Universidad de la República
Facultad de Ingeniería
Instituto de Computación

Integración de una Service Mesh a una Plataforma de Integración basada en Microservicios

Proyecto de Grado

Estudiantes
Victor Pons
Mauricio Morinelli
Emiliano Barcia

Tutor
Guzmán Llambías

Abril 2021

Resumen

Con el advenimiento cada vez mayor de aplicaciones en la nube basadas en el patrón de diseño de Microservicios, las tecnologías de contenerización y gestión de contenedores toman especial relevancia, impactando positivamente en los procesos de *deployment* de las diferentes aplicaciones.

Además de resolver el despliegue de una aplicación*,* es necesario continuar trabajando en tiempo de ejecución o *runtime,* siendo vital observar y controlar lo que sucede con las comunicaciones intra servicios, en especial en lo que refiere a su seguridad y confiabilidad.

En Mayo de 2017, Google, IBM y Lyft junto a otros lanzan Istio, una Service Mesh (SM) de código abierto. Una SM permite gestionar las necesidades a nivel de infraestructura de una aplicación distribuida, organizando, monitoreando, asegurando y recabando datos de las comunicaciones entre servicios.

El objetivo del presente trabajo es el de integrar una SM a una Plataforma de Integración (PI) basada en Microservicios. Una PI es un sistema informático especializado en brindar soporte para la creación de soluciones o flujos de mediación, basados en los *Enterprise Integration Patterns* (EIP), que permitan integrar sistemas heterogéneos. La PI sobre la que se desarrolla el proyecto fue implementada por E. Camejo y J. Bonhomme durante su proyecto de grado: “Plataforma de Integración basada en Microservicios” (Julio 2019). A su vez, dicho trabajo se basa en la tesis de Maestría de A. Nebel: “Arquitectura de Microservicios para Plataformas de Integración” (Octubre 2018).

Como objetivo secundario se plantea agregar un Orquestador de Contenedores, herramienta que permite escalar, gestionar y automatizar el despliegue de aplicaciones contenerizadas.

En procura de los objetivos establecidos, primero se confecciona un criterio de selección para evaluar y elegir entre las opciones de SM disponibles. Luego, motivado por la falta de material previo disponible, se confecciona y aplica una metodología de trabajo general para migrar una aplicación de microservicios contenerizados, a una que incluya una Service Mesh. La inclusión de la Service Mesh elegida simplifica significativamente la arquitectura de la solución, a la vez que incluye nuevas funcionalidades como Seguridad y Circuit Breaker entre otras.

Finalmente se desarrolla un nuevo escenario de ejecución que junto a los ya existentes permite corroborar el correcto funcionamiento de la PI + SM. Este nuevo escenario utiliza a su vez un nuevo componente de integración (*Router*), cuya implementación es parte del alcance del proyecto.

1 Introducción

Este capítulo introduce la temática abordada a lo largo del proyecto de grado, repasando brevemente el contexto sobre el que se desarrolla, los objetivos propuestos y los resultados obtenidos.

La sección 1.1 introduce tecnologías y conceptos claves para comprender el trabajo realizado, sobre los que luego se profundiza a lo largo del documento. Luego, en la sección 1.2 se detallan los objetivos del proyecto de grado. Por su parte la sección 1.3 enumera los aportes que resultan del desarrollo del proyecto. Finalmente la sección 1.4 da cuenta de la organización del documento y sus respectivas secciones.

1.1 Contexto

El desarrollo de aplicaciones distribuidas mediante el empleo del patrón de diseño Microservicios es una realidad en constante crecimiento de un tiempo a esta parte [1], favoreciendo la arquitectura de aplicaciones como colecciones de pequeños servicios independientes, creados en torno a funcionalidades de negocio y comunicados entre sí de forma ligera [2] [3].

La naturaleza distribuida y la necesidad de poder escalar dinámicamente para soportar eficientemente grandes flujos de usuarios hacia una aplicación, impulsaron la creación de herramientas que facilitan el despliegue de nuevos servicios bajo demanda, proceso evolutivo que derivó en lo que conocemos como Orquestadores de Contenedores.

La Figura 1.1 muestra las distintas oleadas evolutivas en el desarrollo de aplicaciones de Microservicios, donde se aprecia que la más reciente corresponde a la aparición de las Service Mesh [4].

Una Service Mesh es un software de infraestructura para aplicaciones de Microservicios: es una construcción que aglomera funcionalidades que anteriormente ofrecían librerías como Hystrix[1] de Netflix[2] o Finagle[3] de Twitter[4], entre otras.

Como tal, persigue el objetivo de abstraer de la lógica de cada microservicio todo aspecto relacionado a la comunicación entre ellos, al operar sobre su intercambio de mensajes conocido como tráfico Este-Oeste [38]. Así mismo recaen sobre la SM tareas como monitorear

[1] https://github.com/Netflix/Hystrix
[2] https://www.netflix.com/
[3] https://twitter.github.io/finagle/
[4] https://twitter.com/

la actividad de los servicios, aplicar políticas globales a la aplicación, ruteo, balanceo de cargas, descubrimiento de servicios, seguridad y autenticación, entre otras [5]. En definitiva, las SM abstraen de los microservicios la lógica de infraestructura e integración, permitiendo que estos se limiten a lógica de negocio.

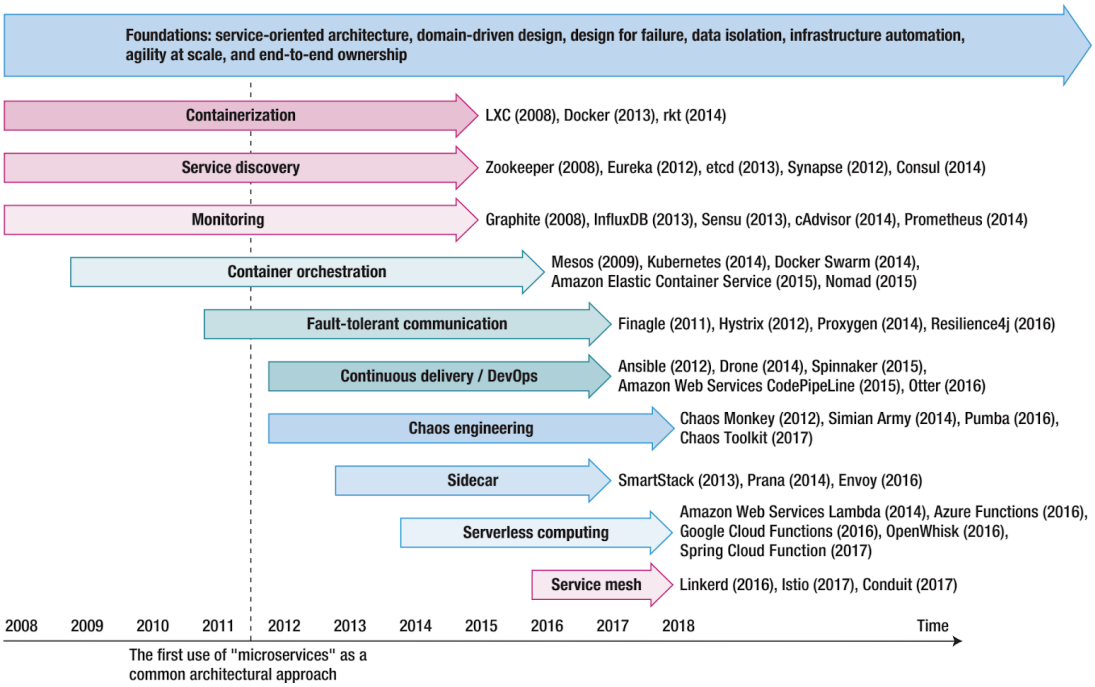

Figura 1.1: Timeline de la arquitectura de Microservicios [4].

Integrar una Service Mesh a una aplicación de Microservicios ya existente implica trasladar hacia ésta las funcionalidades de soporte que estuvieran implementadas a nivel de librerías especializadas o microservicios propios de la aplicación.

1.2 Objetivos

El objetivo principal del proyecto es poder integrar una Service Mesh a la Plataforma de Integración basada en Microservicios desarrollada como parte del proyecto de grado de Camejo y Bonhomme de Abril de 2019 [6]. Dicho proyecto toma a su vez como base de su trabajo, la tesis de Maestría de Nebel de Octubre de 2018 [7]. Nebel estudia y analiza la aplicabilidad del patrón de diseño de Microservicios a la construcción de una Plataforma de Integración que luego, Camejo y Bonhomme desarrollan en su proyecto de grado.

A los efectos de alcanzar el objetivo final, se plantean los siguientes objetivos parciales:

- Estudio de Microservicios, Service Mesh y Orquestadores de Contenedores.

- Relevar y analizar las principales opciones de Service Mesh disponibles. Confeccionar un criterio de selección que permita escoger la SM que mejor se adapte a las necesidades del proyecto.
- Definir y documentar tanto el proceso de integración de la SM como la posterior migración de funcionalidades.
- Agregar a la Plataforma de Integración basada en Microservicios la SM seleccionada, migrando hacia ésta todas aquellas funcionalidades que sea pertinente.
- Agregar un Orquestador de Contenedores a la PI que permita automatizar el despliegue de los contenedores de la misma.
- Agregar un nuevo escenario de ejecución a la suite de escenarios disponibles en la PI. Este escenario agregado, constituye una nueva validación de la solución global, a la vez que utiliza un nuevo componente de integración basado en los *Enterprise Integration Patterns* (*EIPs*) [10].
- Agregar un nuevo Microservicio reutilizable a la PI que desarrolle el patrón de diseño Router [9], parte de los anteriormente mencionados *EIPs*.

1.3 Aportes

A continuación los principales aportes del proyecto de grado:

- Relevamiento de las opciones de Service Mesh existentes en el mercado. Estudio pormenorizado de las 3 opciones identificadas como más populares al momento de su análisis. Selección de la Service Mesh más adecuada para integrar a la Plataforma de Integración.
- Integración de una Service Mesh (Istio) a la Plataforma de Integración basada en Microservicios.
- Integración de un Orquestador de Contenedores: Kubernetes.
- Implementación de un nuevo Microservicio reutilizable: *Router*. Éste microservicio implementa el patrón de diseño *Content-Based Router* que en resúmen permite rutear hacia uno u otro receptor, basado en el contenido del mensaje recibido.

1.4 Organización del documento

El resto del documento se organiza de la siguiente manera:

- En el Capítulo 2 se resume el marco conceptual sobre el que se desarrolla el presente trabajo.

- En el Capítulo 3 se desarrolla el análisis de la base de trabajo sobre la que se construye el proyecto de grado, los requerimientos a cumplir y el alcance final.

- En el Capítulo 4 se describe el trabajo de selección de la SM a integrar, así como el proceso de migración de funcionalidades desde los microservicios de la PI a la SM.

- En el Capítulo 5 se presenta el diseño para soportar el nuevo escenario de integración, incluyendo el nuevo componente de integración requerido.

- En el Capítulo 6 se profundiza sobre la solución propuesta a nivel de Istio y la configuración de cada una de las funcionalidades que provee. También se aborda el desarrollo de los nuevos componentes de integración que fue necesario implementar.

- Luego, en el Capítulo 7 se brindan detalles acerca de la gestión del proyecto.

- Finalmente, en el Capítulo 8 se presentan las conclusiones del trabajo realizado y se brinda un desglose de posibles mejoras y trabajos a futuro.

2 Marco conceptual

Se desarrolla a continuación el marco conceptual que abarca los conocimientos básicos necesarios para entender el trabajo realizado a lo largo del proyecto de grado.

La sección 2.1 plantea el concepto de Microservicios como patrón de diseño de aplicaciones distribuidas, junto con las funcionalidades que habitualmente requiere este tipo de aplicaciones. A continuación en la sección 2.2 se presenta el concepto de Service Mesh, su definición dentro del contexto de trabajo actual y su división en *Data Plane* y *Control Plane*. Luego, la sección 2.3 analiza las funcionalidades que habitualmente presenta una Service Mesh. La sección 2.4 introduce el concepto de Contenedores y la relación de los mismos con Microservicios. También describe a los Orquestadores de Contenedores e introduce Kubernetes. La sección 2.5 da cuenta de la integración entre un Orquestador de Contenedores y Service Mesh. Posteriormente en la sección 2.6 se define la noción de Plataforma de Integración. Finalmente la sección 2.7 da cuenta de otros proyectos relacionados con la temática del corriente proyecto de grado.

2.1 Microservicios

El patrón de diseño de Microservicios estructura una aplicación como un conjunto de servicios ligeramente acoplados, desarrollados cada uno en torno a una funcionalidad de negocio específica [2]. Es una alternativa al diseño más tradicional de desarrollo monolítico, en tiempos de aplicaciones distribuidas en la nube que necesitan escalar rápidamente. Las aplicaciones monolíticas con el paso del tiempo tienden a transformarse en un sistema demasiado grande y complejo de mantener. Cuando las concesiones a nivel técnico hacen que el mantenimiento de la misma exceda los límites razonables de esfuerzo se dice que la aplicación se fosilizó [22].

Al diseñar una aplicación siguiendo la arquitectura de Microservicios cada servicio debe ejecutar en un proceso propio, pudiendo ser desplegado bajo demanda de forma independiente [3]. Es de destacar también que el desarrollo de los distintos microservicios no necesita estar ligado a una decisión de lenguaje: cada equipo de trabajo puede optar por un lenguaje distinto ya sea por su especialización o por razones técnicas (por ejemplo, se suele recurrir a C++ cuando la *performance* es crucial o a Golang[5] por su manejo de concurrencia).

En lo que refiere a las comunicaciones entre sus partes, Microservicios sugiere la implementación de componentes inteligentes y comunicaciones tontas (simples) [2], lo cual coincide con la idea de mantener a los servicios conectados ligeramente.

[5] https://golang.org/

En la figura 2.1 se puede apreciar cómo resuelven la necesidad de escalar ambas arquitecturas. Por su parte, la tabla 2.1 profundiza el análisis comparativo entre Microservicios y Monolíticos, que fundamentan la tendencia migratoria hacia Microservicios [3] [38]

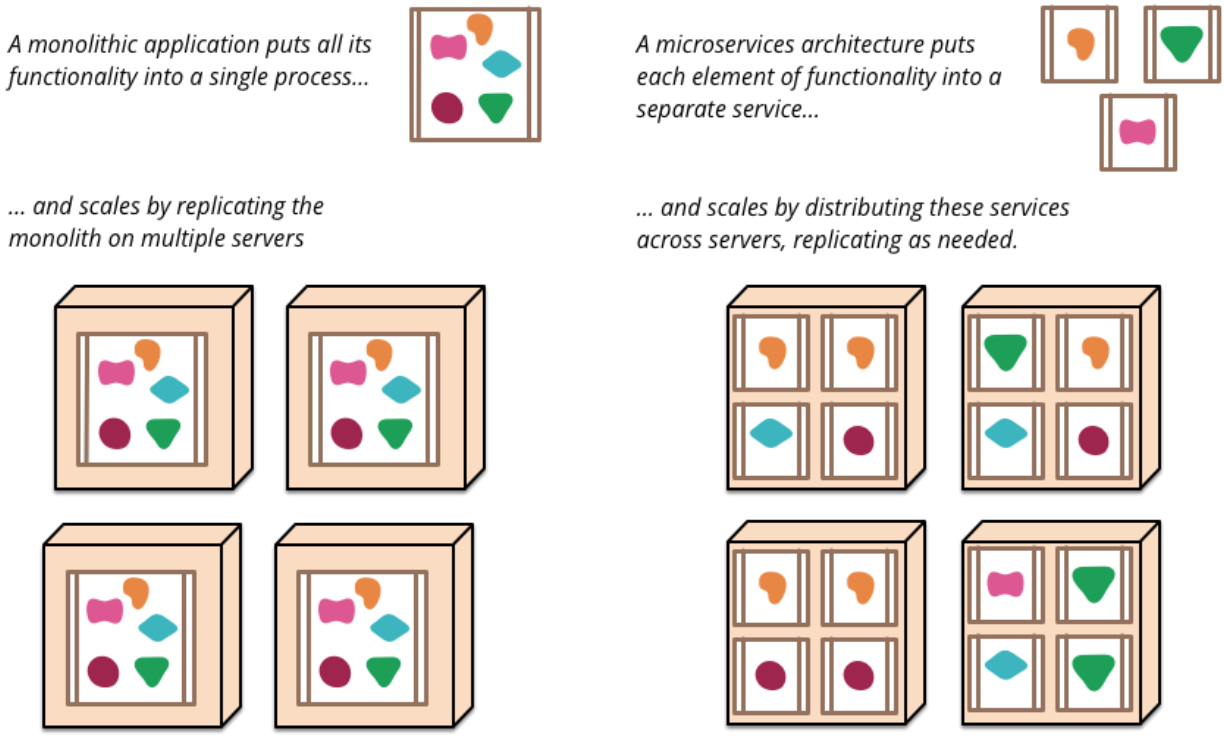

Figura 2.1: Comparación entre la forma en que escalan aplicaciones Monolíticas y de Microservicios [6]

Los aspectos donde Microservicios se destaca por sobre las aplicaciones Monolíticas no son lo único a tener en cuenta para optar por una u otra arquitectura: Microservicios acarrea ciertas complejidades que deben ser consideradas y evaluadas pertinentemente.

Al abandonar una aplicación Monolítica hacia una de Microservicios, lo que antes sucedía a nivel de una unidad involucra ahora varios servicios distribuidos. Esto introduce demoras debido a las llamadas remotas entre servicios, además de posibles fallas en las mismas. Los servicios deben ser capaces de reponerse ante estas fallas, además de lidiar con las demoras y posibles inconsistencias de datos. En particular, es deseable que las aplicaciones basadas en microservicios sean capaces de evitar la propagación de un error originado en un servicio hacia el resto.

[6] https://martinfowler.com/articles/microservices/images/sketch.png

Característica	Monolítico	Microservicios
Impacto de los cambios	Cualquier cambio, por pequeño que sea, implica tener que desplegar nuevamente el monolítico.	Pequeños cambios usualmente impactan en un número reducido de microservicios, que pueden ser desplegados de forma independiente.
	Testear cambios en el código resulta complejo.	Los microservicios son pequeños: testear cambios resulta más ágil.
Mantenimiento	Al aumentar de tamaño, la complejidad del sistema aumenta considerablemente: su mantenimiento resulta costoso.	Aunque aumente de tamaño, los microservicios seguirán siendo de tamaño reducido, manejable por pequeños equipos especializados.
	La curva de aprendizaje para nuevos desarrolladores crece constantemente en relación al tamaño del código.	Al trabajar en un servicio particular, es posible analizar al resto como cajas negras; el código a estudiar por nuevos desarrolladores es reducido.
Escalabilidad	Si la aplicación necesita escalar debe escalar como un todo, sin importar la necesidad particular.	Es posible escalar con mayor granularidad, ya que se escala a nivel de microservicios.
Tecnología	El sistema utiliza (en general) un solo lenguaje de programación para todas sus funciones.	Es posible desarrollar aplicaciones políglotas, donde cada microservicio se desarrolla con el lenguaje que resulte más conveniente.
	Las tecnologías y herramientas utilizadas en el desarrollo inicial quedan ancladas en la aplicación; habitualmente resulta costoso cambiarlas.	Reescribir o migrar un microservicio a una nueva tecnología tiene un impacto reducido en relación al tamaño del mismo.
	Actualizar una versión de alguna librería utilizada resulta laborioso, por lo que usualmente se posterga dicho proceso: la aplicación demora en aprovechar de las novedades introducidas.	Actualizaciones del stack tecnológico impactan sobre los microservicios individualmente, su proceso de actualización puede suceder en etapas independientes para cada uno.

Tabla 2.1: Análisis comparativo entre aplicaciones basadas en Microservicios y Monolíticas.

Otra consecuencia inmediata al pasar de una arquitectura Monolítica hacia Microservicios es tener que resolver la coordinación entre partes. Existen dos enfoques empleados

habitualmente: uno donde la lógica de negocio se centraliza (Orquestación) y otro independiente o distribuído (Coreografía) [23].

Para el caso de Orquestación como se puede apreciar en la figura 2.2, un microservicio cumple el rol de organizar el flujo de ejecución, siendo quién se comunica e intercambia información con cada uno de los otros servicios involucrados. Por otra parte, en Coreografía no existe el rol de organizador del flujo de ejecución, siendo cada servicio quien realiza su tarea y notifica en un canal determinado a quienes estén suscritos.

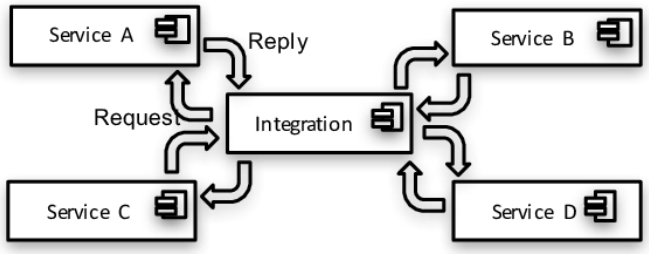

Figura 2.2: En Orquestación, existe un microservicio que centraliza el flujo de ejecución y coordina la colaboración de las partes[7].

Aplicar Coreografía resulta en una implementación más fiel del patrón de diseño de Microservicios en el sentido de la independencia entre servicios [23]. Como contrapartida al no contar con un orquestador, el flujo de ejecución aplicando Coreografía no está centralizado, como se observa en la figura 2.3.

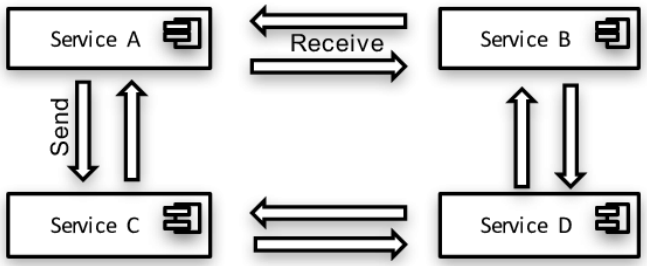

Figura 2.3: En Coreografía se resuelve la coordinación entre partes de forma distribuida, resultando en una implementación más fiel del patrón de diseño de Microservicios[8].

Ante un error en un sistema distribuido es esencial poder identificar la falla, rastrear su origen y poder reproducir su recorrido. Monitorear una aplicación de Microservicios implica conocer el estado de cada servicio individualmente, mientras se observan a su vez las comunicaciones

[7] Imagen obtenida de: https://www.researchgate.net/publication/322842819_Contextual_understanding_of_microservice_architecture_current_and_future_directions

[8] Imagen obtenida de: https://www.researchgate.net/publication/322842819_Contextual_understanding_of_microservice_architecture_current_and_future_directions

entre ellos. Cuando la comunicación con un microservicio falla, el orquestador es quien debe resolverlo. En Coreografía la responsabilidad es de cada servicio participante, lo cual dificulta la tarea.

A continuación se listan algunos de los requerimientos que comúnmente presentan las aplicaciones de Microservicios.

Load Balancing

El Balanceo de Cargas o *Load Balancing* refiere al manejo y distribución del tráfico entrante a una aplicación: el balanceador de cargas actúa como receptor de las solicitudes, que luego distribuye entre las distintas instancias de un servicio capaces de resolverla [46]. En abstracto, el balanceador de cargas se posiciona entre los clientes y los servicios para aplicar las políticas de balanceo de las distintas solicitudes que tenga configurado, como se puede apreciar en la figura 2.4. Se asume que el balanceador de cargas conoce a las distintas instancias de los servicios disponibles, lo cual en la práctica consigue a través de *Service Discovery*.

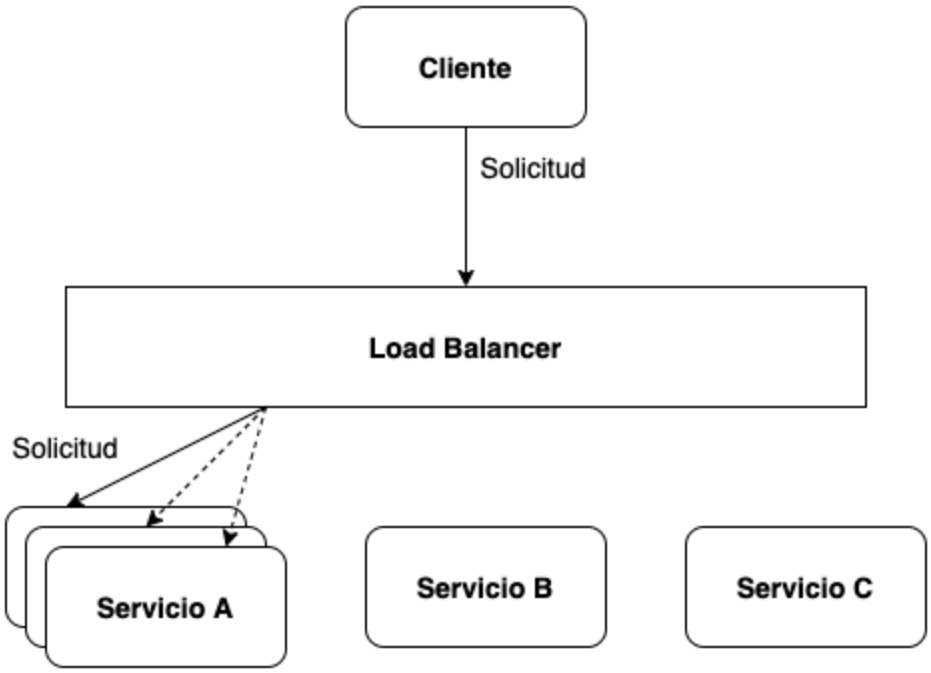

Figura 2.4: El balanceador de cargas conoce a las distintas instancias de los diferentes servicios, información que emplea para aplicar la regla de balanceo que le fue configurada.

Service Discovery

Mientras que en aplicaciones monolíticas las comunicaciones entre servicios suceden a nivel de llamadas inter-proceso, las aplicaciones distribuidas necesitan interactuar sobre la red que las nuclea para resolver las distintas solicitudes que reciben. A su vez, en aplicaciones de Microservicios no es posible asumir ubicaciones preestablecidas para los distintos servicios: el número de instancias de los mismos aumenta y disminuye dinámicamente.

Por todo lo anterior, resulta necesario resolver la forma en que las distintas instancias de los servicios desplegados se dan a conocer entre sí. A esta funcionalidad se la conoce como *Service Discovery* o Descubrimiento de Servicios. Se suele resolver con un componente que registra los servicios desplegados, permitiendo que otros consulten dicho registro (por ejemplo, el componente encargado de realizar el *Load Balancing* descrito anteriormente). La figura 2.5 plantea de manera abstracta como resuelven los Microservicios esta necesidad.

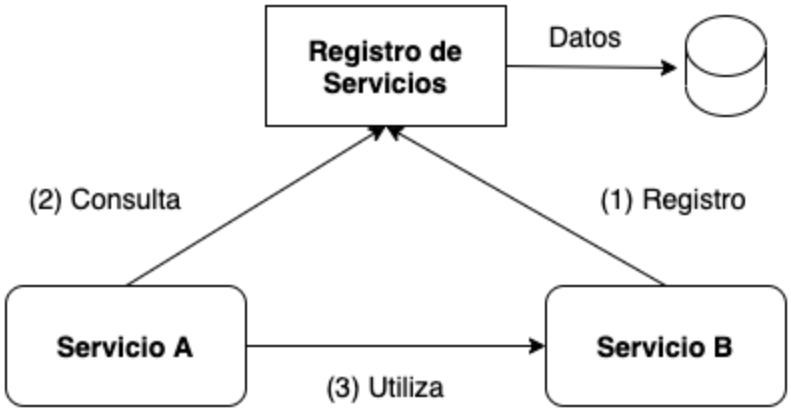

Figura 2.5: Una instancia de un servicio (Servicio B) se registra, tras lo cual otras instancias (Servicio A) pueden consultar dicho registro y comunicarse.

Seguridad

Siendo que los distintos microservicios deben comunicarse a través de la red para colaborar en la resolución de las distintas solicitudes, es necesario contar con mecanismos de seguridad que protejan el intercambio de mensajes.

Dependiendo del stack tecnológico de cada aplicación y sus necesidades, se suelen emplear distintas librerías como Spring Security[9] (en la figura 2.6 representadas por el círculo azul dentro de los servicios) que le brinden autenticación para las comunicaciones entrantes, a la vez que protegen las comunicaciones internas.

Figura 2.6: La seguridad de la comunicación entre microservicios es provista generalmente por librerías de uso extendido, que permiten el uso de protocolos como OAUTH2 y autenticación con JWT[10], entre otros.

[9] https://spring.io/projects/spring-security
[10] https://jwt.io/

Logging

En una aplicación de Microservicios cada instancia de un servicio lleva adelante un Registro o *Log,* donde deja constancia de todo lo que sucede a nivel de errores, advertencias e información sobre su ejecución. En tal contexto, cobra sentido aplicar el patrón de diseño de Centralización de *Logs* [39] representado en la figura 2.7, que permite acceder a los mismos de forma centralizada, evitando así tener que acceder a cada instancia de cada microservicio al momento de querer consultar sus *logs*. Acceso centralizado permite además analizar, realizar búsquedas o incluso configurar alertas que se disparen en base al contenido de los *logs*.

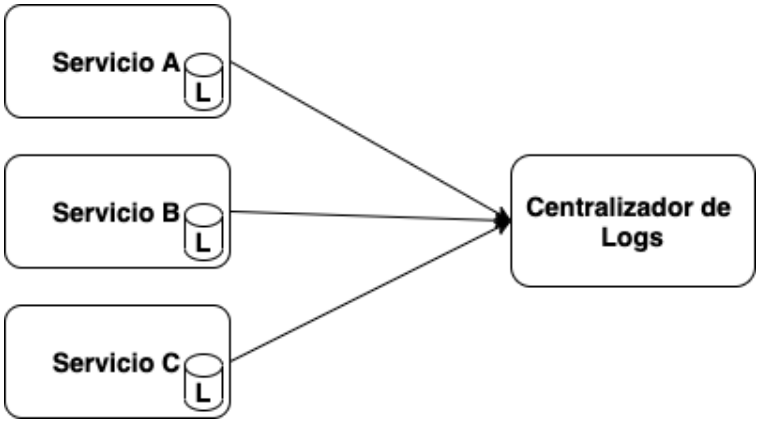

Figura 2.7: Cada microservicio gestiona su *Log* localmente, siendo el rol del Centralizador de *Logs* brindar acceso uniforme y centralizado.

Tracing

Al recorrido que realiza una solicitud desde que ingresa a un sistema hasta que finaliza su procesamiento se lo conoce como Traza o *Trace*. Por su parte, *Distributed Tracing* [40] resuelve la necesidad de entender el camino de los datos en una aplicación distribuida; es una herramienta de diagnóstico que se potencia con los *Logs* vistos anteriormente. Así entonces, la traza de una solicitud que llega a una aplicación de microservicios da cuenta del recorrido que dicha solicitud realiza, mientras los distintos servicios interactúan para resolverla.

La figura 2.8 ejemplifica visualmente el rol de la funcionalidad de *tracing* en una aplicación distribuida. Se suele implementar utilizando un identificador para la solicitud en curso, que la acompaña desde el momento en que ingresa a la aplicación de microservicios. Plasmando dicho identificador junto con un *timestamp* en los *logs* de cada microservicio es que ambas funcionalidades colaboran en el diagnóstico de lo que sucede.

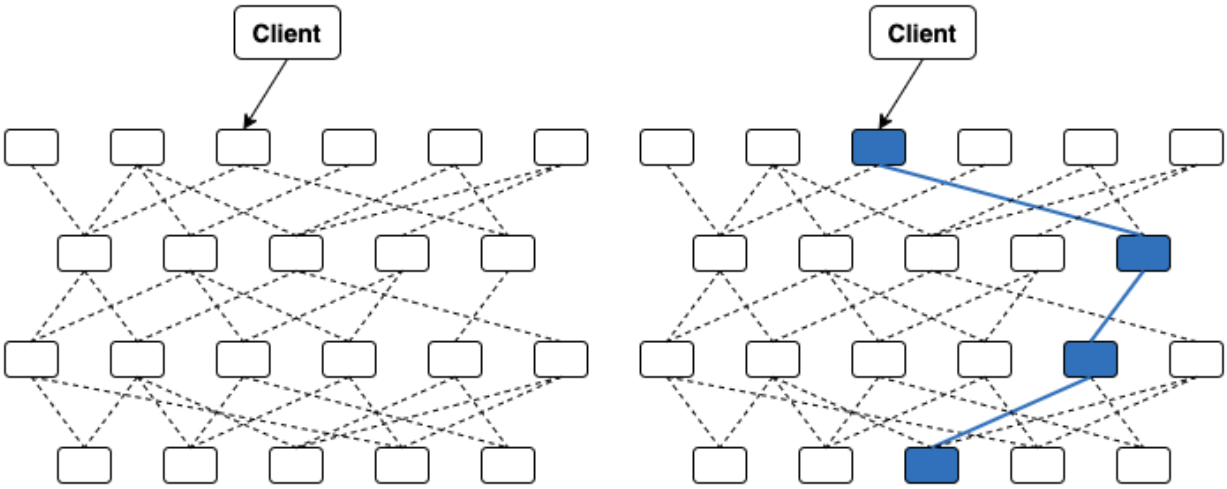

Figura 2.8: No contar con *Distributed Tracing* en una aplicación de microservicios, torna inmanejable poder reproducir y entender el recorrido de una solicitud que ingresa a la misma.

Monitoreo

En el contexto de microservicios, se suele emplear el término Observabilidad[11] para referirse a todas aquellas funcionalidades enfocadas en recabar datos de lo que está ocurriendo con cada microservicio, permitiendo una mejor toma de decisiones, identificar errores e incluso configurar alertas en base a métricas tales como uso de memoria o CPU. Funcionalidades como la Centralización de *Logs* o la Trazabilidad Distribuida forman parte de este conjunto de herramientas.

El monitoreo de una aplicación de Microservicios centraliza la información recabada por las distintas herramientas de observabilidad integradas; brinda una visión en perspectiva del estado de la aplicación a lo largo del tiempo, por ejemplo al controlar el estado de salud de los distintos servicios desplegados [49]. Recabar métricas de rendimiento ayuda a entender y pulir el funcionamiento de la aplicación, identificando por ejemplo, cuellos de botella en la ejecución de los microservicios.

Circuit Breaker

Circuit Breaker es un patrón de diseño que dota de mayor resiliencia a las aplicaciones: se configura un límite (*threshold*) que una vez alcanzado por una instancia de un microservicio, genera que éste sea aislado (*isolated*) del resto de las instancias saludables del servicio. Al momento de aislarlo se inicializa un *timer* que una vez culminado chequea nuevamente la salud del microservicio, enviándole una solicitud. Si responde correctamente, es devuelto al *pool* del que fue removido. En caso contrario continuará aislado y se reinicia el *timer* de chequeo de salud.

[11] https://istio.io/latest/docs/concepts/observability/

La figura 2.9 muestra una máquina de estados que modela el patrón de diseño: el estado *Closed* que representa el funcionamiento normal del servicio, el estado *Open* donde el servicio no es considerado como candidato a recibir solicitudes de la aplicación y el estado *Half Open* donde se evalúa si la instancia del servicio puede volver a ser considerada o aún no.

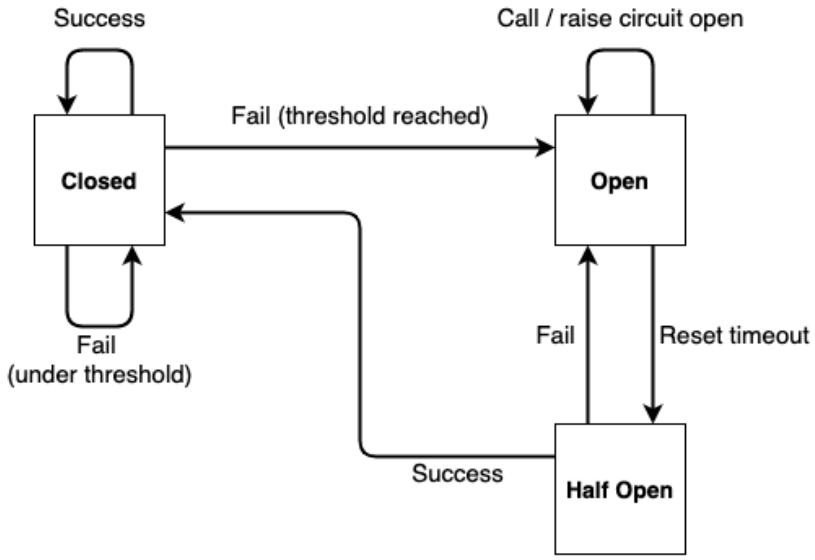

Figura 2.9: Máquina de estado del patrón de diseño *Circuit Breaker*.[12]

Mientras se encuentre en estado *Closed*, el microservicio recibirá solicitudes normalmente, pasando al estado *Open* una vez alcanzado el *threshold* definido. La naturaleza de este límite no es parte de la definición del patrón de diseño; usualmente se implementa como un porcentaje de solicitudes que fallan, que son resueltas en un tiempo superior a un mínimo aceptable o una combinación de ambas.

El estado *Open* modela lo que sucede cuando la instancia del servicio se encuentra aislada del *pool* de instancias consideradas saludables. Del mismo sólo puede pasar al estado *Half Open* por intermedio de un *timer* que comienza a correr una vez la instancia del microservicio pasa del estado *Closed* a *Open*.

Finalmente el estado *Half Open* representa un estado intermedio donde se le envía una solicitud de la aplicación al microservicio: si este contesta satisfactoria o desfavorablemente desde el punto de vista del *threshold*, se lo devuelve al estado *Closed* u *Open* respectivamente.

Testing

Escribir *tests* para aplicaciones de microservicios no escapa a las diferencias que estos presentan respecto a aplicaciones monolíticas. La naturaleza de los *tests* queda determinada por la mayor granularidad de los microservicios, debiendo abarcar tanto su interior como sus interacciones. Es por ello que funcionalidades que en un monolítico pueden ser testeadas con una combinación de *tests* unitarios y *tests* de componente sin abandonar el interior del mismo,

[12] Adaptación de la imagen disponible en: https://martinfowler.com/bliki/images/circuitBreaker/state.png

en microservicios potencialmente involucran distintos servicios interactuando. En consecuencia, a los *tests* anteriormente mencionados se le deben agregar testeos de contrato y de integración [50].

En el ecosistema de Microservicios existen diferentes herramientas especializadas en facilitar el testeo en áreas comunes a todos los microservicios. Estas herramientas cubren aspectos como ayudar en la redacción de testeos unitarios y de integración, testeos de *performance* ante cambios en la carga de trabajo, testeos de contrato para cubrir las interfaces de cada microservicio o testeos de resiliencia, cubriendo el comportamiento de los distintos servicios ante demoras o caídas de los diferentes servicios con los que interactúan.

Configuración Externa

Al comunicarse con servicios de terceros, los microservicios típicamente necesitan información de configuración y credenciales para enviar y recibir mensajes. Por otra parte, es deseable que los microservicios sean capaces de ejecutar en distintos ambientes sin necesidad de ser recompilados o modificados, todo lo cual es posible si se externaliza la configuración de los mismos. Esto le permite a los microservicios consultar por su configuración al ser lanzados, pudiendo incluso recibir actualizaciones de su configuración mientras se encuentran ejecutando.

El patrón de diseño de *Externalized Configuration* o Configuración Externa [51] apunta a cubrir estas necesidades con un componente externo a los microservicios. En la práctica esto se resuelve de distintas maneras según la aplicación. Se suele delegar dicha funcionalidad a un microservicio específico o como se analiza más adelante en el documento, a nivel de un Orquestador de Contenedores o una Service Mesh, como se aprecia en la figura 2.10.

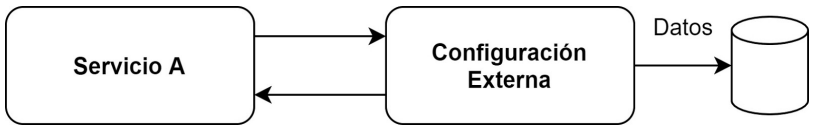

Figura 2.10: Los microservicios consultan con el componente de configuración externa al ser desplegados. A su vez, es posible que el componente envíe actualizaciones mientras ejecuta el servicio.

Despliegue Automático

Como se observaba en la sección 2.1 y en la figura 2.1, Microservicios permite un mayor control de las necesidades de escalar de las aplicaciones, pudiendo hacerlo a nivel de servicios, en lugar de tener que hacerlo a nivel de la aplicación.

Contar con esta mayor granularidad y con herramientas de monitoreo que controlan el estado de cada servicio en tiempo real, genera un ambiente de trabajo ideal para explotar el despliegue automático de los microservicios. Ante fluctuaciones en la demanda de un servicio específico, es posible desplegar o retirar instancias del mismo sin intervención del operador de

la aplicación. Todo lo anterior hace que el sistema sea capaz de soportar las cargas de trabajo estimadas con un eficiente empleo de los recursos.

Del mismo modo, los mecanismos que controlan el estado de salud de las distintas instancias son el insumo ideal para que el despliegue automático retire y reemplace por una nueva instancia, a aquellas que no cumplen con los estándares de rendimiento pre establecidos.

El patrón de diseño Plataforma de lanzamiento de servicios [17] detalla las características del contexto de trabajo donde se maximizan los beneficios de contar con un sistema que se encargue del despliegue de los distintos servicios, manteniéndolos monitoreados y quitando o desplegando nuevas instancias según sea necesario.

Resumen

El patrón de diseño de Microservicios limita el rango de acción de los mismos a una sola funcionalidad de negocio de la aplicación, restando determinar el lugar de residencia de aquellas funcionalidades que sin ser específicamente de negocio, proporcionan herramientas de infraestructura necesarias.

La tabla 2.2 resume las funcionalidades de infraestructura observadas anteriormente junto a una breve descripción de cómo o donde se resuelve habitualmente, en aplicaciones que no cuentan con un Orquestador de Contenedores ni una Service Mesh.

Contexto	Funcionalidad	Componente
Existe más de una instancia de un determinado servicio, capaz de resolver una solicitud.	*Load Balancing*	Librería (Ej Ribbon[13], de Netflix[14]).
Nuevas instancias de servicios desplegadas, deben poder darse a conocer entre sus pares.	*Service Discovery*	Microservicio encargado de mantener registro de servicios/instancias. Librería (Ej Eureka[15], de Netflix).
La comunicación desde y hacia la aplicación, así como los mensajes entre microservicios deben ser seguros.	Seguridad	Se emplean estándares de la industria como OAuth y JWT, mediante el empleo de distintas librerías.

[13] https://github.com/Netflix/ribbon
[14] https://netflix.com
[15] https://github.com/Netflix/eureka

Contexto	Funcionalidad	Componente
Cada microservicio genera un *log* con información respecto a su ejecución. Se desea contar con acceso centralizado a los mismos.	*Logging*	Se suele emplear una librería para el procesamiento y centralización (Ej Splunk[16] o Logback[17]); puede también emplearse un servicio propio. Se suelen emplear Dashboards como Graylog[18] para la visualización.
Una solicitud que llega a la aplicación realiza un trayecto entre microservicios que se desea poder identificar.	*Tracing*	Herramientas como Sleuth[19] y sistemas como Jaeger[20] para el manejo de las trazas. Se suele aplicar el *standard* OpenTracing[21] .
Se debe conocer el estado de la aplicación en general y de cada microservicio en particular, sin importar la cantidad que haya desplegada.	Monitoreo	Se suelen emplear Dashboards (Ej Prometheus[22], Grafana[23]).
Dada una instancia de un servicio que no está respondiendo adecuadamente, se desea contar con un mecanismo que la aísle del sistema.	*Circuit Breaker*	Suele llevarse a cabo a nivel de los microservicios o empleando una librería (Ej Hystrix[24]).
Dada la naturaleza distribuida de las aplicaciones de microservicios, además de testear la lógica individual de cada uno, es necesario cubrir las interacciones entre ellos.	*Testing*	La lógica de negocio de cada microservicio se testea localmente. Luego existen distintas librerías para testear los diferentes momentos de colaboración entre microservicios (testeo de contratos, de integración, de cargas, etc.)

[16] https://github.com/splunk/splunk-library-javalogging
[17] http://logback.qos.ch/
[18] https://www.graylog.org/
[19] https://github.com/spring-cloud/spring-cloud-sleuth
[20] https://www.jaegertracing.io/
[21] https://opentracing.io/
[22] https://prometheus.io/
[23] https://grafana.com/
[24] https://github.com/Netflix/Hystrix

Contexto	Funcionalidad	Componente
Al momento del despliegue y durante su ejecución, es deseable que los microservicios puedan cargar desde una fuente externa, aspectos de configuración como claves y direcciones.	Configuración externa	Un microservicio cumple la función de proveer al resto de los elementos de configuración que necesite para ejecutar.
Se desea contar con una herramienta que gestione y monitorice el despliegue de los distintos servicios, automatizando dichos procesos.	Despliegue automático	En aplicaciones contenerizadas, se emplean Orquestadores de Contenedores (más al respecto en la sección 3.4).

Tabla 2.2: Tabla que resume las funcionalidades comunes a aplicaciones de Microservicios, en qué contexto son utilizadas y mediante qué herramienta, librería o componente suelen ser cubiertas.

2.2 Service Mesh

El concepto Service Mesh refiere a una capa de infraestructura dedicada al manejo confiable de la comunicación servicio a servicio, dentro de una aplicación distribuida [12]. Como se observa en la sección 2.1, en aplicaciones de Microservicios es fundamental resolver la complejidad que resulta de la comunicación entre servicios distribuidos, garantizando que ésta sea segura y efectiva, monitoreando el comportamiento de las distintas instancias de los servicios que la componen, actuando ante cambios en el flujo de mensajes y las interacciones de usuarios externos. Abstraer este conjunto de funcionalidades fuera de los microservicios resulta en una representación más fiel al patrón de diseño de Microservicios, acotando el alcance de estos a la lógica de negocio que resuelven.

En el proceso evolutivo de trasladar fuera de los microservicios las funcionalidades de infraestructura [4] surge el patrón de diseño *Sidecar* [13], cuya sugerencia es implementar las funcionalidades que no sean de negocio en procesos que corren en paralelo a cada microservicio. Usualmente se implementa este patrón de diseño desplegando un *proxy* por cada instancia de microservicio, que funciona interceptando todas las comunicaciones entrantes y salientes para aplicar la lógica necesaria.

Es sobre esta noción de *proxies* corriendo en paralelo al resto de la aplicación que se construyen las Service Mesh actuales, siguiendo la división de roles descrita a continuación.

