UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Sistema escalable de ingreso de peticiones para servicios en la nube

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Cristina Gaspar Rozalén
Tutor: José Manuel Bernabeu Aubán
Tutor externo: Juan José Valero Barjola

Curso 2019-2020

Resumen

En la construcción de aplicaciones en la nube actualmente se suele emplear una arquitectura basada en microservicios. En estas aplicaciones parte de los microservicios son puntos de entrada a la aplicación desde el exterior, típicamente HTTP. Permitir este acceso lleva a enrutar a quien lo demanda hacia el componente deseado mediante proxies balanceadores de carga.

Trabajando en entornos cambiantes, como lo son las aplicaciones desplegadas en la nube, debemos tener en cuenta ciertos aspectos. La cantidad de servicios que hacen uso del balanceador varía con el tiempo, al igual que la carga de balanceo de cada uno de estos servicios. Además, los servicios deben cumplir los acuerdos a nivel de servicios a los que se han llegado con el cliente.

En el presente trabajo de fin de grado se propone la implementación de un sistema distribuido de balanceadores de carga, encargados en su conjunto de distribuir el tráfico entrante de un conjunto de servicios, de forma altamente disponible y escalable, y automáticamente configurable y regulable en base al estado y requerimientos de las aplicaciones a las que sirve; compatible con entornos *Kubernetes*, dado que es habitual orquestador en el despliegue de aplicaciones en la nube.

Para el desarrollo, se ha escogido *Envoy Proxy* como balanceador de carga. A este se le acompañará de un servidor al cual podrá hacer consultas. En lugar de emplear APIs de tipo REST en ellos, se ha escogido comunicación RPC empleando gRPC y el uso de *Protocol Buffers* por su eficiencia en la serialización de datos para el intercambio de mensajes. Como lenguaje de programación se ha escogido *Go*. Posteriormente justificaremos por qué estas tecnologías.

Palabras clave: escalable, microservicios, balanceador de carga, nube, puntos de acceso

Abstract

In the contruction of cloud applications, a microservices-based acrchitecture is often used nowadays. In these applications, part of the microservices are entry points to the application from outside, typically HTTP. Allowing this access leads to routing whoever requests it to the desired component through load balancing proxies.

Working in changing environments, such as cloud applications, we must take into account certain aspects. The number of services that make use of the balancer varies over time, as well as the load balancing charge of each of these services. Furthermore, the services must comply with the service level agreements reached with the client.

In this thesis, the implementation of a distributed system of load balancers is proposed, as a whole responsible of distributing the incoming traffic of a set of services, in a highly available and scalable way, and automatically configurable and ajustable based on the status and requirements of the applications it serves; campatible with *Kubernetes*, as it is usual for the deployment of cloud applications.

For development,*Envoy Proxy* has been chosen as the load balancer. It will be accompanied by a server that it can make requests. Instead of implementing a REST API, RPC communication has been chosen using gRPC and Protocol Buffers for the exchange of messages, due to its efficiency at serializing data for message exange. Go has been chosen as the programming language. Later we will justify why these tecnologies.

Key words: scalable, microservices, load balancer, cloud, front-ends

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

De acuerdo con el *National Institute of Standards and Technology*, la computación en la nube es un modelo que habilita el acceso a través de la red y bajo demanda a un conjunto compartido de recursos que se pueden provisionar y lanzar rápidamente con mínimo esfuerzo o interacción del proveedor [10].

En una definición más popular, la nube no deja de ser otra cosa que Internet, y hablar de computación en la nube es hablar de llevar recursos de un entorno particular a uno común para poder ejecutar software. Estos recursos pasan a ser servicios que se pueden ofertar de forma pública o a un grupo privado.

Para llevar software a la nube primero hay que virtualizar recursos, es decir, simular el software y/o hardware sobre el que otros programas se ejecutan [12].

Un usuario no tendrá por qué adquirir máquinas físicas, configurarlas y desplegar sus aplicaciones sobre ellas. Simplemente podrá hacer el proceso de despliegue por internet en una máquina física de un tercero, quien la configura y mantiene.

Con esto se consigue que los clientes finales de las aplicaciones o servicios puedan consumirlos remotamente, gracias a internet. Por ejemplo, que puedan acceder a una página web.

Mediante la virtualización podemos aislar un sistema operativo dentro de otro, como una muñeca rusa. Tendremos una máquina con su sistema operativo «propio», dentro de la cual podemos ejecutar otro sistema quedando encapsulado, la muñeca pequeña dentro de la grande.

Hay varios programas que permiten obtener este resultado, y en cada uno se le da un nombre distinto al sistema encapsulado. Uno de esos programas es Docker y el nombre que le da a este tipo de sistemas es contenedor [9].

Así pues, los contenedores son una manera de combinar un sistema operativo con aplicaciones empaquetadas. Con ellos se obtiene una forma portable, reusable y automatizable para empaquetar y ejecutar aplicaciones [13]. Este método permite ejecutar en un servidor una aplicación para que quede accesible a través de la nube.

Hecha esta aclaración, nos encontramos en un entorno virtualizado en el cual tenemos varios contenedores. En ellos se están ejecutando los servicios de los que el cliente final quiere hacer uso.

Para poder gestionar la navegación a estos servicios, tenemos que saber dónde se alojan los contenedores que los contienen y ejecutan. En un entorno en el que la cantidad de contendedores es fija podríamos indicarlo estáticamente, escrito en un fichero que no se modifica. Pero puede que el entorno cambie, lo que quiere decir que pueden crearse o destruirse contenedores, con lo que dejar esta información fijada ya no es recomendable.

Por ejemplo, en un momento dado cierto servicio podría estar muy solicitado. Por ello se crearán nuevas instancias del servicio. Si no las tenemos registradas no servirán de nada, porque no podremos redireccionar flujo a ellas.

En este trabajo vamos a tratar la orquestación de arquitecturas basadas en microservicios. Esta arquitectura, como su nombre indica, consiste en pensar en que la aplicación que queremos crear está formada por un conjunto de pequeños servicios, y desarrollar cada uno de estos por separado para que puedan trabajar conjuntamente.

Un ejemplo simple sería el caso de una página web. La podemos entender como dos servicios, uno de ellos frontal que ofrece la parte visual con la que el usuario interacciona, y el otro sería una parte trasera que realizaría la mayor parte del tratado de datos.

Dicho esto, proponemos una solución para el proceso de descubrir en que puntos finales (conocidos como *endpoints*) se encuentran los servicios y distribuir la carga de las peticiones entrantes entre ellos.

Es decir, proponemos un controlador de peticiones de ingreso. A nuestra solución se la ha llamado *Amphitrite*, y a partir de ahora nos referirémos a ella con este nombre.[1].

Antes de continuar, hacemos saber que al final de este documento encontrará una sección glosario, en la cual se pueden encontrar diversos términos que vamos a emplear en diversos puntos. Así, en caso de desconocimiento o duda, se podrán consultar fácilmente.

1.1 Motivación

Cuando un cliente quiere consumir un servicio de una aplicación se deben pautar una serie de pasos de navegación. Por ejemplo, qué dominio y ruta está asociado al servicio, o mediante qué puerto se accede.

También caben ciertas condiciones previas, como podría ser que el demandante esté autorizado a realizar su petición. Típicamente se recurre a un *proxy*, un intermediario, que es quien realizará y gestionará esta lógica.

En entornos cambiantes como son los de aplicaciones desplegadas en microservicios, en los cuales crean y/o destruyen réplicas de estos, a este intermediario también se le da la tarea de distribuir las distintas peticiones entre las réplicas en base a un patrón de orden, que es lo que se conoce como balanceo de carga.

Pero no solo el entorno de los servicios cambia. También el de las peticiones, razón por la que es conveniente que el intermediario sea escalable.

De esta forma podrá atender a esa cantidad variante sin caer en cuellos de botella si, por ejemplo, recibimos de golpe más peticiones de las que por recursos de hardware o capacidad de la red es capaz de atender.

Lo que nos ha llevado a realizar este trabajo es resolver estos retos y otros que puedan surgir, para ofrecer una aproximación alternativa a los controladores de peticiones de ingreso ya existentes y que pueda aportar ciertas mejoras.

Este proyecto se realizó conjuntamente a la empresa *Kumori Systems*, una spin-off de la UPV que desarrolla una Plataforma Como Servicio (PaaS) basada en el orquestador provisto por *Kubernetes*.

[1]Como apunte anecdótico, el nombre se escogió debido a las metáforonas marinas, uso del griego y referencias mitológicas que algunos sistemas *cloud* usan en sus nombres.

Por ejemplo, *Kubernetes* e *Istio* son palabras que vienen del griego y significan, respectivamente, timonel y navegar. Anfitrite (*Amphitrite* en inglés) es la diosa griega del mar tranquilo y esposa de Poseidón.

1.2 Objetivos

El objetivo principal del trabajo es implementar un sistema distribuido de balanceadores de carga. Lo que buscamos es que distribuyan el tráfico entrante a un conjunto de servicios.

Esto conlleva una serie de puntos que debe cumplir. No podrá balancear nada si no sabe con qué tiene que trabajar. De alguna manera, tendremos que hacerle saber al balanceador qué rutas debe llevar a qué servicios.

Pero además, esos mismos servicios pueden replicarse al escalar; pueden crearse o destruirse. Habrá que saber cuándo ocurren estos cambios e informar de ellos propiamente.

Por tanto, sintetizando lo comentado, en el producto final del trabajo hay una serie de hitos que se quieren conseguir. Los objetivos son:

- El sistema debe poder saber dónde se encuentran los servicios a los que tiene que redireccionar.
- Debe poder adaptarse a los cambios en su entorno dinámicamente.
- Permitir por arquitectura el escalado horizontal frente al aumento o disminución del número de peticiones. Nuestra solución será «otro servicio más», por lo tanto debe permitir ser escalado según convenga.

1.3 Estructura de la memoria

La memoria del trabajo se encuentra dividida en los siguientes capítulos:

Estado de la cuestión: hablamos de las opciones disponibles que ofrezcan características similares a las que se quieren conseguir y por qué no se han escogido.

Análisis del problema: se identifican las distintas soluciones posibles y se explica la electa, además de un modelo conceptual del proyecto.

Diseño de la solución: detallamos el diseño y explicamos las tecnologías elegidas y por qué.

Desarrollo de la solución propuesta: explicación de las distintas fases de desarrollo del proyecto.

Implantación: tratamos el encapsulamiento de los distintos elementos del proyecto en contenedores, su alzado y la comunicación entre ellos.

Pruebas: se detallan las pruebas realizadas.

Conclusiones: comentarios finales sobre el proyecto y la relación de su desarrollo con los estudios cursados.

Posible extensión del trabajo: comentario sobre los posibles distintos caminos que pueda tomar el proyecto, así como otros proyectos que surjan a partir de este.

Referencias: listado de las referencias bibliográficas empleadas a lo largo de la memoria.

Apéndice: información adicional del proyecto, como los ficheros de configuración.

CAPÍTULO 2
Estado de la cuestión

El control de peticiones de ingreso no es un problema nuevo; ya existen opciones que lo tratan. Las más recientes se centran en ofrecer soluciones para *Kubernetes*, un sistema orquestador de contenedores (un gestor) [4].

Kubernetes fue creado por Google y actualmente mantenido por la *Cloud Native Computing Foundation* (CNCF), una fundación con el objetivo de ayudar al avance de la tecnología de contenedores.

Así pues, entre los controladores de ingreso actualmente disponibles vamos a tratar dos en concreto. Por un lado encontramos *Ambassador Edge Stack*, un controlador basado en *Envoy Proxy* como *proxy* balanceador de carga. Por otro lado, hablaremos de *Istio*, un sistema más completo y que también emplea el mismo *proxy*. Antes de continuar, vamos a pararnos a explicar en términos generales qué es este balanceador del que hablamos. *Envoy Proxy* es un *proxy* que resuelve el balanceo de peticiones, ofrece un medio de comunicación entre microservicios y además nombra sus recursos independientemente de su localización u origen.

Comenzando, *Ambassador Edge Stack* permite manejar la relación entre usuarios finales y *Kubernetes*. Ambassador es el encargado de descubrir y comunicarle a *Envoy* dónde están los servicios que nos interesan.

Envoy entonces ejerce como intermediario para ofrecer un único punto de entrada para los clientes finales a los servicios que quieran acceder.

De forma más simplificada, ofrece una dirección para que los clientes, a través de ella, accedan a unos servicios determinados.

Siendo una solución en principio adecuada a lo que buscamos, cuenta con la desventaja de que no se puede reconfigurar en caliente. Por lo tanto, si queremos cambiar algo de su propia configuración, tendremos que reiniciar el sistema.

Istio por su lado ofrece gran cantidad de prestaciones. Puede controlar el tráfico de entrada y de salida (tanto del cliente final al servicio como entre servicios), controlar las llamadas a servicios, balancear peticiones, monitorizar y desplegar servicios; entre otras características.

Resulta un sistema muy complejo y grande. En muchas ocasiones todas las propiedades que ofrece quedarán sin darles uso, y probablemente gastando recursos innecesariamente.

Nosotros proponemos *Amphitrite* como una solución intermedia entre ambos sistemas comparados: ofrecemos una solución más flexible que *Ambassador* pero sin llegar a sacrificar su simplicidad, mejorando en este aspecto frente a *Istio*.

CAPÍTULO 3
Análisis del problema

A continuación hablaremos de los distintos puntos que deberá contemplar la solución escogida, proceso que se realizó para poder crear el prototipo arquitectónico de la solución propuesta, de la que posteriormente hablaremos.

Tenemos claro que el problema general a resolver es poder dirigir peticiones de clientes finales hasta el servicio que desean consumir. Pero este cuenta con una serie de problemas hijo que debemos tratar.

En primer lugar, en un sistema de balanceadores de carga se debe escoger el balanceador que se va a emplear. Este debe poder empaquetarse en contenedores y permitir ser escalable. Además, sería conveniente que disponga de un sistema el cual le permita comunicación externa para comunicarle dinámicamente la información que debe manejar.

