta forma un contenedor con vulnerabilidades en su acceso podría provocar el colapso del *host* entero. Los contenedores suelen utilizar imágenes previamente creadas con una finalidad muy precisa, estas imágenes se almacenan y descargan desde repositorios. La posibilidad de descargar imágenes de fuentes no oficiales puede causar la utilización de imágenes fraudulentas que pueden contener *malware*. Finalmente la tecnología de contenedores tiene una vulnerabilidad [20] por la cual desde dentro de un contenedor, si este tiene permisos de tipo administrador, podría llegar a acceder al *host* u otros contenedores con permisos administrador y lo que ello conlleva.

Para conseguir un entorno seguro donde ejecutar código, sera necesario dar solución a ciertos aspectos de seguridad. Se realizará limitando el uso de dichos aspectos, bien sea el de la conexión a internet, no utilizando interfaz de red; o el problema del colapso del sistema, limitando el uso de CPU, memoria RAM e incluso de disco.

3.3.4. *Container Engines*

Un *container engine* es un SW encargado de manejar las comunicaciones entre el cliente, que pudiera ser un usuario o un sistema automatizado, y el entorno de virtualización de contenedores. De lo que un *container engine* no se encarga, es de ejecutar, mantener y eliminar un contenedor.

Algunas de las responsabilidades de los motores de contenedores son [21]:

- Proveer de una API[3] o interfaz para la gestión de los contenedores. Desde la cual se realicen las diferentes funcionalidades de forma sencilla para el usuario. Bien sea generar un contenedor, como ejecutarlo, consultar el estado o borrarlo son algunos de los ejemplos que debe permitir un container engine.

[3]Conjunto de funciones o comandos que generan una interfaz de comunicación entre un cliente y un servicio

- Manejar las imágenes utilizadas en contenedores. En primer lugar, el *container engine* se encargará de importar las imágenes y relacionarlas con el resto de imágenes de las que deriva. Si este fuera el caso, las relaciones serian del tipo padre-hijo. En segundo lugar, el *container engine* se encargará de extraer la imagen al disco. Si el contenedor es solo de lectura simplemente se creará el entorno y mapeará virtualmente a la imagen extraída, trabajando sobre esta. Si por el contrario es de lectura y escritura, implementará una capa más donde se generará un volumen añadido a dicha imagen. De tal manera que a la hora de borrar un contenedor solamente se borrara la capa de escritura, permitiendo múltiples contenedores con la misma imagen sin ocupar proporcionalmente el espacio en disco de esta.

- Construir un archivo de configuración. Se trata de alguna manera estandarizar el proceso por el cual indicaremos características del contenedor como son: la imagen a utilizar, opciones de la arquitectura HW, opciones de red, opciones de configuración internas o incluso nombre de este.

En la práctica existen varios motores que permiten realizar estas tareas, algunos de los más populares son:

- Docker: El más popular de todos, se trata del motor original de los contenedores. Hasta la versión 1.11 se trataba de una aplicación con un *daemon* [4] monolítico que se encargaba de todo, desde la descarga de imágenes, lanzar contenedores, exponer la API, etc. [22]. Lo explicaremos en el punto 3.3.7 con más detenimiento.

- ContainerD: Se trata del proceso que se encarga de gestionar la vida del contenedor sobre el *host*. Es decir, se encarga de que la ejecución del contenedor desde el paso de transferir la imagen o los enlaces de red o del de permitir hablar al contenedor con el kernel del *host*. Implementa a su vez un cliente para permitir que otras herramientas como Kubernetes usen este motor[22].

- Crio-O: Se trata de un motor creado por RedHat en 2016, similar en funcionalidad a ContainerD, pero más ligero. Diseñado para correr desde Kubernetes directamente [23].

Además de los mencionados anteriormente cabe nombrar otros menos populares a día de hoy como son: OpenVz, mesos Containerizer, LXC Linux Containers o CoreOs rkt. En la siguiente imagen 3.1 encontramos los motores más utilizados en el año 2019 según el estudio de la empresa Sysdig [23].

[4]Servicio en forma de proceso que se ejecuta en segundo plano en el sistema operativo

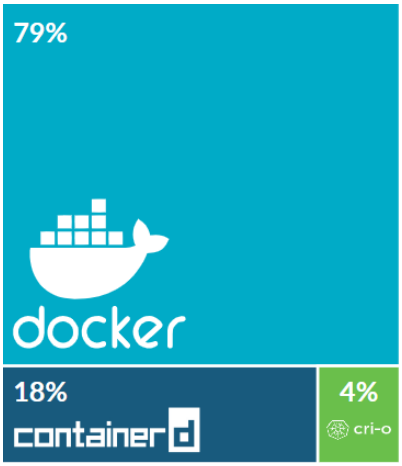

Figura 3.1: Container engine 2019 [23]

3.3.5. Imágenes para contenedores

Una imagen en el contexto de contenedores, es un archivo estático inmutable que contiene todos los códigos ejecutables con el fin de poder ser ejecutado en un entorno de un contenedor. La imagen está formada por librerías, herramientas y otros programas necesarios para ejecutarse en un entorno basado en la utilización de contenedores. Esta imagen comparte el kernel con el propio de la máquina sobre la que se ejecuta el contenedor [24]. Las imágenes habitualmente son obtenidas de un servidor de registro de imágenes y utilizan el formato definido anteriormente llamado Open Container Iniciative explicado en el punto 3.3.1. El OCI define que por cada imagen existirá un archivo que contiene el resto de imágenes o capas de las que deriva y otro archivo con los metadatos específicos de esta última [16].

Capas de Imágenes

Habitualmente las imágenes se pueden generar desde otras imágenes o desde una imagen vacía en la cual se añadirán los elementos necesarios. En el caso de que deriven de otras imágenes, que es lo más habitual, las imágenes generarán una capa extra a la ya existente dando como resultado una imagen en la que se incluyen varias capas [16]. Por ejemplo, partiendo de una imagen de un sistema operativo básico Ubuntu queremos construir una imagen con un servidor *Tomcat*[5] instalado. Esto dará como resultado una imagen con dos imágenes dentro, la de Ubuntu original (que a su vez puede contener más imágenes) y la de la instalación del servidor Tomcat. Esto se puede ver claramente explicado en las siguientes imágenes 3.2, en la cual,

[5]SW que funciona como servidor web

de arriba a abajo se observa como va creciendo una imagen construyendo la misma sobre otras. Esto se observa también en el tamaño final que ocupan las mismas.

Gracias a que las imágenes derivan a su vez de otras imágenes, el tamaño utilizado en disco se ve reducido, pudiendo crear múltiples imágenes derivadas de una sola. De esta forma, si se necesita un entorno de Ubuntu con servidor de correo y otro entorno de Ubuntu con una BBDD, ocupara en el disco del *host* solamente el tamaño de una imagen de Ubuntu además de las capas superiores respectivamente del servidor de correo y el servicio de la BBDD reduciendo en una unidad el espacio ocupado por un entorno Ubuntu.

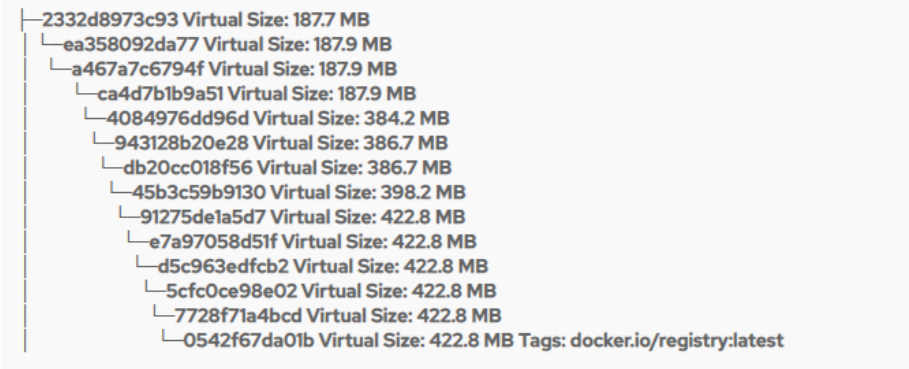

Figura 3.2: Separación de una imagen en sus diferentes capas [16]

Servidor de Imágenes

Un servidor de imágenes consiste en un servidor de ficheros el cual se encarga de almacenar las diferentes imágenes, además de permitir a los usuarios descargarlas y cargarlas. Como explicamos en el punto anterior 3.3.5, una imagen consta de 2 archivos: el que contiene las imágenes de las que deriva y el que contiene los metadatos. De esto tendrá que ser consciente el servidor de imágenes a la hora de devolver el esquema de la imagen pedida.

El servidor de imágenes entra en juego cuando alguien desde un container engine hace la petición de ejecutar una imagen. Primero buscará si dicha imagen existe en la cache local del *host*, si no es así llamara al servidor de imágenes configurado en la máquina en la búsqueda de dicha imagen [16].

Existen dos tipos de servidores de imágenes [24]:

- Públicos: los servidores públicos son los más utilizados habitualmente, están expuestos a internet mediante una dirección IP[6] publica o un servidor DNS[7]. Habitualmente no es necesario estar *logueado* para poder descargar una imagen,

[6]Conjunto de números separados por números que definen una dirección de red de un dispositivo
[7]Servidor que se encarga de asociar un nombre a una dirección IP

pero si es necesario para poder subir una. En estos tipos de servidores no siempre se puede confiar en todas imágenes, pues cualquiera podría modificar una a su gusto para que se controle de forma diferente a la que debería o anuncia, como podría ser *malware*. Para evitar esto, es recomendable usar siempre imágenes oficiales generadas por la propia cuenta oficial del servicio que queremos hacer uso en nuestra plataforma basada en contenedores. Suelen ser el tipo de servidores más usados, y dentro de estos el más usado es `https://www.docker.com/`.

- Privados: los servidores privados en esencia son similares a los públicos, pero con la particularidad de que su acceso está limitado. Habitualmente suelen usarse en LAN[8] dentro de las empresas o en ocasiones diferentes servidores públicos ofrecen dentro de su servicio espacios privados que se pueden alquilar.

3.3.6. Orquestadores de contenedores

Los contenedores se crearon con la idea de generar en ellos microservicios, esto quiere decir que un contenedor - una función. Sin embargo, las aplicaciones que manejamos constan de múltiples microservicios, un ejemplo sería una aplicación que se compone de un servidor Apache para balancear la carga, varios servidores web, un servidor de BBDD y otro microservicio en el cual este instalado un servidor de correo electrónico. Contamos 5 contenedores en una aplicación muy sencilla.

Automatizar el arranque de dichos contenedores es una de las múltiples funciones que realiza un orquestador de contenedores. Además de esto, si uno de los contenedores se cae, ¿Quién nos va a avisar de que se ha caído?, o mejor aún, ¿Quién va a a arrancar otro contenedor con las mismas características?. De esto se encarga un orquestador.

Un orquestador de contenedores es un servicio que se dedica a automatizar la implementación, la administración, el escalado y la red de los contenedores [25].

Los orquestadores tienen dos características básicas [16]:

- De forma dinámica, un orquestador se encarga de repartir la carga entre los *hosts* que forman el *cluster*[9] configurado. Entre otras cosas, esto permite configurar una aplicación de alta disponibilidad por la cual poder arrancar los diferentes nodos de cada servicio si estos se caen automáticamente. También podremos escalar dicha infraestructura automáticamente y de manera muy rápida en función de la carga de la aplicación o del resto de contenedores. Además, como hemos dicho antes, es el encargado de orquestar los diferentes *hosts* que formen el cluster en caso de que hubiera varias máquinas para soportar

[8] Conjunto de dispositivos conectados entre ellos en un espacio físico definido formando una red
[9] Conjunto de recursos informáticos que se comportan como uno solo

dicha aplicación. De esto se encargará un módulo llamado *Master* el cual se comunicará con los *host* que formen el cluster estos llamados nodos. En la figura 3.3 podemos observar un ejemplo de cómo funciona el esquema de cluster que utiliza kubernetes. También proveerá de monitorización sobre los contenedores.