Data Plane y Control Plane

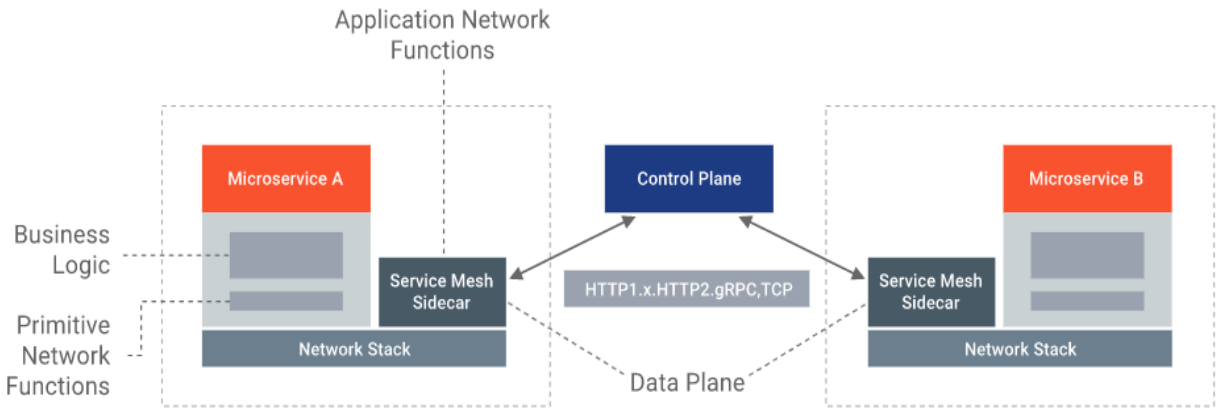

Figura 2.11: Vista en alto nivel de los componentes habitualmente presentes en una Service Mesh[25].

Los conceptos de *Data Plane* y *Control Plane* que aparecen en la figura 2.11, ayudan a esclarecer la confusión generada en torno a las Service Mesh [14]. La creciente popularidad de las mismas y de otras herramientas del ecosistema que van surgiendo, dificultan la tarea de comparar, contrastar y elegir entre las distintas propuestas.

El plano de datos o *Data Plane* actúa sobre cada uno de los paquetes intercambiados en el sistema, aplicando la lógica necesaria para recolectar información acerca de lo que sucede en cada microservicio, chequear su salud o ejercer las políticas que la Service Mesh determine.

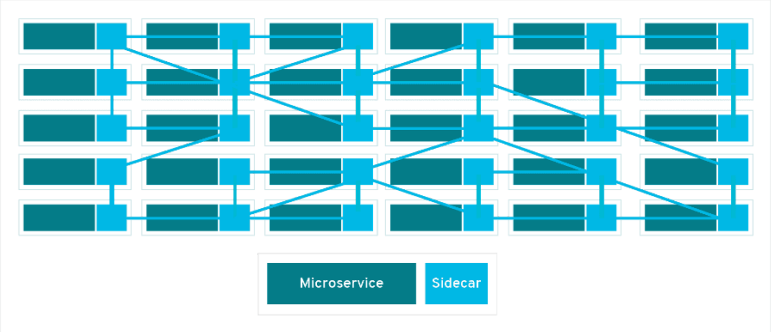

Figura 2.12: Los microservicios se despliegan junto con su *Proxy* (*Sidecar*): su interacción con otros microservicios ocurre sólamente a través de él. [26]

[25] https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/
[26] Imagen obtenida de: https://www.redhat.com/es/topics/microservices/what-is-a-service-mesh

Como se observa en la definición del patrón de diseño *Sidecar* y en la figura 2.12, se suele implementar el *Data Plane* como una colección de proxies desplegados junto a cada microservicio.

Por su parte, el plano de control o *Control Plane* se encarga de configurar y monitorear lo que sucede a nivel global: es quien determina las directivas a seguir por los proxies del *Data Plane*, quien procesa y analiza la información de lo que sucede en cada microservicio, transformando así instancias aisladas de microservicios en una sola aplicación distribuida y coherente.

A diferencia de lo que sucede con el *Data Plane*, el *Control Plane* no interactúa directamente con los mensajes intercambiados en la aplicación: su accionar es sobre metadatos de la Service Mesh. La Figura 2.13 ayuda a entender la separación de roles del *Data Plane* y el *Control Plane*.

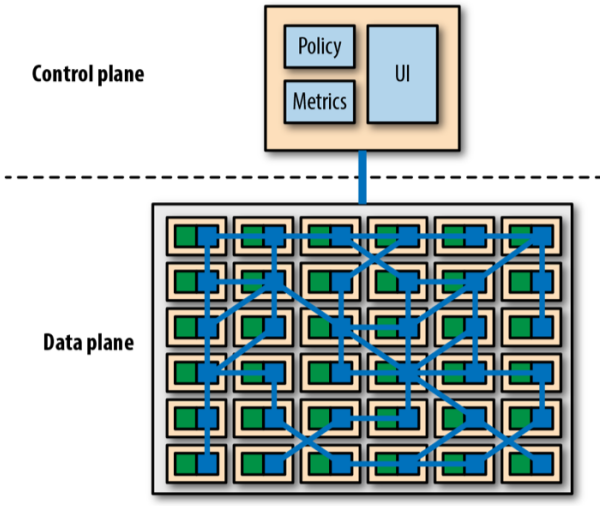

Figura 2.13: División de una Service Mesh en *Data Plane* y *Control Plane*: el *Control Plane* dicta las políticas que luego el *Data Plane* ejecuta. [27]

2.3 Funcionalidades de una Service Mesh

Como parte del estudio realizado, se plantea la necesidad de contar con una lista de las funcionalidades habitualmente encontradas en una Service Mesh. Al no contar con una fuente definitiva a la que acudir, se recurre a las definiciones de Service Mesh provistas por distintos actores del ecosistema[28]. Este relevamiento se resume en la figura 2.14 donde pueden observarse las funcionalidades que más se repiten en estas definiciones.

[27] Imagen obtenida de: https://thenewstack.io/which-service-mesh-should-i-use/
[28] Ver Apéndice D: Definiciones de Service Mesh

Fuente	Load Balancing	Security	Service Discovery	Observability	Remote Config	Loging	Routing	Circuit Breaker	Traceability	Testing
Nginx	X	X	X	X	X			X	X	
VMWAre	X	X		X			X			
Red Hat				X						
Istio	X	X	X	X	X	X		X	X	X
HashiCorp	X	X	X				X			
Glasnostic	X	X	X		X	X	X	X		
Buoyant	X		X		X			X		
Envoy	X	X	X	X	X	X	X		X	
Microservices		X		X	X	X			X	
InfoQ	X	X	X	X	X	X	X	X	X	X
Total	**8**	**8**	**7**	**7**	**7**	**5**	**5**	**5**	**5**	**2**

Figura 2.14: Tabla resumiendo las funcionalidades (columnas) destacadas en las definiciones de Service Mesh provistas por las distintas empresas (filas) del estudio realizado.[29]

Load Balancing

La Service Mesh facilita el balanceo de cargas tanto a nivel de la capa 7 o capa de aplicación, como de la capa 4 o capa de red del modelo OSI. El balanceo puede ser llevado a cabo de manera activa, chequeando antes de rutear una *request*, o de manera pasiva, respondiendo en base a la *performance* que interpreta de la red de microservicios. En consecuencia, aspectos clave como la latencia de cada instancia de un servicio o la cantidad de solicitudes encolados que tiene, cumplen un rol fundamental a la hora de decidir hacia qué instancia rutear un mensaje.

Service Discovery

La Service Mesh mantiene el registro de servicios, automatizando el proceso de altas y bajas al mismo. En colaboración con un Orquestador de Contenedores, el despliegue de nuevas instancias de un servicio puede ser automatizado por completo.

Al apoyarse en una Service Mesh para contar con *Service Discovery*, las aplicaciones evitan tanto el empleo de librerías específicas para cada lenguaje (permitiendo aplicaciones políglotas) como el alto acoplamiento de emplear un microservicio para mantener el registro de servicios.

Seguridad

Mediante autenticación la Service Mesh trabaja sobre los usuarios que intentan establecer una comunicación con la aplicación. Sobre los mensajes intercambiados entre microservicios, utiliza mTLS (*mutual TLS*) [47]. Mediante autorización se mantiene dinámicamente registro de qué servicios puede comunicarse con qué otros servicios.

[29] Relevamiento realizado el 05/08/2019

En consecuencia y asumiendo que el intercambio sucede sobre una red no confiable, la SM garantiza la seguridad de los mensajes intercambiados entre microservicios (tráfico Este-Oeste) [38].

A este enfoque de seguridad, donde se entiende que la misma no está garantizada, aún luego de que las solicitudes ingresan a la aplicación, se lo conoce como *Zero Trust Security* [42]: se asume que la seguridad de la comunicación dentro de la aplicación está comprometida, trabajando sobre la misma como si se tratase de una red pública. *Zero Trust Security* funciona también con la noción de otorgar el mínimo privilegio posible, denegando solicitudes por defecto.

A la hora de garantizar la seguridad de la aplicación, *Data Plane* y *Control Plane* colaboran cada uno desde su rol: el *Control Plane* se encarga de distribuir la configuración que luego los *proxies* aplican al comunicarse entre sí.

Logging

Si bien generar *logs* que permitan un correcto diagnóstico de lo que sucede en su interior continúa siendo responsabilidad de cada microservicio, el *Data Plane* colabora en el registro de lo que sucede a nivel de los mensajes intercambiados, a la vez que se ocupa de su recolección. Queda a nivel del *Control Plane* la centralización y visualización de los mismos.

Tracing

El rol de la Service Mesh dentro de Trazabilidad Distribuida es el de facilitar el manejo de los cabezales HTTP que deben ser propagados por los microservicios, logrando así recrear el recorrido de las solicitudes recibidas.

Mediante soporte para integrar herramientas externas como Zipkin[30] o Lightsetp[31] las Service Mesh completan el alcance de la funcionalidad al permitir visualizar las trazas en un *dashboard*.

Monitoreo

Al generar y recolectar métricas de lo que sucede a nivel de los distintos microservicios, las Service Mesh permiten observar tiempos de respuesta (latencia, saturación), la relación entre volumen de tráfico y errores, además de otros aspectos personalizables de los microservicios y *proxies* del *Data Plane*.

El *Control Plane* centraliza los reportes y permite realizar diferentes consultas. El *Data Plane* en cambio es el encargado de la recolección de dichas métricas, trabajando cada *proxy* sobre el tráfico entrante y saliente al microservicio que acompaña.

[30] https://zipkin.io/
[31] https://lightstep.com/

Circuit Breaker

Aplicando el patrón *Circuit Breaker* a nivel de los *proxies* del *Data Plane*, se abstrae por completo al microservicio de cómo maneja la Service Mesh esta funcionalidad. En colaboración con el *Control Plane*, es posible poner a disposición del operador de la aplicación de microservicios una herramienta que le permita visualizar el estado de cada *Circuit Breaker*.

Al trasladar la aplicación de *Circuit Breaker* de los microservicios a la Service Mesh, se logra emplear la misma herramienta en aplicaciones políglotas. Sin embargo, al abstraer su implementación de la lógica de negocio del microservicio, deja de ser posible contar con funcionalidades útiles como por ejemplo los Fallback Methods[32] de Hystrix.

Testing

El potencial a nivel de testeo que introducen las Service Mesh abarca distintos aspectos del desarrollo de aplicaciones basadas en Microservicios.

Permiten por ejemplo, configurar *fault injections* para simular y monitorear el comportamiento de la aplicación ante potenciales fallas o retrasos en cada uno de sus servicios, verificando la capacidad de recuperación de la misma.

Pudiendo ejercer políticas de ruteo dinámicas controladas por la Service Mesh, se extiende el abanico de posibilidades de testear una aplicación de Microservicios con el objetivo de aumentar la confiabilidad en cada actualización.

Entre otros, es posible aplicar conceptos como *Traffic Shadowing* o *Traffic Mirroring, Canary Releases*[33] *o Blue-Green Development*[34]*.*

Aplicando conceptos de ruteo similares a los descritos anteriormente es posible realizar testeos A/B donde se divide el tráfico de usuarios entre 2 experimentos y se analizan métricas en busca de conclusiones sobre los experimentos realizados.

Configuración Externa

Esta funcionalidad está presente en algunas de las Service Mesh analizadas pero no en su totalidad. Por su parte, Orquestadores de Contenedores como *Kubernetes* o *Docker Swarm*[35] ofrecen esta posibilidad.

Resumen

La tabla 2.3 detalla cómo las problemáticas descritas en la sección 2.3 son abordadas a nivel de una Service Mesh, colaborando *Data Plane* y *Control Plane*.

[32] https://github.com/Netflix/Hystrix/wiki/How-To-Use#Fallback
[33] https://martinfowler.com/bliki/CanaryRelease.html
[34] https://martinfowler.com/bliki/BlueGreenDeployment.html
[35] https://docs.docker.com/get-started/swarm-deploy/

Funcionalidad	Control Plane	Data Plane
Load Balancing	Determina la política de balanceo de cargas a realizar. Mantiene un registro de las instancias de un servicio disponibles.	Consulta el registro de instancias de un servicio disponibles y realiza el balanceo de cargas. Los distintos sidecars se consultan entre sí periódicamente por el estado de salud de cada instancia.
Service Discovery	Es responsable de que se mantenga el registro de las diferentes instancias de servicios disponibles.	Se comunica con el *Control Plane* para mantener los registros de las instancias desplegadas.
Seguridad	Permite diferentes configuraciones que determinan por ejemplo, qué servicio puede comunicarse con qué otro servicio.	Es el principal actor en materia de seguridad, siendo que todo el intercambio de mensajes en la red sucede a nivel de los proxies del *Data Plane.*
Logging	Centralización, consulta y visualización de los *logs* recolectados por el *Data Plane*.	Recolección de los *logs* generados a nivel de cada microservicio; generación propia de *logs* referidos al intercambio de mensajes y el funcionamiento de la Service Mesh en sí.
Tracing	Configuración de los proxies para cumplir con la funcionalidad; visualización de las trazas.	Propagación y generación de los cabezales HTTP que permiten la reproducción del recorrido o traza de una solicitud que llega al sistema.
Monitoreo	Determina las métricas que el *Data Plane* debe recolectar. Permite visualizar las mismas y determinar acciones automáticas en base a ellas.	Cada *proxy* obtiene las métricas requeridas y las transmite hacia el componente que el *Control Plane* determina.
Circuit Breaker	Configura los límites que hacen disparar a los distintos *Circuit Breakers*; permite visualizar el estado de cada uno.	No rutean tráfico hacia las instancias de servicios cuyo *Circuit Breaker* esté abierto.
Testing	Permite configurar los valores de las distintas herramientas de testeo que ofrece la Service Mesh.	Aplica las directivas determinadas por el *Control Plane*.

Funcionalidad	Control Plane	Data Plane
Configuración externa	En los casos donde esta funcionalidad es ofrecida por la Service Mesh, permite guardar los datos a levantar por parte de los diferentes microservicios.	Se encarga de recabar y comunicar al microservicio junto con el que fue desplegado, los valores de configuración inicial obtenidos.
Despliegue Automático	-	-

Tabla 2.3: Análisis de la colaboración entre *Data Plane* y *Control Plane*, funcionalidad a funcionalidad.

2.4 Orquestador de contenedores

Contenedores

Los contenedores son una alternativa a la virtualización[36] mediante el empleo de *Virtual Machines* (VMs), tecnología predominante hasta entonces en la nube [55]. Como tales, empaquetan todo el código y sus dependencias permitiendo que la aplicación pueda ser ejecutada rápida y confiablemente en distintos ambientes [56].

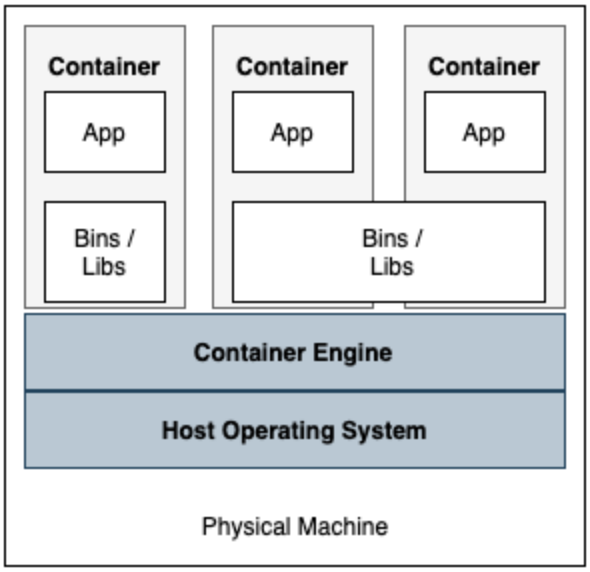

Figura 2.16: Arquitectura de un servicio virtual contenerizado [56].

Los contenedores cuentan con un Gestor de Contenedores que se encarga tanto de su administración como de su despliegue. Para ello, el Gestor de Contenedores emplea imágenes

[36] https://www.redhat.com/en/topics/virtualization

de contenedores: empaquetados que facilitan el despliegue y se convierten en contenedores en tiempo de ejecución. El empleo de estas imágenes habilita a que sean reutilizadas por diferentes aplicaciones. El Gestor de Contenedores permite un eficiente empleo de los recursos disponibles al soportar que los distintos contenedores puedan compartir librerías entre sí, como muestra la figura 2.16.

A diferencia de las máquinas virtuales, los contenedores no incluyen un sistema operativo completo, sino que comparten recursos con el sistema operativo donde son lanzados. La virtualización que realizan del *hardware* las máquinas virtuales, en el caso de los contenedores sucede a nivel del sistema operativo. La figura 2.17 toma la arquitectura presentada en la figura 2.16 y la compara con su símil a nivel de máquinas virtuales.

Contenedores y Microservicios

Las aplicaciones de Microservicios se ven altamente beneficiadas con el uso de contenedores: éstos permiten un *deployment* independiente, confiable y eficiente, aumentan la portabilidad de las aplicaciones, facilitan el manejo del *stack* tecnológico que cada microservicio utiliza y le permiten al desarrollador desentenderse de las particularidades de la máquina o el entorno donde el microservicio va a correr [15].

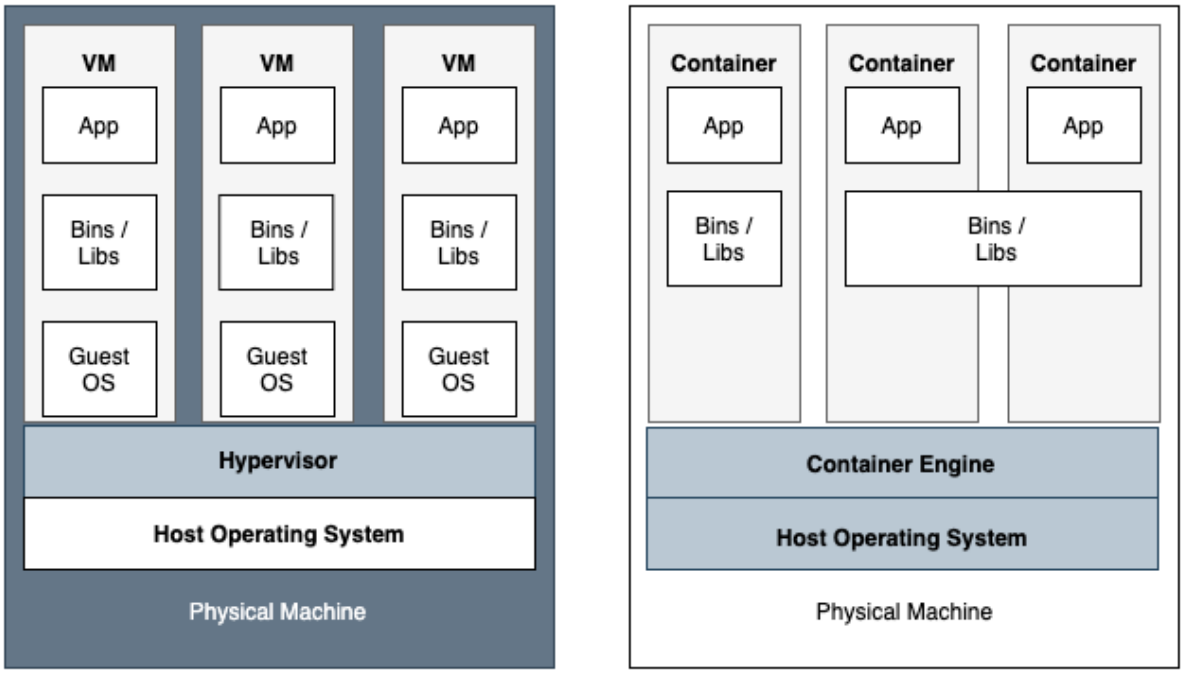

Figura 2.17: Comparación de las arquitecturas según se emplee una máquina virtual o un contenedor. Imagen adaptada de imagen en [56].

El patrón de diseño que vincula microservicios y contenedores sugiere el *deployment* de una única instancia de un determinado servicio por contenedor [16]. Por su parte, el patrón de

diseño Plataforma de Lanzamiento de Servicios [17] detalla los posibles contextos de trabajo donde contar con la integración de un Orquestador de Contenedores cobra más sentido.

Orquestadores de Contenedores

Los Orquestadores de Contenedores automatizan distintos aspectos relacionados con el despliegue, la gestión y las necesidades de escalar de las aplicaciones contenerizadas [48], que de lo contrario deben ser llevados a cabo manualmente. Los Orquestadores funcionan mediante archivos de configuración que especifican la imagen de contenedores a utilizar, si montar o no almacenamiento en el contenedor, donde almacenar los *logs* del contenedor y cómo debe ser armado el enramado de contenedores. A partir de allí, son responsables del ciclo de vida de los contenedores, de manejar eficientemente el despliegue de nuevos contenedores debiendo elegir el momento y el *host* donde realizarlo.
A nivel de recursos, son responsables de su correcto aprovechamiento, debiendo incluso gestionar nuevos recursos a medida que estos están disponibles. Esto lo consiguen por ejemplo, mediante el traslado de un contenedor a un nuevo host, si el original se encuentra escaso de recursos o no disponible.

Los Orquestadores de Contenedores no limitan su accionar al despliegue de los contenedores, sino que también se involucran en el tiempo de vida del microservicio, colaborando en tareas como el monitoreo de la salud de los contendores, ruteo, *service discovery* y *load balancing*, seguridad y configuración externa entre otras. Este solapamiento de funcionalidades entre las que proveen los Orquestadores de Contenedores y una Service Mesh es abordado en la sección 2.5. Por su parte, la tabla 2.4 resume las funcionalidades que es esperable encontrar en un Orquestador de Contenedores.

Funcionalidad	Descripción
Configuración	A partir de un archivo de configuración YAML o JSON, el orquestador levanta la imagen del contenedor deseada, provee de almacenamiento de ser necesario, determina dónde guardar sus *logs* y cómo se vincula el contenedor con el resto, entre otras propiedades.
Planificación	El orquestador planifica el despliegue de un contenedor, encontrando el momento y *host* más apropiado donde desplegarlo, en base a métricas relacionadas a recursos disponibles u otros aspectos configurables.
Monitoreo del ciclo de vida	El orquestador se encarga de monitorear el contenedor durante su ciclo de vida, controlando su salud y asignando recursos si fuese necesario.
Disponibilidad	El orquestador actúa sobre los reportes de salud de los contenedores para asegurar la disponibilidad de los mismos, realojandolos ante eventuales fallas o faltantes de recursos.

Funcionalidad	Descripción
Escalabilidad	Los orquestadores pueden ser configurados para desplegar nuevas instancias de contenedores en base a la demanda de los usuarios de la aplicación. Lo mismo si la demanda es baja, pueden quitar instancias de servicios poco requeridos para liberar recursos.
Load Balancing	En base al monitoreo que mantiene sobre los contenedores, el orquestador es capaz de balancear las cargas entre las distintas instancias de un servicio.
Service Discovery	Siendo el encargado del despliegue de nuevos contenedores, el orquestador de contenedores puede también solucionar el registro y la búsqueda de servicios.

Tabla 2.4: Listado y breve descripción de las funcionalidades principales de un Orquestador de Contenedores.

Kubernetes

Kubernetes es un Orquestador de Contenedores de código abierto que permite manejar servicios contenerizados, facilitando su configuración y posterior automatización: se ocupa del escalado y la disponibilidad de los servicios [11]. Su naturaleza extensible le permite a los equipos de trabajo agregar nuevos elementos a la herramienta según sus necesidades, lo cual ha hecho proliferar un ecosistema de soluciones construidas sobre Kubernetes como OpenShift [37] o PKS[38].

Kubernetes cubre las funciones esperables en un Orquestador de Contenedores: permite que los servicios contenerizados pueden descubrirse entre sí (*Service Discovery*) y balancear el tráfico entre las instancias de un mismo servicio (*Load Balancing*), permite configurar directivas de asignación de recursos y de almacenamiento de datos (local o en la nube), monitorea la salud de los contenedores pudiendo reiniciar, quitar o reemplazar contenedores que no estén funcionando debidamente, a la vez que automatiza procesos de despliegue debido a fluctuaciones en el tráfico de la aplicación o nuevas versiones de los servicios contenerizados, entre otros.

La figura 2.18 provee una vista resumida de la arquitectura de Kubernetes. Un *Pod* es una colección de uno o más contenedores: es la unidad básica en Kubernetes. Por su parte los *Nodes* (Nodos) son máquinas virtuales o físicas donde Kubernetes coloca *Pods* a ejecutar. Un *Cluster* en Kubernetes consiste de al menos un nodo maestro y múltiples nodos trabajadores. El *Master Node* o Nodo Maestro es quien expone la API de Kubernetes, planifica los despliegues y administra el cluster en sí. Los *Worker Nodes* o Nodos Trabajadores son quienes

[37] https://www.openshift.com/learn/topics/kubernetes/
[38] https://docs.pivotal.io/pks/

ejecutan componentes que realizan tareas como monitoreo o *logging,* al tiempo que permiten que las aplicaciones accedan a los distintos recursos.

2.5 Integración Orquestador de Contenedores - Service Mesh

En aplicaciones contenerizadas que utilicen tanto un Orquestador de Contenedores como una Service Mesh, durante el tiempo de vida de los servicios existen áreas de injerencia donde ambas herramientas disponen de la capacidad de resolver las necesidades de los mismos. El solapamiento en funcionalidades como *Service Discovery* y *Load Balancing* entre otras, torna difusa la frontera de acción de ambas herramientas.

Al definir una Service Mesh en la sección 2.2, se hace especial énfasis en su caracterización como una capa de infraestructura que se encarga de que la comunicación servicio a servicio sea segura y confiable. En ocasiones, la Service Mesh se vale de integraciones con otros servicios, herramientas u Orquestadores de Contenedores para llevar a cabo su cometido. No obstante la facultad de colaborar de ambas herramientas, el despliegue del *Data Plane* y el *Control Plane* de una Service Mesh no necesariamente necesita de un ambiente contenerizado [57].

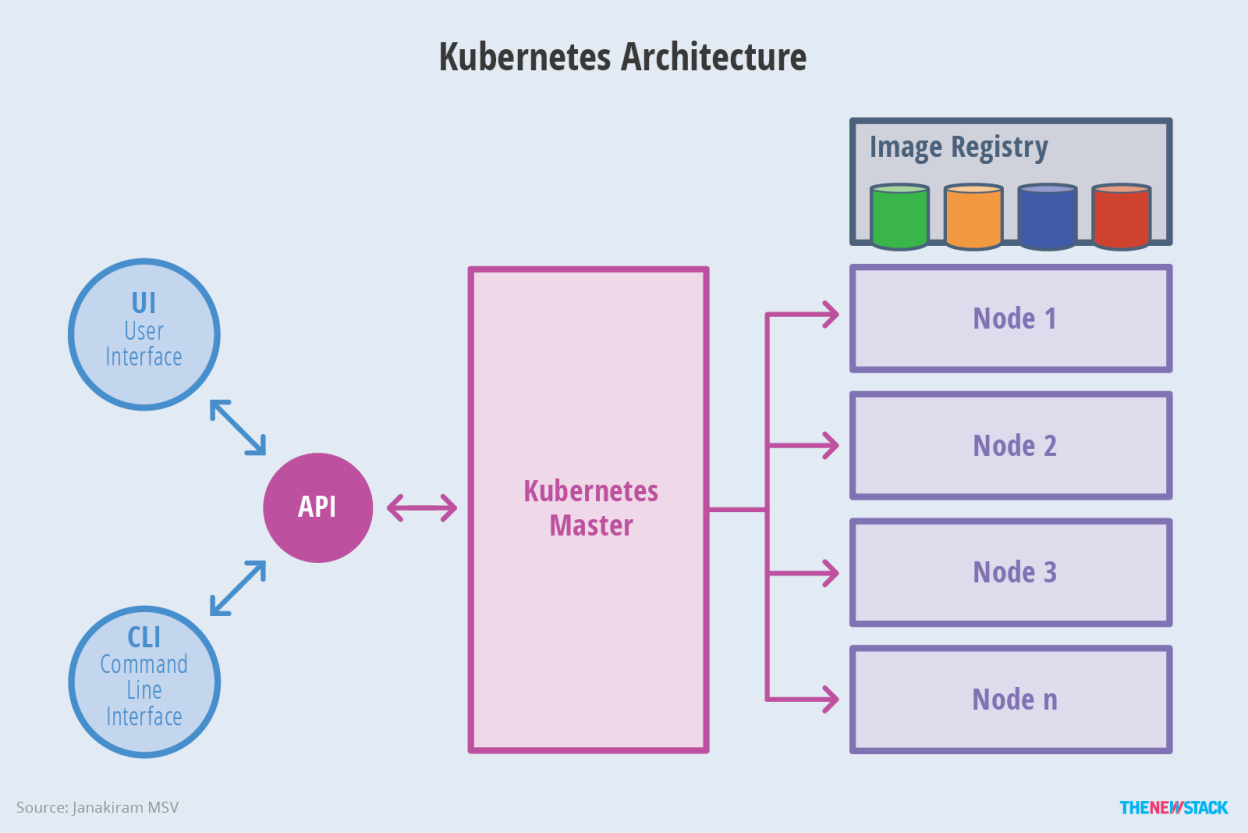

Figura 2.18: Vista simplificada de la arquitectura de Kubernetes [58].

Las Service Mesh permiten ser agregadas a un proyecto existente sin que esto signifique tener que migrar la totalidad de las funcionalidades que ésta brinda: distintos equipos de trabajo pueden optar por contar con una Service Mesh para áreas específicas como pueden ser Observabilidad o Seguridad [57], según sus necesidades.

Conceptualmente es más sencillo trazar una línea divisoria entre ambas herramientas: los Orquestadores de Contenedores son herramientas para manejar y monitorear contenedores [19]. Éstos han resuelto en gran medida el *deployment* de aplicaciones estandarizando su proceso, lo cual reduce drásticamente el tiempo que demanda dicha tarea. Son las Service Mesh las que proveen los mecanismos para estandarizar a continuación, el tiempo de vida de las aplicaciones [20].

2.6 Plataforma de Integración

Plataforma de Integración

Una Plataforma de Integración es un *software* especializado en permitir que sistemas heterogéneos en ambientes distribuidos sean capaces de conectarse de manera confiable, aceptando solicitudes en forma de mensajes sobre los que luego se realizan tareas de mediación para lidiar con las diferencias de dichos sistemas [7].

Las PI permiten crear Soluciones de Integración (también conocidas como Flujos de Integración) en base a componentes de integración reutilizables, desarrollados generalmente en torno a los *Enterprise Integration Patterns* [10]. Una Solución de Integración (SI) es una serie de pasos a ejecutar para lograr la integración de dos o más sistemas. Además de su creación, la PI se encarga de su ejecución y gestión. La figura 2.19 ayuda a comprender los conceptos anteriormente mencionados.

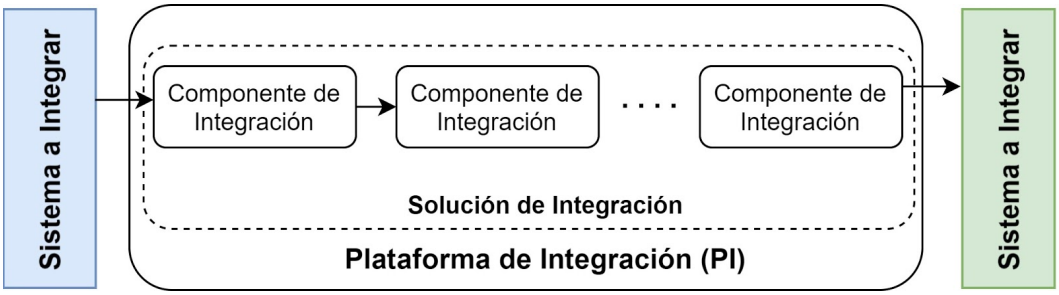

Figura 2.19: Esquema de los conceptos de PI, SI y componentes de integración. [7]

Cada paso de la SI se corresponde con un componente de integración. Se conoce como disparadores a los componentes iniciales de cada SI, que reaccionan con su ejecución ante un estímulo o evento, finalizando la ejecución de la SI cuando termina de ejecutar su último componente. Cada componente luego de terminada la ejecución de su rol dentro de la SI, invoca al siguiente y le transmite la información necesaria.

Los componentes de integración se dividen en tres categorías: básicos, avanzados e interorganizacionales. Los componentes avanzados e interorganizacionales no son abordados en el corriente trabajo. Por su parte, los componentes básicos pueden ser categorizados según el criterio descripto en la tabla 2.5 [7].

Categoria	Descripción
Conectores Simples	Son aquellos componentes que ofrecen una conectividad simple sobre protocolos como HTTP o SMTP.
Conectores Específicos	Son aquellos componentes que cumplen la tarea de resolver la conectividad en situaciones complejas.
Virtualización de Servicios	Resuelven la conectividad con servicios web que sea necesario para integrar los sistemas en cuestión, mediante el envío y/o recepción de información.
Transformación	Permiten efectuar transformaciones de los datos recibidos, como las definidas en los EIPs [10].
Ruteo	Los componentes de ruteo permiten direccionar el flujo de ejecución de una solución de integración, según las definiciones en los EIPs [10]. El ruteo en base al contenido por ejemplo, determina el destinatario basándose en propiedades del mensaje.
Supervisión	Le permiten a la Plataforma de Integración llevar a cabo tareas de monitoreo sobre los datos intercambiados dentro de la PI.
Mensajería	Le permiten a las soluciones de integración realizar operaciones de mensajería como las definidas en los EIPs.

Tabla 2.5: Categorización de los componentes básicos de una Plataforma de Integración.

2.7 Trabajos Relacionados

RoboMQ

RoboMQ[39] se define como una Plataforma de Integración Híbrida (HIP por su sigla en inglés) desarrollada utilizando microservicios, que provee mecanismos de integración avanzados empleando entre otras herramientas, una Service Mesh sobre Kubernetes [26].

Cuenta con una herramienta denominada *Integration Flow Designer* [27] que permite la construcción de flujos de integración de manera visual, mediante *drag-and-drop* de componentes que mapean uno a uno con microservicios de RoboMQ. Ésta característica respecto a su modo de uso la coloca al alcance de usuarios tanto técnicos como no técnicos.

[39] https://www.robomq.io/about-us/

RoboMQ es un caso de estudio de relevancia en la medida en que se trata de una plataforma de integración basada en Microservicios, a la cual se le ha integrado una Service Mesh sobre Kubernetes. Además cuenta con la herramienta mencionada (*Flow Designer*) que permite crear flujos de integración mediante *drag-and-drop*, una idea que es parte de los posibles trabajos a futuro de la PI del presente proyecto de grado.

Snap

Snap[40], la empresa detrás de la red social *Snapchat*[41] entre otras aplicaciones, es un caso de estudio de interés. En Marzo de 2020 publicaron un resumen del proceso migratorio de su infraestructura en la nube: de lo que anteriormente era un monolítico corriendo en un proveedor de servicios en la nube específico, a una aplicación de microservicios sobre Kubernetes, distribuida entre más de un proveedor y con el agregado de emplear una Service Mesh [31] [32].

En sus publicaciones se establece que el trabajo migratorio transcurrió durante dos años y reporta como beneficios para la empresa, una reducción del 65% en costos de computación con sus proveedores de la nube (lo cual resulta en millones de dólares ahorrados), sin descuidar aspectos de seguridad y privacidad, disminuyendo redundancias y ofreciendo un servicio más confiable.

De la etapa previa al desarrollo de su nueva arquitectura hay conclusiones a las que arriba el equipo de ingenieros de Snap que coinciden con las ventajas de integrar una Service Mesh a una arquitectura de Microservicios analizadas anteriormente. Snap decide que la autenticación, autorización y la seguridad de toda la red no pueden ser un aspecto opcional para cada servicio. A su vez, se pretende alcanzar la mayor separación posible entre la lógica de negocio y la infraestructura, además de centralizar el descubrimiento de servicios y el monitoreo de la aplicación.

Establecidos estos y otros parámetros de trabajo, el equipo de Snap evalúa qué necesidades puede contemplar mediante el empleo de herramientas *Open Source* y que otras debe desarrollar por su cuenta. Éste análisis los direcciona hacia el *Data Plane Envoy* en primer lugar, y luego a decidir desarrollar su propio *Control Plane*, siguiendo el patrón de diseño de *Service Mesh* como un *Data Plane* + *Control Plane*.

La vista final de cómo se ve el tráfico y la interacción con los distintos usuarios puede apreciarse en la Figura 2.20. El componente denominado *Switchboard* es el *Control Plane* de la Service Mesh que Snap decidió desarrollar en lugar de emplear alguna de las opciones del mercado.

[40] https://www.snap.com/es
[41] https://www.snapchat.com/l/es/

Contar con un *Control Plane* propio le permite a *Snap* simplificar las tareas de configuración: al no exponer todos los elementos de la API de Envoy, logran mantener reducida la cantidad de configuraciones a soportar [31]. Otro aspecto a destacar del *Control Plane Switchboard* es que siguiendo el mismo lineamiento de exponer las configuraciones deseadas y estandarizar las restantes, sus capacidades fueron extendidas para ocuparse de tareas como administrar clústeres Kubernetes o despliegues a través de Spinnaker[42], una herramienta de *Continuous Delivery* en la nube.

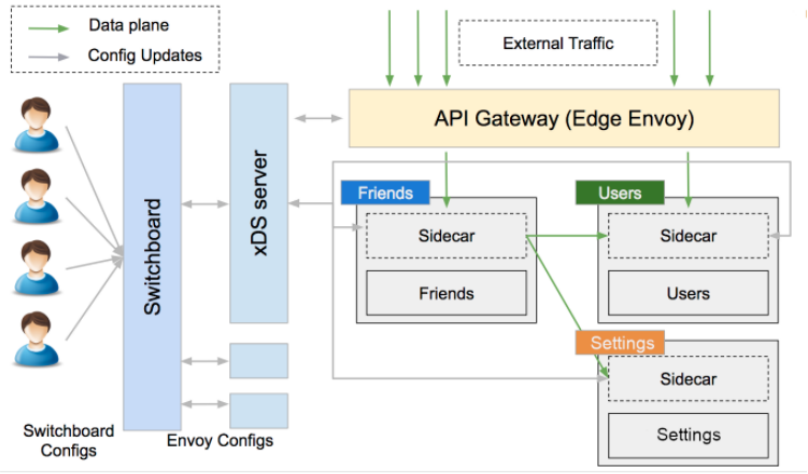

Figura 2.20: Arquitectura de Snap *post* migración: *Switchboard* es su *Control Plane* desarrollado *in-house*, mientras que el *Data Plane* está representado por el deployment de *Envoy* como *Sidecar* junto a cada servicio.

[42] https://spinnaker.io/

3 Análisis de requerimientos

En este capítulo se aborda el contexto de trabajo, se analizan los requerimientos del proyecto de grado y el trabajo a realizar en pos de alcanzar dichos objetivos.

La sección 3.1 abarca el contexto de trabajo. Más específicamente, analiza los componentes, funcionalidades y modos de ejecución de la Plataforma de Integración desarrollada por Bonhomme y Camejo. En la sección 3.2 se detallan los requerimientos iniciales del proyecto de grado. Finalmente la sección 3.3 aborda el alcance y requerimientos finales.

3.1 Contexto de trabajo

El punto de partida de este trabajo es la Plataforma de Integración de propósito general basada en Microservicios de Bonhomme y Camejo, a la que se le debe integrar una Service Mesh.

El trabajo realizado por Bonhomme y Camejo se basa salvo por algunas diferencias, en la arquitectura presentada en la tesis de Maestría de Nebel[43]. Dentro de los requerimientos de su trabajo destacan que la PI debe soportar la creación de Soluciones de Integración en base a componentes de integración. Dichos componentes se coordinan siguiendo los estilos de ejecución de Orquestación y Coreografía descritos en la sección 2.1.

Los componentes de integración se dividen en tres tipos: los Sistemas Externos o aplicaciones cliente, componentes de la Plataforma de Integración y los componentes de las Soluciones de Integración.

La figura 3.1 representa un diagrama de componentes de la PI y a continuación se proporciona una descripción de sus características.

[43] En su trabajo Bonhomme y Camejo dedican una sección a detallar las diferencias de su propuesta respecto de la arquitectura original de Nebel.

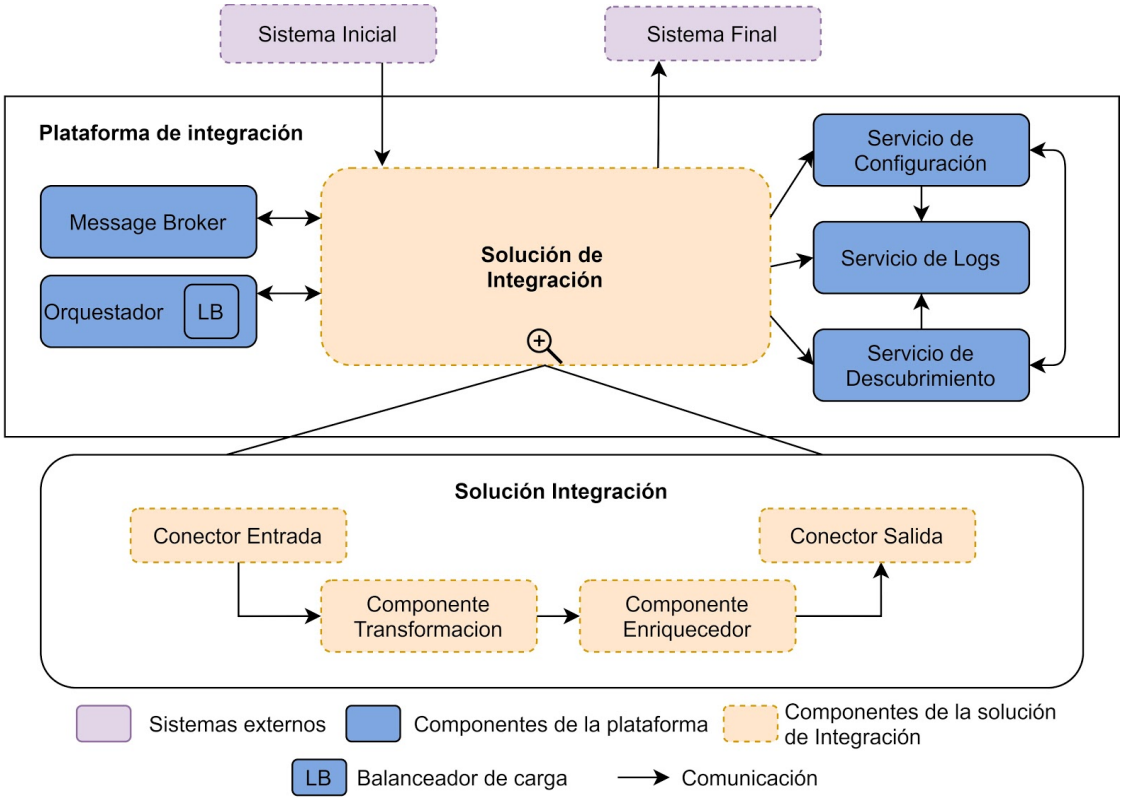

Figura 3.1: Diagrama de componentes de la arquitectura de la Plataforma de Integración.

Aplicaciones cliente (Sistemas Externos)

Los sistemas externos tienen como propósito ejecutar Soluciones de Integración publicadas por la Plataforma de Integración. Estos sistemas se comunican con la PI a través de conectores, o componentes de integración disparadores de una solución. Además, estos sistemas pueden exponer servicios que serán consumidos por Soluciones de Integración. De esta forma, un sistema externo puede por ejemplo enviar un mensaje a la PI, el cual es transformado por una SI y luego es enviado a otro sistema externo. En la figura 3.1 los sistemas externos están representados por “Sistema Inicial” y “Sistema Final”.

Componentes de Integración

Estos componentes forman soluciones de integración. Por un lado se encuentran los conectores o disparadores cuyo objetivo principal es activar soluciones de integración e interactuar con sistemas externos. En la figura 3.1 se observa el disparador “Conector de Entrada” y el conector “Conector de Salida”. Por otro lado se encuentra a los componentes de acción, los cuales efectúan tareas de integración puntuales. Estos ejecutan una acción e intercambian mensajes con el resto de los componentes con el fin de implementar soluciones de integración. En la figura 3.1 estos componentes están representados por el “Componente Transformación” y “Componente Enriquecedor”.

Los componentes de integración, también denominados componentes de la solución de integración, se implementan como microservicios.

Componentes de la PI

A continuación se describen los componentes de la Plataforma de Integración. Éstos representan los elementos de la PI que colaboran con las soluciones de integración que estén ejecutando, realizando tareas de soporte y supervisión; tareas que por otra parte son similares a las que proveen las distintas SM, por lo que son candidatos a ser removidos, una vez integrada la SM.

Servicio de Descubrimiento: También conocido como Registro, el Servicio de Descubrimiento permite registrar y localizar componentes de integración. Los componentes de integración deben registrarse en el servicio de descubrimiento para poder enviar y recibir mensajes. Cuando un componente envía un mensaje a otro componente, primero debe consultar al Registro para obtener la localización del receptor.

Servicio de Configuración Externo: Los componentes de integración obtienen su configuración cuando se están iniciando. Para hacerlo, consultan al servicio de configuración el cual brinda un repositorio y un histórico centralizado de las configuraciones.

Servicio de Logs: Este servicio concentra los *logs* de los componentes de integración, los procesa y los expone en una interfaz que permite visualizarlos y analizarlos.

Broker de mensajería: Permite la comunicación de los componentes a través de mensajes. Se utiliza para implementar comunicación por coreografía.

Balanceadores de carga: Atienden las diferentes comunicaciones entre componentes, repartiendo la carga de peticiones entre las distintas instancias de los servicios ejecutando.

