Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Informe proyecto de grado

Plataforma de Integración basada en Microservicios

Autores
Juan Bonhomme
Enrique Camejo

Tutores
MSc. Ing. Laura González
MSc. Ing. Guzmán Llambías

Octubre 2019

Resumen

En los nuevos contextos en los que operan las organizaciones, los procesos de negocio se llevan a cabo integrando servicios ofrecidos por cada una de ellas. Las Plataformas de Integración consisten en infraestructuras especializadas que brindan mecanismos que facilitan la integración de estos servicios, resolviendo incompatibilidades entre sistemas heterogéneos con el fin de posibilitar su comunicación.

Por otra parte, la arquitectura de microservicios se enfoca en desarrollar una aplicación como un conjunto de pequeños servicios y brinda ventajas en aspectos de alta disponibilidad y tolerancia a fallos. Estas ventajas podrían beneficiar a las Plataformas de Integración, que tradicionalmente poseen arquitecturas monolíticas.

En este sentido, en un trabajo previo se estudió la aplicabilidad de la arquitectura de microservicios para la construcción de Plataformas de Integración. Se propusieron alternativas de arquitectura y diseño, cuya factibilidad técnica fue validada realizando pruebas de concepto. Sin embargo, en dicho trabajo no se realizó la implementación de la Plataforma de Integración.

En este proyecto se realiza una implementación de la Plataforma de Integración propuesta en el trabajo previo mencionado, para algunos escenarios específicos.

Se comienza identificando los requerimientos que esta plataforma debe cumplir. Luego, se realiza un refinamiento de estos y se define el alcance final del proyecto. Se determina que la plataforma cubra los escenarios de Transformación y Enriquecimiento de mensajes. También se desea realizar un seguimiento de manera centralizada de los registros de eventos o acciones de los microservicios y se quiere que la plataforma sea escalable.

Posteriormente se da paso a diseñar la plataforma, enfocándose en los componentes disponibles en la misma y en la forma de comunicación entre ellos. Se define que los estilos de coordinación de los componentes de integración presentes en las soluciones sean coreografía u orquestación.

Finalmente, se implementa la plataforma de integración, elaborando un caso de estudio para evaluar la factibilidad técnica de la propuesta. Respecto a las tecnologías utilizadas, se destaca el *stack* tecnológico de Spring Cloud para la implementación de los microservicios, que se despliegan sobre el gestor de contenedores Docker. Para el registro de eventos centralizado se opta por Graylog, y se elige RabbitMQ como middleware orientado a mensajes para la comunicación de estilo coreográfico en la plataforma.

Palabras claves: plataforma de integración, arquitectura de microservicios, coreografía, orquestación, contenedores.

Contenido

1 Introducción

Desde hace unos años, las arquitecturas de microservicios han sido un patrón de diseño para aplicaciones en la nube con requerimientos de hiper escalabilidad con un nivel de granularidad más avanzado que lo que permiten aplicaciones basadas en arquitecturas monolíticas. Este patrón de arquitectura fue evolucionando con el correr de los años, donde los *frameworks* y plataformas de desarrollo comenzaron a brindar un mejor soporte a esta técnica.

Sin embargo, esto no ocurre así con las Plataformas de Integración en la nube existentes en el mercado, cuya arquitectura monolítica no permite un nivel de escalabilidad de granularidad fina. Elastic.io, RoboMQ y Ritc son esfuerzos que van en ese sentido, pero presentan funcionalidades básicas de integración, a diferencia de las Plataformas de Integración tradicionales monolíticas, que guían los flujos de integración en los patrones *Enterprise Integration Patterns* (EIP) [1].

Una primera aproximación a la construcción de una Plataforma de Integración basada en microservicios fue la propuesta de Andrés Nebel [2], en donde a nivel de implementación se realizaron pruebas de concepto, pero no existe una implementación de referencia que valide su propuesta.

1.1 Objetivos

Este proyecto tiene como principal objetivo el desarrollo de la Plataforma de Integración propuesta en la tesis mencionada anteriormente, permitiendo implementar sus flujos de integración según los patrones de EIP. Se pretende validar la propuesta y ver que tan factible es técnicamente.

Los objetivos específicos del proyecto son:

- Relevar en la industria iniciativas en la construcción de Plataformas de Integración basadas en microservicios o Plataformas de Integración en la nube.
- Diseñar una Plataforma de Integración basada en microservicios, tomando como base lo realizado en la tesis de referencia.
- Implementar un prototipo que valide la propuesta.

1.2 Aportes

Los principales aportes del proyecto son:

- Análisis del marco conceptual en las áreas de microservicios y Plataformas de Integración.
- Implementación de una Plataforma de Integración, tomando como base una arquitectura de referencia.
- Diseño e implementación de componentes de integración para la plataforma, que puedan ser reutilizados en diferentes Soluciones de Integración.
- Diseño de una arquitectura extensible a nuevos componentes y funcionalidades que puedan ser implementadas a futuro, para continuar con el desarrollo del producto.
- Logro de primera versión de implementación del producto. Debido a que en la industria no hay información ni documentación referente a la implementación de las Plataformas de Integración, se puede considerar este proyecto como una referencia de implementación.
- Conocimiento técnico de plataformas e infraestructura base para poder implementar una plataforma con las características requeridas.

1.3 Organización del documento

El resto del documento se organiza de la siguiente manera:

En el Capítulo 2 se realiza un resumen del marco teórico necesario para comprender el resto del documento.

En el Capítulo 3 se presenta un análisis de la problemática planteada, se describen los requerimientos del proyecto y se establece el alcance final del mismo.

En el Capítulo 4 se elabora el diseño de la solución propuesta, mostrando cómo interactúan los diferentes componentes y las principales decisiones de diseño tomadas.

En el Capítulo 5 se brindan aspectos relevantes de la implementación, presentando un caso de estudio y las pruebas realizadas sobre el mismo.

En el Capítulo 6 se describe la gestión del proyecto.

Por último, en el Capítulo 7 se presentan las conclusiones del proyecto y posibles trabajos a futuro.

2 Marco conceptual

El marco conceptual que se desarrolla a continuación permite conocer los conceptos básicos necesarios para el entendimiento del desarrollo de este proyecto. En primera instancia, en la Sección 2.1 se describe el middleware orientado a mensajería y se presentan modelos arquitectónicos de comunicación para el mismo. A continuación, en la Sección 2.2 se describen conceptos generales asociados a Plataforma de Integración. Luego, en la Sección 2.3 se introduce la definición de microservicios, mencionando componentes o servicios comunes. Posteriormente, en la Sección 2.4 se presentan trabajos relacionados con plataformas de integración basadas en microservicios. Finalmente, en la Sección 2.5 se presentan beneficios de utilizar contenedores para implementar microservicios.

2.1 Middleware orientado a mensajería

Middleware representa una capa de software que se encuentra entre la capa superior (aplicaciones y usuarios), y la inferior (sistemas operativos y mecanismos de comunicación básicos). Provee una misma interfaz y medio de comunicación para diferentes aplicaciones, y al mismo tiempo oculta las diferencias de plataformas de comunicación, hardware, sistemas operativos y lenguajes de programación, simplificando el desarrollo de sistemas complejos con distintas tecnologías y arquitecturas (Figura 2.1) [3].

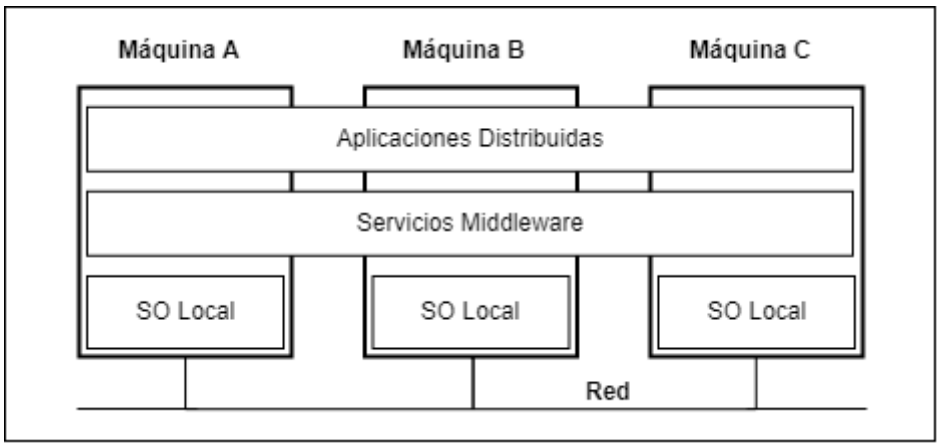

Figura 2.1: Concepto de middleware

Una capa middleware, en general brinda las siguientes funcionalidades dentro de un sistema distribuido [4]:

- *Servicio de descubrimiento*: localiza mensajes, recursos y otros servicios dentro del sistema.
- *Seguridad*: brinda un marco de seguridad en la comunicación entre procesos locales y remotos.
- *Representación horaria universal*: provee un formato universal para la representación horaria en plataformas ejecutando en zonas horarias distintas. Resulta crítico para el mantenimiento de tareas de sincronización entre procesos.
- *Sistema de transacciones*: proporciona transacciones semánticas para garantizar la integridad de los datos, fundamentalmente para asegurar actualizaciones en bases de datos.

Los Middleware Orientado a Mensajería (MOM) [5] apuntan al intercambio de mensajes entre procesos de forma asíncrona. Las aplicaciones solo se encargan de "poner" y "quitar" mensajes de las colas, no conectándose directamente entre ellas. Los términos de cliente y servidor no existen, por lo que se pierde el concepto de petición de servicio. Cualquier entidad puede participar en el intercambio de información en cualquier instante, por lo que no es necesario esperar respuesta, y por ende existe desacoplamiento en la comunicación.

Existen dos modelos de comunicación dentro de este tipo de middleware [5]:

- **Punto a punto (p2p)**
  Se compone por dos nodos de comunicación: uno emisor y otro receptor. Este modelo garantiza la llegada del mensaje, ya que se le envía al destinatario, aunque este no se encuentre disponible. El mensaje se deposita en una cola, para que el receptor pueda aceptarlo cuando se conecte.
- **Publicador/suscriptor**
  Existen diferentes nodos de comunicación, que pueden ser publicadores, suscriptores, o ambos al mismo tiempo. La información se genera o consume en un mensaje o evento, que sirve como vínculo entre publicadores y suscriptores. Este modelo de comunicación brinda una

potente abstracción para comunicación en grupo y multidifusión. Los publicadores pueden propagar una determinada información a un grupo de suscriptores, que están a la escucha de esta difusión de mensajes.

Existen cuatro modelos arquitectónicos de comunicación dentro de publicador/suscriptor [6]:

- *Centralizada con intermediarios:* Un servidor central sirve como punto de enlace entre todas las entidades y todo el tráfico pasa por él (Figura 2.2).

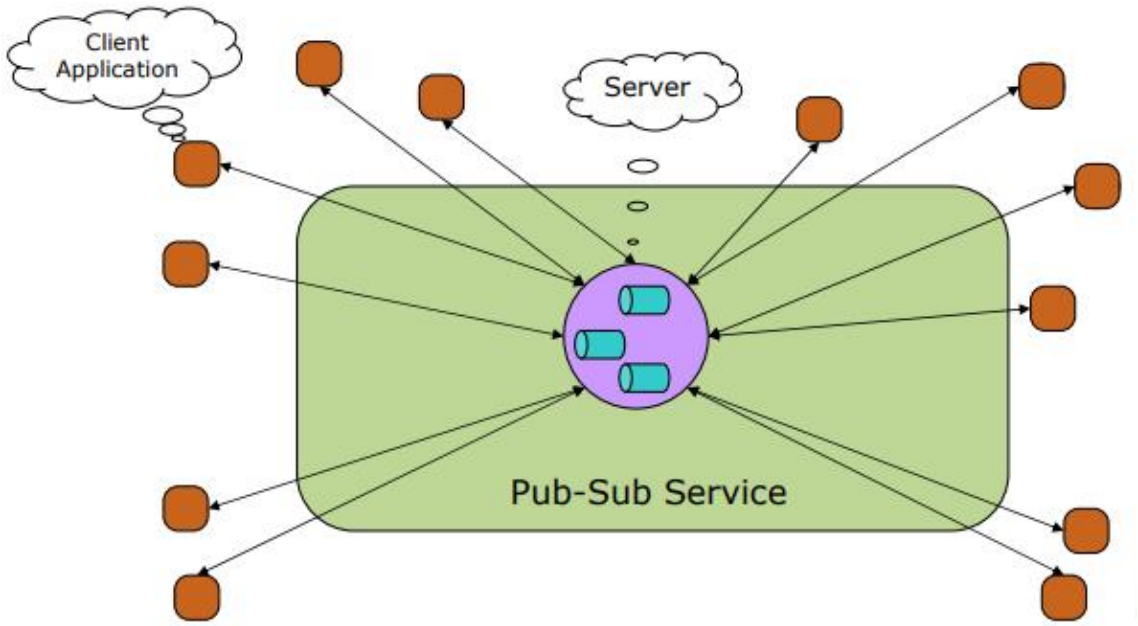

Figura 2.2: Arquitectura centralizada con intermediarios [6]

- *Centralizada con múltiples intermediarios:* Cada cola se encuentra en servidores distintos, interconectados entre ellos.
- *Descentralizada con múltiples intermediarios:* El servicio publicador/suscriptor distribuye los mensajes internamente entre Puntos de Acceso.
- *Descentralizada sin intermediarios:* No existen servidores. Los clientes se comunican punto a punto.

A continuación, se describe una de las tecnologías de MOM.

Advanced Message Queuing Protocol (AMQP[1]) es un protocolo estándar, abierto para la comunicación a través de un MOM, donde su objetivo central es lograr la interoperabilidad entre diferentes sistemas. Surge en 2004 de la mano de empresas como RabbitMQ[2]. A diferencia de otras tecnologías, AMQP no solamente define una API, sino también es un protocolo que define el formato de los datos que son enviados a través de la red como un flujo de octetos. Por lo tanto, cualquier aplicación puede crear e interpretar mensajes siguiendo ese formato de datos, independientemente del lenguaje utilizado.

Brinda un espacio de colas compartido, accesible para todas las aplicaciones que pretendan intercambiar mensajes. El enrutamiento de estos es llevado a cabo por los intermediarios o *brokers*, que redirigen los datos a su destino, basándose en una clave única, que es un identificador asociado a cada mensaje. El mensaje tiene una estructura fija, con propiedades opcionales relacionadas con la prioridad de los datos, el tiempo de vida o modalidad de entrega. El *broker* almacena los datos en una cola si la aplicación suscriptora no puede procesarlos cuando lleguen.

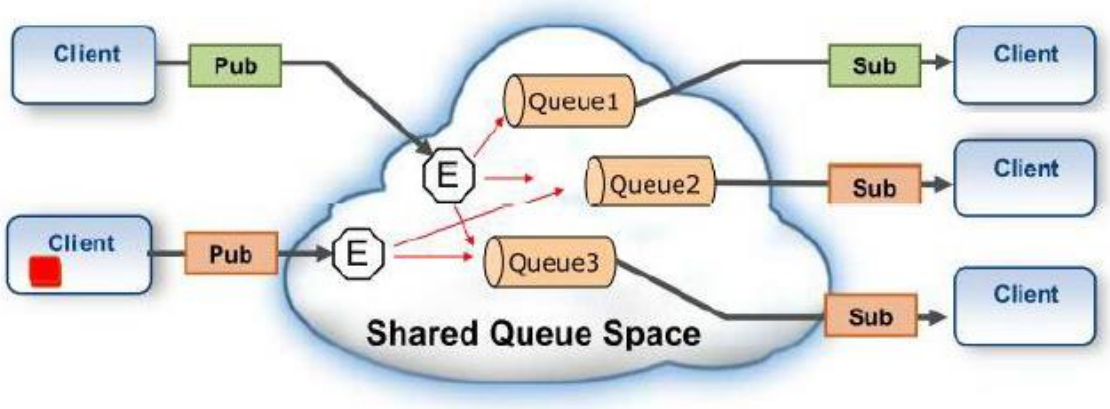

Figura 2.3: Tecnología AMQP [6]

Como se puede observar en la Figura 2.3, los clientes consumidores de información son los que se suscriben a estas colas, recibiendo los mensajes almacenados en la misma de forma activa por parte del *broker* de mensajes.

[1] https://www.amqp.org/

[2] https://www.rabbitmq.com/

Alternativamente, estos clientes pueden obtener los mensajes de la cola como crean conveniente. La cola asegura que los mensajes sean entregados en el mismo orden en el que llegan, cumpliendo con el mecanismo FIFO (*First in, first out*). Las colas también incluyen características asociadas a la persistencia de los mensajes, exclusividad por cliente, perdurable frente a reinicio del *broker*, entre otras propiedades.

Los intercambiadores (*exchanges*) son las entidades a las que se envían los mensajes. Las vinculaciones (*bindings*) especifican como fluyen los mensajes entre un intercambiador (*exchange*) a una cola (*queue*). Existen las vinculaciones condicionales con patrón de reconocimiento, que implica que la vinculación tenga una propiedad y la clave de enrutamiento.

Existen los intercambiadores de tipo temático o por tópicos (*topic exchange*). Estos enrutan mensajes a colas en función de una coincidencia entre la clave de enrutamiento del mensaje y el patrón de enrutamiento de la vinculación. Esta metodología es sencilla de manejar, donde los suscriptores especifican su interés en un tópico, y recibirán todos los mensajes publicados sobre este tema [7].

2.2 Plataforma de Integración

En primer lugar, se presenta una descripción general de Plataforma de Integración. Luego se describe las Plataformas de Integración como Servicio.

2.2.1 Descripción general

Las plataformas de integración (PI) son sistemas informáticos especializados que brindan mecanismos de conectividad y mediación confiables, para facilitar la integración de sistemas heterogéneos, en ambientes distribuidos. La comunicación es mediante mensajes que se intercambian a través de la plataforma [8].

Las PI brindan soporte para crear soluciones o flujos de mediación basados en los *Enterprise Integration Patterns* (EIP) [1]. Estos patrones, que cubren áreas como la transformación y enrutamiento de mensajes, proporcionan una forma independiente de tecnología para diseñar y documentar soluciones de integración.

Las tecnologías de middleware, como servicios web y middleware orientado a mensajería, comprenden servicios que hacen al funcionamiento de una PI. El modelo de arquitectura de software *Enterprise Service Bus* (ESB) [9] es un ejemplo de una PI monolítica.

La aparición de la computación en la nube (*cloud computing*), promueve el desarrollo de alternativas de integración basada en la nube, como la Plataforma de Integración como Servicio (iPaaS), que proporciona soluciones para diferentes requerimientos de integración en la nube [10].

2.2.2 Plataformas de Integración como Servicio (iPaaS)

Gartner define iPaaS como un conjunto de servicios en la nube, que permite desarrollar, ejecutar y administrar flujos de integración, para conectar procesos, aplicaciones, servicios o datos almacenados tanto en redes locales como en la nube, para una o muchas organizaciones [10].

Una iPaaS proporciona las capacidades para admitir, en una nube pública o privada, una variedad de escenarios de integración dentro de la misma organización, integración B2B (*Business to Business*) de comercio electrónico, y aspectos de administración (*governance*), como se observa en la Figura 2.4.

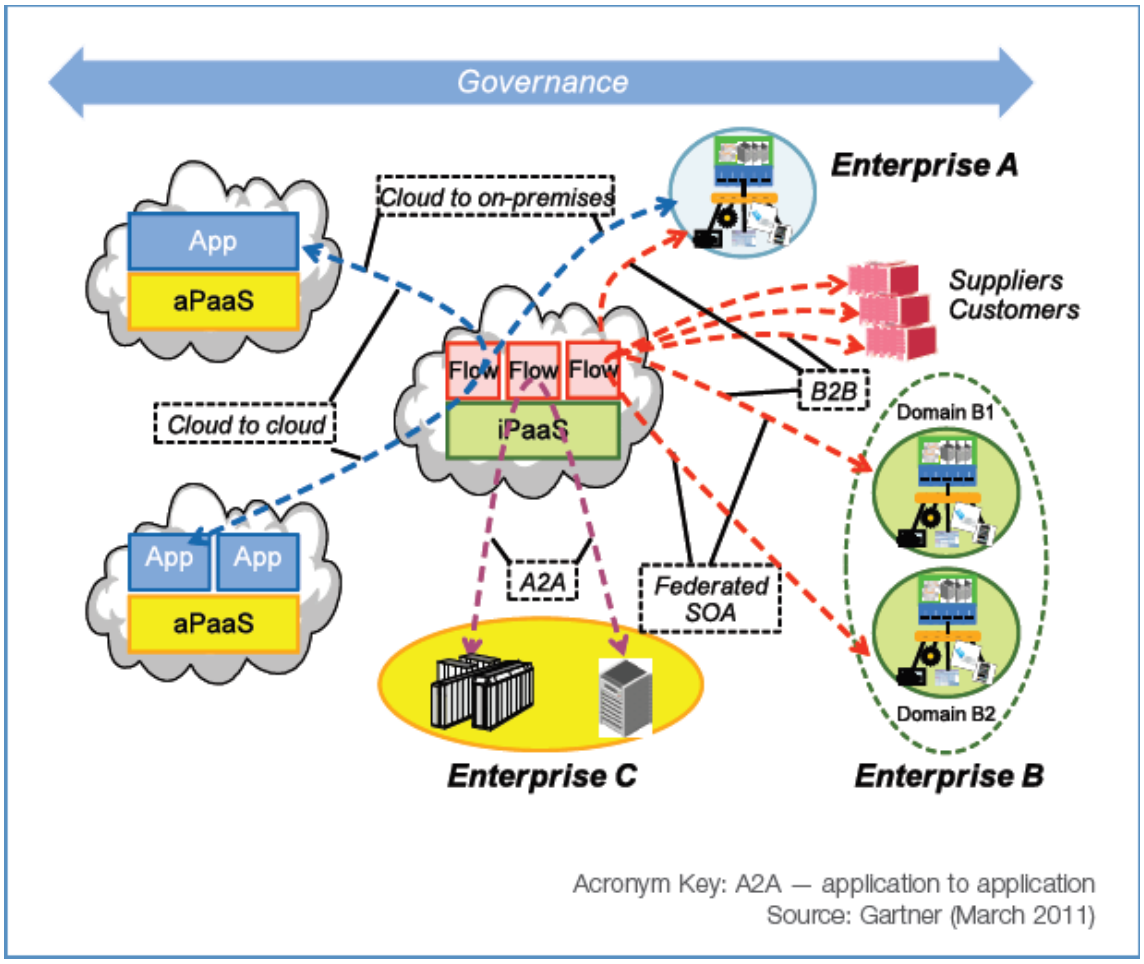

Figura 2.4: Escenario de integración de iPaaS [10]

2.3 Microservicios

En primer lugar, se presenta una descripción general de microservicios y se describen ciertas limitantes que la arquitectura de microservicios resuelve en comparación con sistemas monolíticos. Posteriormente se describe la relación entre microservicios y DevOps. A continuación, se presentan los dos modelos de ejecución para coordinar la comunicación de los servicios. Por último, se explican patrones comunes y buenas prácticas a utilizar en arquitecturas de microservicios.