- Provee de una forma estandarizada de definir las relaciones de una aplicación entre sus contenedores, además de las características de los contenedores que forman la aplicación. De esta manera podemos generar un fichero que arranque todos los microservicios que necesitamos en una aplicación. Además, define de forma dinámica las relaciones entre los diferentes contenedores, de tal forma que entre ellos no se hablan en forma de IP sino de nombre. Esto es necesario a la hora de hacer escalable una aplicación o de poder perder contenedores y reiniciarlos manteniendo estos su conectividad. Además, permite que un desarrollador pueda montar su aplicación de forma local en su equipo domestico, y esta sea automáticamente exportable a un entorno de producción. Esta características suele generarse en un archivo de configuración como podría ser un kube.yaml o docker-compose, el cual explicaremos a continuación.

Un ejemplo de un orquestador básico podría ser el propio servicio de docker-compose [26], que mediante una consola CLI[10] y un archivo de texto, permite generar archivos de configuración de la aplicación además de darle las características en número de replicas que requiere que tenga el entorno.

Sin embargo, este orquestador se queda corto a la hora de emplearlo en sistemas en producción. Los más habituales y utilizados son kubernetes o docker-swarm, que además de permitir realizar lo mismo que el anterior, nos da una infinidad de funciones más, como podría ser la gestión de un cluster, la monitorización de la carga de cada uno de los contenedores, la carga de los mismos dentro de los diferentes nodos que forman un cluster o la escalabilidad de estos contenedores si se cumplieran unas características definidas. En la figura 3.4 podemos ver las estadísticas de los orquestadores más utilizados en 2019.

En el TFG utilizaremos un orquestador de contenedores para la generación de los correctores de código, pero en nuestro caso el orquestador lo realizaremos nosotros en base a nuestras necesidades, intentando mantener los principios de los mismos.

[10]Similar a terminal, es una consola de línea de comandos utilizada para interactuar con los diferentes sistemas operativos

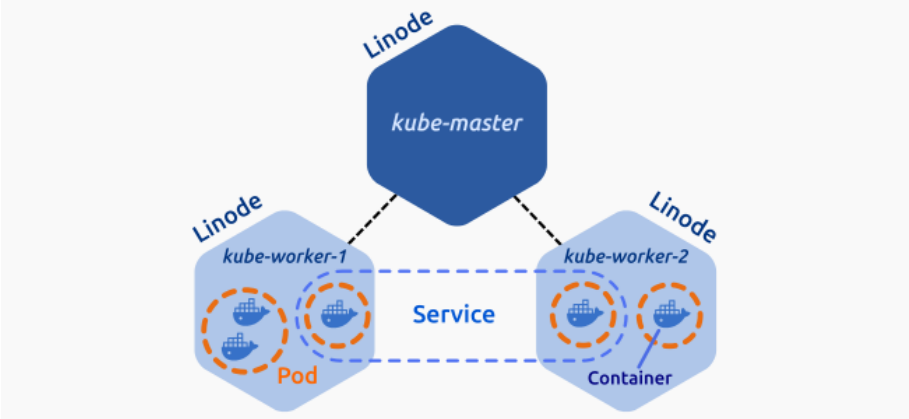

Figura 3.3: Esquema cluster de un orquestador. Obtenido de https://blog.mauriciovillagran.uy/2019/Kubernetes-Lab/

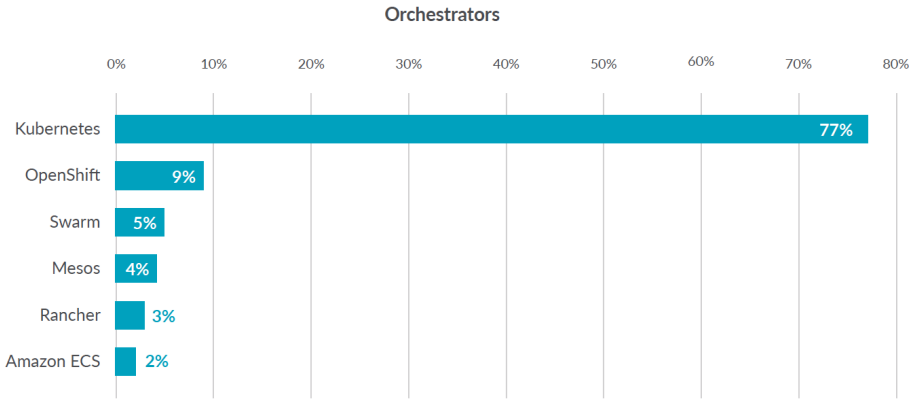

Figura 3.4: Estadísticas de los orquestadores más utilizados en 2019[23]

3.3.7. Docker

Definición

Docker es una tecnología de virtualización desarrollada con el objetivo desplegar aplicaciones de una forma rápida, flexible y automatizada. Las aplicaciones se integran en contenedores, cada uno independiente del resto con sus propios recursos reservados, así como interfaces de red [27].

Docker no necesita de un sistema operativo independiente o de un hipervisor, se basa en el kernel y utiliza el aislamiento de recursos y *namespaces* separados para aislar los contenedores del sistema operativo [27].

Docker como servicio consta de dos partes claramente diferenciadas:

- Docker: se trata del cliente que implementa una API y es el que se encarga de hablar con el demonio o container engine.
- Dockerd: Se trata del demonio o container engine, el encargado de la correcta gestión de los contenedores, así como de las características propias de un motor de contenedores las cuales explicamos en el punto 3.3.4.

Docker además, permite que estos contenedores sean portables y flexibles entre diferentes máquinas, permitiendo desplegar una aplicación desarrollada en un sistema domestico a un servidor o a la nube en cuestión de segundos. Docker es compatible con los diferentes sistemas operativos como Windows, Linux y macOS.

Historia

La tecnología que engloba al producto de Docker, se empezó a desarrollar en el año 2010 por la empresa DotCloud Inc. como servicio PaaS (Plataform as a Service) [27]. En el año 2013 este proyecto fue libreado como un servicio de código libre, permitía ejecutar múltiples aplicaciones con sistema operativo diferentes utilizando contenedores. Al mes de salir ya contaba con más de 10000 personas que habían utilizado esta tecnología en algún momento [28].

En 2014, Docker cambiaba de librería y pasaba de utilizar la LXC a una propia la cual nombraban como *libcontainer*, esto hacía que saliera al público la primera versión estable de Docker: Docker 1.0. En ese mismo año debutaba Kubernetes como orquestador, y Docker se había descargado 2,75 millones de veces desde su página. El año siguiente crecería hasta las 100 millones de descargas. Además, cambiaban el nombre de la compañía a Docker Inc [29, 28].

En 2016 anunciaba la creación de docker-swarm como un nuevo orquestador propio. Además, implementaban la instalación para ejecutar Docker directamente sobre equipos Windows. Desde entonces la empresa se ha centrado exclusivamente en Docker ampliando su negocio en diferentes productos todos alrededor de este mejorando especialmente en el apartado de seguridad [29].

DockerFile

Se trata de un documento de texto que contiene todos los comandos necesarios para construir una nueva imagen. Esto se realizará de forma automática con el comando “docker build”, que ejecutará las instrucciones dentro del fichero DockerFile de forma sucesiva [30].

El DockerFile tiene que incluir como mínimo el comando FROM con el cual se declara la imagen base sobre la que se va a ejecutar el contenedor. Los comandos más importantes en el la sintaxis del DockerFile y algunos de los cuales hemos utilizado en este TFG son los siguientes [31]:

- ADD “[directorio origen o URL]” “[directorio destino]”: copia los ficheros del origen al propio sistema de ficheros del contenedor. Ejemplo “ADD code.java”

- CMD “[application]” “[argument1]”, “[argument2]”, ... : ejecuta un comando dentro del sistema operativo del contenedor cuando este se crea.

- ENTRYPOINT “[application]” “[argument1]”, “[argument2]”, ... : ejecuta un comando del sistema operativo dentro del contenedor cuando este se crea pudiendo ser modificado cuando se ejecuta el comando “run” sin necesidad de volver a construir la imagen.

- FROM “[image name]”: define la imagen base sobre la que correrá el contenedor.

- RUN “[command]”: ejecuta un comando dentro del sistema operativo, el cual es utilizado para crear la imagen del contenedor. Esto implica otra capa en el sistema de ficheros.

- VOLUME [“/directorio1”, “/directorio2”, ..]: monta un sistema de ficheros del *host* en el contenedor.

Además de estos comandos también existen: COPY, ENV, EXPOSE, LABEL, STOPSIGNAL, USER, WORKDIR, ONBUILD.

Docker también nos permite otras funcionalidades como pueden ser las de construir DockerFiles con variables de entorno las cuales se incluirán en la imagen justo antes de ejecutarlas. Dentro del DockerFile estas se declaran con “$NAME”, y desde la API de Docker podremos ejecutarla con

```
docker run -e NAME=ejemplo imagenEjemplo
```

de tal manera que ejecutaremos un contenedor con la “imagenEjemplo” en la cual la variable de entorno HOLA valdrá “ejemplo”.

Un ejemplo de un DockerFile seria la de la figura 3.5, en ella generaremos una imagen que compilará y ejecutará un código Java.

1. En el FROM: indicaremos en el DockerFile que vamos a utilizar la versión 15-alpine de una imagen de OpenJDK. El 15 se refiere al número de *release*, “alpine” se refiere a un Sistema Operativo básico con solamente lo imprescindible instalado, y finalmente OpenJDK es una implementación de Java.

2. En el COPY: copiaremos al entorno del contenedor el fichero main.java en la ruta actual utilizada.

3. En el ENTRYPOINT: finalmente ejecutaremos la línea como comando dentro del Sistema Operativo, de tal forma que primero compilaremos con el comando "javac" y ejecutaremos con el comando "Java". Además, añadimos un timeout el cual se podrá modificar desde fuera del contenedor como variable de entorno. Esto se hará con el carácter "$".

```
FROM openjdk:15-alpine
#Copiamos el archivo main.java al container
COPY main.java .
#Compilamos y ejecutamos el archivo.
ENTRYPOINT javac main.java && timeout -s SIGTERM SEXECUTION_TIMEOUT java main
```

Figura 3.5: Ejemplo de un DockerFile

Docker-compose

Así como el DockerFile era un archivo de configuración sobre una imagen con el fin de ejecutar un contenedor como nosotros queremos. Docker-compose es un archivo de configuración de un orquestador. Es decir, en el archivo docker-compose.yaml podremos generar un servicio con múltiples contenedores los cuales también podremos modificar y cargar variables de entorno en su interior.

En la siguiente figura 3.6 vemos un pequeño ejemplo de un Docker compose en el cual definimos una aplicación que consta de varios servicios como son un servidor Haproxy, un servidor web y una BBDD Mysql. Vemos que en este caso siempre construimos la imagen desde el DockerFile, pero también se podría usar una imagen previamente subida al servidor de imágenes. Además, observamos diferentes opciones de configuración como son las de remapeo del puerto 80 del *host* al del contenedor Haproxy o las variables de entorno que se pasan a los contenedores. Y también importante es la relación dinámica entre ellos por medio de resolución de nombres, con una funcionalidad similar a la de un servidor DNS, usando IP directamente. Esto permite que cada uno de los contenedores sea escalable y no necesitar de controlar el apartado IP para las conexiones mediante red entre ellos mismos.

```
version: '3'
services:
#Balanceador web
    haproxy_web:
        build: ./ServicioHaproxyWeb
        ports:
            - "80:80"
        environment:
            - WEB_SERVICE_IP1=web1
#Creamos los servidores web
    web1:
        build: ./ServicioWeb
        volumes:
            - ./Files:/app/Files
        environment:
            - SPRING_DATASOURCE_URL=jdbc:mysql:replication:
                //mysql-master,mysql-slave/easynotes
#Load de DB
    mysql-master:
        build:
        context: ./
        dockerfile: ./mysql/master/DockerFile
        ports:
            - 3306:3306
        restart: always
        #ADD VOLUME FOR SQL PERSISTENCY
        volumes:
            - ./mysql/master/my.cnf:/etc/mysql/conf .d/mysql.conf.cof
            - ./mysql/master/data:/var/lib/mysql
```

Figura 3.6: Ejemplo de un docker-compose

Capítulo 4

Descripción Informática

Este capítulo resume en detalle el proceso de creación del proyecto, incluyendo la metodología utilizada para el desarrollo del proyecto, así como las herramientas y diseño utilizado.

