UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

"Serverless Computing", Funciones como Servicio (FaaS) para el soporte de cargas computacionales en la nube.

TRABAJO FIN DE MÁSTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Javier Rodríguez Domínguez
Tutor: José Manuel Bernabeu Aubán

Curso 2019-2020

Resum

El treball realitzat ha consistit en la creació d'una plataforma d'execució de funcions com a servei (*FaaS*), ja que constitueixen la peça central del paradigma de la computació *serverless*. Aquesta plataforma pren com a prioritat l'aïllament entre diferents invocacions de les funcions, amb la finalitat d'assegurar el determinisme als seus resultats, així com la seguretat per a la informació dels usuaris. El projecte ha avançat de la següent manera. S'ha dut a terme l'estudi de plataformes *FaaS open source* existents, per a analitzar-les i identificar els seus aspectes a millorar. Partint d'aquests aspectes, en concret l'aïllament, i altres inherents al concepte fonamental de *FaaS*, com el rendiment i l'alta disponibilitat, s'ha realitzat un disseny de l'arquitectura de la plataforma. Aquest disseny ha servit com a guia per al desenvolupament d'un prototip, que compleix amb les funcionalitats relatives a un sistema *FaaS* i és configurable, permitent la programació de noves heurístiques per a la càrrega i execució de funcions. Per últim, s'ha provat el sistema desenvolupat en aquest projecte, validant el seu correcte funcionament i mesurant el seu rendiment, comparant les heurístiques proposades entre elles.

Paraules clau: Funcions con a servei, FaaS, Serverless, Computació al núvol, Aïllament

Resumen

El trabajo realizado ha consistido en la creación de una plataforma de ejecución de funciones como servicio (*FaaS*), pues constituye la pieza central del paradigma de computación *serverless*. Esta plataforma toma como prioridad el aislamiento entre diferentes invocaciones de las funciones, con el fin de asegurar el determinismo en sus resultados, así como la seguridad para la información de los usuarios. El proyecto ha avanzado de la siguiente manera. Se ha llevado a cabo el estudio de plataformas *FaaS open source* existentes, para analizarlas e identificar sus aspectos de mejora. Partiendo de estos aspectos, en concreto el aislamiento, y otros inherentes al concepto fundamental de *FaaS*, como el rendimiento y la alta disponibilidad, se ha realizado un diseño de la arquitectura de la plataforma. Este diseño ha servido como guía para el desarrollo de un prototipo, que cumple con las funcionalidades relativas a un sistema *FaaS* y es configurable, permitiendo la programación de nuevas heurísticas para la carga y ejecución de funciones. Por último, se ha probado el sistema desarrollado en este proyecto, validando su correcto funcionamiento y midiendo su rendimiento, comparando las heurísticas propuestas entre ellas.

Palabras clave: Funciones como servicio, FaaS, Serverless, Computación en la nube, Aislamiento

Abstract

The proposed project has consisted in the making of a function as a service (*FaaS*) execution platform, as it constitutes the central piece of the serverless computing paradigm. This platform takes as a priority the isolation between the different function invocations, in order to assure the determinism in its results, as well as the security for the users' data. The project has progressed the following way. The current open source *FaaS* platforms have been studied, in order to analyze them and identify their potential improvement aspects. Taking those aspects as a reference, specifically the isolation, and other ones regarding the fundamental concept of a *FaaS*, as the performance and the high availability, a platform architecture design has been done. This design has served as a guide for the development of a prototype, that fulfills the functionalities related to a *FaaS* system and it's configurable, allowing the programming of new heuristics for the function loading and execution. Lastly, the system developed on this project has been tested, validating its correct behavior and measuring its performance, comparing the proposed heuristics between them.

Key words: Functions as a service, FaaS, Serverless, Cloud computing, Isolation

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Históricamente, el desarrollo de *software* ha atravesado muchos paradigmas a nivel de diseño e implementación. Desde el uso de *mainframes* hasta arquitecturas basadas en *microservicios*, pasando por alternativas monolíticas o divididas en capas. Estos cambios venían impulsados por los avances en las tecnologías subyacentes, haciendo que, con cada variación de los modelos se obtuvieran cada vez soluciones más eficientes (ya fuera teniendo el cuenta el rendimiento, coste, utilización de recursos, etc).

Con la introducción de los grandes proveedores de infraestructura *Cloud* el desarrollo de *software* se ha visto afectado, siendo posible el despliegue de aplicaciones y servicios sobre los recursos que éstos proporcionan. Esta migración al *Cloud* podía ser ventajosa para algunos usuarios (empresas, fundaciones, particulares, etc), pues algunas características como el sistema de pago por uso o la gran disponibilidad de estos proveedores podían suponer una mejor alternativa su anterior sistema *on-premises*. Esta migración se podía realizar de manera sencilla, pues el proveedor de infraestructura pone a disposición del usuario máquinas virtuales, de manera que tan solo hay que mover el *software* que se está ejecutando en máquinas en los servidores del usuario a estas máquinas virtuales del proveedor *Cloud*. Nos referiremos a esta arquitectura como *serverful computing* [2]. Esta forma de funcionar pone a cargo del mantenimiento de la infraestructura al proveedor, pero deja a cargo del usuario todo el resto del *stack*. En contraposición a esta arquitectura, encontramos aquella que se hace llamar *serverless computing*.

El concepto de *serverless computing* hace referencia a un modelo de programación y una arquitectura basados en la ejecución de fragmentos de código, haciendo que el programador no tenga ningún control de los recursos sobre los que el código corre. Es de ahí de donde proviene el término, pues sigue habiendo servidores en este modelo, pero su gestión ya no depende del usuario. Es un modelo pensado para funcionar en un sistema *Cloud*, pues abstrae al usuario de la infraestructura subyacente. Sin embargo, alguien se tiene que hacer cargo de esa infraestructura, es decir, de aprovisionar los recursos necesarios para el correcto funcionamiento del código. De esta manera, son los grandes proveedores de servicios *Cloud*, como *Amazon Web Services*, *Google Cloud*, *IBM Cloud* o *Microsoft Azure* los que ya ofrecen herramientas que forman parte del ecosistema *serverless*, y con las que se pueden desarrollar aplicaciones de estas características. Sin embargo, existen algunas alternativas para *clusters*/*Clouds* privados (*on premises*) que funcionan de la misma manera, como son *OpenFaas* y *OpenLambda* entre otras. En este último caso, corresponde al administrador del *Cloud* privado la instalación de estas herramientas, así como la propia gestión de los servidores.

Hasta este punto se ha hablado de ecosistema *serverless* , pues a efectos prácticos debe estar formado por varias herramientas para poder emular el funcionamiento clásico de cualquier *software* . No obstante, su componente central, aquello que podríamos conside-

rar como su unidad básica de construcción es el concepto de *Function-as-a-Service* o *FaaS*. Una aplicación diseñada al completo en el modelo *serverless* hará uso de herramientas como bases de datos distribuidas y auto-gestionadas (*DynamoDB*), almacenes de objetos (*S3*), sistemas de colas (*SQS*) o *API*s, lo que podría estar comprendido en el concepto de *Backend-as-a-Service* (*BaaS*). La pieza que une todas estas herramientas es el *FaaS*, pues será la que se encargue de gestionar la lógica de la aplicación a desarrollar, en la que se realice la computación necesaria y la que, finalmente, contendrá el código del usuario.

Para conocer mejor el funcionamiento de un *FaaS*, se pueden tomar dos puntos de vista, el del desarrollador o usuario y el del proveedor del servicio. La aproximación del usuario a la herramienta va a ser simple. Para que las funciones desarrolladas se ejecuten, se requerirá el código, las dependencias utilizadas y un *trigger* o disparador de la función, aunque los requisitos pueden variar según el proveedor. Este disparador normalmente será un evento generado por otra herramienta del ecosistema *serverless* , como un cambio en la base de datos o una llamada desde otra función. En el lado del proveedor encontraremos una lógica mucho más compleja, pues será la parte encargada de correr el código proporcionado por el usuario, adaptándose a la carga que reciba; y controlar los tiempos para aplicar el cobro correspondiente. Para ello, el proveedor tiene que exponer una *API* o proporcionar un sistema para el registro e invocación de funciones, aplicar técnicas de escalado y elasticidad que se adapten a la demanda intentando optimizar la gestión de sus recursos, teniendo en cuenta cuestiones de seguridad, latencia y eficiencia computacional, entre otras; y realizar mediciones de tiempo de ejecución precisas.

Una vez explicado el concepto de *FaaS*, se tiene una visión más clara de por qué son los grandes proveedores de infraestructura *Cloud* los que mejor van a poder ofrecer este servicio. Se puede incluso cuestionar si tiene sentido un *FaaS on premises*, ya que son los propios administradores del sistema los que se van a hacer cargo de proveer los recursos. En el caso de haber un pico de demanda que supere la capacidad computacional del *cluster* sobre el que está desplegado no se va a poder dar el servicio correspondiente, pues no tenemos *infinitos* recursos como en el *Cloud* público. Algunos de los argumentos a favor son la *multi tenancy* o la facilidad de uso. El primero alude al uso de los recursos por más de un usuario. Al proveer esta capa de abstracción por encima del *cluster* (o las máquinas virtuales instaladas en él), es muy sencillo realizar ejecuciones de diferentes usuarios sobre la misma infraestructura, pues cada usuario tiene sus funciones propias, y éstas están aisladas entre ellas. En cuanto a la facilidad de uso, y tal y como se comentaba en el párrafo anterior, resulta sencillo el despliegue y le ejecución de código en un *FaaS*.

Para paliar algunos de los inconvenientes que existen en los despliegues *on premises* de aplicaciones con arquitectura *serverless* existe en concepto de *Cloud* mixto, es decir, la automatización del despliegue de los recursos necesarios en un proveedor de *Cloud* público en el caso de que la demanda supere cierto umbral. Esta solución se debería tener en cuenta, pues el usuario utilizará sus recursos al máximo, y en el caso de que no se pueda abastecer la carga demandada se hará un uso del *Cloud* público, ofreciendo siempre una alta disponibilidad.

La arquitectura *serverless* y, más concretamente, el concepto de *FaaS* suponen un buen campo de estudio, dada su creciente popularidad y sus innovaciones conceptuales. Si bien las soluciones encontradas en el *Cloud* público constituyen la punta de lanza de la tecnología, su estudio es complicado, pues no es posible acceder a su código y su diseño interno. Es por eso que este trabajo tomará como punto de partida el estudio de las alternativas *open source*.

1.1 Motivación

Realicé el Máster Universitario en Computación en la Nube y de Altas Prestaciones mientras trabajaba en el departamento de sistemas distribuidos en el Instituto Tecnológico de Informática[1]. La concepción del proyecto se produjo en estas circunstancias, dando pie a poder realizar un desarrollo en el contexto del *ITI*. En cuanto al tema del trabajo, dentro del temario del máster estudiamos el concepto de *FaaS*, y llegamos a desarrollar un prototipo muy simple que se ajustaba a dicha funcionalidad. Me pareció un proyecto interesante, por lo que decidí orientar mi trabajo hacia este tipo de sistemas, buscando algún punto de vista diferente que pudiera mejorar las soluciones existentes de alguna manera.

El paradigma de computación *serverless* se ha ido popularizando en los últimos años, convirtiéndose en una opción viable, y en ocasiones más rentable que las arquitecturas tradicionales. Esto, sumado a su relativamente corto tiempo de vida, lo convierte en un campo de estudio interesante.

Desde un primer momento, el tratamiento que dan los sistemas existentes al aislamiento de los datos entre diferentes ejecuciones de funciones no me parecía el más adecuado. La identificación de este vacío supone un buen punto de partida para el trabajo, pues guía todo el proceso de documentación y desarrollo del proyecto hacia un punto concreto y definido.

A nivel personal, pienso que el desarrollo de este Trabajo de Fin de Máster será muy beneficioso para mí, pues me permitirá ampliar mi nivel de conocimiento sobre el tema, así como realizar un desarrollo de una plataforma desde cero, proceso en el cuál siempre se obtiene algún tipo de aprendizaje.

1.2 Objetivos

El objetivo principal del proyecto es el diseño y desarrollo de una plataforma *FaaS* que implemente unas políticas de aislamiento fuertes entre las ejecuciones de las funciones. Este objetivo principal se puede desglosar en objetivos más específicos:

- Identificar los problemas de aislamiento que presentan las soluciones actuales y proponer una solución.
- Diseñar un sistema que integre la solución propuesta.
- Realizar un desarrollo del sistema diseñado a modo de prueba de concepto o prototipo, que resuelva los problemas de aislamiento planteados.
- Probar el sistema implementado, tanto para validarlo como para medir su rendimiento.

El resultado tangible del proyecto será el prototipo desarrollado, siendo estos objetivos las respectivas fases de su desarrollo. El cumplimiento de estos puntos específicos en conjunto supondrá el cumplimiento del objetivo final.

[1] https://www.iti.es/

1.3 Metodología

La metodología empleada sigue el patrón marcado por los objetivos. El proyecto comenzó con el estudio de referencias académicas, así como de soluciones existentes, con el propósito de contextualizar el trabajo y adquirir conocimientos sobre la arquitectura y los conceptos fundamentales propios de la tecnología estudiada. A continuación, tomando como referencia la información adquirida, se realizó un diseño exhaustivo de la plataforma a desarrollar, centrado sobre todo en el aislamiento, pero sin ignorar aspectos tan relevantes como el rendimiento o la gestión de recursos. Seguidamente, se pasó a la fase de desarrollo. Se realizó una implementación dentro del alcance temporal del proyecto, utilizando una metodología basada en tareas. Por último se realizaron pruebas sobre la solución desarrollada, con el fin tanto de validarla como de comprobar su rendimiento.

Por otro lado, dentro de la planificación temporal del proyecto, se han concertado reuniones periódicas con el tutor, con el fin de discutir diversas cuestiones, tales como decisiones de diseño o tecnologías a utilizar.

1.4 Estructura de la memoria

La memoria seguirá la siguiente estructura. Se comenzará con el estado del arte en el capítulo 2, donde se realizará un estudio de las plataformas *FaaS open source* existentes, encontrando sus puntos de mejora. A continuación, en el capítulo 3 se realizará un análisis fundamental de este tipo de plataformas y se planteará el problema a solucionar. Los capítulos 4 y 5 expondrán el diseño de un sistema que solucione dicho problema, así como la implementación realizada en base a dicho diseño, respectivamente. Se seguirá con las pruebas realizadas sobre la plataforma desarrollada en el capítulo 6, con el fin de comprobar su correcto funcionamiento, así como su rendimiento. Se realizará en el capítulo 7 una conclusión del trabajo, recuperando los objetivos planteados y argumentado si se han cumplido y de qué manera. Para finalizar, se propondrán cuestiones que han quedado pendientes o mejoras que se pueden llevar a cabo a modo de trabajos futuros en el capítulo 8.

CAPÍTULO 2
Estado del arte

A la hora de plantear el diseño de un *FaaS*, se ha considerado interesante tener en cuenta algunas tecnologías existentes. Esto permite conocer cómo otros profesionales han planteado soluciones al problema que abordamos. La exploración de alternativas se divide en dos grupos: plataformas de código abierto (*open source*) y plataformas de código privativo.

2.1 Plataformas *open source*

Las plataformas *open source* son las más interesantes de analizar desde un punto de vista práctico, pues, si bien no tienen una base de usuarios tan extensa como su contrapartida privada, suelen ofrecer información más detallada acerca de su comportamiento. Esta información se nos proporciona gracias una documentación rica y la posibilidad de echar un vistazo al código fuente, así como con la opción a realizar un despliegue de la tecnología en cuestión en nuestros dispositivos y realizar las mediciones oportunas.

Con el fin de conseguir un muestreo suficiente, se han elegido cuatro plataformas en base a su popularidad (número de estrellas en GitHub). Las plataformas en cuestión son: *OpenFaaS*[8], *Apache OpenWhisk*[9], *Kubeless*[10] y *Fission*[11].

2.1.1. OpenFaaS

OpenFaaS es un proyecto independiente creado por Alex Ellis [1] a finales de 2016. Goza de un gran soporte de la comunidad, con contribuciones muy frecuentes y un gran número de colaboradores. La figura 2.1 muestra un esquema de su arquitectura, así como de las tecnologías que lo componen internamente. Dentro de este esquema podemos diferenciar los siguientes elementos:

[1] https://www.alexellis.io/

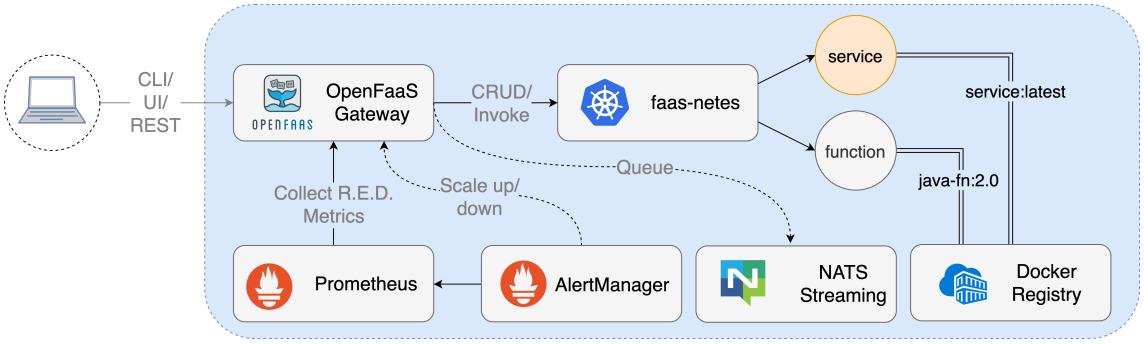


Figura 2.1: Flujo de datos conceptual de *OpenFaaS*

- **OpenFaaS Gateway:** Se trata de la *API REST* que expone la plataforma para el registro e invocación de las funciones. Tal y como muestra la figura, *OpenFaaS* proporciona tanto un *CLI* (Interfaz por línea de comandos) como una *UI*(a modo de *dashboard*). Sin embargo, estas dos no son las únicas maneras de realizar invocaciones de funciones, pues también se proporcionan una extensiva lista de plataformas con eventos compatibles y la posibilidad de configurar webhooks propios. Todas estas posibilidades se pueden encontrar en el apartado referido a los *triggers* en la documentación [2].

- **Prometheus**[3]**:** Recoge métricas del estado de la plataforma y de las funciones, tales como los tiempos de ejecución o el número de réplicas activas en un determinado momento, y las almacena. La documentación también sugiere su fácil integración con *Grafana*[4], para la visualización y monitorización de los datos.

- **AlertManager:** Utiliza las estadísticas de *Prometheus* para escalar automáticamente las funciones. Las políticas de auto escalado son configurables, aunque *OpenFaaS* provee sus heurísticas propias en caso de que el usuario quiera utilizarlas.

  Una característica de los *FaaS* es su capacidad para escalar una función a cero instancias de ejecución, de esa manera no consume ningún tipo de recurso aún estando disponible siempre para ser llamada. *OpenFaaS* proporciona esta funcionalidad, aunque requiere de configuración adicional para ello [5].

- **NATS** [6]**:** Cola para llamadas asíncronas.

- **Faas-netes:** Orquestador de *Docker* en que se desplegarán las funciones. Es el motor encargado de invocar y tumbar las instancias y replicas según ordene el AlertManager. La documentación recomienda utilizar *Kubernetes* en entornos de producción, aunque también existe la opción de hacer despliegues mediante *Docker Swarm*.

- **Function:** La unidad mínima de ejecución. Cada una de las instancias de los programas subidos por el usuario. Corresponden a imágenes de *Docker*, lo que no las ata a un lenguaje específico, es decir, siguiendo el formato establecido, es posible crear funciones en cualquier lenguaje. En el momento en que sean invocadas se descargarán del *Docker Registry*, y cada una de las llamadas que se hagan a partir de este momento será enrutada hacia una de las réplicas disponible, que la ejecutará y devolverá los resultados.

[2] https://docs.openfaas.com/reference/triggers/
[3] https://prometheus.io/
[4] https://grafana.com/
[5] https://docs.openfaas.com/architecture/autoscaling
[6] https://nats.io/

Las funciones en *OpenFaaS* son persistentes en el tiempo, es decir, una sola instancia puede recibir múltiples llamadas. La creación y eliminación de réplicas se basará solamente en la heurística marcada por el AlertManager.

- **Service:** Se entiende como servicio de *Kubernetes*, es decir, un conjunto de instancias que se comunican entre ellas y actúan como bloque. En el caso de desplegar *OpenFaaS* en con otro orquestador de *Docker*, esta función no estaría disponible.

- **Docker Registry:** Repositorio de imágenes de *Docker*. Es el encargado de almacenar cada una de las funciones que los usuarios registren, y será el lugar donde el orquestador irá a buscar las imágenes correspondientes en el momento en el que se produzcan las invocaciones.

Una vez explicado el *stack* de la plataforma, siguen quedando algunas cuestiones básicas por comentar que son de ayuda para comprender su funcionamiento. A continuación se explicarán brevemente:

En cuando a la seguridad, *OpenFaaS* ofrece gestión de secretos y autenticación mediante el estándar *OAuth2* [7]. Existe también la posibilidad de tener múltiples usuarios, limitando a cada uno de ellos a hacer uso de únicamente las funciones que ha desplegado, proporcionando el aislamiento consecuente.

La entrada y salida de datos se realiza haciendo uso de objetos *JSON*. A la hora de registrar la función se especifican los datos de entrada y de salida, a los cuales se tienen que adecuar cada una de las llamadas que se realicen.

Por último, para la instanciación de las funciones se utiliza *Watchdog*[8], una herramienta que sirve como *wrapper* de las funciones y gestiona su paso de argumentos. Existen dos versiones del *software* , la clásica, que realiza un *fork* del proceso en cada invocación y una nueva, que mantiene el proceso abierto y se comunica mediante *HTTP*.