2.3.1 Descripción general

Los microservicios son una forma de construir aplicaciones. Una arquitectura de microservicios busca independizar o desacoplar los componentes individuales de una aplicación, para que cada componente sea una aplicación en sí misma.

El término de microservicios carece de una definición común. Según J. Lewis y M. Fowler [11], el estilo arquitectónico de microservicios es un enfoque para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose mediante mecanismos ligeros. Agregan que estos servicios se construyen en torno a las funcionalidades de negocio y se despliegan de forma independiente mediante mecanismos de implementación automatizados. Manifiestan que los servicios pueden desarrollarse en distintos lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos, y que existe un mínimo de administración centralizada para los mismos.

S. Newman [12] menciona que los microservicios son pequeños servicios autónomos que trabajan juntos, mientras que A. Cockcroft[3] los define como una arquitectura orientada a servicios de bajo acoplamiento con contexto limitado. Ambos hacen énfasis en cierto nivel de independencia, alcance limitado e interoperabilidad.

Nadareishvili, Mitra, McLarty y Amundsen [13], especifican que un microservicio es un componente implementado de forma independiente, de alcance limitado, que admite la interoperabilidad a través de la comunicación basada en mensajes. También definen que la arquitectura de microservicios es un estilo de ingeniería de sistemas de software evolutivos y altamente automatizados.

Los microservicios pueden verse como una evolución del SOA (*Service Oriented Architecture*), cuya función es dividir un sistema complejo en una variedad de unidades independientes, cada una orientada a una función particular, con su propia lógica de negocio e independencia a nivel de desarrollo [11] [12].

Fowler y Lewis describen que muchos productos y enfoques, como por ejemplo los ESB, hacen hincapié en poner inteligencia significativa en el propio mecanismo de comunicación. Por otro lado, hacen referencia a que los microservicios promueven un enfoque alternativo: componentes inteligentes,

[3] Adrian Cockcroft, vicepresidente de *Cloud Architecture Strategy* en Amazon Web Services (AWS).

comunicaciones tontas. Esto implica que las aplicaciones creadas a partir de microservicios pretendan ser lo más desacopladas y lo más cohesivas posibles, ya que poseen su propia lógica de dominio. Este enfoque propone utilizar protocolos livianos de comunicación, como por ejemplo HTTP [11].

En la Figura 2.5 se muestra un diagrama comparando los enfoques descritos por Fowler y Lewis entre ESB y microservicios.

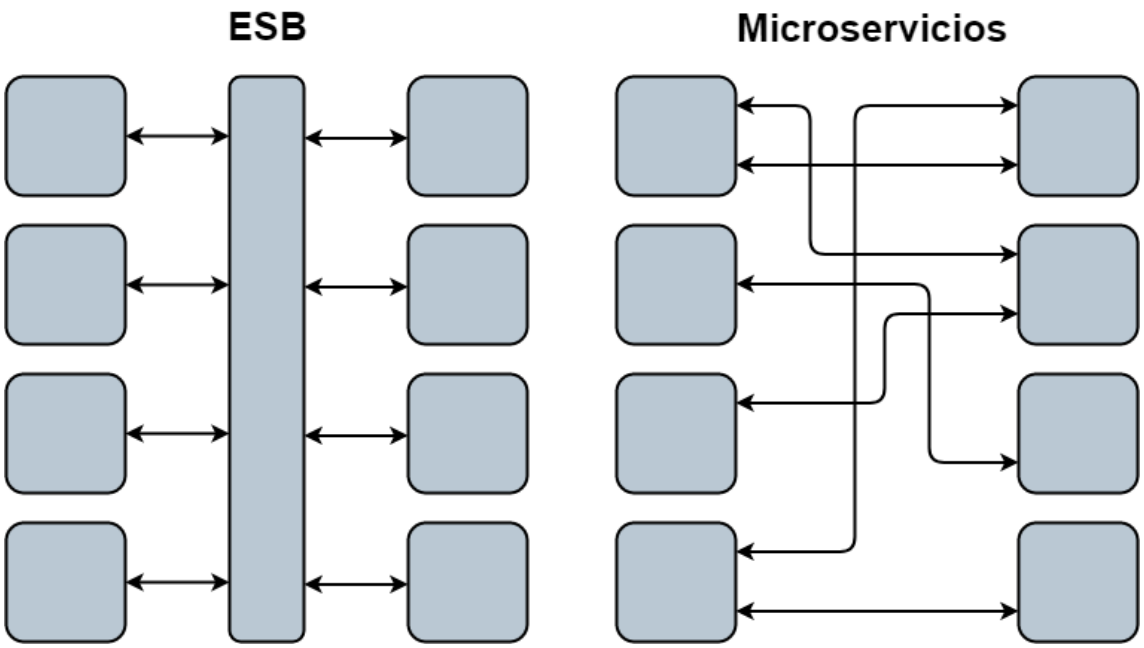

Figura 2.5: Componentes inteligentes y comunicaciones tontas vs ESB[4]

Los sistemas monolíticos concentran la funcionalidad y sus servicios en una base de código única. Las aplicaciones de software que utilizan este diseño tienen ventajas importantes frente a otras arquitecturas, a pesar de que se desarrollaron alternativas más elegantes: son fáciles de implementar y de desplegar, y resulta rápido y sencillo ejecutarlas. El desarrollo de aplicaciones monolíticas suele ser menos costoso debido a la simpleza de su estructura. Sin embargo, una aplicación que agrupe toda su operatividad no significa que sea mejor, especialmente si tiende a aumentar en usuarios, desarrolladores, complejidad y carga [11].

[4] Imagen modificada para ajustarla al caso de ESB vs Microservicios. La original puede encontrarse en https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f

Al tratarse de un único bloque de código, no resulta sencillo actualizar una aplicación monolítica, debido a que el impacto de los cambios necesita volver a desplegar la aplicación en su conjunto. Además, resulta difícil identificar y solucionar problemas específicos debido al tamaño del código. También puede ser agobiante para los desarrolladores, porque necesitan entender todo el código para poder trabajar en él correctamente.

Las aplicaciones monolíticas presentan un desafío de crecimiento, debido a que el aumento del código se vincula con una sobrecarga del sistema informático, lo que repercute en su agilidad.

En la Figura 2.6 puede apreciarse la comparación de la arquitectura de microservicios con la tradicional arquitectura monolítica.

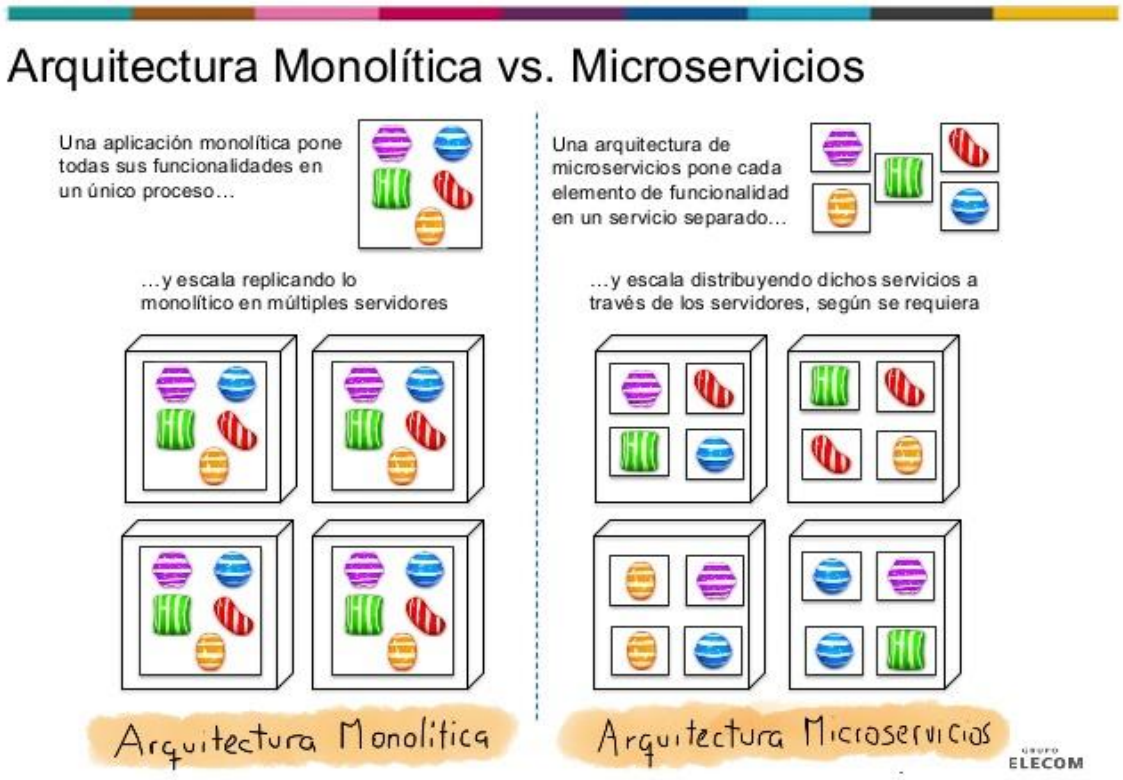

Figura 2.6: Comparación arquitectura monolítica y de microservicios[5] [11]

El gran diferenciador de los microservicios es que los distintos componentes del software pueden desarrollarse y desplegarse de forma independiente. Se refuerza el aislamiento de código, pues al tratarse de componentes separados,

[5] Imagen traducida al español. La original se encuentra en [11]

errores en un componente pueden reducir el impacto en la totalidad del código, contrariamente a las aplicaciones monolíticas [11] [12].

Newman menciona que la arquitectura de microservicios permite utilizar una tecnología diferente en cada componente, utilizando en cada caso la que mejor se ajuste a los requerimientos. Esta arquitectura, también permite la reutilización de servicios, y a su vez se pueden crear otros servicios con mayor complejidad a partir de los ya existentes [12].

Lewis y Fowler afirman que, si bien las aplicaciones monolíticas pueden ser exitosas, alguna de sus limitantes ha generado frustraciones, principalmente el aumento de despliegue de aplicaciones en la nube [11]. Describen también que, si aumenta el tamaño del sistema monolítico, aumenta la complejidad de este, lo cual realizar mantenimiento puede ser complejo. Con microservicios el impacto del mantenimiento es menor, ya que solo se actualiza y despliega los microservicios involucrados en el cambio [11].

Newman menciona como limitante en un sistema monolítico que todo deja de funcionar si el servicio falla. En cambio, con microservicios se pueden construir sistemas que manejen la falla total de los servicios, permitiendo al sistema seguir funcionando [12].

En sistemas monolíticos se debe escalar el sistema completo, incluso si la limitante son componentes puntales. Con microservicios el escalamiento es más eficiente ya que se realiza de forma selectiva [11] [12].

2.3.2 DevOps

Aunque los microservicios se destaquen por su flexibilidad, agilidad, eficiencia y potencial de crecimiento, implican retos importantes para su implementación. Al tener mayor número de componentes, su operación es más compleja, por lo que crear y desarrollar la infraestructura requiere más recursos y más tiempo. Esta es una razón para que la metodología DevOps[6] le ofrezca agilidad al proceso de desarrollo [13].

[6] *Development* (desarrollo) - *Operations* (operaciones)

DevOps es una práctica de ingeniería de software que tiene como propósito unificar el desarrollo y la operación del software. La principal característica del paradigma DevOps es defender enérgicamente la automatización y el monitoreo en todos los pasos de la construcción del software, desde la integración, las pruebas, la liberación, hasta la implementación y la administración de la infraestructura, como se puede ver en la Figura 2.7. DevOps apunta a ciclos de desarrollo más cortos, mayor frecuencia de implementación, lanzamientos más confiables y alineación con los objetivos comerciales [14].

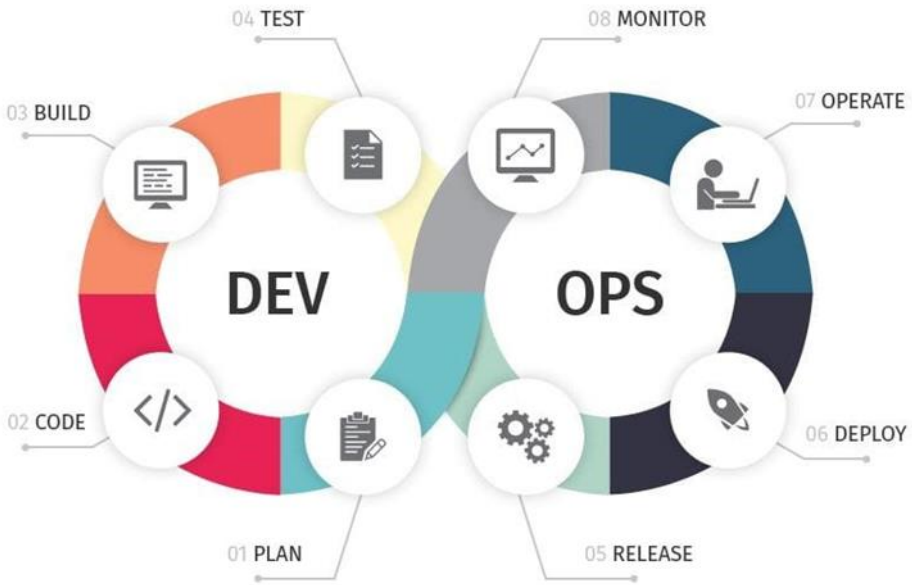

Figura 2.7: Fases del ciclo iterativo de DevOps[7]

[7] La imagen se obtuvo de https://dev.to/ashokisaac/devops-in-3-sentences-17c4

Existen diferentes retos en la operación de sistemas basados en microservicios [15]:

- Efectiva administración de recursos ya sea en *cloud* u *on-premise*.
- Esquemas de monitoreos para servicios distribuidos, incluyendo métricas de aplicación, redes, logs, así como formas de enlazar una operación o transacción de un cliente con los diferentes microservicios que utiliza.
- Prácticas de desarrollo ágil y equipos de desarrollo con enfoque productivo.

Estas capacidades implican una estrecha colaboración entre desarrolladores y operaciones, algo que promueve la cultura DevOps.

2.3.3 Modelo de ejecución

Cuando se diseña una arquitectura de microservicios, es necesario definir cómo se van a coordinar estos servicios. Existen dos estrategias para esto: orquestación y coreografía [16].

- **Orquestación**

Es la opción más clásica. Se basa en que un componente sea el encargado de coordinar las llamadas a los servicios que necesita de forma secuencial, como muestra la Figura 2.8, típicamente mediante llamadas de petición/respuesta. Este componente se encarga de gestionar los errores.

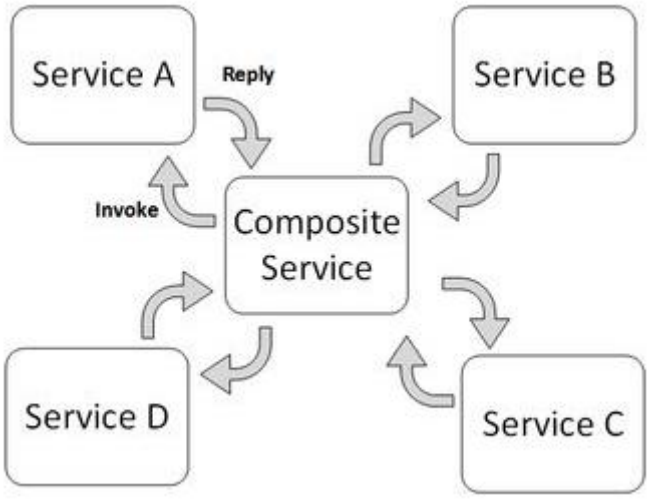

Figura 2.8: Orquestación de servicios[8]

- **Coreografía**

Es la opción que más encaja con la arquitectura de microservicios, pero también es la más compleja de gestionar. Los servicios no se llaman entre sí, sino que se utiliza un sistema de eventos, de forma que cuando un servicio termina su tarea, deja un mensaje y todos aquellos servicios suscritos a ese canal son notificados, de forma que puedan realizar su trabajo. La Figura 2.9 ilustra este concepto.

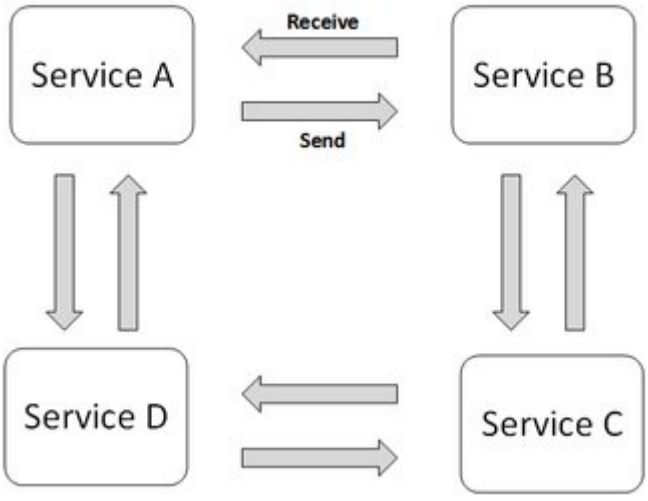

Figura 2.9: Corografía de servicios[9]

[8] La imagen se obtuvo de https://github.com/vaquarkhan/microservices-recipes-a-free-gitbook

[9] La imagen se obtuvo de https://github.com/vaquarkhan/microservices-recipes-a-free-gitbook

El modelo de orquestación tiene como ventaja que el componente encargado de orquestar conoce el proceso de negocio en su totalidad, y al encontrarse centralizada, es más sencillo modificar la lógica de negocio. La principal desventaja es que si el componente orquestador falla, fallará todo el proceso.

La ventaja principal de coreografía es su bajo nivel de acoplamiento. El mayor inconveniente es que la gestión de errores en este modelo es compleja. También se requieren sistemas específicos de monitorización para lograr seguir su flujo y sus dependencias [12].

2.3.4 Patrones

El desarrollo de sistemas basados en microservicios no está exento de utilizar patrones y buenas prácticas para resolver problemas comunes y aumentar la calidad del software. Chris Richardson propone un conjunto de patrones para microservicios [17] que permite abordar problemas conocidos con soluciones efectivas. En la Figura 2.10 se puede observar una categorización y los patrones definidos.

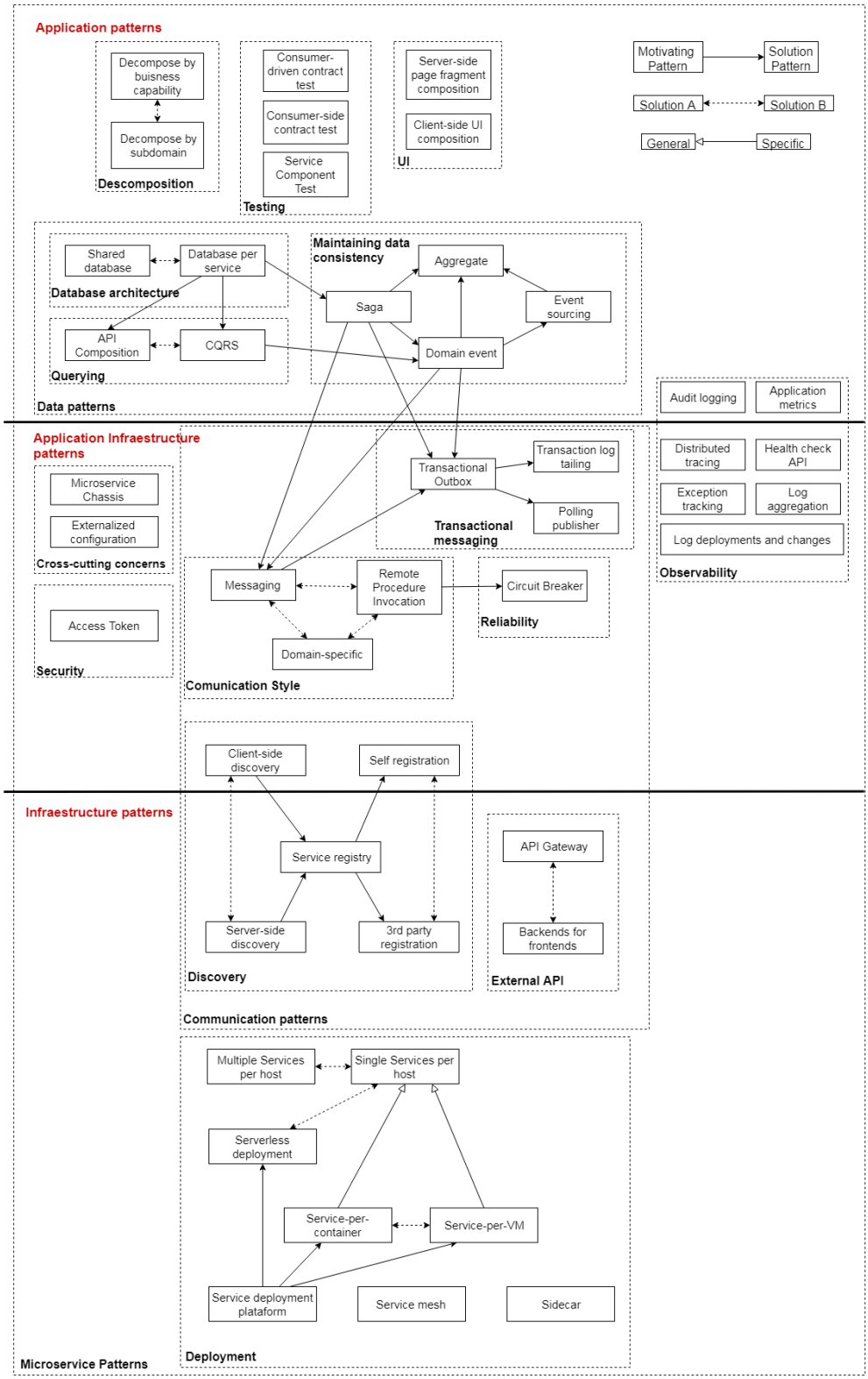

Figura 2.10: Patrones en la arquitectura de microservicios definidos por Chris Richardson[10] [17]

[10] Imagen modificada para una mejor visualización.

A continuación, se listan algunos de los patrones más relevantes definidos por Richardson:

- *Descomposición por capacidad del negocio*: indica que la definición de los microservicios se debe llevar a cabo por las capacidades del negocio.
- *Servicio de registro*: permite el registro y localización de los microservicios. Los servicios clientes consultan al registro para encontrar las instancias de los servicios que están activos.
- *Servicio de configuración externa*: es el encargado de almacenar toda la configuración relacionada a los microservicios y a todo el sistema.
- *Circuit Breaker*: permite monitorizar las comunicaciones entre servicios, previniendo consumir instancias con errores y evitando fallos en cascada.
- *Gestión de logs centralizado*: este servicio se encarga de acumular los logs de todos los servicios y permitir su análisis de forma centralizado.
- *Traza distribuida*: a cada petición externa se le asigna un identificador único. Ese identificador se envía a todos los servicios que están involucrados en la resolución de la petición. Cada entrada en los logs debe contener ese identificador.