4.1. Metodología utilizada

Para la realización del proyecto completo he seguido los conceptos de la metodología Agile, esto se debe a la creación de requisitos de manera dinámica según se ha visto el potencial del proyecto. El juez está dividido en dos partes: la parte propia del juez donde se maneja todo lo referente a peticiones, problemas, concursos etc; y la parte del orquestador de ejecución que es donde se ejecuta el código a juzgar y es la que tratamos en este TFG.

La metodología Agile, existe como concepto desde el año 2001 cuando los CEOs de las principales empresas del sector IT del estado de Utah en Estados Unidos se reunieran y pusieran en común las mejores prácticas de cada compañía. Publicaron el manifiesto Agile, por el cual se publicaban procesos de planificación, creación y testing de cada pequeña implementación [32]. En este caso, para la división del proyecto completo en pequeñas partes utilizamos algunas de las características de la metodología Scrum [33], pudiendo modificar ciertos módulos del proyecto sin tocar los otros además de acudir al Scrum Master (en este caso los tutores) en caso de bloqueos, o necesidad de revisión de la arquitectura.

En mi caso, he aprovechado ciertas características de la metodología Agile a mi favor, como podrían ser:

- Comunicación entre cliente(tutor) y desarrollador(estudiante) continua: Utilizando diversas formas de comunicación ya sea telefónica o mediante chats, la comunicación entre el desarrollador del producto y los tutores ha sido el pilar

principal del proyecto, pudiendo añadir nuevas funcionalidades y probar las ya añadidas en las reuniones mantenidas casi semanalmente.

- Participación activa del cliente: Además de una comunicación exhaustiva con el cliente, se han llevado a cabo pruebas de concepto en las cuales se enseñaba a los tutores la funcionalidad del código. Gracias a esta participación se han descubierto nuevas características que se han podido implementar en el proyecto.
- Simplicidad: El proyecto completo se seccionó en tareas a realizar las cuales a su vez han dividió en diferentes hitos para hacer estas mismas lo más simple posibles. Entre otras formas de organizar las tareas, se usó dentro del proyecto creado en Github, el apartado de Issues donde se creaban diferentes tareas a completar o solucionar.

Centrándonos en el desarrollo del proyecto completo, primero se comenzó a realizar el orquestador, el cual a diferencia del juez ha tenido un componente de requisitos fijos mayor al del juez mismo. Por esta parte en la segunda reunión que se mantuvo con los tutores, se definieron algunos hitos sobre los que trabajar para avanzar correctamente en el proyecto. Estos son

1. Script para ejecutar código en Docker desde bash.
2. Código para ejecutar código en Docker desde Java.
3. Generación de un orquestador y conexión con el juez.
4. Incorporar nuevos lenguajes disponibles en el código.

Finalmente, y con carácter individual, decidí generar una lista de tareas o hitos propios los cuales debía seguir en orden con tal de organizar el desarrollo del código. Estos son los siguientes:

1. Formación sobre los jueces de programación.
2. Formación sobre contenedores y formación sobre RabbitMQ. Utilizando guías y documentos obtenidos durante la carrera con el fin de obtener un pequeño “Hello World” en ambas tecnologías.
3. Diseño de la estructura del proyecto, incluyendo la conexión entre el juez y el orquestador. Además de los objetos que va a manejar el orquestador.
4. Desarrollo de un contenedor de ejecución de código manual con una consola interna.
5. Desarrollo de un contenedor de ejecución mediante un script automático.

6. Desarrollo de una clase en Java que traduzca el script anterior a código Java.

7. Desarrollo de la clase en Java para la ejecución de códigos Java dada una entrada y obtención de la salida. Además de la abstracción de la clase en una base de la cual se hereda según el lenguaje en el que se ejecute el código.

8. Implementación del servicio completo de orquestador y su conexión mediante RabbitMQ utilizando objetos mensaje completamente independientes de la aplicación.

9. Reproducción de más clases para la ejecución en más lenguajes.

4.2. Herramientas SW y tecnologías utilizadas

En esta sección repasaremos brevemente las herramientas utilizadas divididas en secciones.

Los lenguajes utilizados en este proyecto son:

- Java: Como base del proyecto y del código.
- Bash: Para la realización de scripts en Linux
- HTML: Con Mustache para la realización de la interfaz gráfica de pruebas.

Las tecnologías utilizadas en el proyecto dentro del código son:

- Docker: Como tecnología de virtualización de contenedores.
- Docker-Java: Como librería que nos permite conectar con Docker desde código Java y controlar la vida de los contenedores de una manera más sencilla.
- Spring: Como base del proyecto, es un framework que se compone de herramientas y utilidades de las cuales vamos a hacer uso como pueden ser las conexiones y gestiones de la BBDD además de la generación de una interfaz gráfica y una API.
- RabbitMQ: Es un SW de encolado de mensajes el cual utilizaremos para comunicarnos entre el Sistema de juzgado y el sistema de ejecución de códigos.
- Spring AMQP: Es un protocolo de mensajes de Spring que utilizaremos en conjunción con RabbitMQ.

El IDE que se ha utilizado para el proyecto es:

- IntelliJ IDEA.

El Sistema sobre el que se ha desarrollado el proyecto ha sido:

- Una máquina virtual con Ubuntu 18.04, virtualizada con VMWare Workstation, con 8 cores y 8Gb de RAM.

4.2.1. Spring

Spring es un framework diseñado para Java que se compone de herramientas y utilidades enfocadas al desarrollo web [34]. Está especialmente diseñado para ser implementado en BackEnd y consta de diferentes módulos, que son servicios diseñados para realizar una función clara como por ejemplo [34, 35]:

- Acceso a datos: El módulo que permite trabajar con BBDD.
- Contenedor de inversión de control: Permite la configuración de componentes mediante inyección de dependencias.
- Modelo vista controlador: Basado en el protocolo HTTP y servlets, diseñado para el desarrollo web.

Spring se utilizará ampliamente en este proyecto, sin embargo, en la parte del orquestador su uso es muy inferior al del juez. Esto es debido a querer simplificar lo máximo posible el orquestador en búsqueda de simplicidad, de esta manera no utiliza ningún tipo de acceso a BBDD o similares. Sin embargo, me he ayudado de algunos módulos de Srping como podría ser el de inversión de control, utilizando la anotación @Configuration o la anotación @Bean para facilitar la configuración del paso de mensajes. Además, también utilizamos el módulo de Spring AMQP para ello.

4.2.2. Docker-Java

Docker-Java es un proyecto público, en el cual se desarrolló un cliente Java para poder conectarse directamente desde dicho cliente al proceso Docker. Esta librería se conecta con Docker para proporcionarnos la completa funcionalidad de Docker desde un código Java.

Esta desarrollada sobre licencia Apache 2.0, con más de 186 usuarios que han aportado de forma libre sus contribuciones y actualmente los desarrolladores siguen implementando o mejorando nuevas funcionalidades siendo la última *release* del cliente el día 10/12/2020 su versión 3.2.7 [36].

El proyecto está plenamente desarrollado en Java y para su uso se pueden descargar las dependencias del mismo desde maven.

Para el uso de la librería debemos conocer los siguientes objetos:

- DockerClient: Se trata de la configuración de la conexión entre el cliente y el proceso de Docker. Aquí tendremos que configurar manualmente en qué dirección está escuchando el socket de Docker. Esto permite que el código sea portable entre máquinas tipo Unix y Windows. Para el caso de Unix el socket por defecto se encuentra en la dirección "/var/run/docker.sock". Se creará un objeto de tipo DockerClient usando el constructor DockerClientBuilder aportando como dato la localización del socket en Linux. Este objeto será sobre el cual se ejecutarán las acciones reaccionadas con los contenedores, como podría ser listar los contenedores que están corriendo, arrancar o parar un contenedor o matarlo.

- CreateContainerResponse: Se trata de un objeto que en el cual se guardara la referencia al contenedor una vez arrancado este. También, se podrá añadir propiedades que se adquirirán en la creación de este. Este objeto equivaldría a comando *docker create*. Se le pueden dar muchas características a nuestro contenedor. Entre otras destacamos por su uso dentro del proyecto: la deshabilitación de la red, las variables de ejecución que pasamos una vez creado al contenedor, el propio objeto HostConfig o el nombre. Además, es sobre este objeto sobre el que se elegirá la imagen que se va a utilizar para el contenedor.

- HostConfig: Con el objeto HostConfig, podremos generar una configuración inicial de las características que queremos que tenga nuestro contenedor, como bien puede ser la cantidad de memoria dedicada, la cantidad de recursos de procesamiento dedicados o el espacio en disco dedicado a ese contenedor. Este objeto complementa al CreateContainerResponse y debe ser llamado desde este durante la creación de este.

Una vez conocidos los diferentes objetos, pasaremos a unas breves instrucciones simplificadas de cómo se generaría un contenedor que por ejemplo tradujera un texto a otro usando dos ficheros de texto.

En el siguiente ejemplo 4.1 se muestra como es la creación, ejecución y recogida de datos de un contenedor:

1. Se crea la conexión con el socket de Docker generando además el objeto DockerClient mediante la conexión con el socket del proceso de docker-engine.

2. Construiremos la imagen a utilizar. Esto solo será necesario la primera vez que se ejecute dicho contenedor, si esta imagen ya existía no será necesario. Para construir la imagen necesitaremos el fichero del DockerFile y con el objeto DockerFile construiremos dicha imagen.

3. Crear un contenedor. Crearemos el contenedor pasándole la imagen a utilizar, este sería el paso en el que añadiríamos las características que queremos darle

al contenedor pero para esta prueba no será necesario. Con esto generamos el objeto CreateContainerResponse.

4. Copiamos el fichero dentro del contenedor. Una vez el contenedor esté creado pero aún no esté arrancado. Esta función está simplificada con el objetivo de mostrar mejor el ejemplo.

5. Arrancaremos el contenedor desde el objeto DockerClient. Para ello utilizamos la conexión con el docker-engine generada anteriormente y le damos el objeto representante del contenedor.