2.1.2. Apache OpenWhisk

OpenWhisk es una plataform*a que nace a principios en 2015 como un pequeño proyecto de* IBM Research. A finales de 2016 fue admitida en el *Apache Software Foundation Incubator* [9], a la vez que se hacía pública. Su principal contribuyente es Rodric Rabbah [10], uno de sus creadores.

[7] https://oauth.net/2/
[8] https://docs.openfaas.com/architecture/watchdog/
[9] http://incubator.apache.org/
[10] https://rabbah.io/

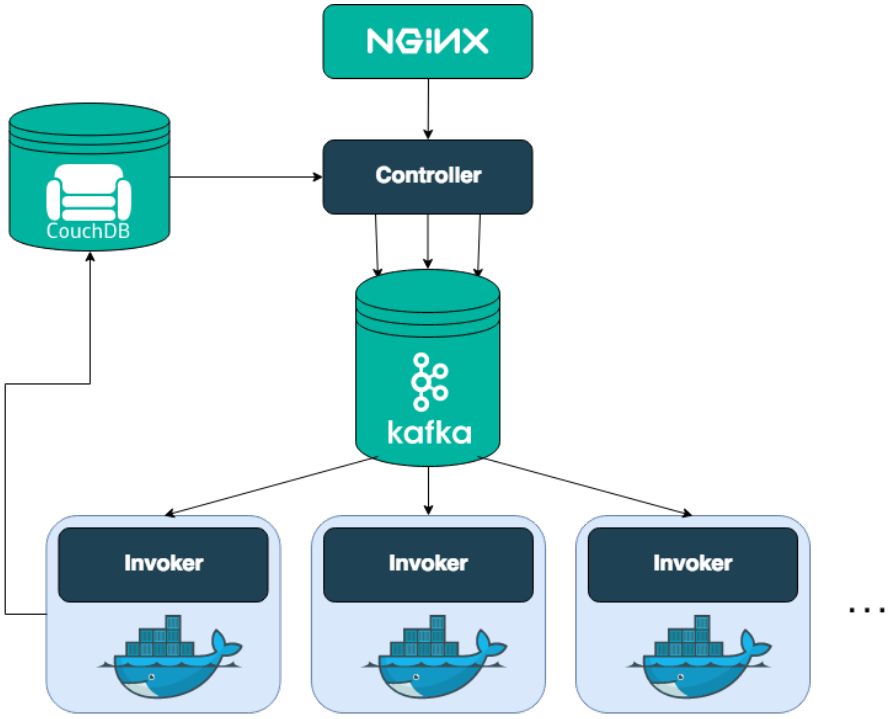


Figura 2.2: Flujo de procesamiento de *OpenWhisk*

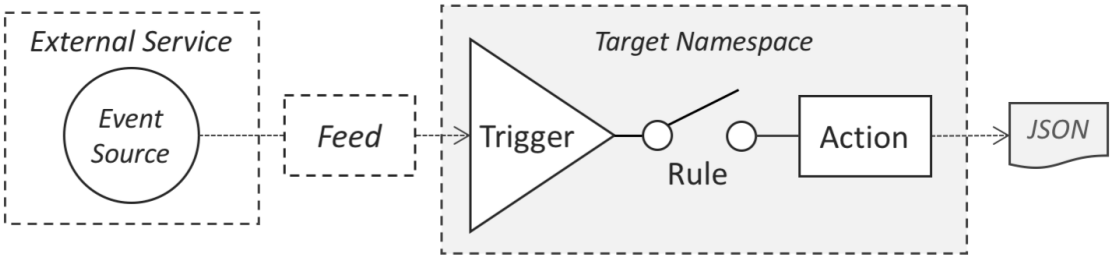


Figura 2.3: Modelo de programación de *OpenWhisk*

Para entender su funcionamiento, procederemos de manera análoga a la subsección 2.1.1, desgranando cada uno de los componentes que aparecen en las figuras 2.2 y 2.3, relacionándolas entre ellas. *OpenWhisk*, al igual que *OpenFaaS*, utiliza muchas tecnologías subyacentes, es por eso que el análisis se hará desde el punto de vista de los elementos que componen el *stack*:

- **Nginx** [11]**:** Se trata de el *gateway* de la plataforma. Es el primer punto de entrada y funciona como un *proxy* inverso, para realizar algunas gestiones como terminación *SSL*.
- **Controller:** Es el mecanismo encargado de recibir las peticiones *CRUD* de la plataforma, todas las llamadas destinadas a crear, invocar o llamar acciones, así como el registro y control de usuarios se realizan en este punto. Tal y como sucede en *OpenFaaS*, se puede acceder a él mediante una interfaz web o bien mediante un *CLI* que proporciona. Comparándolo con la figura 2.2, el controller se encargaría de los apartados de *trigger* y *rule*, ya que es el que recibe los eventos del exterior, los interpreta y pide el lanzamiento de las acciones. Cabe destacar que los *triggers*

[11] https://www.nginx.com/

y reglas no están acoplados, es decir, se declaran disparadores por un lado y se les asocia una regla creada a parte, pudiéndose más tarde separar y asociar a otros disparadores.

El controlador también realiza un balanceo de carga cuando va a hacer las llamadas a las acciones, pues conoce el estado de los *invokers*, y puede decidir qué réplica es más óptima para realizar el trabajo requerido.

- **CouchDB** [12]**:** Es la base de datos del sistema. Se encarga de almacenar los datos y permisos de los usuarios, pues, al igual que sucedía con *OpenFaaS*, *OpenWhisk* es un sistema que permite la *multi tenancy*, por lo que cada usuario sólo podrá lanzar las funciones para las cuales tenga permiso.

  Otra de sus funciones principales es almacenar el código de las funciones, llamadas acciones en *OpenWhisk*. Este código es el que registra el usuario cuando da de alta la acción, y se pide cada vez que se realiza una llamada, pues con él se creará la imagen de *Docker* para su ejecución. Cuando la acción termine, el resultado se almacenará también en *CouchDB*.

- **Kafka** [13]**:** Actúa como cola de mensajes, y como *buffer* entre el controlador y los *invokers*. En caso de fallo de instancias, *Kafka* se encarga de mantener los datos de la invocación, para que no se pierda y se pueda reintentar. En el momento en que la llamada llega a la cola de *Kafka*, al usuario que realizó la petición se le devuelve un ActivationId, con el que luego comprobará el resultado de su ejecución en *CouchDB*.

- **Invoker:** Es la pieza sobre la cual se invocan las acciones. Cuando se realiza la invocación de un acción, el *invoker* coge el código y los parámetros e instancia un contenedor de *Docker* para realizar la ejecución. La forma de levantar este contenedor puede diferir según la configuración que se aplique[14]. El *invoker* será el responsable de replicar las acciones muy solicitadas, así como de tumbar las que no reciban tráfico.

Al igual que sucedía en el apartado de *OpenFaaS*, una visión del *stack* y del modelo de programación no es suficiente para comprender todas las funcionalidades de la plataforma. A continuación se comentarán algunas de las más relevantes:

OpenWhisk permite el uso de muchos lenguajes de programación, para los cuales tiene imágenes predefinidas. No obstante, el usuario puede crear sus propias imágenes de *Docker*, y con ellas realizar ejecuciones en cualquier lenguaje arbitrario, siempre que sea capaz de cumplir los requisitos de entrada y salida por *JSON* que marca *OpenWhisk*. En cuanto al tema de aislamiento, *OpenWhisk* opta también por un sistema *multi tenancy*, lo que previene errores de privacidad de los datos, permitiendo a su vez más un control menos severo en los parámetros de las acciones, pues si una se cae, solo perjudica a su creador. Habría que analizar si realmente el aislamiento que proporciona *Docker* (y *Kubernetes* en el caso de que los *invokers* se desplieguen de esa manera) es suficiente, con el fin de no ver comprometido ninguno de los entornos independientes de los usuarios.

2.1.3. Fission

Fission es un proyecto *open source* creado en agosto de 2016. Es una plataforma centrada en el desarrollo de una arquitectura del tipo *FaaS* sobre *Kubernetes*, y cuenta con un

[12] https://couchdb.apache.org/
[13] https://kafka.apache.org/
[14] https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/configurationChoices.md#invoker-container-factory

gran número de colaboradores y un gran flujo de actividad. Para comprender su funcionamiento se analizará su modelo de programación y su arquitectura, tal y como se venía haciendo en los apartados anteriores.

El modelo de programación está definido por los siguientes elementos: Funciones, Entornos, *triggers*, Archivos y Especificaciones:

- **Funciones:** Es aquello que *Fission* ejecuta, suele ser un módulo con un *entrypoint*, siendo este *entrypoint* una función con una cierta interfaz de entrada y salida.
- **Entornos:** Son las partes que dependen del lenguaje en *Fission*. Un entorno contiene el código justo para correr una función. Como la invocación de funciones se realiza por *HTTP*, un entorno suele ser un servidor *HTTP* con un cargador para inyectar la función. *Fission* permite el uso de entornos creados por el usuario.
- **Triggers:** Es una asignación entre un evento y la invocación de una función. *Fission* permite el uso de *triggers* basados en peticiones *HTTP*, lanzados mediante temporizadores, basados en colas de mensajes (como *Kafka*, *NATS*, etc.), así como basados en eventos de *Kubernetes Watch*.
- **Archivos:** Un archivo es un fichero *zip* que contiene código o binarios compilados. Se dividen en archivos de despliegue (o funciones ejecutables) y en archivos fuente (Dependencias). Se agrupan en paquetes.
- **Especificaciones:** Son ficheros *YAML* que contienen los elementos anteriormente mencionados y sus interacciones. Sirven como manifiestos para indicar a la plataforma cómo tratar dichos componentes.

En cuanto a la arquitectura de la plataforma, se identifican los siguientes elementos:

- **Controller:** El controlador contiene la *API* de *CRUD* de las funciones, *triggers*, entornos, etc. Es el elemento con el que el usuario se comunica, cosa que puede hacer bien por el *CLI* que *Fission* proporciona o haciendo uso de llamadas *HTTP*.
- **Router:** El *router* reenvía las solicitudes *HTTP* a los *pods* de las funciones. Si no hay un *pod* en ejecución para una función, solicita uno al *executor*, mientras mantiene la solicitud. Una vez que la función esté lista enviará la solicitud al pod. El *router* es el único componente sin estado y puede escalar si es necesario, según la carga.
- **Executor:** Se trata del elemento más interesante de la arquitectura, pues es el que controla el ciclo de vida de las funciones. Cuando una petición llega al *executor*, éste comprueba si la función solicitada se encuentra en caché. En caso de que no sea así, él será el encargado de recopilar toda la información necesaria para volver a invocar la función en cuestión.

  Fission provee dos tipos de *executors*: *PoolManager* y *New-Deployment*:

  - **PoolManager:**

    El modelo de *executor PoolManager* gestiona *pools* de *pods* genéricos (*Generic Pod*) y de *pods* especializados, que ya contienen la función y pueden ser ejecutados (*Function Pod*).

    La *pool* inicial de contenedores calientes puede ser configurada. La gestión de recursos se especifica a nivel de entorno y es heredada por los *pods* especializados, es decir una vez un *pod* es invocado no se puede reconfigurar.

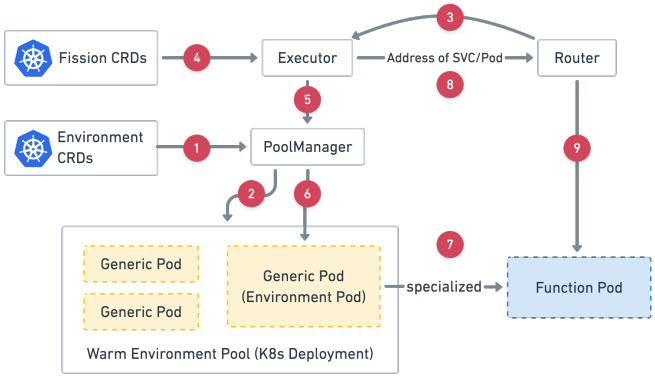


Figura 2.4: Diagrama del *executor PoolManager*

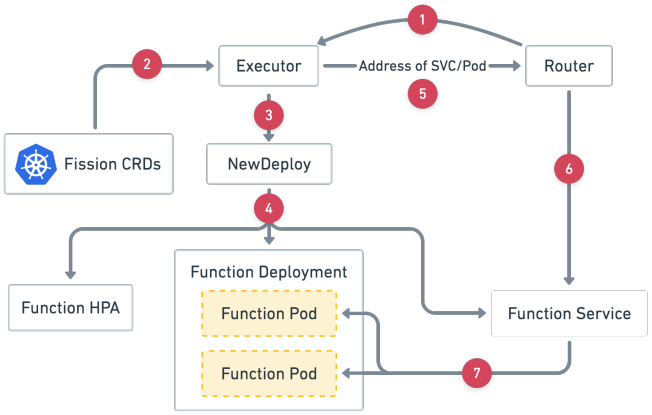


Figura 2.5: Diagrama del *executor New-Deployment*

El funcionamiento es el siguiente, tomando como base los números que se encuentran en la figura 2.4:

Se comprueba si los entornos han cambiado (1). En el caso de ser así, se actualizaría la *pool* de contenedores genéricos, respetando los cambios en los entornos (2). El *router* solicita la dirección de servicio de una función (3). El *executor* extrae la información de la función del correspondiente *CRD* (4). Se invoca al *PoolManager* para que despliegue un *pod* con la función (5). Éste elige un *pod* genérico con el entorno adecuado (6) y lo especializa, añadiendo los elementos necesarios para convertirlo en un *pod* de función (7). La dirección de servicio se devuelve al *router* (8), donde a partir de ahora dirigirá las peticiones (9).

PoolManager es una buena opción a elegir en el caso de que las funciones tengan un ciclo de vida corto y necesiten un tiempo de *cold start* muy pequeño, pues el hecho de mantener contenedores genéricos preparados ofrece un tiempo de respuesta muy rápido.

Sin embargo este sistema también presenta sus limitaciones, pues solo permite una instancia de la misma función corriendo al mismo tiempo, es decir no es posible aplicar replicación. En cuanto al escalado a cero, aunque es posible tener una *pool* de *pods* vacía, no se puede considerar un escalado a cero real, pues hay *pods* genéricos consumiendo recursos.

- **New-Deployment:**
  El modelo de *executor New-Deployment* crea un despliegue de *Kubernetes*, un servicio y un auto-escalador de *pods* horizontales (*HPA*), lo que lo hace adecuado para la ejecución de funciones que gestionan un tráfico masivo.
  Esto permite el autoescalado de las funciones, creando nuevos *pods* a modo de réplicas y distribuyendo la carga entre éstas. Los requerimientos de recursos pueden ser especificados a nivel de función, y sobrescribirán los designados a nivel de entorno. El numero de réplicas mínimo se puede ajustar en la configuración, y será el que se despliegue al inicio de la ejecución. El *HPA* se encargará de invocar tantas réplicas como marque la política de escalado que se haya especificado, así como de acabar con los *pods* que sean innecesarios en periodos de poca carga.
  El diagrama de su funcionamiento se puede ver con un mayor nivel de detalle en la figura 2.5:
  El *router* pide la dirección de servicio de una función (1). El *executor* devuelve la información de la función que encuentra en el *CRD* (2). Se invoca *NewDeploy* para desplegar los *pods* de función (3). *NewDeploy* crea tres servicios de *Kubernetes*: *Deployment*, *Service* y el *HPA* (4). La dirección del servicio es devuelta al *router* (5). El *router* redirige las peticiones a la dirección que se le ha devuelto (6), que se trata del servicio que actuará como balanceador de carga para los *pods* de función (7).
  Este tipo de *executor* sacrifica el tiempo de *cold start* por la capacidad de servir funciones con un tráfico masivo. Esto se puede aliviar exigiendo que el número mínimo de réplicas sea mayor que cero, lo que haría que, en caso de que el tráfico fuera nulo, siempre hubiera una instancia al menos lista para servir las peticiones. Puede ser un escenario a considerar en el caso de que el tiempo de respuesta sea más importante que el consumo de recursos.

Tipo de *executor*	Min. replicas	Latencia	Coste sin tráfico
NewDeploy	0	Alta	Muy bajo, los *pods* se eliminan al cierto tiempo
NewDeploy	>0	Baja	Medio, un número de *pods* siempre está activo
PoolMgr	0	Baja	Bajo, la *pool* de *pods* siempre está activa

Tabla 2.1: Comparativa para la elección del *executor*.

En resumen, la elección del *executor* va a depender íntegramente del caso de uso, tal y como se muestra en la tabla 2.1. Dependiendo de la latencia, coste de recursos y volumen de tráfico será interesante elegir un tipo de *executor* o otro, así como sus respectivas configuraciones.

- **Builder Manager:** Se encarga de vigilar los cambios en el *CRD* de paquetes y entornos, además de gestionar la creación de las funciones. Cuando cualquier entorno cambia, bien contenga archivos ejecutables o dependencias, se encargará de hacer los cambios necesarios para actualizar todos los recursos aplicando las últimas versiones.

- **Storage service:** Es el sistema de almacenamiento de la plataforma. Equivaldría *CouchDB* en *OpenWhisk*, haciéndose cargo de guardar y devolver la información necesaria cuando se requiera.

Existen mas elementos que pueden formar parte de la arquitectura de un despligue de *Fission*, aunque no es obligatorio su uso para un funcionamiento correcto de la plataforma. Tales elementos son: un *logger*, para mantener un registro de todo lo sucedido y poder revisarlo en caso de que se produzcan fallos; *Kubewatcher*, un servicio de monitorización de *Kubernetes*, que permite utilizar los propios eventos que genera como *triggers*; Una cola de mensajes interna, para hacer uso de sus eventos y disparar funciones gracias a ellos; y, por último, un temporizador, con el mismo propósito que los dos elementos anteriormente mencionados.

2.1.4. Kubeless

Kubeless es la propuesta de *Bitnami* [15] para la creación de una plataforma *FaaS*. Empezó en noviembre de 2016, y ha mantenido un ritmo estable de contribuciones hasta la fecha.

Su arquitectura es muy similar a la vista en *OpenFaaS*, por lo que se intuye en la documentación. No tienen una guía concreta ni un esquema oficial que muestre los elementos que componen su *stack* de forma clara.

Se puede dividir su modelo de programación en tres elementos: funciones, *triggers* y *runtimes*. Tal y como se viene comentando en las tecnologías expuestas previamente, las funciones son el elemento ejecutable, el código que se correrá una vez sea llamado. Los *triggers* son el evento que activará las llamadas a las funciones, *Kubeless* permite *triggers HTTP*, basados en cronjobs y aquellos generados por sus servicios de colas (*Kafka*, *NATS*). En cuanto a los *runtimes*, serán entornos en cierto lenguaje preparados para inyectar el código de la función antes de ejecutarla. *Kubeless* soporta algunos lenguajes por defecto, aunque es posible crear *runtimes* para nuevos lenguajes siempre que se siga el modelo que utiliza.

Kubeless utiliza *Kubernetes* como orquestador, haciendo uso de muchos de sus servicios propios para ciertas funciones. Estos servicios son, por ejemplo *Ingress* como *API gateway* y para controlar temas de seguridad y autorización, ya que *Kubeless* es una plataforma *multi tenancy*; o el HPA (*Horizontal pod autoscaler*) como controlador para el autoescalado de las funciones. Ambos se pueden configurar y ofrecen una gran versatilidad en el control de su funcionamiento.

Para el almacenamiento tanto de los *runtimes* como de funciones se utiliza un *Docker Registry*, que puede ser tanto privado como público. En cuanto a la entrada y salida, las funciones reciben la entrada en *JSON*, y la devuelven de la misma manera. Las dependecias irán ligadas al *runtime*, es decir, para gastar librerías propias o que no estén soportadas de primera mano por *Kubeless* es necesario crear nuevos *runtimes*.

En cuanto al aislamiento de los datos, sigue el mismo modelo que las tecnologías anteriores, el *multi tenancy* como escudo para los posibles escapes de información que se puedan generar.

2.2 Plataformas de código privado

Las plataformas de código privado son las primeras que repararon en la arquitectura *serverless* y el concepto de *FaaS*. Si bien su estudio de funcionamiento interno y monitorización de cada una de sus partes va a ser complicado, por la poca información que

[15] https://bitnami.com/

proveen en cuanto a estos aspectos [5], sigue siendo interesante conocerlas y mencionarlas, por su relevancia en cuanto a volumen de tráfico y rendimiento.

Una característica fundamental de cualquier *FaaS* que nos puede ofrecer un proveedor de *Cloud* público es inherente a el propio hecho de estar dentro de un proveedor de *Cloud* público. Cualquiera de las opciones que mencionaremos a continuación es un elemento dentro del gran ecosistema que es uno de estos proveedores. Esto supone una gran ventaja para la comunicación entre el *FaaS* y el resto de servicios, pues el *Cloud* facilita enormemente la integración entre ellos. El almacenamiento de datos lo puede proveer un servicio del *Cloud*., el *API gateway* otro, los *triggers* para las llamadas, etc. Todo el *stack* de componentes que se puedan necesitar podrán ser contratados en el proveedor, con la ventaja de tener unas comunicaciones entre ellos triviales desde el punto de vista del usuario. Sin embargo, esto puede ser perjudicial por otro lado, pues favorece en gran medida el *vendor lock-in*, lo que quiere decir que, en caso de tener que migrar toda la lógica a otro proveedor por cualquier motivo, es posible que sea una tarea ardua, pues habrá que volver a declarar todos y cada uno de los elementos utilizados en el nuevo *Cloud*.