Se podría conseguir mediante la lectura de un fichero de configuración en el cual se le indicará toda la información necesaria para su funcionamiento, pero esto implicaría bien tener que reiniciar el sistema cada vez que el entorno de los servicios que debe manejar cambie. Lo cómodo sería que pueda adaptarse dinámicamente a las variaciones.

Por ello, debe haber un mecanismo que permita descubrir los nuevos servicios, o bien réplicas, que se creen o se destruyan y sea capaz de notificarlo al balanceador para que este se adapte al cambio en caliente.

Como otro requisito no funcional, queremos que el entorno cloud de la empresa pueda hacer uso de esta solución. Adicionalmente, es conveniente que a su vez pueda adaptarse a otras plataformas de orquestación, como entornos *Kubernetes*.

3.1 Solución propuesta

De acuerdo a esta base, se tomaron las decisiones para la arquitectura de nuestra solución: *Amphitrite*.

La primera fue escoger como balanceador *Envoy Proxy*. Se estudiaron otras qué otras opciones estaban disponibles y decidimos comparar entre *HAProxy*, *Nginx* y el escogido.

La razón para decantarnos por él es que en un inicio se creó ya pensado para microservicios y nos da todas las características que buscamos. Dispone de mecanismos para descubrir recursos dinámicamente y una API a la que podemos comunicar los cambios del entorno, por lo que es dinámicamente (y altamente) configurable.

Adicionalmente ofrece todo tipo de métricas, cosa que pese a no estar entre los objetivos principales del proyecto es un buen añadido.

Otro factor que tuvimos en cuenta fue el económico. Mientras que nosotros escogimos una opción gratuita, la opción completa de *Nginx* no lo es y tiene, a groso modo, características similares a *Envoy*.

HAProxy también es gratis pero, de nuevo, sus características son similares y en algunos casos podríamos decir que el equipo desarrollador de *Envoy* actúa antes para adaptarse a cambios tecnológicos.

Hecha esta aclaración, *Envoy* es capaz de descubrir recursos tanto leyendo un fichero estático como dinámicamente a través de uno o varios servidores. Esta es la opción por la que optamos.

Al conjunto de esos servidores y sus correspondientes APIs se le llama xDS, siglas que significan *x Discovery Service*. Hace referencia a «cualquier servicio de descubrimiento» [1].

Con ello nos referimos a que cada API del servidor xDS está encargada de recoger un tipo de recurso *Envoy* en concreto. Las APIs son:

- **LDS** *(Listener Discovery Service)*: encargado de los recursos de *listeners*, es decir, la configuración relacionada con los las direcciones en las que *Envoy* debe atender peticiones de los clientes finales y diversas opciones de control.
- **RDS** *(Route Discovery Service)*: servicio que recoje la configuración relacionada con las rutas que se deben redireccionar.
- **VHDS** *(Virtual Host Discovery Service)*: relacionado con los recursos tipo *virtual hosts*, un nombre con el que encapsular un conjunto de dominios y sus respectivas rutas.
- **CDS** *(Cluster Discovery Service)*: servicio mediante el cual descubir la configuración de los *clusters* en los cuales se alojan los servicios que se quieren manejar.
- **EDS** *(Endpoint Discovery Service)*: mediante él se puede descubir la configuración de los puntos finales o *endpoints* de los servicios a los que se quiere enrutar al cliente final y sus réplicas. Conocer este recurso es útil de cara al propio balanceo de carga.
- **SDS** *(Secret Discovery Service)*: encargado de descubrir los recursos relacionados con certificados, tiques de sesión y demás elementos de control, autenticación y validación.
- **RTDS** *(Runtime Discovery Service)*: mediante este se pueden descubrir recursos con los cuales describir una capa en el sistema de ficheros virtual de ejecución.

Para el interés de nuestro proyecto nos hemos centrado en los servicios LDS, RDS, CDS, EDS. Estos cuatro son pues las APIs que comprende en nuestro caso el servidor xDS, lo que significa que serán sus respectivos recursos los que buscaremos descubrir dinámicamente.

No vamos a hacer uso del RTDS porque no tenemos interés en que el *proxy* gestione ficheros más allá de los que requiera para su funcionamiento o para registro de operaciones. En el caso del SDS, en nuestra lógica y diseño se tomó la deción de acompañar los recursos de tipo *cluster* de los certificados asociados, si es que aplica.

Dada la estructura de este proyecto, solo necesitamos un único servidor de control xDS por réplica de *Envoy*. Siguiendo lo especificado en su documentación, se escogió desarrollar un servidor ADS (*Aggregated Discovery Service*).

Es así como conseguimos implementar la comunicación entre balanceador y servidor en un único flujo. De esta forma se permite el múltiple envío tanto de peticiones como

respuestas de descubrimiento de recursos, especificándolo en un campo del mensaje de petición [2].

3.1.1. Arquitectura del proyecto *Amphitrite*

Hecha esta aclaración, vamos a tratar los componentes que forman nuestro proyecto. En su conjunto, la arquitectura es la que se muestra en la Figura 3.1. Está formada por dos elementos generales: el principal llamado *Amphitrite* y el informador de recursos *Amph-Informant* (o *Informant* abreviado).

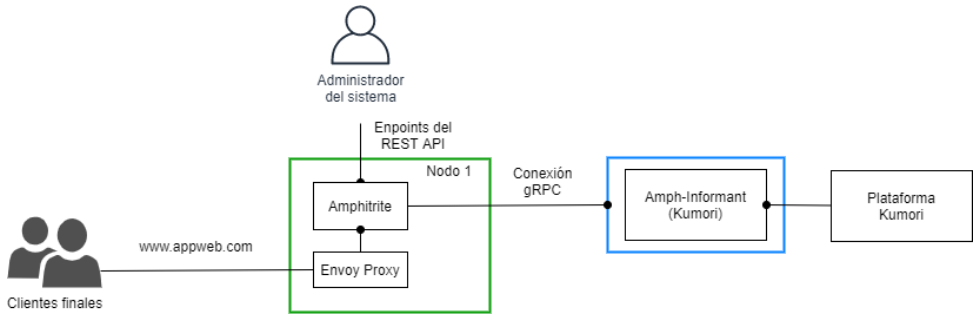

Figura 3.1: Arquitectura formada por dos elementos generales: *Amphitrite* y *Amph-Informant* (o *Informant* abreviado). En este ejemplo tenemos una instancia del primer componente junto a *Envoy*, y una instancia de un *Amph-Informant* para entornos *Kumori*

El componente principal, *Amphitrite*, está formado por una instancia de *Envoy* y una del servidor ADS que hemos desarrollado. De esta forma, *Envoy* le pedirá a nuestro servidor la información sobre los recursos que necesita.

Previamente le habremos indicado en la configuración del *Envoy* qué tipo de recursos son los que va a querer descubrir dinámicamente. En nuestro caso, como ya hemos comentado, serán *listeners, clusters, endpoints* y rutas. El servidor ADS responderá con la configuración de aquellos recursos que conoce.

La forma que tiene el ADS de conocerlos es a través del componente *Amph-Informant*. Recordemos que el servidor ADS es una parte de *Amphitrite*. Pues este componente tiene otro elemento al cual *Amph-Informant* comunica lo que sabe sobre recursos en los que estamos interesados.

Esta pieza es la encargada real de recabar esa información. Se ha decidido desacoplar su lógica del propio componente principal para que este tenga menos carga y, además, separar funcionalidades.

Se comporta como intermediario entre el lugar donde se encuentran los servicios que vamos a tratar y *Amphitrite*, haciendo de traductor para adecuar los datos recabados al formato en el que el ADS es capaz de leerlos.

Dado que *Amphitrite* puede tener varios suministradores, el componente informador cuenta con un adaptador para cada uno. Más tarde trateremos este tema en profundidad.

Por ahora queremos indicar que tendremos como resultado un *Amph-Informant* por cada entorno. En nuestro caso sería uno para entornos *Kubernetes* y otro para el entorno de la empresa *Kumori*.

Cada entorno comunicará lo que para él es necesario para permitir el acceso a los servicios que aloja. Ahora esta frase puede sonar general, pero en el próximo capítulo la desarrollaremos en profundidad.

La intención es que el *Informant* sea un componente genérico y pueda interactuar con distintos orquestadores. En este proyecto se ha escogido trabajar con dos, pero hemos querido dejar la arquitectura preparada por si en el futuro se quieren añadir otros.

Resumiendo, *Envoy* se ejecutará, verá qué elementos debe conocer dinámicamente y comenzará a pedirlos al ADS de nuestro componente *Amphitrite*. A su vez, este sabrá datos sobre ellos gracias a que el otro componente, *Amph-Informant*, le notificará aquellos recursos que el entorno en cuestión le haya comunicado.

CAPÍTULO 4
Desarrollo de la solución propuesta

Llega el momento de tratar lo que ha supuesto el desarrollo de *Amphitrite*. Trateremos las herramientas y tecnologías empleadas para el proceso, así como la preparación del entorno de ejecución del proyecto.

4.1 Tecnologías y herramientas empleadas

A continuación, vamos a hablar del lenguaje escogido, las dependencias que hemos importado para él y del entorno de trabajo para el desarrollo.

4.1.1. Comunicación entre componentes: *gRPC* y *Protocol Buffers*

Yendo por partes, *gRPC* es un sistema de llamada a procedimiento remoto (RPC) de código abierto desarrollado inicialmente por Google. Puede utilizar distintos transportes, aunque la implementación más madura y utilizada sea la basada HTTP/2, y Protocol Buffers como lenguaje de descripción de interfaz de los mensajes y métodos[1].

En términos generales este sistema está pensado para que, en lugar de hacer una petición a una ruta expuesta por el servidor como se haría en una REST API, se invoque directamente a un método implementado en el servidor. Da la sensación de ser un método propio del cliente.

Respecto a los *Protocol Buffers*, son un mecanismo extensible, neutral respecto a plataforma y lenguaje. Fue desarrollado por Google para la serialización de datos estructurados [7].

Se ha decidido emplear comunicación *gRPC* por tres principales razones. La primera es que se decidió emplear *Protocol Buffers* debido a que es más rápido que otros métodos de serialización como XML [6].

La segunda es a raíz de la anterior, y es que *gRPC* está principalmente pensado para ser empleado junto a *Protocol Buffers* por defecto [8]. Por lo tanto, vimos conveniente emplear este sistema de comunicaciones.

La útlima se debe a que es un sistema que permite trabajar con diversas tecnologías y lenguajes. Como puede trabajar con HTTP/2 puede interactuar con la mayoría de las plataformas de despliegue de servicios, que, usualmente, exponen sus *endpoints* mediante HTTP.

[1] https://grpc.io/

Además, junto con *Protocol Buffers* está pensado de tal forma que puedas implementar el servidor en un lenguaje distinto al del cliente si así se quiere. Esto es porque para cada lenguaje que soporta es capaz de generar el código que va a usar el servidor y el del cliente. También el asociado a la serialización y deserialización de los mensajes definidos.

En resumen, el objetivo de escoger estas dos tecnologías es emplear *gRPC* para gestionar las comunicaciones entre los distintos componentes del proyecto y los *Protocol Buffers* para la serialización y deserialización de los mensajes que intercambian entre ellos.

4.1.2. Lenguaje de programación: *Golang*

Primeramente, queremos indicar que el desarrollo del proyecto se ha realizado en Ubuntu 18.04.4 LTS. El lenguaje de programación escogido para el desarrollo ha sido *Golang*, también conocido como *Go*, en su versión 1.12.

Go es fue desarrollado por *Google* y apareció en 2009. Según sus desarrolladores, *Go* es un lenguaje de programación de código abierto que facilita la creación de software simple, confiable y eficiente[2].

En una encuesta hecha por *Stack Overflow* en febrero del año 2020 a 65000 desarrolladores, vemos que se encuentra entre los cinco lenguajes más queridos entre la comunidad de desarrolladores; una subienda respecto al puesto número 10 obtenido el año anterior [11].

Hoy en día es un lenguaje activamente usado por diversas empresas. Entre ellas están, evidentemente, *Google*, la plataforma de *streaming Twitch*, la plataforma de analíticas *Fabric* o la compañía de vehículos de transporte con conductor *Uber*, que además contribuye en gran medida a añadir paquetes para *Go* [3].

Figura 4.1: Resultados de una encuesta realizada por *Stack Overflow* en febrero de 2020 en la que vemos los diez primeros puestos de los lenguajes preferidos por la comunidad de desarrolladores.

En este lenguaje los programas se organizan en *packages* (paquetes). Cada uno es un conjunto de ficheros fuente en el mismo directorio en el que se van a compilar juntos posteriormente, además de que dentro de un mismo proyecto podamos acceder entre paquetes a elementos importando el paquete que queramos emplear.

Un repositorio *Go* está formado por un módulo (aunque pueden ser varios), que a su vez es una colección de paquetes que se lanzan conjuntamente. El directorio del módulo

[2]https://golang.org/
[3]https://www.gowitek.com/golang/blog/companies-using-golang

se define en el fichero *go.mod*, y este será el prefijo para el directorio de importación para todos los paquetes internos[4].

Generalmente, los distintos paquetes del módulo se introducen dentro de una carpeta llamada *pkg* como podemos ver en la Figura 4.2, que se encuentra en la carpeta raíz del proyecto. En esta última, debe encontrarse el fichero *main.go*, el cual tendrá como nombre de paquete *main* y será desde el que se inicialice la ejecución del módulo.

Además del propio lenguaje, también desarrollaron la herramienta *go* mediante la cual podemos ejecutar comandos para manejar el código que estamos implementando. Durante el proyecto nos hemos auxiliado principalmente de estos comandos:

- ***go mod init* (nombre)**: con este comando inicializamos el proyecto. Crea un fichero *go.mod* en el cual, además de lo explicado previamente, se anotan los diferentes paquetes externos que se emplean en el código fuente y sus versiones fijas, tal y como se puede ver en la Figura 4.3.
- ***go mod tidy***: limpia las dependencias no usadas, actualizando el fichero *go.mod*.
- ***go get* (dirección)**: permite añadir dependencias a nuestro módulo e instalarlas.
- ***go build* (nombre)**: permite compilar paquetes. Si lo ejecutamos simplemente acompañándolo del nombre del ejecutable de salida, creará el susodicho de todo el módulo (vease, el ejecutable del proyecto).
- ***go run main.go***: compila y ejecuta el programa. Una alternativa a emplear el comando anterior si solo queremos ejecutar puntualmente, ya que no genera ningún fichero.