En un artículo realizado por Alejandro Ladera para Deloitte [18] se definen algunos componentes que son comunes en las arquitecturas de microservicios:

- Registro
- Servidor Perimetral
- Balanceador
- Servidor de Configuración
- Sistema de tolerancia a fallos
- Gestión de Logs

El autor indica que los 3 primeros suelen considerarse el conjunto mínimo de patrones en una buena arquitectura de microservicios y que el resto aportan un gran valor a la solución.

2.4 Trabajos relacionados

A continuación, se describen productos del mercado que implementan plataformas de integración basadas en microservicios.

2.4.1 RoboMQ

RoboMQ[11] se presenta como una plataforma de integración híbrida que permite conectar cualquier dispositivo, sensor, aplicación Software como Servicio (*Software as a Service*: SaaS) o sistemas empresariales sobre cualquier protocolo de la industria. Afirma que para todos sus servicios de integración adoptaron la arquitectura de microservicios [19].

Se basa en la creación de flujos de integración, donde cada componente se mapea a un microservicio que se aloja en un contenedor [20].

En la Figura 2.11 se puede ver que RoboMQ se divide en 3 capas de componentes: el API Gateway, la infraestructura de mensajería y los microservicios.

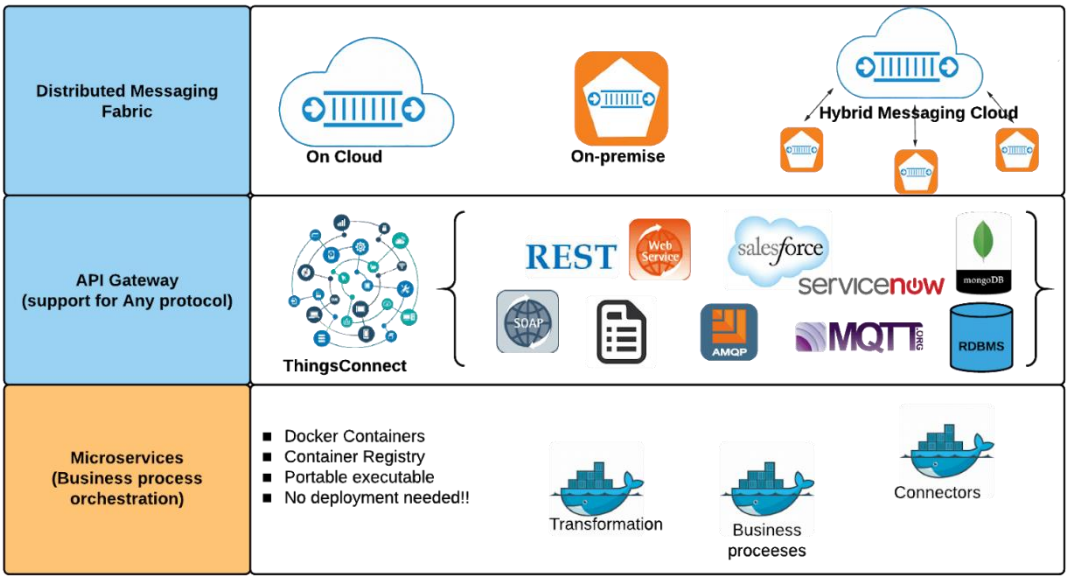

Figura 2.11: Plataforma RoboMQ [19]

[11] https://www.robomq.io

Es capaz de soportar múltiples broker de mensajería, como son RabbitMQ[12], Kafka[13], Kinesis[14], IBM MQ[15] y SQS[16].

RoboMQ destaca las siguientes características como las importantes de su plataforma [21]:

- *Centrado en la nube*: creado para la nube, internet de las cosas (IoT) y un entorno heterogéneo de SaaS, *Commercial Off-The-Shelf* (COTS) y aplicaciones empresariales.
- *Escalable*: la plataforma puede escalar automáticamente mediante escalamiento horizontal y balanceo de carga.
- *Extensible*: construida sobre la base de la fortaleza del middleware orientado a mensajes, se pueden agregar funciones y conjuntos de características sin afectar la aplicación existente.
- *Entrega confiable*: mediante colas de mensajes persistentes.
- *Monitoreo, cuadros de mando, análisis y alertas*: ofrece consolas para el monitoreo en tiempo real y análisis de la actividad en las colas de mensajes.
- *Soporta múltiples protocolos de comunicación*: soporta MQTT, AMQP, STOMP y HTTP / REST.

2.4.2 Elastic.io

Elastic.io[17] se define como una plataforma de integración iPaaS basada en microservicios, diseñada para conectar varias fuentes de datos en la nube y en redes internas de manera rápida, fácil y efectiva.

---

[12] https://www.rabbitmq.com/

[13] https://kafka.apache.org/

[14] https://aws.amazon.com/es/kinesis/

[15] https://www.ibm.com/support/knowledgecenter/es/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q001020_.htm

[16] https://aws.amazon.com/es/sqs/

[17] https://www.elastic.io/

Se basa en la creación de flujos de integración que obtienen la información de la fuente origen, la transforman según reglas definidas por el usuario y se envían a las fuentes destinos [22]. Cada paso del flujo se define como componente de integración y son los encargados de obtener/enviar o procesar/transformar los datos.

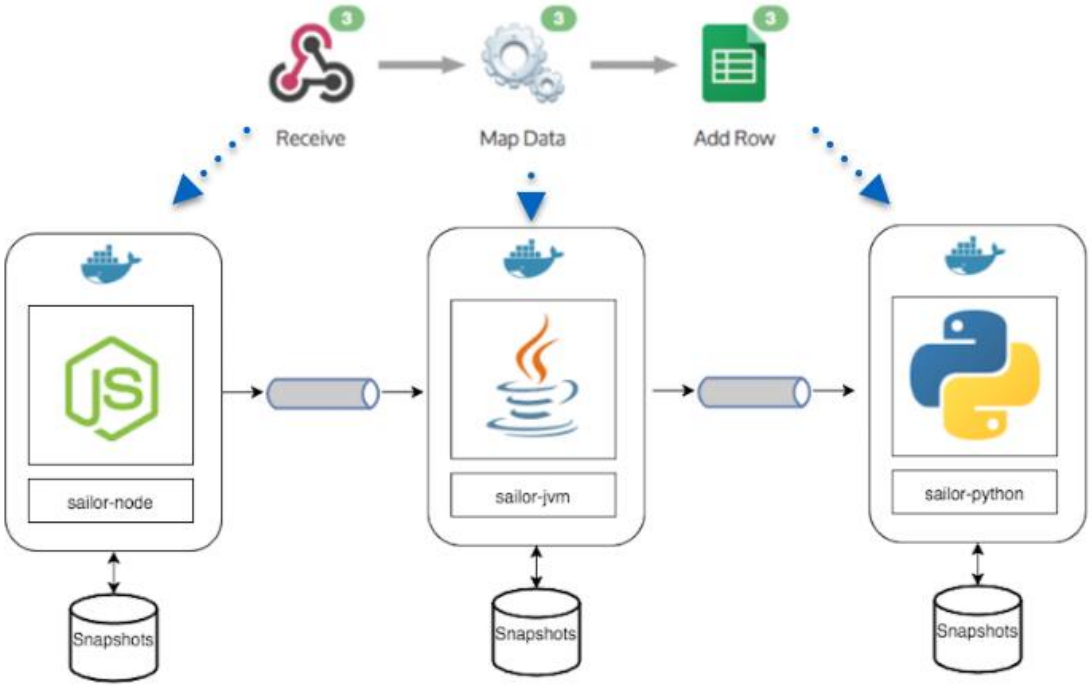

Figura 2.12: Flujo de integración desplegado en contenedores en Docker [23].

En la Figura 2.12 se puede observar un flujo de integración, donde cada componente es desplegado en un contenedor en Docker y se conectan por colas de mensajes.

2.5 Contenedores

Se realiza en primer lugar una descripción general sobre contenedores. Luego se describen beneficios de implementar microservicios con contenedores.

2.5.1 Descripción general

Los contenedores tratan de aislar a las aplicaciones y de generar un entorno estable y replicable para que funcionen. En lugar de alojar un sistema operativo completo, lo que hacen es compartir los recursos del propio sistema operativo *host* sobre el que se ejecutan las aplicaciones [24].

Esta tecnología cuenta con un gestor de contenedores que se encarga de lanzar y administrar a los mismos, pero en lugar de exponer los diferentes recursos de *hardware* de la máquina de manera discreta, lo que hace es compartirlos entre todos los contenedores, optimizando su uso y eliminando la necesidad de tener sistemas operativos separados para conseguir el aislamiento. Este gestor funciona a partir de imágenes que se pueden reutilizar entre varias aplicaciones. Cada una de estas imágenes puede ser incorporada a una "capa", donde se combinan o superponen entre sí para construir un sistema de archivos, formando la base del contenedor. Las capas contienen binarios, bibliotecas o *runtimes* necesarios para ejecutar la aplicación desplegada [25].

Cuando se lanzan uno o varios contenedores a partir de una imagen, a efectos de la aplicación es como si estuviera ejecutándose en su propio sistema operativo, aislado de cualquier otra aplicación que hubiese en la máquina en ese instante. Pero la realidad es que están compartiendo el sistema operativo *host* que hay por debajo. Es decir, los contenedores aíslan aplicaciones, no sistemas operativos completos [26]. Docker[18] es la tecnología de contenedores más utilizada en los últimos tiempos. Existen otras soluciones como Virtuozzo[19], OpenVZ[20] y LXC[21].

En la Figura 2.13 se muestra una comparación de alto nivel entre contenedores y Máquinas Virtuales (VMs).

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

[19] https://www.virtuozzo.com/

[20] https://openvz.org/

[21] https://linuxcontainers.org/

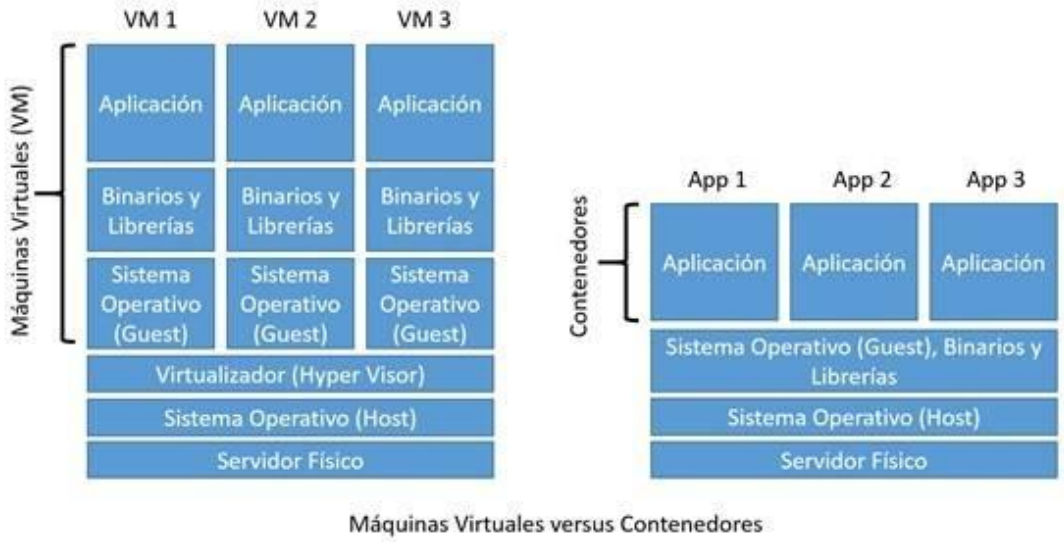

Figura 2.13: Comparación de VM y contenedores[22]

En resumen, los contenedores permiten desplegar aplicaciones más rápido y aprovechar mejor los recursos de *hardware* en comparación con las VMs.

2.5.2 Microservicios con contenedores

La tecnología de contenedores es un muy buen entorno para implementar microservicios. Los contenedores permiten un despliegue más eficiente de estos, pudiendo implementarse en una VM o en entornos nativos [27].

La portabilidad es otro beneficio de la implementación de servicios en contenedores. Cuando una empresa ofrece un gestor de contenedores como un servicio, puede implementar los mismos contenedores que se ejecutan en la nube, lo que hace que la escalabilidad sea más dinámica sin la sobrecarga de levantar VMs y desplegarle aplicaciones [27].

Uno de los beneficios de implementar microservicios es la flexibilidad de elegir el mejor lenguaje de programación y *stack* tecnológico para cada servicio. Si bien esto es bueno para el desarrollo, puede generar dolores de cabeza al momento de desplegar las diferentes aplicaciones. Al empaquetar microservicios en contenedores, el equipo de operaciones solo necesita saber

[22] La imagen se obtuvo de https://www.fayerwayer.com/2016/06/maquinas-virtuales-vs-contenedores-que-son-y-como-elegir-entre-estas-tecnologias/

cómo desplegar los contenedores, sin saber nada de las diferentes aplicaciones que se ejecutan dentro de ellos. Se puede ver a los contenedores como un puente entre desarrolladores (*dev*) y operadores (*ops*) [28].

También se evitan posibles fallas en el servicio debido a la falta de dependencia o versiones que no coinciden en el sistema operativo *host*, ya que el código, el *framework* y todo lo que el servicio necesita está empaquetado en un entorno inmutable. Los contenedores simplifican aún más las operaciones al proteger a los desarrolladores de la necesidad de preocuparse por los detalles, tanto de la máquina, como del sistema operativo [28].

La orquestación de contenedores ayuda a administrar todas las complejidades asociadas con las aplicaciones distribuidas basadas en microservicios. Debido a que están distribuidas, las aplicaciones se ejecutan en un clúster (serie de computadoras en red que se ven como un solo sistema). Si bien la programación de nuevos servicios en un clúster es bastante simple, los orquestadores son importantes para comprender todos los requisitos únicos de sus servicios, sus dependencias y los recursos disponibles del sistema. Un buen orquestador de contenedores puede mantener la aplicación activa si falla un servicio, mover servicios de nodos fallidos, y habilitar actualizaciones continuas para que estos estén "siempre encendidos" [28].

Los orquestadores más populares son Kubernetes [23], DC/OS [24] y Docker Swarm[25].

[23] https://kubernetes.io/

[24] https://dcos.io/

[25] https://docs.docker.com/engine/swarm/

3 Análisis

En este capítulo se realiza un análisis de la problemática planteada para este proyecto. Esta se basa en diseñar e implementar una Plataforma de Integración en la nube basada en microservicios, tomando como referencia la arquitectura presentada en la tesis de Andrés Nebel [2].

En la Sección 3.1 se presenta el contexto del trabajo con relación a la arquitectura de referencia. Los requerimientos iniciales y escenarios por cubrir se especifican en la Sección 3.2, mientras que en la Sección 3.3 se presenta el refinamiento de los requerimientos y alcance final del trabajo. Por último, en la Sección 3.4 se presentan las conclusiones del capítulo.

3.1 Contexto de trabajo

La realidad propuesta se enmarca en brindar una implementación para la Plataforma de Integración definida en la tesis de Andrés Nebel [2].

La PI debe soportar la creación y administración de Soluciones de Integración (SI). Estas son creadas en base a componentes de integración, que ofrecen capacidades de conectividad, mediación y mensajería. En la Figura 3.1 se ilustra un esquema de estos conceptos [2].

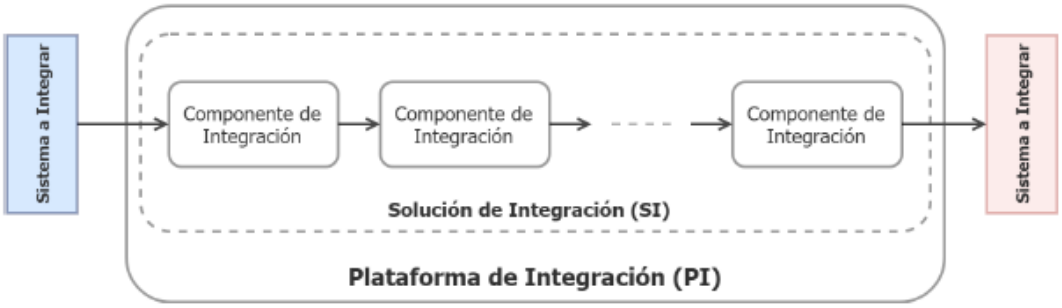

Figura 3.1: Esquema de los conceptos de PI, SI y componentes de integración[26] [2]

[26] Los sistemas para integrar pueden ser varios, se muestran solo dos para mejorar la legibilidad del esquema. Análogo para las SI, pueden desplegarse múltiples soluciones en la misma PI.

Dicha PI es de propósito general y basada en microservicios, apoyándose en los estilos de coreografía y orquestación para la coordinación de los componentes de integración presentes en las diferentes Soluciones de Integración.

3.1.1 Solución de Integración

Se define Solución de Integración (SI) a la serie de pasos que se ejecutan con el objetivo de integrar dos o más sistemas. Estas soluciones son desplegadas, ejecutadas y gestionadas en la PI. Cada paso de una SI es un componente de integración configurable, que realiza una operación de integración específica (por ejemplo, transformación de datos).

La ejecución de una SI comienza en componentes de integración llamados disparadores, los cuales reaccionan a eventos, y finaliza cuando se ejecuta el último componente de la SI. Cada componente de integración ejecuta su acción e invoca al siguiente componente transmitiéndole información. Cada evento que activa un componente disparador ejecuta una nueva instancia independiente de la SI [2].

3.1.2 Componente de Integración

La PI proporciona un conjunto de componentes de integración configurables para implementar SI. Un ejemplo es el componente “transformación”, el cual se configura especificando la transformación concreta a realizar. Los componentes pueden ser de dos tipos: disparador o acción. Los disparadores desencadenan la ejecución de la SI, mientras que los de acción efectúan una tarea de integración puntual.

Los componentes de integración son predefinidos en la PI. Estos componentes pueden ser: conectores, transformadores, ruteo, enriquecedores de contenido, mensajería, entre otros [2].

3.1.3 Ejemplo conceptual

En el siguiente ejemplo se muestran los conceptos definidos en las secciones anteriores. Se requiere integrar tres sistemas desplegados en la nube: Sistema C1, C2 y C3, y comunicarse con un servicio de correo electrónico externo como Gmail. Para atacar esta problemática se utiliza una PI desplegada en la nube y se construye la SI presentada en la Figura 3.2.

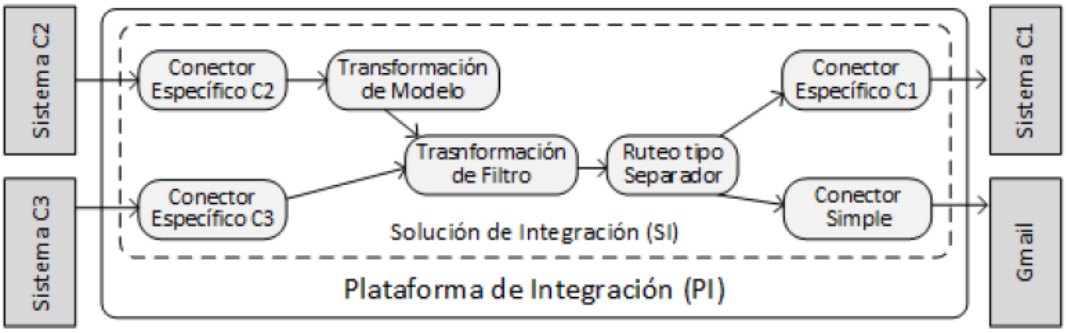

Figura 3.2: Ejemplo de los conceptos de PI, SI y componentes de integración [2]

Los sistemas C2 y C3 envían datos al sistema C1. Los tres sistemas soportan protocolos de comunicación diferentes. Además, C2 maneja un modelo de datos diferente a C1 y C3. Se requiere que los datos que cumplen determinadas condiciones se envíen a C1, notificando por correo electrónico a una casilla de Gmail.

La SI se crea con los siguientes componentes de integración: *Conector Específico*, *Conector Simple*, *Transformación* y *Ruteo*. Debido a que los sistemas utilizan protocoles de comunicación que no son estándar, se usan conectores específicos para obtener datos de los sistemas C2 y C3, y para enviar datos a C1. Los datos enviados por C2 son procesados con un componente de transformación de modelo y los datos que no cumplen las condiciones son descartadas por un componente de transformación de filtro. El componente ruteo de tipo separador envía los datos recibidos a C1 y a una casilla de Gmail, utilizando los conectores correspondientes [2].

3.1.4 Visión global

En la Figura 3.3 se sintetizan los conceptos generales relacionados a la PI y su contexto.

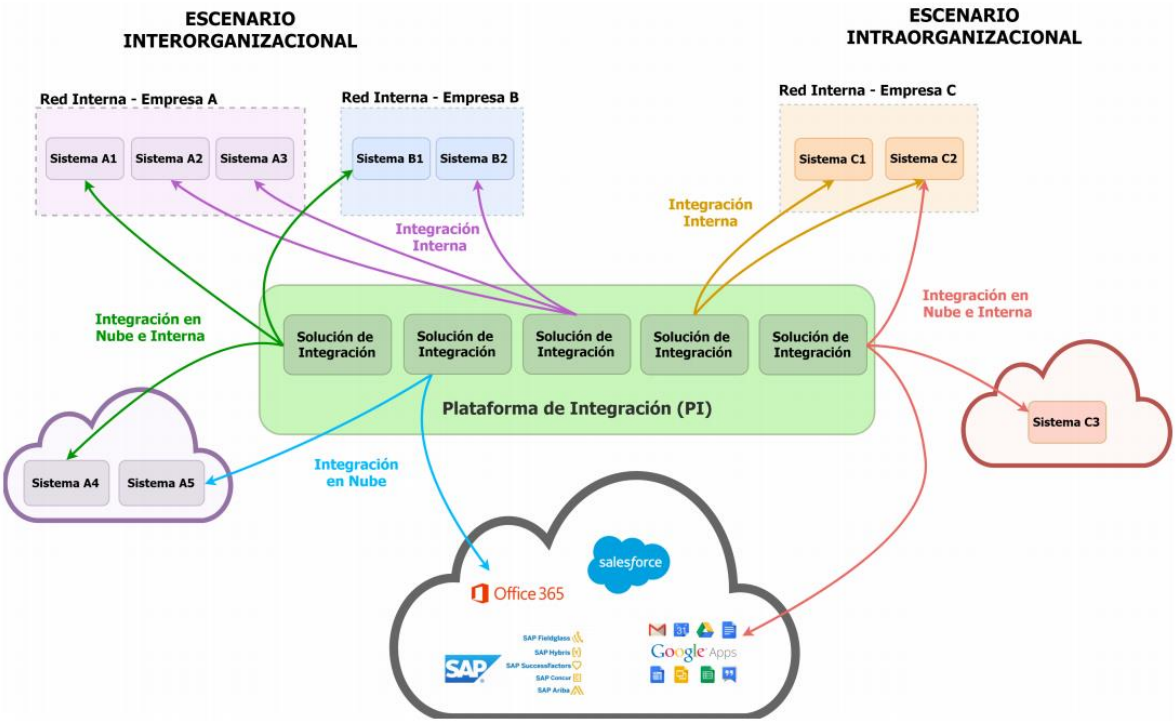

Figura 3.3: Conceptos generales relacionados a PI y su contexto [2]

Se observa que mediante soluciones de integración se pueden integrar múltiples sistemas heterogéneos, los cuales pueden estar distribuidos en reden internas (integración interna) y en la nube (integración en la nube) [2]. Los sistemas para integrar pueden ser de múltiples empresas (escenario interorganizacional) o de una única empresa (escenario intraorganizacional), pueden ser aplicaciones del tipo SaaS o productos de terceros (Salesforce CRM).

