UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Despliegue del servicio RoutingMaps en entornos cloud

Trabajo Fin de Máster

Master Universitario en Computación Paralela y Distribuida

Autor: Óscar Morant Moya

Tutor: Francesc Daniel Muñoz Escoí

2019/2020

Resumen

El proceso de despliegue de servicios en entornos *cloud* es complejo, pues se deben gestionar problemas como las posibles interrupciones del servicio, la pérdida de calidad del servicio en determinadas situaciones, la compatibilidad entre versiones de componentes o la gestión del estado del servicio.

En este trabajo académico se analizan los distintos problemas del proceso de despliegue y se diseña e implementa una solución para cada uno de ellos.

Las soluciones obtenidas se aplicarán en el proceso de migración de la aplicación de escritorio de gestión de enrutamiento de vehículos, RoutingMaps, a un servicio desplegado en entornos *cloud*. En este proceso de migración se diseña e implementa una arquitectura basada en microservicios. Todos los componentes del servicio se ejecutarán en contenedores Docker y serán gestionados por el orquestador de contenedores Kubernetes.

Palabras clave: Despliegue, Disponibilidad de servicio, Docker, Kubernetes

Abstract

The process of deploying services in cloud environments is complex, problems such as possible service interruptions, loss of quality of service in certain situations, compatibility between versions of components or management of service state must be managed.

In this academic work the different problems of the deployment process are analyzed and a solution for each one of them is designed and implemented.

The solutions obtained will be applied in the process of migrating the vehicle routing management desktop application, RoutingMaps, to a service deployed in cloud environments. In this migration process, a microservices-based architecture is designed and implemented. All components of the service will run in Docker containers and will be managed by the Kubernetes container orchestrator.

Keywords: Deployment, Service availability, Docker, Kubernetes

Tabla de contenidos

Índice de ilustraciones

1 Introducción y solución

En este trabajo académico se estudian y analizan los problemas que ocurren actualmente en el despliegue de nuevas versiones de un servicio desplegado en entornos *cloud*. Además de intentar solucionar dichos problemas, se va a analizar, diseñar e implementar el primer despliegue y el de las versiones posteriores del servicio RoutingMaps.

Los primeros capítulos del documento se dedicarán a poner en contexto al lector sobre el problema que intenta resolver el servicio RoutingMaps y la complejidad que conlleva realizar una migración de una aplicación de tipo escritorio a una aplicación web basada en microservicios. A continuación, se realizará un análisis de los problemas de ingeniería que se deben resolver para realizar despliegues de un servicio en entornos *cloud* sin detener su actividad y sin perder los datos de las transacciones en curso.

Una vez identificados y evaluados los problemas, se tratará de diseñar una solución utilizando las técnicas de diseño y herramientas software disponibles. A continuación, se detallarán los cambios implementados en el proyecto de migración para cumplir con el diseño aportado y se describirá de qué forma se ha probado y validado. Para finalizar, se expondrán las conclusiones y lecciones aprendidas durante el transcurso de este trabajo.

2 Estado del arte

En este capítulo, se describe de forma detallada la aplicación RoutingMaps, qué problemática ayuda a resolver y cómo se organizan las distintas partes de las que se compone. Además, se analizan las tecnologías y herramientas necesarias para poder realizar la migración a un servicio en el *cloud* y realizar la gestión de despliegues de nuevas versiones.

2.1 RoutingMaps

En este punto se describen los puntos más relevantes del proyecto RoutingMaps. Este proyecto ha sido desarrollado por el Instituto Tecnológico de Informática y lleva en marcha diez años aproximadamente. En primer lugar se realiza una descripción del problema a nivel algorítmico y, posteriormente, se detallan las tecnologías necesarias para llevar a cabo el proceso de migración de aplicación de escritorio a un entorno *cloud*.

2.1.1 Descripción del problema

El problema de enrutamiento de vehículos (VRP) se definió hace más de 40 años por Dantzig y Ramser [1], y es un problema de optimización combinatoria muy costoso a nivel computacional. Consiste en diseñar una lista de rutas para un conjunto de vehículos que deben servir o visitar a determinados clientes ubicados geográficamente de manera dispersa. Los vehículos parten de un almacén o depósito y se debe asignar a cada vehículo la ruta de clientes a visitar que minimice el coste de transporte.

Existen diferentes variantes del problema que añaden variables y restricciones para aproximar la solución a la realidad. Estas variantes añaden mayor complejidad computacional y hacen que el problema se convierta en intratable, aunque el número de vehículos de la flota sea estático y reducido. A continuación, se describen algunas de ellas [2]:

- **VRP con capacidad variable de los vehículos (HFVRP).** Se tiene en cuenta la capacidad de los vehículos y las demandas de los distintos clientes.
- **VRP con ventanas de tiempo (VRPTW).** Se utilizan restricciones de ventanas de tiempo. Cada cliente tiene un intervalo de tiempo en el que debe ser servido. Se asume la misma capacidad para todos los vehículos.
- **VRP con múltiples depósitos (MDVRP).** Existen distintos depósitos o almacenes desde los que puede partir un vehículo. Se asume la misma capacidad para todos los vehículos.

2.1.1.1 Métodos de solución para VRP

Se han investigado y desarrollado diferentes métodos y algoritmos que obtiene una solución óptima para el problema, pero se obtienen con tiempos computacionales elevados, ya que el problema es NP-Completo. Estos métodos no son aplicables a tamaños del problema mayores. Por este motivo, se han propuesto métodos que no garantizan la solución óptima, pero que obtienen una solución de alta calidad en un tiempo determinado.

Existen distintos tipos de métodos para obtener una solución aproximada. Algunos de ellos utilizan heurísticas o metaheurísticas para explorar el espacio de búsqueda obteniendo soluciones de buena calidad en tiempos razonables, por ejemplo, los

algoritmos genéticos [3], los algoritmos de hormigas [4] o las distintas versiones de búsquedas locales. También existen algoritmos en dos fases, donde primero se realiza un agrupamiento de clientes, para después en una fase posterior realizar la asignación de las rutas a los vehículos. Otro tipo de métodos son los constructivos, que construyen una solución factible gradualmente mientras que se comprueba el coste de la solución; el más destacado es el algoritmo de ahorro de Clarke y Wright [5].

Respecto a la paralelización de los métodos de resolución, en muchos casos es difícil aplicar técnicas de paralelización, ya que los datos del problema no se pueden dividir fácilmente. Existe la posibilidad de paralelizar los distintos nodos en un árbol de búsqueda, pero se suele obtener peor rendimiento que en la versión secuencial con tamaños de problema pequeños.

2.1.2 Aplicación de escritorio

La aplicación de escritorio RoutingMaps cubre una de las necesidades que tienen la mayoría de las empresas de logística y distribución, y es el uso eficiente de los recursos de la empresa. Gracias a esta aplicación, se puede realizar la planificación y optimización de rutas de transporte, para cumplir el objetivo de ahorro en tiempo o dinero.

De manera muy simplificada, la funcionalidad básica de la aplicación es la siguiente. El usuario (es decir, la empresa de logística y distribución que utilice la aplicación) proporciona los puntos de entrega o recogida, y los vehículos disponibles. Una vez introducida la información necesaria, se agrupan los puntos de entrega en paradas a realizar dentro de una ruta y se calculan las rutas que deberán realizar los vehículos. Además, el usuario puede visualizar el resultado en un mapa y modificarlo según sus necesidades o preferencias. Después de realizar los cambios necesarios, puede volver a realizar el cálculo de rutas.

La planificación se suele realizar de forma semanal o diaria. Además, se puede replanificar una planificación que ya ha sido iniciada para que se tenga en cuenta la situación actual de los vehículos y se obtenga una solución de mayor calidad.

Desde el punto de vista técnico, la aplicación plantea distintos retos, ya que los cálculos necesarios para resolver el problema planteado requieren un consumo elevado de recursos computacionales y de memoria principal. A continuación, se describen con más detalle.

2.1.2.1 Arquitectura

La aplicación implementa una arquitectura clásica de tres capas [6]: presentación, negocio y datos. La capa de presentación contiene la interfaz desarrollada con el *framework* WPF [1]. La capa de negocio se desarrolló en tecnologías .NET Framework [2] e implementa las reglas de negocio especificadas en los requisitos de la aplicación. En la capa de datos se utiliza el ORM [7] Hibernate [3] para realizar las operaciones de consulta y guardado sobre una base de datos de tipo relacional.

La aplicación se relaciona con dos componentes más, el algoritmo y el servicio de geolocalización. En ambos casos la comunicación se realiza a través de peticiones HTTP

[1] https://docs.microsoft.com/es-es/visualstudio/designers/getting-started-with-wpf?view=vs-2019
[2] https://dotnet.microsoft.com/download
[3] https://hibernate.org/

a un API de tipo REST [8]. A continuación, se muestra un diagrama con las interacciones entre componentes:

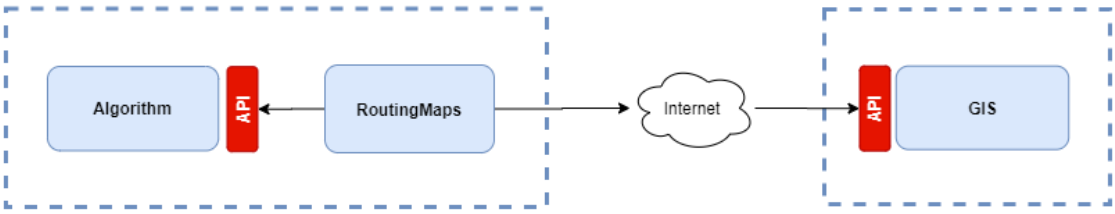

Ilustración 1 – Arquitectura RoutingMaps

2.1.2.2 Geolocalización

Una vez introducidos los puntos de entrega, se deben geolocalizar utilizando sus direcciones. Para ello se hace uso de los servicios de geolocalización (GIS) proporcionados por el proveedor Google [4]. Después de obtener la latitud y longitud de cada punto de entrega, se debe obtener la distancia real entre cada uno de los puntos, lo que se denomina la matriz de distancias. Esta información se almacena en una base de datos y se utiliza para los cálculos del algoritmo.

El uso de la matriz de distancias supone un consumo elevado de memoria principal, ya que si el problema tiene muchos puntos de entrega puede tener un tamaño muy elevado. Si se disponte de *n* puntos de entrega se deberán calcular y almacenar *n * (n - 1)* distancias entre puntos.

El servicio proporcionado por el proveedor Google se utiliza en su modalidad de pago por uso. Actualmente se dispone de 200$ de crédito cada mes para gastar en los distintos servicios, cada solicitud a su API tiene un coste de 0,01$. Debido al coste de uso del API de Google se decidió crear un API propia que hiciera de intermediario y controlara el consumo de cada empresa.

2.1.2.3 Algoritmo

El algoritmo es un componente independiente al que se le envía la información del problema a resolver, los puntos de entrega y los vehículos, y se obtiene una solución que contiene las rutas a realizar por los vehículos. Para resolver el problema se utilizan métodos heurísticos y se ha preparado para admitir restricciones de capacidad de vehículos y ventanas horarias de entrega.

El cálculo de la solución es un proceso computacionalmente costoso, el problema es de orden exponencial y consume gran parte de la CPU de la máquina donde se ejecuta. Es un proceso replicable ya que no tiene ninguna dependencia con ningún componente exterior y no almacena información.

Patrón Mayordomo

El componente del algoritmo se ha desarrollado implementando el patrón Mayordomo [5]. Existe un componente denominado *broker* que recibe las peticiones a través de un API REST y las redirige a los *workers* disponibles. El *worker* obtiene el trabajo a realizar, realiza los cálculos necesarios y devuelve la solución al *broker*. En los siguientes puntos

[4] https://cloud.google.com/maps-platform?hl=es
[5] https://rfc.zeromq.org/spec/7/

se detallan ambos componentes. A continuación, se muestra un diagrama de interacciones entre la aplicación RoutingMaps, el *Broker* y los *Worker*:

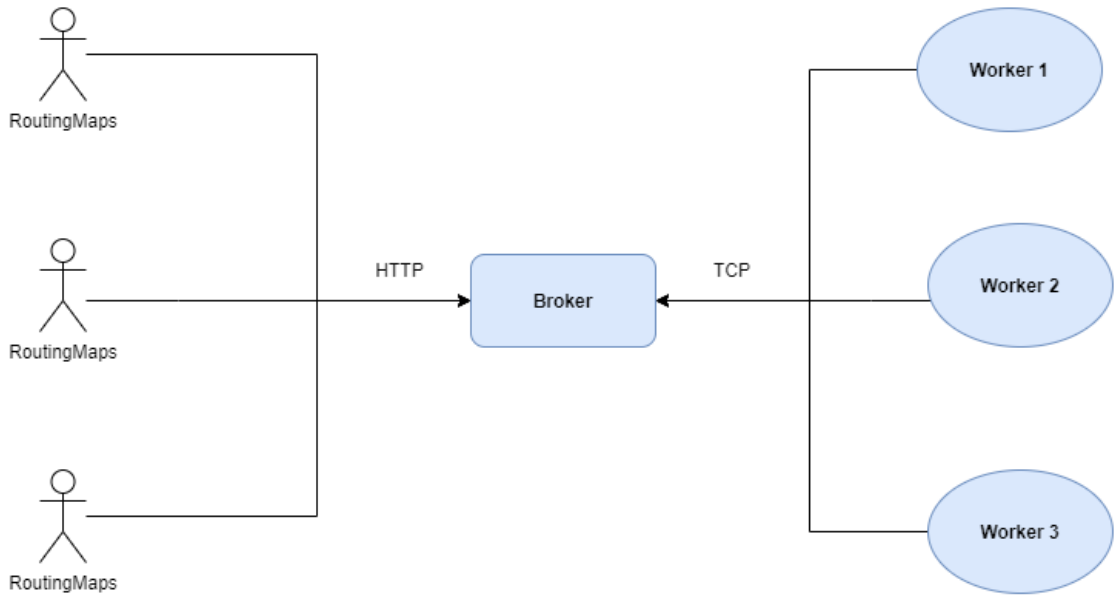

Ilustración 2 – Diagrama de interacción RoutingMaps-Broker-Worker

Broker

El *broker* se desarrolló haciendo uso del *framework* NodeJs [6], el cual se utiliza para publicar el API de tipo REST en el que se reciben las peticiones de optimización por parte de la aplicación.

Las peticiones recibidas se almacenan en la memoria del proceso y se persisten en una base de datos de tipo clave-valor. Existe un proceso interno que se encarga de revisar periódicamente el estado de la lista de peticiones. En caso de existir peticiones pendientes y *workers* disponibles, la petición se extrae de la lista y se envía a uno de los *workers* disponibles.

Además de realizar la gestión de mensajes, el *broker* se ocupa de gestionar el ciclo de vida de los *workers*. Por tanto, al iniciar el proceso se comprueba en la configuración el número de *workers* mínimo y máximo, y se inicia el proceso de escalado hasta alcanzar el número mínimo. El inicio de los procesos de tipo *worker* se puede realizar de dos formas, mediante llamadas al sistema operativo Windows para que inicie el proceso o iniciando contenedores Docker mediante los correspondientes comandos. Una vez iniciado el proceso, se recibe un mensaje de conexión por parte del *worker* y se da de alta en la lista de *workers* disponibles. Además, se comprueba periódicamente la viveza de los *workers* comprobando el tiempo de la última señal de vida. Si se ha superado un tiempo determinado, se elimina de la lista de *workers* y se inicia el proceso de escalado.

A continuación, se muestra un diagrama de flujo de acciones para el proceso de inicio y la interacción entre los clientes, el *broker* y los *workers*.

[6] https://nodejs.org/en/docs/

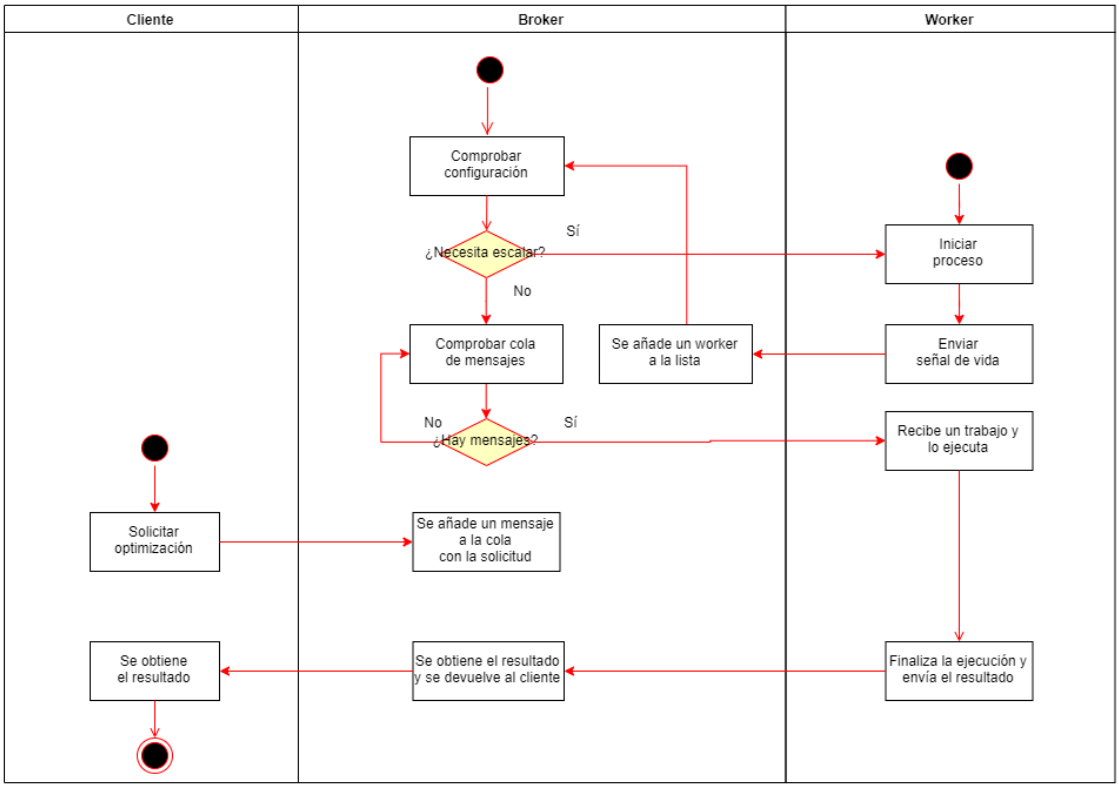

Ilustración 3 - Diagrama de flujo broker

Worker

El *worker* se ocupa de recibir los trabajos, ejecutar los cálculos necesarios para resolver el problema y devolver al *broker* una solución. Al arrancar el proceso se realiza la conexión con el *broker* a través de sockets TCP [9] y envía una señal de vida. A continuación, se queda a la espera de recibir los trabajos a realizar por parte del *broker*. En segundo plano existe un proceso que envía las señales de vida al *broker*.

La comunicación se hace a través de un único canal TCP, minimizando el número de conexiones entre los componentes. Por tanto, existe una conexión por cada *worker* y su uso es bidireccional. Esto es posible, gracias al uso del patrón *Router-Dealer* proporcionado por la librería ZeroMQ [7].

2.2 Máquinas virtuales, contenedores y orquestadores

En este punto se describirán las distintas herramientas y técnicas utilizadas para la distribución y gestión del ciclo de vida del software. Actualmente existen distintas formas o enfoques de publicar un servicio en entornos *cloud*. Se pueden utilizar máquinas virtuales (en adelante MV) con el software necesario preinstalado, generar una imagen de MV y crear las instancias necesarias utilizando dicha imagen. Se pueden utilizar instancias de contenedores, las cuales se crean a partir de imágenes construidas previamente que contienen el software necesario. También se pueden utilizar orquestadores de contenedores, que se ocupan de gestionar el ciclo de vida de contenedores a través de un *cluster* de máquinas físicas o virtuales. En los siguientes puntos se describen las tres formas posibles y sus diferencias.