6. Esperaremos a que el contenedor haya terminado, preguntándole mediante el objeto InspectContainerResponse.

7. Copiaremos el fichero de salida de vuelta en el sistema.

```
// Creamos la comunicación con el docker
dockerClient = DockerClientBuilder.getInstance("unix:///var/run/docker.sock")
.build();
//#Construimos imagen
File dckfl = new File( pathname: "DOCKERS/Dockerfile");
String imageId = dockerClient.buildImageCmd().withDockerfile(dckfl)
.exec(new BuildImageResultCallback()
.awaitImageId();
//#Creamos el container
CreateContainer Response container = dockerClient.createContainerCmd (imageId)
.exec();
//Copiamos el codigo
try {
    copiarArchivoAcontenedor(container.getId(), nombre: "codigo.java" , conti ,
    pathDestino: "/root");
    }catch (IOException e) {
        throw new RuntimeException(e);
    }
//#Arrancamos el container
dockerClient.startContainerCmd( container.getId().exec();
//Comprueba el estado del contenedor y espera
InspectContainerResponse inspectContainerResponse=null;
do {
    inspectContainer Response = dockerClient. inspectContainerCmd (container
    .getId().exec();
} while (inspectContainer Response.getState().getRunning();
//#Copiar salida Estandar
String salidaEstandar=null;
try {
    salida Estandar = copiarArchivoDeContenedor(container.getId(),
    pathOrigen: "root/salidaEstandar.ans");
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Figura 4.1: Ejemplo de la ejecución de un contenedor desde Java utilizando la librería Docker-Java

4.2.3. RabbitMQ

RabbitMQ es un SW de encolado de mensajes también conocido como "message broker" o gestor de listas. Define unas colas a las cuales las aplicaciones se conectan con el fin de transferir mensajes entre ellas [37].

Este SW se encarga de gestionar las colas, incluido el guardado de los mensajes hasta su consumición por la aplicación a la que esta destinado, a partir de ahora les llamaremos consumidores. Un mensaje puede incluir cualquier tipo de información o incluso puede mandar una señal para que otro proceso de una aplicación arranque [37].

Dentro del esquema de RabbitMQ tenemos 4 elementos principales:

- Productor: Se trata del servicio que genera el mensaje y lo envía a RabbitMQ para su gestión.
- Consumidor: Se trata del servicio que recibe los mensajes.
- Exchanges: Se trata del componente que recibe el mensaje una vez es producido por el productor. Es el encargado de distribuir el mensaje a la cola apropiada, teniendo en cuenta las reglas de configuración indicadas respecto al servicio de RabbitMQ.
- Queue: Son las colas propiamente dichas, guardan los mensajes hasta que son consumidos por el consumidor.

En la siguiente imagen 4.2 podemos ver el flujo que tendría un mensaje dentro del esquema de RabbitMQ y cuál es la función de cada uno de sus elementos.

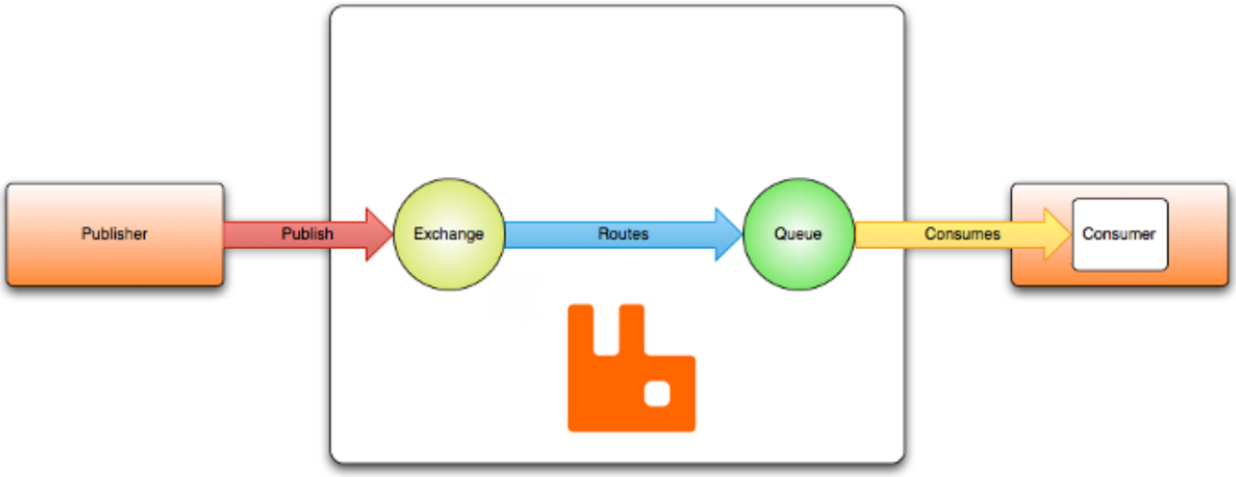

Figura 4.2: Esquema de RabbitMQ. Obtenido desde la página web oficial de RabbitMQ [38].

RabbitMQ lo utilizaremos en nuestro proyecto para independizar el servicio del orquestador de contenedores del servicio del juez propiamente dicho.

AMQP

RabbitMQ utiliza por debajo el protocolo AMQP(*Advanced Message Queuing Protocol*), gracias al cual se acuerda el comportamiento que van a tener los protocolos de envío y recepción de mensajes sobre colas [39].

Este protocolo fue desarrollado a mediados de 2004 por el banco JP Morgan Chaese. Mas tardé el proyecto lo entregarían a un grupo de trabajo formado por empresas como Red Hat, Cisco, IMatix o Microsoft [39].

El protocolo define varias entidades para las conexiones entre un cliente y el servicio RabbitMQ [39]:

- Servidor de mensajes: Es necesario un servidor al cual se conectarán los diferentes clientes, viendo estos siempre el servidor como uno único aunque por detrás pudiera estar implementado como un sistema distribuido.
- Usuario: Se contará con usuarios con credenciales para autorizar o no la conexión a un Servidor de mensajes.
- Conexión: Se generará una conexión física, ligada a un usuario y utilizando protocolos como TCP/IP o SCTP[1].
- Canal: Se generará una conexión lógica con todos los puntos mencionados anteriormente. Un canal no puede operar dos operaciones concurrentemente, siendo necesario la generación de múltiples canales si quisiéramos operar varios mensajes concurrentemente.

Además de estas entidades sobre la conexión, el protocolo AMQP define los siguientes conceptos [40] [39]:

- Exchanges: Son los encargados de enrutar los mensajes a las colas dependiendo de las propiedades que se les quieran dar a los mensajes. Los exchanges además pueden tener las propiedades como que son persistentes ante un reinicio, o auto borrado una vez las colas se han vaciado. Hay varios tipos de exchanges:
  - Directo: Envía mensajes a colas basándose en las *routing keys* del mensaje coincidan con los valores de las colas.
  - De abanico: Envía el mensaje a todas las colas disponibles del exchange.

[1]Diferentes protocolos de transporte en red

  - Topic: Envía el mensaje a las colas filtrando las routing keys mediante cadenas de caracteres definidas en las colas. Por ejemplo, cuando disponemos un mensaje que queremos que vaya a dos colas una general y una especifica.
  - Header:igual que los exchanges de tipo Topic, pero utilizando las cabeceras de los mensajes en vez de las "routing keys".

- Colas: Actuaran como un *buffer*[2] que guardará los mensajes antes de ser consumidos. Acepta propiedades como que pueden ser persistentes ante un reinicio, auto borrado o exclusividad de consumo desde solamente un cliente.

- Mensajes: Se trata de la información que envía el productor al consumidor, además de ciertas cabeceras para insertar propiedades en los mismos como podrían ser tiempo de vida o prioridad. Además, el protocolo cuenta con la posibilidad de generar mensajes de *acknowledgment* para saber que un mensaje ha sido procesado correctamente y borrarlo de la cola correspondiente.

Instalación RabbitMQ

La pieza de SW encargada de soportar la parte de los exchanges y colas de RabbitMQ se llama RabbitMQ Server. Se trata de un proceso que se instala sobre un sistema operativo y que escucha en una conexión TCP a la espera de los mensajes.

A nivel práctico, RabbitMQ server puede ser utilizado de muchas maneras y en muchas plataformas. Puede ser instalado en máquinas Windows, Linux, MacOS e incluso hay servidores en la nube que te ofrecen como servicio un RabbiMQ server para tu uso. Además, una de las instalaciones más populares suele ser sobre un contenedor Docker publicando el puerto del contenedor sobre la máquina *localhost*.

El puerto utilizado para el paso de mensajes es el 5672 mediante TCP, y en el puerto 15672 se encuentra una API basada en HTTP sobre la que se podrá administrar y monitorizar las diferentes opciones de RabbitMQ server.

Para levantar una máquina virtual RabbitMQ Server en un contenedor, es suficiente con lanzar el siguiente comando en un sistema operativo que tenga instalado Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672
-p 15672:15672 rabbitmq:3-management
```

RabbitMQ en Java

Siguiendo la estructura de la que hemos hablado antes, se puede trabajar desde Java varias de las características del servicio RabbitMQ. La más habitual es la

[2]Memoria en la que se almacenan datos de manera temporal para un rápido acceso.

casuística de implementar un productor y un consumidor dentro de nuestro código, pero hay más opciones que se pueden llevar a cabo desde Java como podaría ser la configuración de colas y exchanges del servidor RabbitMQ y el comportamiento de los mismos.

Para trabajar desde Java con RabbitMQ necesitaremos instalar algunas dependencias, se puede utilizar maven para ello. Necesitaremos 3 elementos principales mínimo para ejecutar correctamente una conexión con RabbitMQ desde Java:

- Archivo de configuración: En el archivo de configuración tendremos que indicar las opciones que querríamos para nuestro servicio de mensajes. Esta parte tiene varios niveles de complejidad, empezando con un servicio básico en el que simplemente configuraremos la conexión con el proceso de Rabbit o una configuración un poco más compleja en la que configuramos el número de colas, la relación entre las mismas, la cantidad de consumidores por cola o datos de sesión para la conexión como podría ser usuario y contraseñas. Un ejemplo es el de la siguiente figura 4.3. En este ejemplo configuramos una nueva conexión con la dirección de RabbitMQ Server, en este caso *localhost*, y levantamos dicha conexión en un *Channel*.

```
ConnectionFactory factory = new ConnectionFactory ();
factory.setHost("localhost");
try (Connection connection = factory.newConnection();
     Channel channel = connection.createChannel()) {
}
```

Figura 4.3: Ejemplo configuración RabbitMQ. Obtenido desde la página web oficial de RabbitMQ [41]

- Productor: Para enviar un mensaje a través de RabbitMQ, necesitaremos primero crear la cola. Para ello sobre el canal previamente abierto en el apartado de configuración, declararemos la cola pasando como atributos primero el nombre de la cola, y después diferentes propiedades de la misma, como si va a ser exclusiva, si se borrara automáticamente, si va a ser pasiva o si va a ser duradera entre otras opciones. Una vez publicada la cola, enviaremos el mensaje mediante el *channel* indicando la cola a la que nos queremos dirigir, así como algunas opciones y el mensaje mismo. En este caso al no haber declarado un exchange previamente, no hará falta. Los Mensajes suelen enviarse en forma de cadena de byte o también llamado “byte[ ]”, de esta forma se pueden enviar objetos y estos ser “convertidos” a byte[ ] en el productor y “traducidos” otra vez a su forma anterior en el consumidor. En la siguiente figura vemos un ejemplo de enviar un String que contiene un “Hello World” 4.4.

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
String message = "Hello world!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
System.out.println(" [x] Sent "" + message + "1");
```

Figura 4.4: Ejemplo productor RabbitMQ. Obtenido desde la página web oficial de RabbitMQ [41]

- Consumidor: El consumidor será el encargado de recibir el mensaje además de ejecutar la función para la que se haya programado dicho consumidor. Habitualmente el consumidor se encuentra en una clase diferentes de productor o configuración, o incluso en servicios de programas completamente diferentes. Debido a esto, deberá de contar también con su propia configuración para conectarse, el ejemplo en la figura 4.3 seria valido. Además, también hará falta declarar la cola sobre el *channel* creado en la configuración como se puede ver en la figura 4.4 Una vez tengamos abierta la conexión a la cola, necesitaremos consumir de la misma. Para ello crearemos utilizar la función *consume* del canal abierto. En esta necesitaremos como se hacía en el productor indicar la cola de la que queremos consumir, también necesitamos de la función que tratará el mensaje recibido, el *consumerTag* que es un identificador del consumidor que va a recibir ese mensaje, además de algunas opciones de configuración.

  En el ejemplo de la figura 4.5 podemos observar que cuenta con una función que decodificará el objeto "delivery" que recibimos como mensaje a un objeto de tipo String y lo mostrará por pantalla. Como opciones de configuración en este caso se le pasa el que no envié un ACK[3] a la cola. Esta función se llama directamente desde el canal cuando indiquemos la cola de la que queremos consumir y qué queremos hacer con ella.