De este contexto se desprenden algunos casos de usos que son relevantes para las Plataformas de Integración (Tabla 3.1):

ID	Caso de Uso	
CU_1	Gestión de SI	Crear SI
CU_2		Implementar SI
CU_3	Administrar SI	Desplegar SI
CU_4		Supervisar SI
CU_5		Escalar SI
CU_6	Ejecutar SI	Activar flujo
CU_7		Ejecutar siguiente acción

Tabla 3.1: Casos de Uso basados en arquitectura de referencia [2]

Los casos de uso de Gestión de SI, que incluye Crear e Implementar SI, consisten en agregar y configurar componentes de integración previamente definidos.

El caso de Desplegar una SI implica instanciar los componentes utilizados en la SI y ponerlos en marcha. El caso de Supervisar SI contempla la gestión centralizada y análisis de logs de los componentes de integración. El caso de Escalar la SI, implica generar los mecanismos adecuados para que la plataforma escale, ya sea a nivel de SI o de los componentes de integración. Esta tarea podrá ser de forma automática (si se detecta que ciertos indicadores de rendimiento sobrepasan los umbrales críticos) o de forma manual por algún administrador de la PI.

La PI soportará el caso de uso de Ejecutar una SI. Este comienza cuando un disparador externo activa la SI, la cual desencadena invocaciones a los siguientes componentes de la SI.

El Apéndice B detalla la arquitectura de referencia presentada en la tesis de Andrés Nebel.

3.2 Requerimientos iniciales

Los requerimientos fueron definidos en base a cuatro escenarios que la PI debe soportar, que se describen a continuación.

Firma WS-Security[27]

Un cliente necesita comunicarse con un *Web Service* mediante el protocolo WSS (*Web Service Security*), para ello se comunica con la plataforma que es la encargada de firmar el mensaje. En la Figura 3.4 se muestra un diagrama del flujo de comunicación.

[27] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

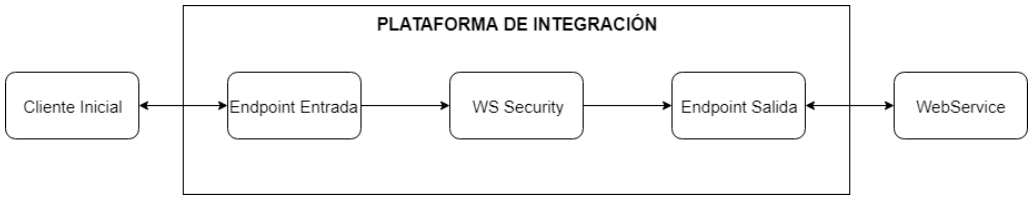

Figura 3.4: Escenario firma WS-Security

Transformación

Un cliente invoca a la plataforma para modificar el mensaje de acuerdo con las especificaciones de un cliente final. De esta forma dos aplicaciones que manejan distintos protocolos de mensajes se pueden comunicar. En la Figura 3.5 se muestra un ejemplo donde un cliente inicial se comunica mediante mensajes JSON y los mismos son transformados a XML.

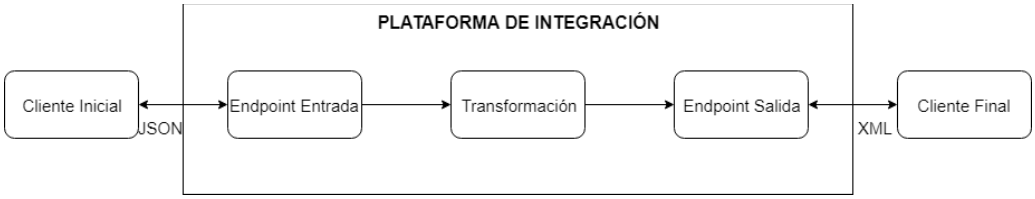

Figura 3.5: Escenario transformación

Notificación

Un cliente invoca a la plataforma y todos los mensajes se reenvían a una lista de suscriptores de este. En la Figura 3.6 se muestra un diagrama con un posible flujo de los mensajes dentro de la plataforma.

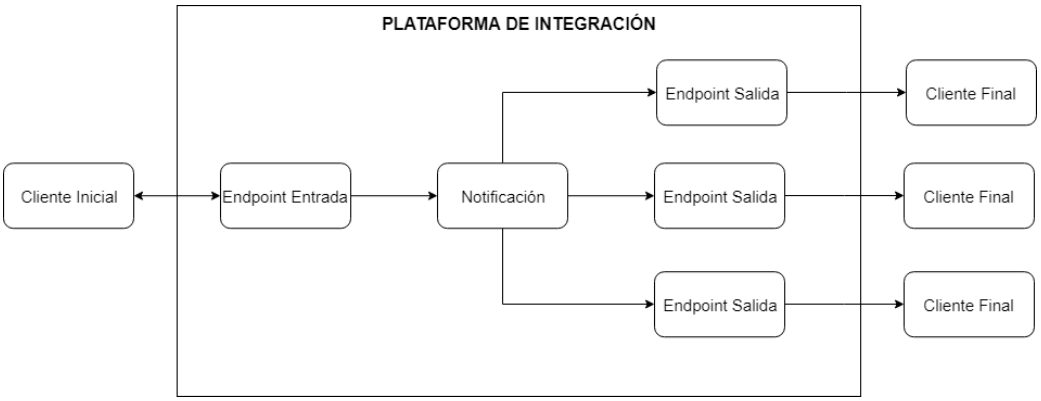

Figura 3.6: Escenario notificación

Ruteo

Los mensajes de un cliente son encaminados a distintos destinatarios de acuerdo con el contenido del mensaje. En la Figura 3.7 se observa cómo un

mensaje se transmite a un cliente final o una cola de mensajes según el contenido del mensaje inicial.

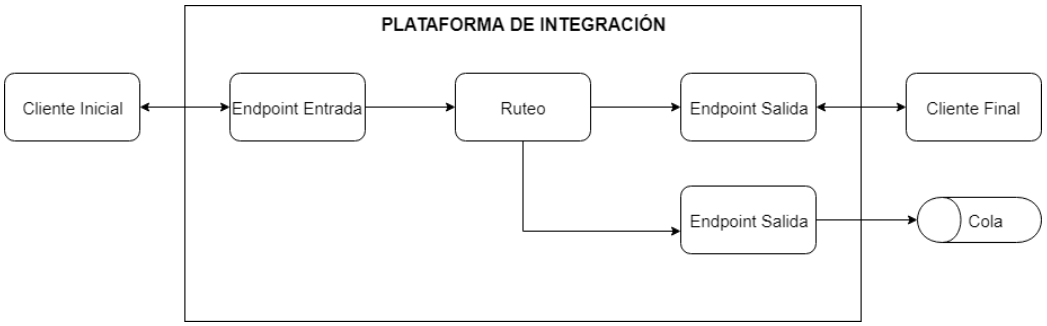

Figura 3.7: Escenario ruteo

A continuación, en la Tabla 3.2 se presentan los requerimientos funcionales relevados inicialmente.

ID	Título	Descripción
REQI_1	Componentes de Integración	Se debe permitir la creación de componentes de integración por parte de los desarrolladores, los que quedarán disponibles para ser utilizados en SI.
REQI_2	Solución de Integración	Se permite agregar nuevas SI a la plataforma, las mismas son agregadas por los desarrolladores.
REQI_3	Componente Transformación	Se debe construir el componente de integración Transformación.
REQI_4	Componente Ruteo	Se debe construir el componente de integración Ruteo.
REQI_5	Escenario Transformación	La plataforma debe soportar el escenario de Transformación.
REQI_6	Escenario WS-Securtiy	La plataforma debe soportar el escenario WS-Security.
REQI_7	Escenario Notificación	La plataforma debe soportar el escenario de Notificación.
REQI_8	Escenario Ruteo	La plataforma debe soportar el escenario de Ruteo.
REQI_9	Estilos de ejecución	Las SI deben soportar los estilos de ejecución coreografía y orquestación.
REQI_10	Autenticación y autorización	Se desea que los clientes que accedan a la PI deban autenticarse y ser autorizados a ejecutar una SI.

Tabla 3.2: Requerimientos funcionales iniciales

En la Tabla 3.3, se presentan los requerimientos no funcionales.

ID	Título	Descripción
REQI_11	Microservicios	Se debe utilizar la arquitectura de microservicios.
REQI_12	Desplegado en la nube	El sistema debe poder ser desplegado en la nube.
REQI_13	Monitoreo	Se requiere contemplar características adicionales que ayuden a monitorizar y gestionar la aplicación desplegada. El usuario administrador debe poder acceder a un monitor donde visualice los logs del sistema.
REQI_14	Escalabilidad y Balanceo de Carga	El sistema debe escalar vertical y horizontalmente. Se requiere que el sistema implemente técnicas de balanceo de carga al momento de invocar componentes del sistema.
REQI_15	Tolerancia a fallos	Se requiere que el sistema pueda detectar y recuperarse rápidamente a los fallos que pueden surgir durante la ejecución.

Tabla 3.3: Requerimientos no funcionales iniciales

3.3 Refinamiento y alcance final

Luego de la baja de un integrante del equipo, en conjunto con los tutores y debido a los plazos del proyecto, se realiza un refinamiento de los requerimientos iniciales presentados en la sección anterior.

Sobre los escenarios presentados en la Sección 3.2, se opta por cubrir el escenario de *Transformación* y *Enriquecedor* con autenticación *HTTP-Basic*[28], dejando de lado *Notificación, WS-Security* y *Ruteo*.

Los requerimientos funcionales finales para implementar en relación con los definidos originalmente se presentan en la Tabla 3.4.

[28] https://www.ietf.org/rfc/rfc2617.txt

ID	ID Inicial	Título	Descripción
REQF_1	REQI_1	Componentes de Integración	Se debe permitir la creación de componentes de integración por parte de los desarrolladores, los que quedarán disponibles para ser utilizados en SI.
REQF_2	REQI_2	Solución de Integración	Se permite agregar nuevas SI a la plataforma, las mismas son agregadas por los desarrolladores.
REQF_3	REQI_3	Componente Transformación	Se debe construir el componente de integración Transformación.
REQF_4		Componente Enriquecedor	Se debe construir el componente de integración Enriquecedor.
REQF_5	REQI_5	Escenario Transformación	La plataforma debe soportar el escenario de Transformación.
REQF_6		Escenario Enriquecedor	La plataforma debe soportar el escenario de Enriquecimiento con autenticación HTTP Basic.
REQF_7	REQI_9	Estilo de ejecución	Las SI deben soportar los estilos de ejecución coreografía y orquestación.

Tabla 3.4: Requerimientos funcionales finales

En la Tabla 3.5, se presentan los requerimientos no funcionales a implementar:

ID	ID Inicial	Título	Descripción
REQF_8	REQI_11	Microservicios	Se debe utilizar la arquitectura de microservicios.
REQF_9	REQI_12	Desplegado en la nube	El sistema debe poder ser desplegado en la nube.
REQF_10	REQI_13	Monitoreo	Se requiere contemplar características adicionales que ayuden a monitorizar y gestionar la aplicación desplegada. El usuario administrador debe poder acceder a un monitor donde visualice los logs del sistema.
REQF_11	REQI_14	Escalabilidad y Balanceo de Carga	El sistema debe escalar vertical y horizontalmente. Se requiere que el sistema implemente técnicas de balanceo de carga al momento de invocar componentes del sistema.

Tabla 3.5: Requerimientos no funcionales finales

En esta etapa se agrega un nuevo escenario que luego sería utilizado en la definición del caso de estudio:

Enriquecedor

Un cliente invoca la plataforma para modificar el mensaje de acuerdo con las especificaciones de un cliente final. En la Figura 3.8 se observa un diagrama donde el enriquecedor es el encargado de agregar la información necesaria para realizar una autenticación HTTP Basic contra el sistema final.

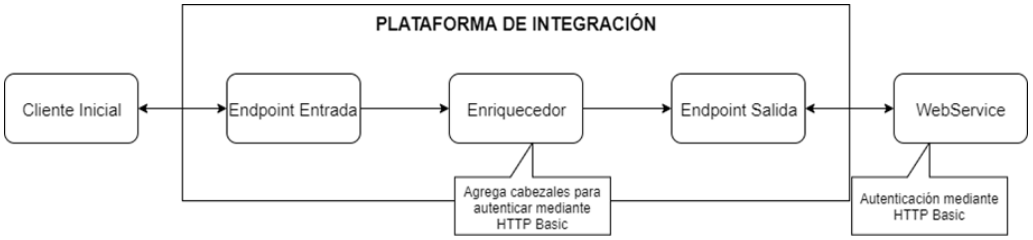

Figura 3.8: Escenario enriquecedor

En la definición de los requerimientos finales se procuró cubrir la mayor cantidad de casos de usos. En la Tabla 3.6 se muestra la relación entre los requerimientos y los casos de usos definidos en la Visión Global (Tabla 3.1):

ID Caso de Uso	Caso de Uso	Soporta PI a construir	Requerimiento
CU_1	Crear SI	Se permite el agregado de nuevas SI por parte de los desarrolladores	REQF_1 REQF_2 REQF_3 REQF_4 REQF_7 REQF_8
CU_2	Implementar SI	Si	REQF_1 REQF_2 REQF_3 REQF_4 REQF_7 REQF_8
CU_3	Desplegar SI	Si	REQF_9
CU_4	Supervisar SI	Gestión y análisis de logs	REQF_10
CU_5	Escalar SI	No automáticamente. Si manualmente.	REQF_12
CU_6	Activar flujo	Si	REQF_2 REQF_3 REQF_4

			REQF_7
CU_7	Ejecutar siguiente acción	Si	REQF_2 REQF_3 REQF_4 REQF_7

Tabla 3.6: Relación entre requerimientos y casos de usos

Basándose en los diagramas de Vista Lógica presentados en la arquitectura de referencia (ver Figura B. 2), en la Tabla 3.7 se determinan las entidades a implementar en la PI a desarrollar, que soportan los casos de uso definidos anteriormente [2]:

Subsistemas	**Entidades**	**Soporta PI a construir**
Componentes de Integración	Conectores Específicos	No
	Conectores Simples	Si
	Mensajería	Si
	Ruteo	No
	Transformación	Si
	Componentes Personalizados	No
Gestión de SI	Repositorio de SI	Si
	Despliegue de SI	Si
	Control y supervisión de SI	No
Utilitarios de Servicios	Seguridad	No
	Virtualización de Servicios	No
	Gestor de Logs	Si
	Gestor de Cache	No
	Descubrimiento de Servicios	Si
	Balanceador de Carga	Si
	Circuit Breaker	No

Tabla 3.7: Entidades que soportará la PI

Dentro de los Componentes de Integración, la plataforma soportará Conectores Simples, Mensajería y Transformación. Se implementarán dos tipos de

Transformadores: uno personalizado y otro basado en enriquecedor de contenido.

La PI que se desea construir brindará los servicios de Gestor de Logs, Descubrimiento de Servicios y Balanceador de Carga.

3.4 Conclusiones

La propuesta de este proyecto se enfoca en construir una PI basada en microservicios de propósito general [2]. Si bien las propuestas analizadas soportan varios protocolos de comunicación con sus plataformas, esta propuesta permite incorporar conectores específicos con el fin de comunicarse con cualquier sistema.

La incorporación de microservicios a la PI de esta propuesta agrega la posibilidad de realizar los dos estilos de comunicación más comunes en este tipo de arquitecturas (orquestación y coreografía), según sea conveniente en cada solución a crear.

4 Solución Propuesta

El objetivo de este capítulo es presentar el diseño de la solución propuesta, tomando como base la arquitectura de referencia [2].

En primer lugar, se presenta en la Sección 4.1 una descripción general. Luego en la Sección 4.2 se muestra el diseño de ejecución entre los microservicios. El diseño de los componentes de integración se especifica en la Sección 4.3 y en la Sección 4.4 se detallan los patrones de diseño utilizados. Por último, en la Sección 4.5 se presenta una discusión respecto a la arquitectura de referencia.

4.1 Descripción general

La descripción general de la solución se basa en el diagrama de componentes que se presenta en la Figura 4.1. Para ilustrar como se compone una SI y como es su flujo, se presenta en la figura un ejemplo.

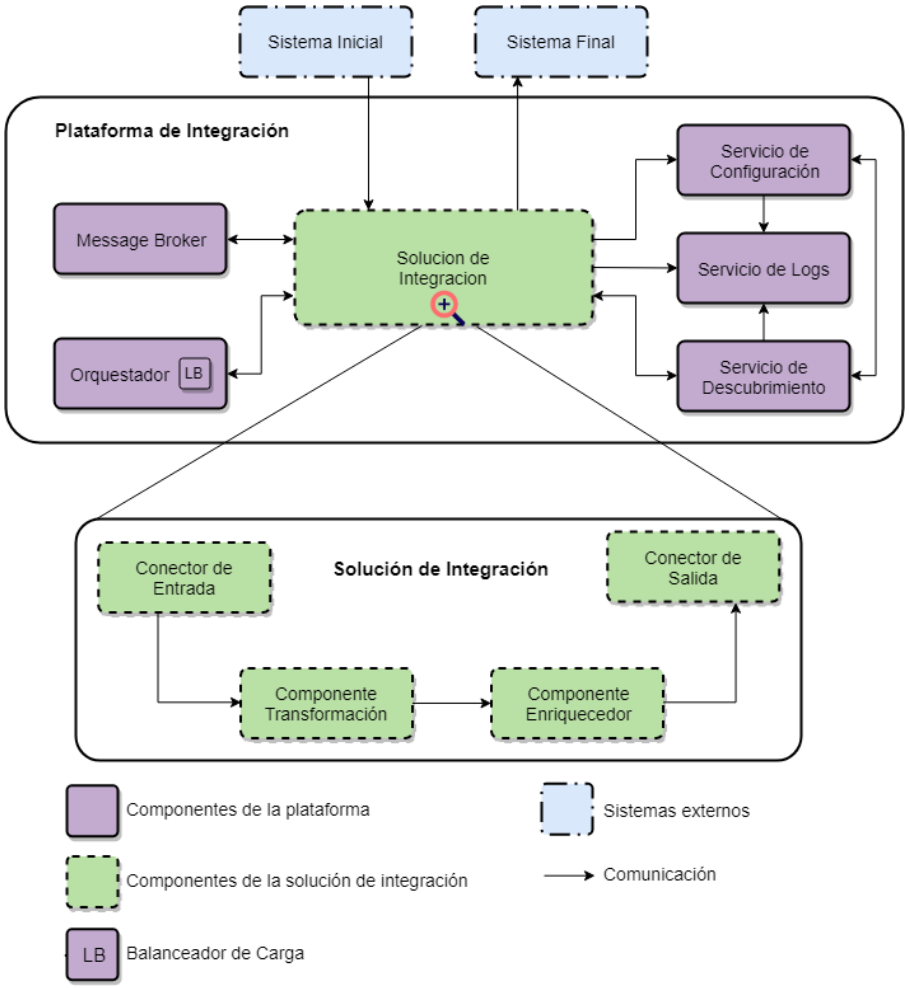

Figura 4.1: Diagrama de componentes de la arquitectura

En la imagen se identifican tres clases de componentes: en celeste (cuadro con borde de puntos y líneas) los sistemas externos con los que la plataforma mantiene algún tipo de comunicación, en violeta (cuadro con borde sólido) los componentes de la plataforma que brindan servicios a los componentes de todas las soluciones de integración y en verde (cuadro con borde punteado) los componentes específicos de una solución de integración.

La comunicación siempre es iniciada por algún sistema externo que desea consumir alguna de las soluciones de integración expuesta por la plataforma. Estos sistemas externos solo pueden comunicarse con los componentes disparadores (conectores), y ellos son los encargados de iniciar el flujo de integración dentro de la plataforma. Los conectores pueden comunicarse con otros componentes de la solución, servicios de la plataforma u otros sistemas externos de los que necesiten obtener información.

4.1.1 Componentes y sus principales características

Se detallan los componentes y sus principales características presentados en la Figura 4.1:

Aplicaciones cliente

Las *Aplicaciones cliente* desarrolladas por terceros tienen como propósito ejecutar o brindar servicios que consuman Soluciones de Integración publicadas por la Plataforma de Integración. Se comunican con esta a través de los conectores o componentes de integración disparadores de la SI.

Componentes de integración

Los *Componentes de integración* coordinados forman parte de una solución de integración dentro de la PI. Existen los componentes conectores o disparadores, que son los encargados de activar una SI. Estos invocan al próximo componente de integración para continuar con la ejecución de la Solución de Integración. Los componentes disparadores son los Conectores de Entrada y Salida de la PI.

También se encuentran los componentes de acción, cuyo propósito consiste en efectuar una tarea de integración puntual. Estos componentes de integración ejecutan su acción e invocan al siguiente componente trasmitiéndole información mediante intercambio de mensajes. Los componentes de acciones en la PI son los Transformadores.

Se resuelve hacer corresponder directamente los componentes de integración de la PI como microservicios, como se sugiere en arquitectura de referencia [2].

Servicio de Descubrimiento

El *Servicio de Descubrimiento* o Registro sirve para registrar y localizar componentes de integración dentro de la PI. Este servicio mantiene una lista de los componentes activos dentro de la plataforma. Un componente cuando es desplegado en la PI, lo primero que hace es registrarse. Cuando un componente necesite enviar un mensaje a otro, consultará al Registro la lista de instancias existentes para ese componente concreto. El Servicio de Descubrimiento aporta abstracción a la localización física de los componentes de integración (microservicios), ya que los componentes que desean comunicarse con determinado microservicio solo deben conocer su identificador.

Servicio de Configuración Externo

El *Servicio de Configuración Externo* sirve para obtener la configuración de cada componente de integración en sistemas distribuidos. Su función es almacenar las propiedades de configuración de los microservicios de la PI. Al arrancar, un componente consulta al servidor de configuración las propiedades asociadas y la descarga del repositorio. El Servicio de Configuración Externo brinda un repositorio centralizado para las configuraciones de los componentes, además de un histórico del mismo.

Servicio de Logs

El *Servicio de Logs* concentra los logs que genere cada componente de integración. Su misión es recoger los logs de todos los componentes desde un lugar centralizado, procesarlos y presentar una interfaz para su explotación masiva.

Broker de mensajería

El *Broker de mensajería* permite la comunicación de los distintos componentes vía mensajes. Se utiliza para implementar el estilo de comunicación por coreografía, los detalles de su uso se presentan en la Sección 4.2.1.