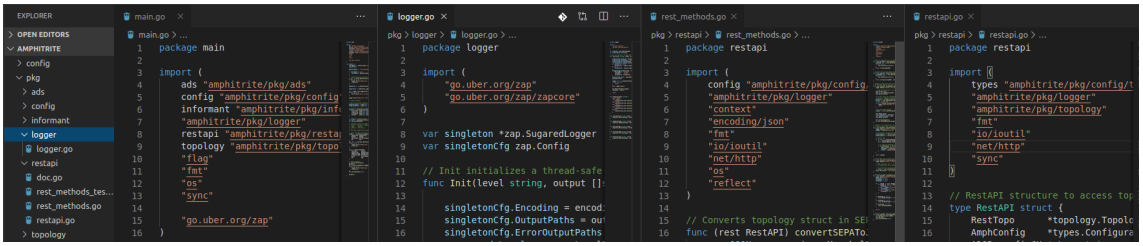

Figura 4.2: Ejemplo de la estructuración de paquetes dentro del componente *Amphitrite*. Como se ve, aquellos ficheros dentro de un mismo paquete comparten el nombre de este en la indicación dentro del código.

Auxiliarmente, para el proyecto hemos importado ciertos paquetes externos. Se pueden observar en la Figura 4.3, entre los cuales están:

- ***envoyproxy/go-control-plane***: necesario para poder implementar la comunicación entre el proxy *Envoy* para el lenguaje *Golang*. Con él, entre otros, traemos la estructura de mensajes que intercambian el servidor ADS del componente *Amphitrite* y el cliente de la instancia de *Envoy*.
- ***gogo/protobuf***: permite trabajar con *Protocol Buffers*, para serializar y deserializar mensajes. Creado a raíz del paquete *golang/protobuf*, mejorando y extendiendo las prestaciones del original.
- ***strechr/testify***: importado para la realización de pruebas. Hablaremos más sobre él en el capítulo correspondiente.

[4]https://golang.org/doc/code.html#Organization

- ***go.uber.org/zap***: mediante él implementado el sistema de registro de operaciones (*logging*).
- ***google.golang.org/grpc***: permite establecer comunicaciones *gRPC* implementadas con *Golang*.

Figura 4.3: Ejemplo de un fichero *go.mod*, en este caso el empleado en el componente *Amphitrite*. Aquellos con el comentario *indirect* se han importado a través de otros paquetes importados que hacen uso de ellos.

Por último, en otros puntos de este documento nombraremos las estrucutras de *Go*. Para tener una idea de qué son, *Golang* no es un lenguaje orientado a objetos, pero lo parece. No ofrece por tanto herencia, pero fomenta la composición y el uso de interfaces.

Es aquí donde entran las estructuras. Son colecciones tipadas que nos permiten agrupar datos. Dan la posibilidad de que un campo sea otra estructura; esta es la parte de composición. Si hay métodos asociados a esa otra estructura, podremos acceder a ellos.

Además, con el sistema de punteros a direcciones de memoria podemos compartir una misma estructura entre varios paquetes o corrutinas. No vas a adentrarnos mucho más, solo queríamos dar una idea general.

4.1.3. Relación entre *Protocol Buffers*, *gRPC* y *Golang*

Hemos comentado cada uno, pero vamos a ver brevemente cómo se relacionan entre ellos. Nosotros vamos hemos escogido como lenguaje de desarrollo *Golang*, y sin embargo los *Protocol Buffers* tienen su propio «lenguage» para definir la estructura de los mensajes que se van a intercambiar.

En nuestro caso, *gRPC* va a ser la vía por la que se intercambian los mensajes. Por ende, también podremos definir junto a los mensajes los métodos que vamos a implementar para su intercambio. Se especifica el nombre, qué va a enviar el cliente en la petición y qué le va a responder el servidor.

Ahora bien, y volviendo al lenguaje, esos métodos hay que implementarlos. Para eso los *Protocol Buffers* tienen un compilador llamado *protoc*. Con un comando podemos generar el código para el lenguaje que queramos. Cuando compilamos, se adaptan las estructuras de los mensajes y la forma de serializarlas al lenguaje escogido.

Este nuevo código se guarda en un fichero y, en *Go*, es un paquete más del que podemos hacer uso. En el mismo fichero se genera también el código para implementar tanto el servidor como el cliente *gRPC* (luego en desarrollo cada uno usará su parte, claro).

4.1.4. Entorno de trabajo

Como editor de texto hemos empleado *Visual Studio Code*, al cual hemos añadido dos extensiones, para facilitar en el desarrollo del código:

- La extensión *Go v0.14.1* desarrollada por *Microsoft*. Da soporte al desarrollo con este lenguaje, desde ofrecer sugerencias en la escritura de código y hacer tareas de *bad smells* hasta ayuda para el desarrollo de pruebas.
- La extensión *vscode-proto3 v0.4.2* desarrollada por *zxh404*, que da soporte al trabajo con ficheros tipo *.proto*, que son los empleados para estructurar los mensajes y servicios implementados mediante *Protocol Buffers*.

4.1.5. *Git* y *GitLab*

En lo referente al control de versiones durante el desarrollo, se ha decidido emplear *Git* y *GitLab*.

Git es un sistema de control de versiones gratuito y de código abierto. *GitLab* es un servicio web de código abierto también, basado en *Git* que permite la gestión de repositorios, alojar wikis (para propositos documentales) y sistemas de seguimiento de errores e incidencias.

4.2 Diseño y desarrollo de los componentes del proyecto

Tras hablar de los aspectos tecnológicos que han afectado al desarrollo, podemos adentrarnos ahora en explicar cada componente que conforma este proyecto detalladamente.

4.2.1. Componente principal: *Amphitrite*

Comenzando por el componente principal, se levanta junto a una instancia de *Envoy Proxy*, cosa que ya hemos referenciado en la Figura 3.1.

En la Figura 4.4 observamos que alza un servidor *gRPC*. Será el que se comunica con el *proxy* balanceador. Es el que conocemos como ADS.

Por otro lado, dispone de un cliente también *gRPC*. Es quien pide los recursos descubiertos al *Informant*, es decir, el que pide que le digan los cambios en el entorno.

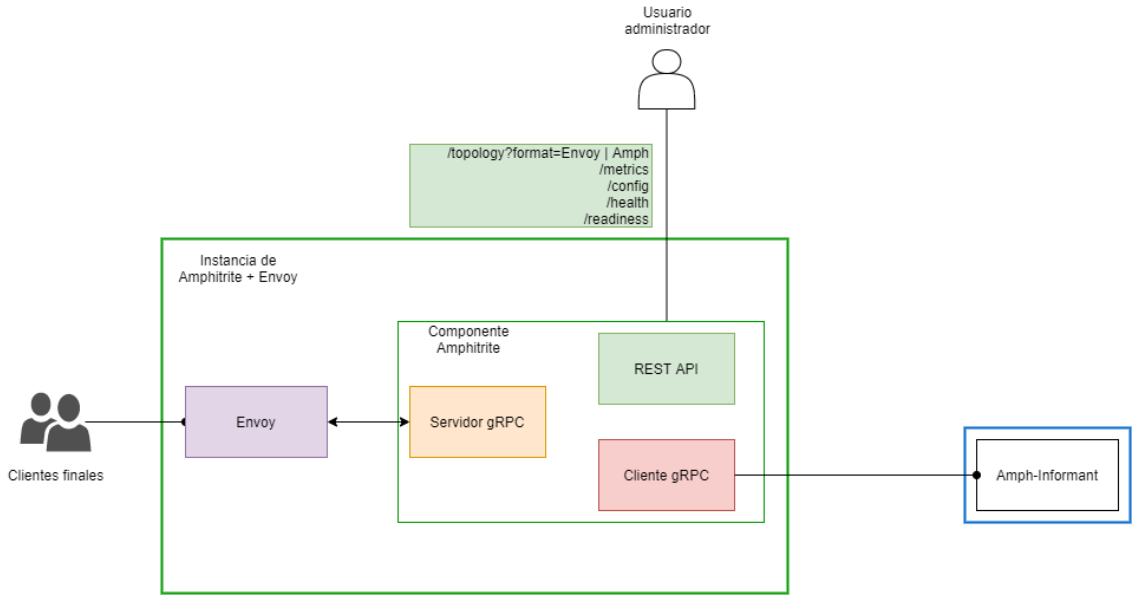

Figura 4.4: Elemento que expone un servidor *gRPC* y otro tipo *REST API*, además de un cliente *gRPC*. Se alza siempre junto a una instancia de *Envoy Proxy*.

El servidor ADS y *Envoy* establecen conexón. Cuando el *Amphitrite* recibe datos del *Amph-Informant*, el servidor ADS se encarga de serializarlos y notificarle a *Envoy* los cambios del entorno, para que ya él los trate como convenga.

Como podremos tener mínimo un *Amph-Informant* por plataforma que soportemos, instanciaremos mínimo un *Amphitrite* por cada tipo que soportemos. En nuestro caso vamos a soportar entornos *Kubernetes* y entornos *Kumori*.

Por ende, si queremos controlar el tráfico entrante a servicios en ambos entornos, instanciaremos mínimo dos instancias de *Amphitrite*: una que se comunique con informadores para *Kubernetes* y otra para los de *Kumori*. Podemos ver un ejemplo de cómo sería esta arquitectura en la Figura 4.5

De esta forma conseguimos poder controlar todos los entornos que manejamos. Si tuviéramos un único tipo de *informant*, podría darse el caso en el que todas nuestras réplicas de *Amphitrite* se conectaran a informadores de un único tipo, por ejemplo *Kubernetes*.

En ese caso, no tendríamos información sobre los servicios alojados en entornos *Kumori*. Como resultado los clientes finales no podrían acceder a ellos.

Con la arquitectura que proponemos tendremos conocimiento de todos, y a su vez, mejoramos el escalado de instancias de ambos componentes.

Quizás ocurre que se demanden más los servicios alojados en *Kumori*; pues en ese caso replicaríamos las instancias asociadas solamente.

Si lo tuvieramos «todo junto», es decir, un cliente *gRPC* en *Amphitrite* por cada plataforma soportada, llegaría a ser contraproducente a la hora de escalar. Ahora queremos dar soporte a dos entornos, pero si en un futuro se añadiesen más la cosa se complicaría. Podría incluso haber cuellos de botella en la red.

Por último hay un sservidor *REST API*. Se encarga de exponer unos *endpoints* pensados para que los usuarios administradores puedan hacer consultas sobre cuál es la configuración del componente o modificarla. Estos son:

- **GET /topology?format=Envoy | Amph**: retorna la configuración de servicios actual, que se ha recibido del *Amph-Informant*. Esta es la de *listeners, clusters, endpoints*

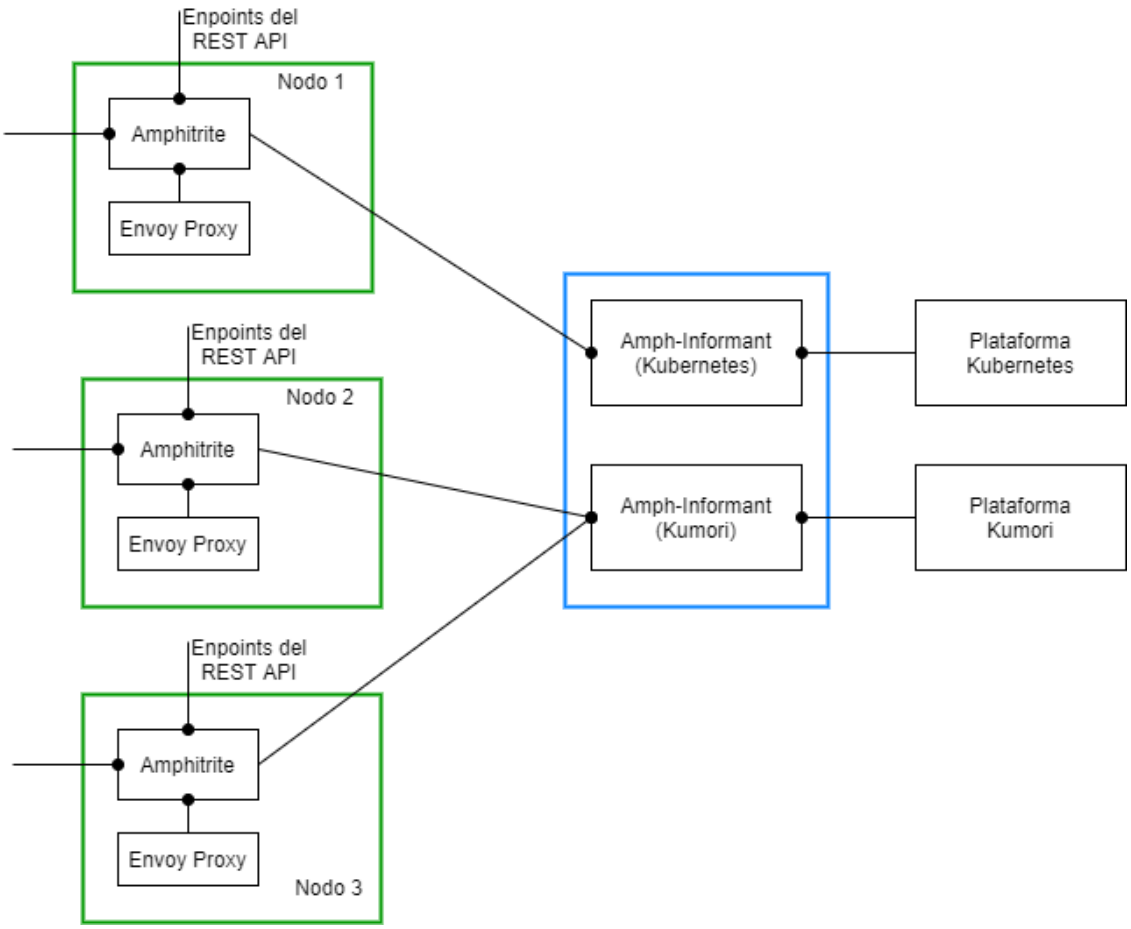

Figura 4.5: En este ejemplo, hemos levantado una instancia del informador por plataforma y vemos mínimo un componente *Amphitrite* conectado a cada una.

y rutas. El parámetro *format* es para especificar si queremos obtener estos datos tal y como los comunica el informador, o en el formato de *Envoy*.