```
DeliverCallback deliverCallback = (consumer Tag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received "" + message + """);
};
channel.basicConsume(QUEUE NAME, true, deliverCallback, consumerTag -> {});
```

Figura 4.5: Ejemplo consumidor RabbitMQ. Obtenido desde la página web oficial de RabbitMQ [41]

[3]Acknowledgement, es un mensaje que se utiliza para comunicar al emisor desde el receptor que un mensaje anterior ha llegado.

RabbitMQ en Java con Spring AMQP

Al igual que explicaba en la sección 4.2.3 referente a la utilización de mensajería desde Java. Existe otra forma de trabajar que es mediante la librería para Spring llamada "Spring AMQP". Se trata de un proyecto integrado en el contexto de Spring destinado a la integración de mensajería utilizando el protocolo definido AMQP. Esta librería facilita la gestión de los componentes de este protocolo.

A diferencia de la implementación original de RabbitMQ para Java, Spring AMPQ provee de una estructura más sencilla a la hora de implementaciones con configuraciones complejas.

Para su utilización contaremos como con en el caso del anterior punto con las mismas 3 partes claramente diferenciadas:

- Configuración: En este caso el archivo de configuración será una clase independiente del productor y emisor. Esta clase será precedida de la anotación de Spring "@Configure". En esta misma clase definiremos la cola a utilizar además de la clase en la que encontramos el productor y consumidor.

  Un ejemplo de configuración básicas es la de la siguiente figura 4.6. Se utilizarán "@Beans" para creará los diferentes objetos. Seguiremos los siguientes pasos:

  1. Crearemos la cola, para ello utilizaremos una anotación de tipo "@Bean" en la que incluiremos un Objeto de tipo "Queue". Este objeto lo construiremos con el nombre "hello" que será el nombre de la cola a utilizar.

  2. Crearemos el receptor del mensaje. Al igual que en la cola lo anotaremos con "@Bean" y además le daremos el perfil de receptor de mensajes. La clase de objeto que creamos aquí, sera la clase que buscará luego cuando reciba un mensaje para tratarlo desde esta.

  3. Finalmente crearemos el objeto emisor. Para ello a diferencia del receptor le daremos el perfil de emisor o remitente. También la clase a la que se haga referencia será la luego utilizada por RabbitMQ para la gestión del mensaje.

```
@Configuration
public class Tut1Config {
        @Bean
        public Queue hello() {
                return new Queue("hello");
        }
        @Profile("receiver")
        @Bean
        public Tut1Receiver receiver() {
                return new Tut1Receiver();
        }
        @Profile("sender")
        @Bean
        public Tut1Sender sender() {
                return new Tut1Sender();
        }
}
```

Figura 4.6: Ejemplo configuración AMQP. Obtenido desde la página web oficial de RabbitMQ [42]

- Productor: El productor para enviar un mensaje a través de una cola necesitara utilizar la función "convertAndSend()" del objeto RabbitTemplate.

  En el ejemplo 4.7, mandaremos el String "Hello World!" a la cola "hello". Seguirá los siguientes pasos:

  1. Primero configuraremos la clase para adecuarnos al entorno correspondiente. Para ellos utilizamos la anotación "@Autowired" con el fin de inyectar el objeto previamente creado en el "config". También obtendremos la cola previamente definida.

  2. Crearemos la función que enviará el mensaje. Para ello se utilizara la función "convertAndSend" del objeto RabbitTemplate, la cual dada una cola convertirá y enviará el mensaje que se le pase a la misma. Con el objeto de poder utilizar esta tecnología en entornos independientes no es necesario pasarle a esta función un objeto de tipo cola, solamente con el nombre de la misma sería suficiente.

```
public class Tut1Sender {
        @Autowired
        private RabbitTemplate template;
        @Autowired
        private Queue queue;
        public void send() {
                String message = "Hello World!";
                this.template.convertAndSend(queue.getName(), message);
                System.out.println(" [x] Sent '" + message + "'");
        }
}
```

Figura 4.7: Ejemplo productor AMQP. Obtenido desde la página web oficial de RabbitMQ [41]

- Consumidor: El consumidor se implementa de una manera mucho más sencilla que en el caso de la API original de RabbitMQ para Java. En este caso al haberlo declarado en el archivo de configuración, podremos generar una función que pase por parámetro el propio mensaje. En este ejemplo 4.8, obtendremos un mensaje y lo mostraremos por pantalla para comprobar que realmente funciona el paso de mensajes. Para ello se siguen los siguientes pasos:

  1. Primero utilizando la anotación "@RabbitListener" le indicaremos cual es la cola de la que debe consumir.
  2. Una vez creada la clase, crearemos la función que tratará el mensaje. La función recibirá el mensaje como parámetro en el mismo tipo de objeto en el que se envío y podrá ser utilizado por el algoritmo o función que queramos.