Balanceadores de Carga

Los *Balanceadores de Carga* sirven para atender las diferentes comunicaciones entre componentes, repartiendo la carga de peticiones entre los microservicios disponibles en la plataforma. Estos balanceos en la PI se efectúan del lado del cliente. Para conocer los microservicios que estén ejecutando, los balanceadores consultan al componente Servicio de Descubrimiento.

El funcionamiento consiste en primera instancia identificar al componente o microservicio que se quiera invocar por el nombre con el que se registra en el Servicio de Descubrimiento, para recuperar la cantidad de instancias de ese microservicio y donde se encuentran. Con dicha información, el balanceador ejecuta el algoritmo de balanceo de carga que tenga definido para determinar a qué instancia del microservicio invocar. El Balanceador de Carga aporta abstracción del número de instancias de los componentes de integración existentes, cual es la instancia invocada y su localización (IP). Dispone de

diferentes implementaciones de algoritmos de balanceo, como Round Robin, basados en tiempos de respuestas, aleatorios, etc. Además, permite configurar la realización de reintentos en las peticiones de forma automática en caso de fallo, así como indicar *timeouts*.

4.2 Diseño de ejecución

En esta sección se detalla el diseño de ejecución implementado para los estilos de coreografía y orquestación utilizado para la coordinación de los componentes de integración que están presentes en una SI.

4.2.1 Coreografía

Considerando lo expresado en la Sección 2.3.3, donde se menciona que coreografía utiliza un sistema de eventos con notificación en la comunicación entre servicios, para el modelo de ejecución de una SI con coreografía se decide utilizar un broker de mensajería. El intercambio de mensajes en el mismo se realiza de forma asincrónica. Los componentes de integración que compongan la SI se encargan de agregar o quitar mensajes en las colas, no comunicándose directamente entre ellos. Se plantea una interacción asincrónica con el fin de que un componente no quede esperando la respuesta a la acción de integración del siguiente componente invocado, ya que este tipo de comunicación permite nivelar la velocidad de procesamiento.

El modelo de comunicación que se elige es el de publicador/suscriptor centralizado con intermediarios (ver Sección 2.1), pudiendo cada componente de integración ser publicador, suscriptor o ambos al mismo tiempo. Este modelo es ideal para situaciones de comunicación en grupo, donde un mensaje enviado es requerido por múltiples entidades, como puede suceder en soluciones de integración a construir en la PI.

El sistema de comunicación utilizado es el protocolo estándar AMQP, que destaca por su interoperabilidad y fiabilidad. Es un protocolo que está enfocado en la comunicación de mensajes asíncronos con garantía de entrega, a través de confirmaciones de recepción de mensajes desde el broker al productor y desde los consumidores al broker.

En la Figura 4.2 se detalla la comunicación de los componentes de una SI con la modalidad de coreografía.

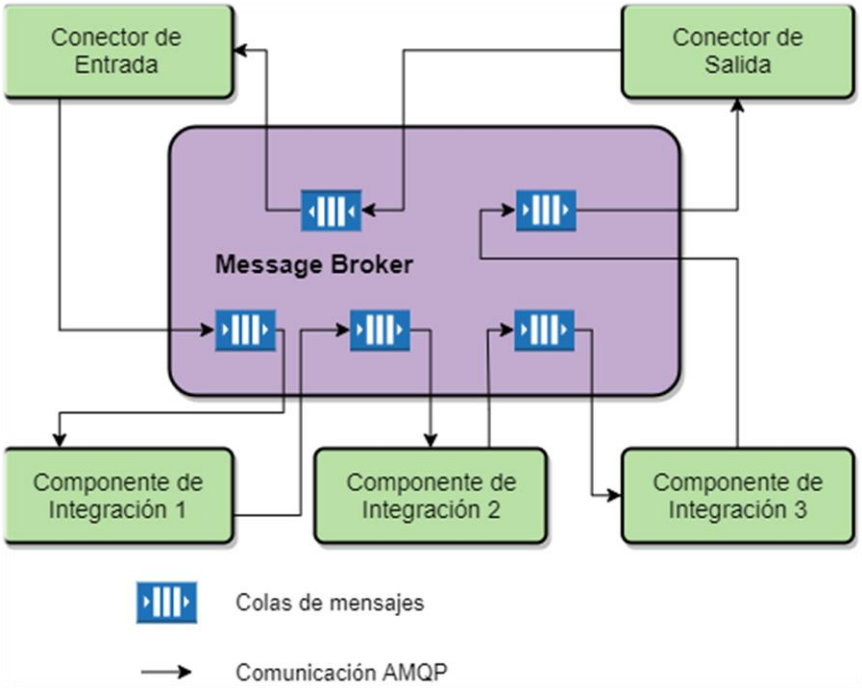

Figura 4.2: Modelo de ejecución de SI con corografía

En la ejecución de una SI con corografía, luego de que un componente ejecute su lógica de procesamiento correspondiente, "publica" el mensaje resultado en una cola ubicada en el broker de mensajería de la plataforma. Cada componente deja mensaje en una única cola específica para la comunicación entre dos componentes. El siguiente componente, luego de su ejecución deja un mensaje en otra cola y es recibido por el componente receptor, y así sucesivamente hasta completar la ejecución de la SI, como indica la Figura 4.2.

4.2.2 Orquestación

Según lo mencionado en la Sección 2.3.3, donde se indica que la orquestación típicamente se ejecuta mediante llamadas de petición/respuesta, se decide implementar el modelo de orquestación de forma sincrónica mediante peticiones HTTP desde el Orquestador a los diferentes componentes de integración que contenga una SI.

En la Figura 4.3 se detalla la comunicación de los componentes de una SI con la modalidad de orquestación.

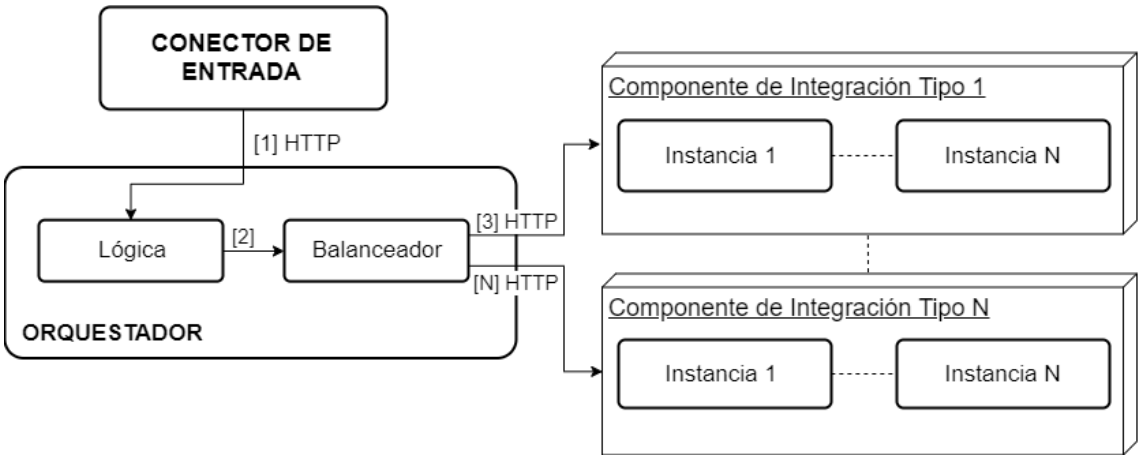

Figura 4.3: Modelo de ejecución de SI con orquestación

El componente Orquestador en primer lugar obtiene la hoja de ruta con los componentes que constituyen una SI. Durante la ejecución bajo esta modalidad, el Orquestador se encarga de enviar los mensajes a cada componente de integración y obtiene un mensaje de respuesta de cada comunicación (resultado de ejecución del componente). Este mensaje de respuesta es el enviado al siguiente componente a ejecutar, y así sucesivamente hasta completar el procesamiento de todos los componentes de la solución.

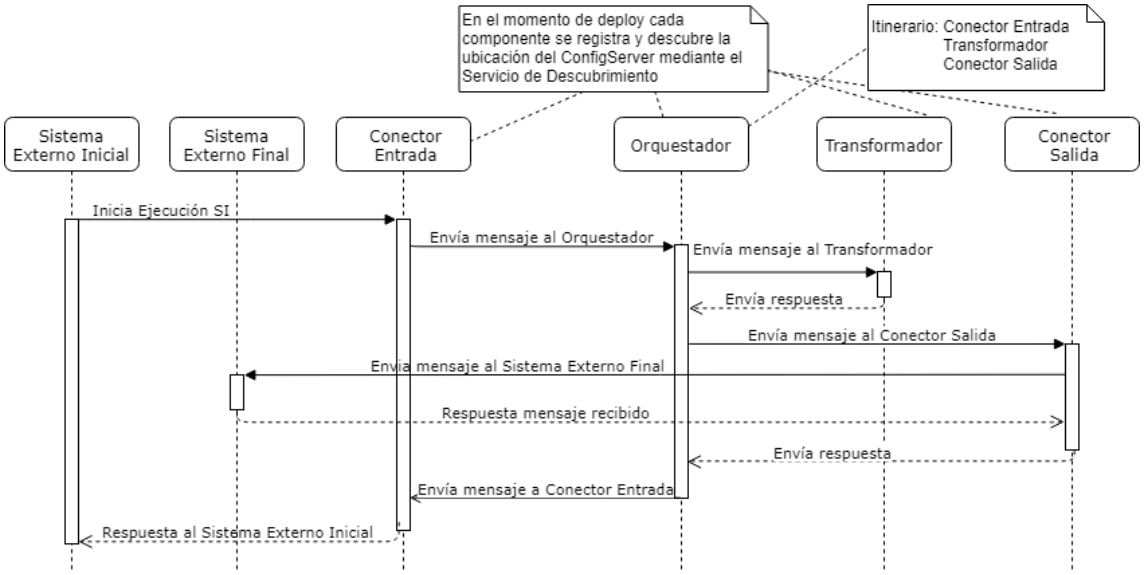

Figura 4.4: Diagrama de secuencia de la ejecución de una solución de integración bajo orquestación.

En la Figura 4.4 se presenta la ejecución de una SI bajo orquestación. Esta comienza cuando el sistema externo invoca la ejecución de una SI sobre el conector de entrada de la plataforma. Mediante el componente de balanceo de

carga, el conector llama al Orquestador mediante una petición HTTP REST. La lógica del orquestador ejecuta el itinerario correspondiente a la SI, realizando peticiones HTTP REST a cada componente presente en la hoja de ruta. Cada invocación sobre los componentes de integración se realiza de forma balanceada sobre todas las instancias del componente. El Orquestador es el encargado de gestionar los errores que puedan ocurrir en la ejecución de la SI.

4.2.3 Formato de mensajes

Los mensajes intercambiados entre los componentes de integración tienen el siguiente formato:

Headers	*idsol*	identificador de solución
	idmensaje	identificador de mensaje
	content-type	tipo de contenido
	idcorrelacion	id de correlación
	tipocomposicion	tipo de ejecución: orquestación o coreografía
	status	código respuesta al orquestador
Payload	*contenido*	

Dentro de los *headers*, los campos **idsol**, **idmensaje** y **content-type** son cabezales que se encuentran en el mensaje enviado desde el sistema que inicia la ejecución de una SI. Los *headers* **idcorrelacion** y **tipocomposicion** se agregan en el conector de entrada. El cabezal idcorrelacion es un identificador que sirve para trazar todo el "recorrido" que hace un mensaje al ingresar a la plataforma. El cabezal tipocomposicion representa el tipo de ejecución de la SI, cuyos valores posibles son orquestación o coreografía, y se obtiene de la configuración de la solución. El *header* **status** solo se agrega a mensajes que ejecutar bajo orquestación y define el código de respuesta que envía el componente una vez finalizado su ejecución al orquestador. El *payload* contiene el cuerpo del mensaje.

4.3 Diseño de componentes de integración

En esta sección se presenta el diseño de los componentes de integración implementados que proporciona la Plataforma de Integración para construir las Soluciones de Integración.

4.3.1 Conector de Salida, Transformación y Enriquecedor

El diseño de estos tres componentes de integración es similar.

El Conector de Salida es el componente de integración encargado de comunicarse con una aplicación de cliente externa a la PI. El objetivo es brindar y/o solicitar información a un sistema externo, formando parte de un paso de la Solución de Integración a la que pertenece.

El componente de integración Transformación se encarga de efectuar transformaciones de datos.

El componente de integración Enriquecedor tiene como fin agregar o enriquecer un mensaje con información obtenida de una fuente externa. Básicamente se comporta como el EIP *Content Enricher* (ver Figura A. 6).

En la Figura 4.5 se detalla el diseño para estos componentes.

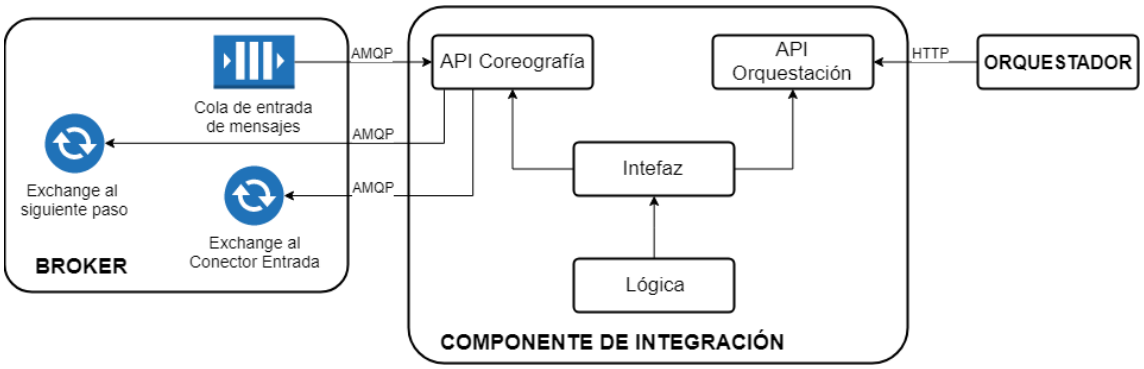

Figura 4.5: Diseño de Transformación, Enriquecedor y Conector de Salida

Cada componente tiene su lógica de ejecución. Por ejemplo, para el componente Conector de Salida, entre otras cosas lo que se debe especificar es la ruta en donde se encuentra la aplicación cliente a ser invocada, junto a los parámetros importantes que contiene el mensaje utilizado en dicha invocación.

Los componentes poseen una API que se utiliza para ejecuciones con coreografía y otra API para ejecuciones con orquestación.

Para coreografía, se define una cola de entrada donde se almacenen todos los mensajes que el componente reciba. A su vez se fijan dos *exchanges*, uno para invocar al siguiente componente en la Solución de Integración, y otro para comunicarse con el Conector de Entrada en el caso de que se produzca un error en la ejecución del componente en cuestión y se necesite notificar el resultado al cliente inicial que ejecuta la SI.

4.3.2 Conector de entrada

El Conector de Entrada es el componente de integración que resuelve las solicitudes a la plataforma de aplicaciones de clientes externos y dispara la ejecución de una SI.

En la Figura 4.6 se observa el diseño de este componente.

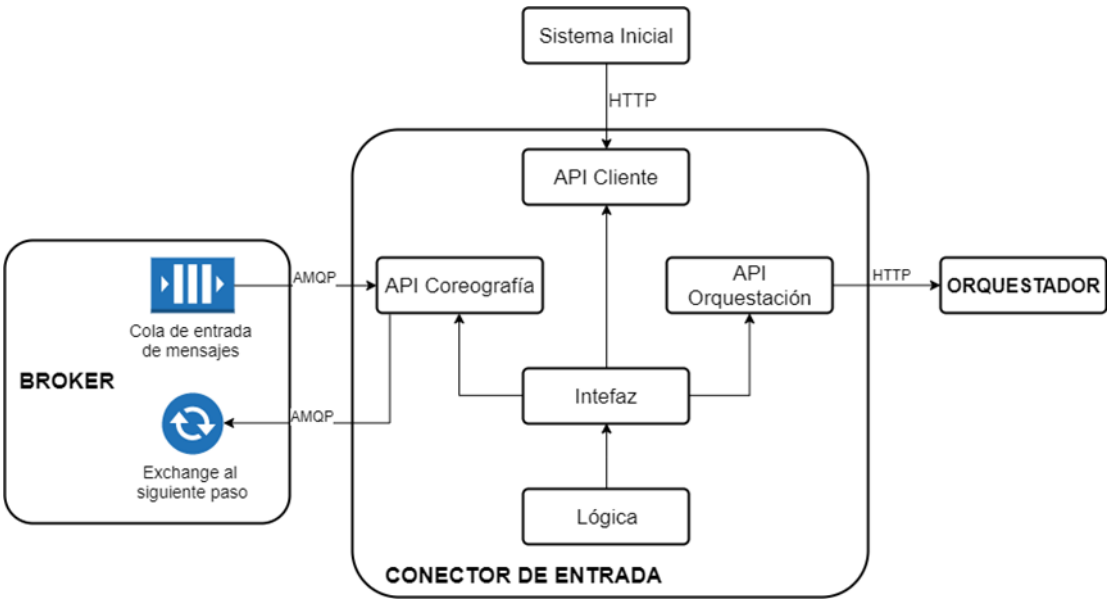

Figura 4.6: Diseño conector de entrada

Como se puede observar el diseño es similar al de los componentes de la sección anterior, con la salvedad que al orquestador es el conector de entrada el que lo invoca para que se encargue de ejecutar la SI. En la Sección 5.2.1 se detalla a bajo nivel el funcionamiento de este componente.

4.4 Diseño en base a patrones

Como se menciona en la Sección 3.3 de alcance, se utilizaron patrones de diseño para introducir atributos de calidad a la plataforma. En la Figura 4.7 se puede

ver un diagrama con los patrones utilizados según las categorías y agrupaciones definidas por Richardson [17]. A continuación, se presenta un desarrollo de los patrones.

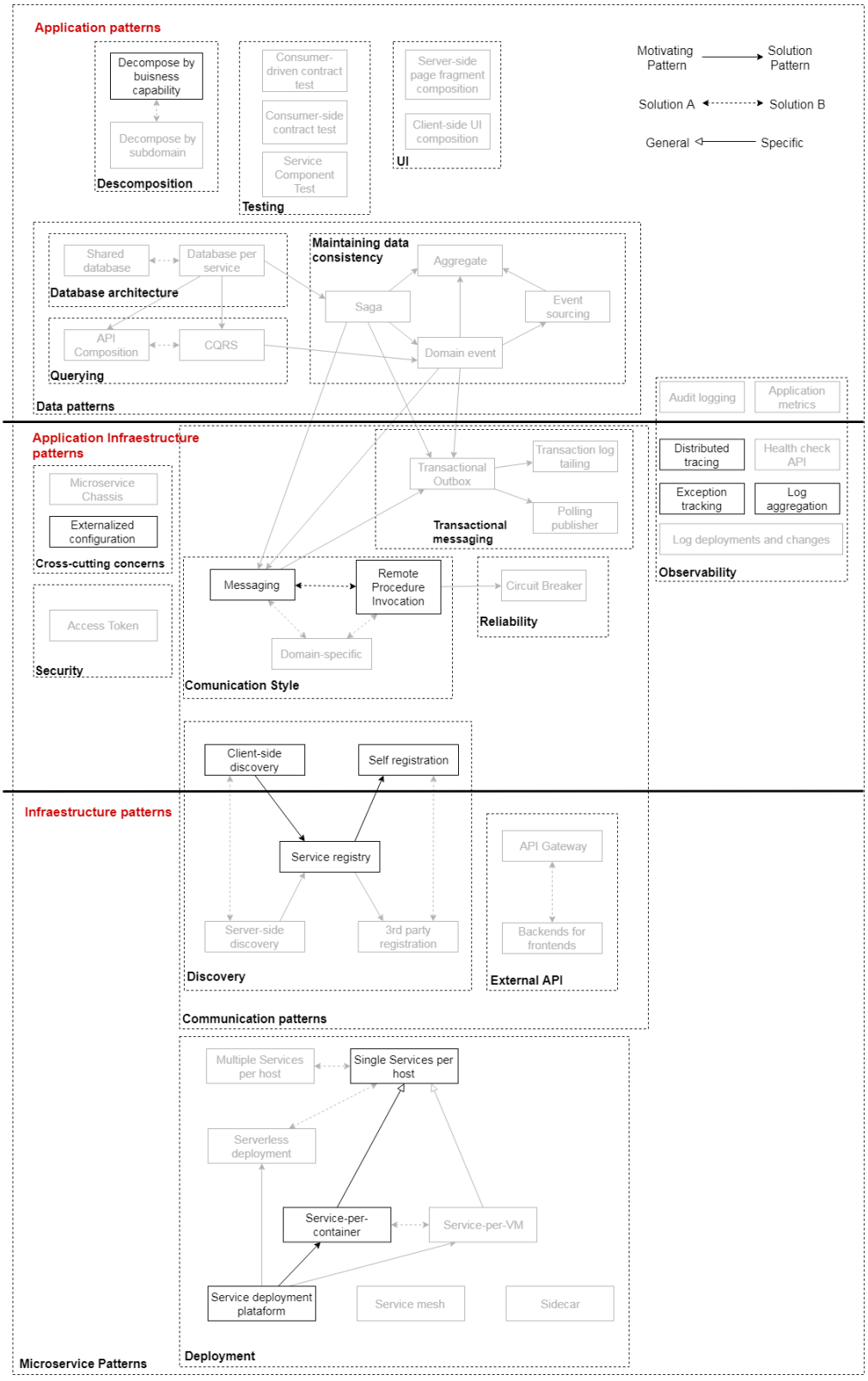

Figura 4.7: Diagrama de patrones utilizados en la PI[29]

[29] Se modifica la imagen original para resaltar los patrones utilizados. La imagen original puede encontrarse en [17]

4.4.1 Patrones de Aplicaciones

Descomposición: se aplica el patrón de descomposición por capacidad del negocio, se definen los microservicios en base a las capacidades de las soluciones de integración (por ejemplo, transformación).

4.4.2 Patrones de Infraestructura de Aplicaciones

Descubrimiento: se realiza desde el cliente. Cuando los componentes inician se registran en el servicio de registro. Cuando otro componente necesita conocer la ubicación del servicio, consulta el registro para obtener la ubicación.

Mensajería: este patrón propone utilizar mensajería asincrónica para la comunicación entre los procesos. Este tipo de mensajería se utiliza para coordinar los mensajes cuando la composición de los microservicios es mediante coreografía.

Configuración externalizada: se utiliza un componente que actúa como concentrador de todas las configuraciones de los microservicios que componen la plataforma. De esta forma es más sencillo administrar y cambiar el comportamiento de los componentes.

Traza distribuida: a cada petición que recibe la plataforma se le agregan identificadores que permiten trazar la petición a lo largo de toda la plataforma. Los identificadores se agregarán a los cabezales de todos los mensajes que circulan en la plataforma.

Agregación de logs: se implementa un componente que se encarga de centralizar los logs de todos los microservicios de la plataforma. De esta forma es posible supervisar la ejecución de todos los microservicios en un solo lugar.