[7] http://zguide.zeromq.org/

2.2.1 Máquinas virtuales

La virtualización es una técnica que permite crear un entorno simulado (máquina virtual) utilizando un software llamado hipervisor. El hipervisor se ejecuta sobre la máquina física (anfitrión) y permite utilizar sistemas operativos de propósito general (huésped). Las máquinas virtuales pueden acceder a los recursos hardware de forma total o parcial dependiendo de la configuración aplicada. A continuación, se muestra una figura con la pila de software tradicional y la virtualizada:

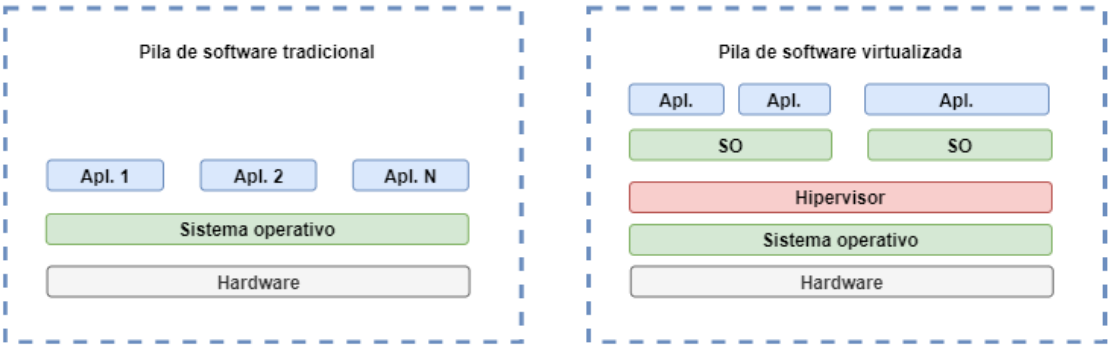

Ilustración 4 - Pila software virtualizada

La virtualización aporta grandes beneficios a los sistemas, ya que permite simular plataformas no disponibles, aporta aislamiento respecto a seguridad y fiabilidad. Permite la escalabilidad vertical y horizontal, ya que se pueden incrementar los recursos de la máquina virtual en tiempo de ejecución o replicar el número de instancias.

Existen gran variedad de plataformas de virtualización como VMWare [8], VirtualBox [9] o Hyper-V [10]. Estos implementan el estándar de máquinas virtuales "*Open Virtualization Format*" [11] (OVF), el cual es abierto, extensible e independiente de la plataforma. Esto permite que una máquina virtual creada usando el formato OVF se puede desplegar en cualquier plataforma que lo soporte.

2.2.2 Contenedores

La tecnología de contenedores se ha convertido en una de las herramientas necesarias para la gestión del ciclo de vida software, ya que se puede utilizar, por ejemplo, en la fase de desarrollo para simular software del que no se dispone, se puede utilizar en la fase de pruebas para iniciar las pruebas con juegos de datos preparados previamente o se puede utilizar para el despliegue y entrega de software.

Los contenedores siguen la filosofía de las máquinas virtuales, donde se tiene un entorno software simulado, pero en este caso de menor tamaño y más ligero. Un contenedor es un proceso del sistema operativo que se ejecuta con ciertas restricciones. Esto hace que el tiempo de arranque sea del orden de segundos, a diferencia de las MV, donde el tiempo de arranque suele suponer varios minutos, dependiendo de los recursos de que dispone.

[8] https://www.vmware.com/es.html
[9] https://www.virtualbox.org/
[10] https://docs.microsoft.com/es-es/virtualization/hyper-v-on-windows/
[11] https://www.dmtf.org/standards/ovf

Los contenedores proporcionan aislamiento respecto al sistema de ficheros y a los demás procesos que se ejecutan en el sistema operativo. Además, permiten limitar los recursos asignados, por ejemplo, la CPU, la RAM o la red.

El uso de contenedores aporta ciertas ventajas sobre las máquinas virtuales, ya que su pila de software es menor. No es necesaria la capa del sistema operativo de cada MV, solo se requiere el SO del host. Se permite compartir las librerías y ejecutables entre instancias de contenedores. A continuación, se muestra una figura con la pila de software para máquinas virtuales y para contenedores.

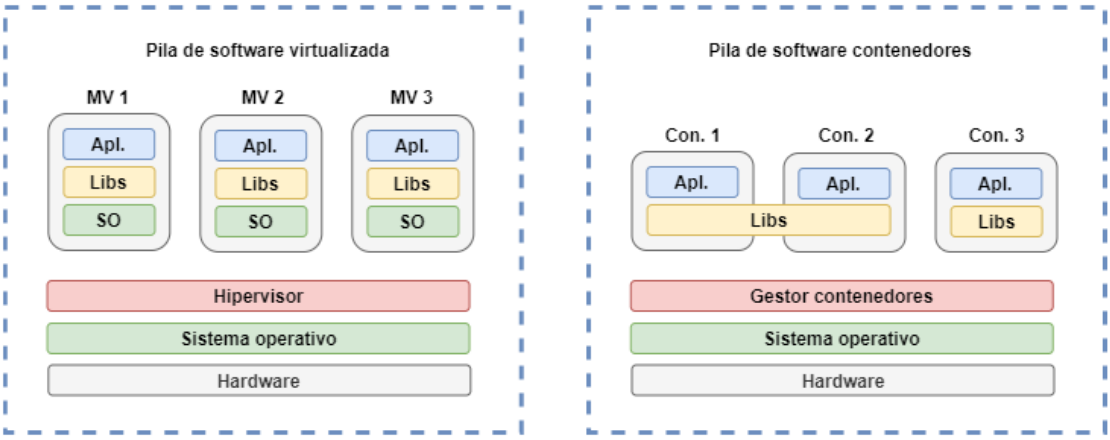

Ilustración 5 - Pila software contenedores

Docker

El gestor de contenedores Docker [12] es uno de los más utilizados en la actualidad. Permite la creación y uso de contenedores en sistemas operativos Linux o Windows. Utiliza recursos del *kernel* de Linux [13] como *cgroups* [14] y *namespaces*, para conseguir el aislamiento entre procesos y permitir que los contenedores se ejecuten como procesos Linux.

Además, se utiliza el sistema de ficheros Overlay [15] que permite tener un árbol de directorios superpuesto y organizado en capas. La capa superior es de escritura y las inferiores de lectura. Este sistema permite compartir las capas de información entre contenedores para su reutilización, por ejemplo, para el sistema operativo y otras capas de software.

Docker proporciona Docker Engine, que es una aplicación cliente-servidor que consta de tres partes, el proceso servidor que se ocupa de gestionar los contenedores, un API REST para comunicarse con el servidor e indicar las operaciones a realizar y un cliente de línea de comandos (CLI), que permite interactuar con el API REST. A continuación, se muestra un diagrama de la arquitectura de este componente, obtenida del sitio oficial de Docker.

---

[12] https://docs.docker.com/
[13] https://www.linux.org/
[14] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01
[15] https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html?highlight=overlayfs

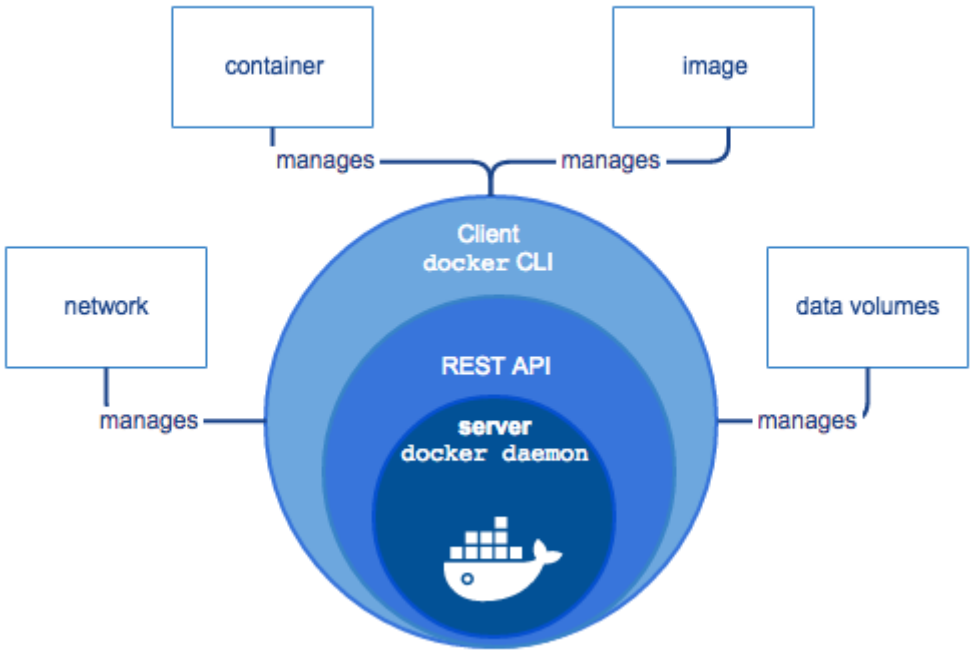

Ilustración 6 - Diseño Docker Engine

Docker administra distintos objetos o recursos como son los contenedores, las imágenes, los volúmenes y las redes. A continuación, se detallarán cada uno de ellos:

- Imagen: es una plantilla que contiene las instrucciones necesarias para crear un contenedor. Una imagen se puede basar en otra y añadir software propio o de terceros sobre ésta. Se pueden crear imágenes propias o utilizar las existentes en registros de imágenes propios o de terceros.
- Contenedor: Es un proceso o instancia en ejecución creada a partir de una imagen. Tiene un ciclo de vida que se puede gestionar a través del CLI o el API de Docker. Un contenedor se puede crear, detener, iniciar de nuevo o eliminar. Además, se pueden conectar a las redes Docker existentes para que puedan interactuar con otros contenedores.
- Volumen: un volumen permite dar acceso a un contenedor al sistema de ficheros del sistema operativo *host*, lo cual permite almacenar la información generada por el contenedor durante la ejecución, aunque el contenedor se detenga o se elimine.
- Red: Docker permite crear un sistema de redes virtuales para la comunicación entre contenedores. Además, un contenedor puede tener acceso a los dispositivos de red del host y heredar su dirección IP.

2.2.3 Orquestadores de contenedores

Los orquestadores de contenedores se utilizan cuando existen contenedores en varios servidores a la vez, o lo que es lo mismo, en un *cluster* de servidores. El orquestador se ocupa de gestionar el ciclo de vida de cada contenedor, balancear la carga de cómputo de cada servidor de forma equitativa, crear grupos de contenedores aislados o de la gestión de errores.

Existen distintos orquestadores *opensource*, como Kubernetes [16], Docker Swarm [17] o Apache Mesos [18]. Este documento se va a centrar en Kubernetes.

2.2.3.1 Kubernetes

Kubernetes es un sistema para automatizar el despliegue, escalado y la gestión de aplicaciones contenerizadas. Puede ser considerada una plataforma de contenedores o una plataforma de microservicios. Proporciona un entorno centrado en los contenedores que orquesta recursos de computación, redes y almacenamiento. Combina servicios de plataforma con servicios de infraestructura y permite la portabilidad entre proveedores de recursos.

Un *cluster* Kubernetes se organiza en nodos de administración (llamados *master node)* y en nodos de trabajo (llamados *node*). En los nodos de trabajo se ejecutan los Pod, que son la unidad básica de trabajo. Cada Pod puede ejecutar uno o más contenedores Docker, todos ellos comparten la misma IP dentro del *cluster*. Además, en cada nodo se ejecuta un agente llamado Kubelet, que se encarga de consultar al API REST del nodo *master* la especificación de Pods necesarios y se ocupa de que estén siempre iniciados y en buen estado. En cada nodo también existe el componente Kube proxy, que se encarga de la gestión de redes y de la comunicación con los Pod desde el exterior del nodo. A continuación, se muestra una ilustración de la arquitectura de Kubernetes.

[16] https://kubernetes.io/es/docs/home/
[17] https://docs.docker.com/engine/swarm/
[18] http://mesos.apache.org/

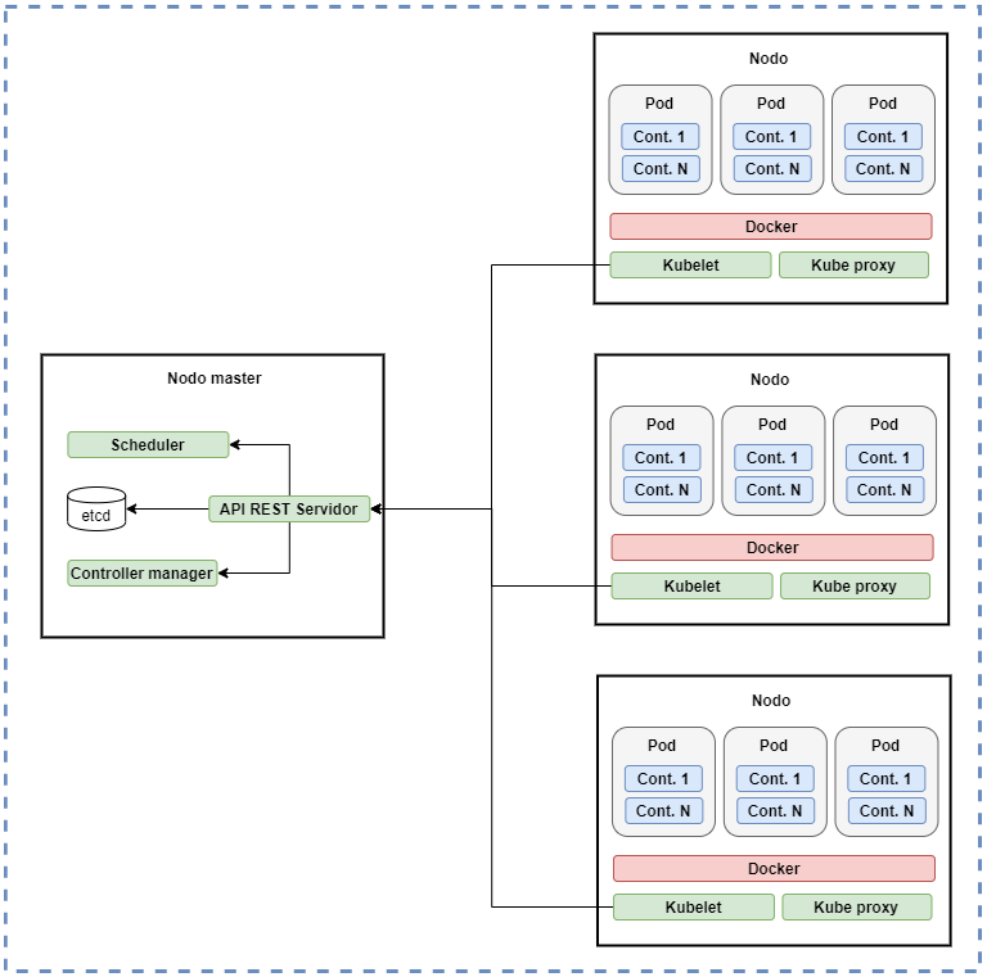

Ilustración 7 - Arquitectura cluster Kubernetes

Kubernetes dispone de distintos objetos, necesarios para organizar los Pod y permitir que sean accesibles desde fuera del *cluster*, por ejemplo, el Deployment, el Replicaset, el Service o el Ingress. A continuación, se describen cada uno de ellos:

- **Replicaset**: se encarga de ejecutar un número de Pods específico en cada nodo del cluster y se ocupa de que siempre exista el mínimo número de réplicas indicado.
- **Deployment**: se encarga de gestionar las actualizaciones declarativas de los Pods y los Replicaset.
- **Service**: permite que un Deployment o Replicaset sea visible y accesible dentro del cluster o desde el exterior si se indica en la configuración. Además, se ocupa de balancear las peticiones entre las réplicas de Pod disponibles.
- **Ingress**: permite acceder a los servicios del *cluster* utilizando los protocolos HTTP o HTTPS, equilibra la carga del tráfico de red y permite utilizar un *proxy* inverso para el enrutamiento de las peticiones.
- **Namespace**: permite crear zonas aisladas o *clusters* virtuales dentro de un *cluster* físico, por tanto, los Pods creados bajo un *namespace* solo podrán comunicarse con los Pods del mismo *namespace* o con el exterior. Esta herramienta es muy útil para definir entornos de aplicación (para *test* o producción) o para mantener distintas versiones de una aplicación.

- **Volume:** un volumen proporciona almacenamiento persistente para el estado de los Pods que se ejecutan en el *cluster*. Este componente es necesario debido a que la información que contiene un Pod se pierde al apagarse.

A continuación, se muestra un gráfico de cómo se organizan algunos de los componentes descritos dentro de un *cluster* Kubernetes.

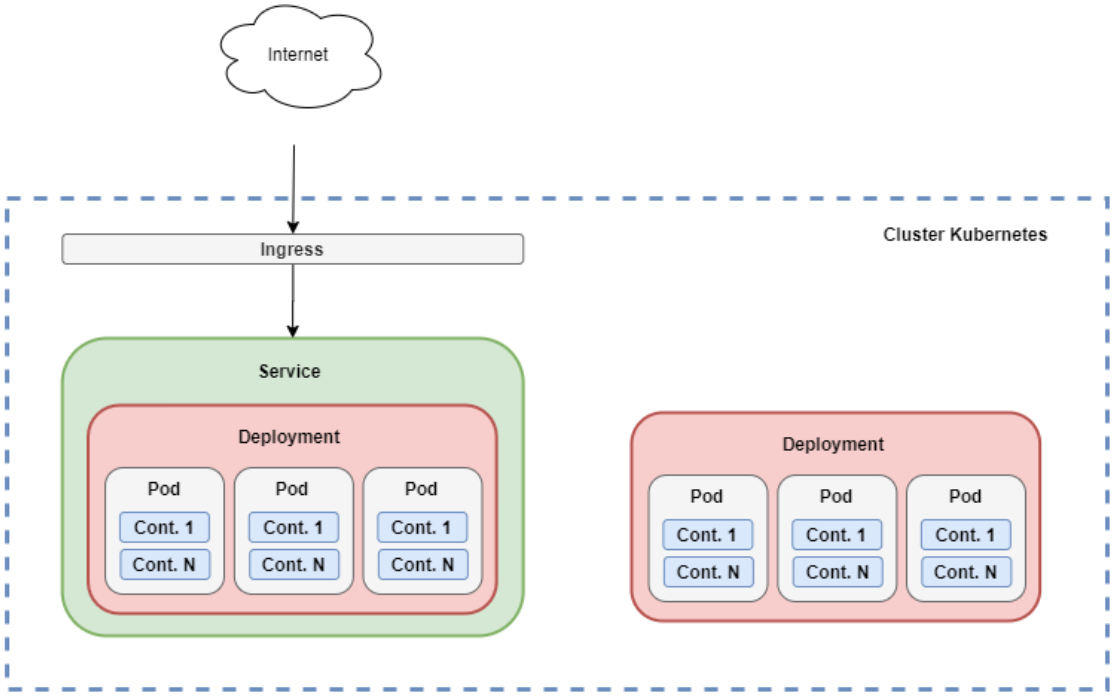

Ilustración 8 - Organización de objetos Kubernetes

Escalado horizontal de Pod

Un elemento fundamental para la gestión del escalado de componentes en Kubernetes es el *Horizontal Pod Autoscaler*. Éste permite incrementar o decrementar de forma automática el número de Pods en un Deployment o Replicaset dependiendo del uso de CPU observado. El componente comprueba de forma periódica el estado de los Pod, a través de un API de métricas que publica cada Pod.