Otra característica que ofrecen los proveedores de *Cloud* privado es la sensación de recursos ilimitados. Al disponer de unos centros de datos masivos, son capaces de soportar una gran carga. Cuando el usuario de un *Cloud* privado realiza el despliegue de una función en su infraestructura sabe que, en el caso de que la carga supere la capacidad del servidor, su función puede llegar a no estar disponible. Esto no va a suceder nunca en los proveedores de *Cloud* público, pues siempre van a tener máquinas disponibles en las que lanzar nuevas instancias de la función en concreto, para asegurarse de que se cumplen los acuerdos de nivel de servicio establecidos.

Algunos de los *FaaS* relevantes, debido a su volumen de usuarios y las compañías que los han desarrollado y los comercializan son: AWS Lambda[16],Google Cloud Functions [17] y Azure Functions [18]. Puesto que su código es privado, y su documentación respecto a características internas escasa, se ha decidido que su estudio queda fuera del dominio de este trabajo.

2.3 Crítica al estado del arte

La arquitectura *serverless* y el concepto de *FaaS* han sido explorados en gran medida, tal y como muestran las soluciones comentadas en las subsecciones anteriores. Existe una gran variedad de plataformas que siguen este modelo y que se encuentran en un estado de madurez suficiente como para ser consideradas una opción válida por los usuarios en fases de producción. Cada plataforma existente tiene sus peculiaridades en cuanto a implementación, tecnologías subyacentes o modelo de programación, entre otros muchos aspectos [4]. Sin embargo, todas tienen como prioridad la optimización de cuatro aspectos, que a continuación serán explicados por orden de prioridad:

- **Rendimiento**. El rendimiento es el aspecto más priorizado en las soluciones *open source* analizadas. El rendimiento de un *FaaS* depende de múltiples factores, y se debe entender dentro del contexto de cada despliegue. Las optimizaciones de rendimiento que se realizan son internas a la plataforma, quedando sujetas a la infraestructura subyacente. Por lo que, al decir que un *FaaS* rinde mejor que otro, hay que comparar bien sus costes teóricos de ejecución o bien despliegues con características de *hardware* y condiciones idénticas.

[16] https://aws.amazon.com/lambda/
[17] https://cloud.google.com/functions
[18] https://azure.microsoft.com/en-us/services/functions/

Considerarlo una prioridad es una aproximación lógica, pues una vez el usuario haya realizado toda la programación necesaria y su flujo de uso esté configurado correctamente, el único aspecto del sistema que le preocupa es lo rápido que se ejecuten sus funciones. De este tiempo de respuesta dependen tanto la fluidez de la interacción con el *FaaS* como el coste monetario de las ejecuciones, si lo hubiera.

- **Facilidad de uso**. La facilidad de uso es una característica inherente a un *FaaS*. Por definición, la arquitectura *serverless* pretende aislar al usuario de cualquier tipo de gestión de la infraestructura sobre la que se ejecuta su código. Por otro lado, para que un usuario considere migrar su lógica a una plataforma nueva, es necesario que ésta proporcione herramientas y mecanismos para que la transición sea lo más sencilla posible.

  Es un aspecto difícil de cuantificar, pero se trata de una prioridad, tanto en las plataformas *open source* como en las privativas, pues todas tienen una guía de uso sencilla de seguir para alguien con una mínima experiencia en programación

- **Alta disponibilidad**. Es una característica que se tiene en cuenta las soluciones *open source* estudiadas. Es interesante considerarla una prioridad, pues aunque los sistemas vistos estén pensados para ser desplegados *on premises*, siempre existe la posibilidad de que se tenga que soportar una carga masiva. La alta disponibilidad va a estar sujeta a algunos factores, como son la replicación, la redundancia y la tolerancia a fallos de los componentes del sistema, entre otros.

  Se trata de un aspecto relevante, porque los flujos de trabajo de los usuarios del sistema se van a ver interrumpidos en caso de sobrecarga o caída.

- **Uso eficiente de recursos**. Se trata de un aspecto que tienen en cuenta algunas de las soluciones estudiadas. Choca un poco con el rendimiento, pues a medida que te deshaces de recursos activos para economizar en el consumo estás sacrificando su uso inmediato en el caso de que fuera necesario.

  La relevancia de este aspecto crece con la escala del sistema, pudiendo ser despreciable en despliegues pequeños *on premises* pero fundamental en despliegues masivos en *Clouds* públicos.

En contraposición, un aspecto que no recibe tanta atención es el **aislamiento de las ejecuciones**. Todas las soluciones analizadas en las secciones anteriores proponen el mismo sistema: la *multi tenancy* como mecanismo de aislamiento frente a una posible fuga de datos entre dos ejecuciones de la misma función o de funciones que hagan uso del mismo contenedor. El mecanismo utilizado es el siguiente: cada usuario registrado tiene a su alcance un conjunto de funciones, que él mismo ha dado de alta. Ese conjunto de funciones nunca va a coincidir con el conjunto de funciones de otro usuario, por lo que nunca se va a dar el caso de que un usuario acceda a los datos residuales de un contenedor antiguo que contuviera la ejecución de una función de otro. No se busca un aislamiento activo entre diferentes ejecuciones de una función o entre diferentes funciones que utilicen el mismo *runtime*, sino que se prima el rendimiento y el aislamiento obtenido es un efecto secundario de esta separación por usuarios.

Si bien esta es una solución válida para muchos casos de uso, presenta un par de puntos de conflicto. En primer lugar, se lleva a cabo una separación por usuario a nivel de recursos. En el caso de que se realice cualquier tipo de precarga, bien de *runtimes* o de funciones, ésta va a ser exclusiva para el usuario propietario, pues puede que dicho entorno de ejecución ya contenga datos o código privado del usuario en cuestión, y quedará fuera del alcance de cualquier otro usuario. Por otro lado, la solución puede resultar indeterminista, devolviendo dos resultados diferentes para dos ejecuciones de la misma

función con los mismos parámetros de entrada. Esto se puede dar en el caso de tener una función que guarde cierta información. La misma función tiene un mecanismo para leer dicha información en caso de encontrarla. El usuario cuando invoca la función no es capaz de controlar si esta se ejecuta en un entorno reciclado de una ejecución anterior o en uno nuevo, por lo que puede obtener, arbitrariamente desde su punto de vista, dos resultados diferentes. Algunos proveedores *Cloud* consideran esto una ventaja, como es el caso de *AWS*[19], pues se puede hacer de forma medianamente controlada, e incluso existen trabajos centrados en la compartición de memoria por parte de diversas funciones [3]. Sin embargo la arquitectura *serverless* proporciona otros mecanismos para el almacenamiento de información, por lo que esta característica excede lo que debería ser el dominio una función.

El trabajo de Stenbom[1] recoge este problema, mostrando los límites a los que hay que llegar para asegurar que una reutilización de un *runtime* previo es segura. En él se puede ver como se alcanza el punto de borrar registros y zonas de memoria del nodo sobre el que se realiza la instanciación del entorno de ejecución, con el fin de poder reutilizar contenedores levantados que proporcionen un aislamiento suficiente.

2.4 Propuesta

Tal y como se expone en la sección 2.3, se ha identificado un vacío respecto al tratamiento del aislamiento entre ejecuciones en las plataformas *FaaS open source*. Este vacío se ha suplido tradicionalmente de forma indirecta gracias a las medidas de seguridad que proporciona la *multi tenancy*. Algunos proyectos han ido más lejos, abordando el problema de primera mano mediante la adaptación plataformas existentes para cumplir con dicho aislamiento.

Lo que se propone en este trabajo es un punto de vista totalmente opuesto, un *FaaS* que toma como punto de partida el aislamiento entre ejecuciones, realizando los compromisos necesarios en otros puntos del sistema. Para ello, se ha elegido el diseño y desarrollo de un prototipo propio frente a la adaptación de un sistema existente. Esta elección viene determinada por diversos factores, como el alcance del proyecto, la legibilidad de la solución final y el control absoluto sobre cada una de las partes que compongan el sistema.

La plataforma a desarrollar debe ser capaz de cumplir las funcionalidades básicas de un *FaaS*, es decir, ser capaz de registrar *runtimes* y funciones, así como de ejecutar dichas funciones. No se deben ignorar el resto de aspectos relevantes de los *FaaS*, recogidos en la sección anterior, aunque se les asigne una prioridad menor.

Por último, se realizarán pruebas de rendimiento sobre el prototipo desarrollado. Si bien el rendimiento de estas pruebas no será comparable al de plataformas existentes y altamente optimizadas, servirá como un punto de partida y una validación del correcto funcionamiento de las diferentes partes del sistema.

[19] https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

CAPÍTULO 3
Análisis del problema

Para poder realizar un juicio pormenorizado de cómo debe ser el sistema a desarrollar, se debe comenzar por el estudio conceptual de lo que es un *FaaS*. De esta manera se puede comprender que piezas son fundamentales y cuáles prescindibles, facilitando así el diseño de cada una de ellas y de su interacción en el sistema final.

3.1 Estructura de un *FaaS*

Cuando se habla de *FaaS* o funciones como servicio se hace referencia a un concepto bastante concreto. Se trata de la ejecución de trozos de código arbitrarios bajo demanda en una plataforma ajena al usuario, sin que éste interactúe de ninguna forma con la infraestructura. Este código debería ser *stateless*, lo que quiere decir que no debería almacenar ni requerir ningún tipo de estado previo. Cada ejecución de dicho código sería independiente y aislada, y, en el caso de querer mantener un estado, éste se situará en un sistema externo al *FaaS*, como una base de datos.

La gran mayoría de *FaaS* se componen de infinidad de elementos, tales como colas o *proxies*, sin embargo, si tomamos como referencia su definición conceptual, estos elementos no son esenciales para su cumplimiento. Para que un *FaaS* se considere como tal debe cumplir únicamente dos funcionalidades básicas: Recibir el código de la función a ejecutar y ejecutar dicho código.

3.1.1. Elementos teóricos

Para poder llevar a cabo ambas funcionalidades, el *FaaS* debe estar compuesto de una serie de elementos a nivel teórico, derivados de las propias funcionalidades. A continuación se expondrán, mencionando la relevancia de cada uno de ellos.

- **Vía de comunicación con el usuario**: Mecanismo mediante el cuál el usuario y el sistema se comunican. Este mecanismo engloba las acciones de envío de funciones, invocación de funciones y obtención de resultados. Estas tres acciones deben estar siempre presentes en un *FaaS*, aunque no tienen por qué pertenecer a la misma vía de comunicación.
- **Entorno de ejecución**: Conjunto de procesos y archivos que, con la configuración adecuada, son capaces de recibir la función y sus argumentos y ejecutarla. Se puede desglosar en muchos elementos, como el *hardware* físico o virtual, el *runtime* o las dependencias de la propia función. Este entorno debe estar correctamente aislado, para garantizar tanto la seguridad del usuario que realiza las ejecuciones como del

sistema que las recibe, en caso de que hubiera algún tipo de intención maliciosa. Por otro lado, el entorno debería carecer de contexto y ser exactamente igual en todas las invocaciones de la misma función, aunque este punto se obvia en muchos sistemas para favorecer el rendimiento.

3.1.2. *FaaS* en la práctica

Siguiendo la definición teórica, puede surgir algún ejemplo curioso, que se adecuaría al modelo de *FaaS* descrito, pero que no es abiertamente considerado como tal:

Una casilla de un *notebook* de *Jupyter*[1] se puede considerar un *FaaS*. La vía de comunicación con el usuario es una interfaz gráfica a modo de *notebook*, permitiendo la subida de código mediante a través del cuadro de texto que proporciona; la ejecución de dicha casilla; y la obtención de resultados, mostrándolos por pantalla. El entorno de ejecución es un *kernel*, que se encarga de recibir dichas funciones y ejecutarlas. La infraestructura sobre la que se ejecuta *Jupyter* y sus casillas puede ser desconocida para el usuario, ocupándose éste únicamente de la programación y ejecución de los fragmentos de código. Como aclaración, es cierto que el uso de un *notebook* de *Jupyter* y sus casillas está muy ligado a aprovechar el contexto que crea el *kernel*, por lo que dos llamadas al mismo código con los mismos parámetros pueden devolver dos resultados diferentes. Esto es comparable a la reutilización de entornos de un *FaaS* tradicional.

El ejemplo previamente descrito sirve para ilustrar que, pese a que teóricamente un *FaaS* puede tomar muchas formas, a la hora de concebirlo se siguen una serie de convenciones, es decir, estos elementos teóricos son puestos en funcionamiento mediante unos mecanismos concretos. Esto es debido a los casos de uso derivados del modelo de programación *serverless* .

Generalmente, el modelo de programación *serverless* va a optar por funciones cortas, repetitivas y con unos niveles de carga difíciles de predecir. Es en este tipo de situaciones donde los usuarios consideran esta arquitectura, pues con procesos más largos, complejos o con una carga más predecible, el aprovisionamiento de infraestructura suele resultar una opción más viable económicamente. *Jupyter* no es la opción más viable para un problema de estas carácterísticas, de la misma forma que un *FaaS* típico no es viable para la exploración de un *dataset*, por ejemplo.

Dada la naturaleza del problema, y teniendo como referencia las soluciones estudiadas en el capítulo 2, la arquitectura de un *FaaS* modélico se compondría de los siguientes elementos:

- **Una API** para servir de vía de comunicación entre el usuario y la plataforma. Esta *API* puede ser accedida de múltiples formas, desde interfaces gráficas a *CLI*s, y permitirá realizar las gestiones necesarias para el correcto funcionamiento del sistema. Un punto a tener en cuenta es la integración con otras herramientas de un ecosistema *serverless*, sobre todo en el punto de la invocación de funciones. Si bien la *API* de un *FaaS* ofrecerá normalmente la posibilidad de realizar una invocación manual de la función, el sistema suele tener otros mecanismos, llamados *triggers*, para realizar dicha invocación automáticamente en base a un evento generado en cualquier parte del ecosistema (ya sea en una herramienta externa o otro *FaaS*). Estos mecanismos pueden incluir canales de servicio o procesos de *polling*. Estos *triggers* no van a ser un elemento definitorio del *FaaS*, aunque suponen una pieza importante, pues gran parte de la funcionalidad e integración del sistema depende de ellos.

[1] https://jupyter.org/

- **Contenedores** para hacer la función de entorno de ejecución. Existen alternativas , aunque suelen ser casos excepcionales. El contenedor como abstracción encaja perfectamente en el rol de entorno de ejecución. Proporciona aislamiento entre diferentes instancias, así como protección para la infraestructura sobre la que se ejecuta. Su ciclo de vida es ágil, favoreciendo el rendimiento de la plataforma y goza de una gran versatilidad, permitiendo cargas tanto ligeras como pesadas. A nivel de replicación y escalado suponen una gran opción también, ya que por naturaleza se integran adecuadamente en ese tipo de situaciones.
- **Estructuras internas complementarias**, que sirvan de apoyo para el flujo de información. No son explícitamente necesarias para el concepto de *FaaS*, pero una solución no tendría una utilidad práctica sin ellas. Son más difíciles de generalizar que los dos componentes anteriores, pues dependen más estrechamente de la implementación de cada sistema. No obstante, siempre suelen aparecer al menos estas dos: una base de datos y un mecanismo de repartición del trabajo. La base de datos tiene la motivación de proporcionar una capa de persistencia para los datos tanto introducidos como resultantes de la plataforma. Por otro lado, los mecanismos de repartición del trabajo son un concepto implícito a un *FaaS*, ya que tiene que existir una forma de que las invocaciones lleguen a los entornos de ejecución y se ejecuten, aunque este proceso se realice de forma trivial. Sin embargo, es una de las partes principales del diseño y una de las que tendrá más peso en el rendimiento del sistema, es por eso que las soluciones estudiadas hacen uso de estructuras externas (tales como colas, balanceadores de carga, etc.) y heurísticas avanzadas para que estos mecanismos funcionen de la forma más óptima posible.

Estos elementos, combinados de la forma correcta, resultarían en una solución válida, utilizable y comparable a nivel estructural con las opciones *open source* disponibles.

3.2 Formalización del problema

Tal y como se expone en la sección 1.2, el objetivo principal del proyecto es el desarrollo de un prototipo de *FaaS*. Una vez sentadas las bases de lo que compone un sistema de ejecución de funciones como servicio (sección 3.1) e identificados los puntos de mejora respecto a los sistemas existentes (sección 2.3), se propone la formalización de los puntos que este proyecto va a tratar, ordenados de mayor a menor prioridad:

- **Aislamiento**: Se ha elegido como punto prioritario, en respuesta a la poca prioridad que se le atribuye en los sistemas existentes. El sistema a obtener debe proporcionar un aislamiento total entre ejecuciones. Es decir, en el momento en el que una invocación de una función concreta llega al entorno de ejecución, éste debe presentarse siempre en un estado idéntico. Esto garantiza una consistencia entre ejecuciones, privacidad entre llamadas y, el última instancia, el determinismo del sistema.
- **Rendimiento**: Tal y como se podía ver en las soluciones estudiadas en el capítulo 2, el rendimiento tiene una gran importancia para el uso de un *FaaS*, y por muy buenas razones. Es por ello que se va a considerar como el segundo punto más prioritario del sistema.
- **Gestión eficiente de recursos**: Tal y como sucede en los sistemas estudiados, la gestión eficiente de recursos suele estar supeditada al rendimiento, colocándola en un orden prioritario inferior. Cierto es que es un punto muy interesante a explorar, y que tiene que existir en mayor o menor medida para no sobrecargar el sistema o desperdiciar el poder de cómputo de la infraestructura.

- **Facilidad de uso**: Como el alcance del proyecto contiene la creación de un prototipo, la facilidad de uso es un punto que no toma tanta relevancia. Se deberán cumplir unos puntos mínimos inherentes a el concepto de *FaaS*, como que el usuario final no debe ser el responsable de controlar la infraestructura sobre la que corre el sistema. Por otro lado, al tratarse de un proyecto que busca una alta versatilidad, la interacción con el sistema puede ser un poco tosca, y quizás confusa para un usuario no experto en el campo.

Esta lista de características servirá como una referencia filosófica para el diseño y desarrollo del sistema final.

3.3 Identificación y análisis de soluciones posibles

Cada uno de los puntos tratados en la sección anterior tiene múltiples formas de ser abordado, a continuación se listarán, con el objetivo de elegir la que mejor se adapte a las necesidades de nuestro proyecto.

3.3.1. Aislamiento

Tal y como se argumentaba en la sección 3.1, el sistema tiene que proporcionar medidas de aislamiento, capaces de proteger la privacidad del usuario, así como a la infraestructura de posible código malicioso. Estas medidas de aislamiento pueden ser obtenidas de muchas formas, dependiendo del nivel de fragmentación que se quiera realizar en el sistema. Cabe tener en cuenta que cada una de las abstracciones que se obtengan con este aislamiento deberá ser capaz de constituir el entorno de ejecución del *FaaS*.

La primera abstracción que surge, tomando un punto de vista general, es un nodo físico. Proporciona unas medidas de aislamiento casi perfectas, pues incluso el *hardware* que intervenga en la ejecución de las funciones no va a ser usado simultáneamente por diferentes invocaciones. Sin embargo, su propuesta roza el absurdo, ya que un nodo físico normal excede en gran medida la capacidad de cómputo que puede requerir una función, lo que supondría un gran malgasto de recursos. Por otro lado, y tomando como referencia las prioridades de este proyecto, las abstracciones elegidas van a tener que ser borradas e instanciadas con gran frecuencia, lo que a nivel de nodo físico supone un gran coste temporal.

Avanzando en la pirámide de la virtualización, la siguiente abstracción encontrada sería la máquina virtual. Al igual que el nodo físico, proporciona unos niveles de aislamiento muy completos, realizando una separación a nivel de sistema operativo. En este punto ya es posible la compartición del *hardware*, lo que supone un gran avance, pues se pueden optimizar mucho mejor los recursos, adaptándolos de una mejor manera a las funciones en concreto. El punto en contra es, al igual que con los nodos físicos, los tiempos de carga. En un sistema con una rotación muy elevada de máquinas virtuales este tiempo puede suponer una gran parte su ciclo de vida, resultando en una pérdida a nivel global de capacidad de cómputo.

La siguiente opción a considerar es la virtualización a nivel de *software* . La abstracción principal a considerar en este punto son los contenedores. Los contenedores permiten tanto la compartición de *hardware* como de sistema operativo. Esto los hace idóneos para la ejecución de funciones en ellos, ya que son lo suficientemente seguros e independientes como para no sufrir una fuga de datos y lo suficientemente ligeros para que su eliminación e invocación no lastren demasiado el rendimiento del sistema.

3.3.2. Rendimiento

A la hora de analizar las diferentes soluciones en lo que respecta al rendimiento de un *FaaS*, hay que tener en cuenta que este rendimiento se ha de medir de forma teórica. Un *FaaS* es un *software* que se ejecuta sobre unos componentes físicos. Si se realizaran las mediciones sin tener ésto en cuenta, no se estaría midiendo el rendimiento del *FaaS* sino el rendimiento del *FaaS* sobre una infraestructura concreta.

Dicho esto, se consideran dos aspectos principales a la hora de medir el rendimiento de un *FaaS*: las estructuras internas del sistema y las políticas de repartición del trabajo. Aunque están estrechamente relacionados, es interesante analizarlos independientemente.

Rendimiento de las estructuras internas del sistema

Se refiere al coste temporal derivado de hacer uso de herramientas no fundamentales pertenecientes al sistema. Estas herramientas suelen ser incorporadas en el sistema como resultado de una funcionalidad extra que se quiere ofrecer. Por ejemplo, si se quiere añadir una capa de persistencia mediante una base de datos, el uso de ella va a suponer un coste, que se verá reflejado en el cómputo global del rendimiento.