```
@RabbitListener(queues = "hello")
public class Tut1Receiver {
        @RabbitHandler
        public void receive(String in) {
                System.out.println(" [x] Received '" + in + "'");
        }
}
```

Figura 4.8: Ejemplo consumidor AMQP. Obtenido desde la página web oficial de RabbitMQ [42]

4.3. Arquitectura

El diseño de esta arquitectura se basó en los principios de abstracción y encapsulación [43]. De esta manera, cada capa incluirá toda la funcionalidad definida para la misma. Además, también utilizamos los principios de comunicación sencilla mediante paso de mensajes y el principio alta cohesión gracias a la acotación de las funcionalidades de estas.

Usamos los conceptos de modularidad y servicios:

Comenzando con módulos, un módulo es un conjunto de acciones denominadas funciones que comparten un conjunto de datos comunes llamados atributos [43]. De esta forma, cada módulo tendrá una descripción abstracta de la funcionalidad que desarrolla.

Por otra parte, un servicio es una colección de recursos relacionados entre si que presentan una funcionalidad completa al usuario o a las aplicaciones [43]. De tal forma que un servicio suele estar formado por uno o varios módulos.

En nuestro proyecto se empezó a diseñar en base a servicios y una vez definidos estos se definieron los módulos de cada cual. En este apartado solo vamos a hablar de los servicios referentes al orquestador de contenedores resumiendo los referidos al juez propio.

Contamos con varios servicios claramente definidos:

- Interfaz web: Basado en un esquema de peticiones a la API que genera el servicio del juez, se podrá diseñar sobre esta una interfaz web para su utilización.

- Juez: El juez es el servicio más grande dentro del sistema, constituye toda la parte de funcionalidad del juez. Este servicio es muy complejo y se tratara en un TFG aparte. Destacamos aquí ciertos módulos importantes que tienen relación con el orquestador:

    - BBDD: Se trata de una BBDD relacional la cual cuenta con multitud de tablas con las diferentes entidades y relaciones entre las mismas. En el caso del envío al orquestador toda la información es extraída y enviada en un objeto de tipo “Result”.
    - Generación de un envío. Un envío, a partir de ahora una *Submission* es el nombre que se le ha dado al objeto que contiene una prueba de corrección de un código. Es decir, está construida con todos los atributos necesarios para la corrección de un código además del código y las diferentes pruebas a realizar.

      Una *Submission* genera objetos de tipo *Result* con cada entrada que tenga que ser ejecutada. Un *Result* es la unidad mínima que contiene el código a ejecutar junto a una entrada y salida.

    - Envío de un *Result*: Módulo encargado de enviar un *Result* a la cola correspondiente.
    - Recepción de un *Result*: Se trata del módulo encargado de recibir un objeto *Result* desde el servicio de RabbitMQ, previamente ejecutado por el orquestador.
    - Corrector de *Submissions*: Módulo encargado de una vez hayan sido recibidos todos los *Results* ejecutados de una submission, comprobar que cada uno de ellos ha dado el resultado esperado y guardarlo en la BBDD.

- RabbitMQ: El servicio de RabbitMQ es el encargado de mover los mensajes entre los servicios del orquestador y el juez:

    - Colas, exchanges y mensajes: Los diferentes módulos de RabbitMQ encargados de la correcta funcionalidad de la misma. Son explicados con detenimiento en el punto 4.2.3.

- Orquestador de contenedores: El orquestador de contenedores ha sido el objetivo de desarrollo de este proyecto. En el contamos con los diferentes módulos encargados de la correcta ejecución del código. Estos son:

    - Recepción del mensaje: Encargado de recibir un mensaje y enviarlo al ejecutor correspondiente según el lenguaje utilizado en el código.
    - Ejecutor del contenedor: Existirá un módulo por cada lenguaje soportado. Se trata de una función que dado el objeto *Result* enviado desde el juez, genera un contenedor con las propiedades de este. Una vez generado lo ejecuta y saca los resultados de este.
    - Envío del *Result* corregido: Encargado de enviar el *Result* una vez ejecutado como un mensaje a través de la cola correspondiente del servicio de RabbitMQ.

En el siguiente esquema 4.9 incluimos los principales servicios y módulos comentados anteriormente.

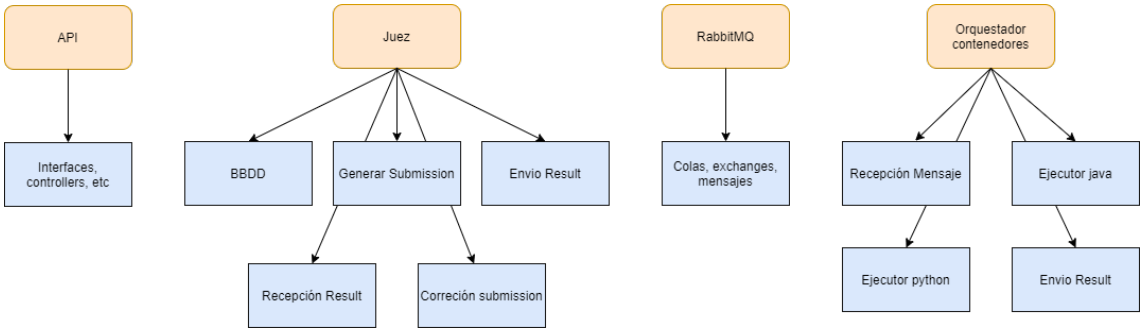

Figura 4.9: Esquema de arquitectura de módulos y servicios de la aplicación de un *Result*

Este esquema está preparado para que cada servicio sea un servicio independiente del resto y pueda ser ejecutado en un servidor diferente. Además, como comentaremos en el punto siguiente 4.3.1, es un diseño escalable en base a la cantidad de *Result* esperando para ser ejecutados.

4.3.1. Escalabilidad del orquestador

El orquestador tiene una arquitectura muy especial, preparada principalmente para ser segura e independiente del juez. Pero, además, debido al diseño utilizado tiene una cualidad muy ventajosa que es la escalabilidad en número de ejecutores que se puede dar.

Cada servicio de ejecución solo puede consumir un mensaje a la vez. Es decir, una vez reciba un mensaje este no podrá consumir otro hasta que se envié el que se estaba ejecutando. Todos los servicios son alimentados desde las colas de RabbitMQ y están limitados a la utilización de un core de la máquina donde se vayan a ejecutar.

Gracias a la utilización de RabbitMQ contamos con el completo control de las colas y la cantidad de mensajes que hay en las mismas. Con esto abrimos la posibilidad de poder arrancar múltiples servicios de ejecución de código de forma dinámica según la demanda de mensajes que haya en la cola, pudiendo incluso especializar máquinas y servicios en lenguajes determinados. En la figura **??** se observa un ejemplo con múltiples orquestadores disponibles para consumir de la cola de ejecución.

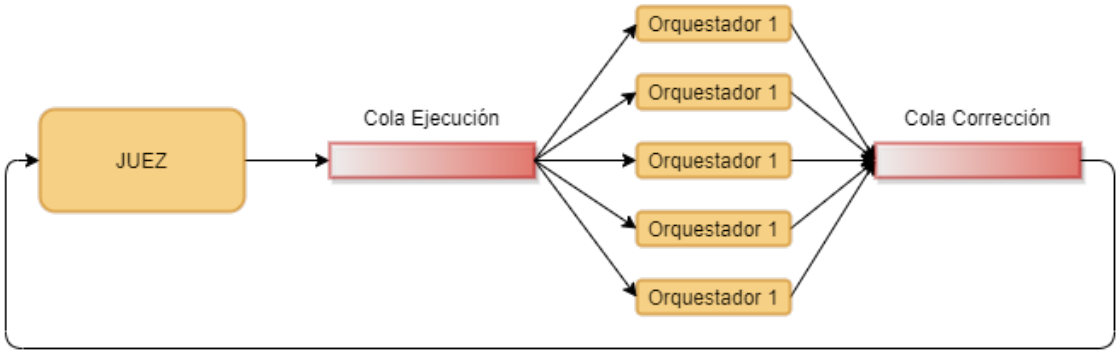

Figura 4.10: Esquema de colas del juez-orquestador.

4.4. Diagrama flujo

La tarea principal del orquestador de contenedores es la ejecución de código. En esta sección explicaremos el diagrama de flujo de esta tarea dividida en dos.

4.4.1. Diagrama de conexión juez-orquestador

Para la comunicación entre el juez y el orquestador y en vista de realizar un servicio independiente se ha utilizado la tecnología de paso de mensajes de RabbitMQ aprovechando la librería de Spring AMQP que existe para Java y Spring. En este punto procedo a explicar el proceso de ejecución de un código respecto del problema que se quiere comprobar.

Una *Submission* es una entidad en la que un usuario intentar resolver un problema. Por lo tanto contiene, entre otras cosas: el código a ejecutar, el lenguaje del código, el problema sobre el que se ha subido, el resultado esperado (*accepted, wrong answer, timelimit, compile error, runtime error*) y finalmente una vez ejecutado el código se tendrá el resultado y ciertos datos de ejecución como tiempo y memoria utilizada. Además, relacionado las entradas del problema con el código de la *Submission* y las propiedades del problema, obtenemos los *Results* que también se guardarán en la *Submission*.

En la siguiente figura 4.11 podemos ver un esquema de la ejecución de una *Submission* utilizando las diferentes colas de RabbitMQ para cada acción. El juez esta preparado para producir objetos de tipo *Result* y añadirlos en la cola de ejecución. El orquestador por su parte está preparado para consumir dicho mensaje de la cola de ejecución y generar con este uno nuevo en la cola de Revisión. Finalmente el juez consumirá el mensaje de la cola de revisión.

El diseño del orquestador se ha realizado contando entre otras con la posibilidad de independizar completamente del juez. Haciendo de esta manera un servicio

independiente y escalable.

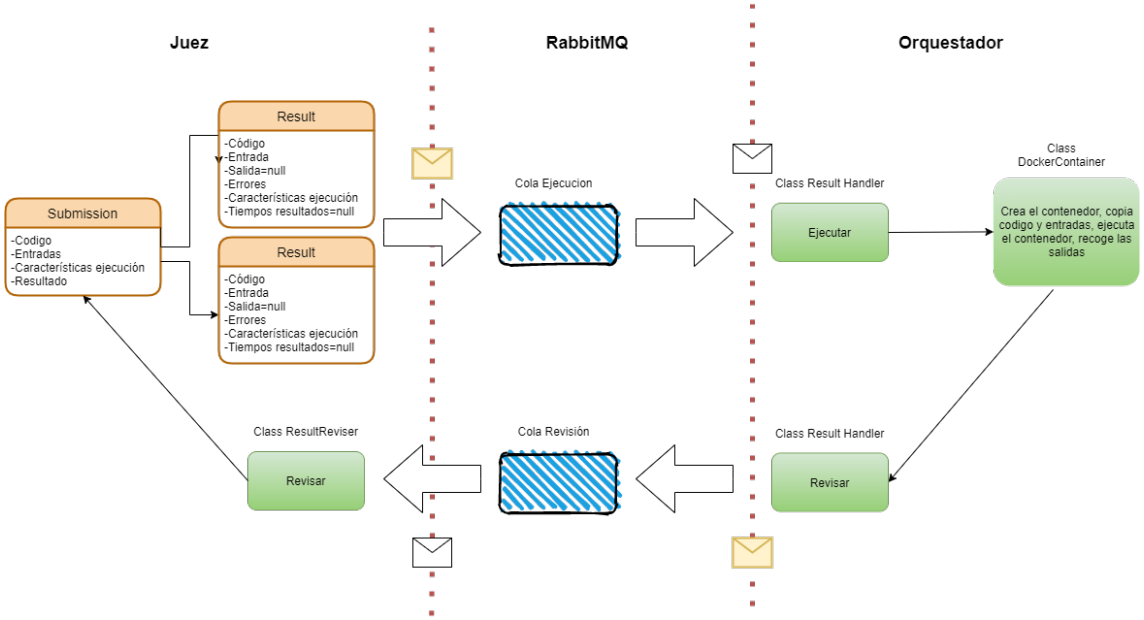

Figura 4.11: Esquema de la comunicación entre el juez y el orquestador

4.4.2. Diagrama de ejecución *Result* en orquestador

En la figura 4.12 observamos un diagrama de flujo de lo que ocurre en el orquestador desde que se recibe un mensaje hasta que se devuelve.

1. Se busca el lenguaje utilizado por el código y según el que sea se creará un objeto de la clase específicamente creada para dicho lenguaje. Esto se ha realizado debido a las diferencias entre diferentes lenguajes como podría ser el comando para ejecutar código o la existencia de un compilador.

2. Una vez creado dicho objeto, se creará dentro el contenedor, dándole las características adecuadas al mismo según se haya elegido en el momento de crear el problema. Si no se hubieran especificado dichas características, se seleccionarán unas por defecto.

3. Se copiará dentro del contenedor el código a ejecutar así como la entrada de datos que se le va a pasar.

4. Se arranca el contenedor y empieza la ejecución interna del mismo.

5. Mediante un bucle, obtenemos el estado del contenedor. En el caso de que no haya terminado se comprueba el tiempo que lleva en ejecución, si supera el limite se matará el contenedor desde fuera.

6. Una vez finalizado el contenedor se obtienen los resultados, así como errores, tiempos y utilización de recursos. Esto se guarda en el mismo objeto de tipo *Result que se había recibido en un principio*

7. Se eliminará el contenedor creado para dicha ejecución y se pone en la cola el mensaje generado.

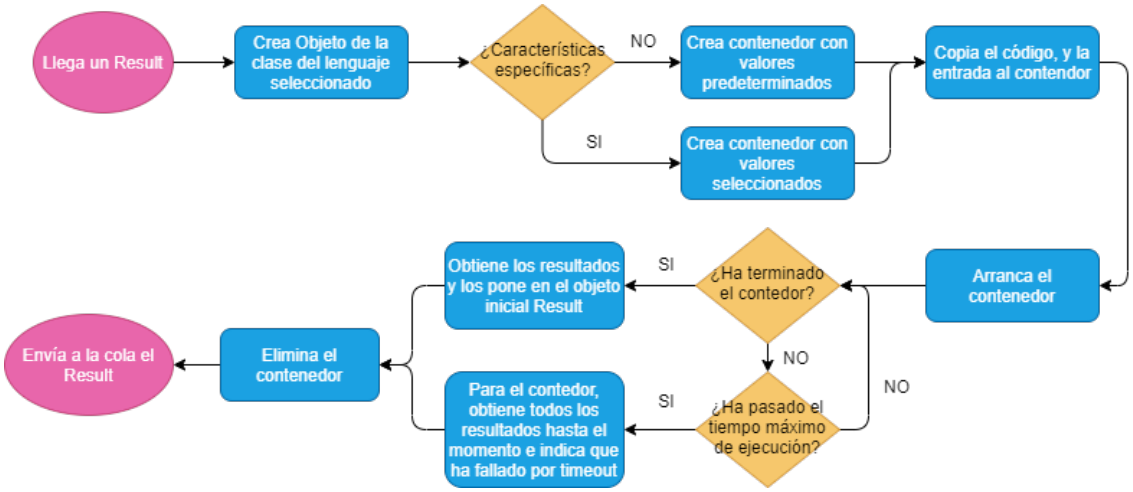

Figura 4.12: Esquema de ejecución de un *Result* dentro del orquestador.

4.5. Diseño de arquitectura

En la arquitectura del proyecto se ha buscado simplificar en gran medida el desarrollo con el fin de poder ampliarlo de una manera sencilla en un futuro. En el caso de los lenguajes disponibles para ejecutar los códigos, se ha diseñado de manera pueden implementarse fácilmente nuevos lenguajes. En esta sección explicamos cómo se ha realizado esto.

4.5.1. Ejecución de código según su lenguaje

Uno de los beneficios del diseño del orquestador y la utilización de contenedores es la facilidad de añadir un nuevo lenguaje para la ejecución de código. Recordemos que para que un contenedor se ejecute, necesita una Imagen. En nuestro caso, para

cada lenguaje de programación existe una imagen modificada para la ejecución de códigos en ese lenguaje.

La imagen sobre la que se trabaja se genera desde un DockerFile. Se ha decido que para mantener los contenedores lo más ligeros y sencillos posibles, utilizaremos como base una imagen "alpine" existiendo para cada lenguaje una con esta característica. Las imágenes alpine son sistemas operativos muy sencillos en los que solamente están instalado las características necesarias para la ejecución de un código. En el ejemplo de la figura 4.13 utilizamos la imagen "python:3.6.12-alpine" para los códigos de python. En la misma figura podemos observar como se crean los diferentes archivos de los cuales se obtendrán las salidas y finalmente el Entrypoint en el que se ejecutará el código con la entrada definida y redirigiendo las diferentes salidas. Además, cuenta con un timeout para no sobrepasar el límite de tiempo de ejecución y una función time que obtiene la memoria utilizada y el tiempo de ejecución del código.