- **GET /metrics**: retorna las métricas que proporciona envoy en formato *prometheus*.
- **GET /config**: devuelve la configuración actual, en formato *JSON*.
- **POST /config**: modifica la configuración actual del componente, pasando la nueva en formato *JSON*. Se debe publicar la configuración completa, aunque solo se haya modificado una sección.
- **GET /health**: permite averiguar si el servidor para *Envoy* está vivo, retornando en tal caso «200 OK».
- **GET /readiness**: permite averiguar si el servidor para *Envoy* está preparado, es decir, que ya ha establecido todas las , retornando en tal caso «200 OK».

Tanto la configuración del componente como la información recabada del *Amph-Informant* se gestionan siguiendo el patrón arquitectónico *singletone*, que consiste en crear una única instancia de un objeto (en caso de *Go*, estructura) garantizando que no habrá duplicados. En una primera llamada para obtener la instancia única, se crea; en todas las llamadas posteriores se retorna esta misma instancia [5].

La intención es que todos aquellos paquetes internos que quieran leer o modificar algún campo de esa estructura dispongan de la misma información. Como las llamadas

a leer cierto campo pueden ser en hilos de ejecución distintos entre ellos, tenemos que garantizar la exclusión mutua durante estas operaciones. Para ello nos hemos ayudado del paquete *sync.Mutex*, disponible en la libreria de *Go*.

Además de estos dos elementos, el sistema de registro de operaciones (*loggin*) también se ha implementado siguiendo el patrón descrito.

Finalmente, queremos indicar que el fichero de configuración del componente se especifica en el apéndice B.1.

4.2.2. Control de versiones entre *Envoy* y *Amphitrite*

Para saber sí *Envoy* está actualizado, cuando pide algún tipo de recurso a *Amphitrite* uno de los campos de la petición es la versión. Inicialmente, para cada tipo de recurso que vaya a consultar, ese campo llega con el valor cero. Podemos ver un ejemplo de este proceso en la Figura 4.6 [2].

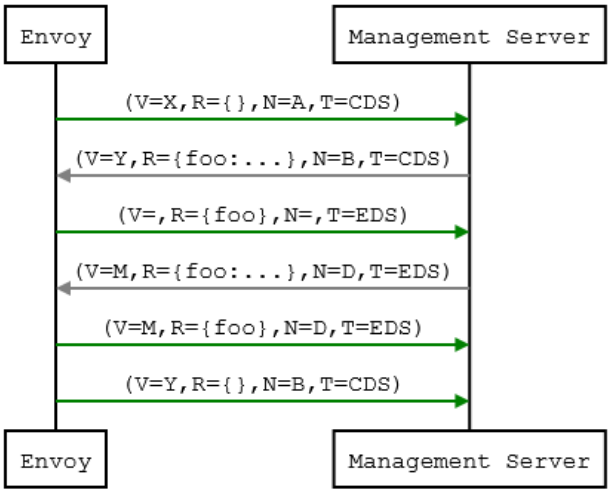

Figura 4.6: En esta figura se observa cómo *Envoy* actualiza la versión de lo que conoce sobre algún recurso (en la figura se repesentan respecto a descubrir *endpoints* y *clusters*).
Cuando reciben la respuesta correspondiente, ponen la versión de sus peticiones igual a la de la última respuesta con cambios.

Es tras la respuesta de nuestro componente que tomará el valor que le retorne el servidor ADS, para futuras peticiones. Si en otro momento ambas versiones (la del componente y la de *Envoy* son la misma, se pausa la petición hasta que haya nuevos cambios.

4.2.3. Componente *Amph-Informant*

Como ya se ha dicho, es el encargado de recabar información sobre los recursos que nos interesan: en qué dirección están, cuál es la relación entre ellos (por ejemplo, a qué ruta está relacionada un *endpoint*) y demás.

Cabe mencionar que, al igual que se ha indicado para el componente previo, en este también empleamos el patrón *singleton* para la misma funcionalidad.

Podemos ver la arquitectura del *Amph-Informant* en la Figura 4.7. Para comunicarse con *Amphitrite* cuenta con un servidor *gRPC* mediante el cual atender sus peticiones. Los datos de los recursos se los comunicará la plataforma en cuestión con la que esté trabajando.

Esta plataforma la indicaremos en el fichero de configuración del componente, el cual podemos ver en el apéndice B.2. El componente dispone de un API adaptador para cada plataforma. Con él estructura los datos recibidos y después los «traduce» a la estructura compartida con *Amphitrite*.

Para comunicarse con la plataforma asignada dispone de un REST API, que será también accesible al usuario administrador. Este último actor querrá acceder al *endpoint* de **/config**, con el que puede consultar o cambiar la configuración del componente. Los campos que puede modificar se especifican también en el apéndice B.2.

Uno de ellos es la plataforma con la que trabajar. De esta forma se podría dinámicamente pasar por ejemplo del entorno *Kubernetes* al entorno *Kumori* en las instancias para las que hicieramos la modificación.

Como ya se ha mencionado previamente, hemos querido implementar de esta forma la arquitectura para dejarla preparada por si en un futuro se quisiera trabajar con otros entornos.

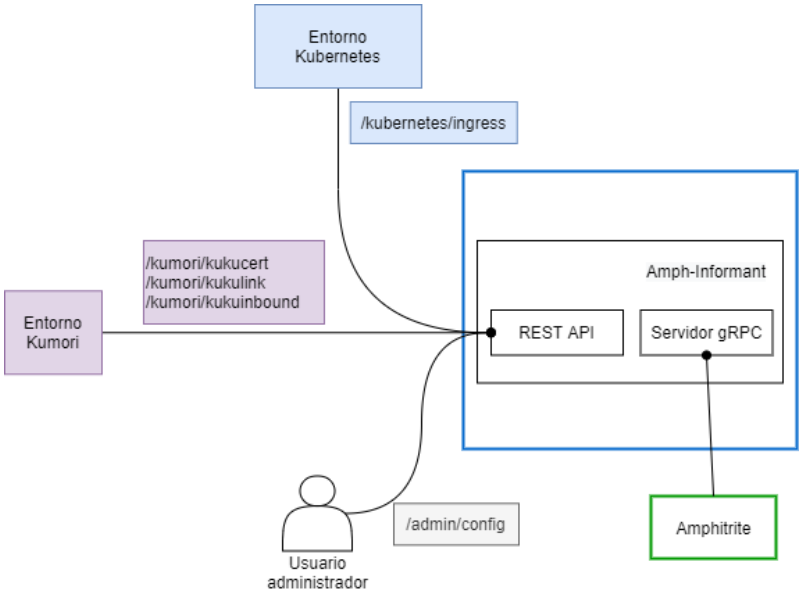

Figura 4.7: Arquitectura del componente *Amph-Informant*, que consta de un servidor *gRPC* y otro de tipo *REST API*. El primero se comunica con *Amphitrite* y el segundo con la plataforma del orquestador escogido.

La plataforma orquestadora podrá realizar peticiones de tipo *POST* a nuestro componente. Enviará la configuración del recurso o recursos que, en su propio entorno, entiende como necesarios para exponer un servicio al exterior. Lo necesario para, resumidamente, relacionar un dominio y una ruta a un servicio.

Como se ha indicado, para cada orquestador ese o esos recursos son distintos. En nuestro caso tratamos con *Kubernetes* y con el entorno cloud de la empresa *Kumori*. A continuación detallaremos qué recursos recibiremos de cada uno.

Vamos a empezar por entornos cloud de *Kumori*. La plataforma enviará tres posibles recursos, todos en formato JSON. Estos son:

- ***KukuCert***: contiene los datos de un certificado expedido para un dominio. Este objeto es referenciado desde los objetos *KukuInbound*.
- ***KukuInbound***: define un punto de entrada al servicio. Debe estar referenciado por un objeto *KukuLink*. En caso contrario, para *Envoy* el servicio no estará disponible.

Para el caso *Kumori* se decidió mostrarle al cliente final una página customizada que muestre el mensaje de servicio no encontrado (el famoso *«error 404»*).

- ***KukuLink***: enlaza un punto de entrada con un servicio desplegado en la plataforma. Se proporciona el nombre de dominio interno del servicio. Es decir, un nombre de dominio cuya resolución son las direcciones IP y puerto de acceso de todas las réplicas de este servicio. Un ejemplo del formato del que hablamos sería 127.0.0.1:8100, donde la parte antes de los dos puntos es la dirección IP y los cuatro números siguientes son el puerto.

Estos datos llegan mediante una petición tipo POST que realiza la plataforma a nuestro componente *Amph-Informant*. Los *endpoints* a los que envía cada tipo de objeto son (respectivamente) ***/kumori/kukucert***, ***/kumori/kukuinbound*** y ***/kumori/kukulink***. Podemos verlos en la Figura 4.7.

Respecto a *Kubernetes*, recibiremos recursos *Ingress*. Este es un recurso propio del orquestador. Expone rutas HTTP y HTTPS para que se pueda acceder a servicios alojados en un *cluster* y permite definir, en la configuración propia del recurso, reglas de tráfico [3].

La idea es que *Amphitrite* actue como lo que para *Kubernetes* es un *Ingress Controler*, un sistema que habilita al *Ingress*. Es decir, que aplica la configuración del *Ingress* para asociar un dominio a un servicio desplegado en un entorno *Kubernetes*.

Para este entorno los datos también llegan por petición POST, igual que en el caso *Kumori*. Para *Kubernetes* solo hay un *endpoint*, que es ***/kubernetes/ingress*** (como vemos en la Figura 4.7).

Además de los *endpoints* para las plataformas orquestadoras, hay otro que es el de */config*. Tiene la misma funcionalidad que hemos descrito en el componente anterior, permitir a usuarios administradores consultar o modificar la configuración del componente.

4.2.4. Proceso de descubrimiento de los *endpoints*

Hemos estado hablando del flujo que siguen los datos hasta llegar a *Envoy*. Y tal y como hemos estado hablando de los recursos y los datos que se reciben sobre ellos no hay mucho más que tratar. Excepto con los *endpoints*.

Para *Envoy*, un *endpoint* tiene que tener asociada una dirección IP, porque el balanceo de peticiones se hace a nivel de los puntos finales. Es decir, queremos distribuir la carga de las peticiones entre los propios *endpoints*.

Esta es una información que no nos llega propiamente desde el entorno, al menos desde los dos entornos con los que hemos decidido trabajar (*Kubernetes* y el de *Kumori Systems*.

Por ello, debemos implementar un medio por el que descubrir propiamente dónde estás los puntos finales. Esto es, cuáles son sus direcciones IP. El resto de los datos (el dominio asociado, la ruta, el puerto en el que atiende peticiones y demás) los podemos obtener desde el entorno.

Dicho esto, vamos a comentar cómo hemos resuelto esta situación y qué implica en el proyecto. Primero de todo, es conveniente que el componente *Amph-Informant* esté desplegado sobre el mismo entorno con el que vamos a trabajar.

Tanto en entornos *Kubernetes* como *Kumori* es necesario que se use como servidor DNS el software de *CoreDNS*. Este reemplazará, en el caso de *Kubernetes*, a su propio

DNS llamado *kube-dns*. Pese a no ser el propio, *CoreDNS* está muy popularizado y se gasta abiertamente.

Consultaremos al este DNS gracias al paquete ***net*** de *Go*. Entre la información que hemos recibido de la plataforma y este paquete podemos hacer una búsqueda para recuperar las direcciones IP asociadas al servicio con el que trabajamos.

CAPÍTULO 5
Implantación

Durante este capítulo vamos a hablar del proceso de alzado y puesta en marcha de nuestro proyecto, *Amphitrite*. Pero primero aclararemos algunos puntos generales.

5.1 Aspectos previos generales

Amphitrite es un proyecto que necesita un entorno que permita el despliegue en contenedores *Docker*. Cada componente que hemos descrito se despliega dentro de un contenedor.

En el capítulo de introducción definimos (en términos generales) qué es un contenedor. Pues una imagen es una captura del estado inicial del contenedor. Digamos que es la forma de definir cómo va a encontrarse el contenedor cuando arranque por primera vez, su estado inicial.

Las imágenes se definen en un archivo llamado *Dockerfile*, y en él escribiremos las instrucciones que se ejecutarán para levantar el contenedor.

Ejemplos de instrucciones (algunas opcionales) son indicar cuál será la imagen base, especificar el directorio de trabajo dentro del contenedor o los comandos de consola que ejecutará al iniciarse.

La más importante es escoger la imagen base. Si la imagen es la «plantilla» del contenedor, la imagen base sería «la plantilla de la plantilla». Le damos un contexto previo a la imagen con la que vamos a crear nuestro contenedor.

Principalmente va asociado a escoger el sistema operativo que se ejecutará en el contenedor y software adicional que queramos tener ya instalado de primeras. Después sigue con el resto de instrucciones que hemos escrito en el *Dockerfile*.

Hechas estas aclaraciones, vamos a continuar con cómo afectan a nuestro proyecto.

5.2 Despliegue del proyecto

El contendor del componente *Amphitrite* levantará este componente junto a *Envoy Proxy*. Por ello la imagen base es la *envoyproxy/envoy:v1.12.1*, que trae como sistema operativo *linux/amd64* y el software de *Envoy* instalado y listo para su uso.

Amph-Informant no tiene muchas dependencias, con lo que hemos escogido como imagen base *alpine/3.0.12* dado que es una distribución de *Linux* y pesa muy poco (2.66 MegaBytes).

Go, al igual que *C* o *Java* es un lenguaje compilado. Esto implica que antes de ser ejecutado tiene que ser traducido a código que entienda la máquina (y no puede hacerlo durante la ejecución, como sí pueden los lenguajes interpretados como *Python* o *NodeJS*). El código compilado resulta en un fichero binario ejecutable.