Funcionamiento de la Plataforma de Integración

Para la coordinación de los mensajes entre los microservicios de la plataforma de integración, una solución de integración puede seguir el modelo de Orquestación o Coreografía.

Coreografía: En el caso de coreografía se utiliza un *broker* de mensajería donde el intercambio de mensajes sucede en forma asincrónica. Los componentes de integración no se comunican directamente sino que publican y consumen mensajes en colas de mensajes del *broker* en cuestión. Se utiliza el modelo de comunicación publicador/subscriptor[44] centralizado con intermediarios y el protocolo AMQP. En la figura 3.2 se muestra la comunicación entre componentes para este caso.

[44] Ver Apéndice A: Glosario

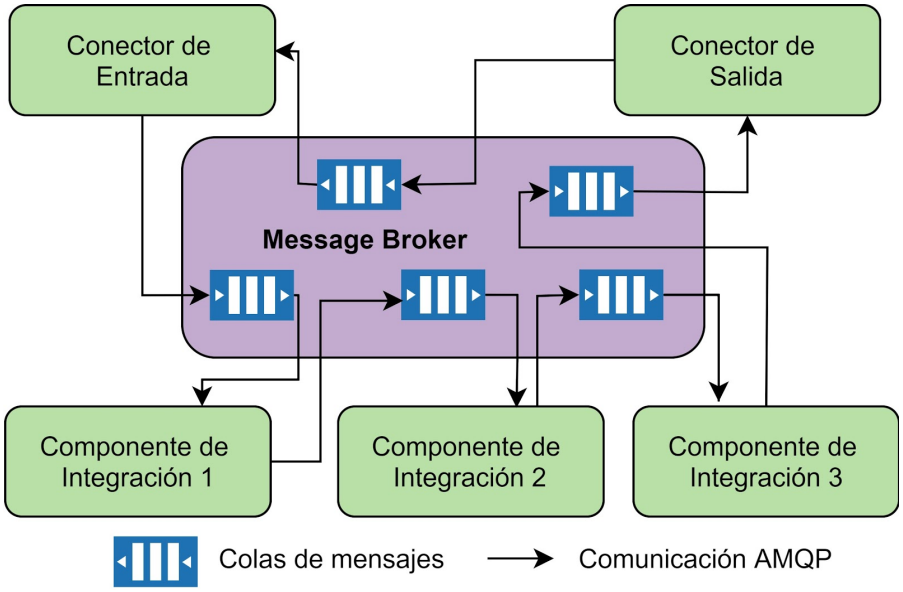

Figura 3.2: Modelo de ejecución para el modo coreografía.

Cada componente de integración ejecuta su lógica interna y publica un mensaje en una cola en particular, el cual será procesado por un componente de integración que se subscribe a dicha cola. El componente receptor ejecuta a su vez su lógica interna, y el proceso se vuelve a repetir sucesivamente hasta completar la ejecución de la SI.

Orquestación: En el caso de orquestación se utiliza un modelo de ejecución sincrónico. El orquestador es el componente que tiene toda la información necesaria para completar la ejecución de una solución de integración e invoca a través de peticiones HTTP a los componentes de integración que requiera.

Como puede verse en la figura 3.3 el conector de entrada recibe la petición inicial del cliente y realiza una llamada HTTP al orquestador. Luego, el orquestador es quien llama a cada uno de los componentes de integración, pasando siempre en el cuerpo de la petición, el mensaje recibido como resultado del componente anterior. El proceso se repite hasta completar la ejecución de la solución de integración.

Escenario de integración

Uno de los requisitos del trabajo realizado por Bonhomme y Camejo[6] consiste en el desarrollo de una Solución de Integración que demuestre las virtudes de la Plataforma de Integración. La misma consiste en la integración de dos sistemas externos: el primero cumple el rol de cliente de la plataforma y se comunica para obtener información de distintas empresas, a partir de sus razones sociales. El segundo sistema externo es quien provee dicha información.

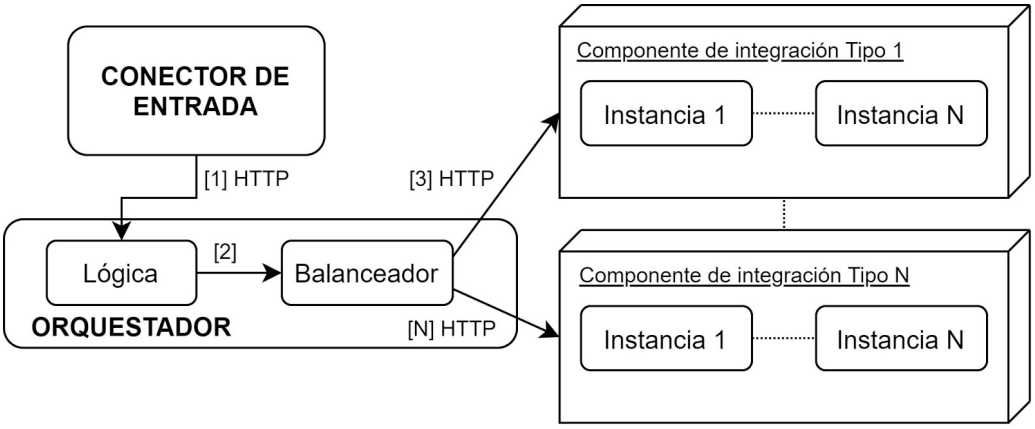

Figura 3.3: Modelo de ejecución para el modo orquestación.

La figura 3.4 desarrolla el caso de uso resuelto por la solución de integración anteriormente mencionada. En él es posible observar cómo el Sistema 1 inicia la ejecución de la solución de integración, que en su ejecución transforma y enriquece el contenido del mensaje original, antes de alcanzar al Sistema 2. La respuesta del Sistema 2 es transformada a su vez, antes de alcanzar al Sistema 1.

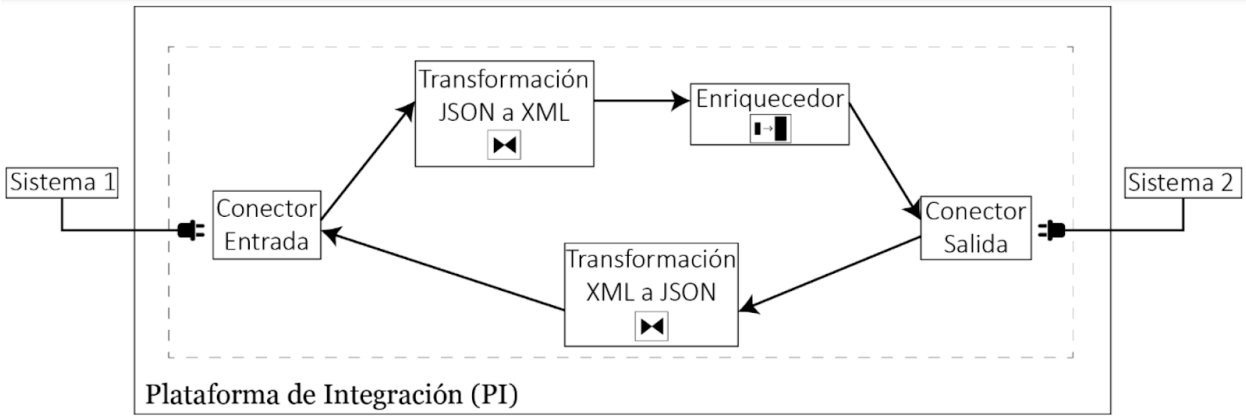

Figura 3.4: Solución de integración recibida.

Funcionalidades de la Plataforma de Integración

A continuación en la tabla 3.1 se puede observar el listado de las funcionalidades que la plataforma de integración soporta. Este listado permitirá analizar qué funcionalidades deben ser migradas al compararlo con las tablas 2.2, 2.3, 2.4 y los requerimientos funcionales de la sección 3.2.

Funcionalidad	Provisto por PI	Observación
Load Balancing	Sí	La PI soporta *Load Balancing* mediante el empleo de *Ribbon*, parte de *Spring Cloud Netflix*[45].
Service Discovery	Sí	La PI soporta *Service Discovery* mediante el empleo de *Eureka*, parte de *Spring Cloud Netflix*.
Trazabilidad	Sí	La PI utiliza *Spring Cloud Sleuth* [46] para brindar Trazabilidad en cada SI.
Logs	Sí	Los microservicios de la PI realizan sus logs a través de *Logback*[47] . A su vez la PI los centraliza y permite su visualización mediante *Graylog*[48].
Configuración Centralizada	Sí	La PI cuenta con un “componente de plataforma” dedicado a esta tarea.

Tabla 3.1: La tabla indica si la funcionalidad es actualmente provista por la PI o no.

3.2 Requerimientos iniciales

Los requerimientos iniciales del proyecto se dividen en las tres áreas detalladas a continuación:

1. Integración de una Service Mesh.
2. Integración de un Orquestador de Contenedores.
3. Implementación de un componente de ruteo y nuevo escenario de integración para validarlo.

Service Mesh

Se desea contar con el patrón de diseño *Circuit Breaker* con el objetivo de detectar fallos, evitar propagación de los mismos y comunicaciones innecesarias entre los servicios. La plataforma no lo ofrece por lo que es una funcionalidad nueva que se decide agregar para aumentar la resiliencia de la misma.

Si bien ya se cuenta con balanceo de carga en la plataforma de integración, es necesario migrar la funcionalidad a la SM con el objetivo de remover la implementación a nivel de microservicios, quitando así la dependencia con la librería utilizada hasta el momento. Además, se busca tener dos algoritmos para el balanceo de carga.

[45] https://spring.io/projects/spring-cloud-netflix
[46] https://spring.io/projects/spring-cloud-sleuth
[47] https://logback.qos.ch/
[48] https://www.graylog.org/

Se requiere que la comunicación entre los componentes cuente con mecanismos de autorización. Para esto deberá ser posible restringir la comunicación a través de configuración la cual indicará qué componentes pueden comunicarse. La comunicación debe ser encriptada (mTLS) y los usuarios de la plataforma deben autenticarse para utilizarla.

Se debe mantener la generación y acceso a los logs de los componentes. Para esto, se deberá centralizar su gestión y visualización, lo que permite realizar distintos diagnósticos.

Por último, se debe mantener la capacidad de contar con configuración centralizada.

La capacidad de testear la plataforma es una nueva funcionalidad con la que se desea contar.

La Tabla 3.2 enumera las funcionalidades que eventualmente debería brindar la Service Mesh a integrar.

ID	Título	Descripción
REQ_FUN_1	Circuit Breaker	La SM debe ser capaz de aplicar *Circuit Breaker*.
REQ_FUN_2	Load Balancing	El balanceo de cargas debe ocurrir a nivel de la SM.
REQ_FUN_3	Service Discovery	Los microservicios de la PI deben poder darse a conocer y descubrir a sus pares a través de la SM.
REQ_FUN_4	Trazabilidad	Mediante el uso de un identificador, debe ser posible reproducir el camino recorrido por cada mensaje dentro de la PI.
REQ_FUN_5	Seguridad (mTLS)	La comunicación a nivel de los microservicios debe implementar *mutual TLS*.
REQ_FUN_6	Seguridad Norte-Sur (autenticación)	La comunicación entrante a la PI debe ser autenticada.
REQ_FUN_7	Seguridad (autorización)	Un servicio debe poder comunicarse con otro sólamente si cuenta con el permiso correspondiente.
REQ_FUN_8	Logs	La SM debe permitir acceder a los *logs* generados por cada microservicio.

ID	Título	Descripción
REQ_FUN_9	Configuración Centralizada	Los microservicios de la PI deben ser capaces de cargar su configuración inicial desde la SM.
REQ_FUN_10	Testing	La SM debe ofrecer herramientas que faciliten el testeo de la PI.

Tabla 3.2: Requerimientos funcionales iniciales para la Service Mesh a integrar.

Orquestador de contenedores

En la Tabla 3.3 se detallan los requerimientos a cumplir por parte del Orquestador de Contenedores.

ID	Título	Descripción
REQ_ORQ_1	Compatibilidad con los contenedores de la PI.	Los microservicios de la PI fueron contenerizados utilizando Docker[49]; se espera que el Orquestador sea capaz de manejarlos.

Tabla 3.3: Requerimientos iniciales para el Orquestador de Contenedores.

Nuevo componente de integración

Se plantea desarrollar un nuevo componente de integración y un escenario de uso que lo involucre. El microservicio en cuestión es un *Content-Based Router*, parte de los *Enterprise Integration Patterns.* Éste tipo de componente de integración permite rutear un mensaje al receptor correspondiente, en base al contenido del mensaje que recibe como puede apreciarse en la Figura 3.5.

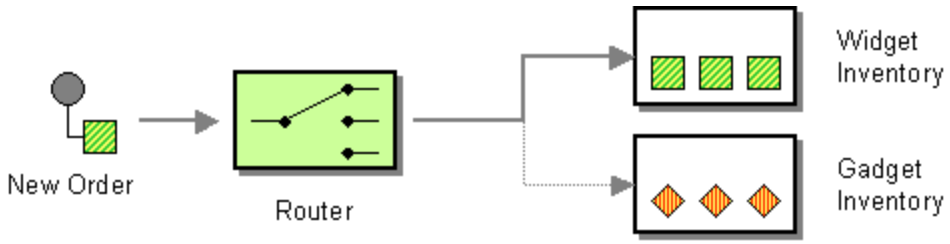

Figura 3.5: Diagrama del *Content-Based Router*.

La Tabla 3.4 muestra los requerimientos iniciales a cumplir por parte del nuevo componente de integración.

[49] https://www.docker.com/

ID	Título	Descripción
REQ_ROU_1	Componente de integración: *Router*.	Se debe implementar el nuevo componente de integración *Content-Based Router*, basado en [9].
REQ_ROU_2	*Router:* separación lógica de sus APIs.	Se debe respetar el diseño propuesto en el proyecto de grado de Bonhomme y Camejo: APIs separadas para recibir los mensajes según se esté utilizando orquestación o coreografía.

Tabla 3.4: Requerimientos finales para el *Content-Based Router.*

Nuevo escenario de integración

Se requiere agregar un nuevo escenario de integración que confirme el correcto funcionamiento de la Plataforma de Integración, a la vez que utilice el componente de ruteo mencionado en la sección anterior. Se destaca que este nuevo escenario de ejecución introduce la complejidad de tener que soportar soluciones de integración con flujos alternativos, los cuales pueden variar según el contenido del mensaje.

El escenario de ejecución a implementar es el siguiente (ver figura 3.6): en base a un mensaje recibido por el router, con una nueva orden de compra para la empresa Antel, éste debe decidir si el mensaje ha de ser dirigido hacia el sistema externo Antel (en caso de tratarse de telefonía fija), Anteldata (en caso de tratarse de un nuevo contrato de internet) o Ancel (en caso de tratarse de telefonía móvil). Que Antel sea el cliente es meramente simbólico, no es parte del alcance del corriente proyecto solucionar ningún problema de integración de una empresa real.

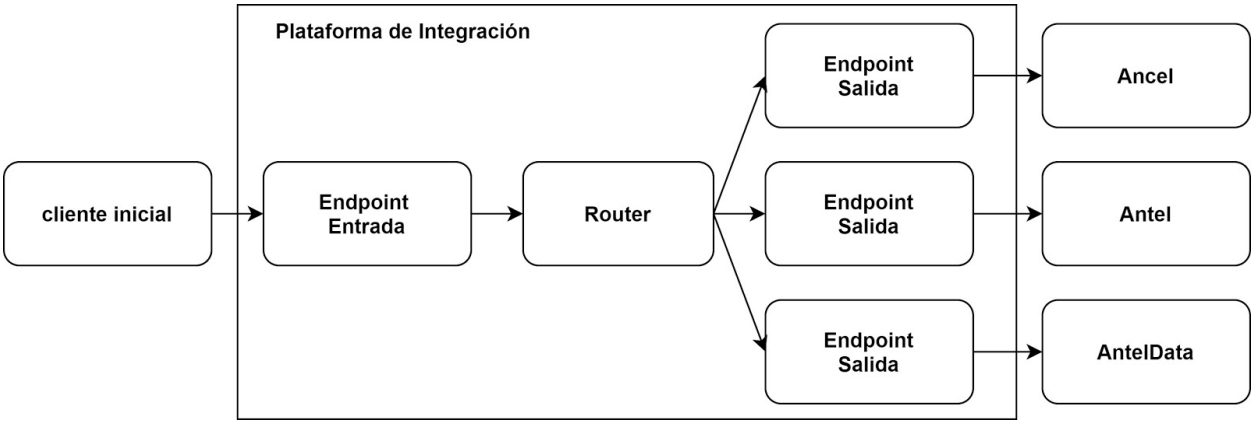

Figura 3.6: Escenario de uso Antel-Anteldata-Ancel. Cuenta con la particularidad de tener un itinerario de ejecución dinámico.

Previo a la existencia del *Router* como componente reutilizable, los flujos de integración de la PI que es posible construir son estáticos, conociéndose desde un principio la secuencia de microservicios a invocar para su resolución. Es sencillo evidenciar esto al ejecutar una SI con Orquestación, ya que el componente que cumple el rol de llevar adelante el escenario de ejecución conoce el itinerario de los microservicios a invocar. A partir del momento en que se permite utilizar un *Router* como parte del flujo, esto deja de ser cierto.

La Tabla 3.5 detalla los requerimientos iniciales del caso de uso. Como simplificación del escenario se acordó que la orden de compra enviada por el Cliente Inicial no puede incluir productos de más de un cliente final.

ID	Título	Descripción
REQ_ESC_1	Escenario Antel-Anteldata-Ancel	La PI debe poder ejecutar el escenario de uso “Antel-Anteldata-Ancel” (Figura 3.6).

Tabla 3.5: Requerimientos iniciales para el nuevo escenario de ejecución.

3.3 Refinamiento, análisis y alcance final

Se realiza una comparación y análisis de las funcionalidades comúnmente encontradas en una arquitectura basada en microservicios (tabla 2.2), service mesh (tabla 2.3) y orquestador de contenedores (tabla 2.4). Estas funcionalidades son contrastadas con los requerimientos funcionales de las tablas 3.2 y 3.3 para determinar las funcionalidades a migrar desde la PI a una solución integral compuesta por la PI, la Service Mesh y el orquestador de contenedores.

Se observa que todas las funcionalidades en 3.1 y todos los requerimientos funcionales en 3.2 y 3.3 se pueden soportar como una colaboración entre la Service Mesh y el Orquestador de Contenedores.

En base a estas observaciones y a conversaciones con el tutor del proyecto, se actualizan los objetivos planteados y se definen aspectos específicos de los mismos.

Service Mesh

Respecto a la lista de funcionalidades originales, se realizan los siguientes ajustes: se detallan características de las funcionalidades *Circuit Breaker* y *Load Balancing.* Se quitan los requisitos iniciales *REQ_FUN_6* y *REQ_FUN_10* y se los considera opcionales. La tabla 3.6 detalla el listado final de requerimientos funcionales.

ID	ID Original	Título	Descripción
REQ_FUN_1_F	REQ_FUN_1	Circuit Breaker	La SM debe ser capaz de aplicar *Circuit Breaker* al fallar una instancia de un servicio. Dicha instancia debe volver a ser chequeada una vez transcurrida una cantidad de tiempo configurable.
REQ_FUN_2_F	REQ_FUN_2	Load Balancing	*Load Balancing* debe ser ejecutado a nivel de la SM, pudiendo configurarse dos algoritmos de balanceo distintos.
REQ_FUN_3_F	REQ_FUN_3	Service Discovery	Los microservicios de la PI deben poder darse a conocer y descubrir a sus pares a través de la SM.
REQ_FUN_4_F	REQ_FUN_4	Trazabilidad	Mediante el uso de un identificador de mensaje, debe ser posible reproducir el camino recorrido por el mismo, dentro de la PI.
REQ_FUN_5_F	REQ_FUN_5	Seguridad (mTLS)	La comunicación a nivel de los servicios debe implementar *mutual TLS*.
REQ_FUN_6_F	REQ_FUN_7	Seguridad (autorización)	Un servicio debe comunicarse con otro sólamente si cuenta con permiso a tales efectos.
REQ_FUN_7_F	REQ_FUN_8	Logs	La SM debe permitir acceder a los logs generados por cada microservicio.
REQ_FUN_8_F	REQ_FUN_9	Configuración Centralizada	Los microservicios de la PI deben ser capaces de cargar su configuración inicial desde la SM.

Tabla 3.6: Requerimientos funcionales finales para la Service Mesh.

Orquestador de contenedores

En la Tabla 3.7 se detallan los requerimientos a cumplir por parte del Orquestador de Contenedores.

ID	ID Original	Título	Descripción
REQ_ORQ_1_F	REQ_ORQ_1	Compatibilidad con los contenedores de la PI.	Los microservicios de la PI fueron contenerizados utilizando Docker; se espera que el Orquestador sea capaz de manejarlos.
REQ_ORQ_2_F	-	Integración con la SM.	El GC debe ser capaz de colaborar con la SM que se integre a la PI.

Tabla 3.7: Requerimientos finales para el Orquestador de Contenedores.

Nuevo componente de integración

La Tabla 3.8 brinda una síntesis de los requerimientos finales a cumplir por parte del componente de integración agregado. Al requerimiento inicial que contemplaba su implementación según el patrón de diseño *Content-Based Router*, se añade la exigencia de que las instancias del *router* puedan ser utilizadas por más de una SI desplegada.

El componente debe ser reutilizable, permitiendo que una misma instancia pueda procesar peticiones de distintos pasos de una SI o de distintas soluciones de integración. Así se podrá optimizar el manejo de los recursos, desplegando réplicas de un mismo componente solo cuando la carga de solicitudes lo amerite, o en caso de que un cliente quiera contar con un componente dedicado para sus soluciones de integración.

ID	ID Original	Título	Descripción
REQ_ROU_1_F	REQ_ROU_1	Componente de integración: *Router*.	Se debe implementar el nuevo componente de integración *Content-Based Router* [9].
REQ_ROU_2_F	-	Reutilización del componente *Router*.	Las instancias del componente *Router* deben permitir ser utilizadas por más de una SI a la vez.

Tabla 3.8: Requerimientos finales para el *Content-Based Router.*

Nuevo escenario de integración

El escenario de uso de Antel se mantiene invariante desde que fuese delineado como requerimiento inicial en la sección 3.2.

Casos de uso y entidades soportadas

A partir de contar con el conjunto de requerimientos finales del proyecto, se construye la tabla de casos de uso de la Plataforma de Integración y las entidades soportadas por la misma.

En la Tabla 3.9 es posible observar la suite de Casos de Uso que soporta la Plataforma de Integración al contar con una Service Mesh.

Caso de Uso		Soporta PI + SM
Gestión de SI	Crear SI	Se permite el agregado de nuevas SI por parte de los desarrolladores.
	Implementar SI	Sí.
Administrar SI	Desplegar SI	Sí.
	Supervisar SI	Gestión y análisis de logs.
	Escalar SI	No automáticamente. Sí manualmente.
	Testear SI	Sí.
	Recuperación PI ante un fallo irrecuperable en un servicio.	Sí: *Circuit Breaker* se encarga de lidiar con instancias de servicios que fallen.
Ejecutar SI	Activar flujo	Sí
	Ejecutar siguiente acción	Sí

Tabla 3.9: Casos de uso de la PI + SM.

En cuanto a las entidades soportadas por la solución compuesta PI + SM, la Tabla 3.10 muestra la evolución de la Plataforma de Integración al contar con los servicios que le brinda la Service Mesh. Además del nuevo componente de integración (*Content-Based Router*), destacan Seguridad, Control y Supervisión de una SI, así como *Circuit Breaker*.

Subsistemas	Entidades	Soporta PI	Soporta PI + SM
Componentes de Integración	Conectores Específicos	No	No
	Conectores Simples	Sí	Sí

Subsistemas	Entidades	Soporta PI	Soporta PI + SM
Componentes de Integración	Mensajería	Sí	Sí
	Ruteo	No	Sí
	Transformación	Sí	Sí
	Componentes Personalizados	No	No
Gestión de SI	Repositorio de SI	No	No
	Despliegue de SI	Sí	Sí
	Control y supervisión de SI	No	Sí
Utilitarios de Servicios	Seguridad	No	Sí
	Virtualización de Servicios	No	No
	Gestor de Logs	Sí	Sí
	Gestor de cache	No	No
	Descubrimiento de Servicios	Sí	Sí
	Balanceador de Carga	Sí	Sí
	Circuit Breaker	No	Sí

Tabla 3.10: Evolución de las entidades soportadas primero por la PI y luego por la colaboración PI + SI.

4 Proceso de migración de la Plataforma

En este capítulo se abordan las distintas etapas del proceso de selección de la Service Mesh y posterior integración a la solución actual.

En primer lugar la sección 4.1 brinda una breve introducción a las Service Mesh más populares. Luego en la sección 4.2, se presenta el proceso de selección de la Service Mesh que será integrada a la Plataforma de Integración. La sección detalla el criterio utilizado para la selección. Finalmente en la sección 4.3, se describe el proceso de migración que se le aplica a la Plataforma de Integración para lograr la integración deseada. También se aborda el rol del Orquestador de Contenedores integrado a la solución.

4.1 Ejemplos de Service Mesh

4.1.1 Consul Connect

En abril de 2014 *Consul* es anunciada por HashiCorp[50] como una herramienta cuya principal utilidad es la de ofrecer *Service Discovery*, junto con configuración remota y chequeos de salud para microservicios. De allí en más, sucesivas actualizaciones de la herramienta extienden la suite de funcionalidades ofrecidas hasta llegar a la inclusión de *Connect*[51]. *Connect* provee por un lado autorización y encriptación para la comunicación servicio a servicio, y por otro dota a la herramienta de observabilidad.

Una de las características sobresalientes de *Consul* es el nivel de interoperabilidad que presenta con otras herramientas [52], al permitir el control individual de cada funcionalidad ofrecida. El *Control Plane* de *Consul* puede ser empleado con el *Data Plane* propio de la herramienta, o utilizando *Data Planes* de terceros como *Envoy*[52].

En la *web* de *Consul* es posible encontrar reseñas de casos de uso interesantes o de clientes notorios como por ejemplo Bloomberg[53] o Mercedes Benz[54].

[50] https://www.hashicorp.com/
[51] https://www.consul.io/docs/connect
[52] https://www.envoyproxy.io/
[53] https://www.hashicorp.com/resources/bloomberg-s-consul-story-to-20-000-nodes-and-beyond
[54] https://www.hashicorp.com/resources/adopting-consul-for-service-discovery-at-mercedes-benz

4.1.2 Linkerd

El primer release de *Linkerd* data de Enero de 2016. En Febrero del mismo año, la versión 0.1.0 es publicada a la vez que pasa a ser un proyecto *Open Source* bajo la licencia Apache License v2 [53]. Un año después ya estaba disponible la versión 1.0.

Linkerd fue construida por dos ex empleados de *Twitter*, sobre la (también *Open Source*) librería *Finagle*, persiguiendo el objetivo de acercarle al público general las ventajas operativas que el equipo de *Twitter* acumuló tras años de trabajar con *Finagle.* Para ello trabajan en la creación de una herramienta auto contenida, con dependencias mínimas y que pueda ser desplegada con un mínimo de esfuerzo, junto a cualquier aplicación ya existente.

Cabe destacar que a partir de Setiembre de 2018 se encuentra disponible la versión 2.0 de *Linkerd*, que constituye una reescritura completa, significativamente más rápida y menos pesada. Ambas versiones continúan siendo desarrolladas por la comunidad y empleadas por clientes como Strava[55] o Paypal[56] entre otros.

4.1.3 Istio

En Mayo de 2017, Google[57], IBM[58] y Lyft[59] anuncian el primer release público de *Istio*, como un proyecto *Open Source* que provee de administración, conexión y seguridad a las aplicaciones de microservicios.

Desde el momento de su anuncio público [54] el equipo de *Istio* plantea claramente el objetivo de la herramienta y el problema que se apresta a resolver: facilitar el desarrollo de microservicios confiables y ligeramente acoplados, mediante una capa de infraestructura que intercede entre los servicios y la red, resolviendo los problemas comunes de transicionar desde aplicaciones monolíticas hacia microservicios (*Service Discovery*, *Load Balancing*, manejo de errores, monitoreo, ruteo y seguridad entre otros).

Un año más tarde, el anuncio de la versión 1.0 de *Istio* da cuenta de su utilización en producción por parte de clientes como eBay[60] o PubNub[61].

[55] https://buoyant.io/resources/how-strava-uses-linkerd-to-avoid-service-outages/
[56] https://linkerd.io/2017/04/25/announcing-linkerd-1-0/
[57] https://www.google.com/
[58] https://www.ibm.com/
[59] https://www.lyft.com/
[60] https://www.ebay.com/
[61] https://www.pubnub.com/

4.2 Proceso de selección de la Service Mesh a integrar

Se decide comenzar por identificar las tres opciones de Service Mesh cuyo uso se encuentre más extendido en la comunidad, sin perder de vista los requerimientos abordados en la sección 3.3. Como resultado, se procede a investigar y comparar Istio[62], Consul[63] y Linkerd-1.0[64].

Al momento de comparar las distintas Service Mesh, Linkerd-2.0[65] ya existía como posible opción. Sin embargo, al tratarse de una reescritura de Linkerd-1.0 en otro lenguaje, no contaba aún con madurez suficiente como para ser tenida en cuenta debido a que ciertas funcionalidades (*Tracing* por ejemplo*)* aún no estaban disponibles [25].

Con las candidatas a integrar identificadas, se resuelve realizar una pequeña demostración del potencial de cada herramienta. El objetivo es poder comprobar rápidamente el nivel de soporte de la mayor cantidad de requerimientos posibles. Las demostraciones de las distintas opciones resultaron lo suficientemente auspiciosas como para considerarlas en la siguiente etapa de evaluación, consistiendo ésta en un análisis de las funcionalidades que cada una incluye.

Para las demostraciones se utilizaron los ejemplos que cada una de las Service Mesh ofrece, complementando con implementaciones de microservicios sencillas. El paradigma de Microservicios y la delegación de responsabilidades hacia la Service Mesh determinan que la lógica de negocio de los microservicios en este caso no necesita ser compleja, por lo que el prototipo resulta ágil, permitiendo enfocar el esfuerzo en las funcionalidades específicas de las Service Mesh.

Metodología de selección

A los efectos de poder comparar la efectividad con que cada Service Mesh implementa los requerimientos del proyecto, se elabora un criterio de ponderación que evalúa cada una de las funcionalidades. Dicho criterio deriva en el sistema de puntos descrito en la tabla 4.1.

En la ponderación se decide dar mayor relevancia a las funcionalidades que la SM soporta y son parte de los requerimientos finales del proyecto. A tales funcionalidades se les otorga 6 puntos cuando no requieren implementación extra, comúnmente conocido como soporte *out-of-the-box*. Cuando requieren de cierta implementación o ajuste recibe 4 puntos. Si la funcionalidad no forma parte de los requerimientos finales, se le otorgan 2 puntos cuando funciona *out-of-the-box* y 1 punto si necesita de una implementación extra o ajuste. Finalmente, si la funcionalidad no es soportada se le asignan 0 puntos.

[62] https://istio.io/
[63] https://www.consul.io/docs/connect/index.html
[64] https://linkerd.io/2017/04/25/announcing-linkerd-1-0/
[65] https://linkerd.io/

Soporta funcionalidad	Requerimiento Obligatorio	Soporte “out of the box”	Puntaje
Si	Si	Si	6 puntos
Si	Si	No	4 puntos
Si	No	Si	2 puntos
Si	No	No	1 punto
No	-	-	0 puntos

Tabla 4.1: Criterio de evaluación que puntúa cada funcionalidad que ofrece una Service Mesh, según la forma en que es soportada.

Una vez acordado el criterio de puntuación a emplear, se confecciona la lista de funcionalidades y características blandas a relevar. Las características blandas aportan información adicional de interés acerca de la SM y serán consideradas en caso de no contar con un claro dominador a nivel de las funcionalidades evaluadas.

Resultado de selección

En la tabla 4.2 se puede observar el resultado resumido de aplicar el sistema de puntos descrito anteriormente[66]. Se encuentra disponible una versión extendida de la misma, como Apéndice C.

En cada columna se indica si la SM satisface la característica con un “Sí” en caso afirmativo, con un “No” en caso negativo y se utiliza “*” si hay observaciones importantes (detalles en Apéndice C). En la columna “Ganador” se muestra cuál de las 3 opciones implementa mejor el requerimiento o empate cuando corresponda.

Característica	Consul	Istio	Linkerd	Ganador
¿Es una service mesh?	Sí	Sí	Sí	Empate
Circuit Breaker	Sí*	Sí	Sí	Istio y Linkerd
Load Balancing	Sí*	Sí	Sí	Istio y Linkerd
Service Discovery	Sí	Sí	Sí	Empate
Tracing	Sí*	Sí	Sí	Istio y Linkerd
Testing	No	Sí	No	Istio

[66] Relevamiento realizado el 15/08/2019

Característica	Consul	Istio	Linkerd	Ganador
Logging	Sí	Sí	No	Istio y Consul
Seguridad: comunicación encriptada entre servicios intra-cluster.	Sí	Sí	Sí	Empate
Seguridad: comunicación encriptada con servicios externos.	Sí	Sí	Sí	Empate
Seguridad: encriptación con cliente (por ejemplo, un *browser*).	Sí	Sí	Sí	Empate
Seguridad: Autenticación intra-cluster, servicio a servicio.	Sí	Sí	No	Consul e Istio
Seguridad: Autenticación de usuarios.	Sí*	Sí	No	Istio
Seguridad: Autorización intra-cluster: qué operaciones puede realizar cada microservicio.	Sí	Sí	No	Consul e Istio
Seguridad: Autorización cliente: que operaciones puede realizar el cliente.	Sí	Sí	No	Consul e Istio
Seguridad: Auditoría.	Sí*	Sí	Sí	Istio y Linkerd
Seguridad: Integridad: los mensajes además de encriptados, llegan intactos? Por ejemplo: md5	Sí	Sí	Sí	Empate
Puntaje total	76	96	74	**Istio**

Tabla 4.2: Análisis (simplificado) de funcionalidades para las Service Mesh. Puntaje adjudicado en base a Tabla 4.1.

Analizando los resultados de la Tabla 4.2 es posible observar que *Consul* no satisface la necesidad de contar con una herramienta que facilite el testeo de la PI, una vez integrada a la SM. Aspectos clave en aplicaciones distribuidas como el balanceo de cargas y la recuperación ante fallas necesitan de trabajo extra por parte de los desarrolladores, para simular situaciones de error que disparen los mecanismos del sistema encargados de lidiar con ellas. Por otra parte, para dar soporte a funcionalidades como *Circuit Breaker* o *Tracing* entre otras, *Consul*

necesita que su *Data Plane* sea intercambiado por *sidecars Envoy*. Según el criterio de evaluación empleado, la dependencia de terceros para otorgar soporte a una cierta funcionalidad es un demérito de la SM.

Al igual que sucede con *Consul*, *Linkerd* carece de una herramienta propia que facilite la tarea de testeo de las aplicaciones que la integren. Otro punto donde no ofrece las mismas funcionalidades que las otras opciones de Service Mesh es a nivel de seguridad.

Para el caso de *Istio*, la totalidad de los requerimientos funcionales existentes son provistos por la colaboración *Istio* + *Kubernetes*, el *stack* por defecto cuando se trabaja con *Istio*.

Si bien el análisis presente en la tabla 4.2 es determinante en la elección de la Service Mesh a integrar, la tabla 4.3 resume el relevamiento de características blandas de las distintas opciones, que hubiese sido empleado para decidir ante un eventual empate.

Usabilidad	**Consul**	**Istio**	**Linkerd**
Panel Administración	Sí	No*	Sí
Visualización trazas	Sí	Sí	Sí
Visualización métricas	Sí	Sí	Sí
Visualización logs	Sí	Sí	Sí
Configuración	**Consul**	**Istio**	**Linkerd**
Interfaz gráfica	Sí	Sí	Sí
Libros	**Consul**	**Istio**	**Linkerd**
Cantidad	0	2	0
Comunidad	**Consul**	**Istio**	**Linkerd**
Difusión	Gitter, Mailing list, Github, Twitter (@HashiCorp)	Discuss, StackOverflow, Slack, Twitter, Github, Google Drive, ServiceMesher	StackOverflow, Slack, Twitter, Github, ServiceMesher, Gitter

Comunidad	Consul	Istio	Linkerd
Frecuencia de liberación	No hay una frecuencia establecida para las liberaciones. Hay publicaciones frecuentes en un blog institucional de Hashicorp de cada liberación importante.	Liberaciones diarias. Protocolo muy claro de liberaciones y control de versiones.	No hay una frecuencia establecida para las liberaciones. Agosto 2019, mantiene un ritmo "constante" de 1 liberación por mes.
Github	Stars: 16913 Releases: 86 Contributors: 515 Licencia: Mozilla Public License 2.0	Stars: 18619 Releases: 71 Contributors: 399 Licencia: Apache-2.0	Stars: 5046 Releases: 76 Contributors: 92 Licencia: Apache-2.0
Socios	Propiedad de Hashicorp	Cisco, IBM Cloud, Huawei, VMWare, RedHat, Pivotal, Google Cloud, entre otros	-
Quien lo usa?	Bloomberg, Stripe, Hello Fresh, Datadog	Ebay, IBM Watson, Trulia, Continental, entre otros.	Monzo, Olark, Zooz

Tabla 4.3: Características blandas, criterio selección de la SM.

Conclusión

En conclusión, las tres opciones proporcionaron una experiencia positiva en la implementación de pequeñas aplicaciones de prueba. Con el objetivo de comparar las funcionalidades que son parte de los requisitos del proyecto de grado se emplea un sistema de puntos. Según este sistema, *Istio* se adapta mejor a los requerimientos de la Plataforma de Integración. Además, se observa que *Istio* cuenta con una comunidad mas activa, con procesos de liberación establecidos, mayor disponibilidad de material y con socios importantes que lo respaldan. Por estos motivos, se escoge a Istio para realizar la migración de la Plataforma de Integración.

La usabilidad refiere al soporte nativo de herramientas con las que es deseable contar, como un panel de administración donde se pueda operar fácilmente con el *cluster* y ver el estado de los distintos despliegues. *Istio* no cuenta con un panel de administración por defecto pero se puede integrar fácilmente con herramientas como *Weave Scope*[67] en *Kubernetes* o *Kiali*[68]. Utilizando *Istio* resulta sencilla la integración para visualización de trazas a través de *Jaeger*[69] *o Prometheus*[70]*,* y *Grafana*[71] para visualización de métricas así como *Kibana*[72] para visualizar *logs*.

El apartado de Configuración refiere a si la Service Mesh posee algún tipo de interfaz gráfica para operar sobre la red. Istio cuenta con *kubectl*[73] (cliente *Kubernetes*) e *istioctl*[74], ambos pueden ser utilizados desde una terminal. Además, a través de *Weave Scope* o *Kiali*, *Istio* cuenta con interfaz gráfica para operar sobre el *cluster*, pudiendo observar configuración existente o aplicar nueva configuración.

4.3 Proceso de migración de la PI a *Istio*

En esta sección se presenta el proceso de migración de la Plataforma de Integración a una integración con *Istio*. Dado que no se ha encontrado ningún ejemplo de migración de una Plataforma de Integración y que además se necesita soportar requerimientos muy específicos al problema, como configuración centralizada, se decide crear un proceso propio que sigue un enfoque no invasivo e iterativo sobre la solución existente.

Este enfoque no invasivo consiste en comenzar migrando aquellas funcionalidades que no requieren cambios significativos en la implementación de la PI. Esto permite tener una buena velocidad al comienzo de la migración pues los desarrolladores no necesitan un conocimiento avanzado del stack tecnológico de la PI. Además, se considera que la migración de

[67] https://www.weave.works/oss/scope/
[68] https://kiali.io/
[69] https://www.jaegertracing.io/
[70] https://prometheus.io/
[71] https://grafana.com/grafana/
[72] https://www.elastic.co/kibana
[73] https://kubernetes.io/docs/reference/kubectl/overview/
[74] https://istio.io/latest/docs/reference/commands/istioctl/

funcionalidades no invasivas sobre la implementación es un buen mecanismo para lograr una primera aproximación de la integración de la Service Mesh con la Plataforma de Integración, pudiendo verificar que las funcionalidades se mantienen correctamente.

El enfoque iterativo permite reemplazar las funcionalidades que se migrarán como pequeños cambios incrementales, lo cual provee dos ventajas. Introducir cambios en forma controlada sobre la plataforma, permite un versionado óptimo del código de la solución así como un versionado de las imágenes de la solución. Se considera la utilización de herramientas como Hyper-V[75] para crear respaldos de cada uno de los pasos de la migración.

A su vez, el enfoque iterativo facilita la detección de errores, permitiendo paralelizar esfuerzos en caso de ser necesario, así como realizar investigación técnica. Ante bloqueantes, permite cambiar la prioridad de las diferentes tareas migratorias.

Primero se decide agregar el Orquestador de Contenedores *Kubernetes* ya que *Istio* tiene una dependencia muy grande con este. Además, al encontrarse la plataforma original contenerizada con *Docker*, resulta sencillo migrarla a un entorno *Kubernetes* con herramientas como *Kompose*[76]. El equipo está familiarizado con *docker-compose*[77] y *Kompose* es una herramienta que permite migrar de *docker-compose* a recursos Kubernetes de forma sencilla. Si bien la transformación no es directa, no se requieren muchos cambios para ajustar la solución.

Una vez que se completa la migración a *Kubernetes* y se verifica que la funcionalidad de la plataforma de integración se mantiene, se procede a chequear que el servicio de descubrimiento por *Kubernetes* funciona correctamente sobre la PI. Para esto se remueve el Servicio de Descubrimiento de la Plataforma de Integración y se ejecuta la solución de integración provista para verificar su correcto funcionamiento.

Luego de verificado el funcionamiento del Servicio de Descubrimiento a través de *Kubernetes*, se procede a eliminar el Servicio de Configuración y se reemplaza con la configuración centralizada provista por *Kubernetes*. Para esto se utiliza *Configmaps*[78].

A continuación, los diagramas de las figuras 4.1 y 4.2 muestran el punto de partida de la Plataforma de Integración a migrar y el resultado de aplicar *Kubernetes* y remover los servicios de configuración y servicio de descubrimiento.

[75] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview
[76] https://kompose.io/
[77] https://docs.docker.com/compose/
[78] https://kubernetes.io/docs/concepts/configuration/configmap/

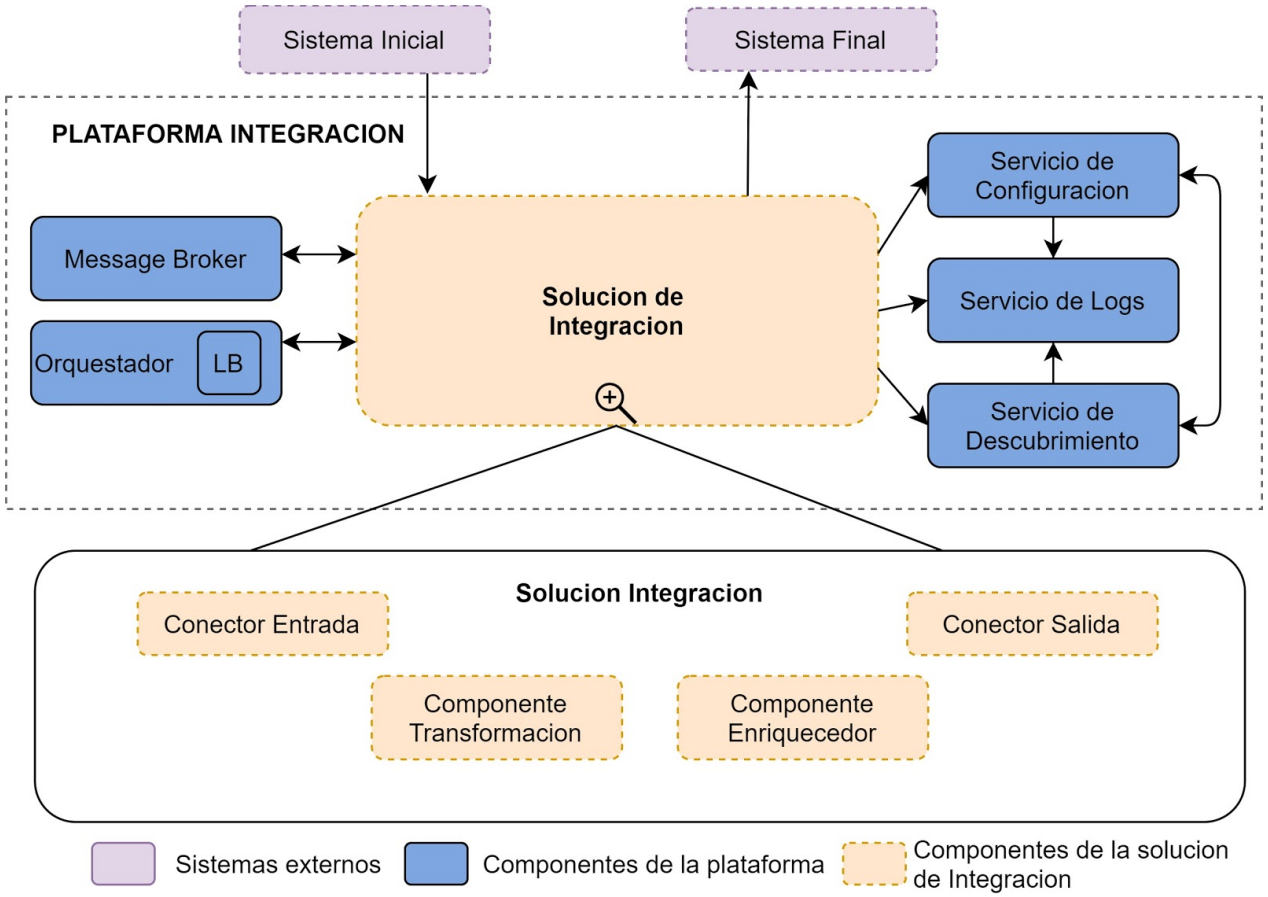

Figura 4.1: Plataforma de Integración original.

En la figura 4.2 se puede observar como la PI comienza a delegar responsabilidades en este caso a *Kubernetes*. El diagrama muestra como el *Message Broker*, el microservicio Orquestador, los microservicios de la solución de integración y el Servicio de *Logs* son gestionados por *Kubernetes*, y por esto se encuentran apilados sobre este. Observar que también se removió el balanceador de carga del microservicio Orquestador pues *Kubernetes* trae su propio balanceador de carga. Una vez removidos estos componentes, se busca integrar la Service Mesh en su configuración por defecto.