Estos costes provenientes de las estructuras internas son difíciles de reducir y evitar, pues normalmente dichas estructuras son intrínsecas a la funcionalidad del sistema. El objetivo es encontrar las herramientas que proporcionen la funcionalidad deseada suponiendo el menor coste temporal posible.

políticas de repartición del trabajo

Las políticas de repartición del trabajo hacen referencia a las diferentes formas de ejecutar una función una vez se recibe su correspondiente llamada. Existe una gran variabilidad en la creación de estas políticas, desde algunas tan sencillas como *watermarks* en el la ocupación del CPU de los nodos hasta basadas en teoría de juegos [6] o en *pools* de contenedores [7]. Para entender los tipos existentes y su funcionamiento se debe explicar en primer lugar la anatomía de una invocación.

Para que una invocación pueda ser ejecutada, el entorno de ejecución tiene que disponer de los siguientes elementos:

- **Runtime**: Es el conjunto de procesos y archivos necesarios para que se pueda interpretar la función. Un ejemplo claro de *runtime* es una instalación de un lenguaje de programación, pues contiene todo lo necesario para que los archivos de ese lenguaje puedan ser interpretados y ejecutados.

- **Dependencias**: Conjunto de archivos complementarios al *runtime* que necesita la función para su ejecución. Se trata de una categoría extraña, pues pueden ir incluidos tanto con el *runtime* como con la propia función. A efectos prácticos se suelen tratar como un elemento independiente, por lo que se considera relevante diferenciarlos.

- **Función**: Archivo o conjunto de archivos que contienen la lógica a ejecutar.

- **Parámetros de entrada**: Argumentos que recibe la función. Serán interpretados por ésta y de ellos dependerá el resultado de la ejecución.

Según el estado del entorno de ejecución que se asigne a una invocación recibida, se pueden dar dos situaciones:

En el caso de que la invocación encuentre un entorno de ejecución con un su *runtime* o su función cargados de antemano se obtiene lo que se conoce como un warm start. Este es el caso óptimo, pues todo el tiempo que se tarda en cargar el *runtime* o la función no se observa desde el punto de vista del usuario. Es decir, la invocación se *ahorra* el tiempo que se cargar el entorno hasta el punto en el que es encontrado.

Por otro lado existe el concepto de *cold start*. Es lo que sucede cuando la invocación es asigada un entorno de ejecución vacío o no compatible con ella, es decir, que el *runtime* o función que ha precargado no coincide con el que necesita la invocación. Es el peor caso, pues el usuario observará el proceso completo de carga de entorno hasta poder realizar la ejecución, lo que supondrá un mayor coste temporal.

El objetivo de las políticas de repartición del trabajo va a ser maximizar el número de *warm starts*, o, de forma equivalente, reducir el número de *cold starts*. Para ello se deben elaborar unas heurísticas que se adapten de la mejor forma posible al caso de uso que quiere resolver el sistema. Estas heurísticas se podrían definir como funciones que tienen como entrada cualquier tipo de métrica obtenida del sistema y como salida la decisión de precargar uno o varios *runtimes* o funciones. A continuación se analizarán dichas métricas y tipos de precarga brevemente:

- **Métricas:** Algunas de las métricas típicas utilizadas son el historial de invocaciones, el porcentaje de carga de cada uno de los entornos de ejecución o del sistema en general, tiempos de ejecución de las invocaciones o incluso patrones identificados en el flujo de llamadas.

- **Tipos de precarga:** En cuanto a los tipos de precarga, se pueden identificar tres principalmente:

  - No precargar nada. Se trata del peor caso, pues asegura *cold starts* en todas las invocaciones. Puede ser mejor que las precargas en casos marginales, donde todas las llamadas llegan a entornos de ejecución no compatibles y se debe borrar un contenedor precargado en cada invocación, con su coste temporal asociado.
  - Precargar *runtimes*. Permite la aparición de *prewarm starts*[2]. En los casos en los que la función no tenga dependencias o éstas sean muy ligeras, va a suponer una gran mejoría respecto a encontrarse un entorno de ejecución vacío. En el caso de funciones grandes, las diferencias se diluyen, pues la carga del *runtime* supone sólo una pequeña parte del proceso. Respecto a la precarga de funciones, sus *prewarm starts* van a ser peores temporalmente, ya que el entorno se encuentra en un estado menos avanzado. Sin embargo, estos *runtimes* precargados son más versátiles, ya que pueden ser compatibles con un gran número de funciones y, en consecuencia, de llamadas.
  - Precargar funciones. Se trata del caso complementario a la precarga de *runtimes*. Van a ser la mejor opción en cuanto a coste temporal en el caso que reciban invocaciones compatibles, pero esta tarea va a ser más complicada que en su contrapartida, pues los entornos precargados son específicos a llamadas de una función concreta. La mejoría de rendimiento respecto a la precarga de *runtimes* va a depender del peso que tengan las dependencias de las funcio-

[2]No confundir con *warm starts*, pues éstos corresponden a las invocaciones que llegan a un contenedor donde ya se ha ejecutado una función previamente, caso que no se contempla en este trabajo.

nes precargadas. Cuanto más pesadas sean dichas dependencias mayor será la mejora de rendimiento.

En resumen, el rendimiento del *FaaS* va a depender del uso de estructuras internas y de las políticas de repartición del trabajo. La optimización de ambos aspectos depende del caso de uso de la plataforma, ya que en base a él se pueden determinar las mejores herramientas y heurísticas a utilizar.

3.3.3. Gestión de recursos

El sistema debe hacer una buena gestión de los recursos computacionales de los que dispone. Para ello, se pretende que el consumo de estos recursos sea el mínimo suficiente necesario para que que el sistema funcione correctamente. De la misma forma, debe ser capaz de extenderse y ajustarse a la demanda que reciba. La soluciones identificadas más claras para conseguirlo son la escalabilidad y la elasticidad.

Un sistema se considera escalable cuando todos y cada uno de sus componentes pueden ser replicados para abastecer una demanda de peticiones mayor. El tráfico entonces debe ser repartido equitativamente entre todas estas réplicas de componentes y se tienen que aplicar mecanismos para mantener un estado global del sistema consistente. En el caso concreto de un *FaaS*, el primer punto que se debe considerar a escalar son los entornos de ejecución, ya que son la pieza principal del sistema y el primer factor limitante que se encontrará a medida que el sistema crezca. Cuando se empiece a tratar con volúmenes de datos masivos, se necesitará la replicación de otros componentes del sistema, como la *API* o la base de datos, y se deberán hacer uso de mecanismos como balanceadores de carga para asegurar el escalado hasta el infinito.

La incorporación de las nuevas réplicas de cada uno de los componentes a medida que el sistema escale se puede realizar de forma manual, sin embargo se recomienda hacer uso de uso de una heurística que realice este escalado de forma automática. Cuando el sistema es capaz de levantar y tumbar nuevas instancias de componentes de forma automática se considera elástico. La elasticidad es un punto clave en la gestión eficiente de recursos, pues permite que el sistema se pueda ir adaptando de forma dinámica a la carga recibida, utilizando los recursos mínimos posibles que puedan abastecer la demanda en cada momento.

3.3.4. Alta disponibilidad

El sistema debe ser altamente disponible, para lograrlo, se deben tener en cuenta principalmente dos aspectos, la redundancia y la tolerancia a fallos de cada uno de sus componentes.

La redundancia es necesaria para no dejar sin funcionalidad al sistema en el caso de que un componente falle. Al producirse este fallo, el tráfico destinado a el componente caído se redirige hacia la instancia redundada. Esto permite que el tiempo de disponibilidad de la plataforma no se vea interrumpido por fallos en el sistema. La redundancia se puede tener en cuenta a diferentes niveles. Lo más normal es realizarla a nivel de máquinas o, en casos excepcionales que trataran con despliegues en *Clouds* públicos a nivel de zonas de disponibilidad o regiones. Esto se debe a que se presupone que la plataforma no cuenta con errores de lógica, y los únicos fallos que se pueden dar son físicos, tales como errores de *hardware* o caídas del sistema eléctrico. Cuando se trata con despliegues *on premises* en un servidor centralizado esta redundancia pierde un poco de interés, pues en caso de el fallo de una máquina todo el sistema caería. No obstante, en el caso de que pueda haber problemas de lógica se puede tener en cuenta.

La tolerancia a fallos está directamente relacionada con la redundancia, ya que, para mantener el sistema funcionando el máximo tiempo posible, aquellas instancias que fallen se deberán poder reincorporar a el sistema y seguir recibiendo tráfico. Para lograr esta tolerancia a fallos, cada uno de los servicios que formen parte del sistema debe estar preparado para fallar, así como para que fallen los servicios con los que se comunica. Una vez haya fallado, el sistema debe identificarlo e intentar volver a ponerlo en marcha. Por otro lado, los fallos deben ser internos a un servicio, intentando no propagarse al resto del sistema y provocar su colapso. Finalmente, los fallos a nivel físico deben ser tolerados y recuperados de la misma manera que los internos al sistema, volviendo a iniciar las máquinas y los procesos necesarios para que los nodos afectados se reincorporen. Algunas de las cosas a tener en cuenta relativas a la resiliencia del sistema son:

- **La arquitectura del sistema**: Va a ser relevante para evitar la propagación de los fallos. En una arquitectura monolítica, un fallo del sistema afectará a todos los servicios, haciendo que el sistema caiga completamente. Una arquitectura basada en microservicios va a soportar mucho mejor el fallo de uno de sus componentes, manteniendo el resto funcionales.

- **La orquestación de los componentes**: Es el aspecto que define qué política van a seguir los componentes en caso de fallo, y el que permitirá que se vuelvan a levantar y se reincorporen en el sistema. En caso de que la plataforma haga uso de una herramienta de orquestación para su despliegue, ésta se encargará de mantener la disponibilidad del sistema. De no ser así, un administrador deberá monitorizar manualmente los componentes y realizar las acciones necesarias para cumplir con esta disponibilidad.

- **La persistencia del estado**: En el caso de que se mantenga un estado global en el sistema, y los componentes necesiten conocerlo para que su incorporación sea correcta, se deberá persistir este estado. De no ser así, el componente que se intente incorporar tendrá un estado inconsistente, lo que potencialmente puede ser un punto de fallo.

3.3.5. Facilidad de uso

Respecto al tema de la facilidad de uso, cabe diferenciar dos puntos a tener en cuenta: la facilidad de uso desde el punto de vista del usuario final y, al tratarse de un sistema pensado para ser desplegado *on premises* , la facilidad de su gestión por parte del administrador del sistema.

La facilidad de uso por parte del usuario depende principalmente de la vía de comunicación entre él y el sistema, es decir, el tipo de interfaz que se va a utilizar para hacer uso de la plataforma. Una buena solución es ofrecer una interfaz gráfica sencilla pero potente, con suficientes opciones para que los usuarios avanzados puedan realizar acciones complejas rápidamente, pero organizada de manera que usuarios menos expertos no se vean abrumados. Como alternativas se puede ofrecer una interfaz de línea de comandos, que, pese a ser poco *user-friendly* puede que se adapte mejor a las cargas de trabajo de usuarios avanzados. Por último, la opción menos intuitiva sería exponer los *endpoints* de la *API* y que los usuarios crearan sus propias herramientas para utilizarlos. Por otro lado, la especificación de los objetos internos puede limitar estas interfaces. Por ejemplo, si un *runtime*, función o *trigger* que tiene que ser declarado por el usuario está compuesto de elementos confusos o difíciles de representar, esto repercutirá en la experiencia de usuario.

Se debe de tener también en cuenta la facilidad de gestión del sistema. Este aspecto engloba cuestiones relativas a la alta disponibilidad y la gestión de recursos, tales como el despliegue, la tolerancia a fallos o el escalado automático del sistema. Se espera que el administrador encargado de desplegar y mantener el sistema tenga unos conocimientos avanzados en el tema, por lo que en este punto no se busca la sencillez de las interfaces sino su potencia. La plataforma deberá tener sentido a nivel de arquitectura y presentar un manual de uso y una documentación completos y actualizados. Todas estas cuestiones se han de tener en cuenta para que la gestión del sistema sea lo más sencilla posible.

3.4 Solución propuesta

Una vez analizadas las soluciones posibles en lo que respecta a cada uno de los apartados del sistema, se procede a seleccionar aquellas que se adaptan mejor a la solución a desarrollar, y sobre las que se realizará su diseño.

- **Aislamiento**: La solución que mejor se adapta a las necesidades del sistema es el uso de contenedores. Proporcionan un buen nivel de aislamiento sin comprometer en gran medida el rendimiento, por lo que será la abstracción elegida para los entornos de ejecución del sistema.
- **Rendimiento**: En cuanto a el rendimiento, se buscarán las estructuras internas con el mejor rendimiento y que se adapten al caso de uso a cubrir. Para las políticas de invocación, se deberá poder hacer uso de heurísticas que realicen precargas tanto de *runtimes* como de funciones. Estas heurísticas deben ser fácilmente programables e intercambiables.
- **Gestión de recursos**: Respecto a la gestión de recursos, los componentes del sistema deben ser escalables y elásticos, manteniendo consistentemente el modelo de datos y distribuyendo la carga de forma adecuada.
- **Alta disponibilidad**: El sistema resultante debe ser altamente disponible. Para ello, debe contar con los mecanismos necesarios de redundancia y tolerancia a fallos.
- **Facilidad de uso**: El sistema debe presentar una interfaz sencilla y potente para su uso por parte de los usuarios. Los mecanismos de gestión del sistema deben ser explicados en un manual de uso, y los componentes del sistema deben estar correctamente documentados.

CAPÍTULO 4
Diseño de la solución

Una vez exploradas las soluciones existentes, analizadas la estructura y las características fundamentales de los *FaaS* e identificados los puntos que se quieren priorizar en el proyecto y la mejor forma de abordarlos, se procederá a realizar un diseño de la plataforma a desarrollar. Este diseño se va a realizar en dos partes. En primer lugar se realizará una descripción de la arquitectura de la plataforma. Esta descripción va consistir en listar los componentes que forman el sistema y las relaciones que existen entre ellos. Tendrá como objetivo mostrar un punto de vista general del sistema. Por otro lado, se describirá detalladamente el diseño cada uno de los componentes, recalcando las peculiaridades o decisiones que se han tenido que tomar. En este punto se podrá obtener una visión a bajo nivel del funcionamiento de cada uno de estos componentes. Para terminar, se propondrán una serie de tecnologías que se adapten al sistema diseñado y sirvan como referencia para su posterior desarrollo.

4.1 Arquitectura del sistema

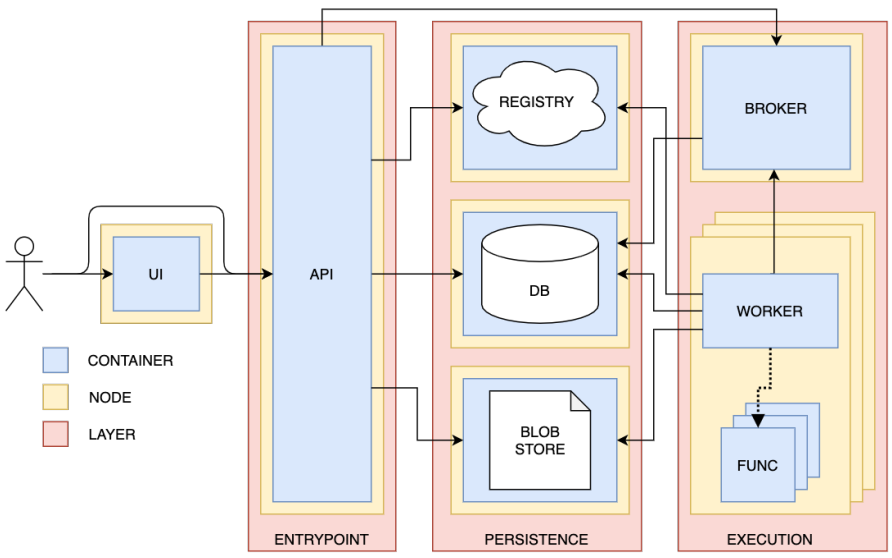


Figura 4.1: Arquitectura del sistema

La arquitectura del sistema se ha diseñado teniendo en cuenta las referencias estudiadas en el capítulo 2, así como el análisis de cada uno de los elementos fundamentales que componen un *FaaS* (realizado en la sección 3.1). Tal y como muestra la figura 4.1, el siste-

ma se puede separar entre capas principalmente: El *entrypoint*, la capa de persistencia y, por último, la capa de ejecución.

La primera capa es el *entrypoint*. Se compone únicamente de una *API*. Su función consistirá principalmente en recibir las peticiones de los usuarios, procesar los datos que sean necesarios, y dirigirlas hacia el componente que las gestionará, en caso de haberlo. La *API* va a aceptar diversos tipos de peticiones, siendo las más relevantes las que se encarguen de registrar *runtimes*, registrar funciones e invocar dichas funciones. Todas estas funcionalidades estarán incluidas en una interfaz con la que el usuario sea capaz de interactuar.

La segunda capa se encarga de la persistencia. Es la que va a mantener los datos que se quieran conservar almacenados en disco, para poder ser recuperados cuando se desee o en caso de caída del sistema y pérdida de la memoria. Existen tres componentes diferentes en esta capa. El primero es un *registry* y hace la función de un almacén de imágenes. Tal y como se explicará con más detenimiento en la sección 4.2, los *runtimes* serán imágenes, y quedarán almacenadas en este *registry*, pudiendo ser accedidas desde cualquier punto del sistema. En segundo lugar tenemos la base de datos. Se encargará de mantener diversos tipos de objetos, tales como las llamadas que se realicen al sistema o los metadatos tanto de los *runtimes* como de las funciones. Por último tenemos el *blob store*. Su funcionamiento es análogo al del *registry*, pero tomando como elemento almacenado las funciones. Las funciones serán archivos binarios comprimidos, y este *blob store* será el encargado de servirlos a los *workers* que los requieran.

En último lugar está la capa de ejecución, que se compone de los *brokers* y los *workers*. Los *brokers* son los encargados de implementar las heurísticas de invocación. Se pueden entender como los controladores de los *workers*, dándoles instrucciones para que se precarguen sus funciones de una forma determinada o simplemente delegando en ellos las llamadas a las funciones. Los *workers* recibirán estas órdenes y serán los encargados de ejecutarlas. Para ello tendrán una serie recursos asignados, sobre los que desplegarán los *runtimes* y las funciones indicadas por el *broker* en cada momento.

4.1.1. Escalado de los componentes del sistema

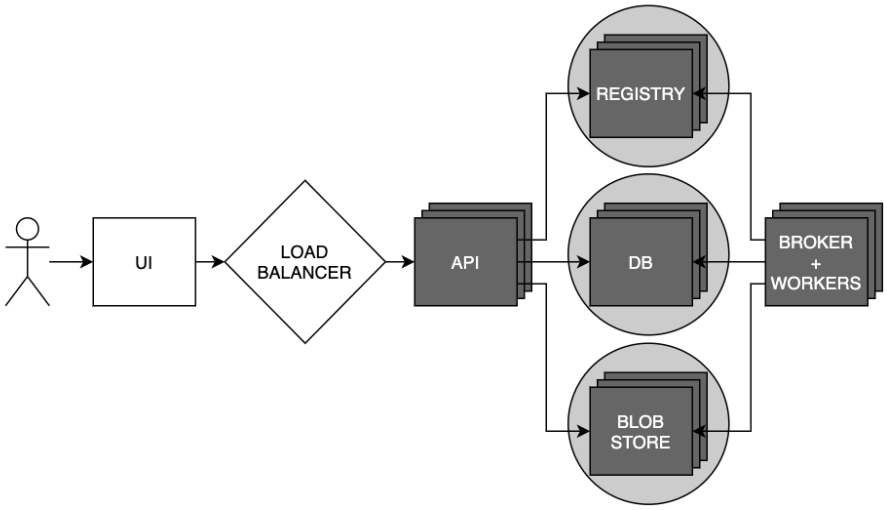


Figura 4.2: Escalado de los componentes del sistema

La figura 4.2 muestra la forma de escalar de los componentes y capas del sistema. Las casillas en blanco representan los elementos que no escalan: la interfaz estará en el

lado del usuario, por lo que no concierne al sistema. En cuanto al balanceador de carga, teóricamente puede escalar, aunque el volumen de peticiones que se deben recibir para que esto ocurra tiene que ser masivo. En cuanto a las casillas oscuras, representan los elementos del sistema que sí que escalan. En el caso de estar envueltas en un círculo, se denota la necesidad de que mantengan un estado consistente, por lo que sus instancias se deberán comunicar para alcanzar eventualmente este estado de consistencia. La parte más interesante a destacar es el escalado de *brokers* y *workers*. Tal y como se ve en el esquema 4.1, la comunicación interna de esta capa es de un *broker* a *N workers*. Esto quiere decir que, cuando se escale la capa, se tendrán un número de *brokers* cada uno con su conjunto propio de *workers*. Se puede entender como que cada *broker* gestiona un *cluster* independiente y cada *worker* una máquina virtual de dicho *cluster*.

4.2 Diseño detallado

Una vez mostrada la arquitectura a nivel general, se procederá a indagar en cada uno de sus componentes, explicando detalles de su diseño a más bajo nivel. Se empezará realizando un diseño de los objetos internos del sistema, con el objetivo de poner en contexto el diseño de los componentes. Finalmente, se tratarán otros aspectos relevantes para el diseño del sistema, tales como el tipo de conexiones entre componentes o sus mecanismos de tolerancia a fallos.

4.2.1. Objetos internos

Existen tres objetos principales con los que debe interactuar el sistema: *runtimes*, funciones y llamadas.

Runtimes

Los *runtimes* del sistema son imágenes de *Docker*. Se registran haciendo la llamada pertinente a la *API*, y es ésta la encargada de procesar esa imagen, es decir, recibirla (bien de el almacenamiento local del nodo o de una fuente externa, como por ejemplo *Docker Hub*) y subirla al *registry* del sistema. Cuando el *broker* estime que es necesario hacer uso de uno de estos *runtimes*, se lo comunicará a un *worker*, que llamará al *registry* y lo descargará.