El problema surje en cómo pasarlo al contenedor a través de las instrucciones de la imagen. Podríamos poner alguna intrucción en la cual compiláramos el código y generáramos el binario. Por ende, implica que tendríamos que tener una imagen base que tenga *Go* instalado.

Realmente, estaríamos trayendo una serie de software que, después de hacer este proceso concreto, ya no usaríamos. Alternativamente podríamos pasarle a la imagen el propio binario compilado, pero no es recomendable.

Por ello, lo que hemos usado la técnica de construir imágenes *Docker* en múltiples etapas (*multi-stage build* en inglés). Con ello, en el mismo fichero *Dockerfile* de cada componente vamos poner las instrucciones para construir una imagen pequeña con la que compilaremos el código y crearemos el ejecutable.

A continuación veremos las intrucciones para el contenedor final, con el detalle de que le indicaremos que coja el ejecutable del escenario anterior y se lo copie. De esta forma, la imagen del primer paso quedará finalmente desechada tras hacer lo que queríamos de ella. Quedará la imagen del último escenario.

Esta es una explicación general del proceso, pero se puede ver más detallado en el anexo B.3. En este anexo también se pueden ver los ficheros *Dockerfile* de cada imagen y el comando de *Docker* con el que alzar cada contenedor.

Finalmente, en una imagen se puede indicar el comando que queramos ejecutar cuando se arranque el contenedor. En nuestro caso hemos creado un pequeño *script* (un trozo de código pequeño, con una función concreta).

El *script* del componente *Amphitrite* se encarga de dejar ejecutándose tanto *Envoy* como nuestro binario. Para el *Amph-Informant* se encarga de ejecutar solo nuestro binario del componente, porque no necesita nada más.

CAPÍTULO 6
Pruebas

A medida que ibamos desarrollando partes del proyecto, fuimos probándolas. La intención era seguir un proceso ágil de desarrollo, en el cual no nos esperáramos a acabar el proyecto para hacer pruebas.

Consideramos mejor probar funcionalidades al momento de implementarlas. Así tenemos fresco qué es lo que queremos hacer para comprobarlo y corregirlo, además de que no arrastramos errores a otros elementos.

Hicimos diferentes tipos de pruebas que vamos a comentar con más detalles en las siguientes secciones. Para la lógica de procesado de datos implementamos pruebas unitarias, es decir, probamos cada función por separado.

Algunos elementos los probamos manualmente, pero antes sabiendo que los métodos de lógica que puedan llamar tengan pruebas unitarias correctas. Con manualmente nos referimos a, por ejemplo, realizar peticiones a la REST API expuesta para administradores.

Lo consideramos así porque realmente lo que queríamos comprobar es que si un usuario hacía una petición a un *endpoint* determinado, se le respondiera. Como hemos dicho antes, la lógica nos habíamos asegurado de probarla antes.

Finalmente, a medida que acabábamos de implementar un componente realizábamos pruebas de implementación con datos simulados. En estas pruebas. verificamos que cada componente cumplía con el flujo determinado que le tocaba.

A continuación, vamos a explicar con más detalle cada grupo de pruebas.

6.1 Pruebas unitarias

6.1.1. Aspectos previos generales

Para este tipo de pruebas, el lenguaje *Go* ofrece indicaciones sobre cómo deben implementarse y herramientas para ello.

Primeramente, cuando lanzamos pruebas de programas *Go* se están probando los *packages* (paquetes) completos (de estos hablamos en la sección en la que introdujimos el lenguaje).

En un mismo paquete puede haber varios ficheros de código. Por cada uno de los que queramos escribir pruebas, se crea un fichero con el mismo nombre pero con el añadido *_test.go*. Además, las funciones en las que implementemos las pruebas unitarias tienen que tener de nombre *TestAaa* (donde *Aaa* sería el nombre del método a probar).

Para implementar las pruebas unitarias, nos hemos ayudado principalmente de estos dos paquetes:

- ***testing***: con este paquete podemos indicar que una función es una prueba automatizada, para que el comando de ejecución de tests la reconozca y ejecute. También nos permite realizar pruebas de rendimiento sobre el código.
- ***github.com/stretchr/testify/assert***: ayuda al desarrollador a implementar comprobaciones como que dos resultados sean iguales, diferentes, que el resultado no esté vacío, entre otros varios. Se escogió porque es un complemento ammigable al paquete *testing* del lenguaje.

Para ejecutar las pruebas implementadas para un paquete, *Go* nos ofrece el comando ***go test***. A este le podemos pasar algunos parámetros que nos aportan información sobre los resultados y la ejecución de las pruebas. En nuestro caso, nos hemos auxiliado de:

- ***-v***: muestra por el terminal qué pruebas está ejecunto en cada momento y por qué parte de cada uno va pasando. Además, si alguna falla, nos muestra al instante cuál y en qué punto.
- ***–cover***: cuando las pruebas automatizadas que hemos implementado acaban con éxito, muestra el porcentaje de caminos de ejecución de nuestro código que cubren.
- ***-coverprofile=çoverage.out"***: además de mostrarnos el porcentaje de cobertura, crea un fichero *.out* que podemos usar con el comando siguiente para visualizar el código que cubrimos.

Auxiliar a ejecutar *go test -coverprofile=çoverage.out"* está el comando *go tool cover -html="coverage.out"*. Lee los datos escritos en el fichero *coverage.out* por el primer comando, y muestra un fichero *.html* en el navegador donde diferenciar fácilmente el códgio cubierto del que no.

Vemos un ejemplo (externo al proyecto y simplificado) en la Figura 6.1. De esta forma, podemos consultar si hay algún camino del que no estamos haciendo pruebas y evaluar si queremos que sea así.

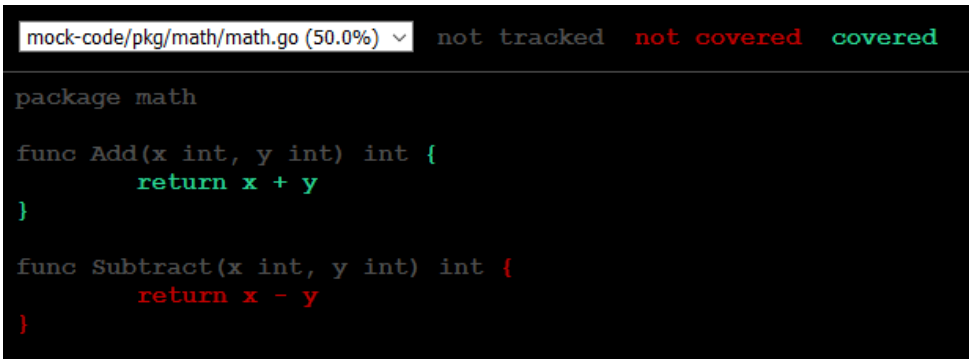

Figura 6.1: En el reporte visual se nos muestra el código del fichero del que automatizamos pruebas. Podemos observar que se nos marca en color verde los caminos que cubrimos con las pruebas, en rojo los que no y en gris los que no entran dentro de las pruebas. Estos últimos serán normalmente declaraciones (el nombre de las funciones o los paquetes que importemos, entre otros).

Para facilitar la ejecución de los tests, hemos usado la extensión ***Go Test Explorer***, para *Visual Studio Code*. Esta herramienta da una interfaz de usuario para ejecutar las pruebas y ver fácilmente si han pasado o no.

Las agrupa por paquetes, y nos permite ejecutar una prueba en concreto, todas las pruebas del paquete o todas las pruebas del módulo (es decir, del proyecto). Podemos ver un ejemplo de salida en la Figura 6.2.

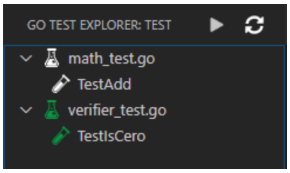

Figura 6.2: Interfaz gráfica en la que podemos ejecutar pruebas y ver fácilmente el resultado. En verde, las que se han ejecutado y superado con éxito. En rojo se nos mostrarán las que han fallado. En blanco, las que no hemos ejecutado aún.

Como mínimo, fijamos que las pruebas deberían cubrir el 70 % de los caminos de ejecución, ya que es un porcentaje que aporta un buen grado de fiabilidad.

6.1.2. Resultados de las pruebas unitarias: componente *Amphitrite*

Empezaremos con las pruebas del componentes *Amphitrite*. Como paso previo, simulamos el fichero de configuración que se le pasa al componente por argumentos para iniciarse, porque los métodos a probar hacen uso de esta configuración. Se puede ver en la Figura 6.3.

Figura 6.3: Descripción de la configuración empleada para la ejecución de las pruebas.

Dicho esto, vamos a comentar las propias pruebas y los resultados de cobertura. Vamos a hacerlo hablando por paquetes y comentando aspectos generales del código a probar.

Hay algunas que hemos automatizado (usando las dependencias que hemos comentado antes) y otras no, ya que no veíamos necesario hacerlo o bien podíamos probarlas manualmente.

Paquete *restapi*. Cobertura obtenida en las pruebas automatizadas: 82.1 %.

Es el que ofrece el REST API para que los usuarios puedan hacer diferentes consultas al componente. En este caso, hemos creado pruebas automatizadas para la lógica asociada a cada *endpoint* que expone. Una vez tuvimos la certeza de que cumplian con lo esperado, probamos los manejadores de cada *endpoint* con el programa *Postman*.

Brevemente, *Postman* nos ofrece una interfaz gráfica con la que hemos podido hacer peticiones HTTP al REST API en cuestión.

Es una herramienta fácil de manejar, y más cómoda que escribir a mano las peticiones en el terminal. En la Figura 6.4 podemos ver un ejemplo de petición de tipo POST al *endpoint /config*.

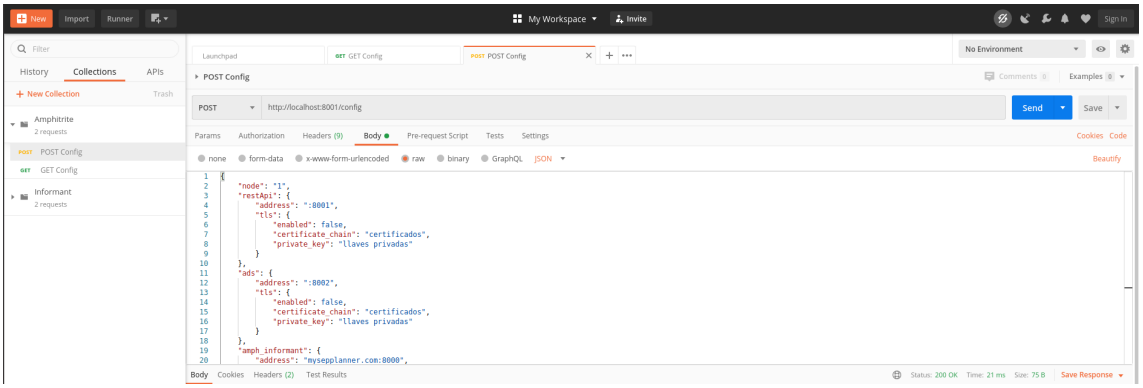

Figura 6.4: La interfaz amigable de *Postman* ayuda a crear y hacer peticioens de todo tipo a REST APIs. Además, tiene un sistema de almacenamiento de peticiones, para que las puedas guardar, repetir y compartir con otras personas. Lo hemos usado a lo largo del desarrollo y de la propia fase de pruebas.

Estas pruebas, en general, se han basado en comparar y comprobar si el resultado de un método es el que se esperaba.

Paquete *topology*. Cobertura obtenida en las pruebas automatizadas: 100 %.

Es el API que contiene las estructuras que comparten este componente y el *Amph-Informant*. Los métodos que tiene son *getters* («obtenedores»), *setters* («colocadores») y *updaters* («actualizadores»).

Para asegurarnos de que actuaban como se esperaba, los probamos en ese orden. Así pudimos usar por ejemplo un método *get* para saber si un *set* había modificado el valor en cuestión.

Paquete *ads*. Cobertura obtenida en las pruebas automatizadas: 92.76 %.

Este es el paquete que maneja todo lo relacionado con el servidor ADS que se comunica con *Envoy*. Desde establecer la conexión hasta el trato de los datos que intercambian.

Se han automatizado las pruebas de lógica interna. Esto son los métodos que se encargan de tratar los datos recogidos para transformalos a estructuras que *Envoy* es capaz de entender, y serializarlas para enviarlas.

Una vez confirmamos que las pruebas nos daban los resultados esperados, pasamos a probar lo no automatizado. Esos métodos son los encargados de atender a las peticiones que haga el cliente de *Envoy Proxy* por *gRPC*.

Son cuatro métodos, cada uno atiende a las peticiones de cada uno de los recursos que comentamos en un capítulo anterior: *clusters*, *endpoints*, *listeners* y *routes*.

Para probar cada método por separado creamos un pequeño cliente *gRPC*, simulando al cliente de *Envoy* . Recordemos que con este sistema podemos hacer llamadas a una función en el servidor a modo de peticiones.

Pues lo que hemos simulado son las peticiones a cada uno de los métodos que envían información sobre cada recurso. Alimentamos al servidor con una serie de datos simulados sobre cada recurso. En el cliente, comprobamos que los datos recibidos eran los que esperábamos.

Realmente no hay varios métodos en nuestro caso. Un servidor ADS para *Envoy* implementa un único método por el cual le hacen peticiones. Esto es por la propia idea del ADS: ofrecer un único canal de comunicación para impedir que las peticiones queden bloqueadas por tener variso canales simultaneamente abiertos.

En un campo de la petición se indica el tipo de recurso que se queire consultar, con lo que hemos estado jugando con este campo para hacer las pruebas.

Paquete *informant*.

Implementa el cliente que hace consultas al *Amp-Informant* para obtener los datos de los recursos. Para probarlo, hemos simulado el servidor del *Amp-Informant*, razón por la que implementamos y probamos primero el servidor. Para tener la fiabilidad de que, si algo fallaba, era el cliente.