Seguimiento de excepciones: se utiliza el componente de agregación de logs para realizar un seguimiento de los errores y excepciones que lancen los distintos microservicios.

4.4.3 Patrones de Infraestructura

Despliegue: se utiliza el patrón de un servicio por contenedor, de esta forma el mal funcionamiento del microservicio o cualquier tarea administrativa sobre el mismo no afecta a otros servicios. También permite prevenir problemas

asociados al aumento de carga en los servicios, los que no afectarán el entorno de ejecución de estos. Permite escalar rápidamente los servicios que se requieran levantando nuevas instancias de un contenedor. Cada componente de integración desplegado solo atiende a una solución particular. Por lo tanto, si dos soluciones utilizan el mismo componente de integración, se deberá desplegar dos instancias del componente en diferentes contenedores, una por cada solución.

4.5 Discusión respecto arquitectura de referencia

La propuesta a realizar en este proyecto se enfoca en brindar una implementación a la PI que se presenta en la tesis de referencia, enfocándose en aspectos de bajo nivel para usuarios de perfil técnico, como el diseño de los componentes, y modalidad de ejecución de SI, dejando de lado interfaces para gestionar y/o administrar la plataforma.

Al igual que la tesis de referencia, el desglose realizado se basa en patrones y buenas prácticas propuestas por Richardson.

La solución propuesta para ejecución con coreografía en este trabajo sigue la misma línea que la propuesta en la tesis de referencia, mientras que para la propuesta con orquestación se presentaron algunas diferencias. Andrés en su tesis propone concentrar los logs de los componentes en el orquestador para enviarlos al Gestor de Logs, en cambio es más fácil y práctico de implementar que cada componente o microservicio se encargue de enviar al Gestor sus propios logs. Esto debido a que la lógica que ejecuta cada componente es independiente al orquestador.

A la propuesta de referencia se le agrega la trazabilidad de las solicitudes sobre la plataforma. Esto se considera importante debido a que la plataforma ejecuta sobre una arquitectura de microservicios de característica distribuida.

Por otra parte, en la tesis de referencia se comenta que la comunicación mediante orquestación sea asincrónica. Tomando como referencia [16], para ejecutar con orquestación es conveniente peticiones solicitud/respuesta, por lo que se considera realizar dicha comunicación de manera sincrónica.

5 Implementación

En este capítulo se detallan las tecnologías utilizadas en el desarrollo y los principales aspectos de la solución. En la Sección 5.1 se presenta una descripción general de la implementación, enumerando las tecnologías utilizadas en cada componente y una descripción de su uso. La Sección 5.2 presenta aspectos relevantes de la solución e implementación, mientras que en la Sección 5.3 se presenta un caso de estudio junto a las pruebas realizadas para validar la solución. Finalmente, los problemas encontrados referentes a la implementación a lo largo del trabajo se detallan en la Sección 5.4.

5.1 Descripción general

En la implementación de la plataforma se utilizaron distintas tecnologías que permitieron cumplir con los objetivos planteados. En la Figura 5.1 se puede ver el diagrama de componentes y las tecnologías utilizadas en cada uno.

Framework Spring Cloud [30] se utilizó en la implementación de los microservicios. Ofrece algunos componentes que permiten desarrollar fácilmente algunos de los requerimientos de la PI. La utilización de la herramienta se debe a una sugerencia de los tutores y a la experiencia previa de los integrantes en el desarrollo de aplicaciones con Spring.

[30] https://spring.io/projects/spring-cloud

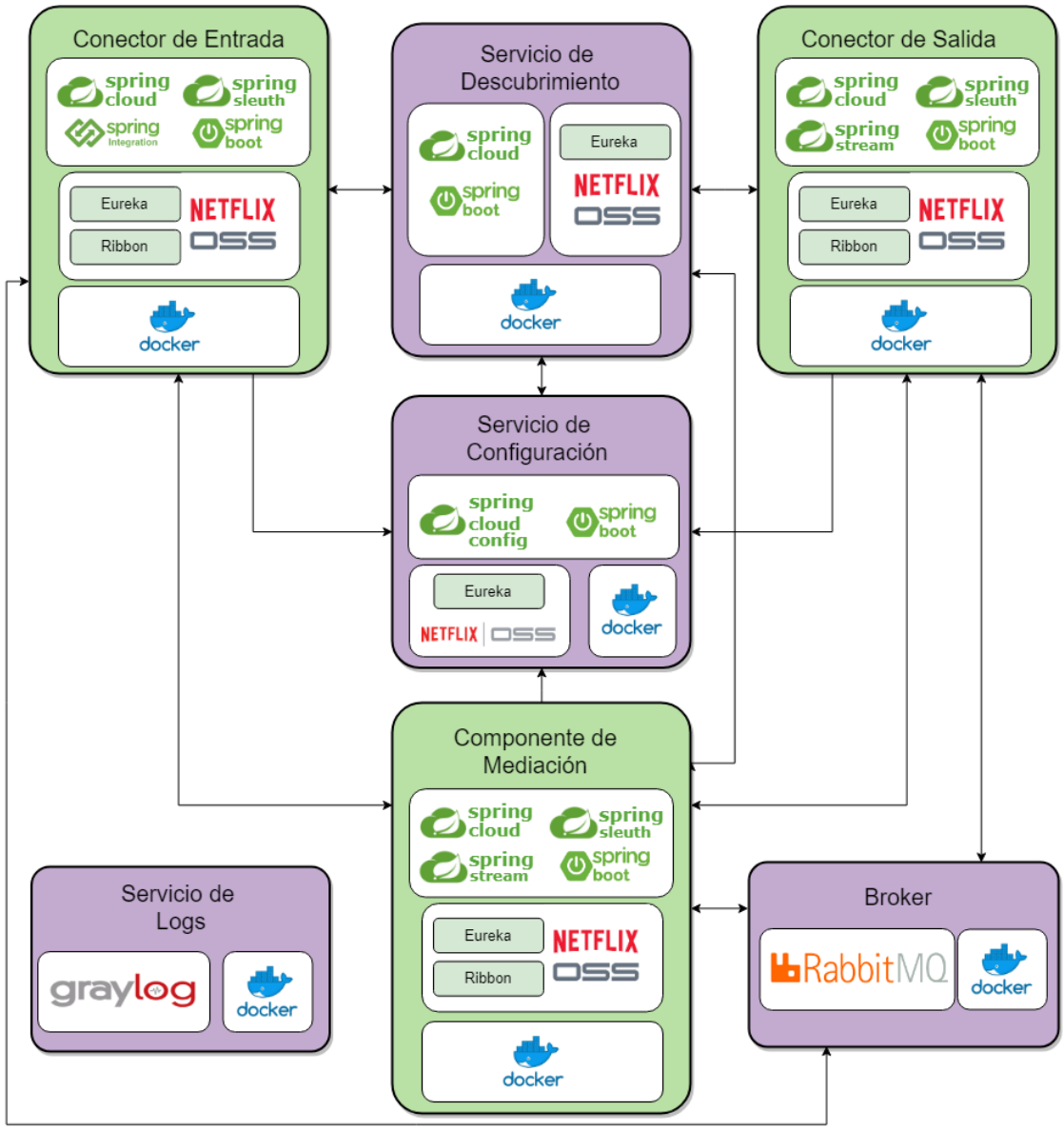

Figura 5.1: Tecnologías utilizadas en cada componente

Spring Boot [31] permite crear aplicaciones *Spring* rápidamente. Dicho componente fue aprovechado en la construcción de todos los microservicios que participan en la plataforma.

Spring Cloud Netflix[32] es una de las tecnologías más importantes utilizadas en el proyecto, provee integración entre aplicaciones *Spring Boot* y la *suite NetflixOSS*. Ofrece servicios de descubrimiento a través de Eureka, servicio de ruteo, balanceo de carga del lado del cliente mediante *Ribbon*, implementación

[31] https://spring.io/projects/spring-boot

[32] https://spring.io/projects/spring-cloud-netflix

del patrón circuit breaker mediante *Hystrix*. Durante el desarrollo de la plataforma, el proyecto *Spring Cloud Netflix* entró en modo de mantenimiento[33], pero dado el grado de avance de la implementación no fue posible cambiar a la tecnología recomendada para las distintas funcionalidades.

Spring Cloud Config[34] permite construir la entidad Configuración Externalizada de la plataforma. Ofrece la posibilidad de centralizar la configuración de los microservicios en un repositorio en Git, de esta forma resulta sencillo cambiar el comportamiento de los microservicios sin necesidad de modificar archivos de configuración distribuidos en varios hosts.

Spring Cloud Stream[35] y *Spring Integration*[36] fueron las tecnologías utilizadas para implementar la comunicación entre los microservicios, permitiendo comunicaciones del estilo *one-way* y *request-response* con aplicaciones externas. Al utilizar *Spring Integration* en el conector de entrada, se dotó a la plataforma de una mayor capacidad de comunicación con sistemas externos por la gran variedad de *endpoints* que ofrece la herramienta.

Spring Cloud Sleuth [37] permite implementar el patrón de *trazabilidad distribuida*. La trazabilidad comienza cuando se recibe una petición desde un sistema externo, el conector de entrada agrega al cabezal del mensaje tres identificadores que permiten trazar el mensaje a lo largo de la plataforma. En la Sección 5.2.2 se describe el significado de los identificadores y su funcionamiento.

Logback[38], es la librería que permite generar los *logs* de información de los microservicios. Permite una integración sencilla con *Spring Boot* y brinda la posibilidad de enviar los logs al centralizador de logs. La configuración por defecto de *Spring Boot* canaliza en *Logback* los mensajes enviados a través de

---

[33] https://spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now#spring-cloud-netflix-projects-entering-maintenance-mode

[34] https://spring.io/projects/spring-cloud-config

[35] https://spring.io/projects/spring-cloud-stream

[36] https://spring.io/projects/spring-integration

[37] https://spring.io/projects/spring-cloud-sleuth

[38] https://logback.qos.ch

las fachadas *commons-logging* y *slf4j*, por lo que se considera valiosa su utilización.

Para implementar el componente *centralizador de logs* existen varias soluciones en el mercado que permiten resolver el problema, pero se analizaron dos soluciones: *ELK*[39] y Graylog[40]. A grandes rasgos, ambos productos ofrecen los mismos beneficios y soluciones, pero la decisión de utilizar *Graylog* se basó en la experiencia previa del equipo en el uso de esta herramienta.

Para el broker de mensajería, se vieron dos soluciones: *Kafka*[41] y *RabbitMQ*[42]. Se decide utilizar RabbitMQ por las capacidades de integración con *Spring Cloud*. Como se mencionó en el capítulo 4.2.1, el sistema de comunicación utilizado es el protocolo estándar AMQP. Los intercambiadores serán de tipo *topic exchange* y las vinculaciones serán condicionales con patrón de reconocimiento.

Para el despliegue de los microservicios se analizaron las opciones de *Docker*[43] y *OpenShift*[44]. El primero es un gestor de contenedores, mientras que el segundo es una herramienta más completa, que se basa en *Docker* para la creación y gestión de contenedores y en *Kubernetes*[45] para la gestión y orquestación de grupos de contenedores. Luego de pruebas de despliegue, se encontraron algunos problemas con el entorno de OpenShift, mientras que hacerlo directamente en *Docker* fue bastante más sencillo. Por este motivo es que el despliegue se decide realizar sobre *Docker*.

[39] https://www.elastic.co/es/elk-stack

[40] https://www.graylog.org/

[41] https://kafka.apache.org/

[42] https://www.rabbitmq.com/

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

[44] https://www.openshift.com/

[45] https://kubernetes.io/es/

5.2 Aspectos relevantes

En la siguiente sección se presentan algunos aspectos relevantes de la implementación de la PI. Los mismos hacen referencia a aspectos positivos y negativos de la plataforma.

5.2.1 Conector de Entrada

El conector de entrada se implementó con Spring Integration. Esta decisión se tomó como resultado de las pruebas de concepto, en las que se presentaron problemas para lograr comunicaciones del estilo *request-response*. Es el único componente desarrollado con esta tecnología. El diseño logrado, permite cambiar los *endpoints* de entrada, dotando al componente -y a la plataforma- la posibilidad de cambiar fácilmente el protocolo de comunicación con el exterior.

En la Figura 5.2 se detalla el funcionamiento de este componente a bajo nivel.

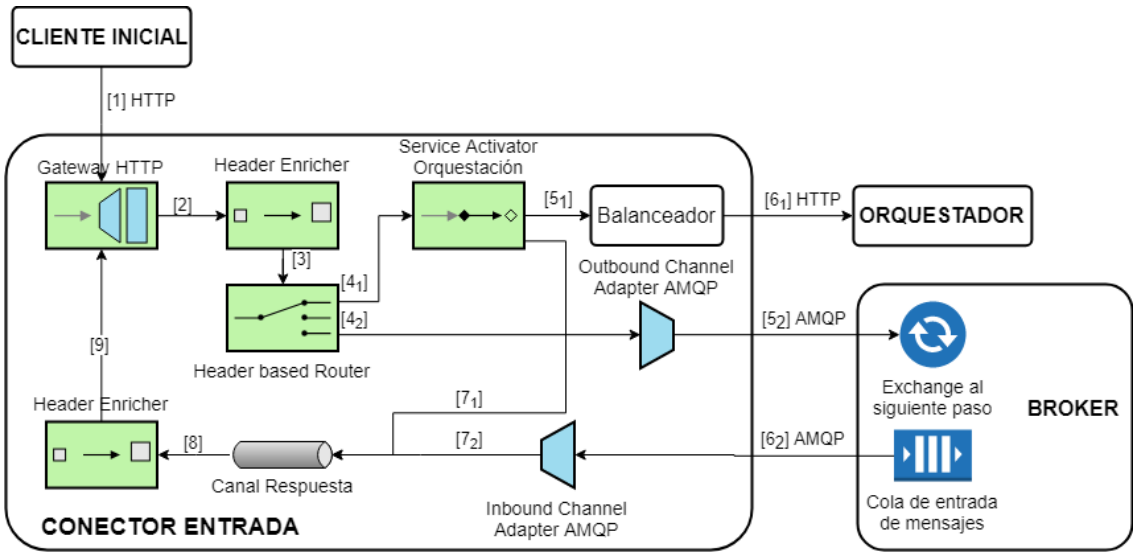

Figura 5.2: Funcionamiento conector de entrada

El Conector recibe el mensaje desde el cliente que desea ejecutar la SI que este dispara, a través de un *Gateway* o adaptador de canal de entrada HTTP. Allí se establece el tiempo límite permitido para responder al cliente (*reply timeout*).

Luego el mensaje pasa a un enriquecedor de cabezales (*Header Enricher*) que se encarga de agregar como *headers* al mensaje, el identificador de correlación y la modalidad de ejecución de la Solución de Integración (coreografía u orquestación) que se obtiene de la configuración de la SI. En este paso se

almacena y relaciona el canal de respuesta al cliente con el identificador de correlación agregado al mensaje.

Posteriormente el mensaje llega al ruteo de cabezales basado en valor (*Header Value Router*), en donde dependiendo del valor del *header* agregado anteriormente sobre la modalidad de ejecución de la SI, se ruteará el mensaje hacia el canal correspondiente. Si la SI que está ejecutando corresponde a coreografía, el mensaje es enviado al broker de mensajería a través del intercambiador correspondiente configurado para el Conector de Entrada, para ser consumido por el próximo componente de integración a ejecutar. En cambio, si la SI corresponde a orquestación, el mensaje es enviado al componente Orquestador, encargado de instrumentar la ejecución de los componentes de integración siguientes.

Por otra parte, para ejecución con coreografía, se define una cola para el Conector de Entrada donde recibirá los mensajes enviados por el resto de los componentes, ya sea cuando se produzca un error o cuando finalice la ejecución de la SI y se desee enviar el resultado o respuesta de esta al cliente inicial.

Para las respuestas del Conector de Entrada al cliente inicial, tanto orquestación como coreografía utilizan un mismo canal que invoca a un *Header Enricher,* el cual establece el canal de respuesta almacenado tomando el identificador de correlación previamente cargado como *header* del mensaje.

5.2.2 Trazabilidad

En las arquitecturas distribuidas existen problemas al momento de hacer el seguimiento de las solicitudes que llegan al sistema. El problema radica en identificar los servicios por los que atraviesa la solicitud y poder generar una traza que indique los servicios consumidos y detectar posibles errores de ejecución. Para resolver este problema se decide utilizar el patrón de trazabilidad distribuida.

En el sistema construido, la trazabilidad de las peticiones tiene dos partes importantes: la primera corresponde a la identificación que asigna el cliente inicial y la segunda es la identificación que asigna la plataforma. Para la primera, es necesario que el cliente inicial agregue un identificador en el cabezal del mensaje que está enviando, el nombre del cabezal debe ser *id_mensaje* y

puede tomar el valor que el cliente desee. La trazabilidad dentro de la plataforma se logra usando *Spring Cloud Sleuth* en el componente de entrada. Esta librería agrega a todos los mensajes recibidos los identificadores de *traceId*, *spanId y parentSpanId*.

TraceId: identifica a la solicitud a lo largo de todo su ciclo de vida dentro de la plataforma (un identificador para la transacción de forma global).

SpanId: identifica a la solicitud dentro de un microservicio, junto al *traceId* es posible determinar el punto exacto por el que está atravesando la solicitud dentro de la plataforma.

ParentSpanId: identifica la operación que invoca el servicio actual, de esta forma es posible armar un árbol de trazabilidad, identificando así todos los servicios invocados y sus dependencias.

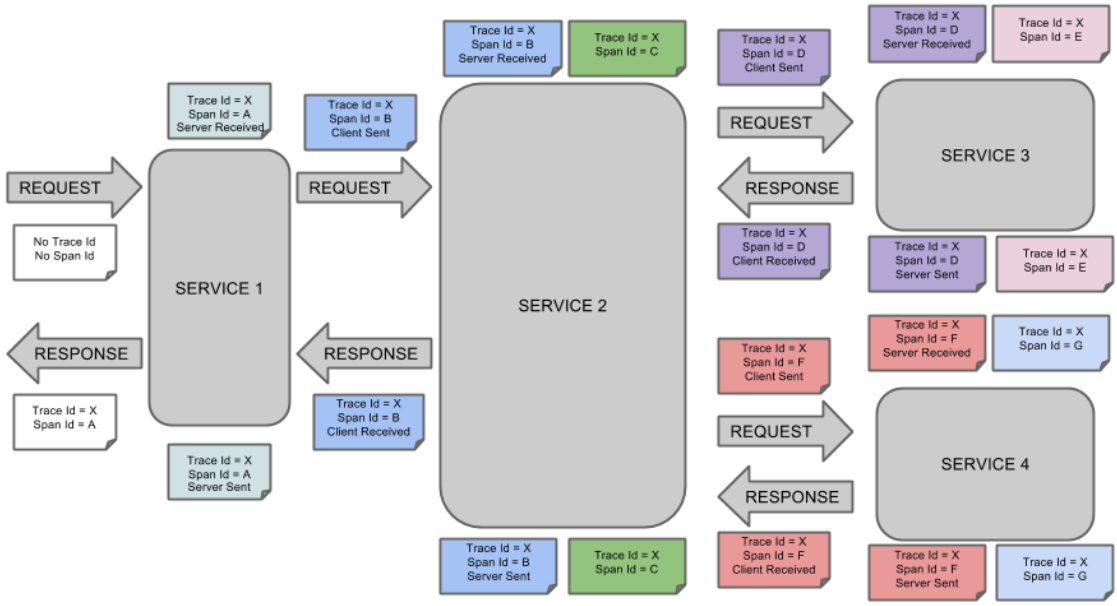

Figura 5.3: Funcionamiento de los identificadores TraceId y SpanId. [29]

En la Figura 5.3, se puede observar cómo se utilizan los identificadores *TraceId* y *SpanId* descritos anteriormente. Luego de recibir una solicitud, el primer servicio le asigna un nuevo valor al *TraceId* y *SpanId*. Al enviar la solicitud al segundo servicio, se genera un nuevo valor para el *SpanId* (que identifica la operación actual) y este proceso continúa mientras la solicitud es procesada por todos los servicios.

Spring Cloud Sleuth permite integrarse con *Logback*, logrando que toda la información que se envía a través de *logs* contenga los identificadores generados por la plataforma. *Sleuth* se beneficia de *B3-Propagation*[46], permitiendo que los identificadores se propaguen en todos los mensajes que generan los componentes de la plataforma. Como la información de los *logs* se envía a *Graylog*, es posible detectar errores y reconstruir el pasaje de los mensajes dentro de la plataforma.

5.2.3 Diseño de componentes

Un aspecto positivo del diseño es que los componentes *Transformación, Enriquecedor y Conector de Salida* pueden ser reutilizados por los desarrolladores para generar nuevos componentes.

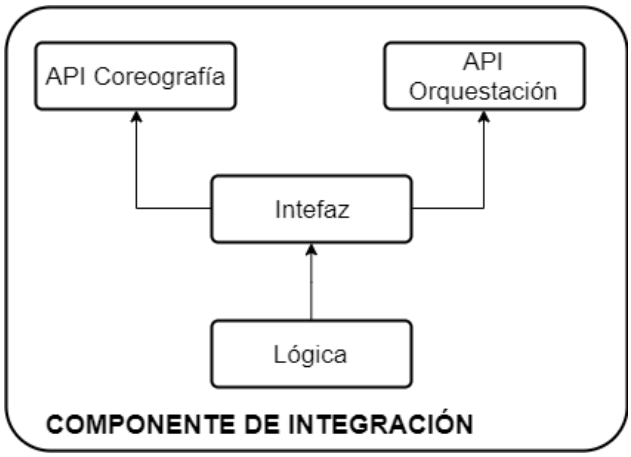

Figura 5.4: Diseño de un componente de Integración

Como se observa en la Figura 5.4, esto es posible por la separación entre la capa de comunicación y la lógica del componente. Los desarrolladores solo deben enfocarse en la lógica que desean entregarle al nuevo componente. De esta forma se logra tener una plataforma extensible con relación a los componentes disponibles para las Soluciones de Integración.

5.2.4 Orquestación

La comunicación de los servicios mediante orquestación planteó la necesidad de evaluar herramientas del mercado que permiten este tipo de ejecución. Para

[46] https://github.com/apache/incubator-zipkin-b3-propagation

ello, es que se realizaron pruebas de concepto para utilizar algunos de los orquestadores de microservicios que existen en el mercado (Conductor[47] de Netflix, Baker[48] de ING Bank, Cadence[49] de Uber).

Todos ellos se basan en el concepto de crear flujos de procesamiento. Rápidamente se dejó de lado las herramientas *Baker y Cadence* por brindar escasa documentación referida a la instalación, configuración y funcionamiento de la herramienta, no se encontraron ejemplos de uso en la comunidad, por lo que se las puede catalogar de herramientas poco maduras.