En la figura 4.3 se observa que la Service Mesh es apilada sobre el Orquestador de Contenedores. Esta representación permite mantener presente que *Istio* es una aplicación contenerizada: los componentes del *Control Plane* de *Istio* y los *proxies Envoy* son desplegados en *Pods* de *Kubernetes*. De hecho, cada microservicio en la solución de integración se despliega junto a un *proxy Envoy* en un mismo *Pod*, y lo mismo ocurre con el microservicio Orquestador. Cada *Pod* está compuesto entonces por dos contenedores, un microservicio y el *proxy* (o *sidecar*) *Envoy*.

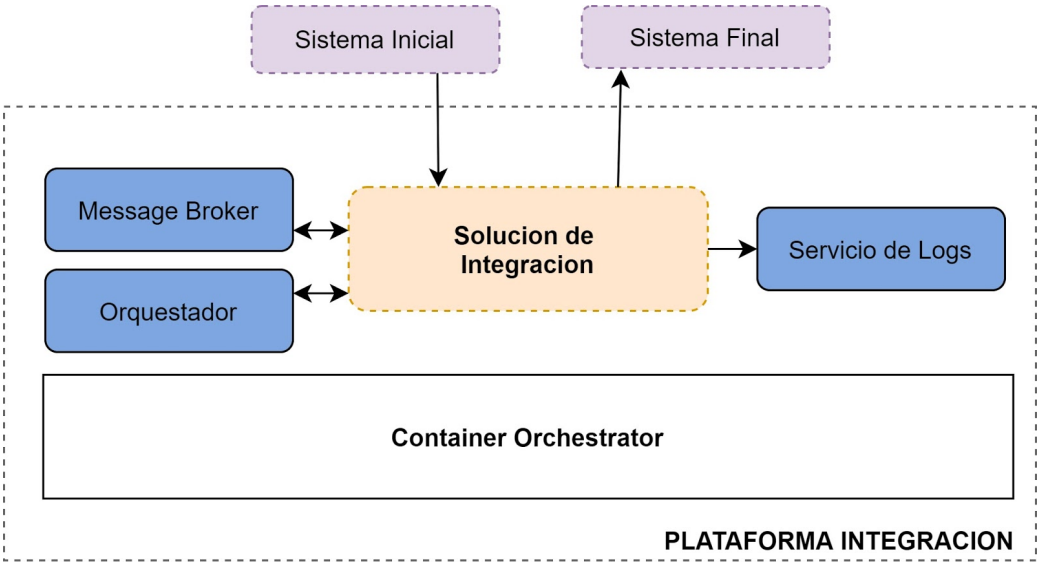

Figura 4.2: Integración *Kubernetes*, eliminación Servicios de Configuración y Descubrimiento originales.

Como los microservicios y los *proxies* conviven en un mismo *Pod*, en la figura 4.3 se observa a la Solución de Integración y el Orquestador contenidos en la Service Mesh, lo cual coincide con las representaciones de Service Mesh introducidas en las figuras 2.12 y 2.13 de la sección 2.2.

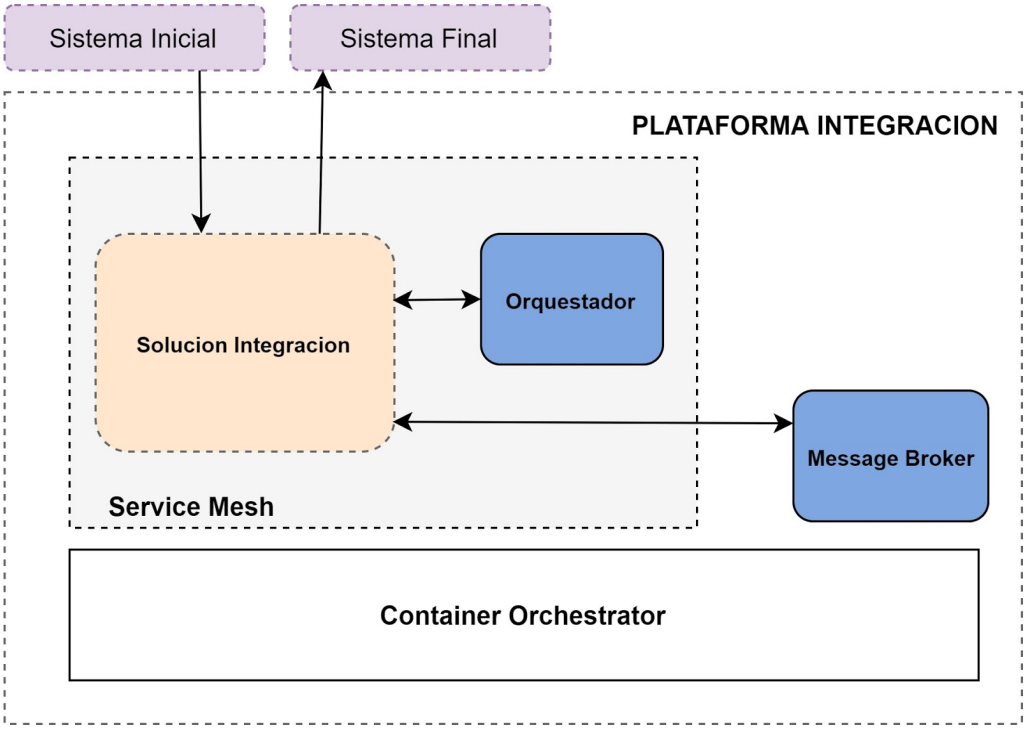

Figura 4.3: Integración de la Service Mesh con configuración por defecto.

Otro aspecto importante a destacar en la figura 4.3 es el hecho de que el *Message Broker* es gestionado por el Orquestador de Contenedores, pero no forma parte de la Service Mesh, es decir, no cuenta con un *proxy Envoy* como *sidecar*. El microservicio Orquestador funciona sincrónicamente, alineado a la naturaleza *request-response* de la Service Mesh y por esto forma parte de la misma. En cambio, la naturaleza asincrónica del *Message Broker* y el escaso soporte de *Istio* y las Service Mesh en general a las comunicaciones basadas en mensajería, derivan en que este componente sea desplegado en *Kubernetes* pero por fuera de la Service Mesh. Esta es una decisión de importancia en el presente trabajo, y que evidencia una de las debilidades y limitantes de las Service Mesh, especialmente importante en un contexto de Plataformas de Integración.

El próximo paso consiste en recuperar las funcionalidad de *log* eliminada anteriormente y utilizar la solución provista por la SM. Además se activa el componente de trazabilidad.

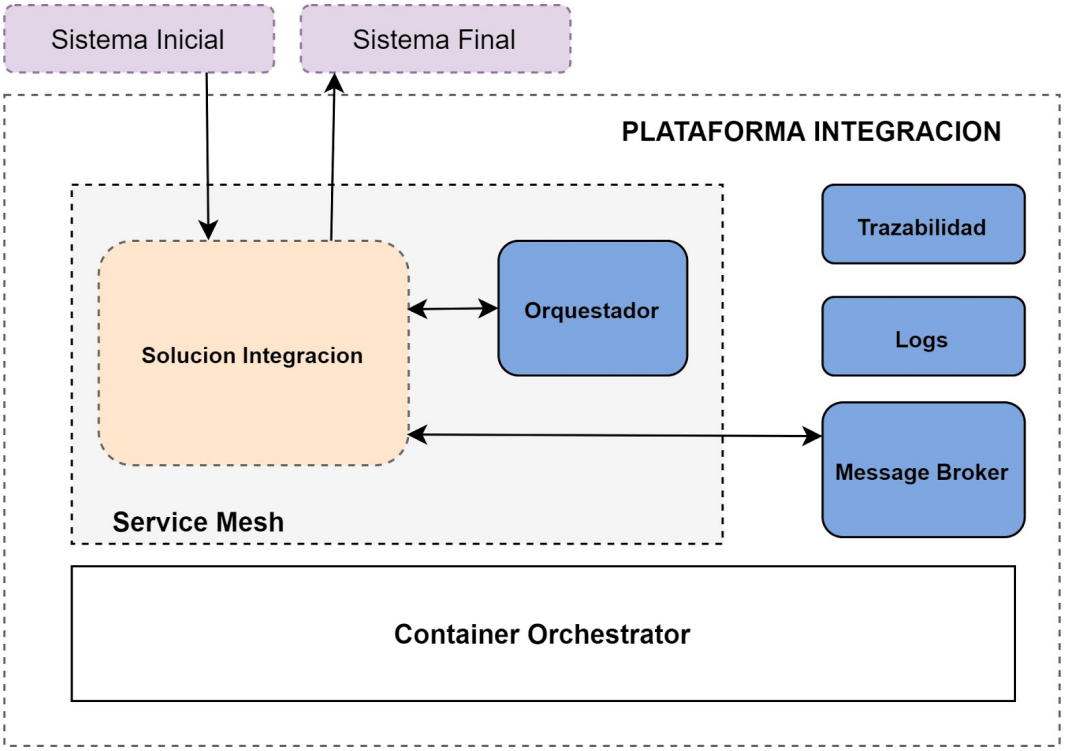

Figura 4.4: Integración de logs y trazabilidad.

En la figura 4.4 se observan los componentes Trazabilidad y *Logs*, gestionados por el Orquestador de Contenedores. Estos componentes no forman parte de la SM ya que solamente se encargan de la recolección de datos y/o visualización de los mismos, por lo que no cuentan con un *proxy* Envoy como *sidecar*. Los *proxies* Envoy y el *Control Plane* envían datos a estos sistemas externos, que se despliegan en *pods Kubernetes*, colaborando para implementar las funcionalidades mencionadas.

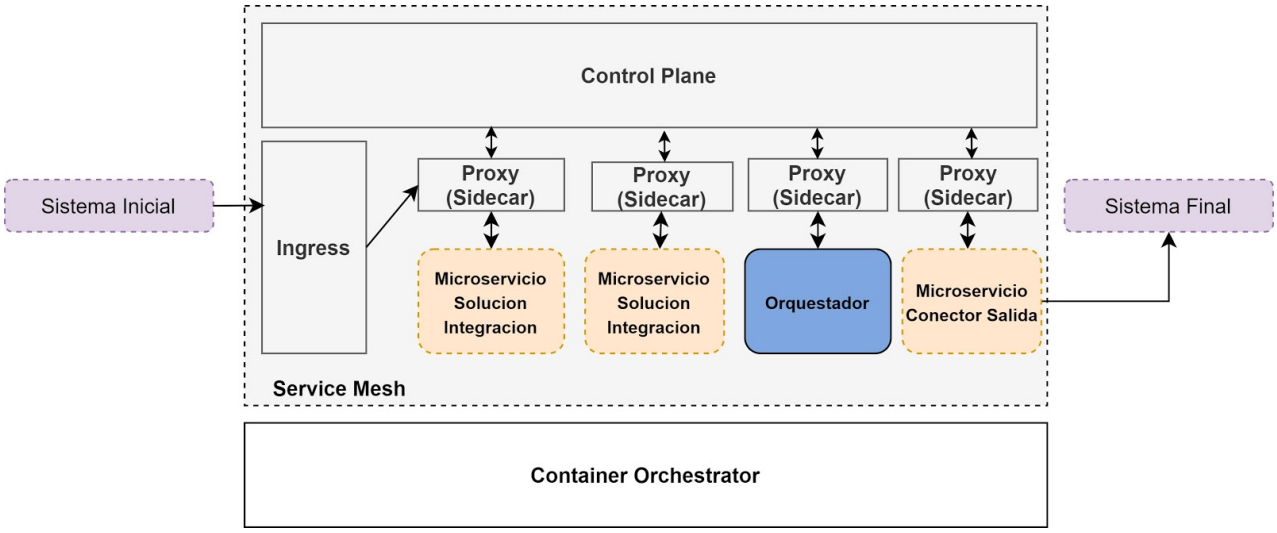

Figura 4.5: Service Mesh, contenido interior.

En la figura 4.5 se han removido componentes para simplificar la representación y lectura. El diagrama permite comprender el rol de los *proxies* en la solución. Estos interceptan la comunicación entre los microservicios *intra-cluster.* Por su parte, el componente *Ingress* que también es un *proxy Envoy*, se encarga de recibir el tráfico entrante al *cluster*. Este es configurado para recibir tráfico externo y redirigirlo al microservicio que corresponda.

La solución final no necesita desplegar más componentes que los que aparecen en la figura 4.5 ya que las funcionalidades de *Load Balancing*, *Service Discovery*, *Circuit Breaker*, *Testing* y *Seguridad* se obtienen aplicando configuraciones a los *Proxies Envoy*. Estas funcionalidades se logran como una colaboración entre el *Control Plane*, los *proxies Envoy*, el Orquestador de Contenedores y las configuraciones que el operador de la SM declara.

El balanceo de carga y el descubrimiento de servicios se puede resumir en la figura 4.6, como tres etapas:

1. La comunicación es interceptada por el *proxy Envoy.*
2. El *proxy* lleva adelante el control de políticas y/o descubrimiento de servicio en colaboración con el *Control Plane*.
3. Una vez obtenida la información necesaria desde el *Control Plane*, se envía el mensaje al *Proxy* que corresponda.

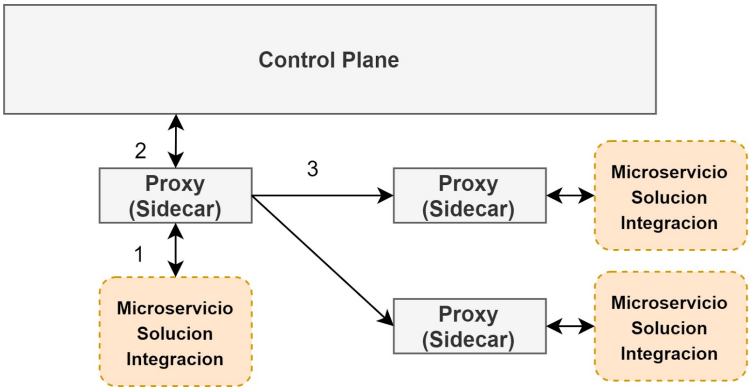

Figura 4.6: Load Balancing y Service Discovery.

Para la implementación de *Circuit Breaker* el razonamiento es análogo, pudiendo visualizarse en la figura 4.7:

1. La comunicación es interceptada por el *proxy Envoy.*
2. El *proxy* controla políticas basado en el algoritmo explicado en la sección 2.1.
3. Una vez obtenida la información necesaria desde el *Control Plane*, se envía el mensaje al *proxy* que corresponda.

Se destaca que el descubrimiento de servicios es una colaboración entre la Service Mesh (*Control Plane*) y el Orquestador de Contenedores. La seguridad es provista por el *Control Plane*, que en el caso particular en *Istio*, sucede a través del componente *Citadel*[79]. La configuración centralizada es provista por el Orquestador de Contenedores a través de *Configmaps* de *Kubernetes*.

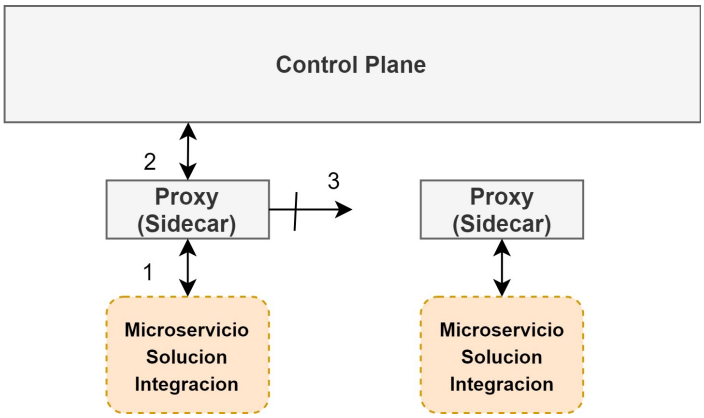

Figura 4.7: Circuit Breaker.

[79] https://istio.io/v1.2/docs/concepts/security/

De esta forma se alcanza la etapa final de la migración, representada en la figura 4.8.

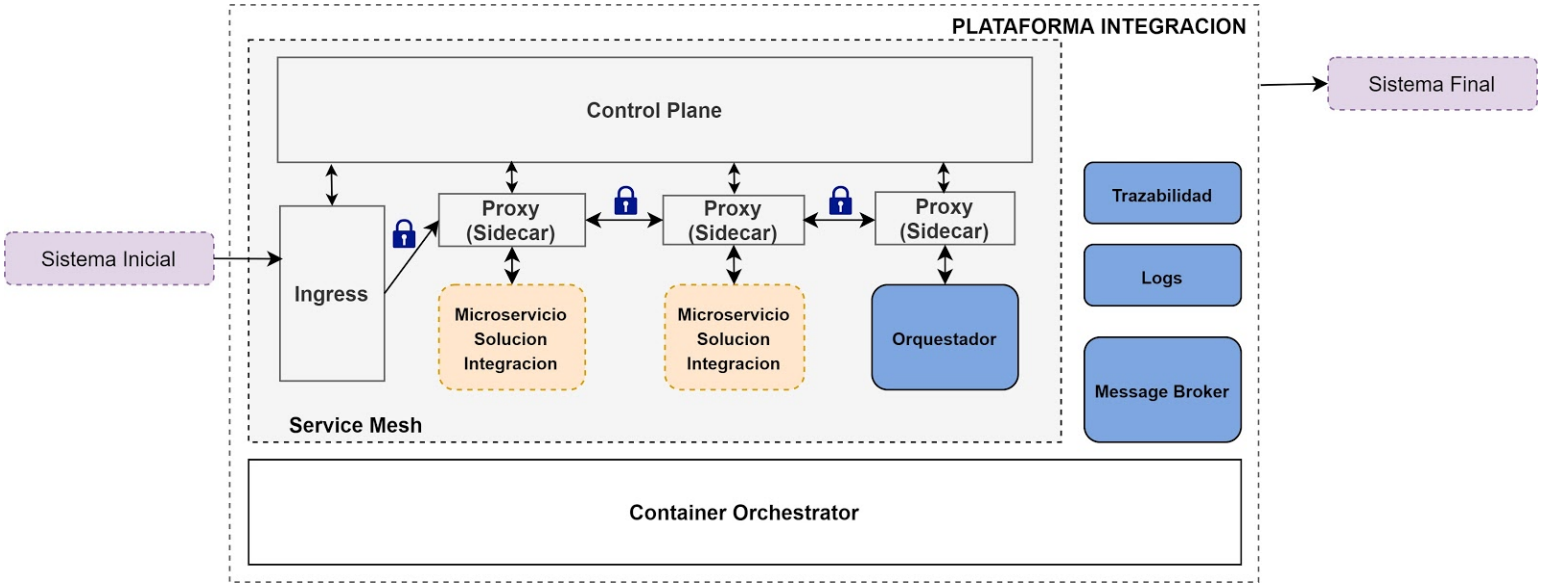

Figura 4.8: Etapa final del proceso de migración de la PI.

5 Nuevo escenario de integración

El objetivo de este capítulo es presentar las decisiones de diseño tomadas para soportar el nuevo escenario de integración introducido en la sección 3.3.

En la sección 5.1 se presenta una visión general de los desafíos enfrentados para la solución del escenario de integración y la SI propuesta. En la sección 5.2 se introducen nuevos conceptos necesarios para comprender la solución y se describe el diseño interno del *router*. En la sección 5.3 se describen las soluciones para los modos de ejecución de orquestación y coreografía. En la sección 5.4 se presentan las soluciones realizadas para los nuevos componentes de integración.

5.1 Descripción general

En el nuevo escenario de integración se simula la recepción de una orden de compra que debe ser enviada al sistema externo Antel, Anteldata o Ancel. Esta decisión debe tomarse basándose en el contenido del mensaje, lo que presenta el desafío de que una SI debe poder tomar flujos de ejecución alternativos.

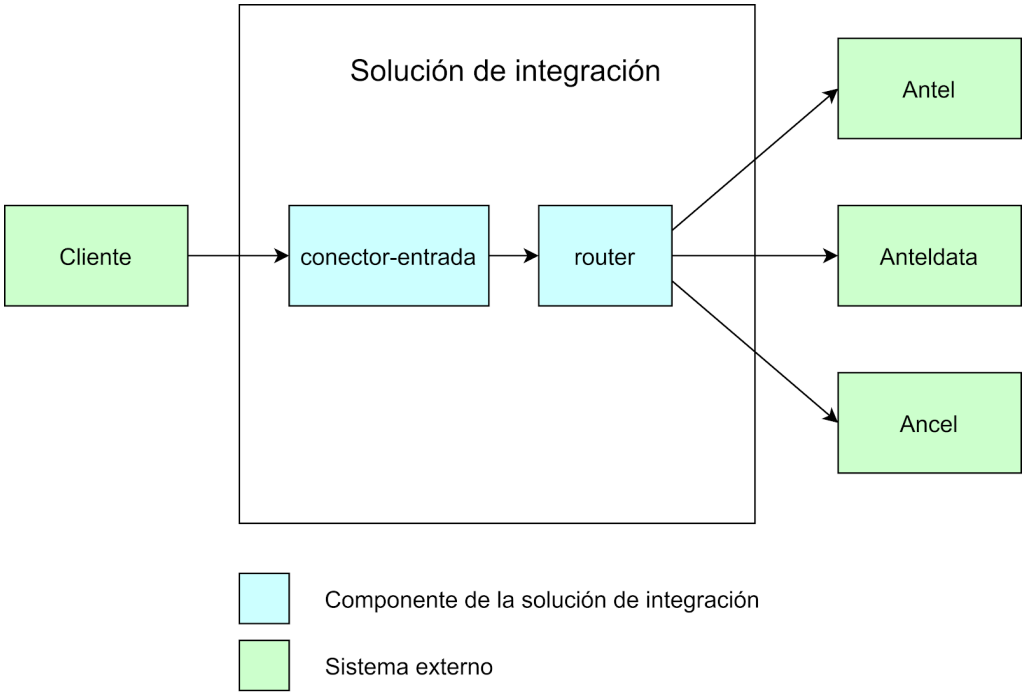

Figura 5.1: Primera iteración de la solución de integración.

Para solucionar el problema de los flujos alternativos se implementa el nuevo componente de integración llamado *Router*, el cual se basa en el patrón *Content-Based Router* de los EIPs.

Como puede observarse en la figura 5.1 solamente con el componente *router* no es posible resolver el problema, ya que el mismo estaría invocando servicios externos y modificando el contenido del mensaje inicial, para satisfacer los requerimientos del servicio externo. Estas tareas exceden las responsabilidades del *router*, por lo tanto se decide agregar dos nuevos componentes a la SI. El “conector-salida-rest” para invocar servicios externos REST y el “modificador-contenido” para modificar el contenido del mensaje.

En la sección 5.4 se detalla el diseño de los nuevos componentes de integración implementados.

En la figura 5.2 se desarrolla la SI propuesta para resolver el nuevo escenario de uso.

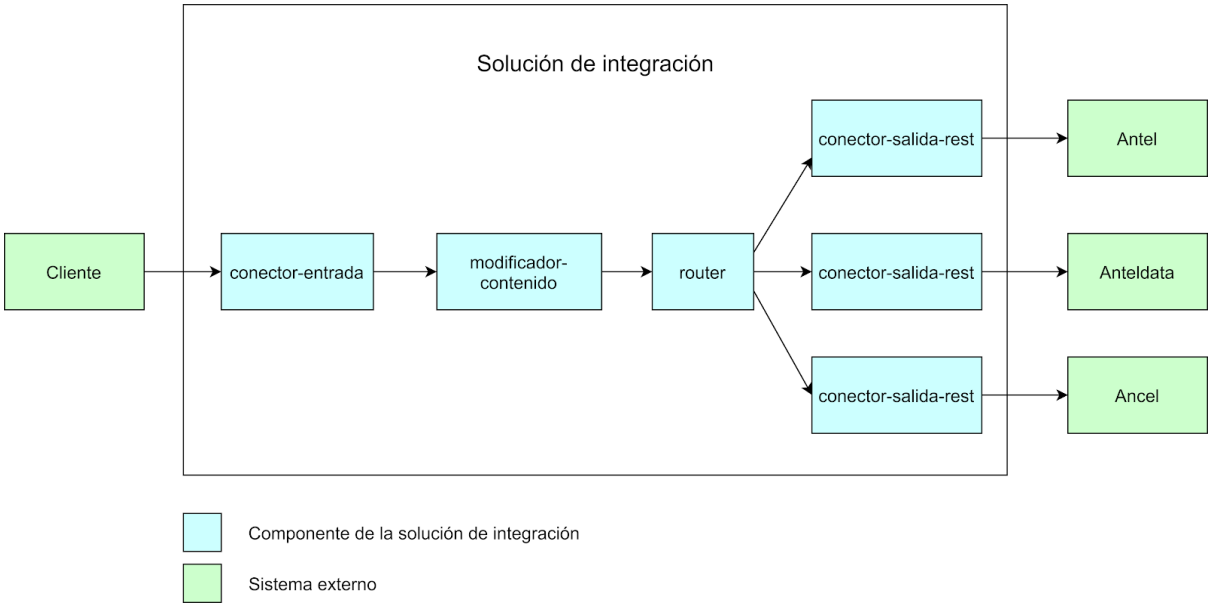

Figura 5.2: Solución de integración para nuevo escenario de uso.

5.2 Diseño interno del Router

En el proyecto de grado de Bonhomme y Camejo se define un itinerario como la secuencia de componentes que se deben ejecutar en una SI. Al no contar con flujos alternativos, en una SI esta secuencia siempre es la misma, lo que significa que el itinerario es conocido desde el momento en que se despliega la SI.

El agregado del *router* permite soportar flujos alternativos, por lo tanto ya no es posible conocer el itinerario completo de una SI al momento de desplegar los componentes. El itinerario se

arma dinámicamente en tiempo de ejecución. Es por eso que se decide manejar distintos itinerarios parciales, los cuales se bifurcan en los *routers*, momento donde se pierde la certeza de cuál es el siguiente componente a ser ejecutado.

En la figura 5.3 se muestra un ejemplo de una SI donde cada color corresponde a un itinerario parcial distinto. Luego se presenta un listado con los posibles itinerarios resultantes de la ejecución en la SI de ejemplo.

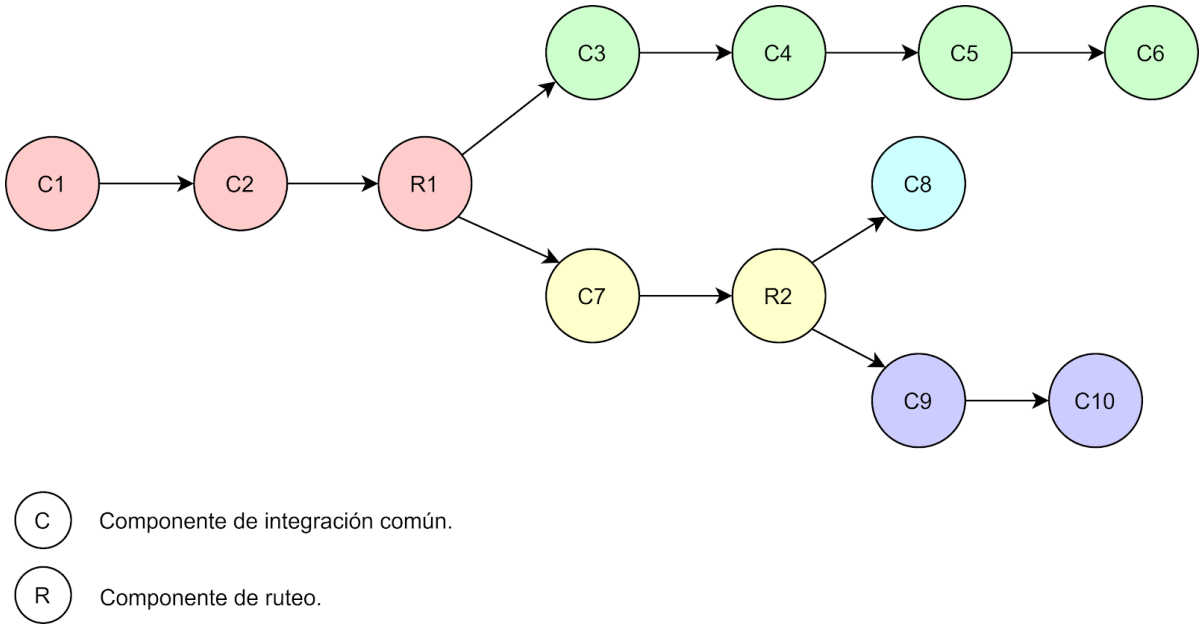

Figura 5.3: Diagrama de ejemplo de una SI con varios itinerarios parciales.

Posibles itinerarios resultantes de la ejecución de la SI:

1. C1, C2, R1, C3, C4, C5, C6
2. C1, C2, R1, C7, R2, C8
3. C1, C2, R1, C7, R2, C9, C10

Como se menciona en la sección 3.3 las instancias del componente *router* deben ser reutilizables, tanto en distintos pasos de una misma SI como por distintas SI. Se decide aplicar el mismo requerimiento a la implementación de los componentes “conector-salida-rest” y “modificador-contenido”. La noción de reusabilidad implica que una misma instancia de un microservicio pueda soportar múltiples soluciones de integración a la vez sin la necesidad de modificación de código o realizar nuevos despliegues.

Para resolver este requerimiento se necesita introducir el concepto de configuraciones de comportamiento, las cuales son los parámetros de ejecución de un componente reutilizable.

A través de estas configuraciones de comportamiento un mismo microservicio podrá comportarse de varias formas sin la necesidad de mantener múltiples réplicas o despliegues y mantener una única versión de código.

Por ejemplo, el componente *router* permite configurar las condiciones a ejecutar sobre el contenido del mensaje y el camino a seguir en caso que se cumpla la condición. En el nuevo escenario de integración se puede ver que el componente “conector-salida-rest” se reutiliza para invocar cualquiera de los tres servicios externos.

La configuración de comportamientos se mantiene en una base de datos que el microservicio puede consultar en tiempo de ejecución y obtener a partir de un identificador de configuración el comportamiento a implementar, específico para la solución de integración en ejecución. En el caso del router, podría obtener los distintos itinerarios que aplican para una solución de integración dada y actualizar el flujo de ejecución en base al mensaje de entrada a la PI. Esta solución implica que se pueden agregar nuevos itinerarios en la base de datos mencionada para definir nuevos flujos de ejecución, sin la necesidad de modificar código o realizar nuevos despliegues del microservicio.

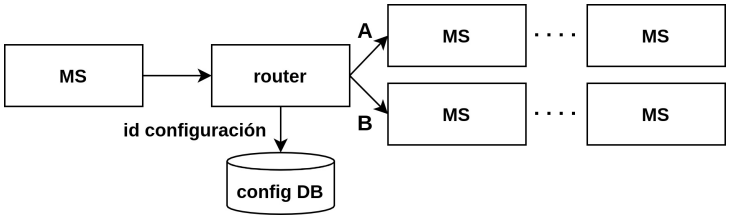

Figura 5.4: Microservicio (MS) router obtiene configuración de comportamiento que determina el camino A o B a partir de un id.

El requerimiento de soportar flujos alternativos, que supone la necesidad de incluir identificadores de las configuraciones de comportamiento en los mensajes enviados a los componentes de integración, impactó en el alcance de la solución. Además de la implementación de los tres nuevos componentes de integración mencionados, y la configuración de la SI para el nuevo escenario de uso, también se realizan ajustes a componentes de la plataforma (conector de entrada, orquestador y *message broker*), para interactuar con estos nuevos componentes.

En la sección 5.3 se describen los ajustes realizados para soportar estos requerimientos.

5.3 Diseño de ejecución

En esta sección se detalla el diseño de ejecución realizado para soportar el nuevo escenario de uso en modo orquestación y coreografía, así como los cambios realizados sobre el formato de los mensajes intercambiados por los componentes de integración.

Orquestación

Como se describió en la sección 3.1 la solución recibida en el modo de orquestación está implementada de forma síncrona mediante llamadas HTTP.

En la figura 5.5 se muestran las invocaciones HTTP realizadas para resolver el nuevo escenario de integración en modo orquestación. Los números indican el orden en que son realizadas las peticiones y se describen a continuación:

1. Invocación inicial desde el sistema cliente al “conector-entrada” para realizar una compra.
2. Se realiza una petición al orquestador para que ejecute la SI.
3. El orquestador invoca al “modificador-contenido” para estructurar el mensaje según el formato que necesita el sistema externo en el paso seis.
4. El orquestador invoca al *router* para hacerse con el resto del itinerario, que obtiene evaluando las condiciones sobre el contenido del mensaje. Las condiciones son cargadas al momento de configurar la SI. Hasta este paso el orquestador no conocía el resto del flujo de ejecución.
5. El orquestador, con el nuevo itinerario provisto por el router, invoca al “conector-salida-rest” para que realice la compra. En este escenario de integración el quinto paso siempre es el “conector-salida-rest”, la diferencia está en la configuración a utilizar, más específicamente qué servicio externo se debe invocar.
6. Llamada al sistema externo de Antel, Ancel o Anteldata.

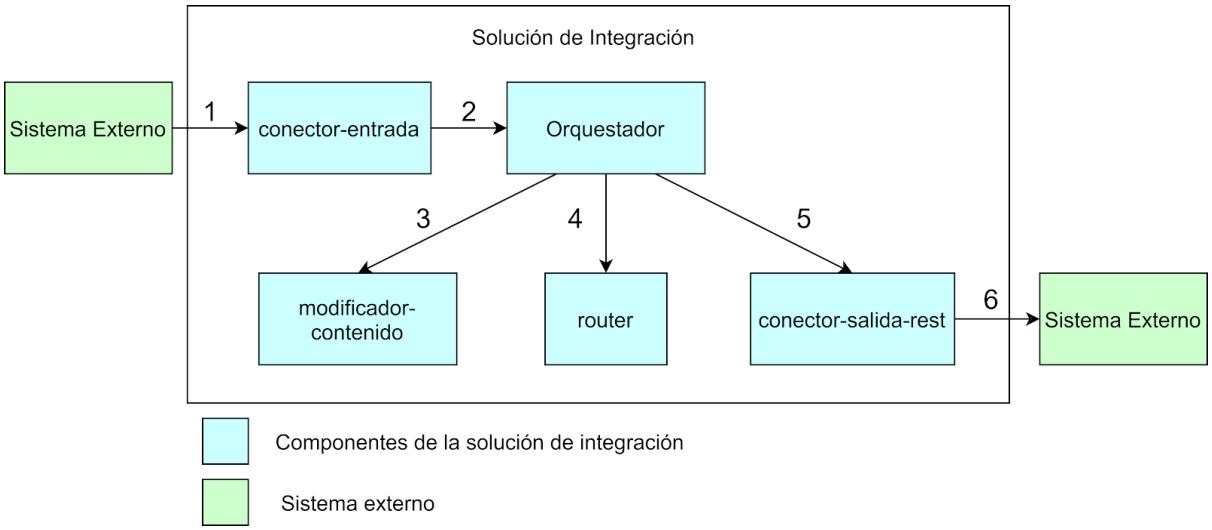

Figura 5.5: Flujo de llamadas HTTP en modo orquestación.

Para resolver este escenario de uso se modifica el componente orquestador ya que no puede contar con un itinerario completo desde el momento en que es invocado, sino que debe ser capaz de actualizarlo cada vez que invoca al componente *router*. Para ello, debe ser capaz de

distinguir cuando está invocando un componente de ruteo para actualizar el itinerario con la respuesta y poder continuar con la ejecución.

El orquestador además de mantener el itinerario que está ejecutando, debe también mantener los identificadores de configuraciones de comportamiento para cada uno de los pasos e incluirlo en el cabezal de las peticiones. Utilizando los identificadores de configuraciones de comportamiento, los componentes obtienen de su base de datos la configuración que deben ejecutar. Por ejemplo el “conector-salida-rest” obtiene la *url*, verbo HTTP y parámetros que debe utilizar en la invocación al servicio REST.

El nuevo itinerario parcial y los identificadores de configuraciones de comportamiento correspondientes a cada paso son devueltos por el *router* en los cabezales de la respuesta HTTP.

Coreografía

En el modo de coreografía cada componente de integración publica un mensaje en la cola del siguiente componente del flujo de integración, utilizando el protocolo AMQP.

Originalmente en la PI cada componente de integración tiene configurado a que cola debe enviar su mensaje luego de procesar el mensaje recibido. A partir de la inclusión del componente *Router,* los itinerarios dinámicos y la reutilización de los componentes, el siguiente componente de integración no siempre es el mismo, sino que depende del contexto de ejecución.
Para saber cual es el siguiente paso de ejecución en cada componente, se mantienen los siguientes cabezales en los mensajes AMQP, como se puede apreciar en la tabla 5.1:

Cabezales	Descripción	Ejemplo
itinerario	Lista con los nombres de los componentes del itinerario en ejecución, separados por coma.	modificador-contenido, *router*
configuraciones	Lista con los identificadores de las configuraciones de comportamiento, separados por coma.	2,3
paso	Puntero al paso de ejecución.	2

Tabla 5.1: Cabezales AMQP en modo coreografía.

Cada componente de integración (CI) debe leer estos cabezales para saber qué configuración obtener del almacenamiento y cuál es el siguiente componente que se debe ejecutar. También debe incrementar en uno la referencia al paso del itinerario. El componente de ruteo debe actualizar los tres cabezales con el resultado obtenido para el nuevo itinerario parcial a ejecutar.

En la figura 5.6 se muestra un ejemplo de una solución de integración y luego en la tabla 5.2, los distintos valores de los cabezales.

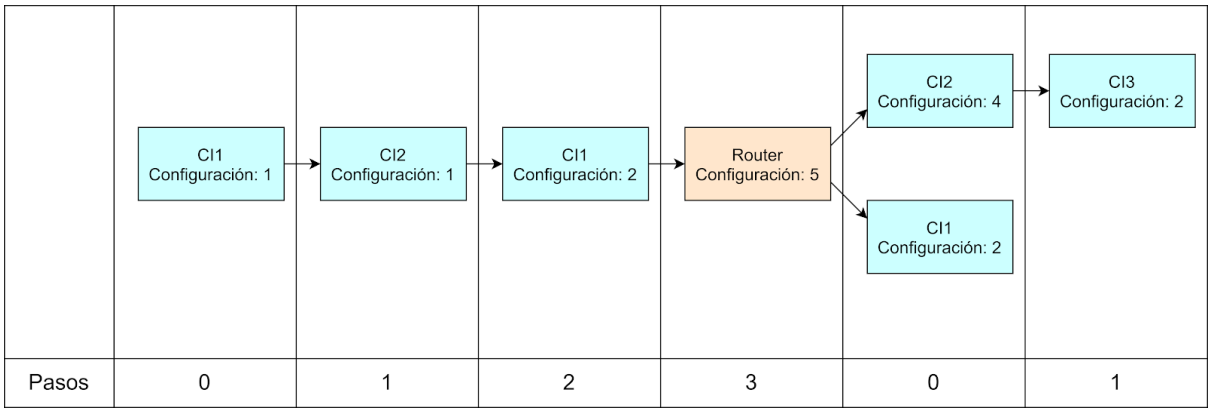

Figura 5.6: Ejemplo de solución de integración.

Cabezales	Valores en primer tramo	Valores en segundo tramo superior	Valores en segundo tramo inferior
itinerario	CI1,CI2,CI1,*Router*	CI2,CI3	CI1
configuraciones	1,1,2,5	4,2	2
paso	Empieza en 0 y se incrementa en cada paso hasta llegar a 3.	0 y 1.	0

Tabla 5.2: Ejemplos de cabezales AMQP en modo coreografía.

Formato de mensajes

El formato del mensaje intercambiado internamente es JSON, y cuenta con tres secciones: *headers*, *exchanges* y *payload*.

En las secciones *headers* y *exchanges* se permite mantener información adicional, con la diferencia que los *headers* son incluidos en los cabezales de invocaciones a servicios externos y los exchanges son solo para el uso interno de la plataforma.

El *payload* es una cadena de texto y el formato del contenido depende de la implementación de cada solución de integración. Por ejemplo, en la SI desarrollada por Bonhomme y Camejo se realizan transformaciones de JSON a XML y viceversa.

Este mensaje, incluyendo las tres secciones, es manejado por cada SI y viaja en el cuerpo de los mensajes HTTP o AMQP, ya sea en modo orquestación o coreografía. Es importante no confundir la sección *headers* del mensaje definido en esta sección con los cabezales AMQP o HTTP utilizados para mantener el itinerario, referencias a configuraciones y referencia al paso en ejecución.

5.4 Diseño de componentes de integración

Para soportar el nuevo escenario de uso, además del nuevo componente ya mencionado (*Router*), se implementan los componentes “conector-salida-rest” y “modificador-contenido”.

El “conector-salida-rest” se encarga de realizar invocaciones a servicios REST. Se puede configurar el método HTTP, URL del servicio y el mapeo de los parámetros en la URL.

El cuerpo de la petición al invocar al servicio REST es el mismo que se tiene en el *payload* del mensaje interno de la SI. Dado que este contenido no siempre coincide con el cuerpo esperado por el servicio REST, se decide implementar el componente ”modificador-contenido”. De esta manera se logra mantener claramente divididas las responsabilidades de cada componente.
Las configuraciones del “modificador-contenido” consisten en una lista de operaciones que permiten modificar el cuerpo del mensaje, *headers* o *exchanges*.

El objetivo del componente “modificador-contenido”, como el nombre lo indica, es permitir la modificación de un mensaje. Para ello se pueden configurar una secuencia de operaciones a ejecutar sobre el mismo.

Los tipos de operaciones implementados son los siguientes:

- AGREGAR_JPATH_EXCHANGE: Se obtiene un valor del *payload* y se agrega como un *exchange*.
- AGREGAR_JPATH_HEADER: Análogo al anterior pero para un *header*.
- AGREGAR_VALOR_EXCHANGE: Agrega un valor configurado como *exchange*.
- AGREGAR_VALOR_HEADER: Agrega un valor configurado como *header*.
- MODIFICAR_PAYLOAD: Modifica el *payload* del mensaje, utilizando valores de los *headers*, *exchanges* y el mismo *payload*.
- REMOVER_EXCHANGE: Remueve un atributo de la sección *exchanges*.
- REMOVER_HEADER: Remueve un atributo de la sección *headers*.

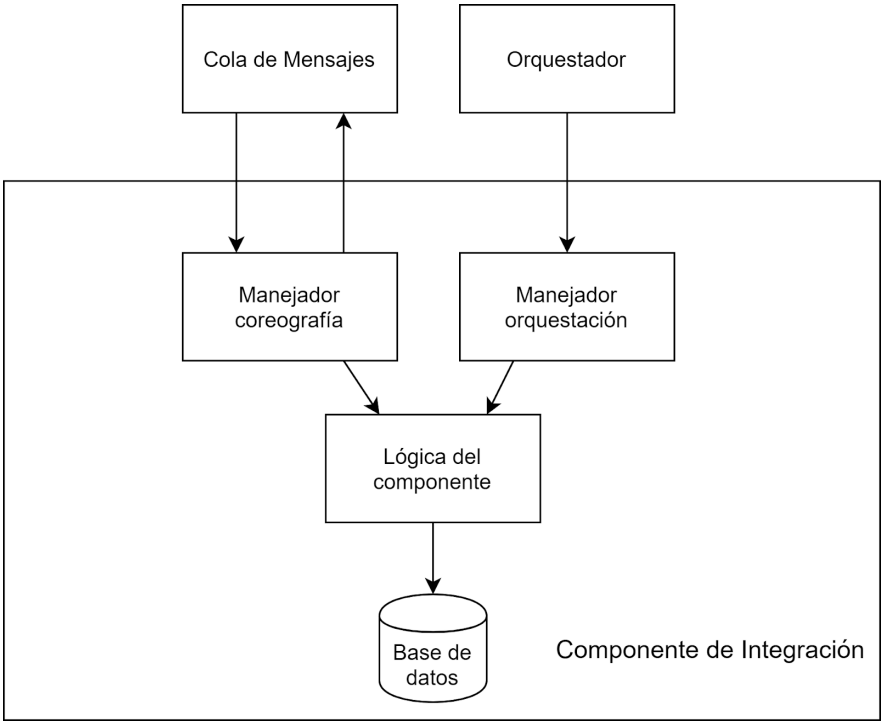

Figura 5.7: Arquitectura interna de un componente de integración.

Como se puede ver en la Figura 5.7 se respeta la arquitectura interna de los componentes implementados por Bonhomme y Camejo, donde los nuevos componentes tienen una lógica centralizada que se invoca internamente en ambos modos de ejecución (orquestación y coreografía). El primer paso de la lógica del componente siempre es obtener la configuración de comportamiento de la base de datos, para luego ejecutar la lógica propia con los parámetros obtenidos.

Una configuración de ruteo está compuesta por un identificador y un conjunto de reglas. Cada regla define una condición que será ejecutada sobre el contenido del mensaje recibido por el *router* y una lista identificando el siguiente itinerario parcial a ejecutar. Recordar que debido a que los componentes de integración son reutilizables, no alcanza con el nombre del microservicio para saber qué ejecutar, sino que también se necesita un identificador de la configuración. Por lo tanto el itinerario parcial almacenado para cada condición de ruteo incluye una lista de pares con el nombre de componente y el identificador de la configuración.

6 Implementación

En este capítulo se analiza tanto la configuración de Istio y Kubernetes que le permite a la PI contar con las distintas funcionalidades requeridas, como el desarrollo de los nuevos microservicios que fue necesario implementar.

En primer lugar, en la sección 6.1 se analizan aspectos generales referentes a la integración de Istio y Kubernetes a la PI.

Luego en la sección 6.2 se abordan aspectos relevantes de la configuración y puesta en funcionamiento de las distintas funcionalidades descritas en la sección 2.3 y que forman parte de los requerimientos enumerados en la sección 3.3

Finalmente la sección 6.3 analiza el desarrollo de los nuevos componentes de integración: el *Content-based Router* agregado en base a los requerimientos detallados en la sección 3.3, el Modificador de Contenido que permite modificar el contenido de los mensajes que recibe, y el Conector de Salida REST, microservicio que permite invocar servicios REST externos a la PI.

6.1 Configuración de Istio y Kubernetes

Aspectos Generales

La solución desarrollada emplea Minikube[80], herramienta que permite desplegar un *cluster* Kubernetes de un único nodo. Esto implica que todo el *cluster* reside en este único nodo o máquina y por lo tanto, todos los microservicios de la PI se despliegan en un único lugar.

Minikube tiene una curva de aprendizaje baja; permite experimentar con un *cluster* Kubernetes de forma rápida y sencilla. El número de funcionalidades que ofrece en comparación a Kubernetes es reducido, pero las pruebas de concepto realizadas al comienzo del proyecto, sumado a la gran cantidad de recursos de estudio disponibles y guías en el sitio oficial de Istio, hacen que sea una herramienta ideal para configurar un nodo donde desplegar la solución del proyecto.