6.1.3. Resultados de las pruebas unitarias: componente *Amph-Informant*

Este componente tiene una carga de lógica similar al anterior; aquí podríamos decir que es un pco más sencilla. Cada API para cada entorno dispone de una serie de métodos *getters* y *adders* («añadidores»).

Las pruebas sobre estos métodos fueron prácticamente iguales (en cuanto a formato) que las que hemos comentado antes, en *Amphitrite*.

La cobertura en estas pruebas automatizadas es del 83.3 %.

Para probar los métodos de intercambio de mensajes con el componente *Amphitrite*, hemos implementado un cliente simulado.

Probamos primero este servidor porque consideramos más relevante asegurarnos de que el trato de datos se hacía corectamente. De haberlo hecho al revés, hubiera sido más complicado hacer las pruebas en el cliente.

Por último, respecto al proceso de descubrir los endpoints lo que hicimos fue de nuevo hacer pruebas manuales, con entornos simulados. Desplegamos un servicio sencillo, creamos algunas réplicas y comprobamos si lo que devolvía el método era coherente.

6.1.4. Comentario final sobre las pruebas

Además de darnos fiabilidad sobre el código escrito, las pruebas nos han ayudado a detectar y corregir fallos. Buena parte de ellos tenían que ver con cómo estábamos haciendo uso de variables a las que se accede por punteros.

Hacer las pruebas nos hizo ver que estábamos teniendo falsos positivos al ejecutar el componente. En algunos elementos no estábamos usando correctamente los punteros, y acababábamos apuntando a direcciones distintas por la copia de variables.

Ocurrió elementos como el API compartida entre nuestros dos componentes y el API compartida con *Envoy*. Los primeros resultados de las pruebas fueron los que nos hi-

cieron decidirnos por implementar el patrón *singleton* (del que ya hemos hablado) para algunas partes del proyecto.

También detectamos un error en el servidor ADS, el cual hacía que si el cliente enviaba una petición con cierre de la conexión, no estábamos manejando correctamente la situación y saltaba un error. Al detectarlo, corregimos la función de cierre seguro, para evitar estas situaciones.

6.2 Mediciones con *ghz*

Además de pruebas, hemos hecho también mediciones de tiempos de respuesta a los servidores *gRPC* de cada componente. El objetivo es comprobar con datos que no son lentos en responder, ya que sería perjudicial a la larga.

Por ejemplo, no podríamos tener esperando a un cliente un minuto para que le lleguen los recursos. Y con varios clientes sería sinónimo de colapso. Por eso hemos decidido analizar los tiempos de respuesta frente a determinadas cargas de trabajo.

Hemos empleado un software llamado *ghz*. Es un programa que nos permite medir los tiempos de respuesta de servidores *gRPC* y su capacidad de carga.

Se puede ejecutar tanto por línea de comando como por código, creando un programa simple a partir del paquete *github.com/bojand/ghz* para *Go*.

Nos vamos a centrar en la opción por comando. A este le tenemos que pasar obligatoriamente el nombre del método que queremos llamar, los datos de la petición que vamos a enviar y la dirección al servidor *gRPC*.

Además de estos argumetos, podemos indicar otros como la cantidad de peticiones que queremos ejecutar o la cantidad de conexiones que queremos establecer. También se puede especificar cuántas peticiones se ejecutarán concurrentemente.

Al ejecutarlo, por defecto muestra por consola los resultados formateados. Esto hace que la lectura sea fácil y rádpida. En la Figura 6.5 podemos ver un ejemplo de salida.

Tras la ejecución, se nos muestra por consola un resumen de la cantidad de peticiones que se han realizado y distintos registros de tiempo para poderlos comparar.

Además, muestra un histograma en el cual podemos ver asociados la cantidad de peticiones con tiempos, lo que nos permite en un vistazo fácil ver cuánto ha tardado el mayor grupo de peticiones.

También devuelve una tabla de distribución de latencia y, finalmente, la cantidad de peticiones que se han resuelto con éxito.

La salida por consola no es ta única. Tenemos la posibilidad de ejecutar un servidor web que ofrece una interfaz de usuario para consultar los resultados obtenidos con mayor detalle.

Esto es *ghz-web*. Para ejecutarlo, hace falta indicarle por configuración la dirección de una base de datos de *sqlite3*, *mysql* o *postgres*. En nuestro caso, hemos escogido un servidor local de *sqlite3*.

La idea es ejecutar el comando de *ghz* «normal» pero indicando que, en lugar de la salida formateada, muestre los resultados en formato *JSON*. Después, se recogen y se envían mediante una petición POST al API de *ghz-web*.

Pero primero debemos haber creado un proyecto en la aplicación web. Los proyectos son una forma de agrupar los informes recibidos. Así podemos consultar el histograma de un grupo de informes.

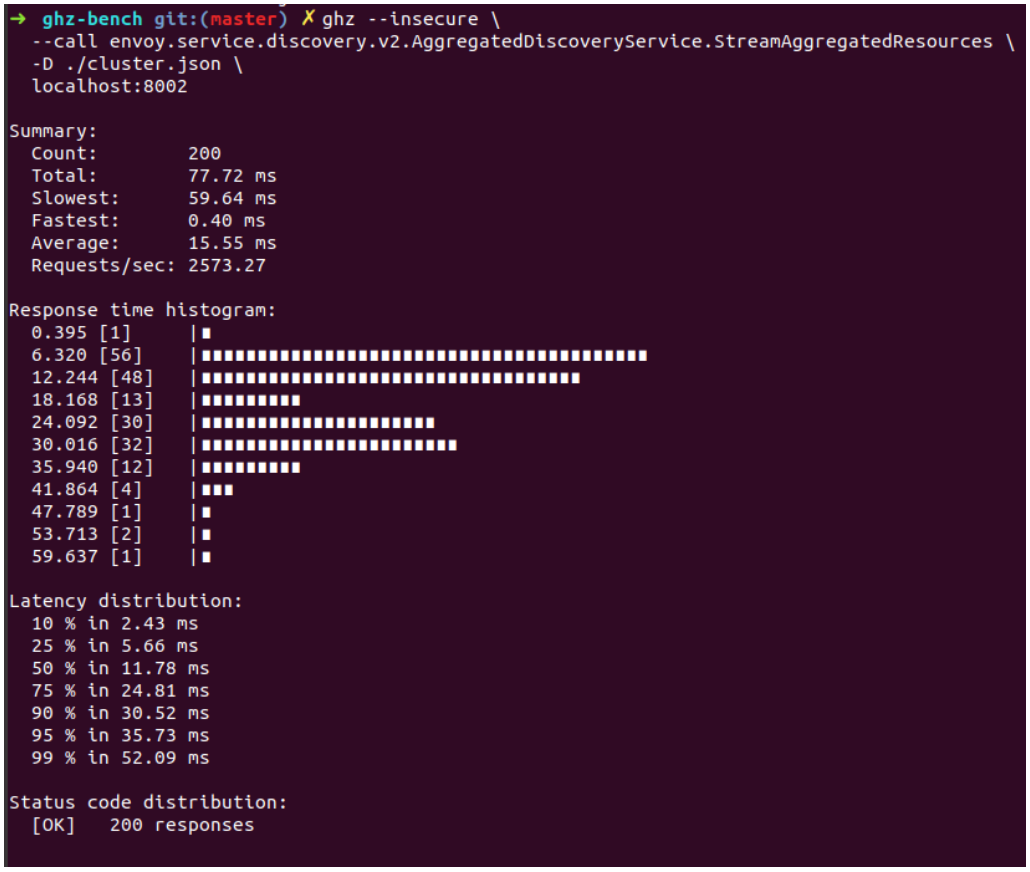

Figura 6.5: Resultados mostrados por consola. Vemos un resumen de datos, un histograma, la distribución de latencia y la cantidad de peticiones que han tenido éxito.

Podremos consultar cada informe. Dentro, podremos ver la información mostrada por la consola, pero con una interfaz gráfica más amigable y con la condiguración que hemos usado para ese concreto. También ofrece comprar entre dos resultados.

Para acabar con esta introducción, queremos comentar que las mediciones se han hecho en una máquina con un Intel Core i5-6200 @2.30GHz y con una memoria RAM de 8GB.

6.2.1. Los resultados que se ofrecen

El software de *ghz* nos ofrece, a partir de los tiempos resultantes, el tiempo medio, cuándo ha tardado la respuesta más rápida y la más lenta, la cantidad de peticiones por segundo y, finalmente en la gráfica, el valor del percentil 95 y el 99.

Vamos a comentar a qué hace referencia cada uno. El tiempo medio de respuesta será el tiempo total que se ha tardado en resolver todas las peticiones entre el número total de peticiones.

Las peticiones por segundo, como su nombre indica, es el número total de peticiones entre tiempo total que se ha tardado en resolver las peticiones.

La petición más rápida y la más lenta son, respectivamente, producto de ordenarlas y obtener la primera y la última.

Por último tenemos los percentiles, que son el valor a partir se encuentran, por debajo, el resto de los valores. Por ejemplo, cuando miremos en la gráfica la línea del percentil 95 nos mostará el valor de tiempo más alto del 95 % de las peticiones.

6.2.2. Resultados de los *benchmarks*: componente *Amphitrite*

Lo que hemos hecho por la parte del servidor ADS es alimentarlo con datos simulados, para que sea los que devuelva en las respuestas. Inicialmente se le ingresan un par de recursos de cada tipo y cada dos segundos añadimos más.

Vamos a comentar la tabla que agrupa los distintos reportes y la gráfica que nos muestra los tiempos resultantes agrupados. En esta última, podemos ver la evolución del tiempo medio, la respuesta más lenta y la más rápida. También nos muestra el percentil 95, el percentil 99 y la evolución de las peticiones por segundo (RPS).

Dicho esto, pasemos a hablar de los resultados en si. Para probarlo con carga, hemos indicado que se establecerá una conexión simultanea y que se realizarán mil peticiones en total. Hemos buscado simular en cierto modo una situación real.

En nuestro proyecto, el servidor ADS solo va a contar con una instancia de *Envoy* con la que trabajar a la vez. En cada petición se retornan 47 recursos en total.

La tabla de resultados obtenida se puede ver en la Figura 6.6, y la gráfica comparativa en la Figura 6.7.

REPORTS

COMPARE DELETE

ID	Date	Total	Average	Slowest	Fastest	RPS	Status
27	1/7/2020 17:15:44 (21 seconds ago)	1.39 s	1.12 ms	6.88 ms	0.65 ms	721.05	OK
26	1/7/2020 17:15:02 (1 minute ago)	1.36 s	1.10 ms	10.67 ms	0.63 ms	733.24	OK
25	1/7/2020 17:14:11 (1 minute ago)	1.36 s	1.10 ms	6.52 ms	0.63 ms	734.79	OK
24	1/7/2020 17:13:50 (2 minutes ago)	1.33 s	1.07 ms	6.19 ms	0.65 ms	750.43	OK
23	1/7/2020 17:13:04 (3 minutes ago)	1.32 s	1.05 ms	7.79 ms	0.61 ms	759.93	OK
22	1/7/2020 17:12:42 (3 minutes ago)	1.43 s	1.16 ms	35.04 ms	0.65 ms	698.47	OK
21	1/7/2020 17:11:52 (4 minutes ago)	1.36 s	1.09 ms	9.52 ms	0.64 ms	734.81	OK
20	1/7/2020 17:10:52 (5 minutes ago)	1.38 s	1.11 ms	7.84 ms	0.64 ms	727.19	OK
19	1/7/2020 17:10:28 (5 minutes ago)	1.36 s	1.09 ms	4.82 ms	0.64 ms	734.04	OK
18	1/7/2020 17:10:08 (5 minutes ago)	1.39 s	1.12 ms	8.98 ms	0.62 ms	720.58	OK
17	1/7/2020 17:09:46 (6 minutes ago)	1.34 s	1.08 ms	8.64 ms	0.66 ms	744.21	OK

Figura 6.6: Resultados mostrados en la aplicación web de *ghz*. Muestra el tiempo total que ha tardado en resolver todas las peticiones, el tiempo medio, el más rápido y el más lento, y finalmente las peticiones por segundo y el estado (resuelta o fallida).

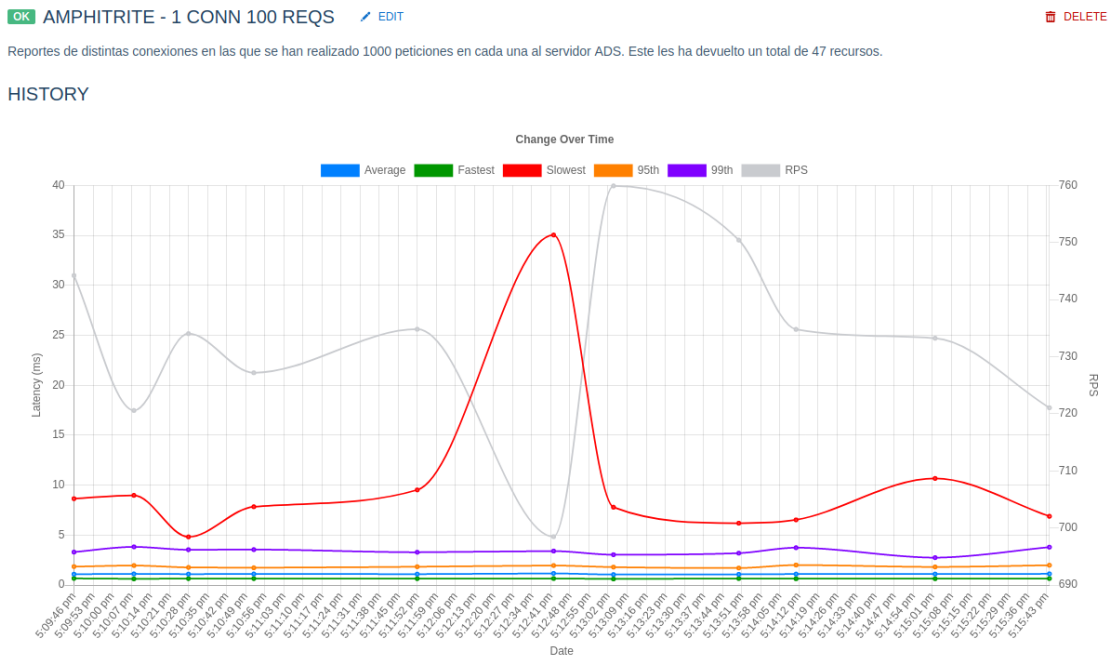

Figura 6.7: Muestra una gráfica que recoge los resultados de todos los informes del proyecto en cuestión. Pinta de azul el tiempo medio, de verde el más rápido, de rojo el más lento. Sobre los percentiles, el 95 aparece en naranja y el 99 en morado. Las peticiones por segundo aparecen en gris.