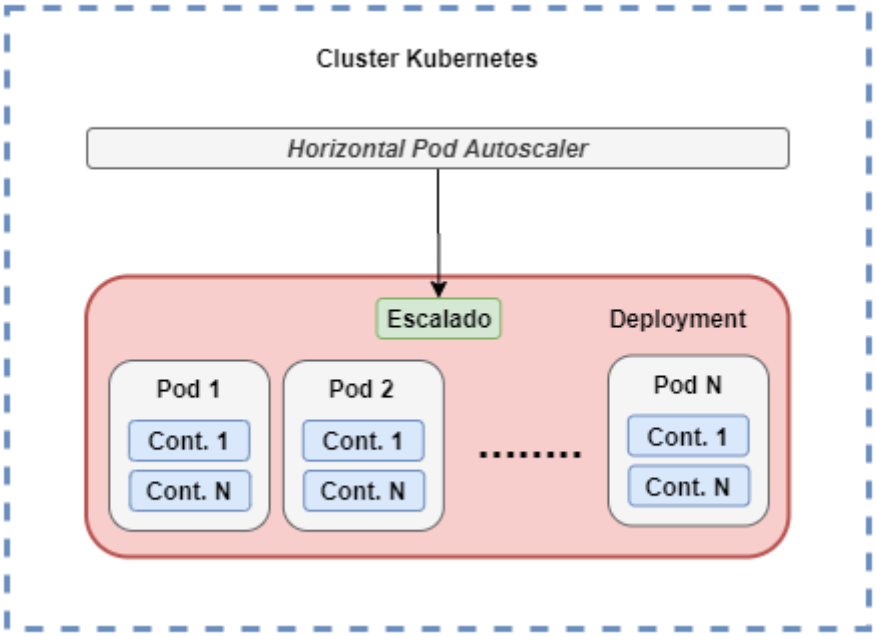

Ilustración 9 - Componente Horizontal Pod Autoscaler

2.3 Herramientas de proveedores *cloud*

En este punto se detallarán las herramientas que ofrecen los distintos proveedores *cloud* para implementar una arquitectura escalable y altamente disponible. Posteriormente se realizará una comparativa de proveedores indicando qué herramientas ofrece cada uno de ellos. A continuación, se describen las herramientas necesarias y más relevantes para el proyecto:

- **Administrador de *cluster* Kubernetes:** esta herramienta permite la configuración y gestión del *cluster*, es muy importante ya que permite el ahorro de tiempo y recursos en la instalación y configuración del software necesario en cada una de las máquinas, y posterior monitorización del estado del *cluster*. El uso de este tipo de herramientas permite a una organización centrarse en el proceso de desarrollo y mantenimiento de las aplicaciones, sin tener que preocuparse de la infraestructura subyacente y sin tener que dedicar personas al mantenimiento del *cluster*.
- **Autoescalado de nodos:** este componente permite incrementar o disminuir el número de nodos del *cluster* de forma automática dependiendo del uso de recursos en cada instante de tiempo. Disponer de este tipo de herramientas ahorra mucho trabajo de ingeniería, ya que la monitorización de cada uno de los nodos del *cluster* y la creación o eliminación de nodos, no son operaciones triviales.
- **Balanceador de carga:** el balanceador de carga permite distribuir las peticiones entre los distintos nodos del *cluster* para equilibrar la carga de forma equitativa.
- **Firewall:** el *firewall* permite restringir o habilitar el acceso al *cluster* desde determinadas direcciones IP, abrir un rango de puertos al exterior o permitir el uso de distintos protocolos de comunicación, como TCP o UDP.
- **Imágenes y copia de seguridad:** el uso de imágenes y copias de seguridad permite guardar una copia de respaldo del estado de cualquier componente

(máquinas virtuales, bases de datos o volúmenes de datos) para recuperarlo en caso de fallo del sistema.

La mayoría de los proveedores de *cloud* actuales ofrecen todas o casi todas estas herramientas, aunque su precio puede variar. Para la comparativa de herramientas se han seleccionado a los proveedores dominantes del mercado, como Google Cloud Platform (GCP) [19], Microsoft Azure [20] o Amazon Web Services (AWS) [21]. También se ha incluido a otros proveedores no tan relevantes, pero que ofrecen precios más asequibles para los mismos servicios, como DigitalOcean (DO) [22] y OVH [23]. A continuación, se muestra una tabla comparativa de las herramientas descritas anteriormente y del coste por mes de una máquina virtual con los recursos mínimos, 1 CPU, 1 GB de memoria RAM y 25 GB de almacenamiento secundario.

	GCP	M. Azure	AWS	DO	OVH
Cluster Kubernetes administrado	Sí	Sí	Sí	Sí	Sí
Auto escalado de nodos	Sí	Sí	Sí	Sí	No
Balanceador de carga	Sí	Sí	Sí	Sí	Sí
Firewall	Sí	Sí	Sí	Sí	Sí
Copia de seguridad	Sí	Sí	Sí	Sí	Sí
Precio mensual MV mínima	6$	8$	8$	5$	4$

Tabla 1 - Comparativa de proveedores cloud

Como se puede observar la mayoría de los proveedores ofrecen todas las herramientas necesarias. La gran diferencia radica en los precios por máquina virtual.

2.4 Frameworks utilizados

En este punto se detallan los distintos *frameworks* que se han utilizado en la implementación del servicio RoutingMaps en *cloud*.

2.4.1 Gestores de colas de mensajes

La gestión de las colas de mensajes es una parte fundamental en sistemas informáticos con componentes que se comunican de forma asíncrona y permite acumular las peticiones en situaciones con gran carga de trabajo.

En este punto se van a describir dos de los gestores de colas de mensajes más utilizados por la comunidad informática, para lo cual se ha realizado una exploración previa del estado del arte. En puntos posteriores del documento, se detallará el análisis y selección de uno de ellos. Los productos que se van a describir son Apache Kafka y RabbitMQ. A continuación, se comentan algunas de sus propiedades y características más relevantes. Para finalizar este punto se describirá el Framework Masstransit, el cual permite abstraerse del gestor de colas de mensaje utilizado y centrarse en la producción y consumo de mensajes.

[19] https://cloud.google.com/docs?hl=es
[20] https://docs.microsoft.com/es-es/azure/?product=featured
[21] https://docs.aws.amazon.com/index.html?nc2=h_ql_doc_do_v
[22] https://www.digitalocean.com/docs/
[23] https://docs.ovh.com/es/public-cloud/

2.4.1.1 Apache Kafka

Apache Kafka [24] es una plataforma de transmisión de datos distribuida, desarrollada en lenguaje Java. Principalmente se utiliza como gestor de colas de mensajes y como gestor de información en tiempo real.

Su arquitectura está diseñada para ser altamente disponible y tolerar fallos en múltiples productores y consumidores de datos. Para conseguir este objetivo se ejecuta en modo *cluster* y aplica el método de replicación pasiva de datos [10]. En este tipo de replicación existe un nodo primario y varios secundarios. El nodo principal recibe el flujo de peticiones y ejecuta las operaciones, y posteriormente envía copias de los datos a los nodos secundarios. En caso de fallo en el nodo principal, uno de los nodos secundarios promociona a nodo principal.

Los mensajes se persisten en cada uno de los nodos según su orden de llegada y se le asigna como identificador el número de registro siguiendo la serie existente.

Los consumidores de mensajes se etiquetan según el grupo de consumidor al que pertenecen y cada mensaje se entrega a una instancia de consumidor de cada uno de los grupos. Esto facilita la replicación de datos. Cada consumidor almacena el identificador del registro consumido, de esta forma se evita procesar mensajes ya procesados sin ocupar apenas memoria del proceso.

Respecto a las librerías de programación, además de Java, existen tanto para tecnologías .NET como para NodeJs, aunque no existen librerías oficiales proporcionadas y mantenidas por el proveedor.

No contiene herramientas de monitorización, ni interfaz de visualización de actividad. Para ello se deben de añadir componentes de terceros.

2.4.1.2 RabbitMQ

RabbitMQ es una plataforma de distribución de mensajes asíncronos desarrollada en Erlang [25]. Permite la ejecución en modo *cluster* para cumplir con los requisitos de escalabilidad y disponibilidad.

Los mensajes se persisten siempre y no se eliminan hasta que se recibe la confirmación por parte del consumidor de que ha sido procesado. Si un mensaje no ha podido ser procesado por un consumidor en un tiempo determinado, vuelve a la cola para reenviarse a otro consumidor. Además, no se envían mensajes a un consumidor mientras esté procesando un mensaje anterior. El reparto de mensajes entre los consumidores se realiza mediante el algoritmo Round-Robin, que permite alternar la entrega de forma cíclica.

Tiene control de viveza de los consumidores mediante el uso de mensajes ACK.

Dispone de herramientas para realizar operaciones de recuperación en caso de que ocurra una partición de red. Aunque se recomienda que, en este caso, los nodos del *cluster* estén en la misma red LAN.

[24] https://kafka.apache.org/documentation/
[25] https://www.erlang.org/docs

Proporciona una aplicación web para realizar tareas de administración y monitorización. En ella se pueden administrar las colas existentes, visualizar los detalles de un mensaje concreto o gestionar los nodos de *cluster*.

El proveedor proporciona librerías para una amplia variedad de lenguajes de programación, entre ellos Java, .NET y NodeJs.

2.4.1.3 MassTransit

MassTransit [26] es un *framework* para tecnologías .NET. Este permite abstraerse del gestor de colas a utilizar e implementar todos los patrones de diseño y comunicación necesarios para una arquitectura *cloud*.

Dispone de implementaciones para utilizar gran variedad de sistemas de colas como RabbitMQ, Azure Service Bus, ActiveMQ, Amazon SQS o colas en memoria principal (solo recomendable para fases de desarrollo o pruebas).

MassTransit trabaja de forma asíncrona y utiliza las librerías de programación paralela TPL, que proporciona .NET, para consumir mensajes de forma simultánea. Se ocupa de la gestión de las conexiones, en caso de desconexión se ocupa de restablecer las conexiones entre los productores o consumidores y la cola de mensajes.

Este *framework* permite implementar gran variedad de patrones de diseño, como el "*Producer-Consumer*", el "*Request-Response*", el SAGAS [11] o la máquina de estados. Uno de los patrones más importantes utilizados es el de la máquina de estados dirigida por eventos.

La definición de los mensajes implicados en los distintos flujos de operaciones se realiza mediante el uso de interfaces. Esto permite definir un contrato de intercambio de información entre los distintos componentes de la plataforma. El *framework* proporciona herramientas de correlaciones entre mensajes y versionado.

MassTransit permite persistir el estado de las operaciones en curso en base de datos relacionales o clave-valor, como SQLServer, MongoDB [27] o Redis. También permite el uso de distintos ORM como Entity Framework [28], NHibernate [29] o Dapper [30]. La persistencia del estado es un punto muy importante para la escalabilidad de los componentes, ya que no se almacena el estado en memoria, lo cual permite realizar la replicación sin añadir complejidad a la arquitectura. Además, en caso de fallo de un componente evita la pérdida de información y facilita la recuperación del estado global del sistema.

2.4.2 Istio

Istio [31] es un producto desarrollado en Golang por Google, IBM y Lift liberado como *opensource*. Permite organizar y gestionar mallas de servicios o microservicios, ayuda a conectar los servicios existentes, añade una capa de seguridad entre servicios y permite la observación y control de estos. Se puede ejecutar en distintas plataformas de gestión de contenedores como Kubernetes, Nomad o Consul.

---

[26] https://masstransit-project.com/
[27] https://docs.mongodb.com/
[28] https://docs.microsoft.com/es-es/ef/
[29] https://nhibernate.info/doc/index.html
[30] https://github.com/StackExchange/Dapper
[31] https://istio.io/latest/docs/

Istio permite administrar el tráfico de entrada y salida de un *cluster* y definir las interacciones de los servicios existentes. Además, permite añadir *timeouts* entre comunicaciones, realizar reintentos, redirigir el tráfico en espejo, redirigir el tráfico según la localización geográfica y utilizar técnicas como *Canary release* y *BlueGreen* para el despliegue de nuevas versiones de una aplicación. En capítulos posteriores se detallarán dichas técnicas.

Istio aporta seguridad en las comunicaciones entre contenedores y pods dentro de un *cluster* a través de TLS mutuos. Además, ayuda a defenderse de ataques de tipo m*an-in-the-middle*, ya que utiliza encriptación de tráfico entre Pods y autenticación mutua, por tanto, solo se pueden realizar comunicaciones entre Pods utilizando certificados de confianza.

Permite monitorizar el tráfico de los Pods y recopilar detalles de telemetría de los servicios. Esto ayuda a identificar problemas casi en tiempo real y permite tomar decisiones sobre el estado de un servicio o componente.

Para conseguir todas las propiedades que se acaban de describir, se utiliza un contenedor *proxy* que se añade en todos los Pods. Esto permite desacoplar la gestión de las comunicaciones de la propia funcionalidad que se está desarrollando.

A continuación, se muestra un diagrama de la arquitectura de Istio, obtenido de la documentación oficial:

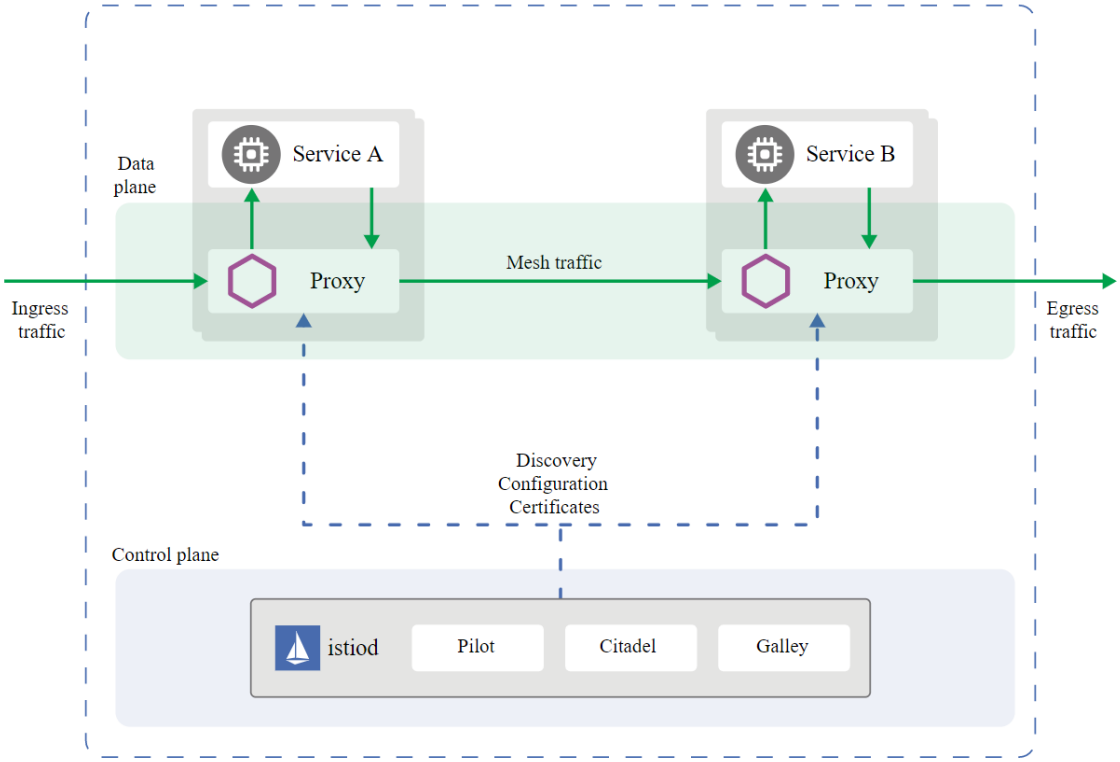

Ilustración 10 - Arquitectura Istio

2.4.3 Prometheus

Prometheus [32] es una herramienta desarrollada en Golang por la empresa SoundCloud y liberada como *opensource*, que permite monitorizar eventos y emitir alertas según una configuración definida. Captura las métricas en tiempo real consultando periódicamente a través de HTTP y permite realizar consultas sobre las métricas almacenadas. Las métricas se almacenan en una base de datos clave-valor.

Prometheus se suele utilizar como extensión de Kubernetes y permite monitorizar y extraer métricas de cada uno de los Pods disponibles.

A continuación, se muestra una ilustración de la arquitectura de Prometheus, obtenida de la documentación oficial. En la parte izquierda se pueden ver las dos formas de recolección de métricas, a través de un g*ateway* de entrada o siguiendo una estrategia *pull* con los distintos componentes conocidos. Además, en la parte superior también se puede ver como se recolecta la información de los servicios de Kubernetes. Toda la información obtenida por los distintos medios se transmite al componente TSDB que se ocupa de persistir los datos en el nodo actual. En la parte derecha se puede ver el gestor de alertas que se ocupa de notificar a través de los distintos medios y los componentes de consulta y representación de la información. Prometheus dispone de una interfaz web de consulta y un API REST, pero se suelen utilizar herramientas como Grafana para hacer una representación gráfica de más calidad.

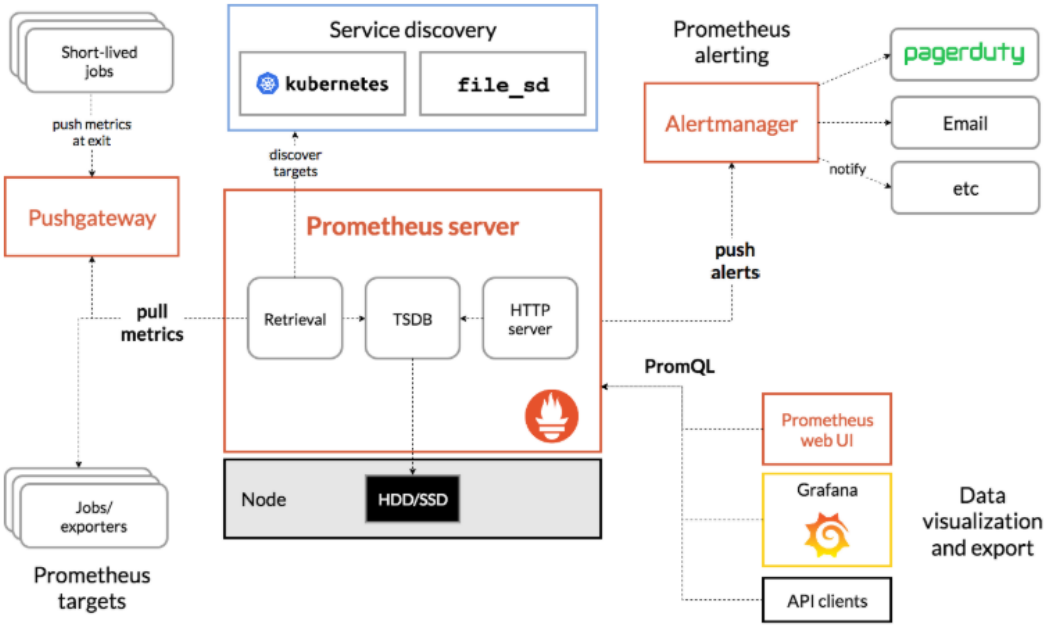

Ilustración 11 - Arquitectura Prometheus

[32] https://prometheus.io/docs/

2.4.4 Grafana

Grafana [33] es una herramienta desarrollada en Golang liberada como *opensource* y que originalmente era un componente de la herramienta Kibana [34]. Permite la visualización de datos de métricas obtenidas desde distintos orígenes de datos. La información se visualiza en cuadros de mandos intuitivos y en multitud de tipos de gráficos distintos. También permite lanzar alertas según una configuración determinada.

2.4.5 SignalR

SignalR [35] es una librería desarrollada por Microsoft que permite la comunicación desde el lado del servidor a una aplicación web cliente haciendo uso de WebSocket [36] o *Long Polling* [37]. Este tipo de herramientas permite el desarrollo de aplicaciones con comunicación asíncrona, ya que se puede enviar al servidor una petición a través de HTTP y recibir su resultado posteriormente a través de SignalR.

SignalR proporciona una API sencilla para realizar llamadas a procedimientos remotos desde el servidor al cliente. El cliente se subscribe a un evento con un identificado determinado y registra una función o acción a realizar cuando ocurra dicho evento. Además, es posible emitir eventos dirigidos únicamente a una conexión, a un grupo de conexiones o a todas. También permite administrar eventos de conexión y desconexión de usuarios y realizar acciones si se requiere.

2.5 Base de datos

2.5.1 SQL Server

SQL Server [38] es un sistema gestor de base de datos relacional desarrollado por Microsoft. Utiliza el lenguaje de desarrollo Transact-SQL, una implementación de SQL que permite consultar y modificar datos, crear tablas y definir relaciones entre ellas. A continuación, se describen algunas de sus características más destacables:

- Soporta transacciones: esto garantiza que las operaciones de modificación de conjuntos de datos sean:
    - Atómicas: se realizan todas las operaciones o ninguna.
    - Coherentes: se respetan todas las reglas de integridad de los datos).
    - Aisladas: una operación no modifica los datos de otra transacción en curso.
    - Durables: al finalizar una operación, las modificaciones perduran en el tiempo.
- Soporta procedimientos almacenados: los procedimientos almacenados permiten ejecutar conjuntos de sentencias SQL de actualización o consulta de datos. Permiten implementar la lógica de negocio de una aplicación, tratando los datos de forma muy rápida y eficiente, ya que las operaciones se realizan en el contexto del motor de base de datos.