Para el caso de la herramienta *Conductor*, se encontró buena documentación y una serie de ejemplos de uso que permitían realizar pruebas sin mayores inconvenientes. De todas formas, se encontraron problemas para la instalación de la herramienta ya que la versión descargada del repositorio Git contenía errores que no permitieron su correcto funcionamiento. Posteriormente se encontró una nueva versión que corregía los errores y funcionaba correctamente. La prueba arroja que el funcionamiento de *Conductor* no era el esperado, debido a que el orquestador va dejando las tareas pendientes en colas de tareas asociadas a los distintos servicios orquestados. Estos servicios deben consultar permanentemente esas colas para obtener sus tareas pendientes. Si bien este comportamiento evita las inundaciones de pedidos sobre el orquestador, no parece la estrategia más adecuada para el fin que se pretende dar a este componente en la PI, donde un orquestador estará dedicado a una única SI.

Haciendo foco sobre el tiempo de procesamiento de la Solución de Integración, sumado a no encontrar una herramienta *open source* que cumpla con el servicio deseado, se toma la decisión de realizar completamente la implementación del orquestador. Su comportamiento consiste en enviar peticiones sincrónicas a los servicios vía mensajes HTTP y obtener un mensaje como respuesta. El orquestador conoce el siguiente servicio a invocar mediante una propiedad en su configuración, que contiene para cada solución de integración la hoja de ruta

[47] https://netflix.github.io/conductor/

[48] https://ing-bank.github.io/baker/

[49] https://github.com/uber/cadence

de servicios. Mediante el componente *Ribbon* se consulta el servicio de descubrimiento y se obtiene la ubicación de una instancia del próximo servicio a invocar.

5.2.5 API Coreografía

En primer lugar, se define una interfaz con los canales utilizados por el componente de integración (colas y *exchanges*).

Por otra parte, se define una clase donde contiene los métodos encargados de implementar la lógica de ejecución que se debe realizar cada vez que el componente reciba un mensaje a través de la cola definida en el mismo. El método tiene el nombre *receive* y posee como parámetro de entrada el mensaje de tipo *Message* (tipo de dato que provee Spring) obtenido desde la cola. Estos métodos se encuentran referenciados con la anotación de Stream Cloud *@StreamListener*, donde se establece en el campo *target* el nombre del canal definido para recibir mensajes en la cola. La ejecución del método finaliza cuando se realiza la operación *send* del mensaje resultado, desde un *exchange* definido previamente.

5.2.6 API Orquestación

Para el intercambio entre el orquestador y los componentes de integración, se define un mensaje canónico en la PI. Dicho mensaje se representa con el formato JSON, mediante el tipo de datos *String*. A continuación, se muestra esta representación:

```
{
        "Headers":{
                "Status": codigoRespuesta
                "idsol": s1
                ...
        }
        "Payload":{
                Contenido Mensaje
        }
}
```

Como se puede observar, en *Headers* existe el campo *Status* que contiene el código de respuesta que envía el componente de integración al orquestador. Por ejemplo, el código de respuesta 200 significa que no hubo errores. En dicha

etiqueta, se incluyen también los *headers* que presente el mensaje en cuestión, como por ejemplo *idsol*, *content-type*, etc. El campo *Payload* contiene el cuerpo del mensaje que se está intercambiando.

La API de orquestación está compuesta por un controlador REST (RestController), que contiene un método HTTP POST que atenderá las peticiones que le realiza el orquestador bajo la ruta *ejecutar*. Este método recibe el mensaje canónico que le envía el orquestador, y retorna el mensaje resultado de la ejecución de la lógica del componente, también con formato de mensaje canónico.

5.2.7 Balanceo de carga y escalabilidad

Al desplegar los microservicios como contenedores en Docker, se tiene la posibilidad de escalar rápidamente al iniciar nuevas instancias del microservicio.

Una vez que la nueva instancia se registre en el servicio de descubrimiento, para soluciones que ejecuten con orquestación, se realizarán peticiones de forma balanceada mediante Ribbon a todas las instancias del servicio.

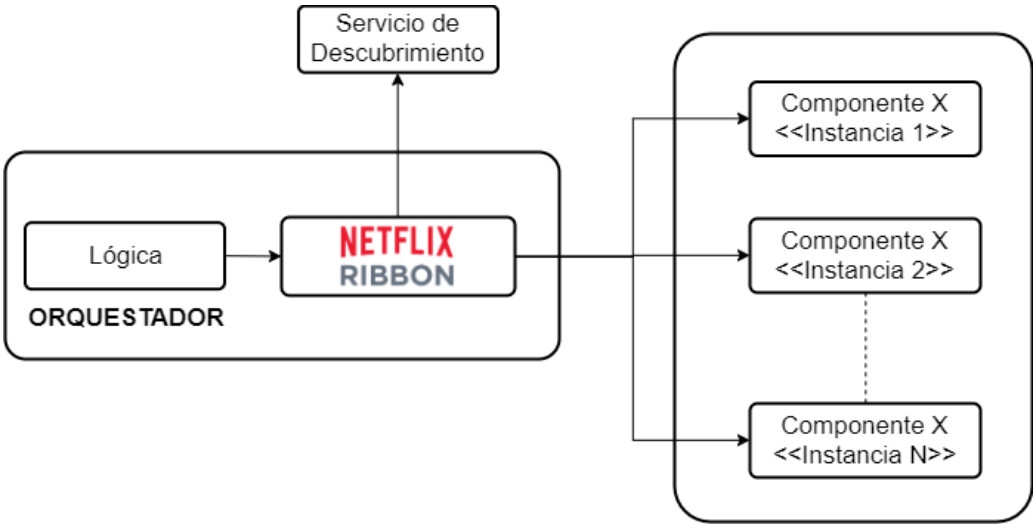

Figura 5.5: Balanceo de carga en orquestación mediante Ribbon

En la Figura 5.5 se observa un diagrama del funcionamiento del balanceo mediante Ribbon en la orquestación. Cuando el orquestador tiene que comunicarse con otro microservicio, el cliente Ribbon consulta al Servicio de Descubrimiento para obtener un listado de todas las instancias activas

registradas para el microservicio. Mediante un algoritmo de *Round Robin* selecciona la instancia con la cual comunicarse y le envía la solicitud.

Las soluciones que ejecuten con coreografía utilizan el broker de mensajería para comunicarse. Este permite varias instancias de consumidores vinculados a una cola, de modo que cada mensaje enviado al broker se consuma de forma balanceada.

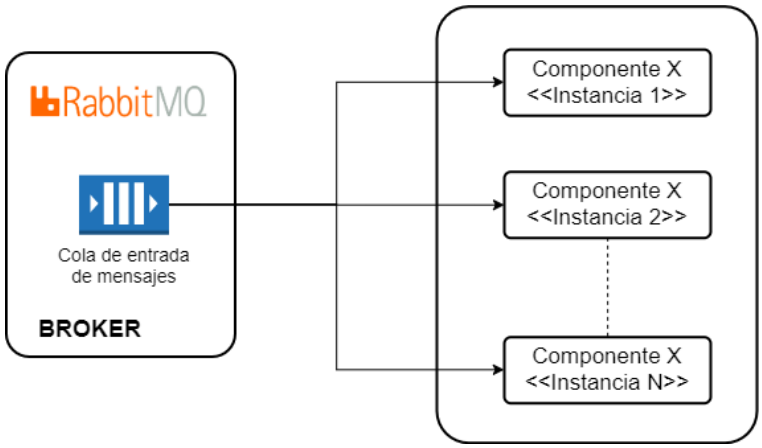

Figura 5.6: Balanceo de carga en coreografía mediante el bróker

En la Figura 5.6 se observa un diagrama del balanceo en coreografía, donde varias instancias del Componente X están vinculadas a una cola de mensajes en el broker. Cuando el broker recibe un mensaje lo deposita en la cola y mediante un algoritmo de *Round Robin* entrega el mensaje a la instancia correspondiente.

Para generar varias instancias de un microservicio, solo se necesita desplegar el microservicio deseado en un nuevo contenedor. Como todas las instancias se registran en el Servicio de Descubrimiento con el mismo identificador, es que quedan registradas como una instancia de un mismo microservicio.

Con las soluciones propuestas anteriormente, es como gran parte de los componentes de la plataforma pueden escalar y ser invocados de forma balanceada. Esto no ocurre para el componente conector de entrada, ya que los sistemas externos no conocen sus instancias disponibles. Un mal entendimiento de la realidad y la necesidad de que la plataforma soportara diversos protocolos de comunicación, influyeron en la decisión de no utilizar un *Gateway* de entrada que centralizara las solicitudes. Por este motivo es que no se pueden mantener múltiples instancias del componente conector de entrada.

5.2.8 Lineamientos para implementar un componente de integración

Como se menciona la Sección 5.2.3, el diseño de los componentes permite a los desarrolladores expandir la plataforma con nuevos componentes de integración.

Para ello podría reutilizarse los componentes que ya brinda la plataforma y hacer las modificaciones necesarias para implementar la nueva lógica.

Aquí es necesario identificar si se desea agregar un componente de entrada o un componente de integración intermediario.

Componente de entrada

En este caso debería reutilizarse el componente *conectorEntrada* que ya brinda la plataforma. El diseño de dicho componente y el uso de la tecnología *Spring Integration* permite cambiar rápidamente el protocolo de comunicación con el sistema inicial. En la Figura 5.2 se observa el diseño a bajo nivel del conector, el sistema inicial establece la comunicación con un *gateway* específico para el protocolo de comunicación que utiliza el sistema inicial.

Al cambiar el *gateway* por uno que implemente el protocolo de comunicación del nuevo sistema, ya se obtendría un nuevo componente conector de entrada, capaz de comunicarse con el nuevo sistema. *Spring Integration* ofrece una gran variedad de *gateways* y adaptadores de canal que permiten la comunicación con sistemas externos. Brinda soporte para comunicación HTTP, FTP, soporta conexiones TCP y UDP, JDBC, JMS, Web Services, etc.[50]

Para finalizar, será necesario configurar la cola de entrada de mensajes dentro del *broker* y el *exchange* con el que debe comunicarse. Esta configuración se encuentra en el archivo XML de nombre *http-inbound-gateway.xml*.

[50] Un listado exhaustivo se puede encontrar en https://docs.spring.io/spring-integration/docs/2.0.0.RELEASE/reference/htmlsingle/#spring-integration-adapters

```
<!-- configuracion de rabbit, exchanges, queues, and bindings -->

<rabbit:topic-exchange
        name="conectorEntradaMessages"
        durable="true">
</rabbit:topic-exchange>

<rabbit:queue name="conectorEntradaQueue" durable="true"  ></rabbit:queue>

<rabbit:topic-exchange name="replayMessages" >
        <rabbit:bindings>
                <rabbit:binding queue="conectorEntradaQueue" pattern="#"/>
        </rabbit:bindings>
</rabbit:topic-exchange>
```

Figura 5.7: Extracto del archivo de configuración http-inbound-gateway.xml

En la Figura 5.7 se muestra la configuración que debe modificarse en el archivo XML *http-inbound-gateway.xml*. La primer entrada *<rabbit:topic-exchange>* define el *exchange* del siguiente paso en la SI. La entrada *<rabbit:queue* define> especifica la cola donde recibirá los mensajes y en el último *<rabbit:topic-exchange>* se asocia la cola al *exchange* donde el resto de los componentes enviarán los mensajes.

Componente de integración intermediario

Al igual que con el conector de entrada, el diseño logrado para los componentes de integración permite separar la lógica de la interfaz de comunicación, siendo posible la reutilización de los componentes.

Cuando se construye un nuevo componente, solo es necesario modificar las clases asociadas a la lógica y agregar el comportamiento deseado. Las clases relacionadas a los estilos de ejecución no es necesario modificarlas, ya que todos los aspectos necesarios para un buen funcionamiento de la comunicación se ubican en el archivo de configuración dentro del servicio de configuración externalizada.

Una vez que se desarrolla la lógica deseada, resta configurar el componente para que pueda recibir y enviar mensajes al broker. Los detalles para realizar la configuración se encuentran en el Apéndice C.

5.2.9 Lineamientos para configurar una Solución de Integración

La configuración de una Solución de Integración se realiza en el archivo de configuración de cada componente de integración que pertenece a la misma. Por ejemplo, si el componente Transformador pertenece a una SI que ejecuta con la modalidad de coreografía, en el archivo de configuración del microservicio se hallarán las configuraciones de los *exchanges* y colas del *broker* de mensajería que utilizará dicho componente.

El componente orquestador tendrá en su archivo de configuración, el itinerario con los nombres de los microservicios en el orden que debe ejecutar para cada Solución de Integración.

La idea inicial era poder concentrar las configuraciones de cada Solución de Integración en un único archivo de configuración, pero no se logró encontrar la manera de poder implementarlo con Spring Cloud.

En el Apéndice C se detalla la configuración de los componentes y en el Apéndice D los pasos necesarios para el despliegue.

5.3 Casos de estudio y pruebas realizadas

Se plantea la elaboración de un caso de estudio, con el objetivo de comprobar la viabilidad y desempeño del desarrollo de la plataforma. El caso de estudio fue desplegado sobre Docker y se encuentra disponible *online*[51].

Luego de construido el caso de estudio, se realizaron pruebas de desempeño para analizar y verificar el impacto de la carga de trabajo en la PI construida.

5.3.1 Descripción caso de estudio

El caso de estudio consiste en integrar dos sistemas, mediante la PI construida. Para abordar esta problemática, se construye la SI presentada en la Figura 5.8 dentro de la PI.

[51] https://gitlab.fing.edu.uy/open-lins/microservices-platform

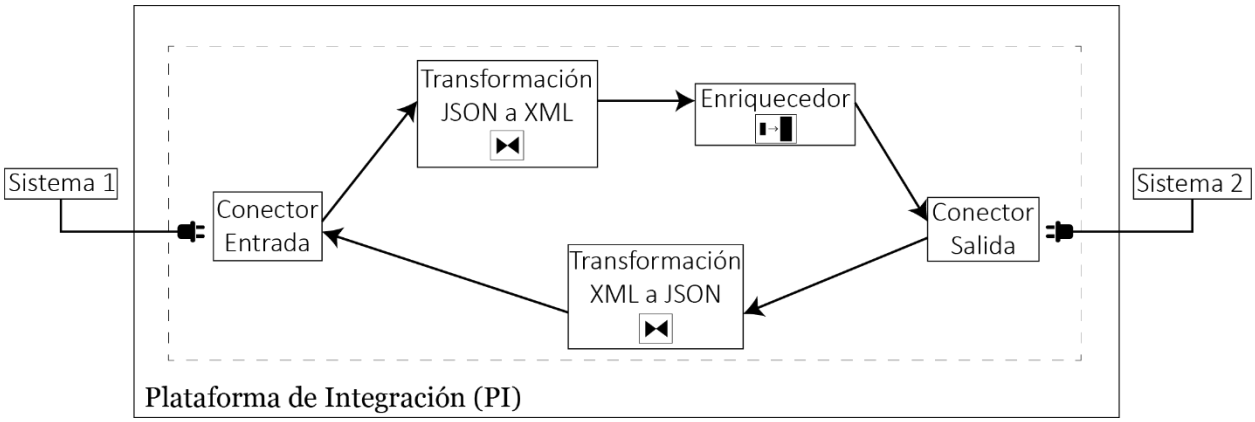

Figura 5.8: Solución de Integración creada en caso de estudio.

El objetivo es lograr que el Sistema 1 obtenga información de diferentes empresas mediante su razón social. Esta información se encuentra en el Sistema 2. El Sistema 2 brinda un servicio web SOAP, donde se obtiene información de una empresa mediante su razón social, mientras que el Sistema 1 maneja un modelo de datos de las empresas diferente al Sistema 2.

La SI se construye con los siguientes componentes de integración: *Conectores de Entrada y Salida, Transformación y Enriquecedor*.

El Conector de Entrada es el encargado de recibir las peticiones del Sistema 1 y luego disparará la ejecución de la SI en la PI. Estas peticiones serán web HTTP REST, donde el cuerpo del mensaje será representado con un formato JSON que contendrá la razón social de la empresa a consultar.

El Transformador JSON a XML justamente se encarga de transformar el formato del mensaje que viene desde el Conector de Entrada de JSON a XML.

El componente Enriquecedor tiene como objetivo agregar al mensaje transformado anteriormente, un *header* de autenticación *HTTP Basic*, generado por un usuario y contraseña que se encuentra en el archivo de configuración para la solución que ejecuta. Este *header* es fundamental para poder consumir el servicio web proporcionado por el Sistema 2, ya que este requiere que el agente de usuario se autentique para responder de manera adecuada a las solicitudes.

El Conector de Salida es el que se encarga de comunicarse con el Sistema 2 sincrónicamente, mediante un servicio web SOAP. La petición o *request* enviada por el conector es el cuerpo del mensaje recibido desde el Enriquecedor, que

contiene la razón social de la empresa a consultar, y como se mencionó anteriormente, se requiere la autenticación de acceso básica, que se encuentra en el encabezado del mensaje. Entre los datos de respuesta del Sistema 2, se encuentra el nombre, RUT, localidad y dirección de la empresa consultada.
El Transformador XML a JSON transforma el formato del mensaje que llega desde el Conector de Salida de XML a JSON.

Por último, es el Conector de Entrada el que le traslada el mensaje que recibe desde el Transformador al Sistema 1.

5.3.2 Pruebas funcionales

Para verificar el correcto funcionamiento de la plataforma se realizan pruebas funcionales, que buscan verificar las distintas situaciones que se pueden alcanzar dentro de la plataforma. Se abarcan los siguientes casos de prueba:

- Ejecución de solución de integración con estilo de ejecución coreografía (caso éxito y error).
- Ejecución de solución de integración con estilo de ejecución orquestación (caso éxito y error).
- Ejecución de solución de integración con estilo de ejecución orquestación con parámetro Razón Social vacío (caso éxito y error).
- Ejecución de solución de integración con estilo de ejecución coreografía con *body* vacío.
- Ejecución de solución de integración con estilo de ejecución orquestación con *header idsol* vacío.
- Ejecución de solución de integración con estilo de ejecución orquestación con *header content-type* vacío.

Los casos de error de las tres primeras pruebas ocurren porque los componentes de integración retornan errores aleatorios (implementados intencionalmente). Un detalle de las pruebas realizadas se puede encontrar en el apéndice E.

5.3.3 Pruebas de desempeño

Las pruebas funcionales permiten evaluar el correcto funcionamiento de la plataforma, pero es muy importante determinar el rendimiento en situaciones donde la carga es mayor. Esto permite encontrar los límites de rendimiento de

la plataforma y las configuraciones iniciales sobre las cuales esta debe escalar. Por tal motivo es que se decide realizar distintas pruebas de estrés mediante JMeter[52] y monitorear el funcionamiento de la plataforma con el producto OneAgent de Dynatrace[53] (se utiliza la versión de prueba de 15 días por ser una herramienta paga). A continuación, se listan las pruebas realizadas:

1. 100 usuarios ejecutando, realizando 1000 consultas con estilo de ejecución orquestación.
2. 100 usuarios ejecutando, realizando 1000 consultas con estilo de ejecución coreografía.
3. 100 usuarios ejecutando, realizando 1000 consultas con ambos estilos de ejecución.
4. 300 usuarios ejecutando, realizando consultas con ambos estilos de ejecución.

Con estas pruebas, se evaluaron los tiempos de respuesta y comportamiento de cada estilo de ejecución de forma aislada y conjunta, con una gran cantidad de usuarios al mismo tiempo.

Todas las pruebas se realizaron en un notebook Dell Latitude-E5570 con las siguientes características:

Procesador: Intel® Core™ i5-6440HQ CPU @ 2.60GHz × 4

SO: Ubuntu 16.04.6 LTS (Xenial Xerus) (kernel 4.15.0-54-generic)

Arquitectura: x86, 64-bit.

RAM: 16GB DDR4-2132

Storage: SSD 250GB

En la Tabla 5.1 se muestra los resultados obtenidos para todas las pruebas, pero solo se analizarán los datos correspondientes a la primera. El resto del análisis se puede encontrar en el Apéndice F.

[52] https://jmeter.apache.org/

[53] https://www.dynatrace.com/platform/oneagent/

Prueba	Muestras	Media en ms	Min	Max	Desv. Estandar	%Error	Rendimiento por segundo	Kb/seg	Media Bytes
1	100000	798	6	5153	613,5	47,4%	121,7	54,3	457,0
2	100000	727	3	3336	413,0	46,9%	135,2	57,4	435,2
3	100000	740	3	7533	959,6	47,1%	131,3	57,2	446,3
4	194220	2350	3	15160	1822,1	46,8%	127,1	56,0	450,9

Tabla 5.1: Tiempo medio de respuesta y rendimiento de las pruebas realizadas con JMeter.

Para las tres primeras pruebas (pruebas con la misma cantidad de muestras), se observa que no hay grandes diferencias en el tiempo medio de respuesta y rendimiento respecto a los estilos de ejecución. En cambio, se notan grandes diferencias en la desviación estándar de las muestras. Esto implica que en coreografía los tiempos de respuesta de las peticiones están más agrupados sobre la media, mientras que en orquestación están más dispersos. En la prueba número tres, la desviación es aún mayor, indicando una mayor dispersión de las muestras. Por ende, se puede concluir que en coreografía se obtienen tiempos de respuesta más constantes, mientras que con orquestación o ambos estilos de ejecución son menos constantes.

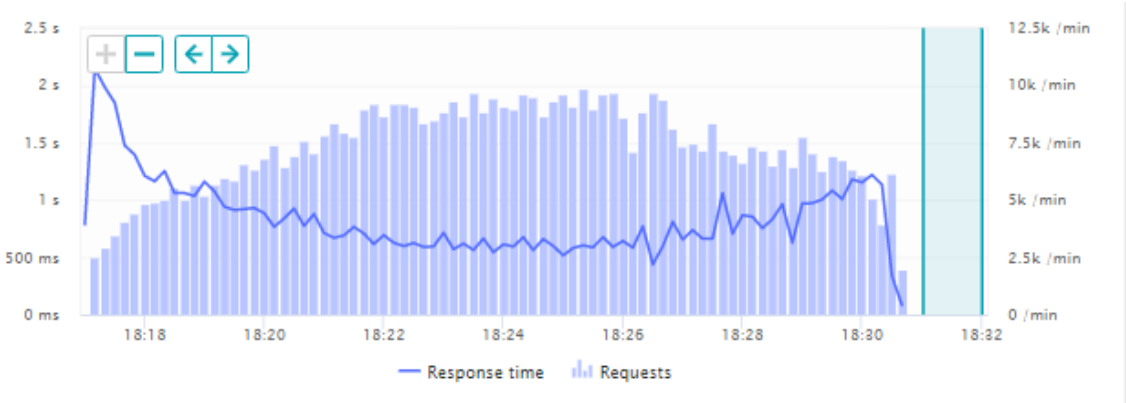

Figura 5.9: Tiempo de respuesta y cantidad de peticiones a lo largo del tiempo para la prueba 1.

En la Figura 5.9 se detalla la distribución de peticiones a lo largo del tiempo. Se observa cómo el tiempo de respuesta comienza en valores altos, mayores a 1 segundo, y van decreciendo hasta alcanzar la media de 800 milisegundos. Hacia el final de la prueba el tiempo de respuesta vuelve a aumentar. Este último comportamiento resulta extraño, por lo que se decide realizar una evaluación del comportamiento de los componentes. En ese análisis, se encontraron algunos problemas de memoria con la *Java Virtual Machine* (JVM) del

conector de entrada. También se puede observar como la plataforma está procesando unas 7000 peticiones por minuto. En la Figura 5.10 se pueden ver los tiempos de respuesta y cantidad de solicitudes procesadas por cada componente de la SI.