Lo primero a comentar es que todas las peticiones se han resuelto con éxito, lo cual ya es en si una muestra de salud por parte del servidor.

A simple vista podemos deducir que las peticiones han tenido un tiempo de respuesta muy similar. El tiempo total oscila sobre los 1.36 segundos. La diferencia entre unas y otras peticiones es de milésimas de segundo.

Sobresale un caso en concreto, en el que ha habido una petición que ha tardado notablemente más que el resto en resolverse. Esto ha hecho que el tiempo total de ese reporte aumente sobre el resto.

En la Figura 6.7 podemos ver este caso como el pico en la línea roja, la de las peticiones más lentas de entre los informes.

Si nos fijamos en las peticiones más rápidas, no vemos apenas ninguna diferencia ni ninguna que destaque. Esto puede indicar que solo podremos conseguir que una petición se resuelva en ese tiempo.

6.2.3. Resultados de los *benchmarks*: componente *Amph-Informant*

Al igual que con el caso anterior, a este componente le hemos alimentado de datos simulados. En total hay cincuenta datos sobre recursos. A diferencia del alterior, este componente puede recibir varias conexiones.

Por ello hemos indicado que en cada medición se establezcan veinte conexiones y se harán un total de mil peticiones al servidor *gRPC*. Los resultados obetenidos aparecen en la Figura 6.8 para la tabla y en la Figura 6.9 vemos la gráfica de resultados.

REPORTS

COMPARE DELETE

	ID	Date	Total	Average	Slowest	Fastest	RPS	Status
	111	3/7/2020 1:01:34 (1 minute ago)	5.35 s	93.41 ms	826.02 ms	0.91 ms	187.06	OK
	110	3/7/2020 1:00:41 (2 minutes ago)	5.14 s	96.54 ms	1.10 s	0.50 ms	194.41	OK
	109	3/7/2020 1:00:17 (2 minutes ago)	5.43 s	95.33 ms	1.00 s	0.47 ms	184.29	OK
	108	3/7/2020 0:59:26 (3 minutes ago)	5.43 s	101.00 ms	1.08 s	0.48 ms	184.17	OK
	107	3/7/2020 0:58:48 (4 minutes ago)	6.39 s	113.27 ms	1.01 s	0.70 ms	156.56	OK
	106	3/7/2020 0:58:17 (4 minutes ago)	5.42 s	99.20 ms	1.05 s	0.66 ms	184.48	OK
	105	3/7/2020 0:57:34 (5 minutes ago)	5.30 s	95.06 ms	824.83 ms	0.41 ms	188.80	OK
	104	3/7/2020 0:56:47 (6 minutes ago)	5.50 s	102.15 ms	1.13 s	0.71 ms	181.87	OK
	103	3/7/2020 0:55:55 (7 minutes ago)	5.10 s	92.11 ms	947.52 ms	0.40 ms	196.00	OK
	102	3/7/2020 0:55:02 (7 minutes ago)	5.15 s	93.50 ms	914.91 ms	0.70 ms	194.30	OK
	101	3/7/2020 0:54:37 (8 minutes ago)	5.50 s	99.14 ms	1.70 s	0.96 ms	181.89	OK
	100	3/7/2020 0:53:50 (9 minutes ago)	5.18 s	92.67 ms	916.76 ms	0.69 ms	193.08	OK
	99	3/7/2020 0:53:25 (9 minutes ago)	6.07 s	109.06 ms	959.01 ms	0.54 ms	164.76	OK
	98	3/7/2020 0:52:14 (10 minutes ago)	5.27 s	95.51 ms	828.30 ms	0.67 ms	189.76	OK
	97	3/7/2020 0:51:32 (11 minutes ago)	5.31 s	91.87 ms	1.08 s	0.45 ms	188.32	OK

Figura 6.8: Resultados mostrados en la aplicación web de *ghz*. Muestra el tiempo total que ha tardado en resolver todas las peticiones, el tiempo medio, el más rápido y el más lento, y finalmente las peticiones por segundo y el estado (resuelta o fallida).

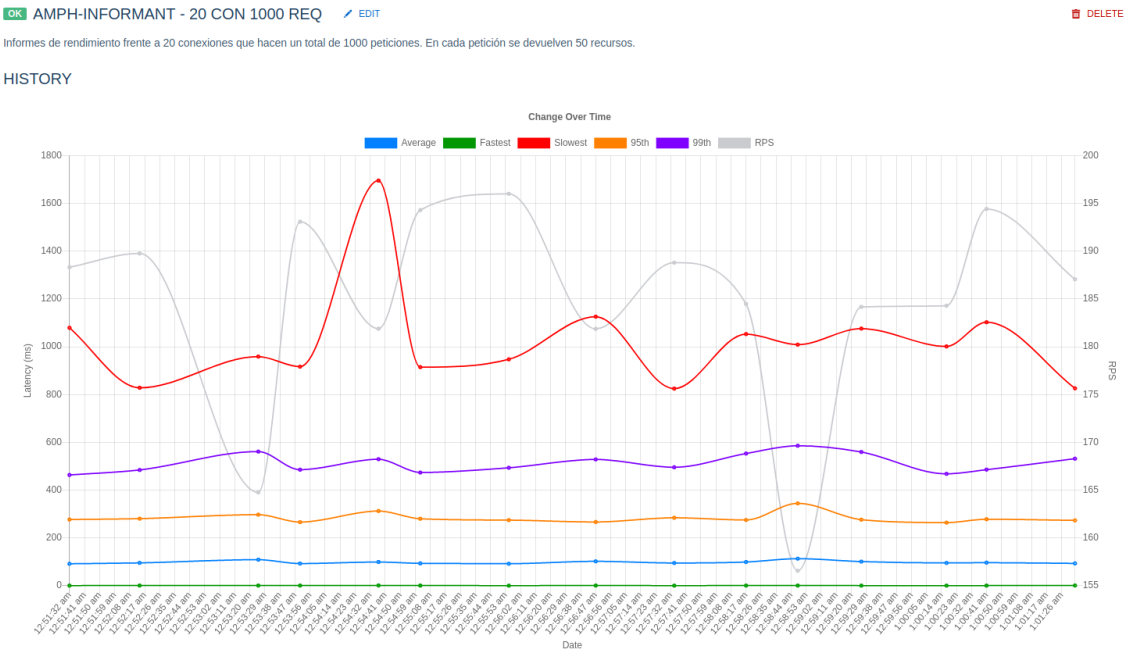

Figura 6.9: Muestra una gráfica que recoge los resultados de todos los informes del proyecto en cuestión. Pinta de azul el tiempo medio, de verde el más rápido, de rojo el más lento. Sobre los percentiles, el 95 aparece en naranja y el 99 en morado. Las peticiones por segundo aparecen en gris.

Mirando la gráfica vemos que la variación entre informes es algo más notable que con el caso anterior. Aparecen dos picos llamativos en el caso de las peticiones más lentas. Podemos considerarlas anomalías, si miramos el percentil 95 y el 99 no marcan una gran diferencia respecto al resto de casos.

La otra anomalía son las caídas de las peticiones por segundo (RPS) en dos puntos, bajando de las 170 RPS. Aquí sí vemos que podrían deberse a que en general esas peticiones fueron más lentas, porque todos sus tiempos suben ligeramente respecto al resto.

Pese a todo, el tiempo medio de respuesta se encuentra estable. Lo cual nos puede indicar que, pese a que haya alguna brusquedad, los tiempos de respuesta no se verán muy alterados.

6.2.4. Conclusiones de los resultados

El que mejor parado ha salido es el *Amphitrite*. Pero en su caso tenía que establecer una única conexión frente a las 20 del *Amph-Informant*.

En cuanto a tiempos, podemos darnos por satisfechos con los resultados. Sí que es verdad que los que hemos obtenido en el segundo caso nos dan a pensar que quizás se podría investigar alguna mejora de tiempos, pero no los consideramos preocupantes en estos instantes.

La razón es que no son tiempos altos teniendo en cuenta la carga sometida. Tener que establecer cada una de las 20 conexiones conlleva un cierto retraso.

A pesar de esto, se están resolviendo las mil peticiones totales en poco más de 5 segundos. Pero, como hemos comentado, la mejora es un camino que se puede tomar en el futuro.

Por lo general, ambos casos se muestran estables en los distintos informes hechos. Este punto nos parece de los más relevantes. Si hubieramos visto grandes variaciones en los tiempos medios o en los percentiles, significaría que algo no está correcto.

Los picos que hemos visto en la línea de las peticiones más lentas se ha verificado que no son relevantes. Hemos visto que son casos puntuales; si hubieran habido varías similares, las líneas de casos más lentos y la de los percentiles estarían más cerca.

Sí que ralentizan el tiempo total, pero pese a todo el tiempo medio de resolución de cada petición está muy por debajo del caso concreto de la más lenta.

CAPÍTULO 7
Conclusiones

La tecnología hace tiempo que avanzó, y hoy en día una gran parte del software se encuentra en la nube. La forma en la que desarrollamos también ha evolucionado, y en entornos *cloud* las arquitecturas basadas en microservicios están a la orden.

Ha medida que la forma de llevar aplicaciones a entornos compartidos evolucionaba, han ido apareciendo problemas a resolver. Y siguen apareciendo. Unos no se han resuelto aún y otros sí. Pero que tengan una solución no impide que salgan alternativas.

En este proyecto hemos hecho precisamente eso: ofrecer una solución alternativa a un problema existente. Existe el problema de cómo vamos a llevar peticiones de ingreso hasta los servicios que tenemos en la nube. Servicios que permiten ser escalados, y cuando se escalen su entorno cambiará.

Así pues, la cuestión es pensar cómo vamos a llevar esas peticiones hasta unos servicios que (de buenas a primeras) es probable que no se sepan las direcciones concretas en las que se encuentran. Pero no solo eso, si escalamos van a haber distintas réplicas. Por tanto, habrá que repartir las peticiones entre esas replicas si queremos aprovecharlas.

Sabemos que existen los *proxies* balanceadores de carga, que hacen lo que su nombre indica. Pero estos por sí solos no son capaces de saber los puntos finales a los que llevar una petición que llegue por cierta ruta de cierto dominio.

De hecho, tampoco son capaces de saber que tienen que atender a las peticiones que lleguen por esas rutas. Esta información hay que alimentársela.

Analizamos la situación y comparamos con otras soluciones ya existentes. Vimos las ventajas y los inconvenientes que presentaban, y decidimos ofrecer algo que ofreciera características ventajosas y aprovechables de cada uno. Buscar un punto medio.

Durante este documento se ha hablado de nuestra alternativa: el proyecto *Amphitrite*. Antes de desarrollarlo, apuntamos una serie de objetivos generales a cumplir.

En estos momentos, podemos decir que esos objetivos se han cumplido. Tenemos un sistema con el cual descubrir dónde están los servicios con los que tenemos que trabajar y a qué rutas y puertos de entrada están asociados.

De esta parte se encarga el *Amph-Informant*. Recogerá la información relativa a lo comentado que llegue desde la plataforma y se la hará saber al componente que trabaja con el balaceador, *Envoy Proxy*.

Somos capaces de adaptarnos a cambios en el entorno. De nuevo, el componente *Amph-Informant* maneja la situación. Cuando se le notifique algún cambio, de nuevo se lo hará saber al elemento *Amphitrite*, para que este se lo comunique a *Envoy*.

La parte del último objetivo principal también la podemos dar por cumplida. La arquitectura diseñada contempla cada componente como microservicios separados que trabajan mano a mano.

Esto nos permite poder escalar el número de réplicas de cada según nos convenga y sin afectar al otro. Si la demanda de peticiones entrantes, que son las que pasarán por el bloque *Amphitrite* y *Envoy*, están siendo muy solicitadas pues escalamos y creamos otra u otras réplicas.

Si coincide que no hay cambios en el entorno de los servicios y por tanto el *Amph-Informant* está ocioso, no se tendrá que replicar.

Dicho esto, actualmente el podemos considerar el proyecto listo para entornos de desarrollo y preproducción. Faltarían ciertos cambios y añadidos, como que la conexión de usuarios administradores al REST API de cada componente sea cifrada.

Además, la nueva plataforma *cloud* de *Kumori* está en la misma situación. Aún no se encuentra lista para producción, con lo que hemos avanzado al paso que ella y por tanto habrá cambios que hacer con respecto a la interacción entre el proyecto y la plataforma.

Durante el desarrollo nos hemos encontrado con ciertas dificultades. La primera y más evidente ha sido trabajar con una tecnología poco conocida para nosotros. Costó un tiempo hacerse a ella, todo a base de consultas en la documentación de cada una.

Hablando de la documentación, tuvimos al inicio del proyecto un error. En uno de los primeros capítulos de este documento comentamos la diferencia entre un servidor xDS y uno ADS (con respecto a la comunicación con *Envoy*). Inicialmente, implementamos el de primer tipo. La vasta mayoría de la documentación hacía referencia a él, y ellos nos hizo confiarnos.

No fue hasta que lo probamos que nos dimos cuenta de que cuando recibía varias peticiones se quedaba bloqueado. En el tipo xDS cada petición va por un «canal» distinto, lo que provocaba luego el bloqueo. Tras varias consultas, comprobamos que existía un tipo mejorado.

El ADS utiliza un único «canal» para comunicar todas las peticiones, variando los valores dentro del mensaje de petición para diferenciarlas. Esto conllevó refactorizar parte del elemento; por suerte, lo vimos al inicio del proyecto.

7.1 Relación con los estudios cursados

En términos generales, hacer este proyecto ha permitido exteriorizar y ver una aplicación «real» a todo aquello que hemos estudiado en la carrera.