```
FROM python:3.6.12-alpine
WORKDIR /root
RUN touch signal.txt
RUN touch time.txt
RUN touch salidaEstandar.ans
RUN touch salidaError.ans
RUN touch salidaCompilador.ans
ENTRYPOINT timeout -5 SIGTERM
    $EXECUTION_TIMEOUT /usr/bin/time -o time.txt -f "%%M" python3
    $FILENAME2 <entrada.in >salida Estandar.ans
    2>salidaError.ans; echo $? >>signal.txt
```

Figura 4.13: Ejemplo de un DockerFile para python3

Para generar la imagen desde un DockerFile se ha creado una pequeña función que utilizando la librería de Docker-Java, dado un DockerFile construye la imagen y obtiene el ID de la misma devolviéndolo como un String. De esta manera se podrá guardar en la entidad Language tanto el nombre del lenguaje como más importante el ID de este en el sistema de contenedores. Según esta desarrollado el código a día de hoy, en cada arranque del proyecto se intenta construir todos los lenguajes, si un lenguaje ya existe no lo construirá por lo que apenas se pierde tiempo de arranque.

Además de la creación de la imagen necesitaremos una clase específica para cada tipo de lenguaje que se encargue de gestionar cada clase de contenedor. Esto es debido a que no todos los lenguajes funcionan igual y por ejemplo en el caso de Java necesitaremos primero compilar la clase para más tarde ejecutarla y en el caso de python directamente se ejecutará. Esta clase hereda de la clase principal que se

llama Docker que contiene todas las funciones necesarias para crear el contenedor, ejecutarlo, esperar a su finalización y obtener los ficheros de salida de estos.

La acción de añadir un lenguaje nuevo al programa demora alrededor de 15 minutos y sus instrucciones se encuentran en el Apéndice A.2.

Capítulo 5

Resultados

En la realización de este proyecto se buscaba, además de conseguir securizar la ejecución de código, el obtener un sistema rápido para la ejecución del mismo. Este capítulo tiene como objetivo analizar el rendimiento obtenido por la propuesta, utilizando para ello casos de uso reales.

Existían varios motivos por los cuales se decidió crear una arquitectura de automatización y evaluación de código mediante el uso de contenedores y no de otros sistemas de evaluación. Entre estos motivos se encuentran:

- Ejecución de códigos en entornos seguros, individuales y equitativos. El objetivo aquí es no modificar el resultado de un ejercicio pudiendo garantizar la seguridad del sistema.

- Velocidad de corrección. Utilizando la tecnología de contenedores, obtenemos un tiempos de ejecución de código muy reducidos, minimizando el tiempo de respuesta.

- Consumo de recursos y escalabilidad. Permitiendo la arquitectura la independencia del orquestador del resto del sistema, en caso de que sea necesario se podrá ampliar la cantidad de nodos que ejecuten el orquestador. Además, gracias al bajo consumo de recursos, se podrán mantener un gran número de ejecutores de código por sistema.

La utilización del orquestador mediante un usuario es complicada debido a que esta diseñado para utilizarlo desde otro sistema automatizado, sin embargo, durante la realización del proyecto y a la finalización del mismo, se han obtenido gratos comentarios respecto al funcionamiento de la corrección de los códigos. Usuarios habituales de jueces de programación se han sorprendido por la velocidad con la que el sistema corrige los códigos, y la capacidad de ello. Además, agradecen que no utilice ningún tipo de filtro previo a la ejecución que pudiera limitar los desarrollos

realizados para la resolución del los problemas. Por parte de compañeros de trabajo y usuarios ajenos al uso de estos jueces, el descubrimiento más grande para ellos es la utilización de sistemas de virtualización dentro de contenedores. Y finalmente los tutores han impulsado este proyecto hasta el punto de generar dos nuevas ramas en búsqueda de la mejora del mismo para finalmente poder utilizarlo en competiciones oficiales.

Continuando con los resultados, la máquina utilizada para el desarrollo del proyecto y las pruebas del mismo es una Maquina Virtual, virtualizada con VMWare Workstation Player 15, a la cual se le han dado 8 cores de CPU y 8Gb de memoria RAM. Sobre la MV se ha instalado un sistema operativo Ubuntu 18.04 en un sistema de ficheros XFS.

Pruebas de tiempo de arranque del entorno virtualizado:

- Máquinas Virtuales: El rendimiento para la MV se ve muy perjudicado por el sistema de ficheros utilizado. Este sistema es necesario para la limitación de disco en la tecnología de contenedores. El tiempo obtenido en la figura 5.1 de 19 segundos se consigue mediante el comando "systemd-analyze". además del arranque del sistema operativo, hay que añadir el tiempo de arranque de la BIOS, este demora una media de 6 segundos, generando un total de 25 segundos en arrancar el entorno virtualizado. En la figura 5.2 podemos ver los resultados del comando.

- Contenedores: Para el caso de los contenedores, el tiempo completo de vida lo medimos imprimiendo el momento en el que es creado y el momento en el que ya hemos obtenido todas las salidas y borramos la máquina. Por tanto, contamos con dos variables más que no estamos incluyendo en el caso de la MV como seria el copiar la ISO de arranque de la misma, obtener los ficheros de salida y borrar finalmente la MV. El resultado de la ejecución de contenedores puede variar con el lenguaje utilizado y por supuesto del tiempo de ejecución del código. En el caso del contenedor Java las pruebas obtenidas se encuentran en la figura 5.3 en la cual podemos ver una duración de 2 segundos aproximadamente. La media de ejecución de otros códigos en otros lenguajes suele encontrarse entre 1 y 4 segundos.

Pruebas de rendimiento y utilización de recursos:

- Máquinas virtuales: En el caso de la ejecución del código directamente sobre una MV Ubuntu mencionada anteriormente, obtenemos un resultado de 0,3 segundos en el tiempo de ejecución del código, además de un consumo total de memoria de 2 GBs. La máquina virtual de Java (JVM) consume aproximadamente 150 MBs de RAM. En la figura 5.2 vemos el tiempo real de ejecución con el comando "time" y la memoria que está utilizando la máquina.

- Contenedores: En el caso de la ejecución mediante contenedores, encontramos una gran diferencia en no necesitar arrancar un nuevo kernel y almacenarlo en memoria. Esto implica una reducción grande en el uso de memoria. En este caso la MV de Java (JVM) consumirá la misma cantidad de RAM pero el contenedor vació apenas utiliza 154 MBs para levantarlo. En el caso del tiempo de ejecución del código dentro de un contenedor son 0,4 segundos. Hay que tener en cuenta que se trata de la misma máquina mencionada anteriormente en las pruebas, pero está limitada a solamente 1 CPU para la ejecución del contenedor. En la figura 5.4 observamos el tiempo real de ejecución obtenido con el comando "time" y la memoria de la MV de Java que crea esta. En la siguiente tabla 5.1 podemos observar los resultados resumidos.

En resumen, y como podemos observar en la tabla de resultados 5.1, la utilización de virtualización mediante contenedores mejora en más de 10 veces el tiempo y memoria RAM utilizada para el arranque y ejecución de un código en un sistema virtualizado. Con ellos ganaremos en el número de códigos a ejecutar en un tiempo determinado, gracias a los recursos utilizados, permitiendo tener varios ejecutores en el mismo sistema, y a la velocidad con la que se ejecutan dichos códigos.

Por ejemplo, dado una infraestructura para utilizar como orquestador en la que contamos con 8 cores y 8 Gbs de RAM, podremos ejecutar 240 códigos por minuto, dando a cada uno de ellos un core completo y 1 GB de memoria. Esto significa que para un problema que tenga 5 casos de prueba, podrá corregir el envío de 48 usuarios por minuto. En un concurso en el que cada usuario realice un envío cada 10 minutos y los envíos sean de forma escalonada, la infraestructura de virtualización mencionada anteriormente podrá mantener a 480 usuarios sin que ninguno de los envíos tenga que almacenarse en la cola para ser ejecutado, es decir, ejecuciones inmediatas.

	Tiempo arranque	**Tiempo ejecución**	**Memoria Total**
MV	25 segundos	0.3 segundos	2 GBs
Container	2 segundos	0.4 segundos	0.15 GBs

Figura 5.1: Tabla de resultados de MV y contenedores.

En la figura 5.2, obtenemos toda la información referente al tiempo de arranque de una MV, el tiempo de ejecución y la cantidad de memoria RAM utilizada. En la primera parte de la imagen observamos una captura del sistema de monitorización de recursos de Ubuntu, sobre la cantidad de RAM utilizada por el SO. Para obtener

el tiempo que demora el sistema operativo en arrancar utilizamos el comando `systemd-analyze` con el cual analizamos el tiempo utilizado en el último arranque para cargar el kernel. En el siguiente comando, utilizamos "time" y el comando de ejecución en java para obtener el tiempo que tarda en ejecutarse el código en cuestión.

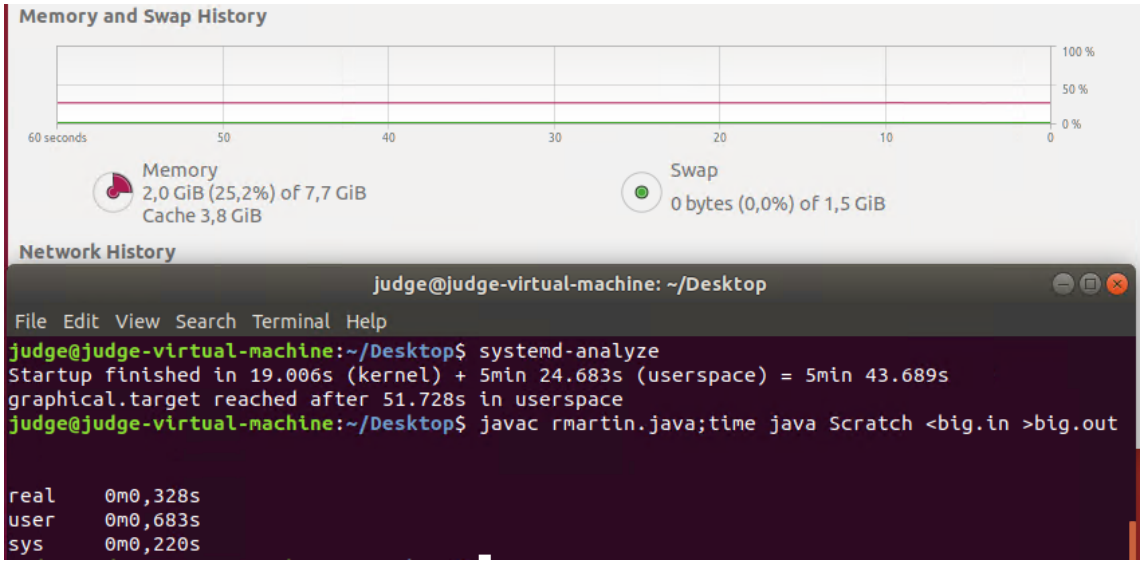

Figura 5.2: Tiempo de arranque, tiempo de ejecución y consumo de memoria de una MV

En la figura 5.3, podemos observar una captura del *log* del orquestador en el cual vemos que desde el momento que escribe que crea el contenedor hasta el momento en el que escribe que el contenedor ha finalizado, pasan menos de 2 segundos. Incluyendo dentro de este tiempo, el arranque del entorno virtualizado y la ejecución del código, por lo tanto el tiempo de arranque del contenedor será el total obtenido menos el demorado por la ejecución del código.

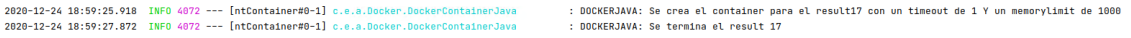

Figura 5.3: Tiempo de arranque y ejecución de un contenedor

Finalmente, en la imagen 5.4 podemos observar desde los valores almacenados en la BBDD, el tiempo de ejecución del código, el cual se obtiene como anteriormente gracias al comando time, y la memoria utilizada por el ejecutor.

ID	CODIGO	EXEC_MEMORY	EXEC_TIME	FILE_NAME
9	import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader;	154528.0	0.4000000059604645	rmartin

Figura 5.4: Tiempo de ejecución código y recursos utilizados por el contenedor

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo, desarrollaremos las conclusiones y trabajos futuros generados gracias a este TFG.

6.1. Conclusiones

El proyecto nacía con la necesidad de crear un nuevo sistema de juez de programación para concursos y clases. Con esto se generaron 2 Trabajos de Fin de Grado que tuve la suerte de poder llevar a cabo. La realización completa del proyecto ha sido muy satisfactoria, utilizando conocimientos adquiridos en la carrera, así como conocimientos nuevos y obteniendo nuevas ideas tanto de tutores como compañeros de trabajo y profesión.

Con la finalización del proyecto se ha conseguido un buen producto, pudiendo implementar varias funcionalidades que no estaban previamente definidas y creando una buena base para futuras ampliaciones. Las conclusiones del juez se amplían en la memoria destinada al mismo.

Por la parte del orquestador, se ha obtenido un sistema el cual es fiable y robusto, a la par que sencillo y escalable. Se ha cumplido con los objetivos planteados en el capítulo 2 y es un buen sistema para la ejecución de código, bien sea mediante el juez diseñado o incluso para otras aplicaciones.

Durante la realización de este proyecto he obtenido múltiples nuevos conocimientos y mejorado otros.

- Nuevos conocimientos sobre contenedores, en especial sobre Docker.
- Nuevos conocimientos sobre bash de Linux y *scripting*.
- Nuevos conocimientos sobre mensajería, específicamente sobre RabbitMQ y AMQP.

- Nuevos conocimientos sobre LaTeX.
- Mejorados los conocimientos de Java, de la programación orientada a objetos.
- Mejora del uso de Spring Data JDBC, de los *controllers* de Spring y de las anotaciones.

Finalmente, este proyecto se ha desarrollado en pensando en licenciarlo bajo código libre, de tal manera que los diferentes estudiantes u organizaciones que quieran adaptarlo a su entorno puedan hacerlo. Este proyecto seguirá teniendo continuidad dentro del la Universidad Rey Juan Carlos, por al menos otros dos estudiantes.

6.2. Trabajos futuros

Gracias principalmente a la arquitectura definida, este proyecto tiene un amplio crecimiento a futuro. Destacando que actualmente existen otros 2 TFGs más desarrollándose que utilizan este como base. Son:

- Creación de una Interfaz Web para el juez. Se trata de la creación de una interfaz web, utilizando la API diseñada en el TFG del juez.
- Mejora del juez. Es un TFG para revisar y mejorar ciertas estructuras utilizadas en este, además de implementar más funcionalidades en la parte del orquestador como añadir nuevos lenguajes o arreglar un fallo de seguridad con respecto a los timeout de los contenedores.

Estas mejoras están siendo realizadas actualmente, pero aparte existen otras que pueden ser explotadas a futuro como son:

- Mejora del orquestador. Se trata de una revisión de la implementación de este a nivel de servicio, como puede ser la implementación de las opciones de escalabilidad mencionadas en el punto 4.3.1. Estas consisten en la escalabilidad del número de servicios de ejecución de código, aprovechando incluso funcionalidades de los contenedores y pudiendo plantear la instalación de la aplicación en una plataforma basada en contenedores, teniendo cada ejecutor de código funcionando dentro de un contenedor y pudiendo levantar de esta manera la aplicación de una forma mucho más escalable y rápida.
- Integración dentro del aula virtual. Mediante la utilización del proyecto Spring-LTI [44] de Raúl Martín Santamaría, podemos incluir la gestión de usuarios al aula virtual centrándonos en la funcionalidad del juez.

Bibliografía

[1] *Worldwide Professional Developer Population of 24 Million Projected to Grow amid Shifting Geographical Concentrations*. Evans Data Corporation. Mayo de 2019. URL: https://evansdata.com/press/viewRelease.php?pressID=278.

[2] RishavSen Choudhury. *Companies hire from these 7 coding websites*. URL: https://medium.com/skillenza/companies-hire-from-these-7-coding-websites-eb3b0ef5f95e.

[3] *What is virtualization?* opensource.com. URL: https://opensource.com/resources/virtualization.

[4] Wikipedia. *Virtualización — Wikipedia, La enciclopedia libre*. [Internet; descargado 27-enero-2021]. 2020. URL: https://es.wikipedia.org/w/index.php?title=Virtualizaci%C3%B3n&oldid=131860993.

[5] RedHat. *¿Qué es la virtualización?* URL: https://www.redhat.com/es/topics/virtualization/what-is-virtualization.

[6] Victoria Fedoseenko. *A brief history of virtualization, or why do we divide something at all*. Nov. de 2019. URL: https://www.ispsystem.com/news/brief-history-of-virtualization.

[7] Paul Wilk. 2018. URL: https://ssh.guru/brief-history-of-virtualization/.

[8] Charles David Graziano. «A performance analysis of Xen and KVMhypervisors for hosting the Xen Worlds Project». En: (2011), págs. 3-4. URL: https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3243&context=etd.

[9] Wikipedia. *Hypervisor*. Oct. de 2020. URL: https://en.wikipedia.org/wiki/Hypervisor.

[10] RedHat. *What is a hypervisor?* URL: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor.

[11] Wikipedia contributors. *Virtual machine — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-November-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=985899745.

[12] Robert P Popek Gerald J.; Goldberg. «"Formal requirements for virtualizable third generation architectures"». En: 1974, pág. 413.

[13] elprocus.com. *What is a Virtual Machine – Types and Advantages*. URL: https://www.elprocus.com/virtual-machine/.

[14] Donald Firesmith. *Virtualization via Containers*. Sep. de 2017. URL: https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html#:~:text=Note%20that%20virtualization%20via%20containers,rather%20than%20the%20underlying%20hardware..

[15] *About the Open Container Initiative*. URL: https://opencontainers.org/about/overview/.

[16] Scott McCarty. *A Practical Introduction to Container Terminology*. Nov. de 2018. URL: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/#h.epuvi2fkxbx2.

[17] *¿Qué es una Terminal en Linux?* URL: https://www.programoergosum.com/cursos-online/raspberry-pi/243-terminal-de-linux-en-raspberry-pi/que-es-una-terminal-en-linux.

[18] Rodrigo Alonso. *Sabes que tu PC tiene BIOS pero, ¿sabes qué es exactamente?* Nov. de 2020. URL: https://hardzone.es/reportajes/que-es/bios-pc/.

[19] *Contenedores frente a máquinas virtuales*. Oct. de 2019. URL: https://docs.microsoft.com/es-es/virtualization/windowscontainers/about/containers-vs-vm.

[20] *VE-2019-5736: RunC Container Escape Vulnerability Provides Root Access to the Target Machine*. URL: https://www.trendmicro.com/vinfo/es/security/news/vulnerabilities-and-exploits/cve-2019-5736-runc-container-escape-vulnerability-provides-root-access-to-the-target-machine.

[21] fatherlinux. *So, What Does A Container Engine Really Do Anyway?* Dic. de 2018. URL: http://crunchtools.com/so-what-does-a-container-engine-really-do-anyway/.

[22] Kibet John. *Docker vs CRI-O vs Containerd*. Dic. de 2019. URL: https://computingforgeeks.com/docker-vs-cri-o-vs-containerd/.

[23] Sysdig. *2019 Container Usage Report*. 2019. URL: https://dig.sysdig.com/c/pf-2019-container-usage-report?x=hJvo1P&utm_source=gated-organic&utm_medium=website.

[24] Margaret Rouse. *container image*. URL: https://searchitoperations.techtarget.com/definition/container-image.

[25] RedHat. *What is container orchestration?* URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration#:~:text=Container%20orchestration%20automates%20the%20deployment,scaling,%20and%20networking%20of%20containers.&text=Container%20orchestration%20can%20be%20used,without%20needing%20to%20redesign%20it..

[26] Docker. *Overview of Docker Compose*. URL: https://docs.docker.com/compose/.

[27] Wikipedia. *Docker (software) — Wikipedia, La enciclopedia libre*. [Internet; descargado 21-noviembre-2020]. 2020. URL: https://es.wikipedia.org/w/index.php?title=Docker_(software)&oldid=130961937.

[28] Emily Mell. *The evolution of containers: Docker, Kubernetes and the future*. Abr. de 2020. URL: https://searchitoperations.techtarget.com/feature/Dive-into-the-decades-long-history-of-container-technology.

[29] Christopher Tozzi. *Docker at 4: Milestones in Docker History*. Mar. de 2017. URL: https://containerjournal.com/features/docker-4-milestones-docker-history/.

[30] *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/#:~:text=A%20Dockerfile%20is%20a%20text,can%20use%20in%20a%20Dockerfile%20..

[31] O. S Tezer. *Docker Explained: Using Dockerfiles to Automate Building of Images*. 2013. URL: https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images.

[32] Maria Tena. *¿Qué es la metodología 'agile'?* URL: https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/.

[33] *WHAT IS SCRUM?* URL: https://www.scrum.org/resources/what-is-scrum.

[34] Wikipedia. *Spring Framework — Wikipedia, La enciclopedia libre*. [Internet; descargado 24-diciembre-2020]. 2020. URL: %5Curl%7Bhttps://es.wikipedia.org/w/index.php?title=Spring_Framework&oldid=122817102%7D.

[35] Yanina Murada. *Conoce qué es Spring Framework y por qué usarlo*. Jun. de 2018. URL: https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/.

[36] URL: https://github.com/docker-java/docker-java.

[37] LOVISA JOHANSSON. *Part 1: RabbitMQ for beginners - What is RabbitMQ?* Sep. de 2019. URL: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html#:~:text=RabbitMQ%20is%20a%20message-queueing,include%20any%20kind%20of%20information.&text=The%20receiving%20application%20then%20processes%20the%20message..

[38] URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html.

[39] Wikipedia. *Advanced Message Queuing Protocol — Wikipedia, La enciclopedia libre*. [Internet; descargado 13-diciembre-2020]. 2020. URL: %5Curl%7Bhttps://es.wikipedia.org/w/index.php?title=Advanced_Message_Queuing_Protocol&oldid=130329111%7D.

[40] LOVISA JOHANSSON. *What is AMQP and why is it used in RabbitMQ?* 2019. URL: https://www.cloudamqp.com/blog/2019-11-21-what-is-amqp-and-why-is-it-used-in-rabbitmq.html.

[41] URL: https://www.rabbitmq.com/tutorials/tutorial-one-java.html.

[42] URL: https://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html.

[43] *Arquitectura de sistemas de información*. URL: https://www.mastergeoinformatica.es/wp-content/uploads/2016/06/FSI-SI-T2-ArquitecturaSI.pdf.

[44] Raul Martin. *Spring LTI*. URL: https://github.com/rmartinsanta/spring-lti.

Apéndice A

Guía de instalación

A.1. Guía de instalación de la aplicación

La instalación del orquestador simplemente requiere de dos componentes:

1. Sistema Operativo: El Sistema Operativo para el que se ha diseñado es GNU/Linux, en concreto Ubuntu 18.6 pero funcionará sin problema en cualquier sistema Unix y así se ha probado. Para su funcionamiento en Windows es necesario modificar la conexión de Docker-Java con la API de Docker.

2. Sistema de ficheros XFS: Para obtener la funcionalidad de limitar el espacio de almacenamiento de los contenedores es necesario utilizar como fileSystem un sistema basado en XFS.

3. Docker-engine: Las instrucciones de instalación de Docker se encuentran en su página web oficial. https://docs.docker.com/engine/install/

4. RabbitMQ: la instalación de RabbitMQ la realizaremos utilizando Docker como servicio de contenedores. De esta forma con el siguiente comando levantaremos una instancia de RabbitMQ.