[33] https://grafana.com/docs/
[34] https://www.elastic.co/es/downloads/kibana
[35] https://docs.microsoft.com/es-es/aspnet/signalr/
[36] https://tools.ietf.org/html/rfc6455
[37] https://www.hjp.at/doc/rfc/rfc6202.html
[38] https://www.microsoft.com/es-es/sql-server/sql-server-downloads

- Consultas distribuidas: si se dispone de distintos orígenes de datos, se pueden enlazar y utilizar de forma conjunta en las sentencias de consulta.
- Dispone de un sistema robusto de privilegios y permisos para todas las acciones y recursos disponibles.
- Dispone de copias de seguridad y restauración.
- Permite la replicación parcial o completa entre distintas instancias de base de datos para mejorar la disponibilidad y tolerancia a fallos.

2.5.2 Redis

Redis [39] es un sistema gestor de base de datos NoSQL en memoria, aunque opcionalmente se puede utilizar en modo persistente. Utiliza estructuras de datos de tipo diccionario (clave-valor) o tablas *hash*, listas ordenadas o conjuntos de datos desordenados. A continuación, se describen algunas de sus características más destacables:

- Permite realizar la persistencia a múltiples niveles (si se configura para tal uso). Inicialmente los datos se almacenan en la memoria RAM y pasado un cierto tiempo los datos se transmiten al disco duro. Además, dispone de cache en el cliente.
- Permite indicar la caducidad de los datos.
- Soporta replicación de tipo maestro-esclavo y los esclavos a su vez pueden ser maestros de otros nodos, lo que permite organizar la replicación en árbol.
- Implementar el patrón publicador-subscriptor.
- Soporta transacciones y bloqueos distribuidos.
- Admite el particionado de datos entre instancias.

[39] https://redis.io/documentation

3 Planificación

En este capítulo se describirá la planificación realizada para este trabajo académico, así como de la migración del servicio RoutingMaps y su posterior despliegue como servicio en entornos cloud. Además, se describirá en qué estado se encontraba el proyecto al iniciar el trabajo académico y en que partes del proyecto han intervenido otras personas del equipo. Para finalizar, se describirá el tiempo necesario para realizar la solución del problema que se intenta resolver y el tiempo necesario en la formación de las personas.

3.1 Situación inicial del proyecto y equipo

Debido a que el proyecto se trata de una migración de un tipo de aplicación a otro, existen partes de éste que ya estaban implementadas y que se han reutilizado o que se han utilizado aplicando modificaciones menores. Esto se debe a que determinados componentes se diseñaron e implementaron con el objetivo de reutilizarse en distintas plataformas y escenarios. Por ejemplo, el componente de cálculo del algoritmo se desarrolló utilizando tecnologías multiplataforma y sin dependencias con sistemas externos. Todo ello ha permitido acelerar el arranque del proyecto y centrarse en el diseño e implementación de la nueva arquitectura. A continuación, se describen algunas de las partes o componentes del proyecto que se han reutilizado:

- El esquema de base de datos se ha conservado al completo y se ha utilizado un subconjunto reducido de las tablas existentes debido a que la funcionalidad ofrecida en esta versión es menor.
- De la aplicación de escritorio, se ha dejado de utilizar la capa de presentación (desarrollada en WPF) y se utilizan las capas de negocio y datos. La capa de negocio se publica haciendo uso de una API REST.
- Para la optimización y resolución del problema se conservan los componentes de optimización y el *broker*. Ambos estaban preparados para ejecutarse en contenedores Docker.

Como ocurre en la mayoría de los proyectos de gran envergadura existe un equipo multidisciplinar que se encarga de diseñar, implementar y probar la funcionalidad de la aplicación. En este caso se dispone de dos personas encargadas de analizar los requisitos funcionales de servicio y posteriormente validarlos. Dos personas han realizado tareas de análisis y diseño de la arquitectura, el responsable del proyecto y yo. Las tareas de desarrollo de los distintos componentes las han llevado a cabo un equipo de cuatro personas, yo incluido.

3.2 Planificación del trabajo académico

La planificación del trabajo académico se ha realizado siguiendo el esquema de planificación en cascada de un proyecto software. Algunas fases tienen cierto solape, esto se debe a que en algunas ocasiones estas tareas suelen intercalarse, a veces al realizar un análisis se tiene que volver a investigar cierto concepto o cuando se prueba una implementación y hay errores se tienes que realizar correcciones. A continuación, se muestra un diagrama de Gantt donde se representa el tiempo estimado para cada fase.

Tareas	Tiempo en días	Marzo	Abril	Mayo	Junio	Julio	Agosto	Septiembre
Investigación	60							
Análisis	60							
Diseño	60							
Implementación	30							
Pruebas	15							
Conclusiones	7							

Ilustración 12 - Gantt planificación

Como se puede observar gran parte del tiempo se invierte en las fases previas a la implementación. Esta distribución de tiempo suele dar como resultado proyectos de mejor calidad.

3.3 Tiempo necesario para dominar las distintas tecnologías

En este proyecto se utilizan una gran variedad de tecnologías para las cuales se requiere cierto tiempo de formación e investigación del personal que integra el equipo de desarrollo. En este punto se intenta detallar el tiempo necesario para adquirir los conocimientos necesarios, aunque esto puede depender de distintos factores como la experiencia profesional previa, el nivel de estudios académicos y las aptitudes de la persona en cuestión. Por este motivo se va a indicar el tiempo invertido aproximado para mi caso particular.

A continuación, se detallan las tecnologías más importantes para el proyecto y los tiempos necesarios para dominarlas:

- **.NET Core:** debido a mi carrera profesional, acumulo más de cinco años de experiencia y no he tenido que formarme a propósito para este proyecto. Si un desarrollador tuviera que formarse desde cero en esta tecnología, debería dedicar uno o dos meses para dominarla a un nivel medio. Aunque esto depende de los conocimientos previos de programación básica.
- **MassTransit:** he tenido que formarme a propósito en esta tecnología para el proyecto y tuve que invertir dos semanas antes de poder implantarla en el proyecto.
- **Docker:** aprendí Docker en varias de las asignaturas de este Máster. Con un mes de formación se puede llegar a un nivel avanzado.
- **Kubernetes:** he tenido que aprender esta tecnología para utilizarla en este proyecto, aunque tenía conocimientos previos gracias a una asignatura de este Máster. He necesitado un mes de formación.
- **Azure DevOps:** Es una herramienta muy intuitiva y existen muchos manuales. En una semana se pude aprender lo necesario para gestionar todas las fases de desarrollo de software.

4 Análisis del problema

En este capítulo, se analizan los problemas y dificultades que conlleva el despliegue de la aplicación de escritorio RoutingMaps y se describe de forma resumida el proceso de migración a un servicio disponible en el *cloud*. Además, se analiza qué problemas se resuelven respecto a la aplicación de escritorio y qué nuevos problemas se deben resolver en el servicio *cloud*.

4.1 Despliegue de la aplicación de escritorio

El despliegue que se realiza de la aplicación de escritorio RoutingMaps es bastante manual y rudimentario, ya que no se utiliza ninguna herramienta que automatice el proceso y requiere de la intervención de una persona en todo el proceso. A continuación, se detallan los pasos a seguir en el proceso de despliegue en los equipos de los usuarios finales:

1. Se realiza un nuevo desarrollo o una corrección de error y se realiza el correspondiente proceso de pruebas y validación del cambio.
2. Se genera un instalador de la aplicación a partir de las librerías y archivos ejecutables de la aplicación. Para ello, el desarrollador responsable de generar el instalador realiza la compilación y publicación del proyecto. A continuación, copia los archivos obtenidos en la aplicación encargada de generar el instalador. Para finalizar, se realizan las configuraciones necesarias y se genera el archivo de instalación.
3. Se preparan los archivos de configuración para cada empresa usuaria. Estos archivos contienen, por ejemplo, los datos necesarios para conectar con la base de datos u otros servicios requeridos.
4. Si hay cambios en el esquema de base de datos o se requiere realizar una migración de datos, se preparan los archivos con los *scripts* correspondientes. Es posible que existan *scripts* de migración de datos específicos para cada empresa cliente.
5. Se envían los archivos entregables al departamento de sistemas de las empresas usuarias, para su posterior distribución a los usuarios finales.
6. Para finalizar, los técnicos de cada departamento de sistemas deberán aplicar los cambios de base de datos, ejecutar el instalador en el equipo de cada usuario y copiar los archivos de configuración en el directorio de instalación seleccionado.

Como se puede observar, el proceso es totalmente manual y es muy susceptible de que ocurran fallos humanos, ya que requiere de un cierto nivel de organización por parte de todas las personas implicadas en el proceso. El desarrollador encargado de generar el instalador debe asegurarse que el contenido del instalador es el correcto y no el de una versión anterior. La gestión de los distintos archivos de configuración de cada empresa cliente se puede convertir en una tarea compleja y es fácil cometer un error al prepararlos, sobre todo si la empresa cliente dispone de distintos entornos. Además, la generación de los entregables la pueden realizar distintas personas, por tanto, se debe seguir un procedimiento para no cometer errores en los distintos pasos. Otro de los focos de posibles errores son los técnicos responsables de realizar la instalación en los equipos de los usuarios finales, ya que también deben organizarse adecuadamente para no

utilizar el instalador de otras versiones o utilizar archivos de configuración que no estén actualizados.

4.2 Transición de aplicación de escritorio a web

Después de casi diez años de desarrollo y mantenimiento de la aplicación RoutingMaps en plataformas de tipo escritorio, se decidió realizar una migración de la aplicación a plataformas de tipo web y *cloud*. Las aplicaciones de tipo web aportan propiedades de gran valor a nivel técnico, no requieren instalación en la máquina del usuario, son independientes del sistema operativo y son fáciles de mantener. Además de estas propiedades, también se desea que se cumplan algunos requisitos indispensables en cualquier aplicación web actual, como son la tolerancia a fallos, el manejo de gran volumen de peticiones o la alta disponibilidad.

El desarrollo de la nueva aplicación web se realizará de forma iterativa y prototipada, tanto a nivel funcional como a nivel técnico. La primera versión será muy simple desde el punto de vista funcional. El usuario importará los puntos de entrega desde un fichero CSV y los geolocalizará. Además, dará de alta los vehículos disponibles, hasta un máximo de cinco. A continuación, se realizará la resolución del problema y se obtendrá una solución, pudiendo realizar cambios en la solución y resolver de nuevo. Para finalizar, el usuario podrá exportar la solución a un fichero CSV. A continuación, se muestra un diagrama con el flujo de acciones:

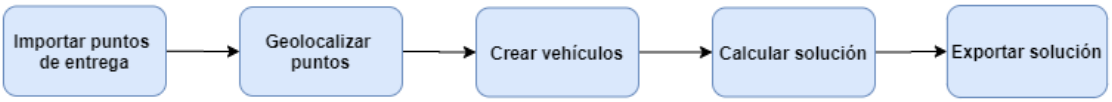

Ilustración 10 - Flujo acciones aplicación web

Existirán otras funcionalidades auxiliares, necesarias para la administración y mantenimiento de la aplicación, como la facturación, el registro de operaciones contables, el registro de actividad y el control de acceso.

En la primera versión de la aplicación, la información de los vehículos o las soluciones obtenidas no se persistirá en una base de datos. Simplemente se almacenará en la memoria del navegador y se eliminará al cerrar la sesión del usuario o al cerrar el navegador.

Desde el punto de vista técnico, para poder cumplir con los requisitos comentados anteriormente, se ha decido diseñar e implementar una arquitectura basada en microservicios, donde los componentes de la plataforma se dividan según su funcionalidad de negocio. Cada componente se ejecutará de forma independiente y podrán comunicarse mediante el protocolo HTTP o mediante *sockets* TCP. Los componentes se comunicarán utilizando mensajes, estos se podrán enviar de forma directa entre componentes o haciendo uso de un gestor de colas de mensajes, que permita acumular los mensajes y procesarlos dependiendo de los recursos disponibles. Para asegurar el aislamiento y facilitar la replicación, los procesos de los componentes se ejecutarán en contenedores Docker. Los contenedores se ejecutarán sobre una plataforma de gestión y orquestación, como Kubernetes, que se ocupe de las tareas de replicación, identificación de servicios o enrutamiento de peticiones. En los siguientes puntos de detallarán cada una de las partes de la plataforma.

4.2.1 Gestión de colas de mensajes

En capítulos anteriores se han descrito los detalles de los gestores de colas de mensajes Apache Kafka y RabbitMQ. En este punto se detallan los motivos de la selección de uno de ellos para su posterior utilización.

Para la selección del producto idóneo, además de realizar el análisis previo, se han implementado prototipos del patrón productor-consumidor para ambos productos, con el objetivo de observar su comportamiento en situaciones de alta carga de trabajo y fallo de alguno de los nodos.

Ambas soluciones han soportado las pruebas de estrés sin perder ningún mensaje y manteniendo el servicio siempre disponible.

Durante las pruebas, se ha observado que el proceso de balanceado de carga entre consumidores en Kafka no se adecúa a los requerimientos del proyecto. Esto se debe a que, en un grupo de consumidores, los mensajes siempre se reciben en la misma instancia y solo se cambia de instancia en caso de error. Esto hace que el trabajo no se reparta de forma eficiente ni igualitaria, ya que en la mayoría de los casos se tienen instancias sin recibir trabajos y consumiendo recursos de la máquina. Se investigó la posibilidad de hacer un rebalanceo de carga de forma cíclica, pero según las especificaciones del producto, esto supone un elevado coste computacional debido a la propia arquitectura del producto. Por este motivo, finalmente se seleccionó la solución RabbitMQ.

4.2.2 Bases de datos

En la aplicación de escritorio la gestión de los datos se realizaba utilizando únicamente el gestor de base de datos SQL Server. En la versión de servicio *cloud* la gestión de los datos se realizará de forma diferente. No solo se utilizarán bases de datos tipo SQL, también se utilizará bases de datos NoSQL, como Redis, o la memoria del navegador del usuario.

El esquema de base de datos SQL heredado de la versión anterior se mantendrá sin cambios y se utilizará para almacenar los puntos de entrega, los usuarios y las configuraciones por defecto del servicio. En cambio, la información de los vehículos no se persistirá en la base de datos y se mantendrá en la memoria del navegador, al cerrar sesión o recargar la página esta información se perderá.

Los resultados del cálculo de la matriz de distancias entre los puntos de entrega se persistirán en una base de datos Redis y se mantendrán mientras el usuario esté trabajando en la solución del problema.

La solución del problema obtenida después del cálculo del algoritmo tampoco se persistirá. Una vez obtenida se enviará al navegador. Posteriormente, el usuario tendrá la posibilidad de realizar la exportación a un fichero en formato CSV.

4.3 Despliegue del servicio cloud

El despliegue de un servicio en entornos *cloud* aporta grandes ventajas sobre el de una aplicación de escritorio. Esto se debe a que el proceso se centraliza en un único punto y esto hace que la probabilidad de producirse un error se reduzca. Además, suele ser un proceso más automatizado, aunque esto depende de la organización y nivel de madurez del proyecto. Suele ocurrir que, aunque se esté desarrollando una nueva aplicación para un servicio *cloud*, la fase de despliegue no se trate en las fases iniciales del proyecto, ya que los esfuerzos se concentran en desarrollar una primera versión. Esto hace que los primeros despliegues también se realicen de forma semiautomática.

La automatización del proceso de despliegue es la clave para reducir la probabilidad de error o fallos al mínimo, ya que no es necesario que intervenga ninguna persona en el proceso.

Una parte fundamental en este proceso es el uso de contenedores Docker, facilita las tareas de generación y entrega de nuevas versiones de software. Otra parte fundamental es el uso de herramientas de integración continua, que se ocupen de gestionar y automatizar las tareas de despliegue, como la compilación del software o la generación de imágenes Docker.

Aunque se tenga un proceso de despliegue completamente automatizado y maduro, existen problemas que se deben resolver. Estos problemas se ocasionan debido a la propia naturaleza de un servicio en *cloud*. El objetivo es tener un servicio escalable, altamente disponible y que soporte fallos de distintos tipos. El hecho de que el servicio esté siempre disponible supone un problema para el proceso de despliegue, ya que no se puede detener el servicio durante cierto tiempo para aplicar los cambios de software, además, existen distintos sectores de la industria (banca, sanidad, transportes, etc.) en los que no se puede permitir una parada del servicio.

En la actualidad existen distintas técnicas para la gestión de los despliegues como, por ejemplo, *Blue Green Deployment* o *Canary Release*. A continuación, se describirá cada técnica y se analizará qué ventajas e inconvenientes tienen.

4.3.1 Canary Release

La técnica de *Canary Release* [12] consiste en crear un entorno paralelo con la nueva versión del servicio *cloud* y redirigir un porcentaje de peticiones de los clientes al nuevo entorno para observar cómo se comporta la nueva versión. El porcentaje de peticiones que se redirigen se va incrementando gradualmente hasta que, finalmente, se deja de utilizar la versión anterior. Esta técnica permite validar de forma segura el despliegue de una nueva versión sin interrumpir el servicio. Además, en caso de ocurrir algún error, se puede dejar de redireccionar las peticiones sin afectar a la mayoría de los usuarios, esto reduce de forma notable los riesgos en el proceso de despliegue.

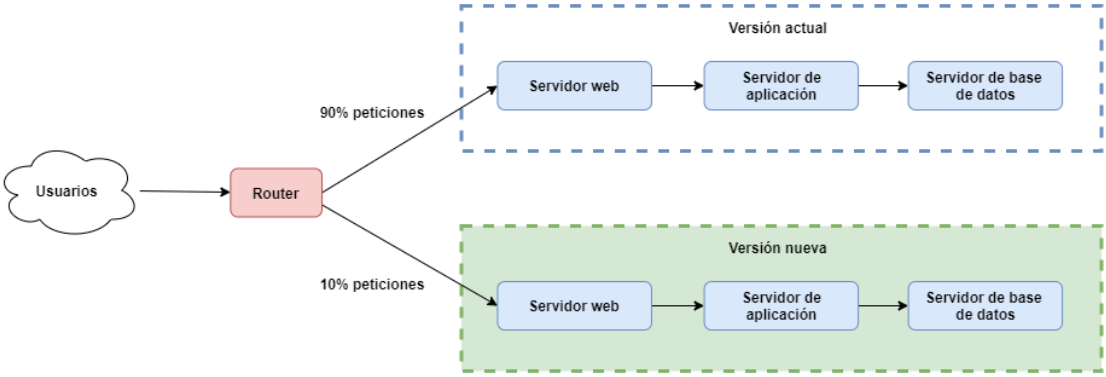

Ilustración 13 - Técnica Canary Release

Esta técnica también permite realizar pruebas de carga para la nueva versión, la carga se puede incrementar de forma gradual y se pueden analizar las métricas para observar el impacto que tiene la nueva versión sobre el entorno de producción.

La selección de los usuarios que son redirigidos puede ser aleatoria, seleccionando a un grupo de empleados de la propia empresa o según una característica del perfil de usuario.

Uno de los inconvenientes de esta técnica es que se deben administrar y gestionar varias versiones del servicio a la vez y esto puede ser una tarea muy compleja, sobre todo en cuanto a la gestión del estado de los datos. Por ejemplo, en caso de error en la nueva versión, la transacción se debe revertir en todos los componentes implicados y aplicarla en la versión anterior.

Otro punto difícil de gestionar es cuando se distribuye software instalable como una aplicación móvil, donde no se tiene tanto control en la actualización del software, ya que el usuario puede no instalarse la nueva versión. Otro caso serían las aplicaciones web de tipo SPA, donde no se recarga habitualmente la página actual.

4.3.2 Blue Green Deployment

La técnica *Blue Green Deployment* [13] consiste en tener dos entornos donde, por ejemplo, el azul sea el de la versión actual del servicio y otro verde donde se disponga la nueva versión del servicio. Existe un *router* como punto de entrada de las peticiones de los usuarios, una vez validado el entorno verde, se redireccionan todas las peticiones hacia él y el azul queda sin actividad. En caso de que exista algún problema con la nueva versión, se puede revertir el cambio y redireccionar las peticiones al entorno azul. Una vez terminado el proceso, el entorno anterior (azul) se puede actualizar y mantener como copia de respaldo, sirviendo también de nuevo entorno para el siguiente cambio de versión.