Antes de optar por Minikube se analizaron otras alternativas. Se logró configurar un *cluster* Kubernetes con múltiples nodos utilizando Kubeadm[81] exitosamente. Sin embargo, una configuración con Minikube resulta más rápida y sencilla.

Se utiliza Minikube versión 1.3.1, Kubernetes versión 1.14.2 e Istio 1.2.5.

[80] https://minikube.sigs.k8s.io/docs/
[81] https://github.com/kubernetes/kubeadm

En la figura 6.1 es posible observar los componentes que existen en un *cluster* Kubernetes. Destaca el API server, una API Rest que permite a clientes manejar el estado del *cluster*. Uno de estos clientes es Kubectl, la herramienta de línea de comandos brindada por Kubernetes.

Istio se despliega sobre el *cluster* y su panel de control trabaja cooperando con el API server de Kubernetes.

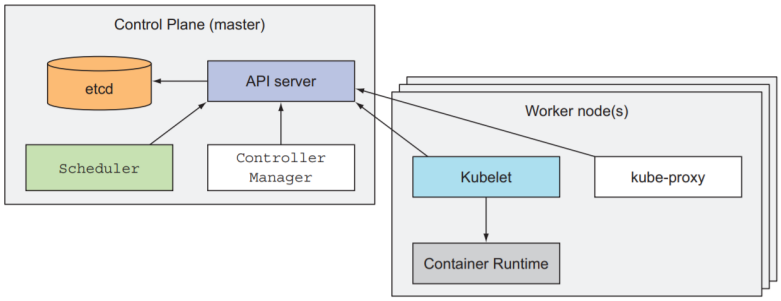

Figura 6.1: Componentes presentes en un *cluster* Kubernetes.

Gran parte de la implementación de la solución se basa en la configuración del *cluster* Kubernetes utilizando kubectl para interactuar con el API server. Istio extiende a Kubernetes y define un conjunto de recursos a través de CRDs[82] (Custom Resource Definition), permitiendo agregar un conjunto de funcionalidades y configuraciones extras.

A través de las configuraciones disponibles en *Kubernetes* y los CRDs definidos por Istio es posible crear *pods*, lo cual resulta fundamental para el despliegue de microservicios, descubrimiento de servicios, centralización de configuración y configuración de *proxies* Envoy, entre otros.

A tales efectos es necesario entender los tipos de configuraciones que es posible declarar y los comandos o peticiones correspondientes que se envían al API server para hacerlo.

La figura 6.2 da cuenta de la forma en que las configuraciones anteriormente mencionadas son recibidas y procesadas a nivel de Kubernetes e Istio:

[82] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

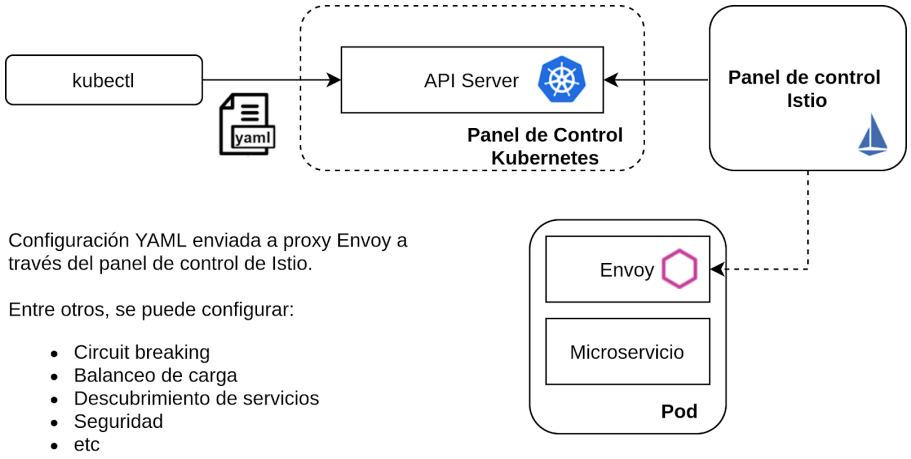

Figura 6.2: A través del API server, las distintas configuraciones tanto para Kubernetes como para Istio son cargadas en archivos YAML.

1. Se le entrega la configuración a cargar al API server de Kubernetes, escrita en formato YAML.
2. Si la configuración es de relevancia para Istio, ésta será interceptada por su panel de control.
3. En caso de ser interceptada, la configuración se procesa y se envía a los *proxies* Envoy para actualizar su comportamiento. Además de aplicarse a nivel de los *proxies*, también puede generar cambios a nivel del panel de control de Kubernetes e Istio.

En la tabla 6.1 se enumeran las distintas configuraciones y recursos más relevantes, que pueden ser introducidas tanto para Istio como Kubernetes, a través del API server. Por su parte la tabla 6.2 lista las distintas tecnologías utilizadas originalmente en la PI, su función dentro de la solución, analizando a su vez si las mismas se mantienen o se remueven tras la integración de Istio y Kubernetes.

Nombre	Plataforma	Descripción
Deployment	Kubernetes	Constructor de alto nivel que se utiliza para desplegar aplicaciones y actualizarlas de forma declarativa.
Service	Kubernetes	Recurso para configurar un único e invariable punto de entrada a un grupo de *pods* que ofrecen un mismo servicio. Cada uno tendrá una dirección IP y puerto que no cambia mientras el servicio exista.
Gateway	Istio	Configura el tráfico entrante y saliente de la red. Esta configuración se aplica a los *proxies* Envoy desplegados en la frontera de la red.

Nombre	Plataforma	Descripción
VirtualService	Istio	Permite configurar cómo es dirigido el tráfico hacia un servicio determinado. Cada *VirtualService* consiste de un conjunto de reglas de ruteo que son evaluadas en orden y contrastadas contra la configuración de la red.
Destination Rule	Istio	Evaluados luego de las reglas de *Virtual Service*, configurando lo que sucede con el tráfico para el destino determinado. De esta forma, permiten aplicar estrategias de balanceo de carga, *circuit breakers* o TLS, entre otros.

Tabla 6.1: Configuraciones y recursos más relevantes de Istio y Kubernetes.

Nombre	Descripción	Descripción
Spring Boot	Permite crear aplicaciones Spring de manera ágil.	Se mantiene.
Spring Cloud Netflix	Provee integración entre aplicaciones Spring Boot y la suite NetflixOSS[83].	Se remueve.
Spring Cloud Config	Permite construir la entidad Configuración Externa Centralizada.	Se remueve.
Spring Cloud Sleuth	Permite implementar el patrón de trazabilidad distribuida.	Se remueve.
Spring Cloud Stream y Spring Integration	Spring Integration extiende el modelo de Spring para soportar EIP. Spring Cloud Stream permite la implementación en base a mensajería e integración con el *broker* de mensajería.	Se mantiene.
Spring Cloud Sleuth	Permite implementar el patrón de trazabilidad distribuida.	Se remueve.
Graylog	Centralizador de *logs* generados por los microservicios.	Se remueve.

[83] https://netflix.github.io/

Nombre	Descripción	Descripción
Logback	Librería que permite generar *logs* con información de los microservicios.	Se mantiene.
RabbitMQ	*Broker* de mensajería.	Se mantiene.

Tabla 6.2: Tecnologías utilizadas originalmente en la plataforma de integración: algunas permanecen y otras se quitan al integrar la Service Mesh.

Despliegue de microservicios

Cada microservicio es desplegado en un *pod* junto a un *proxy* Envoy siguiendo el patrón de diseño *sidecar*. Para desplegarlos se utiliza el recurso Deployment de Kubernetes, el cual permite definir una plantilla en base a la cual se crea un *pod* con los contenedores correspondientes.

Por cada despliegue de microservicio se crea un Deployment Kubernetes. En el archivo YAML de configuración correspondiente, se realizan las declaraciones detalladas en la tabla 6.3.

Nombre	Valor	Descripción
kind	Deployment	Le indica al API server que se quiere crear/actualizar un *pod*.
replicas	1	Cantidad de instancias desplegadas por microservicio.
labels.app	<nombre>	Mapa <clave:valor> que permite categorizar y organizar recursos. Por ejemplo, permite agrupar *pods* bajo un mismo nombre para poder realizar descubrimiento de servicios y direccionamiento de tráfico mediante un Service Kubernetes.
Variable de entorno	SPRING_APPLICATION_JSON	Utilizada para configuraciones de la aplicación Spring.
volumen	“/opt/config”	Se monta un volumen con configuración referente a Logback. Cada microservicio se inicia utilizando la bandera --logging.config referenciando al archivo “/opt/config/logback-spring.xml”
ports.containerPort	<port>	Define el puerto a exponer por la aplicación.

Tabla 6.3: YAML para un Deployment Kubernetes.

Cada solución de integración se despliega sobre un *namespace.* Esto permite una división lógica de las soluciones que se despliegan en el *cluster*, mejorando la organización, autorización y observabilidad sobre cada una de las soluciones que se desplieguen. Para el corriente proyecto se crean los *namespaces* “solucionintegracion1” y “solucionintegracion2”.

Se activa sobre cada *namespace* la inyección automática del *proxy* Envoy en cada *pod.* Cada vez que se realiza un despliegue sobre un *namespace* debemos especificar solamente un contenedor por *pod*. La inyección automática se encargará de agregar el *proxy* Envoy al *pod* creado.

6.2 Configuración de funcionalidades

A continuación se describe la configuración de dos de las funcionalidades requeridas. Las funcionalidades trazabilidad y logging son las que requieren mayor implementación por lo que resulta interesante describirlas en esta sección. Para más detalles sobre el resto de las funcionalidades ver Apéndice F.

Trazabilidad

Istio provee una instancia de Jaeger *out-of-the-box* como sistema que permite el almacenamiento, la agregación y la interpretación de trazas. También brinda la posibilidad de utilizar Zipkin con el mismo propósito. Se decide mantener Jaeger debido a la abundancia de material y ejemplos de su uso junto a Istio.

Cuando una petición ingresa a la Service Mesh el primer *proxy* Envoy en procesarla se encarga de generar un conjunto de *headers* necesarios para que la trazabilidad sea posible, siguiendo los principios de OpenTracing. Bajo estos principios, Envoy se encargará de generar el *header* “x-request-id” si este no existe al interceptar la petición entrante. Este *header* le permite identificar una petición a la plataforma de integración, lo cual brinda la posibilidad de realizar estudios tanto de trazabilidad como análisis de *logs*.

Envoy propaga este *header* a medida que se invocan los microservicios para resolver una petición sobre la PI. Este identificador propagado entre microservicios es lo que luego permite determinar dada una petición, el conjunto de peticiones entre microservicios que la resolvieron.

Además del *header* “x-request-id”, Envoy genera automáticamente los *headers* “x-b3-traceId”, “x-b3-spanid” y “x-b3-parentspanid”. Con este conjunto de *headers* Envoy consigue llevar adelante distintas funcionalidades.

Primero, representar unidades lógicas de trabajo, las cuales tienen un nombre de operación, un tiempo de comienzo y una duración. Esta unidad lógica de trabajo se denomina *Span*.

A su vez estas *Spans* permiten representar la historia de una petición o flujo de ejecución, denominado Traza. Una Traza es entonces, un conjunto de *Spans* relacionadas entre sí a través de un identificador: “x-request-id”.

Finalmente, envía las trazas a un sistema distribuído de trazabilidad como Jaeger.

Uno de los requerimientos de Envoy, y por lo tanto de Istio, es que para obtener trazabilidad en la PI los microservicios deben propagar los *headers* mencionados anteriormente.

Debido a inconvenientes al momento de propagar los *headers* que Envoy necesita se decide remover Sleuth y propagarlos manualmente. Dicha tarea implica modificar los microservicios originales de la PI, lo cual se realiza siguiendo los principios de OpenTracing, resultando en un esfuerzo requerido bajo y permitiendo que la implementación pueda evolucionar adecuadamente en caso que se cambie el sistema de recolección de trazas.

La figura 6.3 ilustra el recorrido de una petición que llega a la solución y cómo se procesa para poder dotar a la PI de trazabilidad. En la tabla 6.4 se detalla el paso a paso con la referencia alfanumérica de la imágen.

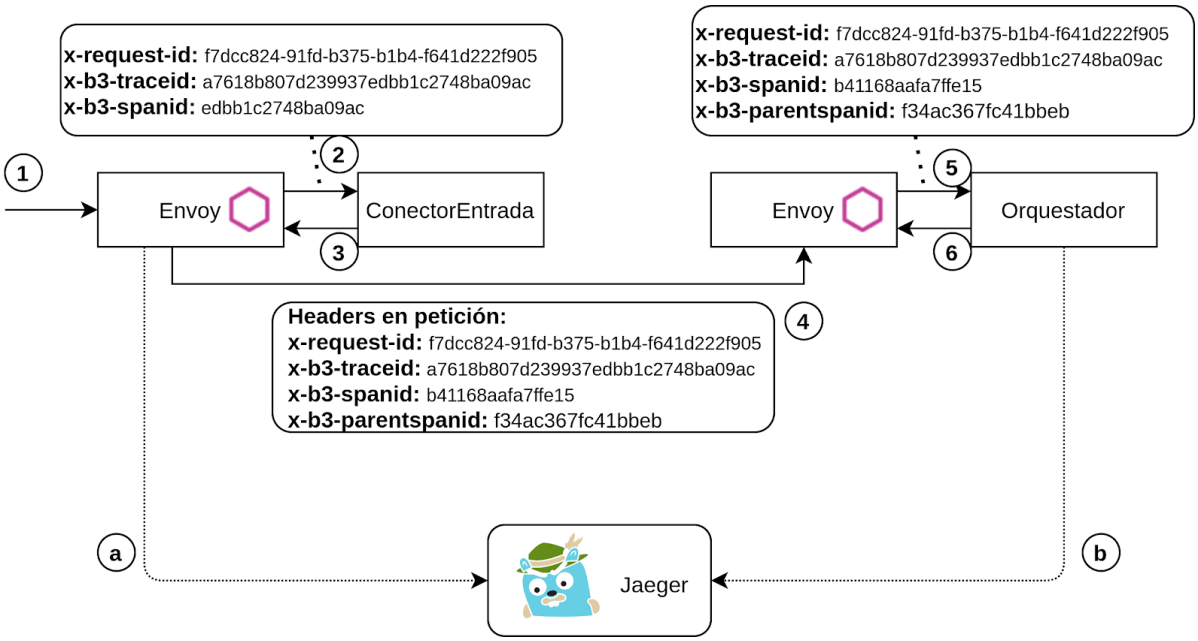

Figura 6.3: Ejemplo paso a paso de una petición que llega a la PI y su procesamiento para dotar a la solución de trazabilidad.

Paso	Descripción
1	Una petición ingresa a la plataforma de integración y es interceptada por un *proxy* Envoy. El *proxy* genera automáticamente el *header* “x-request-id” (en caso de que no exista) y genera los headers “x-b3-*”. Si el *proxy* es el primero en atender la petición, no existirá *header x-b3-parentspanid*.
2	Los *headers* son propagados por Envoy hacia el microservicio “ConectorEntrada” (CE).
3	Luego de realizar su tarea, el microservicio CE debe propagar los *headers* para que Envoy pueda mantener el contexto de la petición. En el ejemplo de la figura 6.3, el microservicio CE realiza una petición al microservicio Orquestador (O).
4	Envoy intercepta la petición de CE a O y agrega los *headers* correspondientes para implementar la trazabilidad. Se destaca el “header x-b3-parentspanid” que permite formar la relación padre/hijo entre CE y O.
5	Idem del paso 2.
6	Idem del paso 3.
a	A partir de los *headers,* Envoy envía información a Jaeger para recolección, almacenamiento y análisis.
b	Idem paso a.

Tabla 6.4: Paso a paso de la figura 6.3 detallado.

Resulta interesante destacar del proceso anterior que el *header* “x-request-id” y “x-b3-traceid” se propagan, inmutables, a todos los microservicios. Además, a medida que se realizan peticiones, Envoy actualiza de forma automática el *header* “x-b3-parentspanid” que permite obtener el orden de invocaciones padre/hijo entre los microservicios.

Por su parte, Jaeger se encarga de consolidar y unificar la información enviada por todos los *proxies* Envoy en la Service Mesh. Permite a su vez, visualizar las invocaciones que se realizan en la plataforma como un diagrama de Gantt. La figura 6.4 presenta dicho diagrama en el *dashboard* Jaeger UI para la versión extendida de la trazabilidad, en una solución de integración.

Si bien el *header* “x-request-id” permite trazar una petición dentro de la Service Mesh, no es suficiente para identificar qué petición que un cliente externo ha generado: es necesario contar con un identificador de peticiones externas. A tales efectos, se utiliza y propaga el *header* “x-client-trace-id” junto con “x-request-id”.

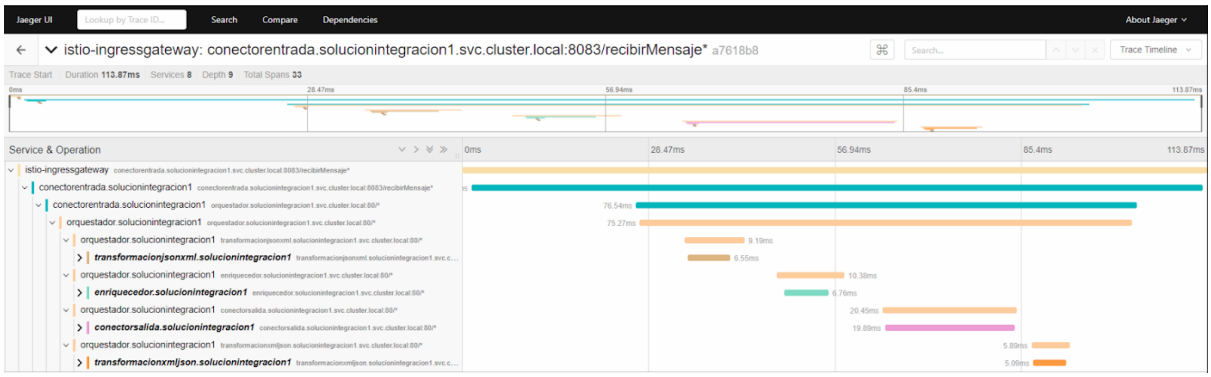

Figura 6.4: Ejemplo de diagrama de Gantt creado por Jaeger, en base a la información enviada por los distintos *proxies* Envoy.

El *header* “x-client-trace-id” es provisto por los clientes externos y es el mecanismo con el que cuentan para consultar por la traza de sus solicitudes. Este *header* se considera “poco confiable”; al ser proporcionado por un usuario externo a la PI no es posible asegurar que sea único. En cambio el par <x-client-trace-id, “x-request-id”> sí es único y permite trazabilidad desde que las solicitudes entran a la PI. La figura 6.5 ilustra el recorrido de los *headers*.

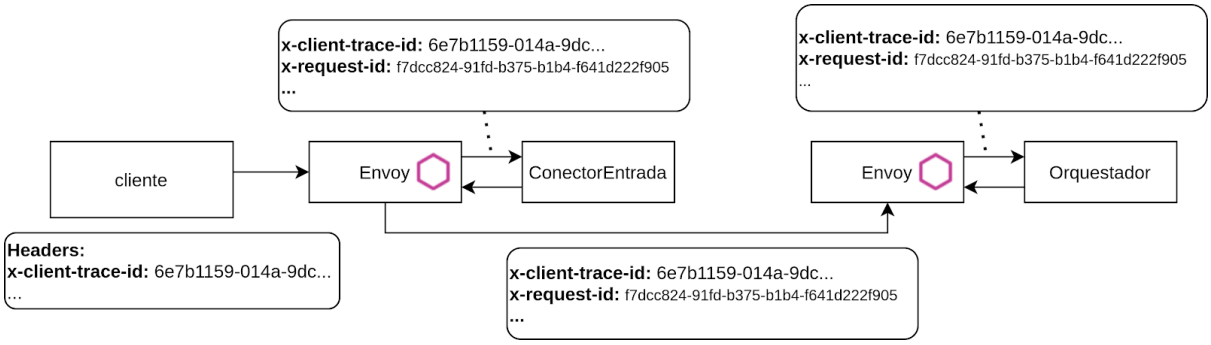

Figura 6.5: Ejemplo de solicitud externa y manejo del *header* “x-client-trace-id” que junto a “x-request-id” permiten trazabilidad.

El análisis de trazabilidad detallado hasta ahora deja de funcionar al ejecutar soluciones en modo Coreografía debido al escaso soporte que Envoy tiene para el protocolo AMQP.

Como forma de mitigar esta carencia se agrega instrumentación *opentracing* para RabbitMQ[84]. Si bien así se consigue observar trazas en el *dashboard* Jaeger UI, se considera que la información no es completa y no permite realizar un buen análisis.

Los *logs* sin embargo, cuentan con los *headers* “x-request-id” y “x-client-trace-id” por lo que es posible analizar las invocaciones entre los servicios desde el *dashboard* de Kibana, permitiendo

[84] https://github.com/opentracing-contrib/java-rabbitmq-client

cierto nivel de observabilidad y trazabilidad en la plataforma para el modo de ejecución por coreografía.

Dotar de trazabilidad a la ejecución coreográfica requiere de una solución invasiva respecto a la PI, quedando entonces por fuera de la solución desarrollada. La ejecución de soluciones asincrónicas resultó ser un punto débil de Istio.

Logging

A medida que el tráfico fluye en la Service Mesh, ésta puede generar un registro de *logs* completo para cada petición o mensaje intercambiado entre los microservicios. Esto es configurable y habilita a un operador a tener el control sobre cómo procesar los *logs*, qué información se debe mostrar, cuando generar estos *logs* y hacia dónde enviarlos.

Istio expone un conjunto de metadatos acerca del origen y destino de las peticiones entre los microservicios, información que resulta útil para definir estrategias de auditoria.

En la solución desarrollada se trabaja en dos tipos de *logs* en el sistema. En primer lugar, se debe poder acceder a los *logs* generados por los *proxies* Envoy para poder analizar el intercambio de mensajes en el sistema, a la vez que se necesita contar con *logs* a nivel de los microservicios.

Istio se concentra principalmente en brindar soporte para los *logs* generados por los *proxies* Envoy. No obstante este detalle, también proporciona las herramientas necesarias para poder recolectar y analizar los *logs* de los microservicios.

Istio cuenta con 3 tipos de CRD que permiten configurar los *logs* generados por los *proxies* Envoy. Por un lado existe el CRD de tipo (kind) “instance” el cual en base a una plantilla *logentry* permite definir los atributos a mostrar en cada *log* generado. La solución desarrollada cuenta con los siguientes atributos configurados: *source, destination, responseCode, responseSize* y *latency*.

Luego con la ayuda del CRD de tipo *handler* se configura a Istio para que sea capaz de procesar las instancias de *logs* definidas anteriormente. Este CRD funciona como un adaptador para configurar el componente Mixer del panel de control de Istio. En esta Solución se ha configurado un *handler* con un adaptador para enviar los *logs* a *stdio* (Standard Input Output).

Por último se deben definir las reglas por las cuales las instancias de *logs* serán enviadas al *handler*. Esto es posible a través del CRD de tipo *rule*. En su configuración se puede especificar una regla de tipo “*match*” la cual indica cuándo se debe enviar los *logs* al *handler*.

En la figura 6.6 se puede observar el flujo de trabajo para la recolección de *logs* de los *proxies* en la Service Mesh. Por su parte la tabla 6.5 enumera los pasos descritos en la imagen.

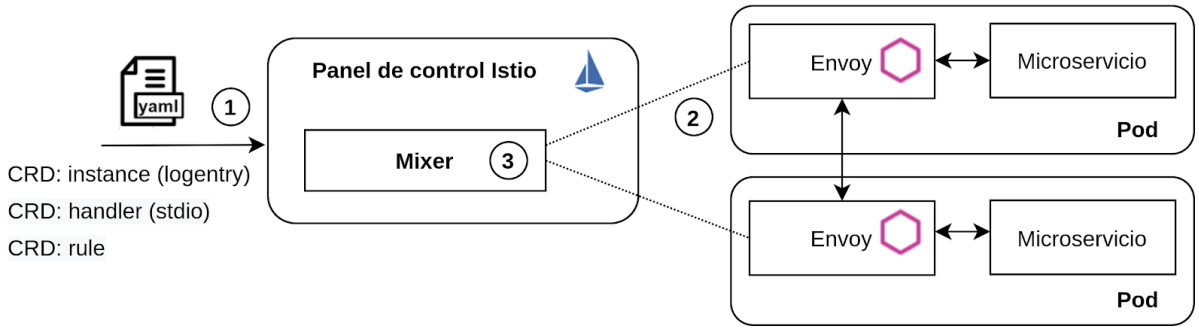

Figura 6.6: Recolección de *logs* a nivel de los *proxies,* por parte de Istio.

Paso	Descripción
1	Se configura a Istio con los CRD de tipo *instance*, *handler* y *rule*. Esta configuración impacta principalmente en el componente Mixer del panel de control.
2	A medida que los *proxies* Envoy intercambian mensajes, enviarán también telemetría en forma periódica al componente Mixer.
3	En base a las configuraciones realizadas en el paso uno, Mixer procesa la información enviada en el paso dos generando los *logs* con el formato deseado. Los *logs* son enviados luego a stdout.

Tabla 6.5: Paso a paso de la figura 6.6 detallado.

Más allá de la tarea que realizan por su cuenta tanto Envoy como Mixer, es necesario configurar el formato e información que se desea observar en la salida.

Una vez resuelta la configuración de los *logs* de los proxies Envoy, se procede con la implementación de los *logs* de los microservicios.

Tomar una decisión respecto al manejo de los *logs* a nivel de microservicios requiere de poder comprender la arquitectura de *logs* en un *cluster* Kubernetes, para poder decidir si se quiere mantener la solución original (integración con Graylog) o decantar por una solución que requiera menos integraciones y aproveche al máximo la infraestructura.

Al evaluar las alternativas posibles, se considera con especial atención que la solución se mantenga lenguaje agnóstica, ya que a medida que la plataforma evolucione, los nuevos lenguajes o *frameworks* que se utilicen también deberán acoplarse a la decisión tomada respecto al manejo de *logs*.

Imprimir en stdout los *logs* generados es uno de los primeros recursos a utilizar, pero resulta necesario tener un acceso persistente en el tiempo a estos *logs*. Se desea mantener el acceso

a estos *logs* si un contenedor falla, un *pod* es expulsado o un nodo del *cluster* deja de funcionar. Si bien Minikube es *single-node*, la referencia es válida para *clusters multi-node*. En un *cluster* los *logs* deben tener un lugar de almacenamiento y ciclo de vida independientes de contenedores, *pods* o nodos. Este concepto se denomina *cluster-level logging*[85].

En Kubernetes generalmente se pueden tomar tres enfoques distintos para lograr *logs* a nivel de *cluster*:

1. Usar un agente de *logs* por cada nodo.
2. Usar un contenedor *sidecar* para *logging* por *pod.*
3. Enviar *logs* directamente a un *backend* desde la aplicación.

En la primera opción, comúnmente el agente es un contenedor que tiene acceso a un directorio de archivos de *logs* de todos los contenedores en el nodo. La solución desarrollada al mismo tiempo que imprime los *logs* en stdout, persiste en archivos dentro de la carpeta "/var/log/containers" de Minikube

Istio permite la integración con recolectores de *logs* como Fluentd[86]. Fluentd se integra fácilmente con ElasticSearch[87], un motor de búsqueda para almacenar y buscar datos, los cuales pueden visualizarse con Kibana. A la colaboración entre estas herramientas se la conoce como *stack* EFK. También es posible utilizar a Fluentd como agente recolector del nodo para que obtenga *logs* de los archivos en la carpeta "/var/log/containers".

A pesar de que es posible configurar a Istio para que Mixer envíe los *logs* directamente a Fluentd, la solución desarrollada normaliza el manejo de *logs*, incluídos los que se generan a nivel de los microservicios, enviándolos a stdout. Al mismo tiempo, son persistidos en el *cluster* en archivos dentro de la carpeta "/var/log/containers", los cuales pueden ser recopilados por Fluentd, persistidos en ElasticSearch y visualizados con Kibana.

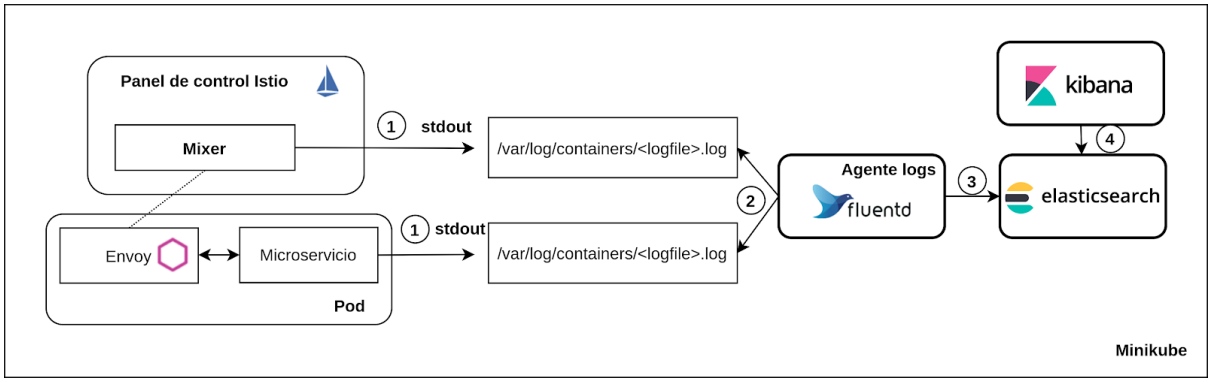

Figura 6.7: Recolección de *logs* a nivel de microservicios por parte de Istio.

[85] https://kubernetes.io/blog/2015/06/cluster-level-logging-with-kubernetes/
[86] https://www.fluentd.org/
[87] https://www.elastic.co/es/

Esta solución permite remover la dependencia con Graylog y simplifica el trabajo ya que resulta sencillo imprimir a stdout en cualquier lenguaje de programación.

La figura 6.7 presenta de manera visual el manejo de *logs* por parte de la solución, mientras que la tabla 6.6 detalla el paso a paso presente en la figura.

Paso	Descripción
1	Mixer y los microservicios envian *logs* a stdout. Estos *logs* son persistidos en el directorio “/var/log/containers/” del *cluster*.
2	Fluentd recolecta los *logs* leyendo de los archivos mencionados anteriormente.
3	Fluentd envía los *logs* a ElasticSearch para persistir y aprovechar el motor de búsqueda.
4	Se pueden visualizar y buscar los *logs* en el *dashboard* de Kibana.

Tabla 6.6: Paso a paso de la figura 6.7 detallado.

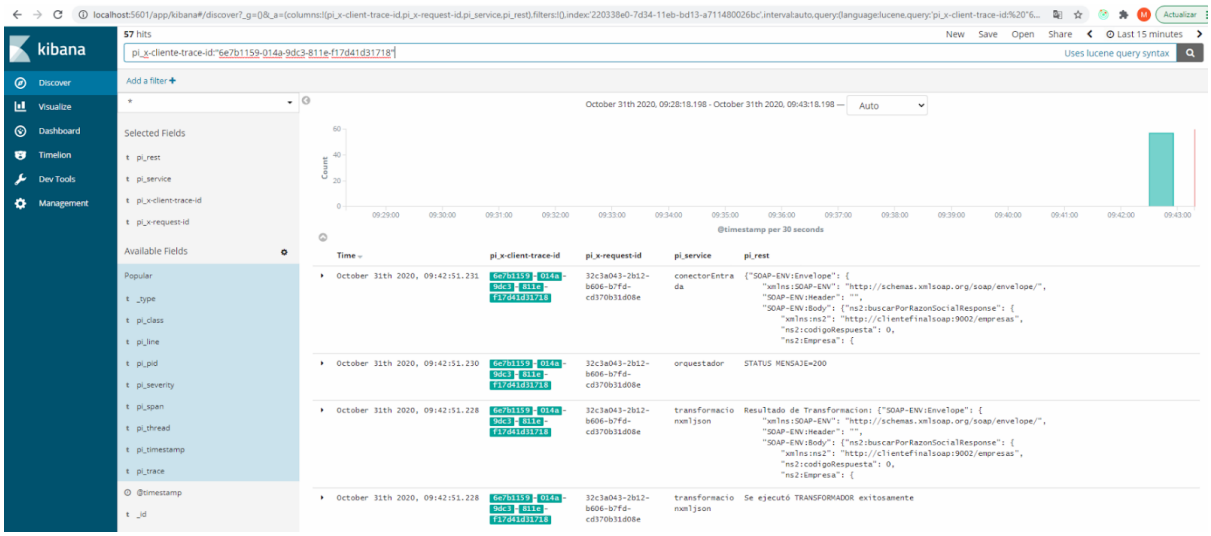

Figura 6.8: Visualización en Kibana de los *logs* generados a nivel de los microservicios.

Por último se destaca que los microservicios han sido configurados para imprimir el *header* “x-client-trace-id” y “x-trace-id” como se menciona en la sección de Trazabilidad. Esto permite realizar búsquedas por estos atributos en el *dashboard* de Kibana, logrando así visualizar todos los *logs* relacionados a una única ejecución de una SI en la plataforma. Las figuras 6.8 y 6.9 dan cuenta de cómo Kibana permite visualizar los *logs* de ambas fuentes (*proxies* y microservicios).

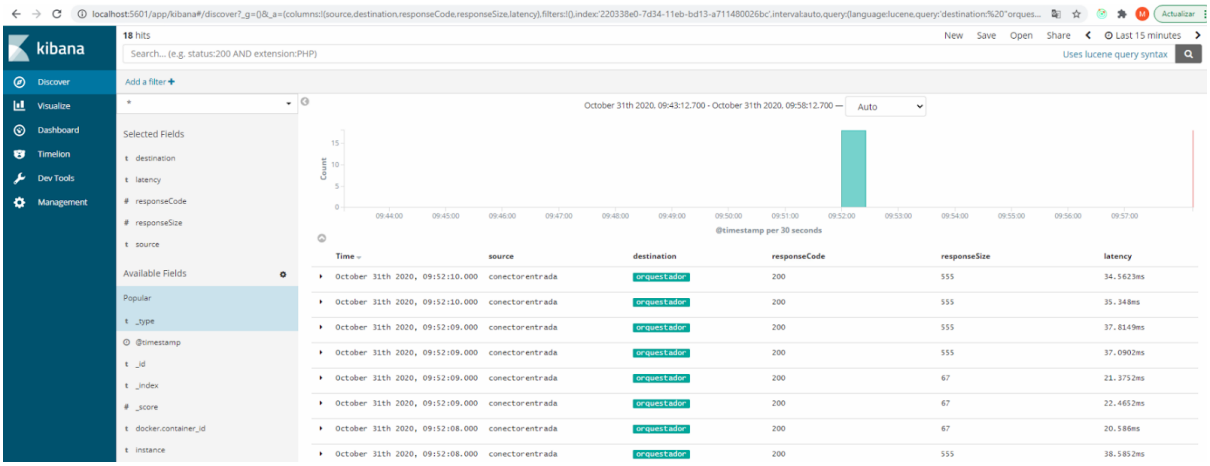

Figura 6.9: Visualización en Kibana de los *logs* generados a nivel de los *proxies* Envoy: peticiones entre servicios.

6.3 Nuevos Componentes de Integración

Se decide implementar los tres nuevos microservicios de la plataforma (router, modificador-contenido y conector-salida-rest) utilizando una tecnología distinta a la del resto de los microservicios, para demostrar que la plataforma es lenguaje agnóstica. A tales efectos se selecciona NodeJs[88] con el *framework* ExpressJs[89] para manejar la ejecución en modo orquestación, y la librería amqplib[90] para manejar la ejecución en modo coreografía.

La decisión de utilizar NodeJs se basa en los motivos listados a continuación:

- Capacidad de escalar eficientemente cuando no se requiere un uso elevado del CPU. Esto es debido a que funciona compartiendo un solo hilo y las tareas de entrada y salida se realizan de manera no bloqueante.
- Amplia disponibilidad de paquetes gratuitos en *Node Package Manager*[91] y una comunidad activa.
- Experiencia del equipo en la tecnología.

Los requerimientos para soportar orquestación son simples ya que solo se debe tener un *endpoint* HTTP que recibe una petición POST. Pese a esto se decide utilizar el *framework ExpressJs*, ya que a futuro podrían agregarse otros *endpoints*, por ejemplo para lidiar con las configuraciones de comportamiento.

[88] https://nodejs.org/es/
[89] https://expressjs.com/
[90] https://www.npmjs.com/package/amqplib
[91] https://www.npmjs.com/

Para manejar la ejecución en modo coreografía se crea una librería interna, llamada “manejador-mensajes”, utilizando el paquete *amqplib*. Se brinda una única función, llamada *handle*, que recibe dos parámetros:

1. El nombre de la cola a la cual el microservicio se debe suscribir.
2. Una función que recibe los siguientes tres parámetros para manejar los mensajes recibidos:
    - Contenido del mensaje.
    - Identificador de la configuración de comportamiento.
    - Una función para publicar el resultado de la operación.

Esta función se encarga de realizar la conexión a *RabbitMQ*[92], incluyendo los reintentos en caso de encontrar errores de conexión, incrementar el puntero al paso de ejecución para saber cual es el siguiente componente en el itinerario, así como consumir y publicar los mensajes utilizando los métodos *consume* y *sendToQueue* respectivamente, de la librería *amqplib*.

Debido a que los nuevos microservicios necesitan almacenar configuraciones de comportamiento, se agregó una base de datos para cada uno. La base de datos seleccionada es *MongoDB*. Para acceder a la base de datos se decide utilizar el *driver* nativo de *MongoDB*[93] debido a que las operaciones a realizar sobre las bases de datos son simples y este *driver* brinda un mejor rendimiento que otras librerías.

Localmente se trabaja utilizando Docker, con la imagen pública de MongoDB que se encuentra en Docker Hub[94]. El despliegue se hizo utilizando MongoDB Atlas[95], el cual es un servicio que permite alojar bases de datos en la nube.

La carga inicial de datos se hace a través de Mongo Seeding[96]. El primer paso para realizar la carga es crear una estructura de carpetas en el sistema de archivos como se muestra en la figura 6.10, donde la carpeta raíz contiene carpetas que representan colecciones de *MongoDB*. En cada carpeta de colección se agregan archivos con extensión *.js* donde se exportan los documentos que se deben importar a la colección.

Se crea una imagen de Docker partiendo de la imagen de Mongo Seeding. La imagen debe contener la carpeta con las colecciones a importar y las variables de entorno necesarias para conectarse a la base de datos. Por último se debe ejecutar la imagen creada al momento de desplegar cada microservicio.

[92] https://www.rabbitmq.com/
[93] https://www.npmjs.com/package/mongodb
[94] https://hub.docker.com/_/mongo
[95] https://www.mongodb.com/cloud/atlas
[96] https://github.com/pkosiec/mongo-seeding

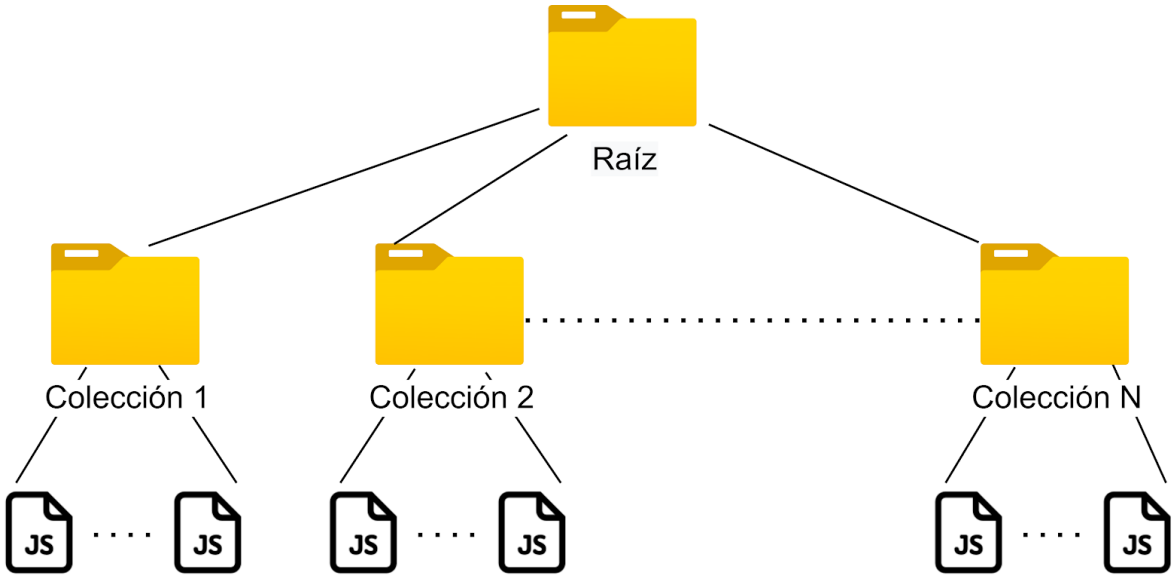

Figura 6.10: Estructura de carpetas para exportar documentos a *MongoDB* usando Mongo Seeding.

Para realizar la comunicación asíncrona entre servicios en el modo coreográfico se mantiene el mismo *message broker* RabbitMQ pero se cambia el tipo de *exchange* utilizado y las colas definidas. Cada componente de integración tiene una cola asociada y se usa el *exchange* por defecto que es de tipo directo.

Un *exchange* de RabbitMQ es el intermediario a donde se envían los mensajes, encargándose este de dirigirlos a las colas que correspondan. El *exchange* Direct envía el mensaje a una cola cuando coincide la clave de ruteo configurada.

A continuación se describe el flujo de mensajes con RabbitMQ del nuevo escenario de uso, para aquellos casos donde no existan errores, como se puede apreciar en la figura 6.11:

1. El servicio “conector-entrada” publica un mensaje al *exchange* por defecto con la clave de ruteo “modificador-contenido”.
2. El *exchange* rutea el mensaje a la cola “modificador-contenido”.
3. El servicio “modificador-contenido” recibe el mensaje, ya que está suscrito a la cola.
4. El servicio “modificador-contenido” publica un mensaje al *exchange* por defecto con la clave de ruteo “router”.
5. El *exchange* rutea el mensaje a la cola “router”.
6. El servicio “router” recibe el mensaje, ya que está suscrito a la cola.
7. El servicio “router” publica un mensaje al *exchange* por defecto con la clave de ruteo “conector-salida-rest”.
8. El *exchange* rutea el mensaje a la cola “conector-salida-rest”.
9. El servicio “conector-salida-rest” recibe el mensaje, ya que está suscrito a la cola.

10. El servicio “conector-salida-rest” publica un mensaje al *exchange* por defecto con la clave de ruteo “conector-entrada”.
11. El *exchange* rutea el mensaje a la cola “conector-entrada”.
12. El servicio “conector-entrada” recibe el mensaje, ya que está suscrito a la cola.

Cabe aclarar que en caso de ocurrir un error se publica un mensaje a una cola de errores. El “conector-entrada” está suscrito a esa cola para retornar el error al cliente.

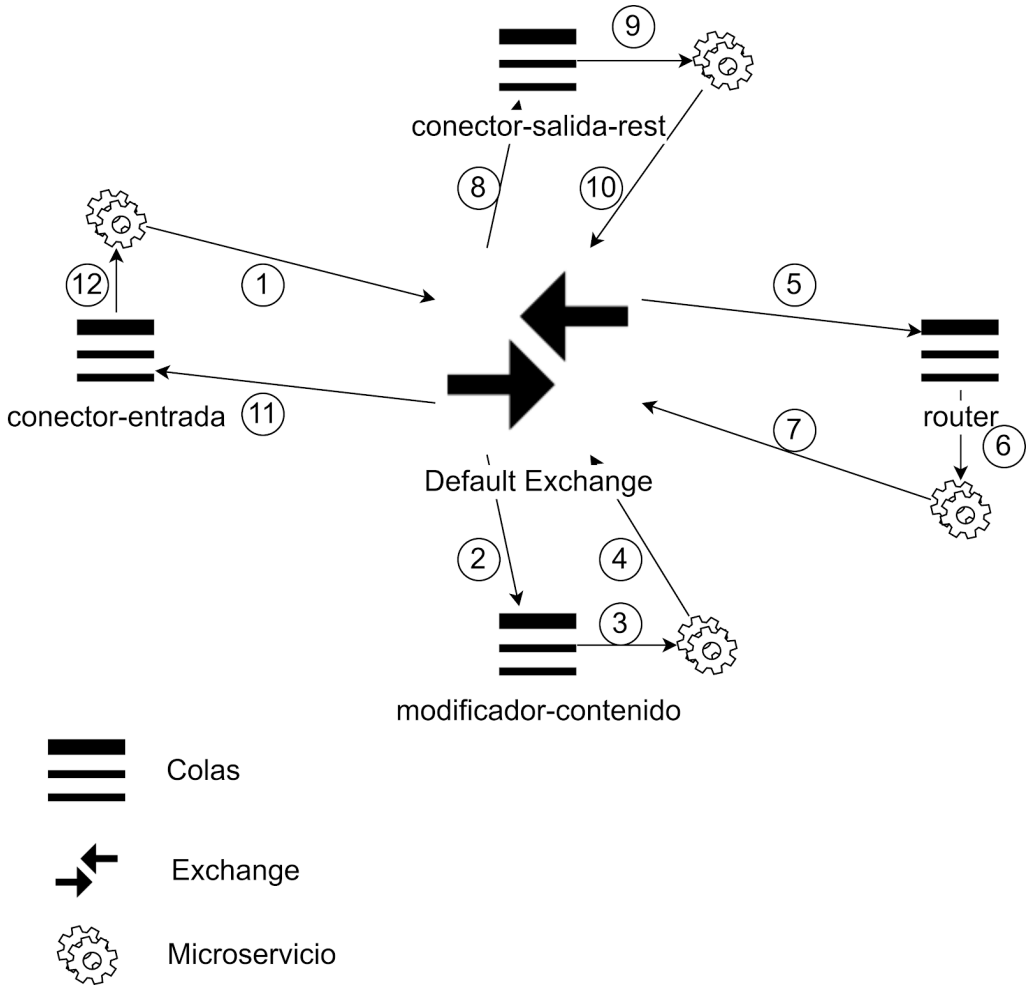

Figura 6.11: Flujo de mensajes entre RabbitMQ y microservicios.

6.3.1 Router

El componente de ruteo basa su ejecución en el contenido del mensaje, que utiliza para decidir hacia donde rutear su respuesta. Es responsabilidad de quien configura la SI que el mensaje que recibe el *router* se encuentre en formato JSON.