La frase es muy ambigua y evidentemente en la carrera hemos hecho aplicaciones «reales». Lo que queremos indicar es que esta nos ha servido como una experiencia completa que enlaza distintos aspectos que se han visto durante la carrera.

De base hemos tenido la oportunidad de pensar y desarrollar una arquitectura desde cero, para dar una propuesta a un problema concreto y existente en el mundo externo al estudiantil.

También hemos podido aplicar técnicas de *scrum* y desarrollo ágil vistas en la carrera para el desarrollo del proyecto. Como consecuencia, nos hemos visto en una situación en la que la organización es uno de los puntos fuertes del proceso de desarrollo.

Otro aspecto ha sido el referente a la documentación, tanto para hacerla como para consultarla. En primero de carrera, estudiando el lenguaje *Java*se fomentaba mucho el familiarizarte a buscar en la documentación dudas sobre algún elemento del lenguaje.

En este proyecto hemos tratado con tecnologías que no conocíamos y, al igual que en el primer año de la carrera, hemos pasado mucho tiempo en sus páginas de documentación.

Del mismo modo, el lenguaje escogido (*Go*) es uno que fomenta (y casi obliga) al desarrollador a añadir documentación sobre las funciones y variables públicas (accesibles desde otros paquetes), al igual que a los propios paquetes.

Sobre todo, durante la rama cursada, la de ingeniería del software, se mostró la documentación como una ayuda al mantenimiento del software. Y este punto es uno que hemos visto fomentado con el desarrollo del proyecto, incluso de cara al propio desarrollo.

En alguna ocasión hemos querido llamar a una función implementada hace tiempo; en lugar de revisar el código, una lectura corta ha sido suficiente.

Relacionado también con la mantenibilidad está otro aspecto enseñado principalmente durante la rama: escribir código limpio.

Al fin y al cabo, tanto para el desarrollador como para cualquier otro que pueda recoger el testigo es mucho más cómodo seguir unas reglas comunes que no ir cada uno por su lado.

CAPÍTULO 8
Posible extensión del trabajo

Tras explicar nuestro proyecto, queremos comentar los posibles caminos o proyectos que puedan surgir a raíz de *Amphitrite*.

Actualmente estamos hemos propuesto un proyecto que da soporte a dos plataformas orquestadores: *Kubernetes* y el entorno cloud de *Kumori*, la de la empresa con la que se han realizado las prácticas y el proyecto.

Se podrían añadir otros entornos a los que dar soporte. Por ejemplo, se podría dar soporte a otros orquestadores populares como *Docker Swarm*, servicios desplegados con *Azure Container Services* o *Mesosphere*.

Dado que nuestra arquitectura está preprada, hacer esto solo conlleva crear crear un nuevo tipo de *Amph-Informant* para la plataforma en cuestión.

Además, habría que implementar el proceso de «traducción» del API del orquestador al API común entre nuestros dos componentes (*Amphitrite* y *Amph-Informant*).

Otro camino sería el de dar soporte a redirecciones de peticiones que sean via *gRPC*. Actualmente no lo contemplamos, pero podría ser un añadido interesante.

Un proyecto que podría nacer de este sería implementar un frontal (una página web) con vista de administrador. Desde ella, un usuario administrador podría hacer las consultas que ahora mismo hace mediante peticiones GET al REST API de cada componente.

Es decir, el objetivo sería tener una interfaz visual con la que pudiera interactuar más fácil y cómodamente. Desde ella, como hemos comentado, se podría hacer consultas, pero también modificar la configuración de cada componente.

En resumen, sería una forma visual de ofrecer la funcionalidad de las REST API expuestas para usuarios administradores.

Por otro lado, y también relacionado con dar una interfaz gráfica para consultas, se podría enlazar el sistema de registros (*logs*) con alguna herramienta externa.

Por ejemplo, se podría enlazar con *Logz.io* y así delegar el sistema de *loggin*. Así se podría consultar desde la propia aplicación de *Logz.io* de forma visualmente fácil y cómoda.

CAPÍTULO 9
Agradecimientos

A José Manuel Bernabeu Aubán, tutor de este trabajo de fin de grado, la ayuda prestada a lo largo de estos meses para la redacción de este documento.

A Juan José Valero Barjola por resolver dudas técnicas y la ayuda ofrecida en general para que este proyecto fuera posible.

Al equipo de Kumori Systems, por darme la oportunidad de realizar este proyecto junto a ellos, por todo lo que me han enseñado durante el periodo de prácticas y por el valor humano que me han aportado durante las prácticas y tras ellas.

A mis padres y amigos, por hacer de revisores externos de este documento y por escucharme cuando lo necesitaba.

Por último, me gustaría agradecer a todos los docentes con los que he tenido la suerte de cruzarme. En especial, a los de la etapa de Educación Secundaria Obligatoria. Ellos y ellas nos enseñaron a apreciar la ciencia y las humanidades. Fueron buena parte de la motivación para escoger estos estudios.

Bibliografía

[1] Envoy Project Authors. xds rest and grpc protocol. https://www.envoyproxy.io/docs/envoy/v1.12.1/api-docs/xds_protocol, 2019. Consultado durante Noviembre de 2019.

[2] Envoy Project Authors. xds rest and grpc protocol. https://www.envoyproxy.io/docs/envoy/v1.12.1/api-docs/xds_protocol#aggregated-discovery-service, 2019. Consultado durante Noviembre de 2019.

[3] The Kubernetes Authors. Ingress - Kubernetes. https://kubernetes.io/docs/concepts/services-networking/ingress/, 2020. Consultado el 03-04-2020.

[4] The Kubernetes Authors. What is kubernetes? - Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2020. Consultado el 03-04-2020.

[5] Mario Castro Contreras. *Go Design Patterns*. Packt Publishing Ltd, 2017.

[6] Google Developers. Protocol buffers - why not xml. https://developers.google.com/protocol-buffers/docs/overview#whynotxml, 2019. Consultado durante Noviembre de 2019.

[7] Google Developers. Protocol buffers. https://developers.google.com/protocol-buffers, 2020. Consultado durante Noviembre de 2019.

[8] gRPC Authors. grpc - guides. https://grpc.io/docs/guides/#working-with-protocol-buffers, 2019. Consultado durante Noviembre de 2019.

[9] Docker Inc. Docker overview docker documentation. https://docs.docker.com/get-started/overview/. Consultado el 03-04-2020.

[10] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[11] Stack Overflow. Stack overflow developer survey 2020. https://insights.stackoverflow.com/survey/2020/#technology-most-loved-dreaded-and-wanted-languages-loved, 2020. Consultado el 2/6/2020.

[12] KA Scarfone. *Guide to security for full virtualization technologies*, volume 800. DIANE Publishing, 2011.

[13] Murugiah P. Souppaya, John A. Morello, and Karen Scarfone. Application container security guide. 2017.

APÉNDICE A
Glosario de términos

- **API** *(Application Programming Interface)*: conjunto de llamadas a ciertas bibliotecas que ofrecen acceso a ciertos servicios desde los procesos y representa un método para conseguir abstracción en la programación.
- **REST API**: interfaz entre sistemas que utiliza directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes.
- **Cluster**: conjuntos o conglomerados de servidores unidos entre sí normalmente por una red de alta velocidad y que se comportan como si fuesen un único servidor.
- **Contenedor**: mecanismo de empaquetado lógico en el que las aplicaciones pueden extraerse del entorno en que realmente se ejecutan, facilitando el despliegue uniforme de las aplicaciones basadas en ellos.
- ***Plug-in***: aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica.
- **RPC** *(Remote Procedure Call)*: programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas. De esta manera el programador no tiene que estar pendiente de las comunicaciones, estando estas encapsuladas dentro de las RPC.
- ***Endpoint***: tipo de nodo de red de comunicación. Es una interfaz expuesta por un comunicante o un canal de comunicación.
- **XML** *(eXtensible Markup Language)*: es un metalenguaje que permite definir lenguajes de marcas desarrollado por el *World Wide Web Consortium* (W3C) utilizado para almacenar datos en forma legible. Proviene del lenguaje SGML y permite definir la gramática de lenguajes específicos para estructurar documentos grandes.
- **JSON** *(JavaScript Object Notation)*: formato de archivo estándar abierto, y formato de intercambio de datos, que utiliza texto legible para almacenar y transmitir objetos de datos que consisten en pares de atributos-valores y tipos de datos de matriz (o cualquier otro valor serializable).
- **Certificado digital**: fichero generado por una entidad certificadora que contiene datos de una persona física, jurídica u organismo y además una clave pública y otra privada, que sirven para dar legitimar a esa persona u organismo.
- **Dominio**: nombre único que identifica una subárea de internet, asociando dispositivos conectados en la red con ese identificador. Un ejemplo sería *google.es*.

- **Dirección IP**: conjunto de números que identifica (de forma lógica y jerárquica) una interfaz en red con un dispositivo que utilice el protocolo TCP/IP.
- **Puerto**: interfaz mediante la cual enviar y recibir datos.

APÉNDICE B
Ficheros de configuración

El proyecto cuenta con un fichero de configuración por componente. Estos son tipo JSON, y principalmente tratan los puertos en los que escucharán y/o harán peticiones. Cabe clarificar que para los contenedores en los que se alzará cada componente, la imagen base contará con una distribuición del sistema operativo *Linux*.

B.1 Fichero de configuración del componente *Amphitrite*

Este fichero deberá encontrarse en el direcctorio */etc/amph-config.json* dentro del contenedor. Aquí podemos ver la plantilla de configuración para el componente:

```
{
  "node": "1",
  "restApi": {
    "address": ":8001",
    "tls": {
      "enabled": false,
      "certificate_chain": "",
      "private_key": ""
    }
  },
  "ads": {
    "address": ":8002",
    "tls": {
      "enabled": false,
      "certificate_chain": "",
      "private_key": ""
    }
  },
  "amphInformant": {
    "address": "<amph_informant_address>:8000",
  },
  "envoy": {
    "lds": {
      "listener_filters_timeout": 5
    },
    "cds": {
      "connect_timeout": 25,
      "lb_policy": "ROUND_ROBIN"
```

```
    }
  },
  "logger": {
    "level": "debug"
  }
}
```

- El campo **node** es un identificador único, que una vez iniciado el sistema no se podrá modificar.
- En los campos **restApi** y **ads** los respectivas puertos en los que atienden cada uno a sus peticiones y si la conexión será segura (en consecuencia requerirá especificar certificado y clave privada de la comunicación).
- En el campo **amphInformant**, la dirección a la que podremos hacerle peticiones al componente *Amph-Informant*.
- En el campo **envoy** podemos opcionalmente indicar algunos campos relacionados con cómo va a trabajar *Envoy*:
  - **listener_filters_timeout**: el tiempo de espera para que todos los filtrados de redireccionamiento se completen en segundos. Por defecto serán 15 segundos. Si se quiere desactivar esta opción, especificaremos cero segundos.
  - **connect_timeout**: el tiempo de espera para nuevas conexiónes a los *clusters*.
  - **lb_policy**: la política de balanceo que se va a seguir. Las opciones son:
    - ROUND_ROBIN
    - LEAST_REQUEST
    - RING_HASH
    - RANDOM
    - ORIGINAL_DST_LB
    - MAGLEV
    - CLUSTER_PROVIDED
    - LOAD_BALANCING_POLICY_CONFIG

B.2 Fichero de configuración del componente *Amph-Informant*

Este fichero deberá encontrarse en */etc/amph-inf-config.json* dentro del contenedor.

```
{
  "port": ":8000",
  "restapi": "8005",
  "logger": {
    "level": "debug"
  }
}
```

B.3 Ejecución del proyecto

En este apéndice, vamos a explicar con más detalles el proceso de construcción de las imagenes *Docker* para nuestro proyecto, así como indicar cómo podríamos usarlas para alzar contenedores.

B.4 Proceso de construcción de imágenes *Docker* en múltiples etapas

Para explicar este proceso vamos a poner de ejemplo el *Dockerfile* del componente *Amphitrite*. Por simplificar, vamos a comentar sobre el propio fichero qué ocurre en cada paso.

```
# Parte 1: Compilamos usando una imagen con Golang instalado.

# Indicamos que la imagen base va a ser golang:1.13 y le daremos la etiqueta de builder (para identificarla, cualquiera vale).
FROM golang:1.13 AS builder
# Escogemos como directorio de trabajo una carpeta nueva llamada /amph-builder
WORKDIR /amph-builder
# Copiamos los manifiestos de Go y descargamos las dependencias, para no volverlas a descargar al compilar.
COPY ./go.mod ./go.sum ./
RUN go mod download
# Copiamos la fuente del proyecto
COPY ./pkg/ ./pkg/
COPY main.go .
# Compilamos para sistemas linux
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 GO111MODULE=on go build -a -o amphitrite .

# Parte 2: Creamos nuestra imagen, usando el resultado de la Parte 1

# Indicamos como imagen base envoyproxy/envoy:v1.12.1
FROM envoyproxy/envoy:v1.12.1
WORKDIR /

# Copiamos el ejecutable y el script de arranque.
COPY --from=builder /amph-builder/amphitrite /usr/local/bin/amphitrite
COPY ./start_envoy_amph.sh ./
# Cambiamos los permisos para poder ejecutar el script de arranque e indicamos que cuando se alce el contenedor, debe ejecutar ese script.
RUN chmod u+x ./start_envoy_amph.sh
ENTRYPOINT ["./start_envoy_amph.sh"]
```

Es el mismo proceso para el componente *Amph-Informant*, cambiando el *script* y evidentemente el código fuente.

Hemos subido las imagenes resultantes al *hub* (respositorio) de *Docker*. Para alzar el contenedor, descargaremos esta imagen (en la versión que queramos) y ejecutaremos el comando *docker run*.

En este comando tendremos que pasar los respectivos ficheros de configuración: el de *Envoy Proxy*, el del componente *Amphitrite* y el del componente *Amph-Informant*.

Se deben alojar, respectivamente, en */etc/envoy-config.json*, */etc/amph-config.json* y */etc/amph-inf-config.json*. Además, deberemos exponer los puertos que cada componente usa para comunicarse (especificados en los ficheros de configuración).