Por otro lado, estos *runtimes* disponen de una serie de metadatos necesarios para que se puedan realizar ejecuciones sobre ellos:

Campo	Tipo	Descripción
dependencies	String (cmd)	Comando para instalar las dependencias
image	String	Imagen de *Docker* asociada al *runtime*
path	String (path)	Path donde se almacenará la función
run	String (cmd)	Comando para ejecutar la función

Tabla 4.1: Metadatos del *runtime*.

Tal y como se ve en la tabla 4.1, cada *runtime* tiene una serie de datos relativos a la forma con la que va a tratar las funciones. Estos datos deben ser conocidos por los desarrolladores de las funciones que hagan uso de él, pues éstas funciones deben seguir estos requisitos impuestos por el *runtime* para su correcto funcionamiento. Es decir, no

existe una convención a nivel de sistema para estos requisitos, sino que son impuestos por cada *runtime* e implementados por las funciones.

Funciones

Las funciones son binarios comprimidos, compuestos de archivos y directorios. Estas funciones son registradas mediante una llamada a la *API*, y ésta se encargará de transmitirlas al *blob store*, para que los *workers* que las necesiten puedan acceder a ellas. Los archivos contenidos en la función tendrán que seguir las convenciones marcadas por el *runtime* en el que se vayan a ejecutar, así como la especificación de entrada y salida a nivel de sistema. Las funciones contienen una serie de metadatos relativos a su subida que se almacenan en la base de datos, tales como el tamaño o el nombre del archivo que se sube al sistema; así como algunos que se añaden manualmente, como el *runtime* al que corresponde la función. No se considera relevante mostrar estos datos, pues no tienen un papel de peso en el funcionamiento del sistema.

Llamadas

Las llamadas o invocaciones a las funciones suponen la unidad de ejecución del sistema. Son declaradas en la *API*, y se almacenan directamente en la base de datos. De la misma manera se dirigen hacia un *broker* disponible, para que él, según la heurística configurada, realice sus operaciones pertinentes. Los datos que componen una llamada son los siguientes:

Campo	Tipo	Descripción	Origen
callNum	Integer	Número de llamada para su identificación	Sistema
funcName	String	Nombre de la función a invocar	Usuario
params	Object	Parámetros de entrada de la llamada	Usuario
status	String	Estado de la llamada	Sistema
result	Object	Resultado de la llamada, salida de la función	Sistema
timing	Object	Tiempos obtenidos en las diferentes fases	Sistema

Tabla 4.2: Datos de las llamadas.

Existen dos tipos de datos a almacenar, algunos aportados por el usuario en su llamada de *API* y otros por el sistema. Por este motivo, las llamadas en la base de datos recibirán actualizaciones cuando haya más información sobre ellas que aportar, es decir, el objeto no será estático.

4.2.2. Componentes

Interfaz de usario

La función principal de la interfaz de usuario es la facilidad de uso. Tal y como se comentaba en la sección 3.2, la facilidad de uso es uno de los aspectos menos prioritarios en la plataforma a desarrollar, por lo que no se ha hecho un estudio muy exhaustivo de este componente.

la solución diseñada es una interfaz de línea de comandos, que permite modificar la configuración del sistema, así como realizar sus despliegues, monitorizar su estado y hacer llamadas a la *API*.

API

La *API* deberá ofrecer las siguientes llamadas principales:

- **POST registerRuntime**. Este es el *endpoint* para registrar los *runtimes*, tal como su nombre indica. Espera un objeto de *runtime* como entrada y devuelve un código de estado *HTTP*. El componente *API*, al recibir esta llamada sube el *runtime* al *registry* y guarda sus metadatos en la base de datos.

- **POST registerFunction/:runtime/:funcName**. Este es el *endpoint* que recibirá las llamadas que se encargarán de registrar funciones. Su entrada será el objeto de función, es decir, el conjunto de archivos que compongan la función comprimidos en un archivo *zip* o un *tarball*. La *API* se encargará de almacenarlos en el *blob store*, así como de informar a los *brokers* de su existencia, para que los *workers* los descarguen y descomprimas en sus directorios designados con la finalidad de no perder tiempo cuando sean necesarios. Devuelve un código de estado *HTTP*.

- **POST invokeFunction**. Este *endpoint* recibirá las llamadas encargadas de invocar las diferentes funciones del sistema. Su entrada son los datos correspondientes al objeto de función que dependen del usuario, es decir, el nombre de la función y sus parámetros de entrada. Una vez registrada la invocación, se le devuelve al usuario el número de llamada, que podrá ser utilizado para consultar su estado y obtener los resultados; se introduce en la base de datos la información parcial del objeto y se envían los datos al *broker* para que se encargue de seleccionar un *worker* para ejecutarla.

- **GET call/:callNum**. Este *endpoint* se encarga de devolver el estado de la invocación que se le pasa en la ruta. La *API* contiene los datos de las últimas invocaciones realizadas en memoria, con el fin de no llamar a la base de datos cada vez que se recibe una de estas peticiones.

Existen más *endpoints*, encargados de devolver la lista y características de los *runtimes* y funciones, así como de mostrar estadísticas sobre el estado del sistema. Al no proporcionar una funcionalidad primordial para el sistema, su diseño no se estima tan relevante, por lo que no se analizarán con detenimiento.

Registry y blob store

El *registry* y *blob store* son dos componentes para los que se realizará un despliegue de un *software* existente, ya que existen múltiples soluciones que cumplen con sus cometidos. Es por ello que su diseño es poco interesante, simplemente se tienen que gestionar las conexiones a ellos por parte de los componentes que los utilicen.

Base de datos

La base de datos va a ser el componente encargado de almacenar los metadatos de los *runtimes* y las funciones, así como los datos de las invocaciones. El diseño de estos objetos a almacenar se muestra en la subsección 4.2.1. La arquitectura de la base de datos es muy sencilla, utilizando una tabla para cada uno de los tipos de objeto. El componente de la base de datos contendrá una interfaz con funcionalidades básicas, con la finalidad de que el resto de componentes puedan realizar acciones *CRUD* sobre los diferentes objetos.

Broker

El *broker* será el componente encargado de repartir el trabajo entre sus *workers*. Esta repartición del trabajo se va a realizar en base a una heurística, tal y como se explica en la subsección 3.3.2. El *broker* debe ser capaz de mantener el estado de cada uno de sus *workers* y sus *spots* de forma consistente, sabiendo en cada momento qué funciones hay ejecutándose, qué huecos libres y qué contenedores precargados. Al contener toda la lógica de la heurística, los errores de consistencia se producirán en este punto y serán propagados a los *workers*, pudiéndose dar casos de no finalidad (*workers* atascados) o de sobrecarga (*workers* lanzando más contenedores que *spots*).

Dada la naturaleza ampliamente configurable de la plataforma que se está diseñando, es interesante que las heurísticas aplicadas a nivel de *broker* sean configurables. Para ello el *broker* proporciona una interfaz para gestionar el estado de los *spots* y los *workers*, aislando la lógica de la heurística y haciendo una labor sencilla su modificación o reemplazo.

Worker

El *worker* es el componente encargado del despliegue de los entornos de ejecución, así como de la propia ejecución de las funciones. Cada *worker* continene un número limitado de *spots*, que corresponde al número de contenedores concurrentes que un *worker* puede gestionar. Este número deberá ser configurado por el administrador del sistema, pues los *workers* no tienen por qué ser homogéneos, pudiéndose dar el caso de que, para el mismo *broker* haya *workers* con diferente número de *spots*.

El *worker* pedirá trabajo al *broker* cada vez que tenga un *spot* disponible para ello. El *broker* enviará el tipo de operación que quiere que el *worker* realice en dicho *spot*. Las operaciones que ofrece la interfaz del *worker* son las siguientes:

- **executeNoPreload**. Realiza el proceso completo de ejecución de una función: lanza el contenedor utilizando la imagen del *runtime*, copia los archivos de la función, ejecuta el comando designado del *runtime* para realizar la carga de dependencias, copia el archivo que contiene el *input* de la llamada, ejecuta el comando del *runtime* que ejecuta la función, copia el archivo de salida y destruye el contenedor.
- **preloadRuntime**. Simplemente lanza el contenedor utilizando la imagen del *runtime* y lo mantiene activo a la espera de una llamada de ejecución.
- **preloadFunction**. Lanza el contenedor utilizando la imagen del *runtime*, copia los archivos de función y carga las dependecias. Mantiene el contenedor activo.
- **execRuntimePreloaded**. Realiza el proceso de ejecución de forma análoga a *executeNoPreload*, partiendo desde un *runtime* precargado.
- **execFunctionPreloaded**. Realiza el proceso de ejecución de forma análoga a *executeNoPreload*, partiendo desde una función precargada, es decir, desde el punto donde se ha copiado la función y cargado los argumentos.
- **clearSpot**. Realiza el borrado del contenedor existente en el *spot* seleccionado. Es necesario para asegurar la finalidad en el sistema. Por ejemplo, en el caso de que todos los *spots* del *worker* estén precargados con un *runtime* y se reciba una llamada que utilice un *runtime* diferente, se deberá liberar un *spot* para que esta invocación se pueda llevar a cabo.

4.2.3. Conexiones

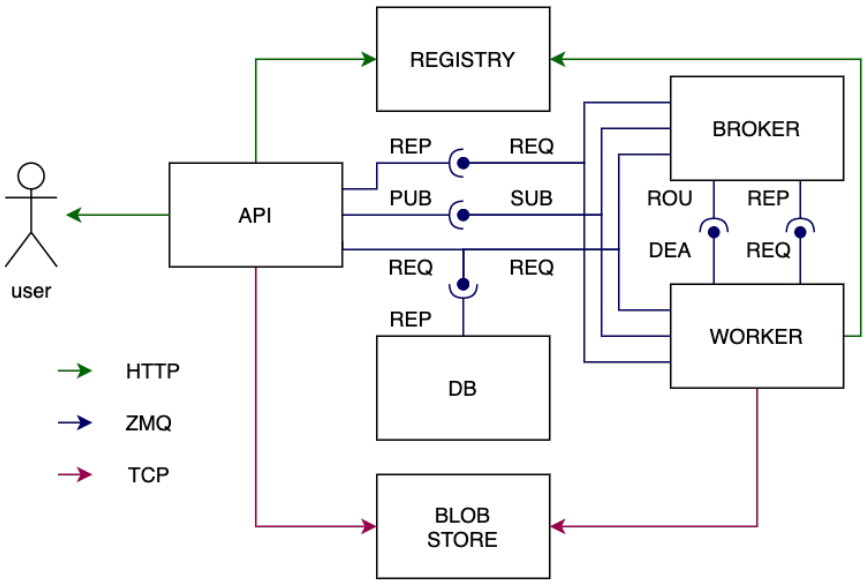


Figura 4.3: Diseño de las conexiones del sistema

La figura 4.3 muestra el esquema de comunicaciones del sistema. Se puede observar en ella que el *registry* y la *API* ofrecen conexiones mediante *HTTP* y el *blob store* ofrece una conexión del tipo *TCP*. Estas conexiones no son muy interesantes, pues la de la *API* se ha definido previamente y las otras dos vienen dadas por la tecnología que constituya sus respectivos componentes. El punto importante de este diseño de comunicaciones son los canales implementados en *ZMQ*[1]. A continuación se explicará en profundidad cada uno de ellos.

- **REQ-REP *API***. Es un canal utilizado como vía rápida para comunicar información desde el *broker* y los *workers* a la *API*. Se utiliza para enviar las respuestas de las invocaciones, pues así se evita el coste temporal de utilizar la base de datos como punto intermedio.
- **PUB-SUB *API***. Es un canal de difusión de los registros que se producen en el sistema. Cuando la *API* recibe un *runtime* o función nueva, se propagan los metadatos por esta vía a los *workers* y los *brokers*, para que ellos los almacenen en memoria. El envío de las invocaciones también se realiza por este canal.
- **REQ-REP DB**. Es el punto de entrada a la base de datos y el canal a través del cuál se expone su interfaz. Cuando cualquier componente quiere llamar a dichos métodos, bien para el registro o la recuperación de información, hacen una petición a la base de datos por esta vía.
- **REQ-REP Broker**. Sirve para realizar el *handshake* entre los *workers* y el *broker*. Cuando se instancia un nuevo *worker*, le hace una llamada al *broker*, indicando el número de *spots* de los que dispone. El *broker* le asigna un identificador, que deberá usar en las posteriores comunicaciones.
- **ROU-DEA Broker**. Es el canal de comunicación principal entre el *broker* y los *workers*. Los *workers* realizan peticiones, devolviendo el estado de sus *spots* y el *broker* les manda, dirigiendo el contenido a un *spot* (y por tanto a un *worker*) concreto.

[1] https://zeromq.org/

4.2.4. Despliegue y gestión del sistema

La forma de desplegar la plataforma va a tener un papel relevante en su comportamiento, pues de ella dependen aspectos tales como la replicación de los componentes, la forma que tendrán de encontrarse o el proceso de recuperación de las instancias caídas. Como el sistema se ha diseñado siguiendo una arquitectura basada en microservicios, cada uno de los componentes del sistema toma la forma de un contenedor independiente. Esto facilita mucho su despliegue y gestión, pues los contenedores son sencillos de levantar, tirar y reiniciar.

Ya que se trata de un sistema distribuido, el primer paso para realizar el despliegue será introducir la información relativa a cada servicio en un archivo de configuración, que la plataforma consumirá en el momento del despliegue. En este archivo de configuración se debe describir la dirección *IP* en la que se encuentra cada componente, así como los puertos que se han seleccionado en dichos nodos para situar cada uno de los canales que los componentes utilicen (figura 4.3). Los componentes leerán esta configuración para poder realizar las conexiones y comunicarse entre ellos.

Los componentes han sido diseñados para que sus fallos no afecten al funcionamiento global del sistema, evitando su propagación. De la misma manera, su conexión se puede realizar en cualquier momento, recuperando en ese momento el estado que haya persistido, o, en caso de tratarse de una réplica nueva, el estado previo necesario para su incorporación. Teniendo esto en cuenta, así como otros aspectos mencionados en el documento, tales como que la plataforma está pensada para ser utilizada en una infraestructura *on premises* , se ha determinado que la forma óptima de realizar el despliegue y gestión del sistema es una herramienta de orquestación de contenedores. La herramienta de orquestación será la encargada de levantar todos los componentes, así como de volver a incorporarlos al sistema en caso de caída. Se deberá hacer cargo de la replicación de los componentes, siguiendo algún tipo de heurística configurada o en respuesta a llamadas por parte del propio sistema.

Partiendo del diseño de componentes basado en microservicios y haciendo uso de una herramienta de orquestación sobre la infraestructura propia, se dota a la plataforma de mecanismos para optimizar la gestión de recursos, ofrecer alta disponibilidad y robustez frente a los fallos que puedan surgir.

4.3 Tecnología propuesta

A continuación se realizará un listado de la tecnología propuesta para cada uno de los componentes del sistema, así como su justificación:

- ***API***: Cualquier servidor de *API REST* es una buena propuesta para este punto, pues no requiere ningún tipo de funcionalidad especial. Dependerá del lenguaje de programación en el que se quiera desarrollar y las preferencias del desarrollador. Algunos ejemplos pueden ser *express*[2] o *flask*[3].

- ***Registry***: Como se va a trabajar con imágenes de *Docker*, la solución que mejor se adapta es utilizar la imagen de *Docker Registry*[4] para este componente. Cubre las necesidades propuestas y permite la replicación de sus nodos.

[2]https://expressjs.com/
[3]https://flask.palletsprojects.com/en/1.1.x/
[4]https://docs.docker.com/registry/

- **Base de datos**: La base de datos debe ser capaz de permitir la replicación de sus nodos, así como utilizar un esquema no relacional, de acuerdo al modelo de datos. Cualquier base de datos que se adapte a estas características es adecuada para el sistema. Algunos ejemplos podrían ser *Cassandra*[5] o *CockroachDB*[6].

- **Blob store**: Las característica principal que debe presentar el *blob store* es la replicación, además de las funcionalidades implícitas a ser un *blob store*. Un ejemplo podría ser *Minio*[7].

- **Broker y workers**: El *broker* y los *workers* van a ser desarrollos propios, por lo que se debe realizar una selección del lenguaje en el que serán programados. Al tratarse de un prototipo, los lenguajes interpretados proporcionan un flujo de trabajo más adecuado, por su agilidad y su adaptabilidad. Se proponen *Node*[8] y *Python*[9], por su relevancia y volumen de uso.

- **Funciones**: Las funciones van a ser, tal y como se ha anticipado en el documento, contenedores de *Docker*[10]. Al elegir como abstracción para los entornos de ejecución los contenedores, *Docker* es la solución más relevante en esa categoría, convirtiéndola en la mejor opción.

- **Conexiones internas**: Como se muestra en la figura 4.3, los patrones de comunicación de las conexiones internas se han diseñado con vista a ser implementados en *ZMQ*. Se ha propuesto esta herramienta por su rendimiento, volumen de usuarios, facilidad de uso y disponibilidad, pues cuenta con librerías para una gran cantidad de lenguajes.

- **Orquestación**: La solución propuesta para la orquestación es *Kubernetes*[11]. Es considerado el estándar en cuanto a orquestación de contenedores, y lo avalan un gran número de usuarios así como el hecho de ser un producto de *Google*. Su funcionalidad se cubre perfectamente a las necesidades del sistema.

[5] https://cassandra.apache.org/
[6] https://www.cockroachlabs.com/
[7] https://min.io/
[8] https://nodejs.org/en/
[9] https://www.python.org/
[10] https://www.docker.com/
[11] https://kubernetes.io/

CAPÍTULO 5
Desarrollo de la solución propuesta

Una vez finalizado el diseño, se proceden a implementar cada uno de los componentes identificados. La solución se ha desarrollado siguiendo una metodología basada en tareas, utilizando como herramienta de gestión la organización en proyectos que ofrece *GitHub*[1]. Estas tareas se iban atacando de una en una, pasando por fases de desarrollo y validación, hasta ser finalmente incorporadas en la base de código.

Uno de los puntos conflictivos en el proceso de desarrollo de la solución fue la dimensión del proyecto. Al tratarse de una implementación propia y partiendo desde cero, el tiempo estimado para desarrollar la solución era muy elevado. Teniendo en cuenta las características y el alcance a nivel curricular que supone el trabajo de fin de máster, este desarrollo, junto al resto de apartados que componen el proyecto, superaban este tiempo asignado con creces.

Es precisamente por estos motivos por los que la plataforma desarrollada que se expondrá a continuación no cumple con todos los requisitos identificados en la fase de diseño, a pesar de llegar a ser completa y funcional. En las próximas secciones se explicará el proceso de desarrollo de los componentes, exponiendo su nivel de ajuste con el diseño realizado en el capítulo 4. A continuación se expondrán otros aspectos relativos al desarrollo que se consideran relevantes, pese a no tratarse de componentes directamente.

5.1 Desarrollo de componentes

5.1.1. Interfaz de usuario

No se ha desarrollado ningún tipo de interfaz de usuario para la plataforma. Tal y como se exponía en la sección la facilidad de uso es uno de los aspectos menos prioritarios en el diseño del sistema. No obstante, se ha tenido en cuenta durante todo el recorrido del proyecto, y la implementación de una herramienta de *CLI* que se comunique con la *API* no parece una tarea muy complicada.

Por tanto, la comunicación de los usuarios con la plataforma se va a realizar mediante el uso de algún cliente *HTTP*, tales como *requests*[2] o *curl*[3]. Los usuarios serán los encargados de construir sus peticiones en base a la especificación y de procesar las salidas devueltas por parte del sistema.

[1] https://github.com/
[2] https://requests.readthedocs.io/en/master/
[3] https://curl.se/

5.1.2. *API*

La *API* no ha supuesto ningún problema en cuanto a su implementación, y su funcionalidad corresponde perfectamente a la descrita en el capítulo de diseño. Se ha desarrollado en *Node*, haciendo uso de la herramienta *Express* para crear el servidor.

Para las conexiones externas expone los *endpoints* necesarios para el registro de *runtimes* y funciones, la invocación de funciones y la consulta de los *runtimes*, funciones e invocaciones presentes en el sistema. Por otro lado, para las internas, enlaza dos puertos de servicio de *ZMQ*, que implementan sus conexiones *PUB-SUB* y *REQ-REP*, y se conecta al puerto de la base de datos.

Uno de los detalles de implementación curiosos es el proceso de recepción de *runtimes*. Se utiliza la herramienta *multer*[4] para la recepción de los archivos comprimidos que componen las funciones. Dado que no se ha implementado el componente de *blob store*, es la propia *API* la encargada de enviar dicho binario a los nodos que contienen los *workers*. Este envío se realiza haciendo uso de *SCP*.

Por último, la *API*, al igual que el resto de componentes, se ha introducido en una imagen de *Docker* autocontenida para su despliegue y utilización de forma sencilla.

5.1.3. *Registry*

El *registry* ha sido introducido siguiendo la especificación del diseño. Se ha utilizado la imagen oficial de *Docker Registry* y se han expuesto sus puertos al exterior para que pueda ser utilizada por el resto de componentes del sistema.

5.1.4. Base de datos

La base de datos utilizada ha sido *lokijs*[5]. Es una solución muy ligera, y que no se corresponde con los requisitos especificados en la fase de diseño. Ha sido elegida como herramienta a utilizar por su conocimiento previo, y consecuente ahorro de tiempo en la fase de desarrollo.

Es una solución que indexa objetos y los almacena en texto plano. No ofrece ninguna de las funcionalidades de replicación necesarias para la instanciación de réplicas, por lo que, en la versión desarrollada del sistema la base de datos será centralizada.

El componente que la aloja ha sido desarrollado en *Node*, y su interfaz de comunicación en *ZMQ*, haciendo uso del patrón *REQ-REP*.