Para ejecutar las condiciones de ruteo se evalúan tres posibilidades para procesar el contenido del JSON y luego escoger la que mejor se adapte a las necesidades del proyecto: JSON path[97], JSON pointer[98] y JSON predicate[99].

JSON path es un lenguaje de consulta, basado en XPath[100], que se ejecuta sobre un JSON y retorna todos los valores que cumplen las condiciones especificadas.

JSON pointer permite obtener un valor específico en un JSON, de forma similar a JSON path pero sin la posibilidad de realizar filtrados sobre arreglos. Tanto JSON path como JSON pointer evalúan contra uno o más valores devueltos por la ejecución de la expresión, en caso de querer realizar evaluaciones distintas a las de igualdad, estas deben ser implementadas.

JSON predicate es un JSON que define condiciones booleanas que pueden ser compuestas utilizando los operadores lógicos *and*, *or* y *not,* junto a condiciones del tipo *contains*, *in*, *less*, entre otras.

Se decide utilizar JSON path considerando que de las opciones evaluadas es la más utilizada, si se tiene en cuenta la cantidad de descargas de las librerías. Pese a todas las funcionalidades brindadas por JSON predicate, también se considera que no tiene un *RFC* asociado, solo un borrador que ya expiró[101].

Las tres opciones cuentan con librerías[102] implementadas para NodeJS, la tecnología escogida para implementar el nuevo componente de integración, por lo que este factor no inclina la balanza en la decisión.

Otro aspecto a destacar del componente de ruteo es la estructura de configuración de comportamiento que se almacena en la base de datos. En la figura 6.12 se muestra un diagrama con la estructura almacenada.

En los atributos “json_path” y “json_path_value” se permite definir un conjunto de reglas a aplicar sobre el contenido del mensaje. El camino de ejecución que se resuelve ejecutar se encuentra en los atributos itinerario y configuraciones. Los atributos “_id” y “solucion_id” identifican la configuración de ruteo y la SI a la que pertenece respectivamente.

Los componentes lógicos encargados de manejar la orquestación y coreografía se diferencian del resto de los microservicios ya que deben agregar los cabezales con el camino a seguir. Cabezales HTTP en caso de orquestación y cabezales *AMQP* en el caso de coreografía.

---

[97] https://goessner.net/articles/JsonPath/
[98] https://tools.ietf.org/html/rfc6901
[99] https://tools.ietf.org/id/draft-snell-json-test-01.html
[100] https://developer.mozilla.org/en-US/docs/Web/XPath
[101] https://tools.ietf.org/id/draft-snell-json-test-01.html
[102] https://www.npmjs.com/package/jsonpath

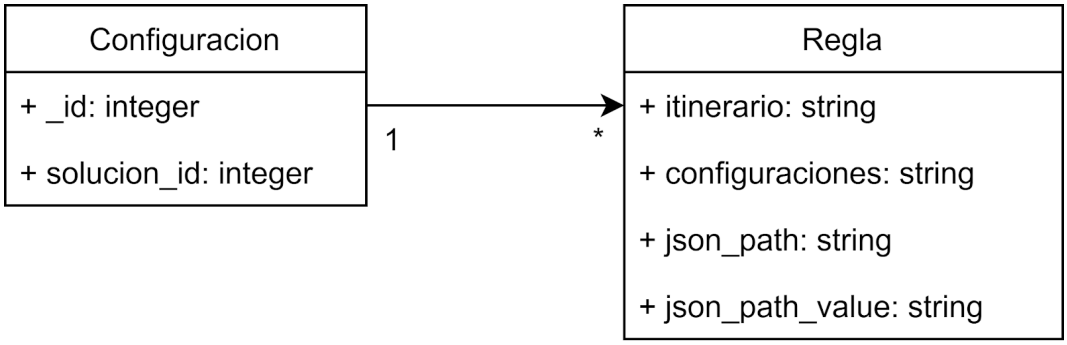

Figura 6.12: Estructura de configuración de comportamiento del Router.

6.3.2 Modificador de Contenido

Este componente se encarga de modificar el contenido del mensaje que recibe, incluyendo las secciones *headers*, *exchanges* y *payload*.

En la tabla 6.7 se muestran las operaciones y los atributos utilizados para cada una de las configuraciones de comportamiento soportadas, ilustradas en la figura 6.13. Una configuración contiene una secuencia de operaciones a aplicar sobre el contenido.

Operacion	Nombre	Valor	json_path	payload
AGREGAR_JPATH_EXCHANGE	Nombre del nuevo exchange.	-	Se aplica al payload para obtener el valor.	-
AGREGAR_JPATH_HEADER	Nombre del nuevo header.	-	Se aplica al payload para obtener el valor.	-
AGREGAR_VALOR_EXCHANGE	Nombre del nuevo exchange.	Valor del nuevo exchange.	-	-
AGREGAR_VALOR_HEADER	Nombre del nuevo header.	Valor del nuevo header.	-	-
MODIFICAR_PAYLOAD	-	-	-	Nuevo valor del payload utilizando tokens.

operacion	nombre	valor	json_path	payload
REMOVER_EXCHANGE	Nombre del exchange a eliminar.	-	-	-
REMOVER_HEADER	Nombre del header a eliminar.	-	-	-

Tabla 6.7: Atributos utilizados según operación a realizar.

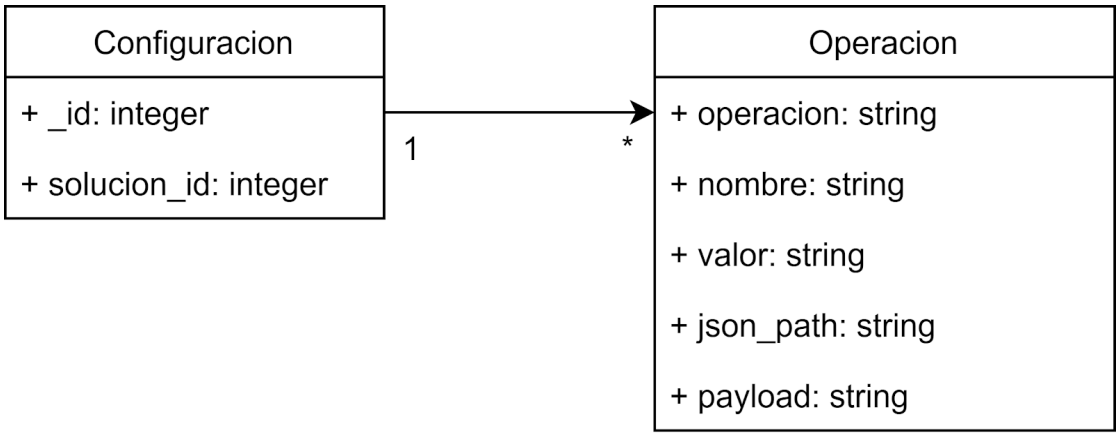

Figura 6.13: Estructura de configuración de comportamiento del modificador de contenido.

Para obtener valores del *payload*, al igual que en el componente *router*, se utiliza JSON path. La operación MODIFICAR_PAYLOAD se realiza utilizando *tokens* para permitir agregar valores de los *headers*, *exchanges* o el mismo *payload*. Para inyectar uno de estos valores se utiliza la siguiente sintaxis:

- ${header.NOMBRE}
- ${exchange.NOMBRE}
- ${payload}

6.3.3 Conector de salida REST

Este componente se encarga de invocar servicios REST externos a la plataforma. Para realizar la invocación se utiliza la librería node-rest-client[103]. En la figura 6.14 se muestra la estructura de las configuraciones de comportamiento soportadas.

[103] https://www.npmjs.com/package/node-rest-client

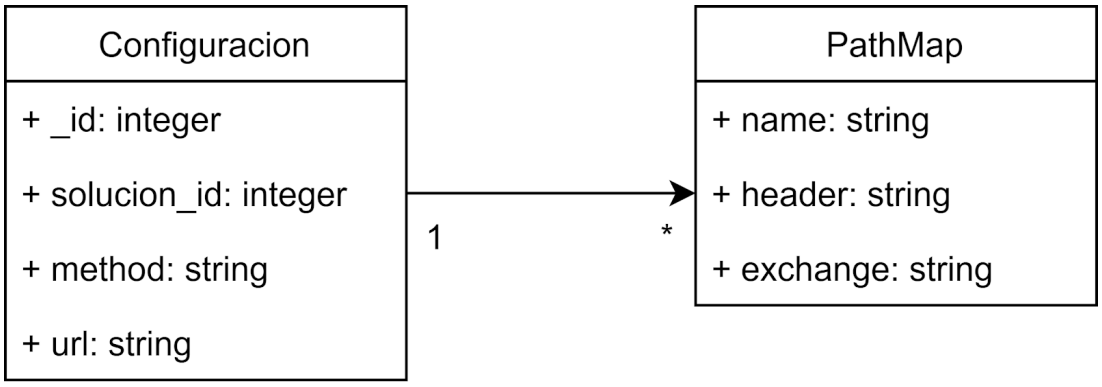

Figura 6.14: Estructura de configuración de comportamiento del conector de salida REST.

El method puede ser cualquiera de los métodos HTTP soportados por la librería node-rest-client:

- POST
- GET
- DELETE
- PATCH
- PUT

La *url* es un *string* parametrizable que contiene el recurso que se debe invocar. Los parámetros se agregan en la *url* entre llaves y los valores se encuentran en los *Path maps* asociados. Estos valores se pueden obtener de *headers* o *exchanges* del mensaje.

A continuación se muestra un ejemplo de configuración que realiza un POST y establece un parámetro de *url* con un valor de cabezal:

```
{
   "_id": 1,
   "method": "POST",
   "url": "http://antel.test:8093/productos/{id}",
   "path_map": [{"name": "id", "header": "producto_id"}]
 }
```

La invocación a este microservicio propaga los *headers* del mensaje recibido en la invocación externa, no así los *exchanges* que son de uso interno en la plataforma. En caso de implementar otro conector de salida en un protocolo que también incluye cabezales debe respetarse la misma estrategia.

7 Gestión del Proyecto

El objetivo de este capítulo es describir el proceso llevado a cabo durante el proyecto a nivel de gestión y planificación.

La sección 7.1 describe el transcurso del proyecto en etapas, a la vez que presenta un diagrama de Gantt basado en la planificación original del trabajo y luego uno adaptado a la duración final del mismo. Por su parte, la sección 7.2 da cuenta de las dificultades encontradas durante el transcurso del trabajo. Finalmente la sección 7.3 presenta las conclusiones a las que se arriba como consecuencia de la gestión del proyecto de grado.

7.1 Planificación del proyecto

Durante las primeras semanas de trabajo se elabora un documento que desglosa los objetivos del proyecto en tareas más pequeñas y sencillas de estimar. La Tabla 7.1 resume dicho documento.

Mes	Título	Tarea
Abril	Familiarización con las herramientas, patrones y tecnologías involucradas.	Repaso de conceptos relacionados como Plataforma de Integración, REST, etc.
		Microservicios: comparación con aplicaciones monolíticas, estudio de ventajas y desventajas de ambas.
		Service Mesh: estudio del concepto y de algunas de las opciones disponibles (*Istio*, *Linkerd, Consul*).
		Estudio de tecnologías utilizadas en el proyecto original (*Java Spring*[104], *Rabbit*).
		Orquestadores de contenedores: estudio del concepto y de algunas de las opciones disponibles (*Kubernetes*, *Docker Swarm*).
	Casos de uso de Service Mesh a nivel académico y empresarial.	¿Aplicaciones de Microservicios en producción que cuenten con una Service Mesh integrada? ¿Alguna que a la vez sea Plataforma de Integración?

[104] https://spring.io/

Mes	Título	Tarea
Abril	Planificación del pasaje de proyecto con el equipo anterior.	Paso a paso para poder desplegar la Plataforma de Integración. Pequeña demo de la misma.
		Listado de preguntas generales (¿Tareas que harían distinto si empezaran de nuevo? ¿Tecnologías y herramientas evaluadas y descartadas? ¿Nociones de lo que és una Service Mesh y cómo impactará su integración a la Plataforma de Integración?
		Recopilación de documentación, resúmenes, diagramas y el borrador de la documentación final.
		Evaluación post reunión y ajustes en las estimaciones realizadas.
Mayo	Deployment de la aplicación.	-
	Agregar un Orquestador de Contenedores.	Evaluar las opciones disponibles, optar por una de ellas, justificar en la documentación.
Junio - Julio	Agregar una Service Mesh al proyecto.	Evaluar las opciones disponibles, optar por la que mejor se adapte a las necesidades del proyecto.
		Integrar la Service Mesh, manteniendo a nivel de los Microservicios las funcionalidades que luego serán migradas a la SM.
		Plasmar en la documentación del proyecto los resultados obtenidos.
	Deployment del proyecto.	Evaluar plataformas donde desplegar el proyecto.
		Realizar el deployment en la plataforma elegida.
Agosto - Setiembre	Activación de la Service Mesh.	Evaluar qué servicios implementados a nivel de Microservicios quitar de la PI y activar en la SM.
		Encender uno a uno dichos servicios en la SM, a la vez que son quitados de la PI.
		Realizar un estudio comparativo de los pros/contras del cambio realizado.

Mes	Título	Tarea
Agosto - Setiembre	Activación de la Service Mesh.	Actualizar informe hasta el momento.
	Presentación preliminar del avance del proyecto.	Preparación de una presentación preliminar del avance del trabajo realizado.
Octubre	Extender con nuevos microservicios las funcionalidades de la PI.	Router: microservicio que rutea las solicitudes que recibe.
Noviembre	Lidiar con los pendientes del proyecto.	Relevamiento de requisitos que aún permanezcan pendientes, parcialmente implementados o no estén funcionando correctamente.
		Evaluar la posibilidad de cubrir los pendientes encontrados o mitigar su impacto.
		Implementar lo que se entienda pertinente y diseñar un *roadmap* para cubrir a futuro lo que quede pendiente.
Diciembre	Defensa y entrega del proyecto.	Redacción de lo que aún reste escribir de la documentación.
		Preparación de la presentación y defensa del proyecto.
		Revisión del trabajo a entregar.

Tabla 7.1: Tabla con objetivos iniciales desglosados en tareas.

Partiendo de la tabla anterior, se desarrolla el diagrama de Gantt presentado en la Figura 7.1. Por su parte, la figura 7.2 muestra el diagrama de Gantt que contempla la duración final del proyecto. La diferencia entre ambos es consecuencia de distintos factores de los que usualmente afectan proyectos de larga duración.

Etapas del proyecto

El proyecto se dividió en tres etapas. La primera, que abarcó el periodo de tiempo previo a tomar contacto con el código de la Plataforma de Integración y las semanas inmediatamente posteriores; la segunda etapa, que transcurre a partir de contar con la Plataforma de Integración y hasta luego de haber realizado la presentación preliminar de los avances del proyecto y finalmente la tercera etapa, que se extendió desde entonces y hasta la entrega de la versión final del proyecto y su documentación.

La figura 7.1 resume en un diagrama de Gantt, la planificación original del proyecto. Por su parte, la figura 7.2 muestra el diagrama de Gantt que contempla la duración final del proyecto. La diferencia entre ambos es consecuencia de distintos factores de los que usualmente afectan proyectos de larga duración. Por entenderlo como una estrategia adecuada para el equipo de trabajo, la documentación del proyecto transcurrió en paralelo con el resto de las tareas.

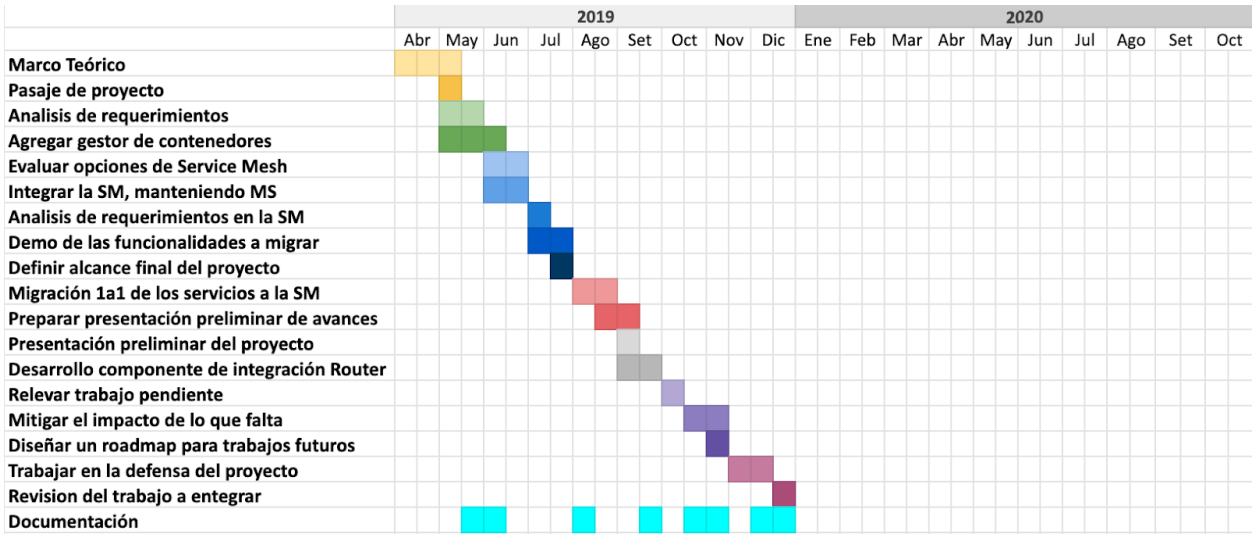

Figura 7.1: Diagrama de Gantt, resultado de la planificación inicial del proyecto.

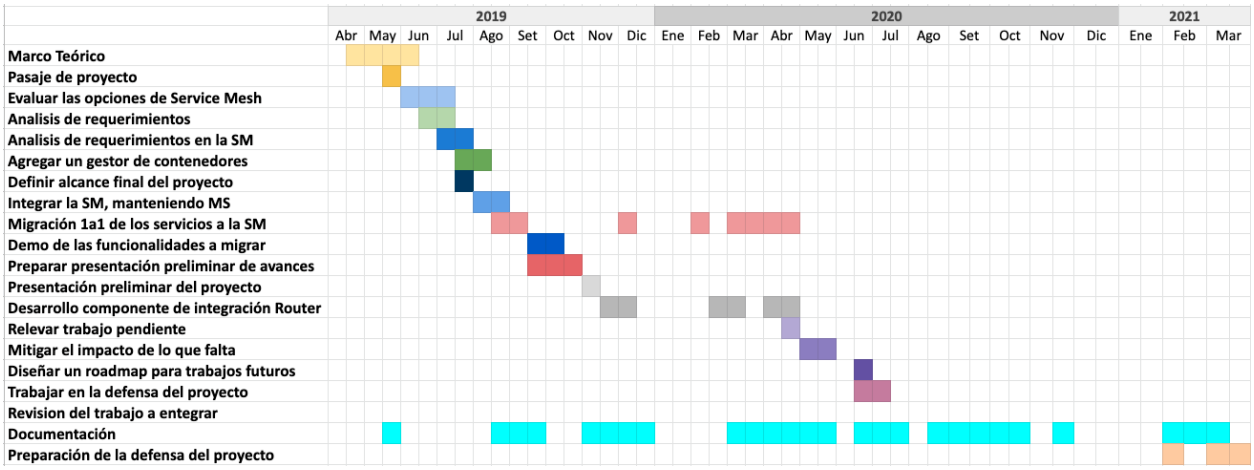

Figura 7.2: Diagrama de Gantt con los tiempos que efectivamente demandó el proyecto.

Marco Teórico y primera etapa de trabajo

La etapa de estudio e investigación inicial fue fundamental para poder entender la problemática, el trabajo realizado anteriormente y el objetivo de este proyecto. A tales efectos fue necesario que el equipo se familiarize con patrones de diseño como Microservicios o Plataforma de Integración, a la vez que se analizaba el trabajo realizado por Nebel primero, y

por Bonhomme y Camejo después. Además, fue necesario nivelar los conocimientos del equipo en tecnologías empleadas para el desarrollo de la Plataforma de Integración, como es la *suite* de librerías de *Java Spring*.

La anterior adquisición de conocimiento fue la base sobre la que se cimentó el estudio de las Service Mesh y los Orquestadores de Contenedores, concluyendo la primera etapa de trabajo luego de lograr desplegar la PI, pudiendo observar su funcionamiento al ejecutar una SI.

Análisis, Service Mesh y Presentación Preliminar.

El pasaje a esta etapa de análisis ocurre de la mano de haber estudiado las opciones de Service Mesh disponibles, en pos de identificar sus fortalezas y debilidades.

Con los requerimientos del proyecto y el material recabado sobre cada Service Mesh se procede a seleccionar la candidata a integrar, confirmando la viabilidad o no de poder brindar cada una de las funcionalidades requeridas.

Debido a la fuerte simbiosis de *Istio* con *Kubernetes*, su elección como SM a utilizar implica contar con un Orquestador de Contenedores sin que esto represente un esfuerzo adicional considerable. Así, el trabajo consistió en agregarle Istio a la PI primero, para progresivamente ir migrando una a una las funcionalidades correspondientes de la PI a la SM.

El hito que señaló el final de esta etapa de trabajo fue la preparación de una presentación del trabajo hecho hasta el momento.

Router, tareas pendientes y documentación.

Desde el punto de vista de la implementación, la etapa final del trabajo consistió en el desarrollo del nuevo componente de integración reutilizable (*Content-Based Router*) para poder implementar el nuevo caso de uso solicitado, terminando a su vez con la migración de funcionalidades de la etapa anterior.

La documentación del proyecto atraviesa las tres etapas en paralelo con el resto de las tareas, cobrando mayor trascendencia en la etapa final.

7.2 Dificultades encontradas

Como en todo trabajo colaborativo de investigación, el devenir del proyecto supuso enfrentarse a diferentes problemas y obstáculos no previstos que afectaron la planificación y ejecución del mismo en los plazos originalmente establecidos.

Estudio del problema y las etapas anteriores de trabajo

La temática del proyecto representa un reto en sí mismo, siendo que Service Mesh es una tecnología aún en tempranas etapas de desarrollo. A la dificultad intrínseca de trabajar con

herramientas nuevas se le sumó el hecho de estar realizando una tercer etapa de trabajo sobre la misma línea, lo cual implicó tener que interiorizarse con el diseño planteado por Nebel primero y luego con la PI de Bonhomme y Camejo.

Cada funcionalidad a migrar desde la PI a la SM implicó un análisis, un estudio de su funcionamiento en la PI y del componente de integración que la llevaba a cabo, quitar dicho componente y activar la funcionalidad a nivel de la SM, para finalmente corroborar el correcto funcionamiento de la integración PI+SM.

Haber comenzado a trabajar sin contar con la versión final de la PI ni con su documentación, dificultó las etapas tempranas de trabajo, a pesar de haber contado con un ida y vuelta entre el equipo anterior y el actual.

Tecnologías involucradas

Además de la relativa juventud de las Service Mesh o el desarrollo de aplicaciones basadas en Microservicios, el proyecto tuvo la dificultad agregada de abarcar un amplio abanico de tecnologías nuevas para el equipo de trabajo, así como de patrones de diseño y herramientas.

Entorno

Levantar una SI de la PI requiere de una computadora potente, aspecto del proyecto del que nos interiorizamos al momento de reunirnos con el equipo anterior. Integrar una SM aumentó incluso estos requerimientos.

Content-Based Router

La librería empleada para interactuar con *RabbitMQ* desde *NodeJS*, amqplib[105], no realiza reconecciones automáticamente en caso de perderse la conexión. Este problema queda en evidencia al ver una gran cantidad de reinicios en el servicio luego de ejecutar por un periodo de tiempo prolongado. Para solucionar este inconveniente se implementó la reconexión desde la librería “manejador-mensajes”, agendando un intento de reconexión en el evento *close* de la conexión.

7.3 Conclusiones

La planificación inicial del proyecto carecía de una comprensión cabal del alcance del mismo y sufrió de los distintos inconvenientes que surgen en proyectos de carácter educativo/investigación y largo alcance. Fueron varios los factores que influyeron en que fluctuase la capacidad de dedicación del equipo.

En cuanto a los imprevistos mencionados en la parte 7.3, fueron en su mayoría dificultades que debimos enfrentar o que logramos mitigar en etapas iniciales del trabajo.

[105] https://www.squaremobius.net/amqp.node/channel_api.html

8 Conclusiones y trabajos a futuro

La sección 8.1 resume los resultados y conclusiones a los que arriba el equipo de trabajo luego de concluido el proyecto de grado. Luego la sección 8.2 propone una serie de trabajos a futuro en base a ideas que surgieron durante el transcurso del trabajo, debilidades u oportunidades identificadas.

8.1 Conclusiones

Como conclusión general del proyecto, se ha logrado integrar una SM a una PI basada en microservicios. Para ello, fueron seleccionadas y analizadas las SM *Istio*, *Consul* Connect y *Linkerd*. Se opta por Istio pues cumple con todos los requerimientos funcionales del proyecto: *Circuit Breaker*, balanceo de carga, descubrimiento de servicios, trazabilidad, seguridad, centralización de la configuración, *logs* y *testing*. Fue posible poner en funcionamiento tanto las nuevas funcionalidades requeridas, como las que fueron migradas desde la PI hacia la SM. La elección de Istio permite a su vez, abarcar el requisito de integrar un Orquestador de Contenedores (Kubernetes).

Posteriormente se agrega el nuevo componentes de integración reutilizable solicitado *Content-Based Router*. Para corroborar su correcto funcionamiento, se agrega un nuevo escenario de ejecución a la plataforma que lo emplea, junto a otros dos nuevos componentes.

Resulta interesante observar cómo muchas de las inquietudes generadas al finalizar el desarrollo de la PI por parte de Bonhomme y Camejo son resueltas tras contar con la integración de una SM.

En cuanto a la solución planteada, se logra poner en funcionamiento *Load Balancing* a nivel de *Istio*, implementando más de un algoritmo de balanceo. Se implementa a su vez *Circuit Breaker*, evitando la comunicación con servicios incapaces de responder a peticiones. Se centralizan los *logs* utilizando el *stack Fluentd*[106], *Elasticsearch* y *Kibana*. Se normalizan los *logs* en los distintos lenguajes utilizados en la plataforma, facilitando el uso de consultas y la generación de reportes en *Kibana*. Se agrega trazabilidad al sistema y se emplea *Jaeger* para su visualización. Poder trazar las solicitudes permite cumplir uno de los objetivos del proyecto: dado un identificador de petición provisto por el cliente (*x-client-trace-id*) es posible brindar soporte en caso de fallas en el sistema. Se implementa *Mutual TLS* para las comunicaciones servicio a servicio *intra cluster* a través de *Citadel*. Dado que *Istio* provee de identidades a los servicios, se implementa autorización mediante *RBAC*[107]. Por último, se autentica a clientes del

[106] https://www.fluentd.org/
[107] https://istio.io/v1.1/docs/reference/config/authorization/istio.rbac.v1alpha1/

cluster a través de *JWT* y *JWKS*. Se debe mencionar que no fue posible implementar *TLS* fuera de la SM.

Se implementa una segunda solución de integración que reutiliza parcialmente servicios de la plataforma heredada. Además, se implementan tres nuevos servicios (*router*, modificador de contenido y conector salida *rest*) en un lenguaje diferente al utilizado anteriormente, lo cual evidencia la naturaleza lenguaje-agnóstica de la PI. Para esta implementación se presta atención a la reutilización de los servicios, dentro de la misma solución y entre soluciones.

Más allá de las conclusiones acerca de los objetivos cumplidos, hay conclusiones a nivel de aprendizaje acumulado cuyo valor resulta interesante destacar. Una SM puede ser de utilidad pero es necesario analizar detenidamente los requerimientos de la aplicación con la que se va a integrar. En el caso particular de las PI, debe tenerse en cuenta que las SM no soportan todos los protocolos de comunicación existentes, lo cual puede imposibilitar a un cliente de poder comunicarse con la PI. Este problema solamente afecta a los servicios encargados de recibir peticiones de clientes, ya que la comunicación interna se realiza siempre con los mismos protocolos.

Tal es el caso del balanceador de carga en la frontera del cluster como punto de ingreso. Por un lado permite tener múltiples instancias del conector de entrada, a la vez que es posible soportar múltiples soluciones de integración al mismo tiempo. Por otro lado, se implementa con un *proxy* Envoy, en particular es un Ingress Gateway. Este *proxy* introduce limitantes en la plataforma ya que el mismo cuenta con un soporte limitado de protocolos en capa 4 y capa 7 del modelo OSI. Por otro lado la PI original implementa el conector de entrada con Spring Integration lo que permite cambiar fácilmente el protocolo de comunicación con el exterior. Se concluye que introducir un Ingress Gateway implica contar con nuevas funcionalidades como soporte de múltiples soluciones de integración al mismo tiempo, balanceo de carga, seguridad y trazabilidad entre otros, pero también implica que se soporten menos protocolos de comunicación con el exterior. Una posible solución sería dejar por fuera de la SM los servicios encargados de recibir peticiones de clientes externos. Como contrapartida se pierden los beneficios de contar con una SM para dichos servicios.

Lo anterior refleja una conclusión importante del trabajo. Generalmente una PI intenta soportar la mayor cantidad de protocolos de comunicación con el exterior posibles: integrar una SM requiere realizar un análisis del impacto a nivel de los protocolos de comunicación soportados. En este trabajo se reconoce la limitante y se decide introducir un Ingress Gateway de todas formas ya que en este caso en particular se ejemplifica una integración donde los clientes externos utilizan los protocolos TCP/HTTP, protocolos que Istio y Envoy soportan muy bien.

En más de una ocasión sucedió que, al trabajar en la integración de la SM y la PI, soportar las soluciones de integración mediante orquestación (comunicación sincrónica) resultaba natural, mientras que al trabajar con coreografía (comunicación asincrónica) las fricciones eran mayores. Con el transcurso del proyecto de grado y un mayor conocimiento de las tecnologías

involucradas, la inquietud de que Istio no soporta correctamente la comunicación asincrónica fue en aumento. Se decidió consultar a referentes del ecosistema e investigar prácticas comunes al integrar una SM a una aplicación con comunicación asíncrona. Tanto la consulta realizada como nuestra investigación en paralelo arrojan conclusiones similares: las implementaciones actuales de SM se centran en comunicaciones sincrónicas, relegando mensajería a nivel de los microservicios, lo cual viene a contradecir parcialmente la división de responsabilidades que plantea el patrón de diseño SM. La problemática está planteada y existen diferentes opciones de mantener la abstracción planteada por SM adaptada a comunicación mediante mensajería [36]. En la sección 8.2 se profundiza al respecto. En particular, en el caso del broker de mensajería RabbitMQ que utiliza el protocolo AMQP y no es soportado por Istio, se opta por no introducir un proxy delante del mismo, afectando funcionalidades como la trazabilidad en coreografía.

La curva de aprendizaje de *Istio* no es de menospreciar, si bien conocer y manejar tecnologías como *Docker* y *Kubernetes* allanan el camino. Por la misma razón, es imprescindible analizar la realidad sobre la que se está trabajando, evaluando si integrar una SM es lo que el proyecto necesita. En caso afirmativo y dependiendo del rol que la SM va a ocupar, es posible limitar la integración sólamente a las funcionalidades necesarias.

Dentro de las opciones de SM disponibles las hay que ofrecen una gran variedad de funcionalidades y un alto grado de personalización a costa de una mayor complejidad en su manejo (*Istio*), y las hay con una curva de aprendizaje más reducida, concentrándose su área de servicio en algunas funcionalidades centrales (*Linkerd*). Anticipar las necesidades del proyecto permite optar por una herramienta u otra.

8.2 Trabajo a futuro

A continuación se examinan posibles mejoras, trabajos pendientes y desarrollos futuros de interés para el proyecto.

Luego de terminado su trabajo, Bonhomme y Camejo plantean como posibles trabajos a futuro la inclusión de un Gateway de entrada que permita que el componente “Conector de Entrada” pueda escalar, hacer que la solución sea tolerante a fallos, agregar seguridad y autenticación, unificar los archivos de configuración en uno sólo, extender la *suite* de componentes de integración con nuevos microservicios reutilizables, así como agregarle monitoreo a la plataforma.

Uno de los puntos débiles de la Plataforma de Integración original es la imposibilidad de escalar el conector de entrada. En este trabajo se resuelve esta limitante cuando la solución se ejecuta en modo orquestación, quedando pendiente para el modo coreografía. Una posible solución a este problema es contar con una base de datos (ej: Redis) que permita obtener la instancia del conector de entrada que debe procesar la respuesta.

Para lograr proveer a la solución de un balanceo de cargas adecuado, además de dotarla de autenticación para los clientes que la utilicen es necesario agregar un *Gateway* de entrada. A través de él sería posible levantar la limitante de escalabilidad del componente de entrada.

En lo que refiere a la tolerancia a fallos, la Service Mesh aumenta la resiliencia de plataforma al aplicar *Circuit Breaker*.

Seguridad y autenticación eran también parte de los requerimientos iniciales del corriente trabajo y fueron activadas a nivel de la Service Mesh con éxito.

La inclusión del *Router* como componente de integración reutilizable extiende los servicios disponibles para crear nuevas soluciones de integración; en particular la implementación de un *Router* fue parte de los requerimientos iniciales del proyecto de grado de Bonhomme y Camejo, siendo excluído de la misma como parte un recorte del alcance.

Finalmente el monitoreo de la plataforma es posible gracias a la Service Mesh y fue incluído en el corriente trabajo como parte de los requerimientos iniciales del mismo.

Interfaz gráfica

Los usuarios de la Plataforma de Integración se verían beneficiados al contar con una interfaz gráfica que les permitiese crear Soluciones de Integración mediante mecanismos de *drag-and-drop*.

En la sección 2.7 se analizó brevemente el ejemplo de la plataforma de integración RoboMQ que emplea una interfaz gráfica denominada *Integration Flow Designer,* para permitirle a sus usuarios crear soluciones de integración.

Durante el transcurso del proyecto, se recurrió a diferentes diagramas para poder visualizar, entender y discutir la interacción entre componentes de integración, componentes de la propia plataforma y luego también de la Service Mesh, al momento de llevar adelante las distintas soluciones de integración empleando coreografía u orquestación. Ésto evidencia el potencial de que fuese la propia plataforma la que compusiese dichos diagramas a partir de una SI existente, o que incluso como sucede con RoboMQ, permitiese confeccionar nuevas SI arrastrando y soltando sus componentes en un entorno visual amigable con el usuario.

En lo que refiere a la Service Mesh, resultaría interesante poder contar con una interfaz que permitiese cambiar aspectos de la configuración (como el *timeout* del *Circuit Breaker* o el algoritmo a aplicar por *Load Balancing*), generando los archivos YAML correspondientes.

Como un nivel más de trabajo sobre esta idea, podría desarrollarse una herramienta de simulación que le brindase la posibilidad al usuario de generar variantes sobre una SI, siendo luego capaz de comparar métricas de performance y estadísticas sobre la SI original y su transformada.

Autoscaling

Luego de contar con Istio y Kubernetes integrados a la PI, la posibilidad de automatizar el escalado de los servicios constituye una posibilidad real de mejora para la plataforma, demostrando el potencial de la misma y de las tecnologías involucradas.

Kubernetes maneja dos niveles de escalabilidad conocidos como *Horizontal Pod Autoscaler*[108] (HPA) y *Vertical Pod Autoscaler*[109] (VPA). Ambos pueden coexistir ya que responden a necesidades diferentes.

HPA le permite a *Kubernetes* aumentar o disminuir según sea necesario, el número de réplicas de los *pods*, normalmente utilizando como disparadores los niveles de uso de memoria y CPU, aunque las métricas empleadas en la toma de decisiones pueden ser otras. A tales efectos contar con Istio monitoreando la actividad de la aplicación resulta súmamente útil.

VPA aumenta o disminuye los niveles de CPU disponibles para los *pods*, contando incluso con una funcionalidad denominada *VPA Recommender* que monitorea el uso de CPU y memoria (incluyendo posibles eventos *Out of Memory*) y sugiere qué niveles de dichos recursos brindarle al *pod*.

Despliegue en la nube

En cuanto al despliegue del proyecto, el mismo se realizó sobre Minikube[110] por ser este un ambiente adecuado para el desarrollo. Poder desplegar la solución en la nube solucionaría el problema de necesitar una máquina potente para utilizar la plataforma, siendo además un uso de la herramienta más parecido a los que se puede encontrar en ambientes de producción.

Pruebas de carga

Contar con una Service Mesh integrada a la plataforma implica desde el punto de vista de la arquitectura de la solución, una mejora sustancial en la división de roles, que permite trabajar en equipos pequeños y especializados en la lógica de negocio del servicio que estén desarrollando. Más allá de esta ventaja que impacta en los tiempos de desarrollo y la organización de los equipos, hubiese sido interesante contar con datos recabados de la ejecución *pre* y *post* Service Mesh de una solución de integración.

Comunicación asincrónica

En la sección 8.1 se observa que las Service Mesh actuales no soportan correctamente las comunicaciones mediante intercambio de mensajes asincrónicamente. Como una posible solución a la mensajería en aplicaciones que utilizan una SM, surgen las Event Mesh

[108] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[109] https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler
[110] https://minikube.sigs.k8s.io/docs/
[111] https://github.com/aeraki-framework/aeraki

impulsadas por empresas como *Solace* [37]. Las Event Mesh son una capa de infraestructura que convive y colabora con la SM para dar soporte a comunicaciones asincrónicas en aplicaciones distribuídas.

Cabe destacar que Envoy evoluciona continuamente y cuenta con soporte reciente para *Kafka* por lo que podría esperarse soporte para RabbitMQ en un futuro.

Extender Istio/Envoy

El manejo de comunicaciones que no sean TCP/HTTP resulta desafiante en Istio limitando los protocolos soportados por la plataforma de integración. Una línea de trabajo interesante sería extender tanto Istio como Envoy para soportar más protocolos de comunicación y generar un *framework* que permita facilitar dicha extensión. Ya existe un trabajo con este objetivo: Aeraki[111].

9 Referencias

[1] Francesco, Paolo & Malavolta, Ivano & Lago, Patricia. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. 21-30. 10.1109/ICSA.2017.24.

[2] C. Richardson, What are microservices» [En línea]. Available: https://microservices.io/. [Último acceso: 10 3 2020].

[3] J. Lewis y M. Fowler, «Microservices,» 2014. [En línea]. Available: https://martinfowler.com/articles/microservices.html. [Último acceso: 10 3 2020].

[4] Jamshidi, Pooyan & Pahl, Claus & Mendonça, Nabor & Lewis, James & Tilkov, Stefan. (2018). Microservices: The Journey So Far and Challenges Ahead. IEEE Software. 35. 24-35. 10.1109/MS.2018.2141039.
Available: https://www.researchgate.net/publication/324959590_Microservices_The_Journey_So_Far_and_Challenges_Ahead. [Último acceso: 14 3 2020].

[5] InfoQ, Service Mesh Ultimate Guide: Managing Service-to-Service Communications in the Era of Microservices [En línea]. Available: https://www.infoq.com/articles/service-mesh-ultimate-guide/. [Último acceso: 14 3 2020].

[6] J. Bonhomme, E. Camejo, Plataformas de Integración basada en Microservicios, 2019.

[7] A. Nebel, Arquitectura de Microservicios para Plataformas de Integración, 2018.

[8] Istio, What is Istio? [En línea]. Available: https://istio.io/docs/concepts/what-is-istio/. [Último acceso: 19 5 2020].

[9] Enterprise Integration Patterns, Content-Based Router [En línea]. Available: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html. [Último acceso: 15 3 2020].

[10] Enterprise Integration Patterns, Why Enterprise Integration Patterns? [En línea]. Available: https://www.enterpriseintegrationpatterns.com/patterns/messaging/. [Último acceso: 15 3 2020].

[11] Kubernetes, Qué es Kubernetes? [En línea]. Available: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/. [Último acceso: 15

3 2020].

[12] William Morgan, What's a Service Mesh? And why do I need one? [En línea]. Available: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/. [Último acceso: 16 3 2020].

[13] C. Richardson, Pattern: Sidecar [En línea]. Available: https://microservices.io/patterns/deployment/sidecar.html. [Último acceso: 21 3 2020].

[14] M. Klein, Service mesh data plane vs. control plane. [En línea]. Available: https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc. [Último acceso: 21 3 2020].

[15] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," 2015 IEEE 14th International Symposium on Network Computing and Applications, Cambridge, MA, 2015, pp. 27-34. [En línea]. Available: https://arxiv.org/pdf/1511.02043.pdf. [Último acceso: 22 3 2020].

[16] C. Richardson, Pattern: Service instance per container [En línea]. Available: https://microservices.io/patterns/deployment/service-per-container.html. [Último acceso: 22 3 2020].

[17] C. Richardson, Pattern: Service deployment platform [En línea]. Available: https://microservices.io/patterns/deployment/service-deployment-platform.html. [Último acceso: 22 3 2020].

[18] Vayghan, Leila & Saied, Mohamed & Toeroe, Maria & Khendek, Ferhat. (2019). Kubernetes as an Availability Manager for Microservice Applications. [En línea]. Available: https://arxiv.org/pdf/1901.04946.pdf. [Último acceso: 22 3 2020].

[19] NGINX, What is a Service Mesh [En línea]. Available: https://www.nginx.com/blog/what-is-a-service-mesh/. [Último acceso: 23 3 2020].

[20] William Morgan, The history of the Service Mesh [En línea]. Available: https://thenewstack.io/history-service-mesh/. [Último acceso: 23 3 2020].

[21] Consul, Consul Connect Envoy [En línea]. Available: https://www.consul.io/docs/commands/connect/envoy.html. [Último acceso: 23 3 2020].

[22] Fritzsch, Jonas & Bogner, Justus & Zimmermann, Alfred & Wagner, Stefan. (2018). From Monolith to Microservices: A Classification of Refactoring Approaches. Available: https://arxiv.org/pdf/1807.10059.pdf. [Último acceso: 28 2020].

[23] T. Cerny, M. J. Donahoo y M. Trnka, «Contextual understanding of microservice architecture: current and future directions,» ACM SIGAPP Applied Computing Review, vol. 17, nº 4, pp. 29-45, 2018.

[24] G. Pardo-Castellote y A. Corsaro, «Analysis of the Advanced Message Queuing Protocol (AMQP) and comparison with the Real-Time Publish Subscribe Protocol (DDS-RTPS Interoperability Protocol),» Julio 2007. [En línea]. Available: https://www.omg.org/news/meetings/workshops/RT-2007/04-3_Pardo-Castellote-revised.pdf. [Último acceso: 30 3 2020].

[25] Linkerd, A guide to distributed tracing with Linkerd [En línea]. Available: https://linkerd.io/2019/10/07/a-guide-to-distributed-tracing-with-linkerd/ [Último acceso: 12 4 2020].

[26] RoboMQ, Who we are [En línea]. Available: https://www.robomq.io/about-us/. [Último acceso: 12 4 2020].

[27] RoboMQ, Integration Flow Designer [En línea]. Available: https://www.robomq.io/integration-flow-designer/. [Último acceso: 12 4 2020].

[28] RoboMQ, Hybrid Integration Platform [En línea]. Available: https://www.robomq.io/hybrid-integration-platform-hip/. [Último acceso: 12 4 2020].

[29] MuleSoft, API-led connectivity (whitepaper) [En línea]. Available: https://www.mulesoft.com/lp/whitepaper/api/api-led-connectivity. [Último acceso: 23 4 2020].

[30] MuleSoft, MuleSoft announces Anypoint Service Mesh [En línea]. Available: https://www.mulesoft.com/press-center/october-2019-release-anypoint-service-mesh. [Último acceso: 26 4 2020].

[31] Snap, From Monolith to Multicloud Micro-Services: Inside Snap's Service Mesh [En línea] Available: https://eng.snap.com/monolith-to-multicloud-microservices-snap-service-mesh/. [Último acceso: 12 5 2020].

[32] Snap, Monolith to Microservices: Migrating Snap's Architecture using a Service Mesh [En línea] Available: https://www.infoq.com/news/2020/04/snap-architecture-service-mesh/. [Último acceso: 12 5 2020].

[33] Istio, General FAQ [En línea] Available: https://istio.io/faq/general/#what-is-istio. [Último acceso: 19 5 2020].

[34] Istio, Feature Status [En línea] Available: https://istio.io/about/feature-stages/. [Último acceso: 21 5 2020].

[35] InfoQ, The Potential for Using a Service Mesh for Event-Driven Messaging [En línea] Available: https://www.infoq.com/articles/service-mesh-event-driven-messaging/. [Último acceso: 28 6 2020].

[36] Solace, What is an Event Mesh? [En línea] Available: https://solace.com/what-is-an-event-mesh/. [Último acceso: 28 6 2020].

[37] MEGARGEL, Alan; SHANKARARAMAN, Venky; and WALKER, David K.. Migrating from monoliths to cloud-based microservices: A banking industry example. (2020). *Software Engineering in the Era of Cloud Computing*. 85-108. Research Collection School Of Information Systems. [En línea] Available: https://ink.library.smu.edu.sg/sis_research/4725 [Último acceso: 28 6 2020].

[38] Microsoft, Tip of the Day: Demystifying Software Defined Networking Terms - The Cloud Compass: SDN Data Flows [En línea] Available: https://docs.microsoft.com/es-es/archive/blogs/tip_of_the_day/tip-of-the-day-demystifying-software-defined-networking-terms-the-cloud-compass-sdn-data-flows [Último acceso: 17 7 2020].

[39] C. Richardson, Pattern: Log aggregation [En línea]. Available: https://microservices.io/patterns/observability/application-logging.html. [Último acceso: 19 7 2020].

[40] C. Richardson, Pattern: Distributed tracing [En línea]. Available: https://microservices.io/patterns/observability/distributed-tracing.html. [Último acceso: 18 7 2020].

[41] Istio, Observability [En línea]. Available: https://istio.io/latest/docs/concepts/observability/. [Último acceso: 20 7 2020].

[42] Microsoft, Enable a remote workforce by embracing Zero Trust security [En Línea]. Available: https://www.microsoft.com/en-us/security/business/zero-trust [Último acceso: 20 7 2020].

[43] Envoyproxy, hot restart, Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/operations/hot_restart

[Último acceso: 26 7 2020]

[44] Service Mesh - the Microservices in Post Kubernetes Era. Jimmy Song [En línea]. Available: https://jimmysong.io/en/blog/service-mesh-the-microservices-in-post-kubernetes-era/ [Último acceso: 26 7 2020]

[45] Envoyproxy, life of a request. [En línea]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/life_of_a_request [Último acceso: 26 7 2020]

[46] NnginX, What is Load Balancing?. [En línea]. Available: https://www.nginx.com/resources/glossary/load-balancing/ [Último acceso: 8 8 2020]

[47] Istio, Security. [En línea]. Available: https://istio.io/v1.4/docs/concepts/security/#mutual-tls-authentication [Último acceso: 12 8 2020]

[48] Red Hat, What is container orchestration?. [En línea]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration [Último acceso: 13 8 2020]

[49] C. Richardson, Pattern: Health Check API [En línea]. Available: https://microservices.io/patterns/observability/health-check-api.html [Último acceso: 25 8 2020]

[50] T. Clemson, Testing Strategies in a Microservices Architecture [En línea]. Available: https://martinfowler.com/articles/microservice-testing [Último acceso: 26 8 2020]

[51] C. Richardson, Pattern: Externalized Configuration [En línea]. Available: https://microservices.io/patterns/externalized-configuration.html. [Último acceso: 26 3 2020].