```
docker run -it --rm --name rabbitmq -p 5672:5672
-p 15672:15672 rabbitmq:3-management
```

A.2. Guía añadir un nuevo lenguaje

Para añadir un lenguaje necesitaremos seguir los siguientes pasos:

1. Crear un DockerFile. Sera necesario crear una carpeta dentro del directorio DOCKERS cuyo nombre sea el lenguaje a implementar. Dentro de esta carpeta

crearemos el fichero DockerFile, el cual nos basaremos en el resto ya existentes cambiando el ENTRYPOINT y la imagen de la que derivamos

2. Añadir al basicController la inicialización del DockerFile así como la inclusión del lenguaje en la BBDD.

3. Crear la clase correspondiente en la carpeta Docker, llamándola DockerLenguaje (cambiando Lenguaje por el lenguaje a tratar). Tendremos que implementar las diferentes características y variables de entorno del lenguaje en cuestión. Las entradas y salidas no debería hacer falta modificarlas, si acaso añadir o retirar según el lenguaje.

4. Añadir en la clase ResultHandler el "case" para el lenguaje

5. Para que también lea los códigos que entran desde los problemas mediante los ZIPs, tendremos q modificar la clase ZIPHandlerService la función SelectLenguaje añadiendo la extensión y traduciéndola al lenguaje que toca.