En cuanto a su adaptabilidad hacia el futuro, las consultas y operaciones que realiza son muy sencillas, y su interfaz está bien documentada, por lo que el desarrollo de un componente que utilice otro motor de bases de datos e implemente dicha interfaz parece sencillo.

5.1.5. Blob store

El *blob store* no se ha desarrollado, tal y como se adelantaba en la sección relativa a la *API*. Su funcionalidad se encuentra en ésta, pues es la encargada de mandar a cada uno de los *workers* los archivos comprimidos que contienen las funciones. Los *workers* serán los encargados de descomprimir dichos archivos para luego copiar su contenido en los entornos de ejecución correspondientes.

[4] https://www.npmjs.com/package/multer
[5] https://github.com/techfort/LokiJS

Uno de los problemas que esto supone es que los *workers* no son capaces de recuperar su estado en caso de caída, pues la *API* no dispone de ningún mecanismo para reenviar dichos binarios, pese a que los almacena. Esto se podría conseguir añadiendo un método en su interfaz correspondiente al canal *REQ-REP*, pero queda fuera de su dominio, e interrumpiría potencialmente su flujo de trabajo.

La incorporación de un *blob store* como *Minio* debería ser sencilla, simplemente se deben aplicar los cambios necesarios en la *API* y el *worker* para realizar las peticiones al servicio, y configurar éste de manera que se adapte correctamente al sistema.

5.1.6. Broker

El *broker* se ha desarrollado siguiendo las decisiones tomadas en la fase de diseño. Se ha desarrollado en *Node*, al igual que todas las implementaciones que se han realizado en el sistema. En este caso, se trata de un lenguaje idóneo, pues su diseño dirigido por eventos es de gran ayuda en este tipo de situaciones, donde las comunicaciones son tan relevantes.

El *broker* expone dos puertos de *ZMQ*, un *request-reply* y un *router-dealer*. El primero se utiliza a modo de *handshake* para el descubrimiento y la incorporación de nuevos *workers*. Éstos realizan una llamada, indicando el número de *spots* de los que disponen. El *broker* les contesta asignándoles un identificador. El segundo sirve para realizar todo el resto de comunicaciones, pues el *broker*, conociendo la identidad de los *workers* que le hablan y qué *spots* tienen asignados, será capaz de dirigir las llamadas hacia nodos concretos.

Por otro lado, el *broker* se conecta a la *API*, para recibir información sobre los registros de nuevos *runtimes* o funciones, así como a la base de datos, para realizar consultas e insertar datos.

A continuación, se explicarán con detenimiento las estructuras de datos y heurísticas que componen el *broker*:

Estructuras de datos

A nivel interno, el *broker* cuenta con las siguientes estructuras de datos, que se explicarán brevemente para ilustrar su comportamiento:

- **runtimeStore**: almacena todos los metadatos de los *runtimes* registrados en la plataforma.
- **functionStore**: almacena todas las funciones registradas en el sistema. Se deben conocer en este punto porque el *broker* debe tomar decisiones que requieren conocer si existe algún *spot* con un *runtime* cargado compatible con dicha función.
- **CallQueue**: sirve como *buffer* para todas las llamadas que han llegado al *broker* y no pueden ser inmediatamente mandadas a ejecutar. En condiciones en las que el dimensionamiento de la plataforma es adecuado, y, en consecuencia, la demanda no supera la cantidad de recursos del sistema, nunca debería ser utilizada.
- **workerStore**: contiene la información relativa a los *workers*. En él podemos encontrar la cantidad de *workers* disponibles y sus identificadores, así como el número de *spots* de los que dispone cada uno y su estado. Es un elemento primordial para el comportamiento del *broker*, pues va a tener mucha relevancia en el cómputo de las heurísticas, y su utilización se tiene que realizar de forma correcta, o es posible que se manden instrucciones equivocadas a los *workers*, produciendo fallos de lógica, como contenedores huérfanos.

Estas estructuras de datos, junto al consumo de la interfaz que proporcionan los *workers* van a constituir el *framework* sobre el que se desarrollarán las heurísticas y políticas de precarga e invocación de funciones.

Heurísticas implementadas

Se han desarrollado tres heurísticas, una que no realiza ningún tipo de precarga de contenedores, una que precarga *runtimes* y una que precarga funciones. A continuación se explicará su funcionamiento detenidamente.

- **No preload**.

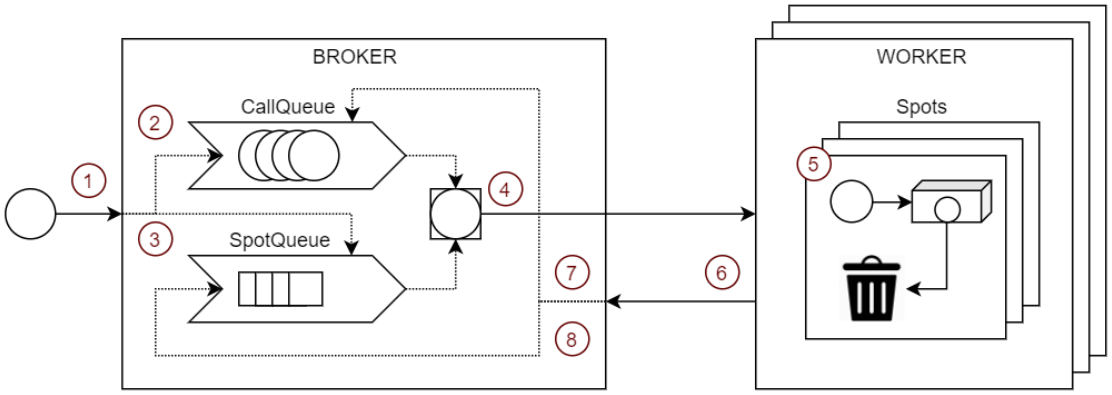


Figura 5.1: Esquema de la heurística "No preload"

Es el primer tipo de heurística que se desarrolló, así como la más sencilla de las tres, ya que no tiene que gestionar contenedores precargados. Los *spots* presentan únicamente dos estados, "ejecutando" y "libre". Su funcionamiento es el mostrado en la figura 5.1:

Cuando se recibe una llamada (1), se comprueba si existe algun *spot* libre, de no ser así se almacena en el *callQueue* (2). Cuando sí existe un *spot* disponible (3), el estado del *spot* se actualiza a "ejecutando", y se manda al *worker* que contenga dicho *spot* la orden para que ejecute la llamada (4). Se realiza la ejecución (5), instanciando el contenedor y eliminándolo al fin de la ejecución. Cuando esta llamada finaliza, el *worker* contacta al *broker* para avisarle de que vuelve a estar disponible (6). En este punto, se comprueba si hay alguna llamada en espera en la *callQueue* (7). De ser así, se asigna directamente al *spot* que se acaba de liberar, para su ejecución (4). En caso contrario, el *spot* vuelve a la cola, a la espera de nuevas llamadas (8).

De esta manera, dos eventos pueden disparar la ejecución de una llamada: la llegada de una llamada nueva, en caso de haber un *spot* libre; o la liberación de un *spot*, en caso de que la *callQueue* no esté vacía.

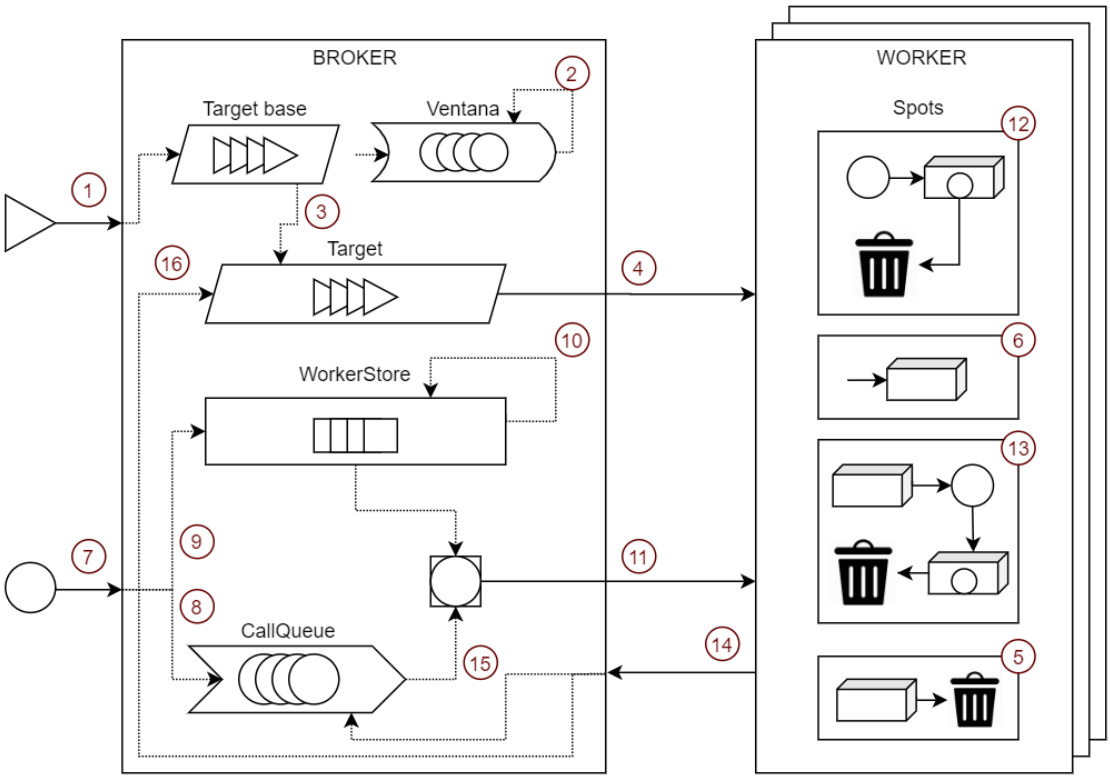


Figura 5.2: Esquema de la heurística "Preload runtime"

- **Preload runtimes**.

  Es una heurística bastante más compleja que "No preload", pues debe tratar con contenedores precargados y sus estados. Los *spots* presentan dos estados, "precargado" y "ejecutando". A parte, hace uso de estructuras de datos adicionales: una array dinámica que se denominará "ventana", para representar el historial de llamadas en los últimos N segundos, así como un "target" que mostrará el objetivo de precarga de *runtimes* en *spots* al que aspira el sistema. Su funcionamiento es el mostrado en la figura 5.2:

  Cuando se registra un *runtime* en el sistema, se actualiza el *target* base (1), que corresponderá al *target* en el caso de que la ventana se encuentre vacía en algún momento. Este *target* base se obtiene repartiendo equitativamente el número de *runtimes* entre los *spots* disponibles. En caso de que haya más *runtimes* que *spots*, los últimos registrados no estarán representados en este *target* base.

  Por otro lado, el funcionamiento de la ventana es el siguiente. La ventana contiene las llamadas recibidas en el sistema en los últimos N segundos. Esta ventana se comprueba cada M segundos, actualizando el *target* a una ponderación de los *runtimes* de las funciones cuyas llamadas se encuentren dentro de ella (2). Esto quiere decir que, si por ejemplo tenemos 10 *spots* y en los últimos N segundos se han recibido 40 llamadas del *runtime* A y 60 del *runtime* B, 4 *spots* deben precargar el *runtime* A y 6 el B. En caso de no haber recibido una llamada en los últimos N segundos, aplicará el *target* base (3) y refrescarán todos los *spots* (4), eliminando su *runtime* actual (5) y precargando los correspondientes (6). Los valores N y M son configurables, a mayor el valor de N, más estable será la precarga, pues tiene en cuenta un mayor volumen de llamadas. Al aplicar valores bajos de N se consigue una precarga centrada en las invocaciones más recientes, lo que le proporciona mayor adaptabilidad.

Cuando se recibe una llamada (7) se pueden dar tres situaciones: la primera es que todos los *spots* se encuentran en estado "ejecutando". En este caso la llamada se encola en *callQueue* (8). Existe una segunda situación en la que hay *spots* en estado "precargado", pero ninguno contiene el *runtime* correspondiente a la función de la llamada. En este caso se selecciona un *spot* precargado arbitrario para la eliminación de su entorno de ejecución (9). Se cambia su estado a "ejecutando" (10) y se manda la función a ejecutar a dicho *spot* (11), que deberá eliminar el *runtime* existente (5), cargar el *runtime* correspondiente y ejecutar la función (12), es decir, nos encontramos frente a un *cold start*. El último caso es aquél en el que el *runtime* correspondiente a la llamada realizada se encuentra precargado. En este caso se cambia su estado a "ejecutando" (10) y se procede a ejecutar la función sobre dicho *spot*, en un contexto de *prewarm start* (13). Una vez finaliza la ejecución, con su correspondiente borrado del contenedor, el *worker* realiza una llamada al *broker* (14), para indicarle dicha finalización. En este punto pueden darse dos situaciones. La primera sucederá si el *callQueue* no está vacío (15). En ese caso, se asigna al *spot* que acaba de finalizar la ejecución la primera llamada en la cola. Ejecutará dicha función como *cold start* (12), cargando el *runtime* en tiempo de ejecución. La otra situación sucede al encontrarse un *callQueue* vacío. En este caso, el *broker* comprobará el *target* (16) y asignará un *runtime* al *spot* (4) para que el *worker* lo precargue (6). Cuando finalice la precarga, el *worker* avisará al *broker* y sólo en este punto se cambiará el estado dicho *spot* a "precargado"(10).

- **Preload functions**.

  El comportamiento de esta heurística va a ser análogo al de "preload *runtimes*". La única diferencia es que el nivel de precarga al que se llegará es al de la función. Las ponderaciones y refrescos de ventana y *spots* suceden de la misma forma, así como todo el flujo de llamadas, actualizaciones del estado de los *workers* y creación y uso de estructuras de datos.

5.1.7. Worker

El desarrollo del *worker* ha seguido los patrones marcados en la fase de diseño. Se ha implementado en *Node*, al igual que el resto de componentes fruto de desarrollos propios. Se ha conseguido que proporcione una interfaz sencilla, con funcionalidades bien definidas, pero potente, con el fin de adaptarse a todas las heurísticas que puedan ser programadas a nivel de *worker*.

Se conecta a canales ofrecidos por el *broker*, cuyo comportamiento es explicado en dicha sección, así como a la *API*, con el fin de recibir los *runtimes* y funciones registrados, así como de mandar los resultados obtenidos de las ejecuciones; y por último a la base de datos, con el fin de persistir el resultado de las funciones y recuperar cierta información en caso de ser necesario.

Su funcionalidad es la definida en el apartado de diseño, siendo capaz de ejecutar funciones desplegando todo su *stack*, precargar *runtimes* y ejecutar a partir de ellos, precargar funciones e invocarlas desde este punto y borrar contenedores precargados.

Una particularidad de la implementación es, por ejemplo, el hecho de que el *worker* debe ser capaz de acceder al cliente de *Docker* de la máquina subyacente, ya que debe ser capaz de lanzar y eliminar contenedores.

Existen algunos puntos presentes en el diseño que no han sido implementados en la solución. El primero es, tal y como se comentaba en las subsecciones que explicaban la *API* y el *blob store*, la recepción de los archivos de función. El *worker* recibe los archivos de función por *SCP* y los descomprime, pues luego tiene que copiar dichos datos a los entornos de ejecución. El problema surge en el momento en el que el *worker* cae. Los *workers* no están preparados para ser relanzados en un sistema en marcha, pues no se han desarrollado las comunicaciones para que el *worker* pida dichos binarios a la *API* (aunque estuviera fuera de sus competencias sería posible realizarlo), o al *blob store* una vez fuera implementado.

Esta es solo una de las partes del problema, pues lo mismo sucede con las estructuras de datos en memoria, no se persisten, por lo que un *worker* no puede recuperar el estado, vital para su funcionamiento, ya que es posible que le lleguen funciones que, al haberse caído y levantado, no conozca. Esto tiene algunas implicaciones obvias. La más grave es que los *workers* no son escalables en tiempo de ejecución, es decir, no se pueden añadir más *workers* una vez el sistema arranca. No obstante, sí que es dimensionable, lo que significa que puede ser arrancado con el número de *workers* que se desee, y éstos serán funcionales mientras no caigan.

La solución a este problema es sencilla, consiste en implementar una rutina en el momento del despliegue de un nuevo *worker* que sea capaz de recuperar el estado de la base de datos y del *blob store* antes de realizar el *handshake* con el *broker* y que sus *spots* formen parte del conjunto de *spots* disponibles.

5.2 Desarrollos complementarios

5.2.1. Despliegue y orquestación

Tal y como se ha podido ver en la sección de desarrollo de componentes, los aspectos relacionados con la alta disponibilidad, la replicación y la tolerancia a fallos han sido algunos de los menos trabajados a la hora de implementar la solución. Esto tiene un impacto directo a nivel de despliegue y orquestación, pues los parámetros de diseño especificados no han podido ser cumplidos dentro del esfuerzo dedicado a este desarrollo.

La solución utilizada para el despliegue y orquestación de la plataforma son una serie de *scripts* realizados en *Bash*. Estos *scripts* eran utilizados dentro del flujo de trabajo de desarrollo, y, realizando una serie de cambios, sirven para poner todo el sistema en funcionamiento. Estos *scripts* toman como partida un manifiesto de despliegue, que describe la topografía del *cluster* o conjunto de máquinas donde se va a realizar el despliegue, así como parámetros de configuración como los puertos de cada uno de los canales, la política de invocación o heurística del *broker*, y el número de *spots* de los que dispone cada uno de los *workers*. Entra en cada una de las máquinas remotas y levanta los contenedores asociados a cada componente. Existe también un *script* para tumbar y limpiar todos los servicios levantados.

Se trata de un mecanismo de despliegue un poco ad-hoc y muy primitivo en comparación a las tecnologías propuestas en la fase de diseño. No proporciona ningún tipo de balanceo de carga, ni de tolerancia a fallos. *Docker* permite el levantamiento automático de contenedores una vez caen. Se han añadido esos parámetros a contenedores que se consideran no conflictivos, como el *registry* o la *API*, aunque es un mecanismo que no se

puede utilizar en aquellos que no están preparados para la reinserción en la plataforma, como es el caso de los *workers*.

La solución a este problema no es muy complicada, pero para que constituya una solución completa debe ser respaldada añadiendo estos factores de replicación y tolerancia a fallos en cada uno de los componentes. Se trataría de crear este despliegue en *Kubernetes*, estableciendo políticas para el escalado y elasticidad de los componentes y añadiendo elementos como el balanceo de carga al enfrentarse a situaciones de tráfico masivo.

5.2.2. Logging

Durante el desarrollo de la plataforma, se hizo uso de la herramienta *Winston*[6] para realizar el *logging* del sistema, con el fin de poder solucionar errores de lógica de una manera sencilla, así como para tener un registro de todas las acciones que sucedían en el sistema. Esta funcionalidad ha persistido hasta la versión final de desarrollo, y es de gran utilidad para conocer el estado de cada uno de los componentes.

Cada uno de los componentes escribe el *log* en un documento dentro de su sistema de archivos. Esto proporciona completitud, pero es un poco tedioso en el caso de, por ejemplo, tener que entrar al nodo de cada uno de los *workers* desplegados para leer su *log*. En las primeras fases del desarrollo, cuando todavía se trabajaba en un único nodo, se realizó una fusión de los archivos de *log* en un mismo lugar. Ésta fusión funcionaba de la siguiente manera: todos los componentes escribían a un archivo de *log* único, que aparecía en su sistema de archivos gracias a un volumen *Docker* compartido.

Esta forma de proceder se vio frustrada por la distribución del sistema en diferentes nodos. La solución al problema es configurar *Winston* para que escriba en un servidor remoto. Es un cambio que no se ha desarrollado, pues no era de vital importancia para el sistema, pero se ha tenido en cuenta y supone un punto a mejorar.

[6] https://github.com/winstonjs/winston

CAPÍTULO 6
Pruebas

Con el fin de asegurarse del correcto funcionamiento del sistema, así como de medir su rendimiento y poder comparar las diferentes políticas de repartición del trabajo se han realizado las siguientes pruebas:

6.1 Validación del sistema

Las pruebas de validación se han llevado a cabo a lo largo del desarrollo del sistema. Dado el carácter de prototipo del que goza el proyecto, no se han tenido en cuenta pruebas muy exhaustivas en cuanto al funcionamiento de la plataforma. Se han realizado algunas, para asegurarse de que funciona en los casos de uso comunes, pero no se han aplicado técnicas de *testing* avanzado ni se han analizado cada una de las situaciones posibles en las que el sistema podría fallar.

El proceso de validación consistió en un *debugging* de desarrollo bastante típico, en el que el programador prueba una funcionalidad después de ser implementada. Ayudándose del *logging* del sistema intenta trazar el problema al punto del código donde sucede, con el fin de resolverlo. Durante el desarrollo, se elaboraron una serie de *scripts* a modo de prueba base automatizada, que se ejecutaban cada vez que se añadía una funcionalidad.

Con el fin de probar todas las funcionalidades implementadas, se crearon dos *runtimes*, uno en *Node* y otro en *Python*, así como funciones con carga de dependencias externas y sin ella para cada uno de estos dos *runtimes*. Esta creación de *runtimes* y funciones era necesaria para asegurarse de que las heurísticas de precarga del *broker* funcionaban correctamente, comprobándolas mediante la monitorización del estado de los contenedores *Docker* de los nodos físicos sobre los que se desplegaba.

En resumen, las técnicas de validación aplicadas son las correspondientes a un desarrollo de este tipo, teniendo que proponer pruebas más exigentes en el caso de querer utilizar el sistema en entornos de producción.