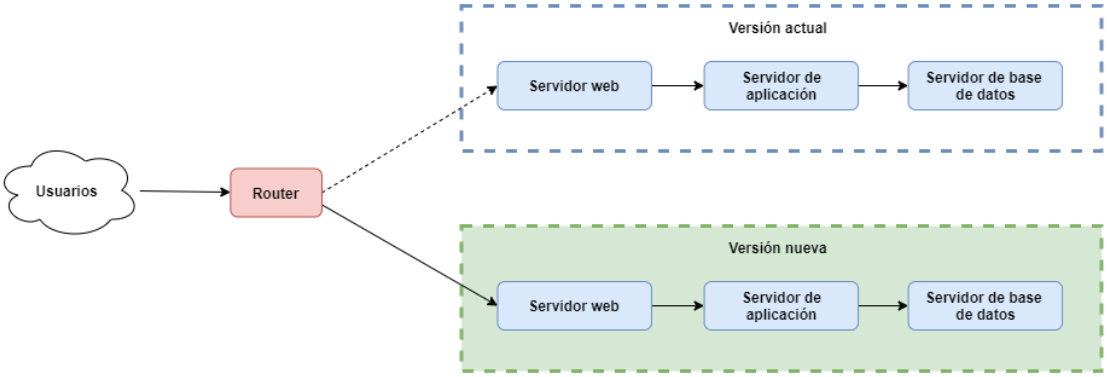

Ilustración 14 - Técnica Blue Green Deployment

Un inconveniente de esta técnica son las transacciones realizadas en el nuevo entorno y que se podrían perder por un fallo. En estos casos, se deben procesar las transacciones en ambos entornos, para evitar la pérdida de información. Otro inconveniente es la gestión de los cambios en el esquema de base de datos, ya que el proceso de actualización se debe realizar por fases y el proceso puede ser complejo.

4.3.3 Patrones de desarrollo software

En este punto se comentan los patrones de diseño software que permiten la compatibilidad entre dos versiones de un servicio o aplicación.

4.3.3.1 ParallelChange

La técnica *ParallelChange* [14] se aplica a nivel de software. En ocasiones las interfaces o contratos del software pueden cambiar creando incompatibilidades con sus consumidores. En esta técnica se propone dividir el cambio en tres fases: expandir, migrar y contraer.

Dado un cambio de software donde se modifica una interfaz, en la fase de expansión en lugar de modificar directamente la función o método en cuestión, se añadiría un método para la nueva signatura y se mantendría la anterior. De esta forma la nueva interfaz admitirá tanto la versión antigua como la nueva y los consumidores podrán utilizar ambas. En la fase de migración, se actualiza a los clientes para que utilicen la nueva versión, esta fase se puede alargar si existen consumidores externos de los cuales no tenemos el control. Una vez que todos los consumidores han migrado a la nueva versión, se puede aplicar la fase de contracción, que consiste en eliminar de la interfaz los métodos con la firma antigua.

Este tipo de técnicas reducen el riesgo de los cambios al facilitar la migración de clientes y aplicar los cambios de forma incremental

4.3.3.2 Branch by Abstraction

La técnica *Branch by Abstraction* [15] permite realizar cambios de software a gran escala y de manera gradual, publicando lo cambios regularmente mientras el cambio se encuentra en desarrollo.

Inicialmente se tiene un módulo o pieza de software que se utiliza por distintos consumidores o clientes.

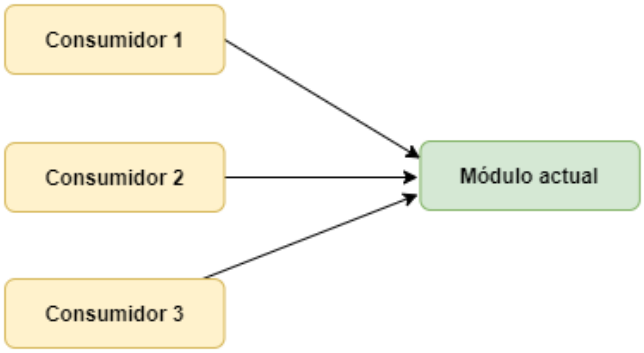

Ilustración 15 - Situación inicial en Branch by Abstraction

A continuación, se crea una capa de abstracción que actúa entre el módulo y su consumidor. Además, se modifica el consumidor para utilizar la nueva capa de abstracción.

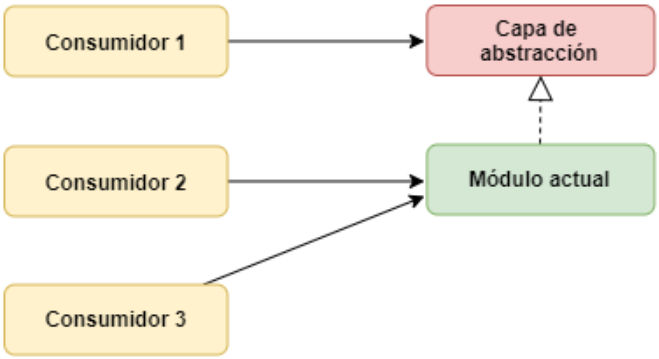

Ilustración 16 - Creación de la capa de abstracción

De forma gradual, se migran todos los consumidores para utilizar la capa de abstracción hasta que todas las interacciones se realizan a través de ella.

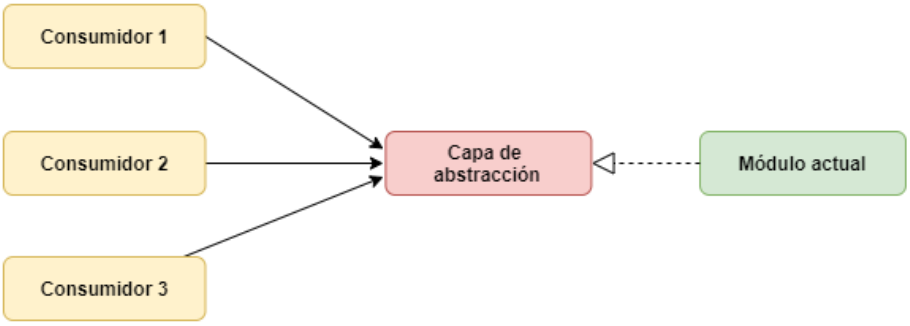

Ilustración 17 - Todos los consumidores usan la capa de abstracción

En este punto, ya se puede empezar el desarrollo de la nueva funcionalidad en el módulo. Este módulo se implementará para que se utilice la capa de abstracción.

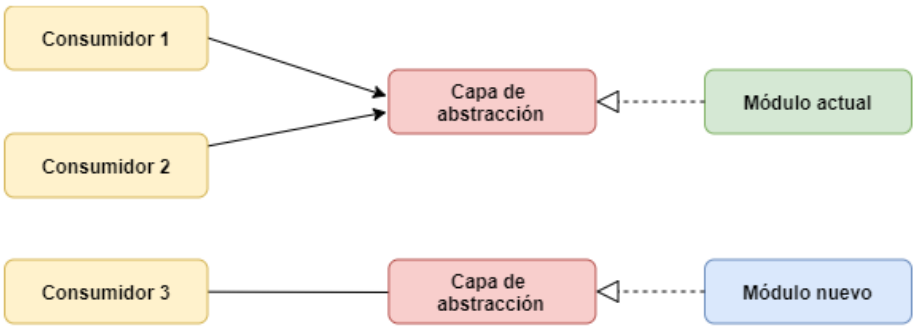

Ilustración 18 - Creación de nuevo módulo

Una vez terminado el módulo, se puede migrar de forma gradual todos los clientes o consumidores.

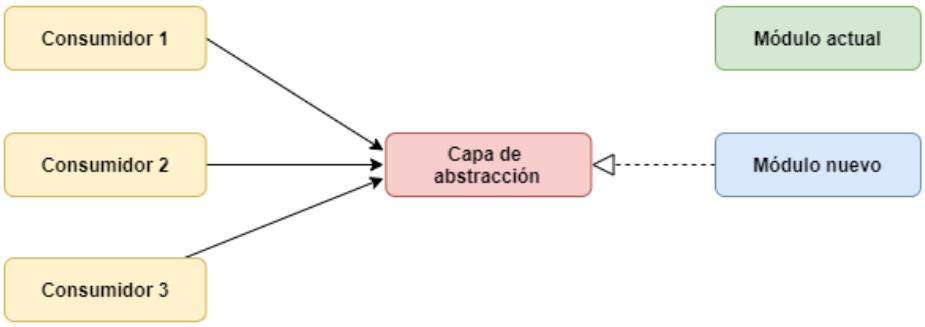

Ilustración 19 - Todos los consumidores usan el nuevo módulo

4.3.4 Conclusiones

Como se ha podido observar, el proceso de despliegue se simplifica mucho, sobre todo en el aspecto operativo y se ha reducido considerablemente la probabilidad de que ocurran errores, ya que la intervención humana se minimiza y solo se interviene a la hora de validar el proceso. Por el contrario, el proceso ha adquirido cierta complejidad en el momento de aplicar el cambio ya que el servicio no se puede detener y hay que gestionar con mucho cuidado las transacciones distribuidas y su estado. Además, en algunos casos se necesita modificar el proceso de desarrollo de software para prevenir ciertas situaciones.

5 Diseño de la solución

En este capítulo se tratará el diseño para la migración a un servicio en *cloud* de la aplicación RoutingMaps y el diseño de la solución propuesta para la gestión del despliegue de la primera versión del servicio y sus posteriores actualizaciones.

5.1 Diseño de la arquitectura

El diseño de esta nueva arquitectura persigue un objetivo claro, soportar un gran volumen de peticiones. El cambio de tipo de aplicación lo hace un requisito necesario, ya que con las aplicaciones de tipo escritorio todos los componentes se ejecutan en la misma máquina para un usuario, por tanto, es más difícil llegar a saturar la máquina del usuario. En cambio, en las aplicaciones web todos los usuarios ejecutan la aplicación en el mismo entorno. Por este motivo hay que preparar dicho entorno para soportar cargas variables, sin que la calidad del servicio disminuya o el servicio se interrumpa. Otro punto importante es el uso eficiente de los recursos, para ello, los recursos del sistema se deberán modular dependiendo de la carga recibida.

A continuación, se muestra una ilustración donde se puede ver, de manera general, los componentes que constituyen la arquitectura de la plataforma. En la parte inferior se representa el *hardware*, que serán las máquinas que forman parte de los centros de procesamiento de datos. La siguiente área que se puede visualizar es el *cluster* de Kubernetes, el cual está formado por las máquinas virtuales proporcionadas por el proveedor *cloud* y en cada una ellas se instalará el software de Kubernetes necesario para crear el *cluster*. En la parte superior, se representan los distintos componentes de la plataforma en forma de contenedores Docker y las interacciones con sistemas externos.

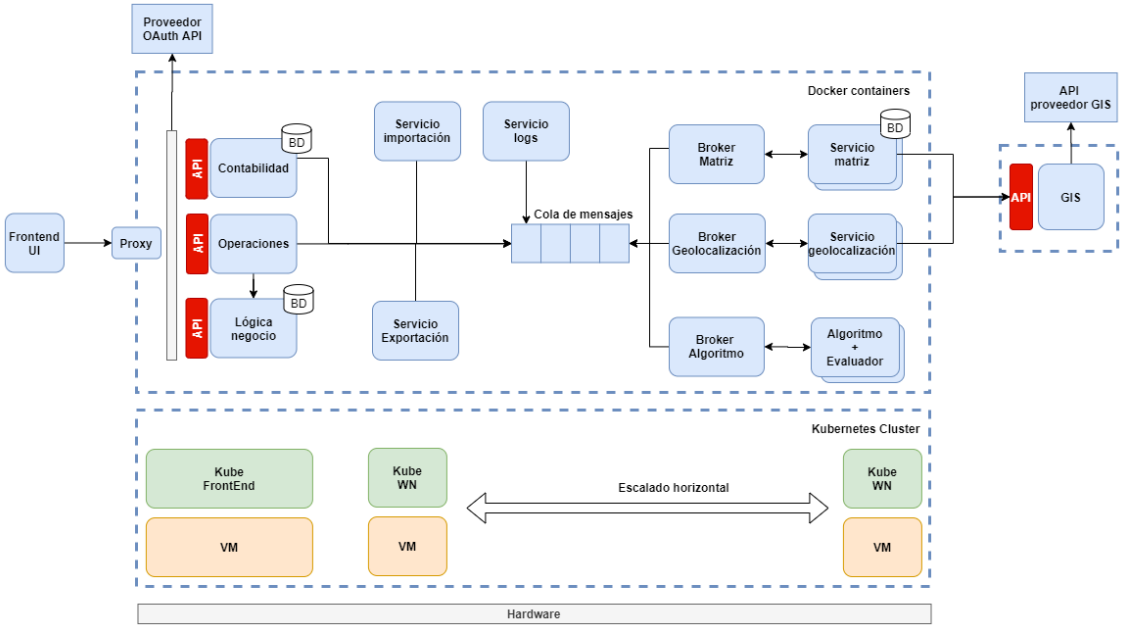

Ilustración 120 - Diseño arquitectura

Si nos centramos en la comunicación entre componentes, se puede observar que la mayoría se comunican únicamente con el gestor de colas de mensajes. Esto se debe a la

aplicación de patrones de diseño como *Domain event* [16]*, Event sourcing* [17] y *Saga*, en los que los componentes reaccionan a determinados eventos o mensajes. A continuación, se detallan cada uno de los componentes:

- **Lógica de negocio:** contiene la lógica de negocio de la aplicación y utiliza el código heredado de la capa de negocio de la aplicación anterior. Realiza operaciones como, por ejemplo, almacenar u obtener los puntos de entrega.
- **Operaciones*:*** se encarga de gestionar las operaciones que requieren la coordinación de varios componentes, generando y administrando los flujos de mensajes. Por ejemplo, la operación de optimización requiere enviar un mensaje para el cálculo de la matriz de distancias y, una vez recibido el resultado, enviar el problema al componente del algoritmo. Las peticiones que se realicen desde la aplicación web al componente de operaciones se resolverán de forma asíncrona, ya que algunas de las operaciones requieren de tiempos de resolución del orden de minutos. Por tanto, una vez haya recibido la petición HTTP y se haya validado su información, se devolverá un código HTTP 200 indicando que la petición se ha recibido correctamente. El resultado de la operación se devolverá a la aplicación web haciendo uso de la librería SignalR.
- **Servicio de importación*:*** este servicio se encarga de recibir los archivos CSV con la información de los puntos de entrega y los pedidos, y extraer la información.
- **Servicio de exportación*:*** este servicio se encarga de recibir la solución obtenida por el algoritmo y transformarla en un archivo CSV.
- **Servicio de matriz*:*** el servicio de matriz se ocupa de la gestión del cálculo de distancias. Realiza llamadas al servicio de GIS de Google para obtener la distancia entre dos puntos y almacena dicha información en una base de datos para utilizarla en futuras peticiones.
- **Servicio de geolocalización*:*** el servicio de geolocalización se ocupa de obtener la latitud y longitud de una dirección dada. Para ello realiza llamadas al servicio GIS de Google.
- **Algoritmo + Evaluador*:*** el servicio de cálculo de algoritmo y evaluación se encarga de obtener una solución a un problema de optimización de rutas dado. Además, se ocupa de evaluar las modificaciones de las soluciones realizadas.
- ***Broker* (Matriz, Geolocalización, Algoritmo)*:*** el servicio de *broker* para los servicios de matriz, geolocalización o algoritmo, ya se ha detallado en puntos anteriores. Se ocupa de recibir las peticiones y redireccionarlas a las instancias *worker* disponibles.
- ***Proxy inverso:*** el *proxy* inverso se encarga de redireccionar las peticiones a los servicios internos correspondientes
- **Servicio de logs*:*** el servicio de *logs* se encarga de registrar la actividad de la plataforma y almacenarla en una base de datos.
- **GIS*:*** el servicio de GIS interactúa con el API proporcionado por Google. Además, gestiona los usuarios y la licencia de uso de estos.
- **Contabilidad*:*** este servicio se encarga de registrar las transacciones contables para su posterior facturación. Este elemento no se podrá detallar en este trabajo académico.
- ***Frontend UI:*** este componente contiene la aplicación web que se descarga en el navegador del usuario.

5.1.1 Operaciones

En este punto se detallan las operaciones más importantes de la aplicación web y cómo interactúa cada componente de la arquitectura para llevarlas a cabo.

Importación de puntos de entrega

En esta operación el usuario da de alta en la aplicación web los puntos de entrega utilizando un archivo en formato CSV. Las acciones realizadas por cada componente son las siguientes:

1. El usuario selecciona el archivo en su sistema de ficheros y lo envía.
2. El componente Operaciones recibe el archivo, genera un mensaje con el contenido y lo envía a la cola de mensajes.
3. El Servicio de importación recibe el mensaje y lo procesa.
4. Una vez extraída la información del CSV, se genera un mensaje con el resultado y se envía a la cola.
5. El componente Operaciones recibe el mensaje.
6. Se envía el resultado al componente Lógica de negocio para procesar y almacenar la información de los puntos de entrega.
7. El componente Operaciones devuelve el resultado a la aplicación web. A continuación, el usuario será redirigido al siguiente paso.

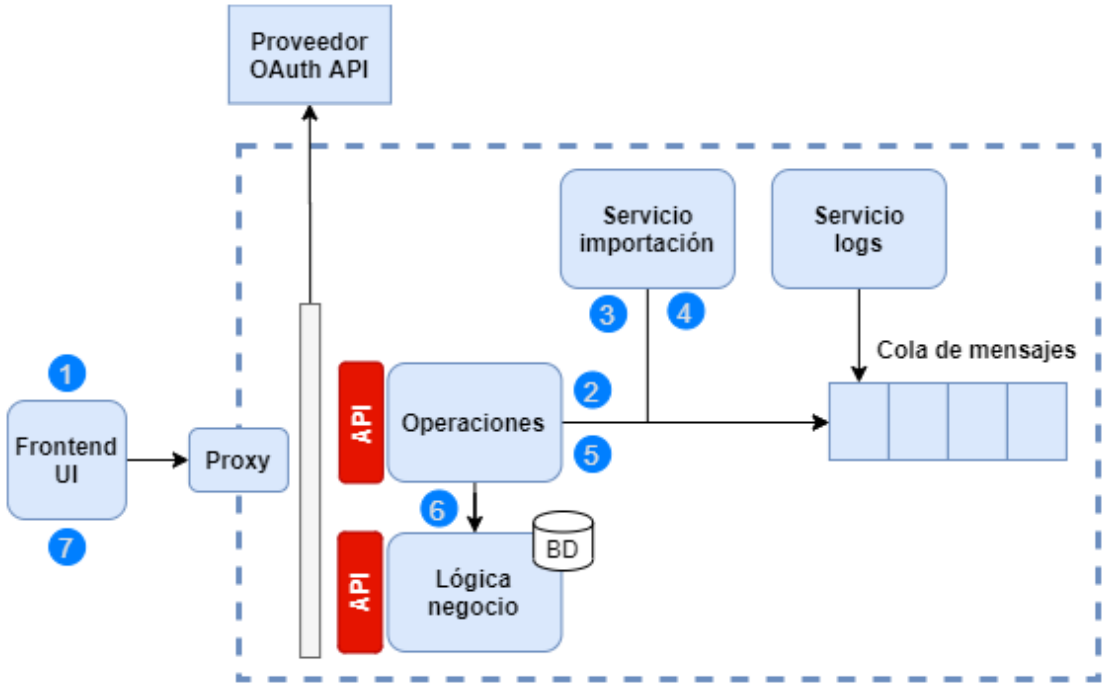

Ilustración 19 - Operación importar puntos de entrega

Geolocalización

La operación de geolocalización consiste en obtener la latitud y longitud de cada una de las direcciones de entrega proporcionadas por el usuario. Las acciones realizadas por cada componente son las siguientes:

1. El usuario selecciona el punto que desea geolocalizar e inicia la acción.

2. El componente de Operaciones recibe la petición y genera un mensaje para consultar el estado contable del usuario.
3. El componente de contabilidad recibe la petición, realiza las comprobaciones necesarias y responde a la petición generando un mensaje y enviándolo.
4. El componente de Operaciones recibe la respuesta. En caso afirmativo, genera un mensaje para solicitar la geolocalización.
5. El *broker* de geolocalización recibe la petición y la añade a su cola interna. Si hay *workers* disponibles se redirecciona la petición.
6. El servicio de geolocalización genera la petición al GIS y espera la respuesta.
7. El servicio de GIS recibe la petición, comprueba que el usuario es válido y no ha consumido su licencia y realiza la petición a Google.
8. La respuesta del API de Google se devuelve al servicio de geolocalización, a continuación, al bróker y para finalizar llega a Operaciones.
9. El resultado se devuelve al usuario. La lista de direcciones de entrega se actualiza con la información de las respectivas latitudes y longitudes, y los puntos de mapa se repintan para actualizar su posición.