[52] HashiCorp, Introduction to Consul [En línea]. Available: https://www.consul.io/docs/intro. [Último acceso: 20 9 2020].

[53] William Morgan, Linkerd: Twitter-style Operability for Microservices [En línea]. Available: https://linkerd.io/2016/02/18/linkerd-twitter-style-operability-for-microservices/. [Último acceso: 20 9 2020].

[54] Istio, Introducing Istio [En línea]. Available: https://istio.io/latest/news/releases/0.x/announcing-0.1/. [Último acceso: 20 9 2020].

[55] Li, Zheng (Eddie) & Kihl, Maria & Lu, Qinghua & Andersson, Jens. (2017). Performance Overhead Comparison between Hypervisor and Container Based Virtualization. 955-962. 10.1109/AINA.2017.79.

[56] Docker Inc., What is a Container? [En línea]. Available: https://www.docker.com/resources/what-container. [Último acceso: 22 9 2020].

[57] L. Calcote y Z. Butcher. Istio: Up and Running. O'Reilly Media, Octubre 2019.

[58] The New Stack, Kubernetes: An Overview [En línea]. Available: https://thenewstack.io/kubernetes-an-overview/. [Último acceso: 27 9 2020].

[59] G. Hohpe y B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2003.

[60] A. Khatri y V. Khatri. Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linderk and Consul. Packt Publishing, Marzo 2020.

Apéndices

A. Glosario

Blue-Green Development: Estrategia de despliegue de software que en todo momento mantiene una copia del servidor de producción, utilizada con nuevo código para testeo. Una vez listo para el despliegue, se rutea a este servidor y se utiliza el anterior para testear un futuro release.

Canary Releases: Estrategia de despliegue de software que reduce el riesgo de introducir errores en producción, mediante el despliegue progresivo de los cambios a un porcentaje inicialmente reducido de los usuarios.

Citadel: Componente de Istio encargado del manejo de certificados y otros aspectos de seguridad: https://istio.io/v1.2/docs/concepts/security/

ConfigMaps: Kubernetes emplea ConfigMaps para guardar datos no confidenciales, en forma de <llave, valor> : https://kubernetes.io/docs/concepts/configuration/configmap/

Continuous Delivery: En el contexto del desarrollo de software, refiere a la habilidad de desplegar en producción de manera segura y rápida, cambios o nuevas funcionalidades sostenidamente.

docker-compose: Herramienta que permite emplear archivos YAML para configurar los servicios de aplicaciones contenerizadas utilizando Docker.

drag-and-drop: Refiere a la posibilidad de arrastrar un elemento virtual, y soltarlo en un nuevo lugar, desencadenando así una acción.

Eureka: Librería de Netflix que implementa Load Balancing: https://github.com/Netflix/eureka

Exchanges: Abstracción que refiere al intermediario a donde se envían los mensajes, en el contexto de intercambio de mensajes.

Fault Injections: Es una técnica empleada en el testeo de software que introduce fallas en el código de una aplicación para medir, por ejemplo, los niveles de tolerancia a fallos de la misma.

Grafana: Solución (*open-source*) de monitoreo y manejo de alertas, para aplicaciones de microservicios: https://grafana.com/grafana/

Graylog: Herramienta que permite visualización centralizada de los *logs* generados por los microservicios de una aplicación: https://www.graylog.org/

Hystrix: Librería de Netflix diseñada para aislar puntos de acceso a servicios o sistemas remotos en caso de fallas, mejorando la resiliencia de la aplicación que la utiliza**:** https://github.com/Netflix/Hystrix

istioctl: Herramienta de linea de commandos para Istio: https://istio.io/latest/docs/reference/commands/istioctl/

Itinerario: Refiere al concepto abstracto que modela la secuencia ordenada de microservicios que debe invocar el Orquestador, para llevar a cabo una solución de integración.

Jaeger: Herramienta de código abierto que implementa *Distributed Tracing,* para aplicaciones de microservicios: https://www.jaegertracing.io/

JWT (Jason Web Tokens): Estandard (RFC 7519) para crear tokens que permiten propagar identidad y privilegios sobre una red: https://jwt.io/

Kiali: Panel de administración integrable con Istio, que permite visualizaciones de los microservicios en tiempo real: https://kiali.io/

Kibana: Herramienta de visualización centralizada de los *logs* generados por los microservicios de una aplicación: https://www.elastic.co/kibana

Kompose: Herramienta que facilita la transición hacia Kubernetes, de usuarios acostumbrados a trabajar con **docker-compose**.

kubectl: Herramienta de linea de commandos para Kubernetes: https://kubernetes.io/docs/reference/kubectl/overview/

LightStep: Sistema de *Distributed Tracing*: https://docs.lightstep.com/

Logback: Herramienta de gestión de *logs:* http://logback.qos.ch/

Message Broker: En el contexto de comunicación mediante intercambio de mensajes, refiere a la solución de mensajería utilizada (por ejemplo, **RabbitMQ**).

Mesos: Apache Mesos abstrae tanto de máquinas reales como virtuales, la CPU, la memoria y otros recursos para permitir la construcción de sistemas distribuidos que sean elásticos y tolerantes a fallos: http://mesos.apache.org/

Modelo de comunicación publicador / subscriptor: Permite que una aplicación anuncie eventos de forma asincrónica, a varios consumidores interesados.

Mutual TLS: Refiere a la autenticación de ambas partes utilizando el protocolo **TLS**.

Oauth2: Protocolo ampliamente utilizado para autorización en redes: https://oauth.net/2/

out-of-the-box: Refiere a un objeto que está listo para utilizarse sin previa configuración.

Payload: En el contexto de enviar/recibir mensajes, el *Payload* de un mensaje refiere al contenido del mensaje, lo que originalmente se quiere enviar.

Prometheus: Solución (*open-source*) de monitoreo y manejo de alertas, para aplicaciones de microservicios: https://prometheus.io/

RabbitMQ: Broker de mensajería: https://www.rabbitmq.com/

Splunk Logging: Herramienta de gestión de *logs,* desarrollada en Java: https://github.com/splunk/splunk-library-javalogging

Spring Cloud Sleuth: Componente de trazabilidad, parte de la librería Spring Cloud: https://github.com/spring-cloud/spring-cloud-sleuth

Testeo A/B: Refiere a realizar el lanzamiento de dos versiones de un mismo elemento y medir cuál funciona mejor respecto a algún parámetro determinado.

TLS (Transport Layer Security): Protocolo de seguridad para comunicaciones sobre una red.

Traffic Shadowing/Mirroring: Estrategia de despliegue que copia el tráfico de producción asincrónicamente hacia un servicio que no está en producción y quiere ser testeado.

Virtual Machines (VMs): Herramienta que simula ser una computadora real, sobre la que es posible ejecutar distintos programas.

Weave Scope: Panel de administración integrable con Istio, que permite visualizaciones de los microservicios en tiempo real: https://www.weave.works/oss/scope/

Zipkin: Sistema de *Distributed Tracing*: https://zipkin.io/

B. Enterprise Integration Patterns

En 2003 G. Hohpe y B. Woolf introducen una serie de patrones basados en mensajería, para resolver problemas de integración a nivel empresarial [59]. Estos patrones, denominados Patrones de Integración Empresarial (EIP en inglés), son soluciones de diseño abstractas: no especifican aspectos de implementación o tecnología. La figura B.1 resume los patrones fundamentales, divididos en seis categorías: construcción de mensajes, ruteo de mensajes,

transformación de mensajes, terminales de mensajería, canales de mensajería y administración de sistemas.

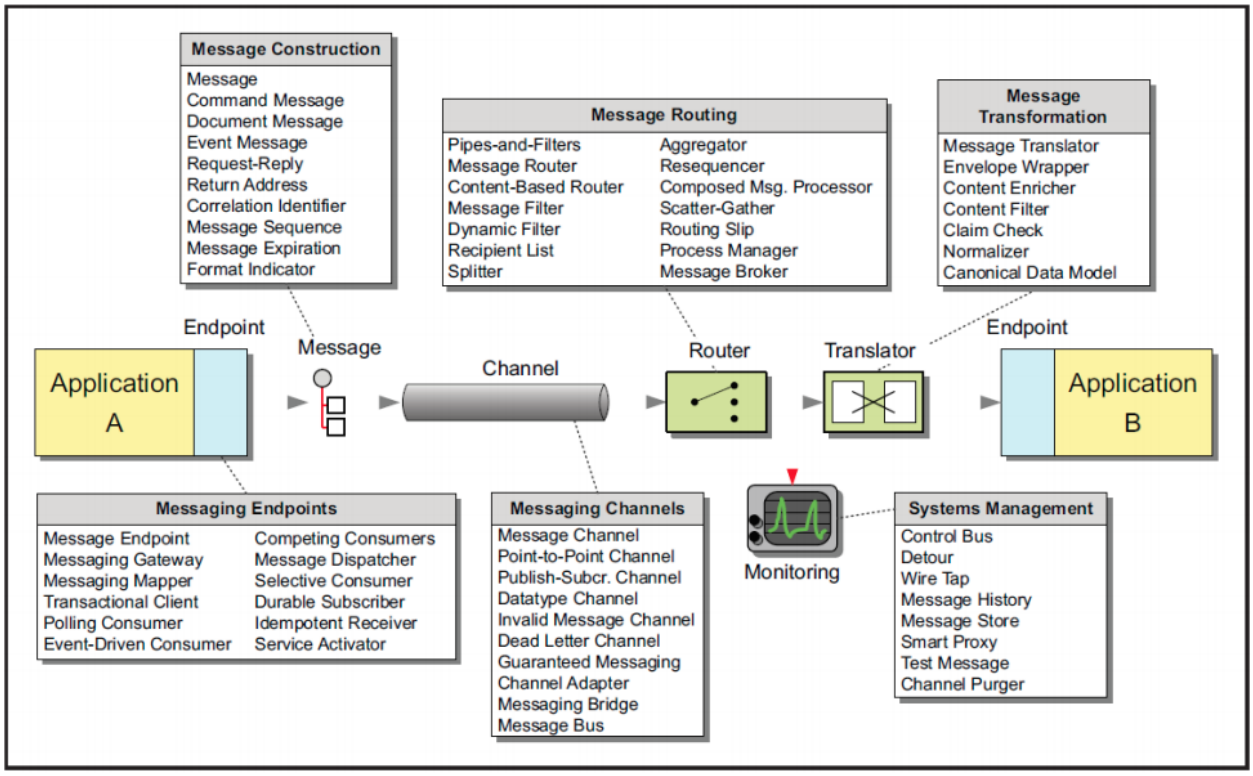

Figura B.1: resumen de las categorías en que se dividen los EIPs [9].

A modo de ejemplo y para que estén disponibles como referencia, se describen a continuación algunos de estos patrones, de interes para el corriente trabajo:

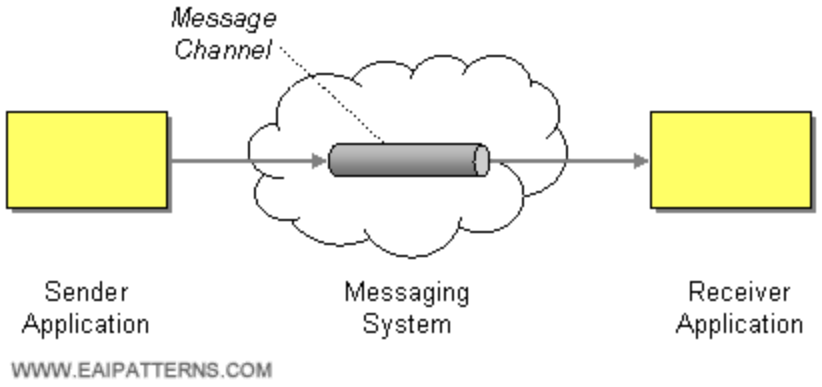

Figura B.2: EIP Message Channel [9].

Message Channel: Conecta a las aplicaciones utilizando un canal de mensajería, donde el emisor escribe información en el canal que luego el receptor lee, como se puede apreciar en la figura B.2.

Content-Based Router: La figura B.3 detalla cómo el *Content-Based Router* redirecciona cada mensaje al receptor correspondiente en base al contenido del mensaje.

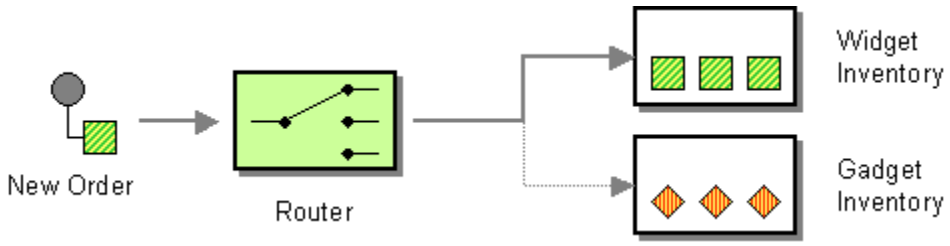

Figura B.3: EIP Content-Based Router [9].

Message Translator: Traduce el formato del mensaje entrante al formato especificado, como se aprecia en la figura B.4.

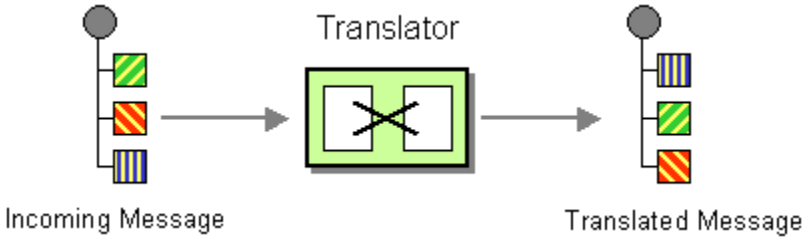

Figura B.4: EIP Message Translator [9].

Content Enricher: La figura B.5 muestra como el *Content Enricher* accede a una fuente de datos externa para aumentar los datos originales con información faltante. El Enriquecedor es un tipo especial de Transformador.

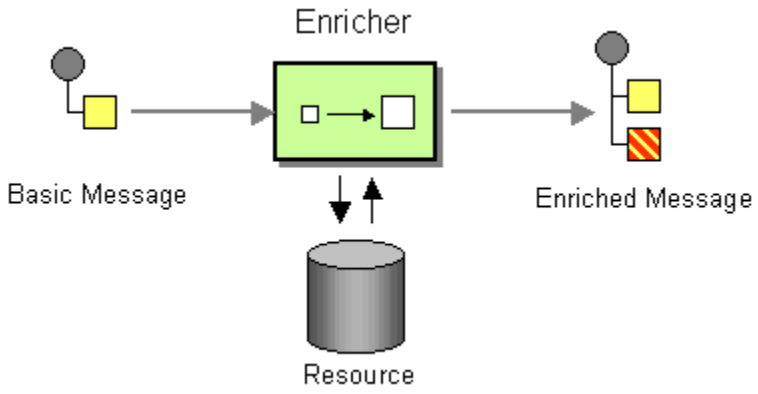

Figura B.5: EIP Content Enricher [9].

C. Análisis completo de funcionalidades para las Service Mesh.

Consul	Respuesta	Comentarios	Puntaje
¿Es una service mesh?	Sí	Consul (Service Discovery, Configuración, Load Balancing) + Connect (Autorización, Autenticación, Encriptación, Logs)	6
Circuit Breaker	Sí*	Disponible si se utiliza Envoy como dataplane.	4
Load Balancing	Sí*	RR por defecto. Otras posibilidades integrando con Fabio (ebay).	4
Service Discovery	Sí	Core functionality de Consul.	6
Tracing	Sí*	Disponible si se utiliza Envoy como dataplane.	4
Testing	No		0
Logging	Sí	Cada nodo genera sus propios logs, que pueden ser redirigidos hacia donde el usuario quiera. Es posible integrar Graylog con Consul también.	6
Seguridad: comunicación encriptada entre servicios intra-cluster.	Sí		6

Consul	Respuesta	Comentarios	Puntaje
Seguridad: comunicación encriptada con servicios externos.	Sí		6
Seguridad: encriptación con cliente (por ejemplo, un *browser*).	Sí		6
Seguridad: Autenticación intra-cluster, servicio a servicio.	Sí	Configurable desde la UI de Consul.	6
Seguridad: Autenticación de usuarios	Sí*	No soporta JWT. A partir de la versión 1.5, tanto servicios como usuarios finales puede autenticarse con Consul Connect mediante "Kubernetes Service Account" tokens. También es posible autenticarse mediante Envoy si se utiliza como proxy.	4
Seguridad: Autorización intra-cluster: qué operaciones puede realizar cada microservicio.	Sí		6
Seguridad: Autorización cliente: que operaciones puede realizar el cliente.	Sí	No es "out of the box": el proceso de "salir a producción" con Consul Connect demanda bastante trabajo y real entendimiento de las partes que lo componen. No hay mucho material al respecto salvo por guías (no exhaustivas) provistas por Hashicorp.	6
Seguridad: Integridad: llegan los mensajes intactos?	Sí		6
Puntaje total			**76**

Tabla C.1: Análisis de funcionalidades de Consul, versión extendida de la Tabla 4.2.

Linkerd	Respuesta	Comentarios	Puntaje
¿Es una service mesh?	Sí		6
Circuit Breaker	Sí	Disponible utilizando finagle. Puede ser utilizado a nivel de conexión o de request.	6
Load Balancing	Sí	Disponible utilizando finagle. P2C Least Loaded, P2C Peak EWMA, Aperture: Least Loaded, Heap: Least Loaded y round robin.	6
Service Discovery	Sí	ZooKeeper ServerSets, Consul, Kubernetes, Marathon, ZooKeeper Leader, Curator y Rancher.	6
Tracing	Sí	Zipkin.	6
Testing	No		0
Logging	Sí	Cada nodo genera sus propios logs, que pueden ser redirigidos hacia donde el usuario quiera. Es posible integrar Graylog con Consul también.	6
Seguridad: comunicación encriptada entre servicios intra-cluster.	Sí	TLS en cliente y servidor.	6
Seguridad: comunicación encriptada con servicios externos.	Sí		6
Seguridad: encriptación con cliente (por ejemplo, un *browser*).	Sí	TLS en servidor.	6
Seguridad: Autenticación intra-cluster, servicio a servicio.	Sí		6

Linkerd	Respuesta	Comentarios	Puntaje
Seguridad: Autenticación de usuarios	No		0
Seguridad: Autorización intra-cluster: qué operaciones puede realizar cada microservicio.	Sí	Se podría llegar a lograr configurando certificados TLS de cliente y servidor.	2
Seguridad: Autorización cliente: que operaciones puede realizar el cliente.	No		0
Seguridad: Auditoría.	Sí	TraceLog.	6
Seguridad: Integridad: los mensajes llegan intactos?	Sí		6
Puntaje total			**74**

Tabla C.2: Análisis de funcionalidades de Linkerd, versión extendida de la Tabla 4.2.

Istio	Respuesta	Comentarios	Puntaje
¿Es una service mesh?	Sí	*Control plane* + *data plane* (Envoy).	6
Circuit Breaker	Sí		6
Load Balancing	Sí	Pilot configura a los proxies Envoy para realizar balanceo de carga. Round Robin. Random. Weighted. Least Requests. Geographic location support. Supports sticky sessions.	6
Service Discovery	Sí	Aprovecha SD provisto por Kubernetes o plataforma subyacente.	6

Istio	Respuesta	Comentarios	Puntaje
Tracing	Sí	Aplicación debe propagar headers HTTP apropiados. Soporta trace sampling basado en porcentaje. Compatible con: Zipkin, Jaeger, LightStep.	6
Testing	Sí	Delays y aborts (400, 404, 503, etc) con porcentaje.	6
Logging	Sí	Soporta logs personalizados. Ejemplo en el stack Fluentd/ElasticSearch/Kibana.	6
Seguridad: comunicación encriptada entre servicios intra-cluster.	Sí	Acepta Permissive mode, acepta ambos, texto plano y tráfico mTLS al mismo tiempo. Mejor onboarding TLS.	6
Seguridad: comunicación encriptada con servicios externos.	Sí		6
Seguridad: encriptación con cliente (por ejemplo, un *browser*).	Sí		6
Seguridad: Autenticación intra-cluster, servicio a servicio.	Sí	Dos tipos: service to service y Origin authentication (end user authentication). JWT Y OpenID Connect (Auth0, Firebase Auth, Google Auth, etc).	6
Seguridad: Autenticación de usuarios	Sí	JWT con mTLS. SPIFFE implementation.	6
Seguridad: Autorización intra-cluster: qué operaciones puede realizar cada microservicio.	Sí	off, on, on_with_inclusion, on_with_exclusion ServiceRoles y ServiceRoleBinding who is allowed to do what under which conditions.	6

Istio	Respuesta	Comentarios	Puntaje
Seguridad: Autorización cliente: que operaciones puede realizar el cliente.	Sí		6
Seguridad: Auditoría.	Sí	Se usa los logs, que pueden ser personalizados y luego se visualizan gracias a Kibana y Fluentd. Los proxy envoy imprimen logs a stdout y se pueden definir reglas para dirigirlos a Fluentd.	6
Seguridad: Integridad: llegan los mensajes intactos?	Sí	TLS se encarga de la confidencialidad, integridad y autenticación. La cipher suite de Envoy incluye múltiples elecciones.	6
Puntaje total			**96**

Tabla C.3: Análisis de funcionalidades de Istio, versión extendida de la Tabla 4.2.

D. Definiciones de Service Mesh

Fuente	URL
NGINX	https://www.nginx.com/blog/what-is-a-service-mesh/
VMWare	https://www.vmware.com/topics/glossary/content/service-mesh
Red Hat	https://docs.openshift.com/container-platform/4.1/service_mesh/service_mesh_arch/understanding-ossm.html
Istio	https://istio.io/latest/docs/concepts/what-is-istio/
HashiCorp	https://www.hashicorp.com/resources/what-is-a-service-mesh
Glasnostic	https://glasnostic.com/blog/service-mesh-istio-limits-and-benefits-part-1
Buoyant	https://buoyant.io/2020/10/12/what-is-a-service-mesh
Envoy	https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc
InfoQ	https://www.infoq.com/articles/service-mesh-ultimate-guide/

Tabla D.1: Tabla de definiciones de Service Mesh consultadas durante el trabajo.

E. Setup de la solución

Se ha realizado la instalación de la plataforma en un sistema con las siguientes características descritas en la tabla E.1:

Nombre	Valor
Procesador	Intel(R) Core(™) i7-7700k CPU @ 4.20GHz 4.20 GHz x64
RAM	32 GB
Sistema Operativo	Windows 10 Pro Version 1909 OS build 18363.1379

Tabla E.1: Características del sistema donde se despliega la solución.

Se utiliza Hyper-V Manager Version 10.0.18362.1 para el manejo de las maquinas virtuales de Docker y Minikube en Windows.

Para la instalación de Minikube con la versión 1.14.2 de Kubernetes, 16GB de RAM y 4 CPUs se debe ejecutar:

minikube start --vm-driver hyperv --memory=16384 --cpus=4 --kubernetes-version=v1.14.2

Una vez finalizado el setup de Minikube se procede a instalar Rabbit con la ayuda de Helm[111]:

- helm init
- helm repo add "stable" "https://charts.helm.sh/stable"
- helm install --name broker --set rabbitmq.username=guest, rabbitmq.password = guest stable/rabbitmq

Se puede verificar la instalación de Kubernetes ejecutando el comando “kubectl get pods -n kube-system” que arroja la salida visible en la figura E.1:

[111] https://helm.sh/

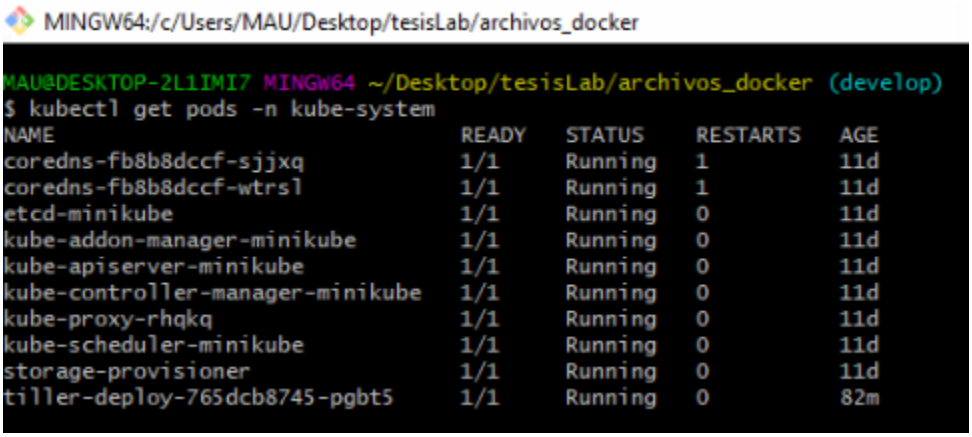

Figura E.1: Ejecución del comando “kubectl get pods -n kube-system”.

También se verifica la instalación de RabbitMQ, figura E.2:

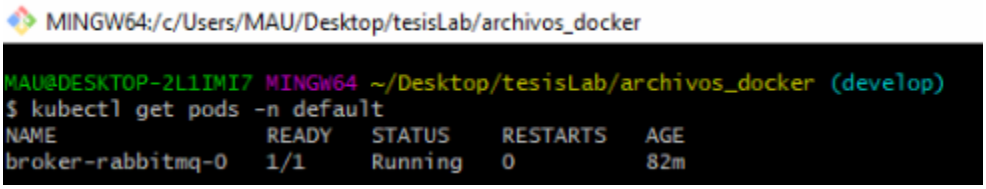

Figura E.2: Verificación de la correcta instalación de RabbitMQ.

A continuación se procede con el *build* de los microservicios. Los .jar generados deben moverse al directorio “tesis_lab/archivos_docker/files”. El siguiente paso es crear imágenes *docker* utilizando los archivos .jar. Las imágenes se crean dentro de Minikube. Para hacerlo primero se debe ejecutar el siguiente comando:

- eval $(minikube docker-env)

Luego se procede con la creación de las imágenes:

- cd tesis_lab/archivos_docker
- docker build --file=alpine_base --tag=alpine-jdk:base --rm=true .
- docker build -t conectorentrada -f conectorentrada .
- docker build -t conectorsalida -f conectorsalida .
- docker build -t enriquecedor -f enriquecedor .
- docker build -t orquestador -f orquestador .
- docker build -t clientefinalsoap -f clientefinalsoap .
- docker build -t transformacionjsonxml -f transformacionjsonxml .
- docker build -t transformacionxmljson -f transformacionxmljson .

Se pueden verificar la creación de las imágenes ejecutando el comando “docker images” que muestra las imágenes creadas dentro de MiniKube (imagen E.3):

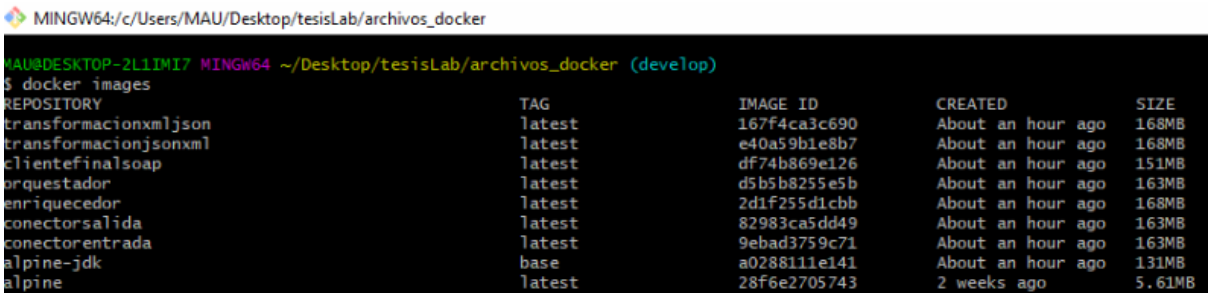

Figura E.3: Verificación de la correcta creación de las imágenes docker.

Para evitar escribir “kubectl” en la línea de comandos cada vez que se desea invocar un comando en la API de Kubernetes, es posible generar un alias de la siguiente forma:

- alias k=kubectl

De esta forma basta con escribir “k” en vez de “kubectl”.

Se procede con la instalación de Istio 1.2.5 en Minikube:

- cd tesisLab/istio1.2.5/istio-1.2.5
- k apply -f install/kubernetes/istio-demo.yaml

Se puede verificar la instalación de Istio con el comando “k get pods -n istio-system” (figura E.4).

A continuación se crea el *namespace* para la solución de integración a migrar, el cual se decide denominar “solucionintegracion1”:

- kubectl create namespace solucionintegracion1

Además se habilita la inyección automática de los *proxies* Envoy como *sidecars* en los *pods*.

- kubectl label namespace solucionintegracion1 istio-injection=enabled

Luego se procede con la creación de *ConfigMaps*:

- cd tesisLab/service_mesh/
- k create configmap logconfig --from-file=logback-spring.xml -n solucionintegracion1
- cd tesisLab/service_mesh/solucion_integracion_1/
- k create configmap piconfiguracionesconectorentrada --from-file=configuraciones/conectorentrada/conectorEntrada-docker.yml -n solucionintegracion1
- k create configmap piconfiguracionestransformacionjsonxml --from-file=configuraciones/transformacionjsonxml/transformacionjsonxml-docker.yml -n solucionintegracion1
  k create configmap piconfiguracionestransformacionxmljson --from-file=configuraciones/transformacionxmljson/transformacionxmljson-docker.yml -n solucionintegracion1

- k create configmap piconfiguracionesenriquecedor --from-file=configuraciones/enriquecedor/enriquecedor-docker.yml -n solucionintegracion1
- k create configmap piconfiguracionesconectorsalida --from-file=configuraciones/conectorsalida/conectorsalida-docker.yml -n solucionintegracion1
- k create configmap piconfiguracionesorquestador --from-file=configuraciones/orquestador/orquestador-docker.yml -n solucionintegracion1

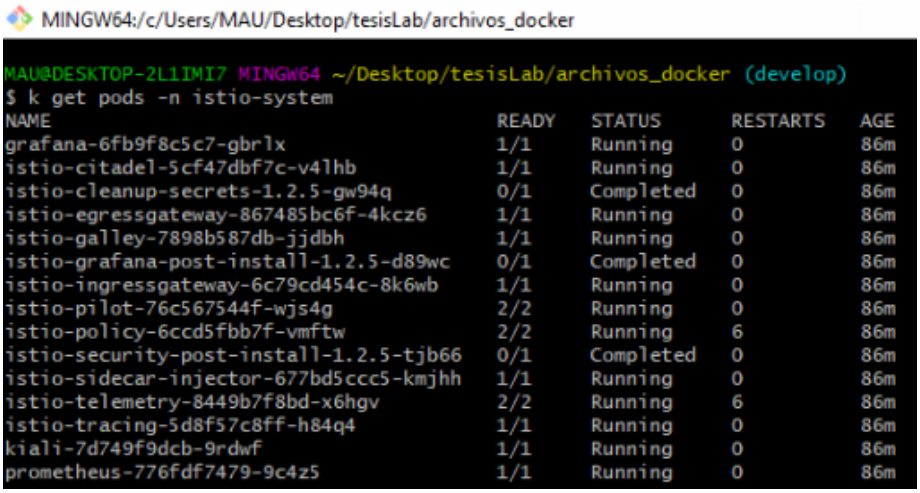

Figura E.4: Verificación de la correcta instalación de Istio.

El siguiente paso es crear los servicios para la solución de integración 1:

- k apply -f orquestador/orquestador-service.yaml -n solucionintegracion1
- k apply -f conector_entrada/conectorentrada-service.yaml -n solucionintegracion1
  k apply -f enriquecedor/enriquecedor-service.yaml -n solucionintegracion1
- k apply -f transformacion_xml_json/transformacionxmljson-service.yaml -n solucionintegracion1
- k apply -f transformacion_json_xml/transformacionjsonxml-service.yaml -n solucionintegracion1
- k apply -f conector_salida/conectorsalida-service.yaml -n solucionintegracion1
- k apply -f clientefinalsoap/clientefinalsoap-service.yaml -n solucionintegracion1

Y luego se crean los *Deployments*:

- k apply -f orquestador/orquestador-deployment.yaml -n solucionintegracion1
- k apply -f conector_entrada/conectorentrada-deployment.yaml -n solucionintegracion1
- k apply -f enriquecedor/enriquecedor-deployment.yaml -n solucionintegracion1
- k apply -f transformacion_xml_json/transformacionxmljson-deployment.yaml -n solucionintegracion1
- k apply -f transformacion_json_xml/transformacionjsonxml-deployment.yaml -n solucionintegracion1

- k apply -f conector_salida/conectorsalida-deployment.yaml -n solucionintegracion1
- k apply -f clientefinalsoap/clientefinalsoap-deployment.yaml -n solucionintegracion1

La figura E.5 muestra como se chequea el correcto despliegue tanto de los *pods* como de los servicios.

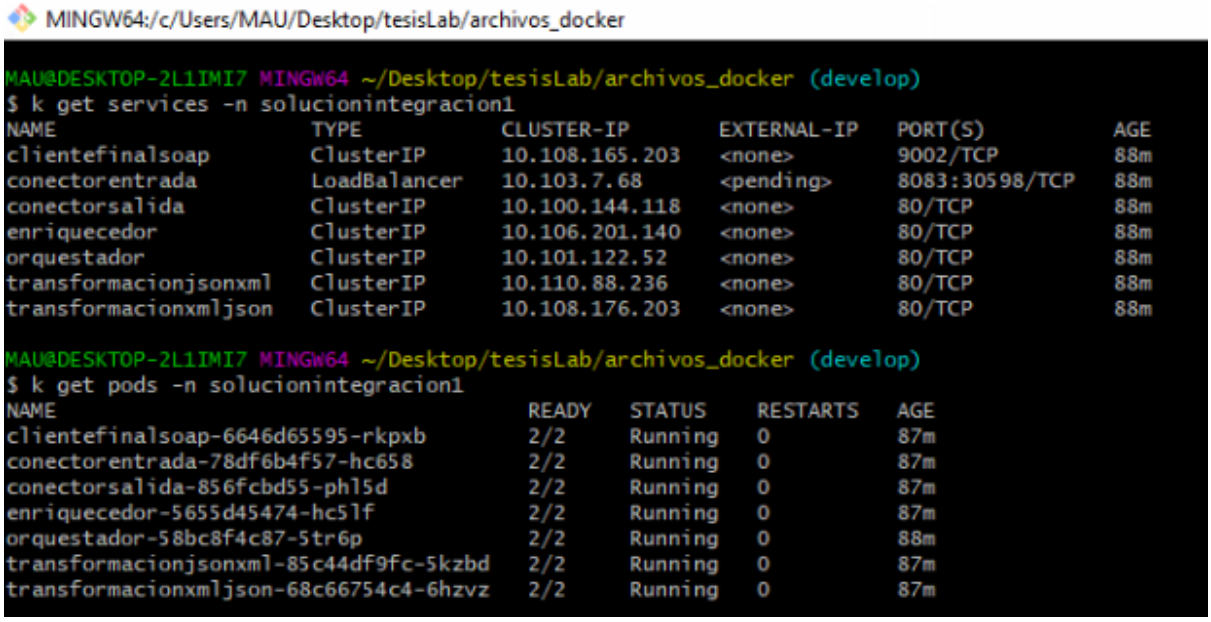

Figura E.5: Chequeo del despliegue de *pods* y servicios.

En la figura anterior se observa que cada *pod* tiene dos contenedores y por eso indica 2/2. Los contenedores corresponden al microservicio y al *proxy* Envoy que se inyecta automáticamente.

De esta forma culmina el *setup* de la solución de integración original, prosiguiendo con la instalación del nuevo caso de uso denominado “solución de integración 2”.

En este caso se necesita simular 3 sistemas externos: Ancel, Antel y AntelData los cuales van a ser desplegados con Docker en el *host*. Notar que el contexto es un sistema Windows por lo que existe una máquina virtual para Docker también gestionada por HyperV. El aspecto fundamental aquí es notar que Ancel, Antel y AntelData se despliegan fuera del cluster Minikube, simulando servicios externos, como se puede apreciar en la figura E.6.

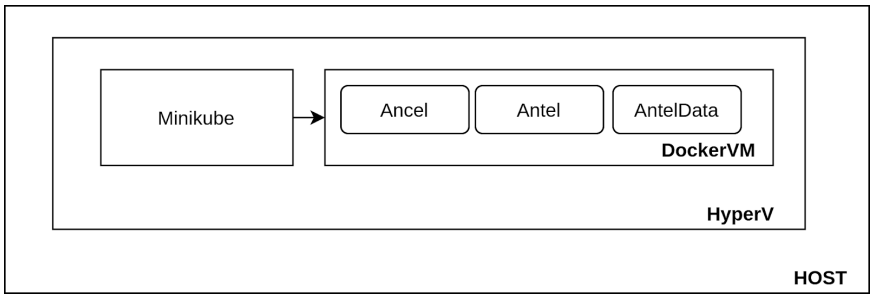

Figura E.6: Chequeo del despliegue de *pods* y servicios.

Para la instalación se deben ejecutar los siguientes comandos en una nueva consola:

1. cd tesisLab/archivos_docker/solucion_integracion_2
2. docker network create plataforma_comunes
3. docker-compose -p solucion_s2 -f s2-compose.yaml up --build

El requerimiento de que los comandos se ejecuten en una nueva consola es debido a que el comando “docker” tiene que apuntar al contexto del *host* y no de Minikube, de lo contrario los servicios serán creados dentro de MiniKube. La figura E.7 da cuenta de la ejecución del paso 2, mientras que la figura E.7 chequea los contenedores creados en el paso 3.

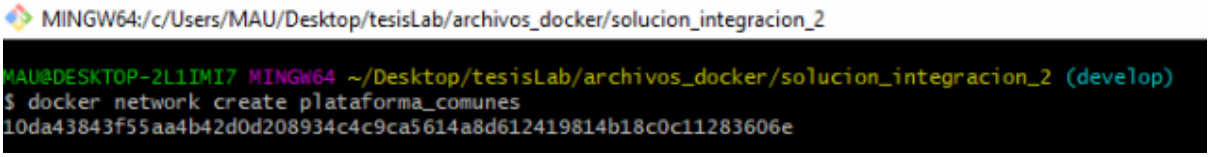

Figura E.7: Ejecución del comando “docker network create plataforma_comunes”.

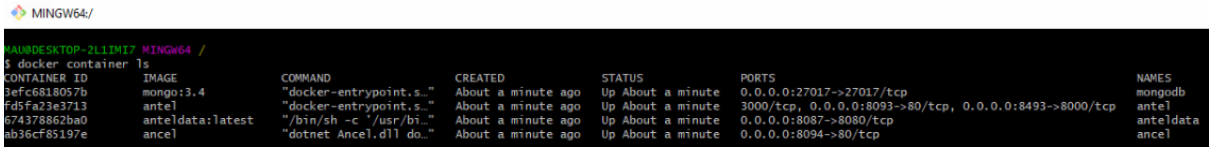

Figura E.8: Verificación de contenedores del paso 3.

Volviendo a la consola anterior, o de lo contrario asegurando que el contexto sea el correcto con el comando “eval $(minikube docker-env), se procede a realizar el *build* de los nuevos microservicios que se implementaron para la plataforma de integración:

- modificador-contenido
- router
- conector-salida-rest
- Además se realizaron modificaciones al microservicio orquestador y conectorentrada para la solución de integracion 2, generando el microservicio orquestador2 y conectorentrada2

Luego se prosigue con la ejecución de la siguiente serie de comandos:

- Generar .jar para orquestador2 y conectorentrada2. Mover .jar a tesisLab/archivos_docker/solucionintegracion2/files
- cd tesisLab/archivos_docker/solucionintegracion2
- docker build -t conectorentrada2 -f conectorentrada .
- docker build -t orquestador2 -f orquestador .
- cd tesisLab/plataformaintegracion
- docker build -t modificador-contenido -f modificador-contenido/Dockerfile .
- docker build -t router -f router/Dockerfile .
- docker build -t conector-salida-rest -f conector-salida-rest/Dockerfile .

Una vez terminados se prosigue chequeando que las imágenes hayan sido creadas exitosamente, como se muestra en la figura E.9.

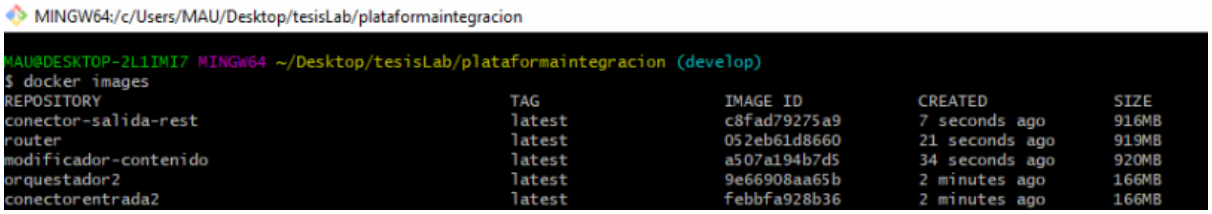

Figura E.9: Verificación de imágenes creadas.

A continuación se debe crear el namespace para la solucionintegracion2 y activar la inyección automática del *proxy* Envoy como *sidecar*:

- kubectl create namespace solucionintegracion2
- kubectl label namespace solucionintegracion2 istio-injection=enabled

El siguiente paso es realizar las configuraciones a través de *ConfigMap* para la solución integracion 2:

- cd tesisLab/service_mesh/
- k create configmap logconfig --from-file=logback-spring.xml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/configuraciones/orquestador/
- k create configmap piconfiguracionesorquestador2 --from-file=orquestador2-docker.yml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/configuraciones/conectorentrada/
- k create configmap piconfiguracionesconectorentrada2 --from-file=conectorEntrada2-docker.yml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/configuraciones/modificadorcontenido/

- k apply -f modificadorcontenido-docker.yaml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/configuraciones/router/
- k apply -f router-docker.yml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/configuraciones/conectorsalidarest/
- k apply -f conectorsalidarest-docker.yml -n solucionintegracion2

Se procede a desplegar servicios y *deployments* para la solución de integracion 2:

- cd tesisLab/service_mesh/solucion_integracion_2/orquestador/
- k apply -f orquestador2-service.yaml -n solucionintegracion2
- k apply -f orquestador2-deployment.yaml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/conector_entrada/
- k apply -f conectorentrada2-service.yaml -n solucionintegracion2
- k apply -f conectorentrada2-deployment.yaml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/modificador_contenido/
- k apply -f modificador-contenido-service.yaml -n solucionintegracion2
- k apply -f modificador-contenido-deployment.yaml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/router/
- k apply -f router-service.yaml -n solucionintegracion2
- k apply -f router-deployment.yaml -n solucionintegracion2

- cd tesisLab/service_mesh/solucion_integracion_2/conector_salida_rest/
- k apply -f conector-salida-rest-service.yaml -n solucionintegracion2
- k apply -f conector-salida-rest-deployment.yaml -n solucionintegracion2

La figura E.10 da cuenta del chequeo del despliegue anterior.

Por último se configura el punto de entrada al *cluster*, el *Ingress Gateway* y el *stack* EFK para el manejo de *logs*:

- cd tesisLab/service_mesh/
- k apply -f ingress-gateway.yaml
- tesisLab/service_mesh/solucion_integracion_1/
- k apply -f logging-stack.yaml

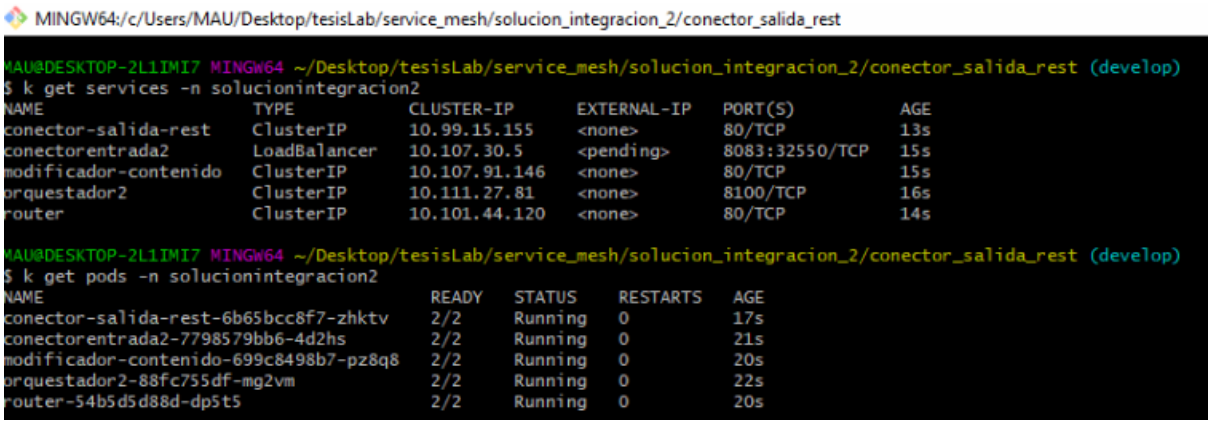

Figura E.9: Verificación del despliegue para la solución de integración 2.

En la figura E.11 se observa cómo se chequea el despliegue del Ingress Gateway. Luego, la figura E.12 da cuenta del chequeo del despliegue del *stack* EFK.

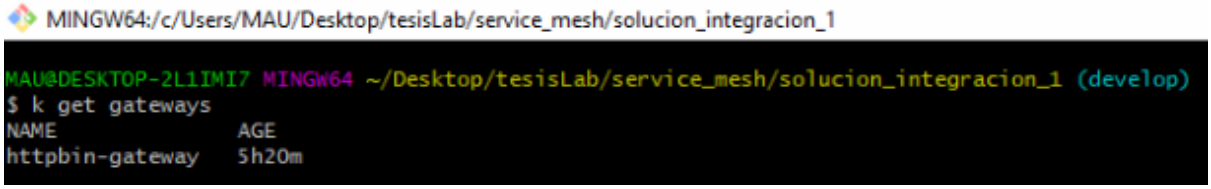

Figura E.11: Verificación del despliegue del Ingress Gateway.