6.2 Pruebas de rendimiento

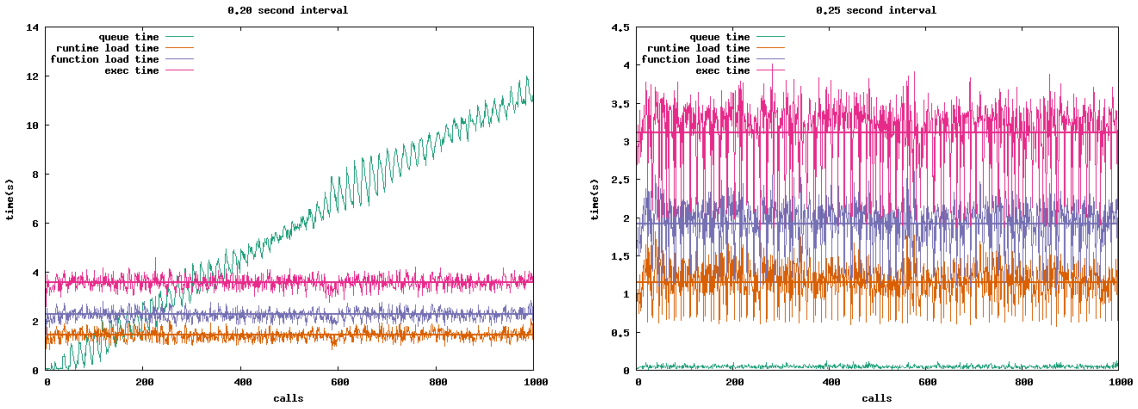
Para comprobar el rendimiento del sistema, se han realizado una serie de pruebas, mostrando cada una de las heurísticas implementadas y comparándolas finalmente. Estas pruebas de rendimiento sirven también a modo de validación, ya que en ellas se muestra como el sistema funciona correctamente en relación a la implementación.

6.2.1. Entorno de pruebas

El entorno de pruebas consiste en un *cluster* de cuatro nodos de ejecución, con un nodo externo dedicado a la gestión. Los nodos de ejecución tienen un procesador con 4 *cores*, 8 GB de *RAM* y el sistema operativo *Ubuntu 20.04*[1]. El nodo de gestión contiene la totalidad de los componentes del sistema, a excepción de los *workers*, que se alojan cada uno en un nodo de ejecución. Estos *workers* se han configurado asignándoles 4 *spots* a cada uno, uno por core del procesador. Esto hace un total de 16 *spots*, y, por tanto, la posibilidad de ejecutar 16 funciones simultáneas en el sistema.

Las pruebas mostradas a continuación se han llevado a cabo ejecutando una función "hello-world" simple en *Node*. Esto permite la comparación directa de las heurísticas sin aspectos externos que alteren los datos. En caso de utilizar otro tipo de funciones o mecanismos se explicará en el punto donde sean usados.

6.2.2. No preload



(a) Carga inadmisible **(b)** Carga admisible

Figura 6.1: Pruebas de rendimiento de la heurística "No preload"

La heurística "No preload" es la menos interesante desde un punto de vista de rendimiento, pues todas las invocaciones van a seguir un mismo patrón y van a devolver resultados similares, pues una vez asignado el *spot* el procedimiento es siempre el mismo.

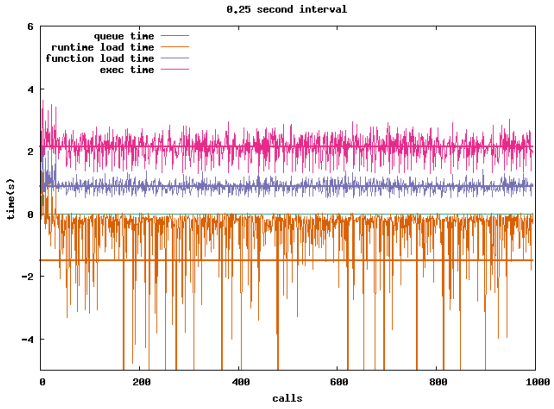
No obstante, puede servir como ejemplo para explicar el dimensionamiento del sistema. Tal y como se explica en la sección 5.1.6, en caso de que se dimensione correctamente la plataforma en función de una carga concreta, nunca se va a tener que utilizar la cola de llamadas, pues siempre debería existir un *spot* libre. Como el sistema desarrollado no cuenta con mecanismos de elasticidad, lo que sucede cuando la carga es mayor de lo que se puede soportar es lo mostrado en la figura 6.1a. Se puede ver como el tiempo de la cola, es decir, el tiempo que esperan las llamadas hasta que se les asigna un *spot*, crece de forma lineal. En este caso, llegan 5 llamadas por segundo al sistema, lo que lo lleva hasta el punto de saturación. Se puede ver que el resto de tiempos, el de carga de *runtimes*, carga de funciones y ejecución se mantienen constantes.

Por otro lado, la figura 6.1b muestra el comportamiento del sistema con un ratio de 4 llamadas por segundo. Al contrario que en el ejemplo anterior, se puede ver que en este caso el sistema puede con la carga, manteniendo el tiempo de cola constante en 0. El resto

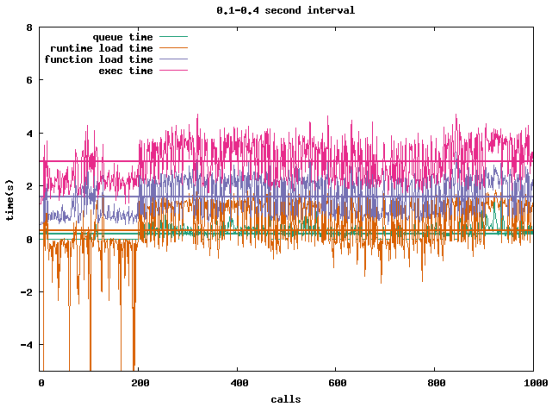
[1] https://releases.ubuntu.com/20.04/

de tiempos son muy similares al ejemplo de la carga inadmisible, simplemente un poco más bajos. Esto puede ser debido al menor *overhead* y tráfico que soportan los nodos.

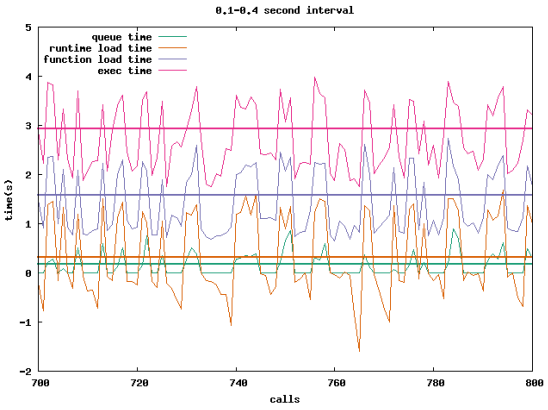
6.2.3. Preload runtime



(a) Flujo constante



(b) Flujo aleatorio



(c) Detalle de flujo aleatorio

Figura 6.2: Pruebas de rendimiento de la heurística "Preload runtime"

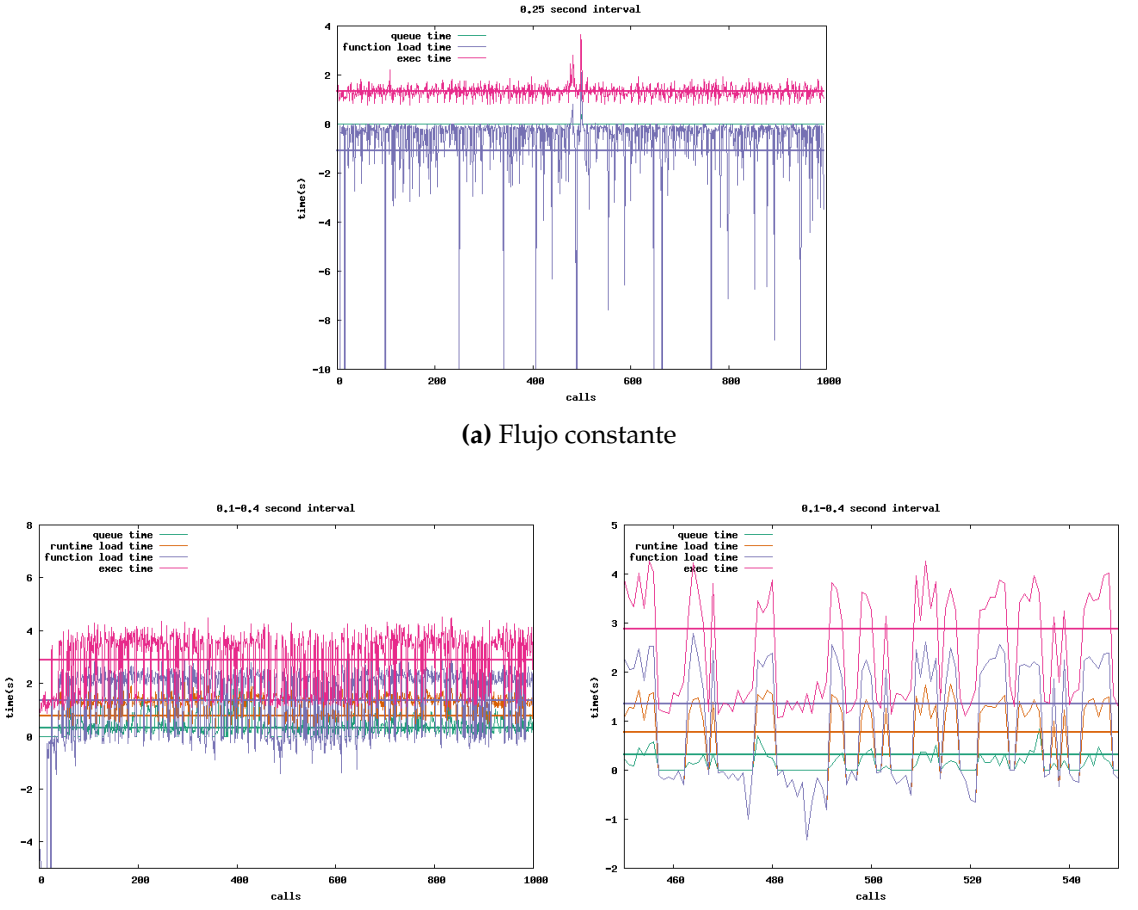
La siguiente heurística a considerar es "'Preload runtime". Ésta, al realizar precargas de entornos de ejecución y utilizar los mecanismos de ventana, comentados en el capítulo 5, supone un punto de vista más interesante a la hora de probar su rendimiento. Para ello se plantearán dos escenarios: uno igual al del tráfico admisible constante visto en la heurística anterior y otro con variabilidad en los tiempos de llamada, para simular un entorno más fiel a la realidad.

El primer caso es el mostrado en la figura 6.2a, con un flujo constante de 4 llamadas por segundo. Se puede observar como los tiempos de cola, carga de función y de ejecución se mantienen constantes, encontrándose la cola en el 0, lo que indica que la carga es admisible. Por otro lado, los tiempos de carga de *runtime* devuelven valores negativos. Esto se debe a que se miden en el momento de levantar el contenedor, y, por tanto, este valor indica el tiempo que el contenedor llevaba levantado cuando se produjo la llamada. Se puede ver como hay valores tan negativos que se salen incluso de la gráfica. Esto se debe a que la rotación de contenedores precargados no es uniforme, de hecho, se realiza siempre siguiendo un orden, se puede entender como una pila. Cuando un *spot*

indica que está precargado se sitúa arriba de la pila. Esto es debido a que éstos se han precargado con el último *target* de la heurística, adaptándose mejor al tráfico.

El segundo caso es el indicado en las figuras 6.2a y, en más detalle en la 6.2c. En este caso se han realizado las invocaciones con una periodicidad aleatoria de entre 0.1 y 0.4 segundos. La gráfica es interesante, pues se puede ver que se aprovecha de los contenedores precargados originales hasta más o menos la llamada número 200, cuando ya empieza a no poder precargar *runtimes* en la mayoría de las invocaciones. Esto es debido a la aleatoriedad del flujo, pues es posible que en ese punto se hayan realizado muchas llamadas en un corto período, lo que ha hecho que la cola de llamadas crezca, inhabilitando la precarga. A partir de ese punto, se puede ver, tal y como indica la gráfica detalle, que se llega a una especie de equilibrio, donde las invocaciones fluctúan entre contenedores precargados y sin precargar. Este equilibrio se puede romper, volviendo a la precarga en caso de tener una racha de llamadas muy separadas temporalmente, o decantándose hacia la no precarga, en caso de que lleguen muchas llamadas juntas y crezca la cola.

6.2.4. Preload function



(a) Flujo constante

(b) Flujo aleatorio

(c) Detalle de flujo aleatorio

Figura 6.3: Pruebas de rendimiento de la heurística "Preload function"

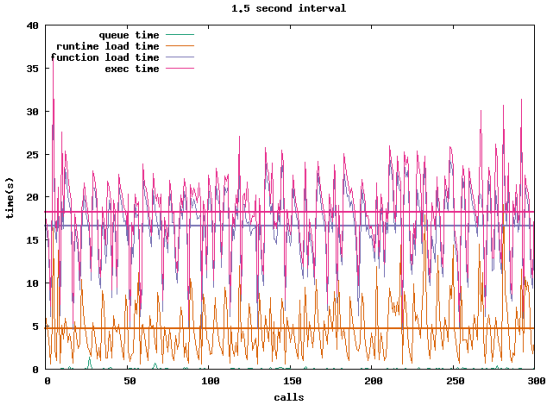
La heurística a considerar en este punto es ”Preload function”. Su comportamiento es análogo al del punto anterior, por lo que sus resultados y peculiaridades se prevé

que se mantengan. Las pruebas realizadas sobre esta heurística serán las mismas que las mostradas en el punto anterior.

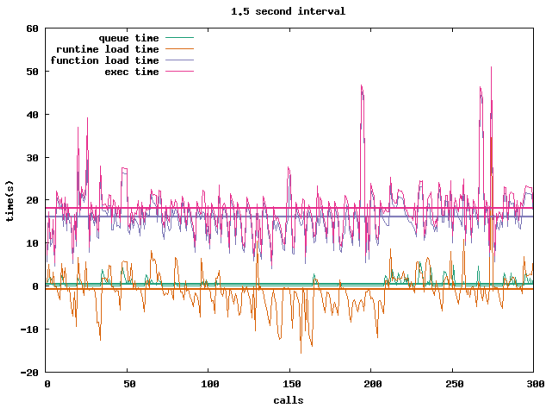
En primer lugar tenemos la invocación constante de 4 llamadas por segundo, mostrada en la figura 6.3a. Se puede ver que la gráfica es muy similar a la encontrada en la sección anterior, a diferencia de que los valores negativos están magnificados en este caso. Esto se debe a que la rotación de precarga de funciones es más rápida que la precarga de *runtimes* y posterior paso de funciones en tiempo de ejecución, por lo que los *spots* que se quedan bajo de la pila tardan más en ser utilizados.

En segundo caso contempla el flujo aleatorio, mostrado en las gráficas 6.3b y 6.3c. La diferencia más considerable respecto a la precarga de *runtimes* es la ausencia del periodo inicial de precargas. En este caso a partir de al rededor de la llamada 30 se puede observar que se entra ya en el equilibrio de invocaciones *prewarm* y *cold*. Esto puede ser debido a que, en valor absoluto, la precarga de la función es más lenta que la del *runtime*. Esto puede suponer que, al llegar una nueva llamada, las precargas no hayan finalizado, haciendo que dicha llamada tenga que esperar un instante en la cola hasta que acabe la precarga. Se puede observar claramente en la gráfica detalle como el tiempo de cola fluctúa entre 0 y medio segundo, así como las diferencias de tiempo entre las ejecuciones *prewarm* y *cold*, que alcanzan los 2 segundos.

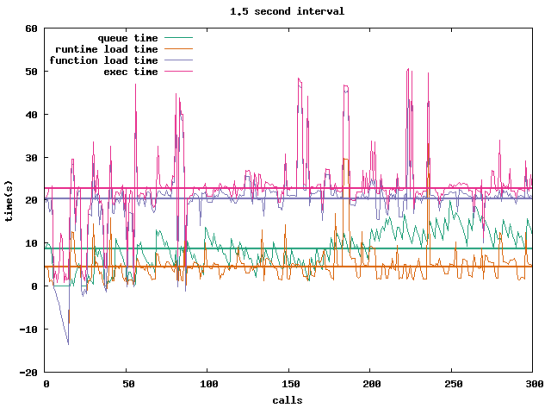
6.2.5. Comparativa



(a) Heurística "'No preload"



(b) Heurística "'Preload runtime"



(c) Heurística "'Preload function"

Figura 6.4: Pruebas de rendimiento con funciones de gran peso

Para esta útima sección se han realizado pruebas sobre una función mucho más pesada, para ver como reaccionaba cada una de las heurísticas a ella. La función comparte el código y *runtime* de las pruebas realizadas anteriormente, no obstante, se ha generado un archivo de 1GB que se pasa a modo de dependencias en tiempo de carga de la función. Las figuras 6.4a, 6.4b y 6.4c muestran su comportamiento en las heurísticas "No preload", "Preload runtime" y "Preload function" respectivamente.

El comportamiento de la primera es el esperado. Existen fluctuaciones en cada uno de los puntos de ejecución, y los *runtimes* se cargan de forma mucho más lenta que en las funciones ligeras, puede ser debido a la sobrecarga que se puede estar produciendo en el disco o sistema de archivos del nodo. la función tarda mucho en cargar pero su ejecución es relativamente rápida.

En el caso de la precarga de contenedores, se puede ver como se están dando *prewarm starts* a lo largo de todas las ejecuciones prácticamente. No obstante, esta precarga no es muy útil, pues pasar la función en tiempo de ejecución ralentiza mucho el proceso, y resulta en un tiempo total elevado.

Por último, la precarga de funciones es inexistente. Es la alternativa que peor rendimiento da, y parece que no puede ni soportar la carga, pues su tiempo de cola va creciendo a medida que avanza el número de ejecuciones. Al igual que sucedía en la gráfica 6.3b, el tiempo de ejecución es mucho más pequeño que el de carga de funciones, lo que hace que las rotaciones sean lentas y que no se pueda dar la situación de precarga.

Heurística		**Intervalo 0.25s**	**Random 0.1s-0.4s**	**Función 1GB 1.5s**
No preload	*runtime*	1.147	-	4.687
	función	1.913	-	16.667
	total	**3.113**	**-**	**18.248**
Preload runtime	*runtime*	-1.495	0.320	-0.772
	función	880.09	1.580	16.187
	total	**2.136**	**2.927**	**18.013**
Preload function	*runtime*	-	0.763	4.417
	función	-1.070	1.345	20.243
	total	**1.315**	**2.879**	**22.737**

Tabla 6.1: Comparativa de tiempos entre las heurísticas

Para finalizar la sección de pruebas, la tabla 6.1 sirve como resumen de los resultados obtenidos en cuanto a rendimiento. Muestra las medias de cada uno de los tiempos obtenidos. Así pues, se puede observar que, en un flujo constante de invocaciones, y considerando que todas aciertan (*prewarm starts*), el mejor rendimiento lo da la precarga de funciones. En cuanto a el caso del flujo aleatorio, la precarga de funciones es también la más rápida, aunque no existe prácticamente diferencia frente a la precarga de funciones. En este punto no se ha probado la heurística "No preload", aunque se pueden tomar sus valores del intervalo 0.25, ya que el tiempo de invocación medio es el mismo. Por último, tratando con funciones muy pesadas, el mejor rendimiento lo proporciona la precarga de *runtimes*.

Por último, no se han realizado comparativas con sistemas existentes, tanto *open source* como privativos sobre *Clouds* públicos. Puede suponer un trabajo futuro interesante, con el fin de cuantificar el rendimiento que se sacrifica por estas mejoras de aislamiento.

CAPÍTULO 7
Conclusiones

Para finalizar la memoria se van a presentar las conclusiones obtenidas. Se introducirá el capítulo con una visión general del proyecto llevado a cabo, se analizará el cumplimiento de los objetivos planteados en la sección 1.2, se hará un repaso de los conocimientos adquiridos durante su realización y, por último se relacionará con los estudios cursados.

7.1 Visión general del proyecto

La solución presentada en esta memoria hace referencia al final del esfuerzo que supone el Trabajo de Fin de Máster y teniendo en cuenta su peso académico y temporal expresado en la guía docente del Máster Universitario en Computación en la Nube y de Altas Prestaciones en la Universitat Politècnica de València.

Desde un punto de vista general, el proyecto se puede considerar satisfactorio. Se ha seguido una buena dinámica de trabajo, siguiendo las pautas marcadas por la planificación de manera correcta, aplicando técnicas contrastadas y definidas, y, finalmente, consiguiendo cumplir los objetivos planteados. En primer lugar, se ha realizado un buen trabajo de documentación, analizando en detalle soluciones relevantes *open source* y tomando como referencia aportaciones académicas, consiguiendo identificar correctamente los problemas de aislamiento que presentan. Seguidamente, se ha propuesto una arquitectura que aborda y resuelve los problemas encontrados en las soluciones disponibles públicamente, utilizando patrones típicos de la computación distribuida y teniendo en cuenta los aspectos y características relevantes en un *FaaS*. Se ha realizado una implementación a modo de prototipo, con el fin de materializar la arquitectura diseñada. Este prototipo, pese a no ajustarse plenamente al diseño, ataca y resuelve correctamente el problema de aislamiento identificado. En último lugar, se han realizado pruebas para medir el rendimiento del prototipo, comparar las heurísticas de repartición del trabajo que propone y validar su correcto funcionamiento.

Dicho esto, el alcance que se ha limitado para el trabajo ha sido erróneo, subestimando el tiempo requerido para realizar cada uno de los objetivos y repercutiendo en la planificación temporal de su desarrollo. Por otro lado, en la fase de diseño, se ha realizado una estimación errónea del tiempo que se iba a tardar en realizar su implementación, lo que ha resultado en la presentación del sistema en el estado en el que se muestra en este documento.