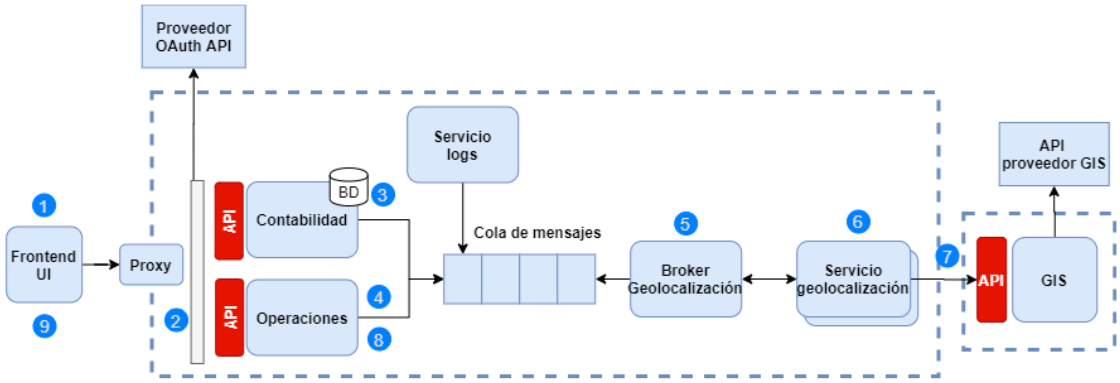

Ilustración 20 - Operación de geolocalización

Resolución del problema y optimización

Después de geolocalizar cada punto de entrega e introducir la información de los vehículos disponibles el usuario puede proceder a la resolución del problema y obtención de una solución. Las acciones realizadas por cada componente son las siguientes:

1. El usuario inicia la acción de resolución desde la aplicación web.
2. El componente de Operaciones recibe la petición y genera un mensaje para consultar el estado contable del usuario.
3. El componente de contabilidad recibe la petición, realiza las comprobaciones necesarias y responde a la petición generando un mensaje y enviándolo.
4. El componente de Operaciones recibe la respuesta. En caso afirmativo, genera un mensaje para solicitar el cálculo de la matriz de distancias de las localizaciones implicadas en el problema.
5. El *broker* de Matriz recibe la petición y la añade a su cola interna. Si hay *workers* disponibles se redirecciona la petición.
6. El Servicio de matriz solicita el cálculo de distancias al GIS. Al recibir la respuesta, se guarda el resultado obtenido en su base de datos para futuras peticiones y se resuelve la petición. El *broker*, a continuación, envía un mensaje a la cola con el resultado.

7. El componente de Operaciones recibe la información de la matriz de distancias y la añade a la información del problema. A continuación, se genera un mensaje para el cálculo del algoritmo.
8. El *broker* de Algoritmo recibe la petición y la añade a su cola interna. Si hay *workers* disponibles se redirecciona la petición.
9. El algoritmo realiza el cálculo necesario para la resolución del problema y lo devuelve al *broker*. El *broker* a su vez genera un mensaje con el resultado y lo envía a la cola.
10. El componente de Operaciones recibe el resultado y lo envía a la aplicación web.
11. La aplicación web recibe el resultado y lo muestra al usuario ilustrando en un mapa las rutas obtenidas.

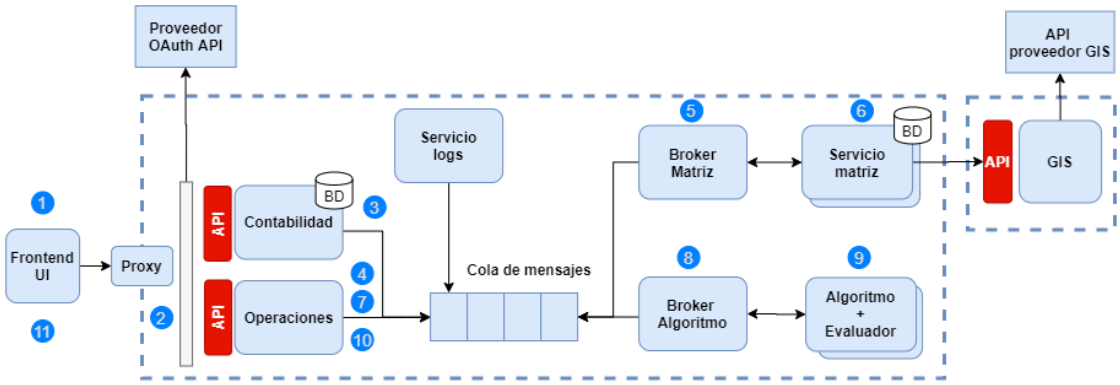

Ilustración 21 - Operación de resolución del problema y optimización

Exportación de rutas

Para finalizar el proceso el usuario tiene la opción de exportar la solución obtenida en un archivo con formato CSV donde se incluyen las rutas a realizar por los vehículos. Las acciones realizadas por cada componente son las siguientes:

1. El usuario inicia la acción desde la aplicación web.
2. El componente de Operaciones obtiene la información de la solución obtenida, genera un mensaje y lo envía a la cola.
3. El Servicio de exportación recibe la información, genera el archivo y envía el mensaje de respuesta a la cola.
4. El componente de Operaciones recibe el resultado y lo reenvía a la aplicación web.
5. La aplicación web recibe el resultado e inicia el proceso de descarga del archivo.

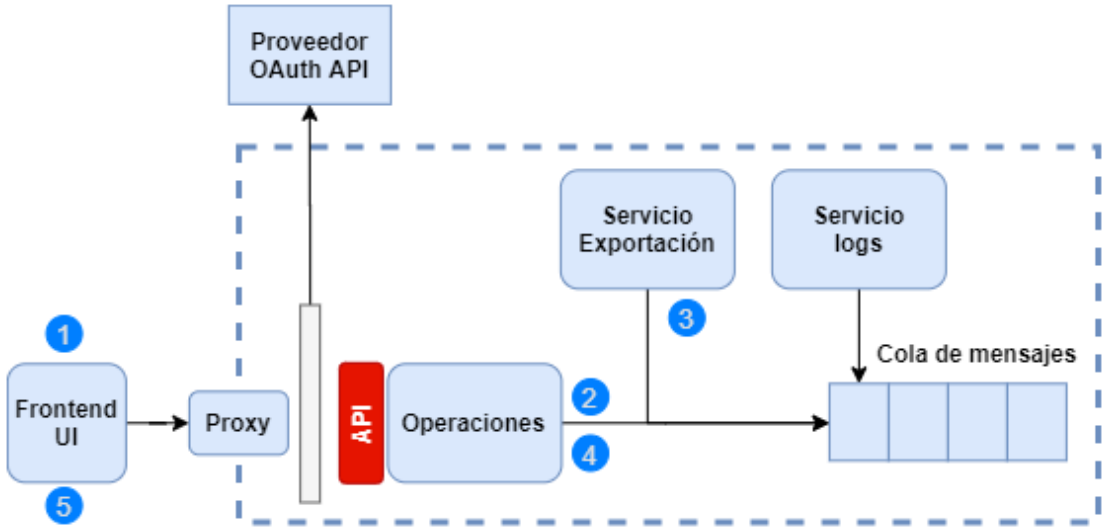

Ilustración 22 - Operación de exportación de rutas

5.1.2 Problemas de algunos componentes al actualizar

Cualquiera de los componentes descritos en este punto puede provocar errores si se actualizan durante el transcurso de una operación, pero el componente de Operaciones requiere de una mayor atención en este aspecto. Como ya se ha comentado, éste se ocupa de orquestar las transacciones distribuidas. El hecho de actualizarlo durante el transcurso de una transacción puede suponer un problema complejo, ya que se debe gestionar con mucho cuidado el estado de la transacción en curso y las versiones de los mensajes implicados.

El componente está preparado para no almacenar en memoria principal el estado de la transacción mientras se está llevando a cabo en otro componente. Esto facilita el reintento de la transacción. Además, el *framework* MassTransit realiza la gestión del versionado de mensajes. Aunque se disponga de estos medios, se podrían dar situaciones complejas de manejar. Por ejemplo, el componente de Operaciones envía un mensaje a otro componente para realizar un paso de la transacción general y se almacena el estado en un soporte secundario. A continuación, se realiza una actualización de los componentes implicados. Al obtener la respuesta, el componente de Operaciones deberá recuperar el estado general y tratar el mensaje, ambos posiblemente de la versión anterior. Estos casos se deben contemplar en la fase de desarrollo para que no ocurran problemas de inconsistencia de datos o un fallo completo del componente.

5.2 Diseño de la solución para el despliegue

Para diseñar el despliegue de un servicio en *cloud* sin tiempo de inactividad y con la mínima probabilidad de error, se deben adaptar la mayoría de las fases de desarrollo de software. El software se debe preparar para que sea compatible entre versiones, siempre que sea posible y debe ser observable para poder monitorizarse en tiempo de ejecución. Además, deben existir test unitarios y de integración para aportar la mayor cobertura posible al código desarrollado. El proceso de compilación de software y despliegue debe ser automático e igual en todos los entornos disponibles. También debe existir un plan de despliegue y reversión donde se especifique qué componentes se modifican, a qué datos podría afectar y cómo recuperar el estado del servicio en caso de error. En los siguientes puntos se describirá con detalle el diseño propuesto y las técnicas que se deberán utilizar para llevarlo a cabo.

5.2.1 Requisitos de un despliegue

Para que un despliegue cumpla con los requisitos de tiempo de inactividad cero y minimización de errores se deben aplicar técnicas denominadas de *Continuous Delivery* [18]. A continuación, se describen algunas de las técnicas y recomendaciones más importantes:

- Creación de plan de despliegue. Este debe incluir:
    - Los pasos que se deben realizar en el primer despliegue.
    - Cómo ejecutar las pruebas de humo para verificar el despliegue.
    - Pasos para anular el despliegue en caso de error.
    - Pasos necesarios para realizar la copia de seguridad previa y restaurarla.
    - Debe especificar dónde encontrar los registros de actividad.
    - Debe proporcionar los métodos de monitoreo de la aplicación.
    - Un registro de los problemas de despliegue de versiones anteriores y sus soluciones.
- El despliegue debe ser igual en todos los entornos y automático, utilizando herramientas de integración continua. Si es difícil replicar el entorno de producción, los demás entornos deben ser una simulación lo más parecida posible a la realidad.
- Se debe disponer de pruebas de humo [19] (también conocidas como “Build Verification Testing) para ejecutar la funcionalidad básica y verificar que el despliegue se ha realizado de forma correcta. Los datos generados por estas pruebas se deben eliminar al finalizar el proceso.
- El personal de operaciones debe disponer de las herramientas necesarias para monitorizar el servicio y comprobar su estado en todo momento.
- El personal de operaciones debe poder revertir el despliegue realizado y volver a la versión anterior siguiendo el plan de despliegue.
- Todos los componentes del servicio se deben promover en conjunto para evitar problemas de incompatibilidad de versiones. Además, hay que tener en cuenta la compatibilidad de componentes y servicios de terceros.
- Debe existir un plan de reversión que tenga en cuenta el estado de la base de datos y los posibles archivos generados por el usuario. Además, este plan se debe haber probado y verificado.

- La corrección de errores críticos o graves se debe desplegar de forma automática, como si fuera una funcionalidad nueva más. Esto evita que se cometan más errores en ciertas situaciones.
- Hay que desacoplar el despliegue del servicio de la modificación del esquema de base de datos. Esto se consigue desarrollando versiones del servicio compatible con la versión anterior y posterior de la base de datos.
- Se recomienda que los despliegues se realicen de forma continua y con la menor frecuencia posible. Esto hace que los cambios en el servicio sean menores y minimiza el riesgo de error.

Compatibilidad entre versiones

Como ya se ha comentado, la compatibilidad entre versiones es fundamental para realizar el despliegue de una nueva versión con éxito. Hay que tener en cuenta los cambios en el esquema de base de datos y en las interfaces o contratos software. El uso de esta técnica evita la mayoría de los problemas que surgen para mantener el estado del servicio en el momento del despliegue de una nueva versión o en el caso de reversión de cambios. Se reduce la complejidad de los procesos de reversión y se evita una posible pérdida de datos.

Para permitir la compatibilidad entre versiones se utilizará la técnica *ParallelChange*, la cual se ha detallado en el capítulo de análisis. Además, las versiones del servicio se implementarán de forma que sean compatibles con el esquema de base de datos actual y con el nuevo. En la siguiente versión del servicio se eliminará la compatibilidad con la versión previa de base de datos. A continuación, se muestra una ilustración con la dinámica a seguir en cada nueva versión.

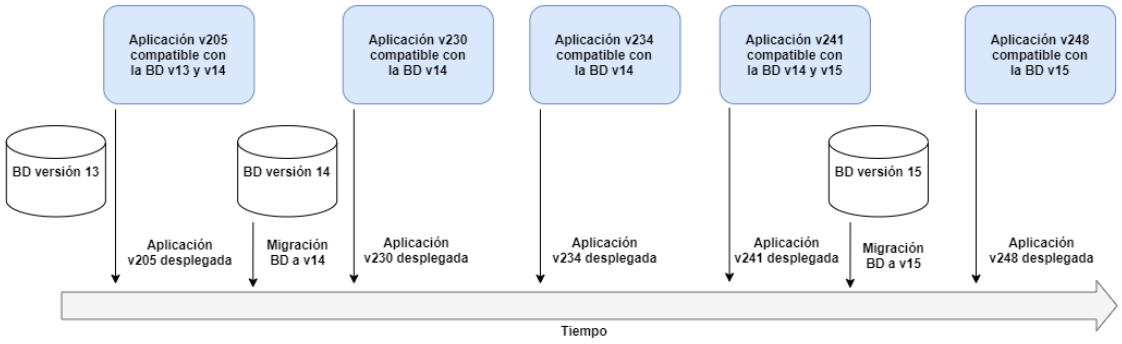

Ilustración 21 - Compatibilidad de versiones

Software observable

El software que se desarrolle debe ser observable, esto significa que en cualquier momento el personal de operaciones debe poder monitorizar el estado del servicio y la actividad que ha habido en un momento del pasado o en el actual. Para ello, el software que se desarrolle debe poder escribir las acciones que se suceden en un registro de actividad o *log* para que quede constancia de lo ocurrido.

También es posible observar el estado de salud de los microservicios a nivel de proceso o contenedor Docker. En la actualidad la mayoría de tecnología, como .Net Core, permiten consultar el estado del proceso o el uso de recursos a través de una URL en el

API REST (/health). Esta URL puede ser usada por el orquestador de contenedores para tomar decisiones, como el reinicio del contenedor.

Pruebas

El uso de pruebas automáticas es fundamental en el desarrollo de software de calidad. Se pueden implementar en distintos ámbitos, pero la más útil para el contexto de la fase de despliegue es la prueba de humo. Habitualmente se utiliza con API REST. Consiste en probar la funcionalidad básica del servicio llamando desde la aplicación *frontend* al API REST, simulando alguna de las funcionalidades disponibles. Si alguna de las pruebas realizadas modifica el estado del servicio, se deben añadir los pasos para eliminar la información generada. Además de verificar la funcionalidad básica, sirve para verificar que las configuraciones son las correctas y que la aplicación *frontend* está interactuando con el entorno correcto.

5.2.2 Solución propuesta

Para cumplir con los requisitos que se acaban de detallar en el punto anterior se propone la siguiente solución. Se propone implementar como técnica de despliegue *Canary Release*, esta se selecciona debido a que en caso de error el problema afecta a un número menor de usuarios del servicio y el entorno de la nueva versión puede ser de menor tamaño (tener menos recursos) debido a que tiene un tráfico menor, esto supone un ahorro en los costes de recursos. Además, se va a utilizar esta técnica de forma diferente. Las peticiones entrantes se van a redireccionar a los dos entornos, de esta forma no hay que aplicar complejos procesos de recuperación de datos en caso de error y reversión.

Para solucionar los problemas del despliegue y monitorización, se proponer utilizar las herramientas de terceros Istio, Prometheus y Grafana. Haciendo uso de Istio se puede aplicar la técnica *CanaryRelease* ya que está preparado para redirigir un porcentaje de peticiones según las necesidades del servicio. Además, permite redirigir el tráfico en espejo a distintas versiones del servicio dentro de un *cluster* Kubernetes. Istio dispone de módulos integrados para Prometheus y Grafana, gracias a estas herramientas se puede monitorizar el estado de cada Pod y visualizase en paneles de control por parte del personal de operaciones.

Para implementar las pruebas de humo se propone disponer de una URL en la aplicación *frontend*, por ejemplo “https://dominio.es/test”, donde se ejecuten las operaciones básicas (importar, geolocalizar, optimizar y exportar) sin intervención del usuario. Una vez iniciado el servicio, se realizará una petición a dicha URL para ejecutar las pruebas y verificar el proceso. Después de finalizar todas las operaciones básicas, se comprobará en el registro de actividad o *log* si existe algún error. Si no existe, esta fase del despliegue se puede dar por válida.

6 Implementación

En este capítulo se detallarán los pasos necesarios para configurar el entorno de ejecución e implantar la integración y despliegue continuo en el proyecto existente. Además, se explicará el uso de la herramienta Azure DevOps para la gestión de todo el proceso de desarrollo y entrega de software.

6.1 Entornos y configuraciones necesarias

Actualmente el servicio RoutingMaps se encuentra desplegado en los servidores gestionados por la empresa. Esto se debe a que algunas partes del proyecto aún se encuentran en fase de desarrollo y de momento solo tiene acceso al servicio un grupo reducido de usuarios. Se han realizado pruebas de despliegue en proveedores *cloud*, pero para esta fase del proyecto se utilizará un *cluster* Kubernetes simulado.

6.1.1 Cluster Kubernetes simulado

Actualmente existen tres máquinas en el departamento que simulan un cluster Kubernetes estático, aunque en el futuro deberá cambiar. Dichas máquinas se han configurado para que una de ellas haga las funciones de nodo *master* y las otras dos de nodos de trabajo. A continuación, se muestra una imagen de la organización de servidores y de la organización de componentes. En uno de los nodos de trabajo se ejecutan los componentes de operaciones y en otro los de cálculo algorítmico.