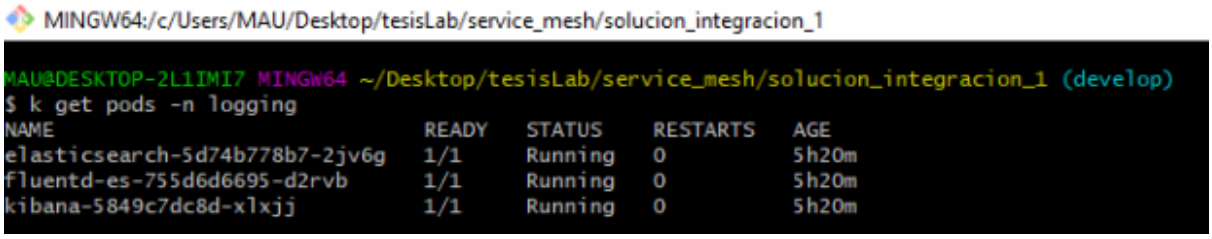

Figura E.12: Verificación del despliegue del *stack* EFK.

Una vez alcanzado este punto del proceso de despliegue de la solución, es posible enviar peticiones al *cluster*. Para averiguar la dirección IP y puerto al cual enviar las peticiones se deben ejecutar los siguientes comandos:

- export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}'), determina el puerto
- export INGRESS_HOST=$(minikube ip), determina la ip

En la figura E.13 se observa que es posible enviar peticiones a la IP 192.168.1.5 en el puerto 31380. Por su parte, la figura 6.32 muestra un ejemplo utilizando Postman[112] como cliente:

[112] https://www.postman.com/

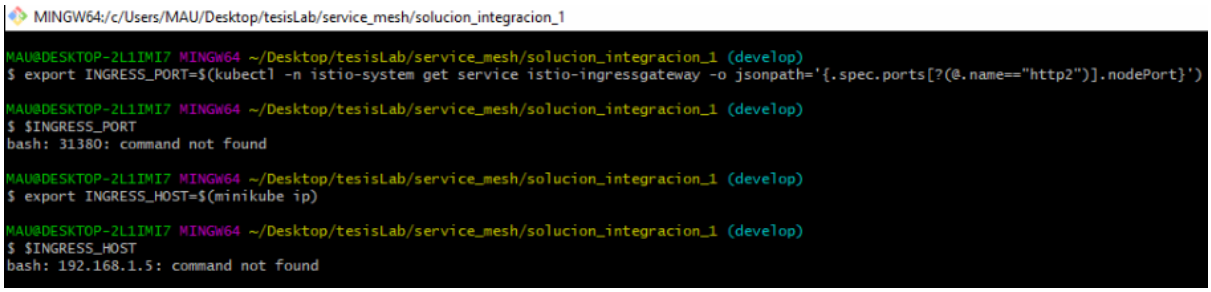

Figura E.13: Verificación de peticiones enviadas al *cluster* de la solución.

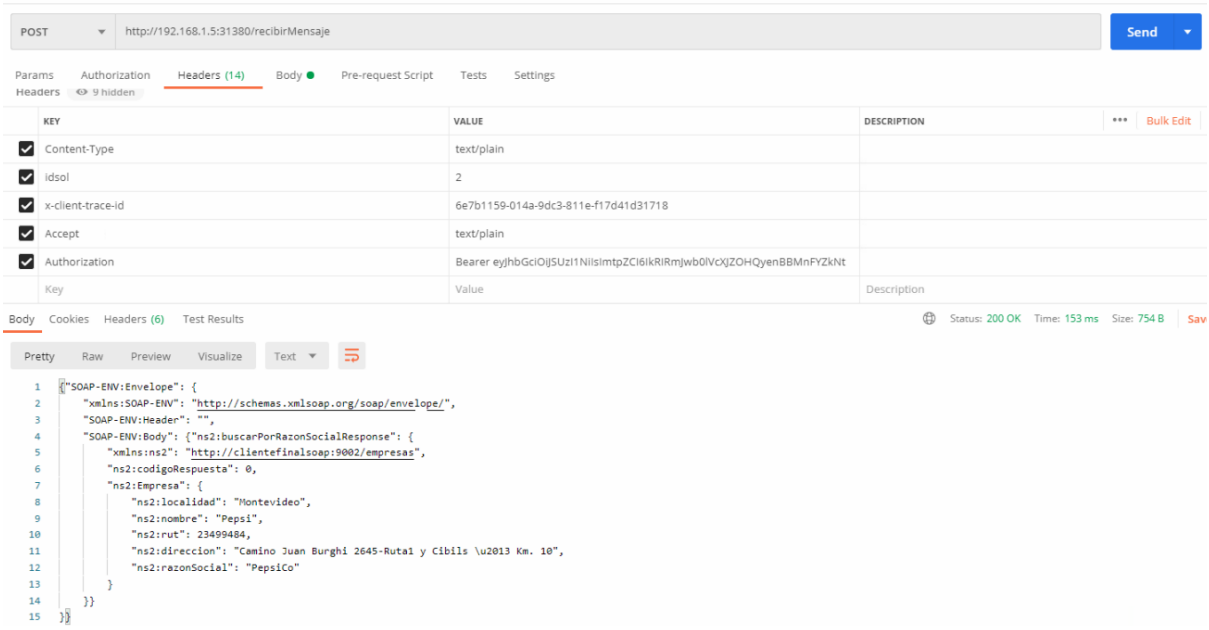

Figura E.14: Verificación de peticiones enviadas al *cluster* utilizando Postman como cliente.

Para exponer Jaeger y poder acceder al *dashboard* se debe ejecutar el siguiente comando:

- kubectl port-forward -n istio-system $(kubectl get pod -n istio-system -l app=jaeger -o jsonpath='{.items[0].metadata.name}') 16686:16686

Se podrá acceder desde el navegador a traves de http://localhost:16686, como muestra la figura E.15:

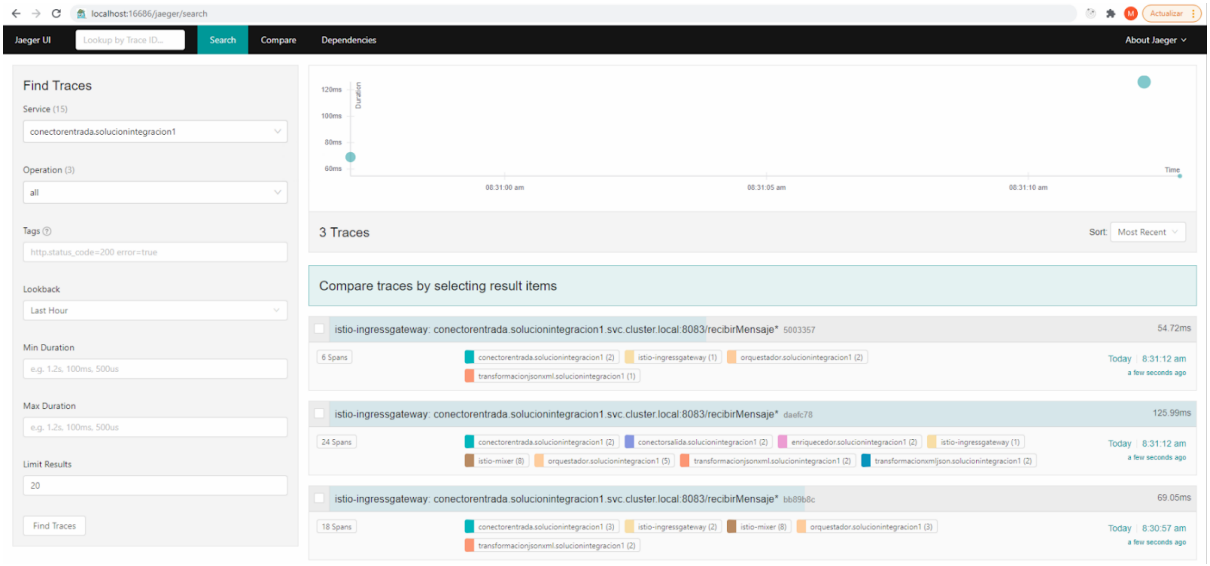

Figura E.15: Acceso a Jaeger a través de un navegador.

Para exponer Kibana y poder acceder al *dashboard* se debe ejecutar el siguiente comando:

- kubectl -n logging port-forward $(kubectl -n logging get pod -l app=kibana -o jsonpath='{.items[0].metadata.name}') 5601:5601

Se podrá acceder desde el navegador a través de http://localhost:5601 (figura E.16).

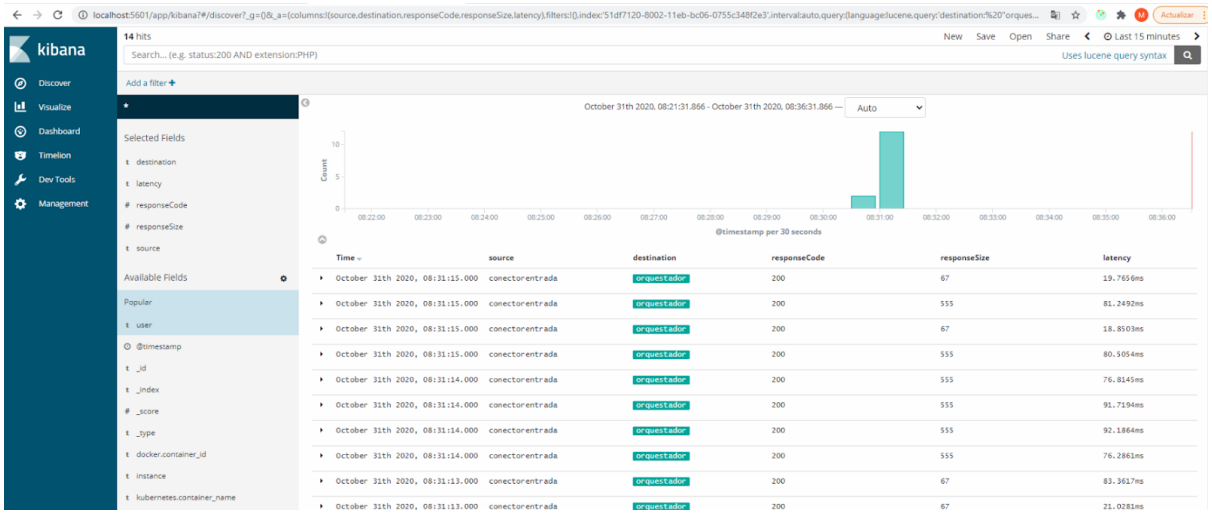

Figura E.16: Acceso a Kibana a través de un navegador.

También se instala Weave Scope, una herramienta de visualización y monitoreo para Kubernetes. Esta herramienta es externa a Istio pero resulta sumamente atractiva para visualizar el estado del *cluster*.

Para instalarla se debe ejecutar el comando:

- kubectl apply -f "https://cloud.weave.works/k8s/scope.yaml?k8s-version=$(kubectl version | base64 | tr -d '\n')"

La figura E.17 muestra el chequeo de la instalación de Weave Scope[113].

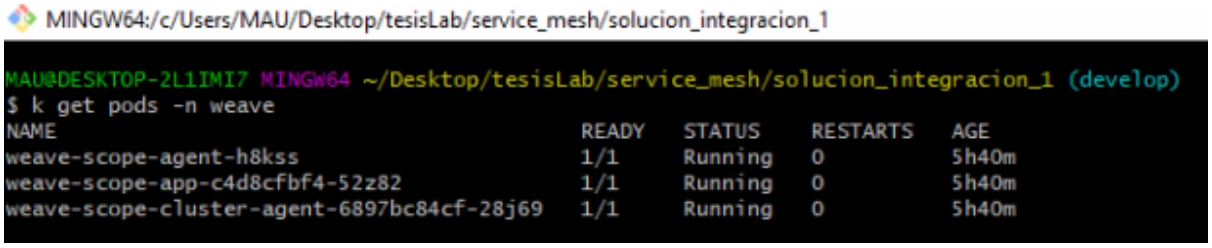

Figura E.17: Chequeo de instalación de Weave Scope.

Para poder acceder desde el navegador se ejecuta el siguiente comando:

- kubectl port-forward -n weave "$(kubectl get -n weave pod --selector=weave-scope-component=app -o jsonpath='{.items..metadata.name}')" 4040

La figura E.18 muestra una de las vistas de Weave Scope.

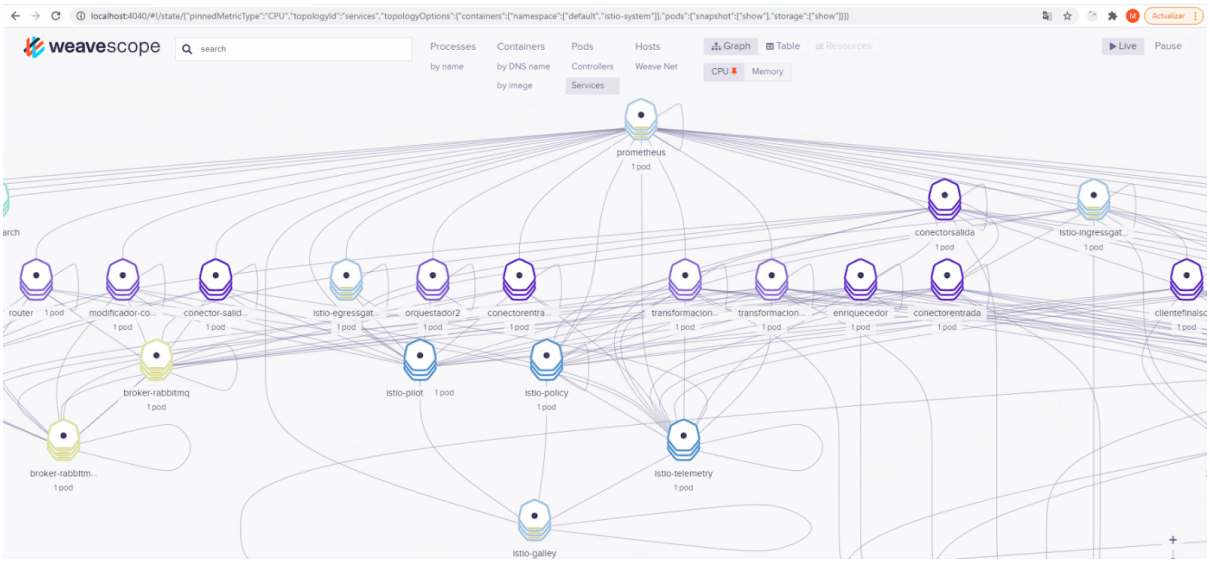

Figura E.18: Visualización del *cluster* utilizando Weave Scope.

Istio trae su propia herramienta con el mismo propósito. Para configurar Kiali y acceder desde el navegador se deben ejecutar los siguientes comandos:

- KIALI_USERNAME=$(echo "admin" | base64)
- KIALI_PASSPHRASE=$(echo "admin" | base64)

[113] https://www.weave.works/oss/scope/

```
cat <<EOF | k apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: $KIALI_USERNAME
  passphrase: $KIALI_PASSPHRASE
EOF
```

- kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=kiali -o jsonpath='{.items[0].metadata.name}') 20001:20001

Se puede acceder desde el navegador a través de http://localhost:20001. Un ejemplo de visualización con Kiali puede apreciarse en la figura E.19.

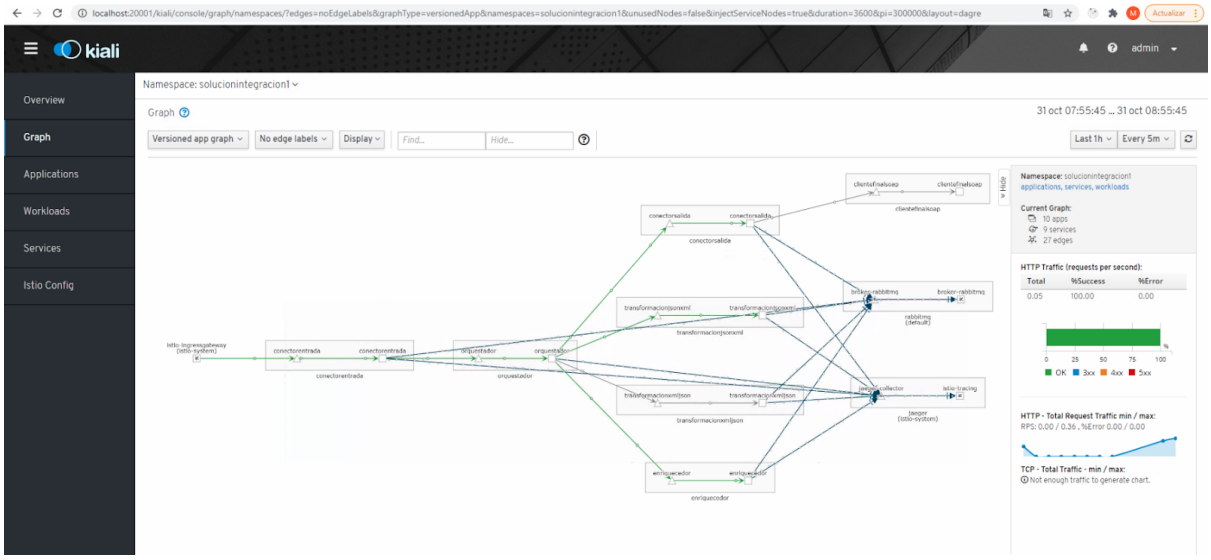

Figura E.19: Visualización del cluster utilizando Kiali.

F. Configuración de funcionalidades

Descubrimiento de servicios

Para la implementación de descubrimiento de servicios se crea un Service de Kubernetes por microservicio (relación 1:1 entre Deployment y Service Kubernetes). En su archivo de configuración YAML se realizan las siguientes declaraciones, descritas en la tabla F.1:

Una forma de visualizar un *Service* Kubernetes es pensando que estos apuntan a un *pod* o grupo de *pods* identificados por una etiqueta. Este mecanismo permite asociar un nombre a los microservicios, pudiendo así realizar descubrimiento de servicios sobre ellos.

Nombre	Valor	Descripción
kind	Service	Le indica al API server que se quiere crear un servicio.

Nombre	Valor	Descripción
selector.app	<nombre>	Indica que este servicio agrupa a aquellos *pods* que tengan un atributo *label* declarado con nombre <nombre>, permitiendo referirse a los microservicios por <nombre> e intercambiar mensajes por resolución DNS.
ports.port y ports.targetPort	<port>	Configuran un mapeo entre puertos.

Tabla F.1: YAML para un *Service* Kubernetes: en este caso para implementar descubrimiento de servicios.

Un Service se conforma del par IP:port, el tráfico que es enviado a esta dirección es redirigido hacia un pod en el puerto <targetPort>. Si <targetPort> y <containerPort> (definido en el Deployment asociado) son iguales, el microservicio será capaz de recibir este tráfico.

La figura F.1 ilustra cómo es posible visualizar un Service Kubernetes y la forma en que el tráfico se reparte entre los distintos *pods* bajo el mismo Service.

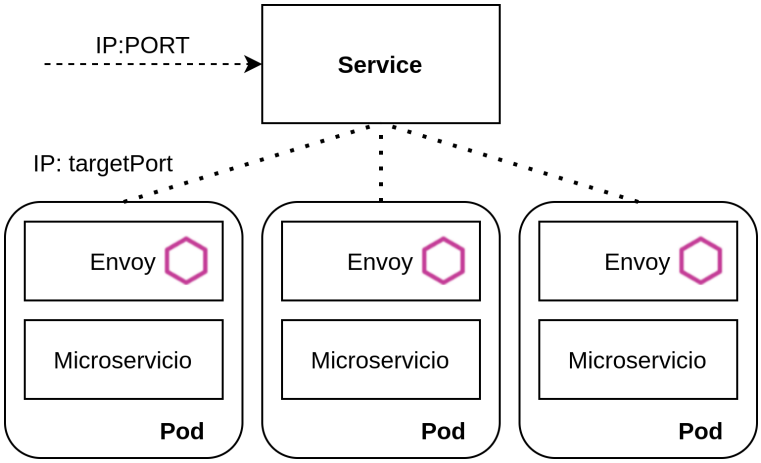

Figura F.1: *Service* Kubernetes.

Circuit Breaking

Para la implementación de *circuit breaking* se crea un Destination Rule de Istio. En su archivo de configuración YAML se realizan las siguientes declaraciones detalladas en la tabla F.2.

Nombre	Valor	Descripción
spec.host	<nombre>	Indica el nombre de un Service y por lo tanto, de su microservicio asociado.

Nombre	Valor	Descripción
outlierDetection.consecutiveErrors	<numero>	Cantidad de errores consecutivos hasta que un *pod* es eyectado y no permite más conexiones.
outlierDetection.interval	<numero>	Intervalo de tiempo para el análisis de errores consecutivos.
outlierDetection.baseEjectionTime	<numero>	Tiempo mínimo durante el cual un *pod* permanece eyectado sin recibir conexiones. En base a este atributo se calcula el tiempo de eyección, el cual aumenta según la cantidad de veces que el servicio falló.
outlierDetection.maxEjectionPercent	<numero>	Porcentaje máximo de *pods* que pueden ser eyectados.

Tabla F.2: YAML para un Destination Rule Istio que configura *Circuit Breaking*.

Balanceo de carga

Por defecto Istio realiza un balanceo de carga empleando el algoritmo Round-Robin. Para constatar que efectivamente se lleva a cabo es necesario contar con dos o más réplicas de un microservicio, actualizando el atributo “replicas” en el Deployment correspondiente.

En la solución desarrollada se incrementan a dos las réplicas del microservicio Enriquecedor, tras lo cual se define un nuevo Destination Rule, cuyo archivo YAML de configuración contiene las entradas descritas en la tabla F.3.

Nombre	Valor	Descripción
spec.host	<nombre>	Indica el nombre de un Service y por lo tanto de un microservicio.
trafficPolicy.loadBalancer.simple	RANDOM	Indica el algoritmo de balanceo de carga. Con esta configuración hemos pasado de un algoritmo Round-Robin a RANDOM por lo que el microservicio que atiende una petición es seleccionado en forma aleatoria.

Tabla F.3: YAML para un *Destination Rule* Sitio que configura balanceo de cargas.

Seguridad

Uno de los aspectos fundamentales para la seguridad de un sistema es el concepto de Identidad. Se necesita poder configurar a las entidades del mismo para poder determinar si una entidad “A” puede interactuar con otra entidad “B”. En base a la identidad se desarrollan los conceptos de Autenticación y Autorización en Istio.

En este contexto, Autenticación es el proceso de tomar una credencial (por ejemplo un certificado de seguridad) y verificar que la misma es válida. Esta credencial deberá contar con una identidad asociada. En Istio los servicios utilizan un certificado X.509[114] como credencial para comunicarse. La identidad del servicio se encuentra embebida en el certificado. Los *proxies* Envoy se encargan de realizar esta verificación en las comunicaciones servicio a servicio; cuando dos servicios se comunican, Envoy intercepta los mensajes y verifica la autenticidad del certificado X.509.

El concepto de Autorización se refiere a si una entidad, previamente autenticada, se encuentra habilitada para realizar una determinada operación. Istio implementa SPIFFE[115] (Secure Production Identity Framework For Everyone) el cual es un framework que especifica cómo

[114] https://istio.io/v1.2/docs/concepts/security/#pki
[115] https://spiffe.io/

representar y emitir identidades. La implementación contempla los 3 conceptos principales de SPIFFE:

1. El concepto de identidad, utilizada por los servicios para comunicarse. Esta identidad está representada como una URI. En el caso de Istio, se utiliza la plataforma subyacente como fuente de verdad (también denominado *trust domain*), en este caso Kubernetes. En resumen, la fuente de verdad es la que se encarga de darle un nombre a una entidad y se la asume confiable. Con Kubernetes podremos nombrar a las entidades usando el recurso ServiceAccount.

   Una vez definido el nombre de una entidad siguiendo el protocolo SPIFFE, Istio genera la identidad como la siguiente URI: spiffe://\<domain\>/ns/\<namespace\>/sa/\<serviceaccount\>

2. SPIFFE codifica la identidad en un SVID (SPIFFE Verifiable Identity Document). En el contexto de Istio, este documento toma la forma de certificado X.509 y la identidad o URI se encuentra en el atributo SAN del mismo.

3. Una API para emitir los SVID. Citadel, que forma parte del panel de control de Istio, es quien se encarga de emitir certificados X.509.

La figura F.2 ilustra el proceso descrito anteriormente a alto nivel, en el contexto Minikube de la solución desarrollada. A su vez la tabla F.4 contiene una breve descripción del paso a paso presente en la figura F.2

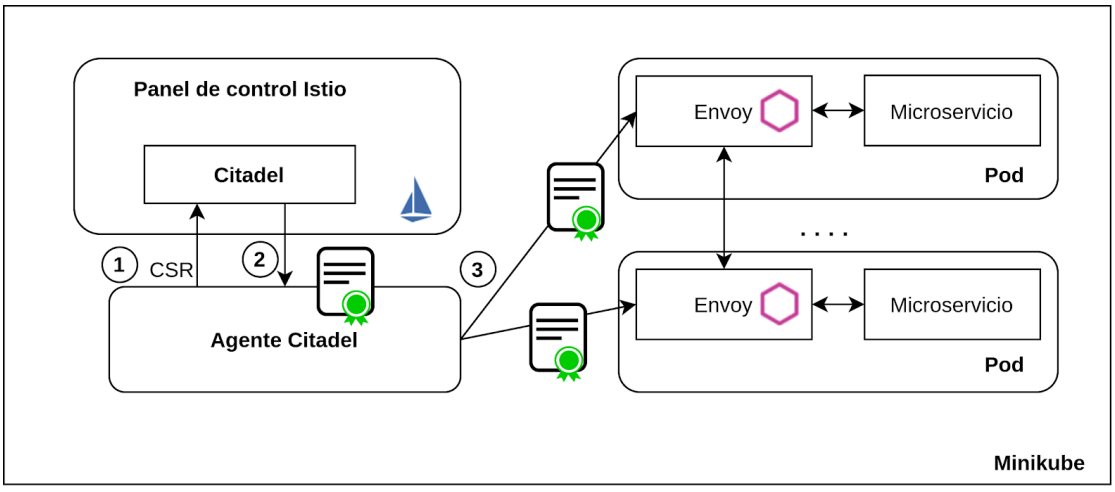

Figura F.2: Petición CRS y emisión e intercambio de los certificados X.509.

Paso	Descripción
1	El Agente Citadel desplegado en el único nodo de la solución (Minikube), envía una petición CSR (Certificate Signing Request) a Citadel a través de su API. Este agente actúa en nombre de las entidades que solicitan una identidad.
2	Citadel transforma el CSR en un certificado X.509 con la identidad embebida en el atributo SAN y envía al Agente Citadel.
3	El agente envía los certificados a los *proxies* Envoy.

Tabla F.4: Paso a paso de la figura F.2 detallado.

Luego de ser enviados estos certificados a los *proxies* Envoy, los microservicios de la solución cuentan con identidades asignadas. Esto permite implementar autenticación y autorización. Al intercambiar certificados dos *proxies* Envoy pueden autenticarse y establecer una conexión, como se muestra en la figura F.3.

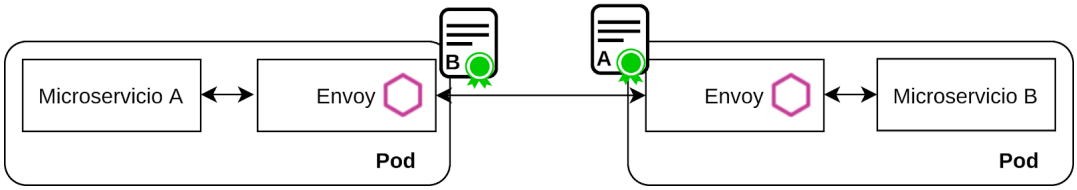

Figura F.3: Intercambio de certificados entre *proxies* Envoy de dos microservicios.

Tras el doble intercambio de certificados realizado por los proxies diremos que estos se han conectado vía mTLS (mutual TLS), lo hace posible encriptar las comunicaciones entre los microservicios protegiendo los mensajes que se intercambian y evitando ataques del estilo *man-in-the-middle.*

El primer paso en la implementación consiste en definir identidades para los servicios. En Kubernetes se puede utilizar un ServiceAccount para definir la identidad de un *pod*. Luego, Istio podrá utilizar esa identidad para crear una identidad que sigue el formato SPIFFE.

Nombre	Valor	Descripción
kind	ServiceAccount	Le indica al API server que se quiere crear un *ServiceAccount*.
metadata.name	proygrado-orquestador	Identificador del *pod*, el cual es posible referenciar en el recurso Deployment de un microservicio.

Tabla F.5: YAML para un ServiceAccount que define la identidad de un *pod*.

A partir de contar con un ServiceAccount, es posible referenciarlo desde el Deployment de un microservicio. Por ejemplo en el *deployment* del microservicio Orquestador se utiliza la referencia “serviceAccountName: proygrado-orquestador”, indicando el ServiceAccount a utilizar por el *pod*.

Una vez realizadas estas declaraciones, Istio podrá generar una identidad para el microservicio Orquestador siguiendo el formato SPIFFE. La figura F.4 permite ver la identidad generada por Istio para el microservicio. Se observa la identidad como una URI en el atributo SAN del certificado X.509 que reside en el *proxy* Envoy:

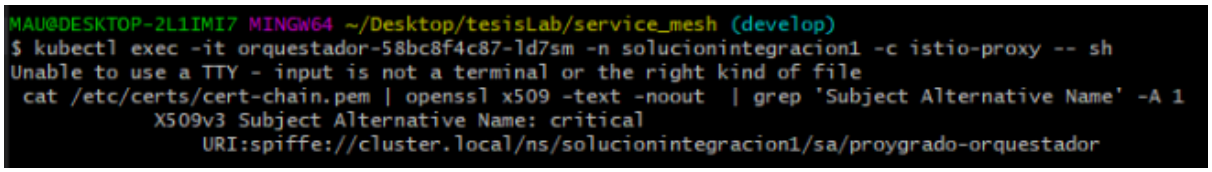

Figura F.4: Ejemplo de identidad generada para el microservicio Orquestador. La identidad es spiffe://cluster.local/ns/solucionintegracion1/sa/proygrado-orquestado

El segundo paso en la implementación es definir políticas de autenticación. La solución se desarrolla en una primera etapa sin autenticación ni autorización, y luego se procede a requerir autenticación, exigiendo el intercambio de certificados entre los proxies (mTLS).

Para realizar las configuraciones se utilizan los CRDs MeshPolicy y DestinationRule. El primero permite definir políticas de autenticación que afectan a toda la Service Mesh. En su archivo de configuración YAML podremos encontrar las declaraciones resumidas en la tabla F.6.

Nombre	Valor	Descripción
kind	MeshPolicy	Indica que es una política que aplica a toda la Service Mesh.
metadata.name	default	Es un requerimiento de Istio que su nombre sea *default*.
spec.peers.mtls.mode	PERMISSIVE	Indica que los servicios en la Service Mesh aceptan peticiones encriptadas utilizando TLS, permitiendo también comunicaciones por texto plano (sin encriptación).

Tabla F.6: YAML para un MeshPolicy Istio que define políticas de autenticación que aplican a la totalidad de la Service Mesh.

La configuración anterior establece políticas de autenticación solamente para los receptores en un intercambio de mensajes. Para poder completar una comunicación con mTLS se debe establecer la política para los emisores, configurando DestinationRules que utilicen mTLS. En la tabla F.7 puede consultarse un ejemplo de YAML para una DestinationRule que fuerza TLS por parte del emisor.

Nombre	Valor	Descripción
kind	DestinationRule	Le indica al API server que se quiere crear un DestinationRule.
namespace	solucionintegracion1	Indica que la configuración se aplica solo en el namespace solucionintegracion1.
spec.host	*.solucionintegracion1.svc.cluster.local	Indica con un "*" que la configuración debe aplicarse sobre todos los servicios que cumplan con la expresión.
spec.trafficPolicy.tls.mode	ISTIO_MUTUAL	Configura al servicio para utilizar mTLS.

Tabla F.7: YAML para un DestinationRule Istio que aplica TLS en los emisores de mensajes.

El tercer paso de la implementación consiste en definir políticas de autorización. Básicamente se define qué servicios se pueden comunicar entre sí. Istio utiliza un sistema RBAC[116] (Role-based access control). Las tablas F.8, F.9 y F.10 enumeran las entradas de los archivos YAML para estos recursos.

Nombre	Valor	Descripción
kind	ClusterRbacConfig	Le indica al API server que se quiere crear un ClusterRbacConfig.
spec.mode	ON_WITH_INCLUSION	Indica que se deberá verificar políticas de autorización solo para aquellos servicios especificados en el atributo spec.inclusion.service

Tabla F.8: YAML para un ClusterRbacConfig Istio que indica para qué servicios es necesario chequear políticas de autorización.

[116] https://istio.io/v1.1/docs/reference/config/authorization/istio.rbac.v1alpha1/

La combinación de *ServiceRole* y *ServiceRoleBinding* especifica “quién” tiene permitido hacer “que”.

Nombre	Valor	Descripción
kind	ServiceRoleBinding	Le indica al API server que se quiere crear un *ServiceRoleBinding*.
spec.subjects.user	cluster.local/ns/solucionintegracion1/sa/proygrado-orquestador	Indica quién tiene permitido realizar lo que indica el *ServiceRole* asociado.

Tabla F.9: YAML para un *ServiceRoleBinding* Istio que le adjudica a un *ServiceRole* a un servicio.

Por último se configura autenticación de usuario final (*end-user authentication* o *origin authentication*) via JWT. Esto brinda la posibilidad de autenticar a los usuarios finales que envían peticiones a la plataforma de integración. Istio permite configurar emisores válidos para los JWT que se envíen.

Nombre	Valor	Descripción
kind	ServiceRole	Le indica al API server que se quiere crear un ServiceRole.
spec.rules.methods	GET, POST	Define las operaciones que se pueden realizar, como por ejemplo las operaciones GET y POST.

Tabla F.10: YAML para un ServiceRole Istio que define qué métodos le están permitidos al *ServiceRoleBinding* asociado.

Para configurar autenticación de usuario final se utiliza el CRD de tipo Policy con los siguientes atributos descritos en la tabla F.11. Por su parte la figura F.5 ilustra cómo se aplica autenticación de usuario final.

Nombre	Valor	Descripción
kind	ServiceRole	Le indica al API server que se quiere crear un ServiceRole.

spec.targets.name	conectorentrada.solucionintegracion1.svc.cluster.local	Especifica que la política sea aplicada sobre el microservicio “conectorentrada”.
origins.jwt.issuer	testing@secure.istio.io	Especifica que solo se aceptarán JWTs emitidos por testing@secure.istio.io
origins.jwt.jwksUri	https://raw.githubusercontent.com/istio/istio/release-1.2/security/tools/jwt/samples/jwks.json	Especifica el *endpoint* JWKS[117], que contiene las claves públicas para verificar el JWT recibido.

Tabla F.11: YAML para un *Policy* Istio que define configura autenticación para usuarios finales.

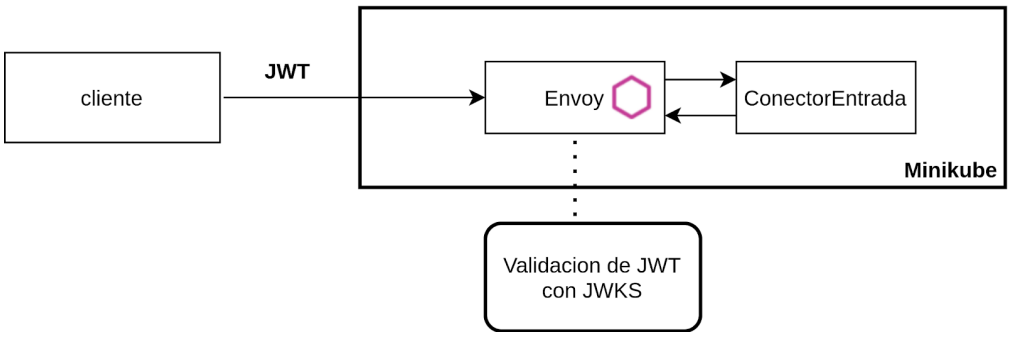

Figura F.5: Autenticación de usuario final mediante JWT.

Testing

Istio permite inyectar errores en la capa de aplicación con soporte para protocolo HTTP. La inyección de errores es un método de testeo que permite introducir errores en un sistema para estudiar su comportamiento y capacidad de recuperación.

La inyección de errores (*fault-injection*) en Istio es una herramienta conveniente ya que evita que el operador de la plataforma tenga que simular la caída de *pods*, cambios en la latencia y/o errores en paquetes en la capa TCP.

El objetivo principal es testear la capacidad de recuperación punto a punto (*end-to-end*) del sistema como un todo, para evitar una mala experiencia de usuario.

Istio permite la inyección de dos tipos de errores:

[117] https://auth0.com/docs/tokens/json-web-tokens/json-web-key-sets

1. *Delays*: permiten agregar latencias para, por ejemplo, simular congestión en el sistema.
2. *Aborts*: permiten simular fallas en los servicios, generalmente toman la forma de errores HTTP o errores en conexiones TCP.

Resulta interesante observar que si se inyecta un error de tipo *abort* se puede simular un error HTTP sin la necesidad de que la petición sea enviada al microservicio de destino. El *proxy* de origen no envía la petición, sino que simplemente genera el error y lo envía al microservicio correspondiente.

En la figura F.6 se puede observar que en caso de que se haya configurado un error de tipo *abort,* el *proxy* Envoy de origen no envía la petición all *proxy* de destino (1) y en cambio simula la respuesta y le envia el codigo de error al microservicio de origen (2).

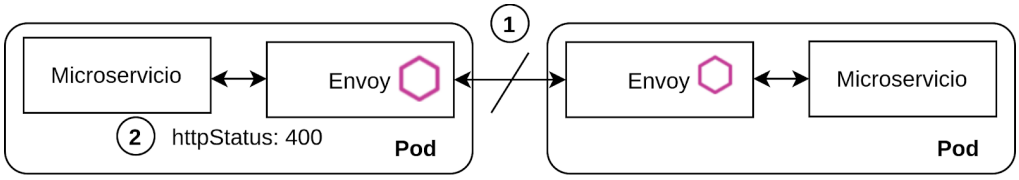

Figura F.6: Visualización del comportamiento de los proxies al inyectar errores del tipo *abort*.

Para realizar la configuración se utiliza un VirtualService donde se pueden configurar los siguientes campos descritos en la tabla F.12:

Nombre	Valor	Descripción
kind	VirtualService	Le indica al API server que se quiere crear un VirtualService.
http.fault.abort.httpStatus	<numero>	Indica que se debe abortar con error HTTP de tipo “número” (int32).
http.fault.abort.percentage.value	<numero>	indica el porcentaje de las peticiones que deben abortar con el error http.fault.abort.httpStatus.
http.fault.delay.fixedDelay	<value>	Indica la latencia a simular antes de enviar la petición al siguiente servicio. “value” tiene el formato 1h/1m/1s/1ms.
http.fault.delay.percentage.value	<numero>	Indica el porcentaje de las peticiones que deben simular la latencia.

Tabla F.12: YAML para un VirtualService Istio que define una inyección de errores.

Las inyecciones de errores en Istio aprovechan las configuraciones soportadas por VirtualService, por lo que se pueden definir reglas de tipo *match* que permiten aplicar la inyección de falta sólo si la regla se cumple. Esto resulta muy conveniente, pudiendo configurar errores y testeos sin la necesidad de realizar cambios en el código. Simplemente cambiando un *header* en la petición por ejemplo, es posible ejecutar distintos casos de prueba, como en el siguiente ejemplo:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: enriquecedorfaultinjection
spec:
  hosts:
  - enriquecedor
  http:
    - fault:
        abort:
          httpStatus: 400
          percentage:
            value: 100
      match:
        - headers:
            x-client-trace-id:
              exact: 6e7b1159-014a-9dc3-811e-f17d41d31719
      route:
        - destination:
            host: enriquecedor
    - route:
        - destination:
            host: enriquecedor
      match:
        - headers:
            x-client-trace-id:
              exact: 6e7b1159-014a-9dc3-811e-f17d41d31720
      fault:
        delay:
          percentage:
            value: 100
          fixedDelay: 7s
```

En el ejemplo anterior se configura un VirtualService que simula un error de tipo 400 en caso de que el *header* “x-client-trace-id” de la petición del cliente tenga el valor “e7b1159-014a-9dc3-811e-f17d41d31719”. En cambio si el *header* contiene el valor “6e7b1159-014a-9dc3-811e-f17d41d31720” simula una latencia de 7 segundos. Para la latencia el *proxy* de origen espera 7 segundos antes de enviar la petición al *proxy* destino. En la figura F.7 se observa la inyección de un error 400 y cómo esto se procesa a nivel de Istio.

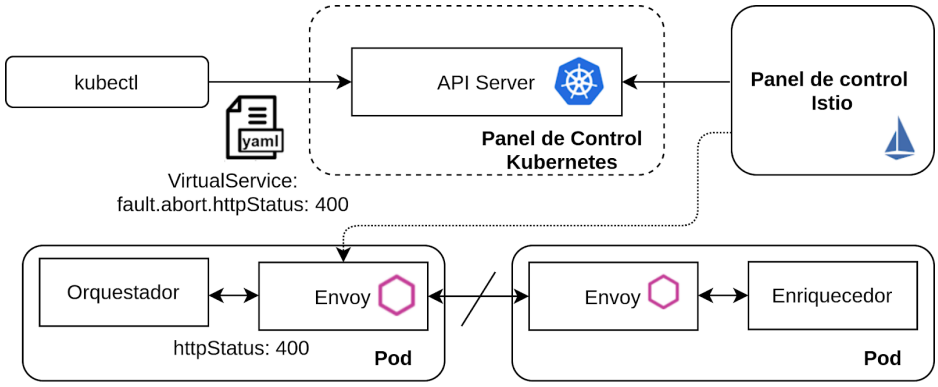

Figura F.7: Inyección de un error tipo 400.

La solución desarrollada permite sólamente inyectar errores para poder evaluar la plataforma de integración. Es responsabilidad de los desarrolladores de los microservicios implementar un buen manejo de errores. En el ejemplo de la figura F.7, el microservicio debe ser capaz de atrapar errores de tipo 400 y evaluar cómo proseguir.

Configuración centralizada

Para la configuración centralizada se busca remover las dependencias con Spring Cloud Config de forma tal que la plataforma sea lenguaje agnóstica, permitiendo que los microservicios que sean implementados en otro lenguaje o con otro *framework* no necesitan integrarse con Spring Cloud Config.

Kubernetes a través de ConfigMaps permite implementar configuración centralizada. Se utiliza a su vez, la variable de entorno SPRING_APPLICATION_JSON presente en la implementación en la plataforma de integración original.

Un ConfigMap es un mapa que contiene pares clave-valor, donde los valores pueden ser literales o archivos de configuración. De esta forma, es posible pasar el contenido de un ConfigMap a un contenedor como variable de entorno, a la cual puede acceder luego cada microservicio.

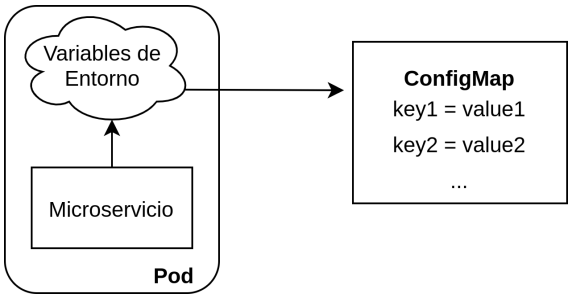

Figura F.8: Diagrama simplificado de acceso de un microservicio a su *ConfigMap* a través de una variable de entorno.

En el contexto de trabajo de esta solución interesa popular la variable de entorno SPRING_APPLICATION_JSON, siendo que es utilizada por Spring para cargar configuración en las aplicaciones.

Por su parte, los archivos de configuración de la plataforma de integración se migran a archivos que puedan ser referenciados para generar ConfigMaps en la plataforma. Kubernetes permite la creación de ConfigMaps a partir de archivos con el siguiente comando:

“kubectl create configmap ConfigMapName --from-file=FileName”

El comando anterior crea el ConfigMap con nombre “ConfigMapName” a partir del archivo “FileName”, almacenando el contenido del archivo “FileName” bajo la clave “FileName”.

Por ejemplo para la configuración del microservicio “Enriquecedor” se crea el YAML “enriquecedor-docker.yaml” y se ejecuta el comando anterior con los valores de la tabla F.14.

La figura F.9 ilustra la carga del ConfigMap del microservicio “Enriquecedor”.

Nombre	Valor
ConfigMapName	piconfiguracionesenriquecedor
Filename	configuraciones/conectorentrada/enriquecedor-docker.yml

Tabla F.14: YAML con la configuración del microservicio “Enriquecedor”.

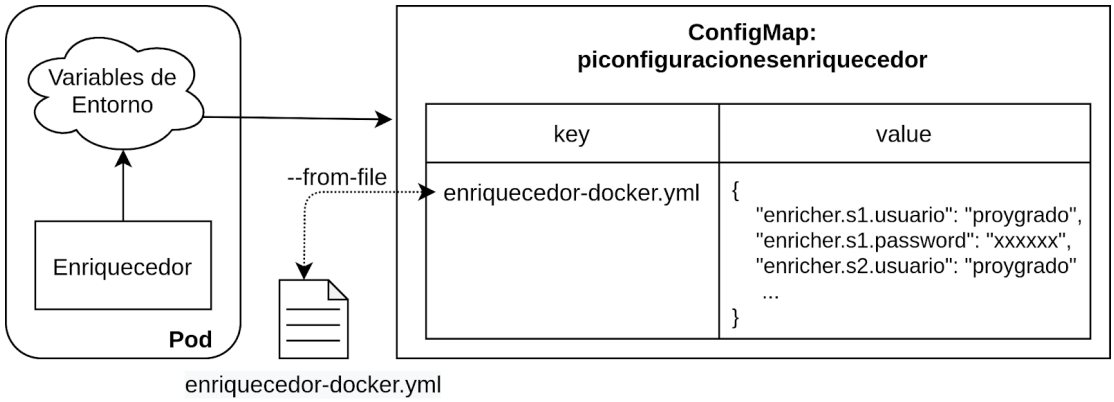

Figura F.9: Diagrama detallado de acceso del microservicio “Enriquecedor” a su ConfigMap a través de una variable de entorno. El diagrama a su vez muestra cómo sucede la carga del ConfigMap.

Luego en el Deployment del microservicio “Enriquecedor” se puede configurar la variable de entorno SPRING_APPLICATION_JSON para tomar el valor de la clave “enriquecedor-docker.yaml” del ConfigMap “piconfiguracionesenriquecedor”.