Es por estos motivos por los que el trabajo, dentro del marco definido, se considera satisfactorio, cumpliendo con todos los objetivos propuestos desde un punto de vista general. Sin embargo, la plataforma que se ha obtenido como resultado se presenta en fase

de prototipo, y se requeriría cierto desarrollo complementario para poder considerarse un *software* completo y confiable.

7.2 Cumplimiento de los objetivos

A continuación se realizará una revisión de los objetivos planteados en la sección 1.2. Tal como se comenta en aquel punto, a medida que se ha desarrollado el proyecto estos objetivos se han podido ir alterando. En cada una de las secciones siguientes se explicará en qué nivel se han cumplido, así como las alteraciones que han sufrido.

7.2.1. Identificar los problemas de aislamiento

Este objetivo se ha planteado mediante el estudio en detalle de las plataformas *open source* existentes. Se ha realizado un análisis exhaustivo de aquellas con mayor nivel de relevancia, tal y como se puede ver en el capítulo 2 de la memoria.

Una vez identificado el problema, se ha realizado una crítica, señalando los puntos a mejorar, en el apartado 2.3. Esta mejora consiste, principalmente, en impedir la reutilización de funciones instanciadas para más de una invocación, con el fin de asegurar la consistencia y la seguridad de los datos introducidos. Esto, pese a tener un impacto negativo en el rendimiento de la plataforma, aumenta su aislamiento en gran medida, y por lo tanto se ajusta a la visión que se ha argumentado para el sistema.

Se puede considerar el objetivo como cumplido, pues se han identificado correctamente los problemas de aislamiento que presentan las soluciones actuales y se ha propuesto una solución sobre la que sostener el posterior diseño del sistema.

7.2.2. Diseñar un sistema que integre la solución propuesta

Este objetivo se ha trabajado en el capítulo 4 de la memoria. Se ha partido de la identificación del problema de aislamiento, así como de la arquitectura mostrada por otras plataformas y de los conceptos fundamentales que componen un *FaaS*.

Se han tenido en cuenta muchos aspectos de diseño, que se han ordenado por orden de prioridad y han servido como referencia a la hora de tomar decisiones. Así pues, se realizará un pequeño listado de ellos, que sirva como resumen a muy alto nivel del diseño de la solución.

- **Aislamiento**. Los aspectos más relevantes que se han diseñado en cuanto al aislamiento son el uso de contenedores y la decisión de eliminar cada contenedor una vez finalice la invocación de la función contenida en él. Los contenedores se eligieron como abstracción para los entornos de ejecución, pues debido a su ligereza y versatilidad suponen la opción más adecuada frente a virtualizaciones a más bajo nivel. Respecto a la decisión de no reutilizar contenedores, a pesar de su severidad, es la única forma de asegurar el nivel de aislamiento que se requiere, pues la separación por usuarios no es suficiente para lograr el determinismo en las invocaciones.
- **Rendimiento**. Las decisiones de diseño más relevantes en cuanto al rendimiento de la plataforma son el uso de contenedores, las heurísticas de precarga de contenedores y las tecnologías utilizadas. Los contenedores suponen la alternativa de virtualización para alojar los entornos de ejecución más rápida de desplegar y reiniciar. Respecto a las heurísticas de precarga y políticas de invocación se presenta una interfaz para que se puedan desarrollar e integrar fácilmente en el sistema.

Por último, la propuesta de tecnologías para las diferentes partes del sistema se ha realizado teniendo en cuenta el rendimiento de las mismas.

- **Gestión de recursos**. El aspecto más relevante respecto a la gestión de recursos es la escalabilidad de los componentes del sistema. Esta escalabilidad permite el ahorro de recursos computacionales en periodos de baja demanda, así como el uso al completo de ellos, de la forma más óptima posible cuando esta demanda crece.

- **Alta disponibilidad**. Los dos puntos más importantes que se han tenido en cuenta respecto a la alta disponibilidad son la tolerancia a fallos y la orquestación. La tolerancia a fallos es un aspecto intrínseco de cada componente, que permite su reincorporación al sistema en marcha en caso de que sufra una caída o un problema de lógica. La orquestación utiliza dicha tolerancia a fallos a nivel de componente y la extrapola a nivel de plataforma. Por otro lado, utiliza las medidas de escalado, habilitando la elasticidad del sistema. Por último, permite la configuración de cada uno de los componentes y el despliegue de forma sencilla de la plataforma, facilitando así la labor de los administradores de sistemas.

- **Facilidad de uso**. El aspecto más relevante de la facilidad de uso es la comunicación por parte de los usuarios con el sistema. Para que resulte sencilla a la vez que potente, se han propuesto soluciones a nivel de diseño, tales como interfaces gráficas o *CLI*s.

En base a estas decisiones previamente listadas, se ha realizado un esquema de la arquitectura que debe tener la solución a desarrollar. Por tanto, se puede considerar el objetivo del diseño del sistema en base a la solución propuesta como cumplido, pues este diseño se ha completado, teniendo en cuenta todos los aspectos comentados y explorando en profundidad múltiples opciones.

7.2.3. Implementar un prototipo del sistema

El proceso de implementación realizado es el descrito en el capítulo 5. Tal y como se ha comentado en diversos puntos de la memoria, este desarrollo no se adecua completamente al diseño de la plataforma, teniendo puntos flacos que deben ser cambiados en un futuro, si se pretende utilizar la plataforma en entornos de producción.

No obstante, el sistema funciona a nivel de prototipo, cumple con los requisitos de lo que debe ser teóricamente un *FaaS*, implementa sus funcionalidades básicas y resuelve de forma correcta los problemas de aislamiento identificados. Por tanto, pese a no poder considerarse un *software* completo y final, se puede tratar como una versión intermedia utilizable, comparable a una *alpha*.

Algunos de los puntos a mejorar en cuanto a la implementación son:

- **Persistencia**. Los objetos de función no se guardan en un componente independiente, si no que lo hacen en la *API* y los *workers*. Esto impide que cuando un *worker* cae pueda recuperar dichos objetos y, por tanto, invocar las funciones correspondientes.

- **Escalabilidad**. Los *workers* no son capaces de escalar en tiempo de ejecución, pues no existen las comunicaciones necesarias para que, al incorporarse un nuevo *worker*, sea capaz de obtener el conjunto de *runtimes* y funciones registrados antes de su incorporación. Es por ello que el *worker* tampoco dispone de tolerancia a fallos, ya que levantarse de una caída es un comportamiento simétrico al de una nueva incorporación.

- **Orquestación**. La orquestación que se ha implementado es muy pobre, y no cumple con los requisitos de facilidad de despliegue, tolerancia a fallos y estabilidad que propone el diseño.

En conclusión, se puede considerar el objetivo de desarrollar el sistema diseñado a modo de prototipo como completo, teniendo en cuenta el alcance del proyecto, ya que resuelve el problema de aislamiento propuesto y cumple con las funcionalidades de un sistema *FaaS*. Sin embargo, se requieren ciertas mejoras en un futuro para poder definirlo como una solución completa y fiable.

7.2.4. Realizar pruebas sobre el sistema

Este objetivo se ha llevado a cabo en el capítulo 6. Para su cumplimiento, se han realizado pruebas tanto a nivel de validación del sistema como para comprobar su rendimiento.

Las pruebas de validación de sistemas realizadas han sido llevadas a cabo de acuerdo con el alcance del sistema. Por tanto, han sido pruebas de uso sencillas, relativas al desarrollo de un prototipo. Se han comprobado todas las funcionalidades de la plataforma para casos de uso normales, sin entrar en análisis en mucho detalle ni garantías de robustez.

Por otro lado, las pruebas de rendimiento han sido mucho más completas, realizando experimentos y comparativas sobre cada una de las heurísticas a nivel de *broker* propuestas. Se han relacionado dichas pruebas con la implementación llevada a cabo, comprobando que el comportamiento de las funcionalidades descritas es correcto. La completitud absoluta de estas pruebas no es algo factible, por lo que se podrían considerar otros tipos de experimentos interesantes a realizar. Sin embargo, los planteados son suficientes para mostrar el rendimiento a nivel básico del sistema y su correcto funcionamiento.

En conclusión, se puede considerar el objetivo de probar el sistema implementado para validarlo y medir su rendimiento como completo, pues se han realizado pruebas que tienen en cuenta los dos aspectos, y se han obtenido resultados tangibles que lo demuestran.

7.3 Conocimientos adquiridos

Llevar a cabo este proyecto creo que ha sido muy positivo para mí, pues he tenido la oportunidad de profundizar mis conocimientos sobre el paradigma de computación *serverless*, y más concretamente sobre el concepto de *FaaS*. Esto me ha hecho conocer nuevas plataformas y herramientas, así como publicaciones académicas relacionadas con el tema, lo que ha expandido mi visión del campo.

Pienso que uno de los conceptos aprendidos más relevante es la forma de (auto) gestionar el desarrollo de un proyecto, y más concretamente tratándose de un *software* propio que parte de cero. Desde las fases de concepción y lluvia de ideas hasta la implementación y las pruebas finales es muy importante seguir una planificación estipulada y realizar estimaciones de tiempo. Estas técnicas han sido aplicadas de forma laxa a lo largo del proyecto, siguiendo metodologías más desarrolladas en la parte de implementación, pues eran las que más familiarizadas tenía. Los problemas y soluciones de planificación encontrados a lo largo de este trabajo me servirán para proyectos futuros.

A nivel profesional pienso que el trabajo realizado es beneficioso, pues, a pesar de que conocía con anterioridad las tecnologías y herramientas utilizadas, siempre se adquiere

soltura con ellas cuando se hace uso de ellas a nivel de implementación. Por otro lado, puedo exponer la solución desarrollada en un apartado de proyectos personales a nivel de *portfolio*, así como compartirla siguiendo las pautas del *software open source*, para su uso y modificación por parte de la comunidad.

Por último, a nivel personal, pienso que este trabajo ha sido beneficioso para mí. El hecho de compaginarlo con un trabajo a jornada completa ha sido duro, pero me ha enseñado valores importantes como la gestión del tiempo o de las prioridades personales.

7.4 Relación del trabajo desarrollado con los estudios cursados

Los conocimientos adquiridos en el Máster Universitario en Computación en la Nube y de Altas Prestaciones han supuesto una parte fundamental en el desarrollo de este proyecto. El máster está dividido a nivel académico en dos ramas, que, pese a estar relacionadas, difieren bastante en temática. Esas ramas son, como bien indica el nombre de la titulación la computación paralela y la computación distribuida. El proyecto se engloba dentro de la temática de la segunda rama, y es específico a ésta, por lo que va a estar relacionado casi exclusivamente con conceptos estudiados en las asignaturas de dicho campo.

La concepción del trabajo nace en el proyecto propuesto en la asignatura "Cloud Computing", pues su trabajo final era la implementación de un *FaaS* sencillo en base a un diseño dado. Es por tanto, la materia del máster más relacionada con el proyecto, pues es en la que se explicó el concepto de *FaaS*. Otras dos asignaturas muy relevantes han sido "Diseño y arquitectura de servicios distribuidos escalables" y "Plataformas de gestión de contenedores", pues tienen una relación directa con el proyecto desarrollado. La primera ha sido de gran utilidad en la fase de diseño, sobre todo al medir la escalabilidad del sistema, pues proporcionó conceptos de replicación, redundancia y balanceo de carga, utilizados en la propuesta realizada. La segunda expuso conceptos fundamentales y a bastante bajo nivel del funcionamiento de los contenedores, muy útiles a la hora de trabajar con ellos y montar una arquitectura que los toma como pieza fundamental.

A nivel general, asignaturas como "Conceptos de la computación en grid y cloud", "Infraestructuras de cloud público" y "Seguridad en sistemas distribuidos" han sido de gran ayuda, ya que han servido para conocer mejor el ecosistema *Cloud*, punto de partida y referencia para el proyecto desarrollado. Por otro lado, la asignatura "Fundamentos de los algoritmos distribuidos" ha explicado técnicas utilizadas en la implementación del proyecto, tales como patrones de programación y gestión de la consistencia de los datos en el sistema.

CAPÍTULO 8
Trabajos futuros

Tal y como se ha podido ver a lo largo del documento, existe margen de mejora para la solución desarrollada. Los objetivos del proyecto han sido cumplidos en lo que respecta al alcance del proyecto, tal y como se ha argumentado en el capítulo 7. Sin embargo, hay puntos que no se han podido cerrar del todo, debido al alcance del Trabajo de Fin de Máster, sus plazos y carga lectiva asociada.

A continuación, se hará un resumen de cada uno de los puntos donde existe este margen de mejora, en un orden de prioridad descendente. Estos puntos corresponden a la finalización de la implementación de la plataforma respecto al diseño, la propuesta de nuevas heurísticas para mejorar el rendimiento y, por último, la realización de estudios a nivel de *runtime*.

8.1 Completar la implementación del diseño

La solución desarrollada presenta puntos discordantes con el diseño realizado. En el capítulo 5 se hace referencia a estos puntos, explicando también las mejoras que se deberían llevar a cabo para ajustarse a dicho diseño. Con la finalidad de agruparlos y plantear un plan de ruta definido, se volverán a comentar en esta sección.

Para no dejar el sistema inservible durante el desarrollo de estas mejoras, y, con vistas a que cada una de ellas vaya mejorando su funcionalidad incrementalmente, la planificación seguiría este orden:

- **Implementación del *blob store***. Es el primer aspecto a solucionar, pues de él dependen la tolerancia a fallos y el escalado de los *workers*. El desarrollo del *blob store* consistiría en el despliegue de una instancia de *Minio*, y, según el tipo de comunicación que utilice, el diseño de una interfaz a modo de driver. Sería necesario establecer las comunicaciones necesarias, concretamente con los *workers* y la *API*.

- **Escalado de los workers**. Una vez implementado el *blob store*, ya existen todos los mecanismos de persistencia necesarios para que un *worker* que se incorpore al sistema sea capaz de obtener su estado al completo. Para ello se debe programar que, en la iniciación del *worker*, éste pida los datos que necesite y los almacene en memoria. Cuando esto se haya completado, se podrán añadir nuevos *workers* en tiempo de ejecución, así como volver a introducirlos en el sistema en caso de caída.

- **Orquestación**. Solucionados los puntos referentes al *blob store* y el escalado de *workers*, se procederá a desarrollar la orquestación del sistema. Su implementación solucionará tres puntos fundamentales. En primer lugar la configuración y la facilidad

de despliegue. La orquestación sustituirá al metodo de *scripting* actual, haciendo que sea más sencillo el despliegue por parte de los administradores de sistemas. En segundo lugar, permitirá la elasticidad, pues, al todos los componentes del sistema ser escalables, se podrán desarrollar heurísticas para replicarlos en el momento adecuado, así como para detener instancias no necesarias. Por otro lado, se deberá integrar un balanceador de carga si se pretende tener un escalado masivo (figura 4.2), cosa que permite *Kubernetes* directamente. Por último, se deberán programar dentro de las políticas de orquestación, la manera de tolerar los fallos. Una vez todos los componentes son capaces de reincorporarse a un sistema en tiempo de ejecución, habría que ver en que ocasiones se deben aplicar estas técnicas, e indicarlo así en sus políticas.

- **Interfaz de usuario**. Se trata del último de los puntos a mejorar. Tal y como se podía ver en la sección 5.1, no se ha desarrollado ningún tipo de interfaz para la plataforma, y los usuarios tienen que llamar a la *API* haciendo uso de un cliente genérico. La mejora propuesta consistiría en el desarrollo de dos interfaces, una basada en la línea de comandos y otra gráfica a modo de *web service*. La primera proporcionará funcionalidad para usuarios avanzados, proporcionando un mecanismo de comunicación potente. Su diseño es simple, pues debe consumir la *API*, quizás introduciendo algo de lógica para hacer la vida más sencilla a los usuarios. Por otro lado, la interfaz gráfica deberá seguir un proceso de *frontend*, realizando *mock-ups* de cada una de sus pantallas y validándolas con pruebas empíricas. Su finalidad es ser utilizada por usuarios menos expertos, por lo que existe la opción de recortar algunas funcionalidades complejas para mejorar la facilidad de su uso.

8.2 Mejoras de rendimiento

Una vez completado el sistema en base al diseño, sigue existiendo margen de mejora, aunque este no está directamente relacionado con el desarrollo del *software* en sí, si no que se centra en mejorar su rendimiento haciendo cambios en puntos que están específicamente diseñados para ello, como es el caso de las heurísticas presentes en el *broker*. Por otro lado, el rendimiento del sistema se puede mejorar también desde el punto del usuario, definiendo de forma específica los *runtimes* y las funciones para aprovechar al máximo los *prewarm starts*, es decir, los contenedores previamente cargados. A continuación se explicarán estas dos aproximaciones.

8.2.1. Proponer nuevas heurísticas

La utilización de nuevas heurísticas puede suponer una mejora en el rendimiento del sistema. Podemos diferenciar dos tipos de heurísticas: las que operan a nivel de *broker* y las que operan a nivel de orquestación.

Las primeras pueden ser modificadas bien alterando las métricas sobre las que se toman las decisiones o el resultado de dichas decisiones, es decir, el tipo de precarga a realizar. Algunos ejemplos de métricas a considerar pueden ser el porcentaje de uso de *CPU* de cada *worker*, el peso en disco de las funciones o patrones previos de invocación. En cuanto a el tipo de precarga, se puede plantear una precarga mixta, pudiendo contar con *runtimes* y funciones precargadas al mismo tiempo en *spots* diferentes. Cada una de estas heurísticas va a funcionar mejor o peor según el caso de uso al que se someta el sistema. No existe una aproximación perfecta, pero resulta interesante realizar una exploración de heurísticas, y, la interfaz que proporciona el *worker* para llevarla a cabo facilita mucho esta tarea.

El segundo tipo de heurística va a tener una relevancia más a nivel de gestión de recursos que de rendimiento. Se deberá decidir, tomando como referencia bien métricas recolectadas por parte del orquestador de las máquinas físicas activas o datos proporcionados por los propios componentes, en qué momento levantar nuevas réplicas o eliminar existentes. El componente más relevante en este aspecto serán los *workers*, pues al resto de herramientas no se le va a realizar nunca un escalado hacia abajo. Al igual que sucede en el caso del *broker*, las métricas escogidas se tienen que adaptar al caso de uso, y no existe una solución única. A nivel de toma de decisiones, en el caso concreto de los *workers*, el orquestador sólo debe decidir si levantar más instancias, eliminarlas o dejar el número existente

8.2.2. Realizar estudios a nivel de runtime

Otra mejora a considerar para el rendimiento de la plataforma es el dominio del contenido de los *runtimes* y las funciones. Como estos dos objetos son definidos por el usuario, este tipo de exploraciones son externas a la plataforma, y se pueden realizar sin alterarla, no obstante, quedan supeditados a las heurísticas a nivel de *broker* implementadas en ella.

El planteamiento de estos estudios es sencillo. Un *runtime* consiste en el conjunto de archivos y procesos para que se pueda interpretar la función. El ejemplo más básico es la instalación de un lenguaje de programación, pero partiendo de ese punto se puede extender, pudiendo incluir dependencias o incluso funciones. Esto implica que, para funciones con dependencias diferentes pero que utilicen el mismo lenguaje de programación existen dos escenarios: el registro del *runtime* básico y el paso de dependencias a nivel de función o la creación de un *runtime* con todas las dependencias de todas las funciones incluidas en él y el registro de las funciones sin dependencias. Este modelo sería extensible hasta el punto de tener las funciones incorporadas en el *runtime* y que los objetos de función registrados sólo fueran *handlers* para la selección de la función en concreto y el paso de sus parámetros.

Tomando un punto de vista mas analítico, existen dos variables a maximizar: el tiempo de precarga de los contenedores y el *hit rate* (número de precargas que aciertan y ejecutan la función). En el caso de tener un *runtime* omnipotente, que contuviera todas las funciones registradas en la plataforma, su *hit rate* sería perfecto, acertando todas las funciones, pero su tiempo de precarga muy pobre, pues se trataría de una imagen muy pesada al contener todo el código y dependencias de todas las funciones. El caso contrario sería la precarga de las funciones de forma individual, con *runtimes* que solo contengan el lenguaje de programación y, por tanto funciones que únicamente contengan sus dependencias. Se puede estimar una carga de entornos de ejecución muy rápida pero un *hit rate* pobre en estos casos.

A parte de estos dos ejemplos anteriores, existen infinidad de casos intermedios, como múltiples *runtimes* con niveles diferentes de dependencias incorporados y funciones que se adaptan a ellos. El usuario deberá conocer el tipo de carga de trabajo que va a ejecutar en la plataforma y definir sus objetos en consecuencia.

Bibliografía

[1] Oliver Stenbom. Refunction: Eliminating Serverless Cold Starts Through Container Reuse. *Imperial College London* 2019.

[2] Eric Jonas et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *UC Berkeley* 2019.

[3] Simon Shillaker, Peter Pietzuch. FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing. *Imperial College London* 2020.

[4] Mohammad Shahrad et al. Architectural Implications of Function-as-a-Service Computing. *Princeton* 2019.

[5] Liang Wang et al. Peeking Behind the Curtains of Serverless Platforms. *USENIX ATC* 2018.

[6] Manuel Stein. The Serverless Scheduling Problem and NOAH. *Nokia Bell Labs* 2018.

[7] Ping-Min Lin, Alex Gikson. Mitigating Cold Starts in Serverless Platforms. *Carnegie Mellon University* 2019.

[8] Documentación de *OpenFaaS*. Consultado el 15/11/2020 en https://docs.openfaas.com/

[9] Documentación de *OpenWhisk*. Consultado el 13/11/2020 en https://openwhisk.apache.org/documentation.html

[10] Documentación de *Kubeless*. Consultado el 17/11/2020 en https://kubeless.io/docs/

[11] Documentación de *Fission*. Consultado el 15/11/2020 en https://docs.fission.io/docs/