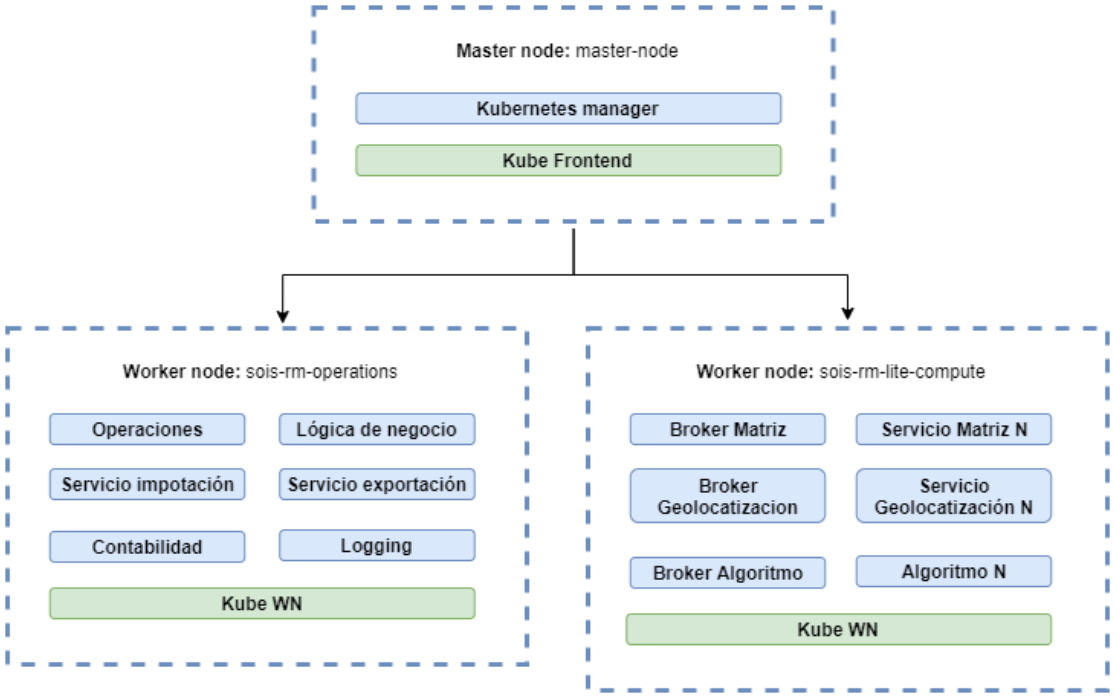

Ilustración 22 - Cluster simulado

Microk8s

Para llevar a cabo la configuración del *cluster* se ha utilizado la herramienta Microk8s [40], la cual permite instalar los componentes necesarios de Kubernetes y sus extensiones de forma rápida y sencilla. A continuación, se muestran los comandos necesarios para su instalación en el nodo *master*:

```
# Se actualizan los paquetes de Ubuntu y se instala el gestor de paquetes snap
sudo apt update
sudo apt install snapd

# Se instala Microk8s usando el gestor de paquetes
sudo snap install microk8s --classic

# Se verifica la instalación y se comprueba el estado de los módulos de microk8s
microk8s.status

# Se añaden los comandos como alias, de esta forma se puede trabajar más rápido
# Ejemplo: "ku get pods"
snap alias microk8s.kubectl kubectl
snap alias microk8s.kubectl ku

# Se guarda la configuración
sudo microk8s.kubectl config view --raw > $HOME/.kube/config

# Se habilitan los módulos de microk8s para la gestión de DNS, volúmenes de almacenamiento y
permisos
sudo microk8s.enable dns storage rbac

# Se habilita un rango de puertos para poder usarlos a partir del puerto 80
# Se edita el archivo
vi /var/snap/microk8s/current/args/kube-apiserver
# Se añadir en la última línea del archivo
--service-node-port-range=80-60000

# Se reinicia el servicio para que se apliquen los cambios
sudo systemctl restart snap.microk8s.daemon-apiserver
```

En este punto ya se dispone de un *cluster* Kubernetes de un nodo. Microk8s permite añadir nodos al *cluster* de forma fácil. Para ello el nodo *master* genera un token que se

[40] https://microk8s.io/docs

utilizará para autenticar en la operación de adición. El nodo *worker* deberá tener instalado microk8s previamente. A continuación, se muestran los comandos necesarios:

```
# Comando para obtener desde el nodo master el token de unión de nodos worker
microk8s add-node
# Resultado
        Join node with: microk8s join [IP]:25000/UqEOFUVUdpeCGSZaWiJiqwxWkzBoDkUq

# Se ejecuta en el nodo worker
microk8s join [IP]:25000/UqEOFUVUdpeCGSZaWiJiqwxWkzBoDkUq
```

Si se necesitara eliminar nodos del *cluster,* se deberían seguir los siguientes pasos:

```
# Listar los nodos disponibles
kubectl get nodes

# Eliminar un nodo desde el nodo master
microk8s remove-node [nombre nodo]

# Ejecutar en el nodo worker
microk8s leave
```

Asignación de Pods a nodos del *cluster*

En ocasiones puede ser necesario que ciertos Pods se ejecuten en nodos del *cluster* con unas características determinadas, por ejemplo, con mayor número de CPUs o memoria RAM. Para ello se pueden añadir *labels* a los nodos del *cluster* e indicar en la declaración de los Pod que se ejecutarán en los nodos que coincidan con el *label*. A continuación, se muestran los comandos y declaraciones necesarias:

```
# Se añaden labels a los nodos
kubectl label nodes master-node rm.topology.kubernetes.io/nodetype=admin
kubectl label nodes sois-rm-operations rm.topology.kubernetes.io/nodetype=operations
kubectl label nodes sois-rm-compute rm.topology.kubernetes.io/nodetype=compute

# Selectores a utilizar en la declaración de Pods
nodeSelector:
        rm.topology.kubernetes.io/nodetype: admin

nodeSelector:
        rm.topology.kubernetes.io/nodetype: operations

nodeSelector:
        rm.topology.kubernetes.io/nodetype: compute
```

Docker registry

Para poder utilizar las imágenes Docker del proyecto, se deben añadir como secreto de Kubernetes las credenciales para el Docker Registry y utilizarse en todos los Pods que se creen. Para ello se deberá ejecutar el siguiente comando:

```
kubectl create secret docker-registry regcred --docker-server=[DNS]/rm/docker-images --docker-username= --docker-password= --docker-email=
```

6.1.2 Istio

Como ya se ha comentado en capítulos anteriores Istio es un producto que ayuda a gestionar y monitorizar una malla de servicios. En este punto se aborda como implantar Istio en el proyecto existente. En primer lugar, se describe como instalar el producto en el *cluster* Kubernetes y posteriormente como se ha tenido que modificar el proyecto para permitir el redireccionamiento de peticiones.

Instalación

El proceso de instalación se puede realizar de dos formas, ambas son igual de sencillas. La primera opción es añadir Istio como un módulo más de la herramienta Microk8s y la segunda es descargar directamente el software desde el sitio oficial de Istio y ejecutarlo. En este caso se ha optado por la segunda opción debido a que se tiene más control y libertad sobre la instalación para modificarla si se requiere. Se puede modificar la instalación editando los archivos de declaración de Kubernetes y aplicando los cambios. A continuación, se muestran los pasos para realizar la instalación:

```
# Se descarga y descomprime el software en la carpeta actual

curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.6.1 sh -


# Se accede a la carpeta que se acaba de crear y se añaden los archivos binarios a la variable PATH

cd istio-1.6.1

export PATH=$PWD/bin:$PATH


# Se aplican los archivos de declaración Kubernetes para iniciar los servicios de Istio

istioctl manifest apply --set profile=default
```

Unos segundos después de ejecutar estos comandos estarán todos los servicios disponibles. Istio instala adicionalmente los servicios necesarios para utilizar las herramientas Prometheus y Grafana.

Para poder realizar las tareas de gestión y monitorización Istio necesita añadir en todos los Pods del proyecto un contenedor adicional que implementa el patrón *Sidecar* [20]. Para ello se ejecuta un comando que indica a Istio que debe inyectar el contenedor adicional en todos los Pods para un espacio de nombres. El comando es el siguiente:

```
kubectl label namespace default istio-injection=enabled
```

Si se obtiene un listado de los Pods con el comando “kubectl get pods”, se podrá observar que todos los Pods para el espacio de nombres por defecto constan de dos contenedores.

Adaptación del proyecto

Una vez preparado el entorno de ejecución se ha necesitado realizar ciertas adaptaciones al proyecto para poder utilizar Istio según los problemas que iban surgiendo en la implantación. Además, se ha necesitado realizar bastantes pruebas de concepto hasta poder comprender el funcionamiento del producto y adaptarlo a las necesidades del proyecto.

Las primeras pruebas de concepto se realizaron generando dos versiones del componente Frontend donde se podía distinguir con claridad al acceder al sitio web si era una versión u otra. En este punto ya se detectaron problemas que obligaron a modificar los archivos de declaración de Kubernetes. Para iniciar dos versiones del mismo servicio Kubernetes con el mismo nombre, por ejemplo, “frontend-service” se deben crear distintos Deployment y añadirles etiquetas que ayuden a distinguir la versión y que Istio sepa a qué versión debe redireccionar cada petición según su URL. Este cambio se podía aplicar para un servicio en el cual el acceso es gestionado por Istio, pero existen otros servicios como Operaciones o Lógica de negocio (existen otros) en los cuales no se puede distinguir la versión a la que van dirigidas las peticiones ya que no siempre es requerido pasar por el proxy inverso. Este problema ocurre con todas las comunicaciones entre componentes dentro del *cluster*. Esto ha obligado a modificar los archivos de declaración para incluir el identificador de versión como sufijo en el nombre de cada objeto Kubernetes, por ejemplo, “frontend-[versión]”. De esta forma al utilizar los DNS proporcionados por Kubernetes en las distintas interacciones entre componentes, las direcciones se gestionan correctamente. Para abordar este problema se ha probado a utilizar espacios de nombres distintos por versión, pero no se ha logrado hacer que funcione el redireccionamiento de Istio. En un futuro se continuará intentando, ya que la solución actual requiere que los archivos de declaración se modifiquen en cada versión.

Otro de los problemas encontrados es el direccionamiento de las peticiones dentro de los contenedores, tanto del *frontend* como del *backend*. Esto se debe a que el direccionamiento de los recursos dentro del contenedor no es el mismo que el indicado en la URL de la petición. Por ejemplo, en el caso del componente Frontend al utilizar la URL “https://[dominio]/[versión]/front” su URI correspondiente sería “/[versión]/front”, esta URI se utilizará para localizar los archivos HTML y Javascript dentro del contenedor, pero al crear la imagen Docker los archivos se depositan en un directorio genérico sin saber a qué versión pertenece la imagen, esto hace que al usar una URI que depende de la versión los recursos no se encuentren. Para solucionar este problema, se ha modificado el arranque del contenedor para que se obtenga como parámetro la URI que se va a utilizar y los recursos se copien al directorio correspondiente. En el caso de los contenedores para las APIs del *backend* el problema es distinto. Por ejemplo, al realizar una petición al API de Operaciones se utiliza la URL “https://[dominio]/operations/[versión]/api/[controlador]” su URI correspondiente sería “/[versión]/api/[controlador]”, pero ese recurso no existe dentro del API; el recurso correcto sería “/api/[controlador]”. La solución a este problema la proporciona Istio, el cual permite realizar una reescritura de la URI para eliminar el prefijo “/[versión]/…”.

Después de solucionar los problemas que acabamos de comentar la implantación de Istio ha sido relativamente fácil, ya que la configuración para el redireccionamiento entre versiones usando porcentajes es muy intuitiva y el sitio oficial de Istio dispone de bastantes ejemplos para todas las opciones.

6.2 Azure DevOps

Azure DevOps [41] es una herramienta proporcionada por Microsoft que cubre la gestión de todo el ciclo de desarrollo de software y permite la implementación de metodologías ágiles. Posee muchas utilidades para la toma de requisitos, la gestión de historias de usuario y tareas, gestión de recursos y equipos, documentación, gestión de repositorios de código, ejecución de pruebas y automatización de compilaciones y despliegues. Se puede utilizar *online* o instalarse en los servidores administrados por una organización.

Para este proyecto concreto la empresa ha utilizado la herramienta desde el inicio, aunque solo para la gestión del proyecto, gestión del código y en algunos casos para la compilación y análisis del código implementado. Gracias a este trabajo académico se han podido completar las fases restantes, las cuales se detallan en los siguientes puntos.

La herramienta Azure DevOps necesita disponer de un grupo de agentes o máquinas donde se ejecuten los trabajos. En este caso todas las operaciones se ejecutarán en el nodo *master* del *cluster* Kubernetes, aunque en el futuro se deberá utilizar una máquina solo para este fin.

6.3 Integración continua

El proceso de integración continua implica distintas fases del proceso de desarrollo de software. Estas fases son: la implementación del código y su adición al repositorio, la compilación del proyecto, la ejecución de las pruebas automáticas y la generación de los artefactos resultantes del proyecto. Para iniciar una fase todas las anteriores deben finalizarse sin errores. A continuación, se muestra un diagrama para ilustrar el proceso.

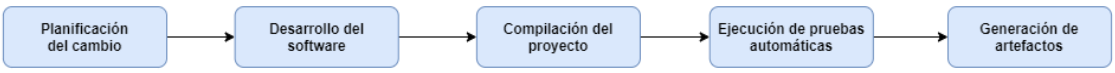

Ilustración 23 - Fases integración continua

Respecto a la fase de generación de artefactos, para la mayoría de los componentes del proyecto el artefacto de salida consiste en una imagen de contenedor Docker. Por este motivo esta fase incluirá la compilación de las imágenes y su posterior subida al registro de imágenes Docker correspondiente.

Azure DevOps dispone de la sección “Pipelines - Compilaciones” donde se crea la secuencia de pasos para automatizar la generación de artefactos e implementar la integración continua. A continuación, se detallan los pasos necesarios. Todos ellos se inician al añadir un nuevo fragmento de código en el repositorio por parte de los desarrolladores. En primer lugar, se detallarán los pasos necesarios para el componente Frontend y, posteriormente, los pasos necesarios para todos los componentes del *backend*, esto se debe a que se utilizan tecnologías distintas (NodeJs y .NET Core) y por tanto el proceso de compilación es distinto.

[41] https://docs.microsoft.com/es-es/azure/?product=featured

Frontend

1. Se instalan las dependencias del proyecto con el comando “npm install”.
2. Se compila el proyecto con el comando “npm build”.
3. Se realiza el análisis de calidad de código con la herramienta SonarQube y se publican los resultados.
4. Se construye la imagen Docker haciendo uso de un archivo docker-compose-ci.yml donde ya se ha indicado previamente el nombre o identificador de la imagen.
5. Se sube la imagen Docker al registro utilizando el mismo archivo .yml del paso anterior.

Componentes backend (Operaciones, Lógica de negocio, Broker, etc.)

Para cada componente de la parte *backend* del proyecto se realizan los siguientes pasos

1. Se descargan las dependencias del proyecto.
2. Se compila el proyecto.
3. Se ejecutan las pruebas automáticas.
4. Se construye la imagen Docker haciendo uso de un archivo docker-compose-ci.yml donde ya se ha indicado previamente el nombre o identificado de la imagen. En este caso, este archivo es de más utilidad ya que en él se declaran muchos componentes.
5. Se sube la imagen Docker al registro utilizando el mismo archivo .yml del paso anterior.
6. Se copian los archivos de declaración .yml de Kubernetes en el servidor los cuales se utilizarán en la fase de despliegue. Estos archivos se deben preparar para cada versión, ya que pueden cambiar, por ejemplo, para añadir un nuevo parámetro a un componente o añadir nuevos archivos de declaración.

Después de realizar todas estas acciones, el proyecto estaría listo para desplegar todos los componentes en un *cluster* Kubernetes, ya que se dispone de imágenes Docker para todos ellos y archivos de declaración .yml para iniciar los Pods correspondientes.

6.4 Despliegue continuo

El proceso de despliegue implica la creación de una nueva versión del servicio y es un proceso semiautomático ya que requiere la intervención del personal de operaciones para avanzar entre las distintas fases. Esto se debe a que la mayoría de las veces se requiere realizar un proceso de validación del servicio en algunas fases.

Azure DevOps dispone de la sección “Pipelines - Versiones” donde se crea la secuencia de pasos necesaria para desplegar una nueva versión en el entorno de pruebas o producción. A continuación, se describen las fases en las que está dividido un despliegue:

1. **Despliegue de los servicios en el *cluster* Kubernetes.** Para ello se debe crear en el servidor de gestión del *cluster* Kubernetes un nuevo directorio nombrado con el identificador de versión y copiar los archivos de declaración .yml. Después se ejecuta el comando “kubectl create -f [nombre-componen]-service.yml” para crear los objetos Kubernetes (Pods, *Deployments* o *Services*) necesarios para crear cada componente de la plataforma. Para mayor comodidad todos los comandos se añaden en un archivo *Bash* para ejecutarlos todos de una

vez. Después de unos segundos todos los Pods de la plataforma estarán arrancados y listos para funcionar. En este punto la nueva versión de la web se podrá encontrar en la URL "https://[dominio]/[versión]/login" y la versión previa continuará encontrándose en la URL "https://[dominio]/login". De esta forma las dos versiones pueden coexistir y se puede validar el despliegue realizado sin interrumpir el servicio de la versión actual.

2. **Ejecución de las pruebas de humo.** Se ha creado una nueva sección en la aplicación web en la URL "https://[dominio]/[versión]/smoke-test" para ejecutar de forma automática las operaciones de importación, geolocalización, optimización, evaluación y exportación. En este punto se llama a dicha URL para ejecutar las pruebas y posteriormente se revisan los registros de actividad en busca de errores.
3. **Aplicar la técnica Canary Release.** Una vez comprobado el despliegue se puede aplicar la técnica Canary Release, para ello se modifica el redireccionamiento que realiza Istio. Se deshabilita la URL "https://[dominio]/[versión]/login" y se indica que ambas versiones están disponibles en URL "https://[dominio]/login". Además, se indica que un determinado porcentaje de peticiones se redirigirán a la versión anterior y otro porcentaje a la nueva. Inicialmente la nueva versión recibirá un porcentaje muy reducido de peticiones y se irá incrementando gradualmente si no se producen errores.
4. **Validación del despliegue.** Una vez alcanzado el nivel de confianza suficiente en la nueva versión, se puede dar por válido el despliegue y modificar el *router* para que la nueva versión reciba todas las peticiones.
5. **Detener los servicios Kubernetes de la versión anterior.** La validación del despliegue implica solo un cambio en el enrutado de peticiones, pero los servicios Kubernetes de la versión anterior continúan activos. Para finalizar el proceso se deben detener todos los servicios y liberar los recursos. Este paso puede parecer el más sencillo de la secuencia, pero tiene ciertas implicaciones a nivel de disponibilidad, ya que es difícil saber en qué punto la nueva versión es completamente válida y ya no necesitamos el entorno anterior. En caso de tener que volver a la versión anterior, se necesitarían unos segundos para arrancar todos los componentes. Algunos componentes como la base de datos o el gestor de colas de mensajes podrían tardar casi un minuto debido a que tienen proceso de arranque costoso.

Al iniciar el proceso de despliegue se deben indicar como parámetros los identificadores de la nueva versión y la anterior.

Todos los pasos se realizan ejecutando *scripts* de Linux previamente preparados, estos se almacenan en un directorio del repositorio de código para que se puedan utilizar desde la herramienta.

Para cada paso descrito existe un paso de reversión. Por ejemplo, si se acaba de validar el despliegue se podría volver a la configuración de Canary Release y de la misma forma con todos los pasos hasta revertir cualquier cambio realizado.

6.5 Monitorización

Como ya se ha comentado en anteriores puntos del documento una vez desplegado el servicio y validado, se necesita conocer el estado de los servicios y si las operaciones se están ejecutando con normalidad. Para esta tarea se utilizan las herramientas Prometheus y Grafana. Prometheus se ocupa de recolectar las métricas de uso del servicio, como tiempos de respuesta a las peticiones HTTP o el uso de recursos de cada componente. Toda la información recolectada se visualiza de forma intuitiva en los paneles de control proporcionados por Grafana.

6.5.1 Prometheus

Como ya se ha comentado anteriormente Prometheus se añade como un servicio Kubernetes más al instalar Istio y recibe todas las métricas recolectadas por los contenedores *sidecar* que Istio inyecta en cada Pod iniciado. Esto hace que el trabajo de recolección de datos sea muy rápido de implantar en el proyecto ya que el desarrollador no tiene que modificar nada en el proyecto existente. Basta con acceder a la interfaz web de Prometheus e introducir en su buscador el prefijo "istio…" y se obtienen todo tipo de métricas. Esto es de gran ayuda a la hora de iniciar las tareas de monitorización, pero dado en gran número de resultados que aparecen se plantea la duda sobre "¿Qué se debe monitorizar?" y que va a ser de más utilidad para el proyecto. Sobre todo, para tomar decisiones de escalado, análisis de futuras optimizaciones o reversión de una versión desplegada.

A continuación, se describen algunas de las métricas más útiles:

- **istio_requests_total:** permite saber el número de peticiones HTTP totales en un intervalo de tiempo para todos los servicios en los que Istio ha añadido el contenedor *sidecar*.
- **istio_request_duration_milliseconds:** indica la duración de cada petición HTTP para las peticiones gestionadas por Istio.
- **istio_request_bytes:** permite consultar el tamaño en *bytes* de las peticiones HTTP.
- **istio_response_bytes:** permite consultar el tamaño en *bytes* de las respuestas HTTP.

Prometheus tiene su propio lenguaje de consulta PromQL (*Prometheus Query Language*) el cual permite filtrar los resultados de las métricas descritas por el nombre de componente que las ha generado, realizar sumatorios o calcular ratios, entre otras muchas operaciones.

Además de las métricas proporcionadas por Istio, existen otras que resultan interesantes a la hora de tomar decisiones de escalado. Una de las más importantes es la del estado de las colas de trabajos en los componentes de cálculo algorítmico o de la matriz de distancias. Dependiendo de esta información se pueden tomar decisiones de escalado y esto afecta directamente a la calidad del servicio proporcionado al usuario final. En estos momentos los componentes de tipo *broker* no están preparados para que se pueda consultar dicha información desde el exterior, aunque sí que se guarda en los registros de actividad. En próximos desarrollos se abordará la posibilidad de enviar esta información a Prometheus haciendo uso del cliente .NET del que dispone.

6.5.2 Grafana

Grafana es otra de las herramientas que se incluye en la instalación de Istio y se configura por defecto con una gran variedad de paneles de control proporcionados por Istio. En ellos se puede visualizar, por ejemplo, el estado de salud de componentes del proyecto o el uso de CPU y RAM en un momento determinado. También es posible añadir paneles preconfigurados para otros productos como Kubernetes o RabbitMQ.

En Grafana es posible configurar distintos orígenes de datos, en esta instalación estaba por defecto Prometheus y se ha añadido la base de datos SQL que se encarga de almacenar el registro de actividad del proyecto. Esto nos ha permitido crear paneles de control personalizados que utilizan consultas SQL definidas por el equipo de desarrollo.

Grafana tiene la posibilidad de configurar alertas y notificaciones. Esta es una funcionalidad muy útil ya que permite avisar a los administradores del servicio cuando el servicio se detiene, ocurren errores o se produce un incremento anormal del uso de recursos, lo cual puede ser indicador de un ciberataque. Añadir una alerta es muy sencillo, basta con dirigirse a un panel determinado y acceder a la sección de alertas, en él se deberá indicar una cota superior o inferior para iniciar el proceso de alerta y un intervalo de tiempo (en segundos, minutos u horas) desde que se sobrepasa la cota hasta que finalmente se envía la notificación. Para este proyecto, de momento, se han configurado alertas cuando se supera el 80% de uso de recursos de CPU y RAM del *cluster* Kubernetes y cuando se registra algún error en el registro de actividad.

A continuación, se muestra algunos de los paneles configurados:

Uso de recursos

En la siguiente imagen se muestra el uso de recursos del conjunto de máquinas que forman el *cluster* Kubernetes. Este panel se ha añadido desde el buscador de paneles preconfigurados. En la parte superior se puede ver un conjunto de gráficos que muestra por colores el porcentaje de recursos utilizados, de izquierda a derecha se muestra la RAM, la CPU y el almacenamiento en disco. A continuación, se muestra para cada recurso la cantidad utilizada y la cantidad total. En la parte inferior se muestra el tráfico de entrada y salida de red para un instante de tiempo.

Ilustración 24 - Panel uso de recursos de cluster

Registro de actividad

Se ha creado un panel para visualizar la actividad registrada por cada componente del proyecto. Para ello se ha añadido como origen una base de datos de tipo SQL Server donde se almacenan los eventos.

La tabla de registro consta de los siguientes campos:

- **Fecha:** en este campo se almacena la fecha y hora en la que ocurre un evento con precisión de milisegundos.
- **Nivel:** este campo indica el nivel de gravedad del evento ocurrido. Los valores posibles son “Fatal”, “Error”, “Warn”, “Info” y “Debug”.
- **App:** se corresponde con la aplicación o componente que crea el registro.
- **Método:** en este campo se informa el método o clases del código donde se crea el registro.
- **Parámetros:** consiste en un campo de texto donde se guardan los parámetros de un método. En caso de ser un objeto, se serializa.
- **Mensaje:** es un campo de texto libre donde el desarrollador puede indicar lo que está ocurriendo en el código.
- **Usuario:** en este campo se almacena el token JWT del usuario de la aplicación a la que pertenece la operación en curso.
- **Excepción:** en caso de error en este campo se almacena toda la información que proporciona la excepción lanzada por el sistema.

En la siguiente imagen se muestra el panel creado, donde se puede observar una gráfica de líneas que representa el sumario de registros de actividad de cada nivel en intervalos de tiempo. Además, cada nivel se muestra con un color según su nivel de gravedad, rojo para “Error” o naranja para “Warn”. En la parte inferior se muestra una tabla con un subconjunto de las columnas del registro para aportar información al usuario sobre el evento ocurrido.

Ilustración 25 - Panel de registro de actividad

Actividad por componente

En la siguiente imagen se muestra uno de los paneles preconfigurados para Istio donde se puede ver la actividad por componente del proyecto, en este caso se muestra el componente de Operaciones. Aunque hay algunos paneles que indican que no hay datos, se puede ver, por ejemplo, el porcentaje de peticiones que no han acabado en error de HTTP 500, el volumen de peticiones de clientes o la duración de peticiones de cliente en un intervalo de tiempo.

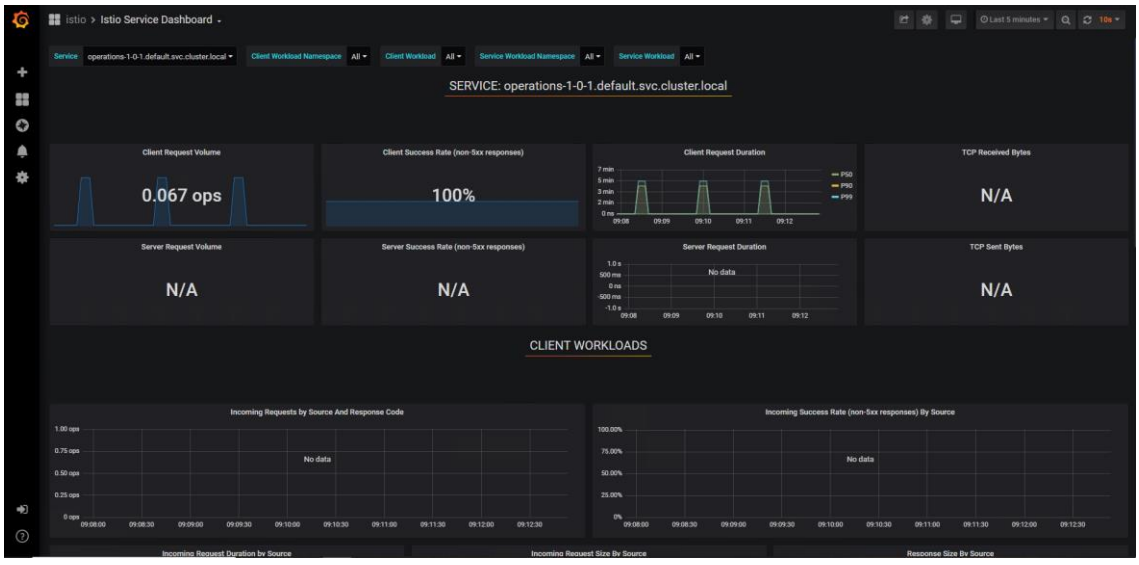

Ilustración 26 - Panel de actividad por componente

7 Pruebas realizadas

En este capítulo se describirán las pruebas realizadas para validar las implementaciones y cambios realizados en el proyecto existente.

7.1 Puntos débiles y críticos

Para probar las soluciones implementadas como el despliegue de una nueva versión de forma automática y el redireccionamiento de un porcentaje determinado de peticiones entre versiones, se ha planteado el siguiente escenario. Existía una versión del servicio en funcionamiento y un grupo de usuarios utilizándolo. A continuación, se ha desplegado en el mismo *cluster* Kubernetes una nueva versión del servicio. Durante el proceso de despliegue varios usuarios más han intentado acceder al servicio. Al finalizar las pruebas, ninguno de los usuarios reportó ningún problema funcional, comportamientos inesperados o incoherencias en los datos. Además, se ha podido validar que en unas ocasiones el usuario accedía a una versión o a otra.

Los resultados de las pruebas se deben en gran parte a que la aplicación web es de tipo SPA y por tanto el navegador web no se recarga en cada cambio de pantalla. Esto hace que sea difícil que el usuario coincida justo en el momento del despliegue en el cual se está realizando el cambio en el *router*. Durante las pruebas al finalizar el desarrollo, sí que se ha llegado a encontrar un breve lapso de segundos en los que ninguna de las versiones está disponible. Aunque eso sucede en un porcentaje muy bajo de las ocasiones. Esto se debe al tiempo que se necesita en aplicar los cambios del enrutado de Istio.

Respecto al uso de recursos, se ha podido observar que la calidad del servicio empeoraba en cuanto al tiempo de respuesta de las peticiones o los tiempos necesarios para realizar las operaciones de cálculo. Esto se debe a que se está utilizando un *cluster* estático y los recursos se tienen que dividir entre dos versiones. Además, las versiones no comparten ningún componente, lo cual hace que se tengan que iniciar varias instancias de componentes que tienen un elevado consumo de recursos, como el gestor de colas o la base de datos.

7.2 Tiempo necesario para el despliegue

El despliegue de una nueva versión del servicio, aunque se realice de forma automática, requiere invertir tiempo en cada una de sus fases. A continuación, se describen las actividades necesarias y el coste estimado de tiempo, sin contar el tiempo necesario para realizar los cambios en el software:

- **Preparar los archivos de declaración de Kubernetes**: al finalizar el desarrollo de una nueva versión se deben preparar los archivos de declaración de los objetos Kubernetes. Dependiendo del cambio realizado podría significar incrementar simplemente los identificadores de versión en los archivos o añadir un nuevo parámetro en la configuración de algún componente, pero puede haber casos en los que se creen nuevos componentes o se modifique las comunicaciones entre ellos. En el caso de un cambio simple pueden suponer **30 minutos** aproximadamente. Si los archivos cambian de

forma sustancial y además de crearlos se tienen que validar puede suponer **5 horas**. Esta acción la realizará una persona con perfil de desarrollo.

- **Arranque de los servicios de una nueva versión:** cuando el cambio software ya está preparado para ejecutarse, los equipos de desarrollo y operaciones deben decidir cuándo iniciar el proceso de despliegue. Para iniciar el proceso la persona encargada de esta tarea debe dirigirse a la herramienta Azure DevOps, localizar el servicio en el apartado de versiones e iniciar la fase de arranque introduciendo los parámetros para los identificadores de versión anterior y nueva. Al iniciar la acción los servicios estarán disponibles en unos minutos. Todo el proceso puede suponer **5 minutos**.
- **Pruebas de humo:** esta acción requiere iniciar la prueba y validar en el registro de actividad que los resultados son correctos. En el caso peor puede suponer **30 minutos**.
- **Validación manual de la funcionalidad:** después de ejecutar las pruebas de humo el equipo de calidad del software debe comprobar manualmente que todas las pantallas y componentes del servicio funcionan como es debido. Dependiendo del cambio realizado puede suponer de **5** a **15 horas**.

Las demás acciones necesarias para el despliegue no se han incluido debido a su bajo coste en tiempo.

8 Conclusión

Como se ha podido observar migrar una aplicación de escritorio a un servicio en el *cloud* es una tarea muy compleja y es un cambio de paradigma completo. Esta tarea requiere de conocimientos técnicos avanzados y cierta experiencia en el campo de la computación distribuida. El cambio también requiere formar al equipo existente, lo cual puede tener un coste elevado en tiempo y dinero. Es cierto que la mayoría de las plataformas y entornos *cloud* tienden a facilitar el trabajo al desarrollador, pero esto no exime de la necesidad de tener una base sólida en la materia.

Respecto al proceso de despliegue de nuevas versiones, hoy en día sigue sin existir un procedimiento 100% seguro. Solo existen herramientas para minimizar el riesgo mediante la automatización. Cabe destacar que dicho proceso de automatización lo debe de implementar una persona y, por tanto, se pueden producir errores de implementación. Por ello, además de automatizar el proceso, se debe probar adecuadamente cada fase y operación del proceso completo.

Otro punto difícil de abordar es la validación del despliegue y en qué punto podemos desechar la versión anterior. Además, una vez validado el despliegue y pasado un tiempo es muy difícil volver a una versión anterior, el proceso de reversión se debe preparar adecuadamente para evitar la pérdida de datos.

Respecto al estado del proyecto, como puntos pendientes más importantes cabe destacar el despliegue en un proveedor *cloud* y probar el proyecto con un volumen de usuarios grande. Para ambos puntos habrá que esperar a que algunas funcionalidades estén finalizadas.

En cuanto a las dificultades ocurridas en el proyecto, además de las dificultades técnicas por tratar con herramientas nuevas, destacaría la escasez de técnicas y conocimientos para abordar el problema de despliegue de nuevas versiones. Es la fase del proceso de entrega de software de la que menos información se encuentra si la comparamos con otras como, por ejemplo, la fase de desarrollo.

8.1 Asignaturas del máster

Para poder realizar este trabajo académico ha sido necesario adquirir ciertos conocimientos previos. A continuación, se describe qué asignaturas de Máster han ayudado a aportar dichos conocimientos:

- **Cloud Computing:** en esta asignatura aprendí las propiedades mínimas de cualquier servicio desplegado en el *cloud*, como son la alta disponibilidad, el manejo de gran volumen de peticiones, la escalabilidad, la elasticidad, el acceso seguro o la tolerancia a fallos. También aprendí la importancia de la gestión del estado del servicio y las dificultades que conlleva. Todo ello me ha servido para realizar el diseño de la nueva arquitectura basada en microservicios.
- **Diseño de Aplicaciones y Servicios Escalables:** esta asignatura me ha servido para conocer los modelos de replicación existentes. Aunque no se ha implementado ninguno de ellos, las herramientas que se han utilizado sí los

utilizan y es importante conocer estos detalles a la hora de seleccionar una herramienta u otra para implantarla en el proyecto.

- **Infraestructuras de Cloud Público:** aunque el proyecto todavía no está ejecutándose en un proveedor *cloud*, sí se ha preparado para ello. Gracias a esta asignatura aprendí qué servicios y herramientas poseen los proveedores y cómo utilizarlas. En esta asignatura también se dieron algunas nociones sobre las estrategias existentes de despliegue, como Blue Green y Canary Release.
- **Plataformas de Gestión de Contenedores:** todos los componentes de este proyecto se ejecutan en contenedores Docker y estos se gestionan mediante Kubernetes. Las prácticas realizadas en esta asignatura han servido de gran ayuda para la implementación de esta parte del proyecto.
  Programación de Sistemas Cloud: en esta asignatura profundicé en mis conocimientos de ZMQ y en los distintos patrones de comunicación disponibles. Mucho de estos conocimientos se han utilizado para la implementación del componente *Broker*.

8.2 Competencias transversales

Conocimiento de problemas contemporáneos

El servicio RoutingMaps desplegado en un entorno *cloud* aportará grandes beneficios a la sociedad ya que su objetivo es la gestión eficiente de los recursos de una organización. Esta gestión eficiente se consigue por dos vías distintas.

En primer lugar, las empresas del sector logístico y de transportes utilizan este servicio para ahorrar costes en carburantes, al reducir el uso de carburantes también se reducen las emisiones de gases de efecto invernadero. Existen otros casos de uso, como el diseño de circuitos integrados, donde también se puede ahorrar en el consumo de materiales al producir dispositivos electrónicos de uso doméstico o industrial.

En segundo lugar, al ser un servicio disponible en *cloud* permite utilizar los recursos e infraestructuras de manera más eficiente ya que se comparte entre distintos usuarios y solo se incrementan cuando las necesidades de cómputo lo requieren. Otro beneficio de este tipo de servicios es que permite a las empresas de menor tamaño utilizarlo solo cuando lo necesitan sin necesidad de invertir grandes cantidades de recursos en infraestructuras. Esto hace que pueda utilizarse por un mayor número de organizaciones y, por tanto, se democratiza el uso del servicio.

Planificación y gestión del tiempo

El proyecto ha seguido la planificación realizada y el tiempo invertido se adecúa a lo estimado. Cabe destacar que la fase de implementación tenía una gran dependencia del estado del proyecto de migración general. No se podía desplegar el servicio sin que la funcionalidad básica estuviera implementada. Finalmente, la fase de implementación se ha podido realizar sin mayor problema.

Dada mi experiencia profesional y mi nivel académico la gestión del tiempo y tareas no ha supuesto ningún problema. Es cierto que se han tenido que abordar problemas nuevos para los que se requería haber realizado estudios de postgrado e investigar nuevas

tecnologías desconocidas hasta el momento. Todo ello no ha impedido desarrollar el proyecto con normalidad.

9 Referencias

[1] G. B. Dantzing and J. H. Ramser, “The Truck Dispatching Problem” in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, pp.15–64, 1964.

[2] Eva Alfaro Cid. Dept. Soluciones de optimización inteligentes. Instituto Tecnológico de Informática. 2020. “RoutingMaps | Problemas De Rutas. Taxonomía”. Camino de Vera, S/N, Edif. 8G - Acc. B - 4ª Planta Valencia – España. [online] Disponible en: https://routingmaps.com/problemas-de-rutas-taxonomia [Consultado el 02/09/2020].

[3] J. H. Holland. “Adaptation in Natural and Artificial Systems”. University of Michigan Press, Ann Arbor. 1975.

[4] B. Bullnheimer, R. F. Hartl and C. Strauss. “Applying the Ant System to the Vehicle Routing Problem”. Paper presented at 2nd International Conference on Metaheuristics, Sophia-Antipolis, France. 1997.

[5] G. Clarke and J. Wright “Scheduling of vehicles from a central depot to a number of delivery points”, Operations Research, 12 #4, 568-581, 1964.

[6] Fowler, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 17-24, 2003.

[7] Lorenz, Martin, Guenter Hesse and Jan-Peer Rudolph. “Object-relational Mapping Revised - A Guideline Review and Consolidation.” ICSOFT-EA (2016).

[8] Roy Thomas Fielding: “Architectural Styles and the Design of Network-based Software Architectures”, tesis doctoral, University of California, Irvine, EEUU, 2000.

[9] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," in IEEE Transactions on Communications, vol. 22, no. 5, pp. 637-648, May 1974, doi: 10.1109/TCOM.1974.1092259.

[10] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, Sam Toueg: Optimal Primary-Backup Protocols. 6th International Workshop on Distributed Algorithms and Graphs (WDAG), Haifa, Israel: 362-378 (1992).

[11] Richardson, C. “Microservices Patterns”. Shelter Island, New York: Manning Publications, pp.110-146, 2019.

[12] Sato, D., 2014. Canary Release. [online] martinfowler.com. Disponible en: https://martinfowler.com/bliki/CanaryRelease.html [Consultado el 02/09/2020].

[13] Fowler, M., 2010. Blue Green Deployment. [online] martinfowler.com. Disponible en: https://martinfowler.com/bliki/BlueGreenDeployment.html [Consultado el 02/09/2020].

[14] Sato, D., 2014. Parallel Change. [online] martinfowler.com. Disponible en: https://martinfowler.com/bliki/ParallelChange.html [Consultado el 02/09/2020].

[15] Sato, D., 2014. Branch by Abstraction. [online] martinfowler.com. Disponible en: https://martinfowler.com/bliki/BranchByAbstraction.html [Consultado el 02/09/2020].

[16] Richardson, C. “Microservices Patterns”. Shelter Island, New York: Manning Publications, pp.160-168, 2019.

[17] Richardson, C. “Microservices Patterns”. Shelter Island, New York: Manning Publications, pp.183-220, 2019.

[18] Humble, J. and Farley, D. “Continuous Delivery”. Upper Saddle River, NJ: Addison-Wesley, 2011.

[19] Dustin, E., Rashka, J. and Paul, J. “Automated Software Testing”. Reading, Mass. [u.a.]: Addison-Wesley, pp. 43-45, 1999.

[20] Richardson, C. “Microservices Patterns”. Shelter Island, New York: Manning Publications, pp.410, 2019.