Name	Response time median	Failure rate	Requests ▾
TransformacionController transformacion.jar	1.46 ms	0 %	5.01k /min
OrquestadorController orquestador.jar	529 ms	0 %	5.01k /min
conectorEntrada-1 conectorentrada.jar	743 ms	0 %	5.01k /min
EnriquecedorController enriquecedor.jar	1.39 ms	0 %	4.05k /min
ConectorController conectorsalida.jar	256 ms	0 %	4.04k /min
TransformacionController transformacion.jar	2.13 ms	0 %	3.25k /min
EmpresasEndpoint clientefinalsoap.jar	2.33 ms	0 %	3.25k /min
ApplicationsResource eureka.jar	566 µs	0 %	14 /min

Figura 5.10: Tiempos de respuesta por componente

En la Figura 5.11, se grafica el uso de memoria de la JVM en el conector de entrada. Se aprecia como a las 18:26 se alcanza el límite de memoria utilizada en el espacio Old Gen de la JVM, lo que dispara la ejecución del *full garbage collector,* suspendiendo la ejecución de la aplicación. Esto explica el aumento en los tiempos de respuesta al final de la prueba.

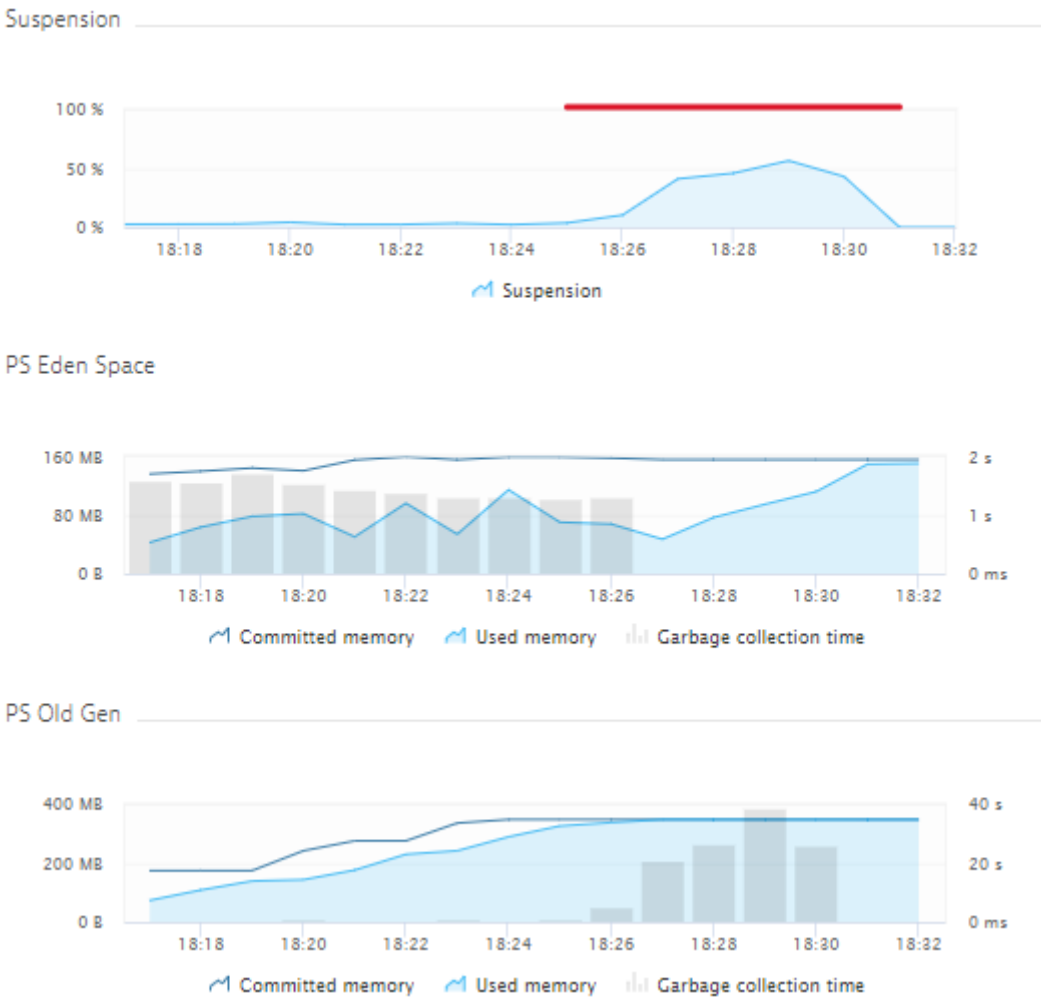

Figura 5.11: Uso de memoria en la JVM del conector de entrada

5.4 Dificultades encontradas

Como todo proyecto de desarrollo, en su transcurso se presentan dificultades de variada índole y este proyecto no es la excepción. El objetivo de esta sección es presentar las dificultades y problemas ocurridos en la etapa de implementación.

5.4.1 Reimplementación del conector de entrada

Durante la prueba de concepto, el conector de entrada se implementó utilizando las mismas librearías de comunicación que el resto de los componentes de integración: Spring Cloud Stream. Esta librería está orientada al manejo de flujo de datos en un sentido (*one way*), por lo cual no fue posible implementar el tipo de solicitudes *request-response*. Para poder cumplir con ello, fue necesario reescribir todo el componente con la tecnología Spring Integration.

5.4.2 Limitación tecnológica y tipo de dato *Message*

Al trabajar con *Spring Cloud* y decidir utilizar el tipo de dato *Message*, se restringe la plataforma al uso del *framework* Spring Cloud para la manipulación de mensajes, ya que no sería posible implementar un microservicio en otro lenguaje. Lo más conveniente hubiera sido utilizar un tipo de mensaje canónico personalizado para la plataforma de tipo *String*, para brindar la posibilidad de utilizar otros lenguajes de programación distintos a Java entre los diferentes microservicios que integren la PI.

El problema de utilizar el tipo de dato *Message* se detecta en una etapa muy avanzada del proyecto y no fue posible tomar acciones para corregirlo, ya que hubiera llevado a una revisión total del desarrollo y un gran atraso en la implementación. Este problema afecta a los dos tipos de comunicación que existen dentro de la PI: orquestación y coreografía. Si bien la comunicación por orquestación entre componentes usa un mensaje canónico personalizado, los métodos implementados en la lógica de los componentes de integración utilizan el tipo de datos de *Spring Cloud*, por lo que no están exentos del problema.

6 Gestión del proyecto

En este capítulo se describe el proceso que el equipo llevo adelante durante la realización de este proyecto.

6.1 Planificación del proyecto

Inicialmente se elaboró una planificación general del proyecto, con el fin de tener una estimación de los tiempos que este llevaría, como se muestra la Figura 6.1.

Hito - Entregables	Periodo		Tarea		
	15-Abr	1-May	nivelación microsevicios		Nivelación de conceptos microservicios
Planificación	1-May	15-May		estudio EIP, tecnologías, soluciones	Estudio patrones EIP, estudio de arquitectura propuesta en maestría, spring cloud, análisis de soluciones existentes, comparación de tecnologías
	15-May	1-Jun	análisis req y definir alcance		Estudio patrones EIP, estudio de arquitectura propuesta en maestría, spring cloud, análisis de soluciones existentes, comparación de tecnologías, comienzo análisis requerimientos
	1-Jun	15-Jun			Análisis requerimientos y definición de alcance, estudio patrones EIP, estudio de arquitectura propuesta en maestría, spring cloud, análisis de soluciones existentes, comparación de tecnologías.
Informe Requerimientos	15-Jun	1-Jul	diseño y arquitectura		Diseño y arquitectura
Informe Evaluación Tecnologías	1-Jul	15-Jul		prototipo	Diseño y arquitectura, comienzo implementación prototipo
Informe Diseño y Arquitectura	15-Jul	1-Ago			Diseño y arquitectura, implementación prototipo
	1-Ago	15-Ago			Implementación prototipo
Demo e informe de avance implementación	15-Ago	1-set	presentación intermedia	implementación y testing	Presentación intermedia, implementación solución
	1-set	15-set			Implementación solución
Informe de avance implementación casos de usos	15-set	1-Oct			Implementación y testing
	1-Oct	15-Oct	documentación final		Implementación y testing
Informe de avance implementación casos de usos	15-Oct	1-Nov			Implementación y testing, documentación
Revisión documentación	1-Nov	15-Nov			Documentación
Revisión documentación	15-Nov	1-Dic			Documentación y presentación final
	1-Dic	15-Dic	presentación final		

Figura 6.1: Planificación general del proyecto

En el diagrama de Gantt de la Figura 6.2 se aprecia la planificación inicial general del proyecto.

Debido a diferentes factores externos, como baja de integrante, licencias y viajes, carga de trabajo por responsabilidades laborales, dedicación a otras

materias y enfermedades, se presentaron desviaciones considerables respecto a la planificación inicial. El diagrama de Gantt presentado en la Figura 6.3, muestra el transcurso real del proyecto.

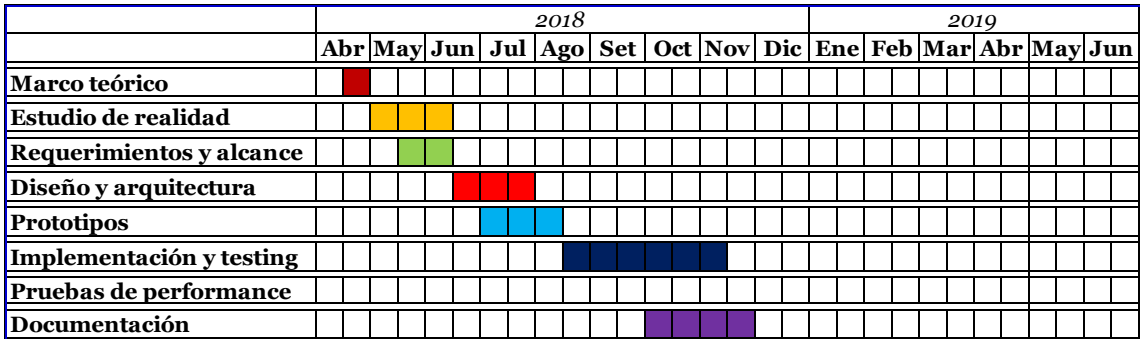

Figura 6.2: Diagrama de Gantt de la planificación inicial del proyecto

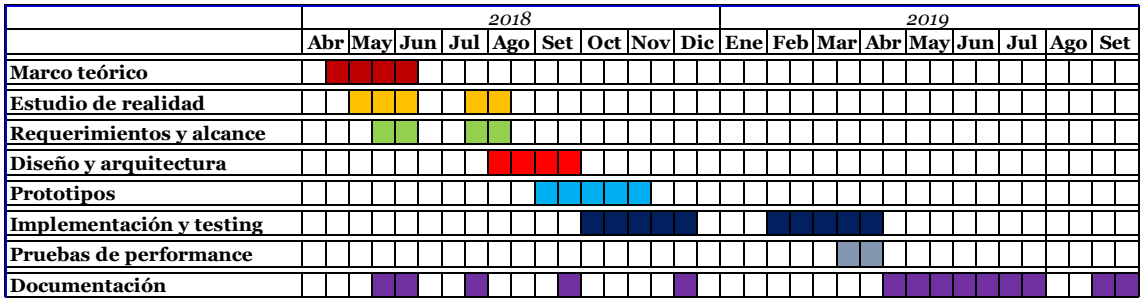

Figura 6.3: Diagrama de Gantt del transcurso real del proyecto

6.2 Etapas del proyecto

El proyecto estuvo dividido en cuatro grandes etapas. En esta sección se mencionan cada una de ellas.

6.2.1 Marco conceptual y análisis del proyecto

Dadas las características del proyecto, los primeros pasos implicaron realizar un estudio sobre el marco conceptual o teórico sobre el tipo de software que se desea construir. Estos conceptos implicaron realizar una investigación y nivelación del equipo sobre la arquitectura de microservicios, así como los beneficios y desafíos que presenta.

El proyecto consistía en la implementación de una plataforma de integración basándose en una arquitectura de referencia [2], por lo que el equipo tuvo que estudiar y comprender dicho diseño y arquitectura.

Se investigaron y analizaron algunas soluciones existentes sobre PI para comprender mejor el negocio de la problemática a desarrollar.

Esta etapa incluyó un análisis de requerimientos iniciales del proyecto definido con los tutores. Luego se definió el alcance final del proyecto, quitando algunos requerimientos previamente definidos.

6.2.2 Diseño de la solución propuesta

Luego de que el equipo incorporara los conocimientos requeridos para desarrollar el proyecto, se da paso al diseño de la arquitectura de la solución propuesta.

Previo a la construcción del prototipo, se estudiaron y compararon diferentes tecnologías para implementar la Plataforma de Integración.

Durante esta etapa se realizó la creación de un prototipo de la PI, para ir incorporando y comprendiendo el funcionamiento de las nuevas tecnologías, así como para mitigar riesgos tempranamente. El prototipo se basó en construir una SI sencilla, ejecutando bajo la modalidad de coreografía.

6.2.3 Implementación y pruebas

Esta etapa, donde el objetivo fue la construcción final del producto, consistió en continuar el desarrollo de la plataforma, tomando como base el prototipo desarrollado previamente.

Se creó el componente Orquestador, y se elaboraron las interfaces correspondientes en los componentes para ejecutar una SI bajo dicha modalidad.

Cuando finalizaba el desarrollando de un componente de integración, se realizaban pruebas del funcionamiento puntual del componente. Luego se probaba si funcionaba correctamente como componente de una solución de integración.

Los tutores del proyecto definieron un caso de estudio con un problema más cercano a la realidad en comparación al utilizado en el prototipo, donde se creó una SI basada en coreografía y otra SI basada en orquestación para cumplir con dicho planteamiento.

Esta etapa fue interrumpida por licencias y vacaciones de los integrantes del equipo.

Finalmente se realizaron pruebas de performance sobre la PI, basándose en la ejecución de las soluciones de integración del caso de estudio definido.

6.2.4 Documentación

A diferencia del resto de las etapas en la que se dividió el proyecto, la documentación se fue elaborando parcial e informalmente a lo largo del mismo. Sin embargo, tomó mayor protagonismo durante el final del proyecto.

La herramienta utilizada para la elaboración del documento fue Microsoft Word, manejando para la edición colaborativa documentos versionados en Google Drive.

6.3 Dificultades encontradas

Durante el transcurso del proyecto se encontraron varios problemas que se debieron superar. Dada la naturaleza del problema, algunos de ellos afectaron el desarrollo normal del proyecto.

6.3.1 Entendimiento de realidad

Una de las primeras dificultades fue entender correctamente la realidad del problema y el producto que se quería construir. Como ya fue indicado, el presente trabajo se basó en una arquitectura de referencia desarrollada en otra tesis. Como material de referencia inicial, se dispuso un artículo que resumía el trabajo final y recién sobre el cierre de la fase de implementación se logró tener acceso al trabajo final de la tesis. Un error cometido por el grupo al inicio del proyecto fue no mantener reuniones con el autor de la tesis de referencia, ya que hubiera aportado mucho valor el poder profundizar su propuesta y evacuar dudas de primera mano.

Otro inconveniente fue que en las reuniones con los tutores no se manejaba la misma terminología, lo que llevó a mal interpretar muchas de las definiciones que se estaban tomando.

6.3.2 Componente Transformación

Durante la prueba de concepto el enfoque se centró en resolver el componente de transformación. En su resolución se perdió mucho tiempo definiendo la lógica y la mejor forma de realizar las transformaciones, cuando ese no era algo esencial en el proyecto. Los tutores identificaron ese problema y se acordó avanzar en el proyecto.

6.3.3 Variedad de tecnologías

En la Sección 5.1 se detallaron todas las tecnologías utilizadas en el proyecto. Algunas referentes a lenguajes de programación, otras al manejo de contenedores, otras para la gestión de logs. Muchas de ellas son complejas y cuentan con basta documentación, y lograr comprender su funcionamiento y alcance fue todo un reto. Por lo tanto, lograr que todas las herramientas y tecnologías funcionen bien en conjunto se tornó difícil. Por este motivo que muchas de las decisiones se basaron en el conocimiento y experiencia previa, así como la compatibilidad con las herramientas ya seleccionadas.

6.3.4 Baja de un integrante

En plena etapa de definición del alcance, se produjo la baja de un integrante. Debido a ello, el resto del equipo debió absorber la carga de trabajo hasta que se realizó una revisión del alcance ya definido. También trajo algunos momentos de incertidumbre que provocaron parte de los atrasos en los meses de junio y julio del 2018.

6.3.5 Hardware insuficiente y entorno de pruebas

A lo largo de todo el proyecto se tuvieron limitaciones para levantar entornos de prueba, ya sea para pruebas de concepto, la propia plataforma o herramientas que se quisieron evaluar. Las computadoras a disposición carecían de los recursos suficientes para levantar los ambientes, es por ese motivo que se empezó a buscar la posibilidad de trabajar con servidores virtuales en la nube, como OpenShift o *Mi nube* de ANTEL[54]. Se descartó OpenShift, ya que el servidor que este brinda gratuitamente tenía pocos recursos. Para el producto

[54] https://minubeantel.uy/

de ANTEL, se pudo acceder a un código promocional para probar el servicio. Se intentó configurar el servicio, pero no fue posible instalar el software requerido para manejar contendores, debido a que no fue posible actualizar el *kernel* del servidor para instalar Docker. Esto no permitió cumplir con el requerimiento de desplegar la PI en la nube. Sin embargo, se cree que no habría ningún inconveniente de realizar dicho despliegue, ya que se utilizó tecnología de contenedores.

Finalizada la etapa de implementación, se logra acceder a un equipo con las características adecuadas para soportar todo el entorno que compone la plataforma. En ese equipo se realizaron todas las pruebas de rendimiento descriptas en la Sección 5.3.3.

6.3.6 Documentación

Los tiempos en la etapa de Documentación se extendieron demasiado. Uno de los motivos fue la falta de práctica de los integrantes en realizar este tipo de documentos, lo cual, sumado al nivel del detalle y formalidad requerido, hizo que se retrasaran los tiempos. También hubo demoras en *feedback* de documentación por parte de los tutores.

6.3.7 Orquestador

Poder realizar pruebas de concepto sobre el componente orquestador *Conductor* mencionado en la Sección 5.2.4 insumió más tiempo del estimado en principio por el equipo, ya que se plantearon inconvenientes en lograr instalar el componente completamente y entender su funcionamiento, algo que finalmente no se logró. Si se toma en cuenta el poco tiempo y el mínimo esfuerzo empleado por el equipo en el desarrollo propio de este componente, claramente se puede concluir que de haber tomado esta decisión inicialmente los tiempos del proyecto no se hubieran desviado por este motivo.

7 Conclusiones y trabajo futuro

Las conclusiones del trabajo se presentan en la Sección 7.1. Luego se describe el trabajo futuro en la Sección 7.2, con las mejoras y desafíos pendientes.

7.1 Conclusiones

Como conclusión general del proyecto se logró validar la propuesta de arquitectura que se utilizó como referencia. Se realizó el diseño y la implementación de una Plataforma de Integración basada en microservicios.

Con respecto al trabajo relacionado en la industria, se encontraron productos iPaaS que afirman estar construidos con microservicios. Si bien en ellos se destacan varias características indicadas en la arquitectura de referencia utilizada en este trabajo, no son de propósito general. Estos productos no especifican estilos de ejecución en sus flujos (coreografía u orquestación), ni indican si están basados en patrones EIP como sucede en plataformas monolíticas. Este trabajo es un primer aporte en esa dirección.

Se logró diseñar dos alternativas de comunicación entre microservicios en la Plataforma de Integración: coreografía mediante un middleware de mensajería asincrónico, y orquestación a través de peticiones HTTP *request/response* sincrónicas.

La solución elaborada para ejecución con coreografía sigue la misma línea que la propuesta en la tesis de referencia. Sin embargo, para la solución bajo orquestación se presentaron algunas diferencias: se define comunicación sincrónica y se determina que cada componente envíe sus logs al Gestor de logs, en contrapartida a la propuesta en la tesis de referencia.

En esta propuesta se agrega la trazabilidad, algo fundamental debido a la característica distribuida que tiene la arquitectura de microservicios.

En el diseño de este trabajo se realiza una refinación de componentes y microservicios más profunda de la efectuada en la propuesta de referencia. Se logra un diseño que separa la capa de comunicación en dos APIs para los estilos

de ejecución de las SI con la lógica de cada componente de integración, permitiendo la reutilización de componentes.

Se tomaron en consideración para el diseño de la PI los patrones de Richardson. La incorporación de estos permite describir la arquitectura de microservicios con patrones.

En lo que respecta a la implementación de la PI, se confirmó que las propuestas de diseño realizadas para ejecutar Soluciones de Integración con coreografía y orquestación fueron viables técnicamente.

En cuanto a las tecnologías utilizadas, se comprueba que Spring Cloud es un *stack* tecnológico completo para implementar y coordinar microservicios en la nube. Se descubrió en Graylog una buena solución para el almacenamiento centralizado de logs, permitiendo realizar diferentes consultas y filtros sobre los datos, entre otros beneficios. La utilización de RabbitMQ como broker de mensajería aporta confiabilidad, colas altamente disponibles, permite la escalabilidad de las aplicaciones, enrutamiento flexible, soporta múltiples protocolos, entre otras ventajas.

Lograr implementar diferentes componentes de integración con la tecnología adecuada para cada caso, remarca una de las propiedades que se presentan al construir microservicios. Se trata de la utilización de Spring Integration en el componente Conector de Entrada y de Spring Stream en el resto de los componentes de integración.

Para permitir que las soluciones ejecuten con orquestación, en primera instancia la intención fue utilizar un componente orquestador del mercado, pero los productos analizados no tuvieron la madurez suficiente para ser incorporados a la PI. Esto derivó en que se decida realizar la implementación del componente. Como ventajas se destacan la fácil implementación y testeo. Una desventaja puede ser que al haberse implementado completamente desde cero, tal vez se haya omitido algún aspecto que afecte el rendimiento de un orquestador.

En referencia a las pruebas de desempeño, no se encontraron grandes diferencias de rendimiento entre los distintos estilos de ejecución, aunque el comportamiento de coreografía se notó más constante en los resultados. Esto se

debe a una degradación en el rendimiento del conector de entrada al utilizar orquestación y el uso de un MOM en coreografía controla y nivela la velocidad con la que los componentes reciben los mensajes de forma asincrónica, a diferencia las peticiones HTTP sincrónicas. Se logró verificar el límite de rendimiento de los componentes a partir del cual sería necesario escalar.

Un aspecto débil de la plataforma es la imposibilidad de escalar el conector de entrada. A pesar de ello, un punto fuerte del conector es la posibilidad de cambiar fácilmente los protocolos de comunicación con sistemas externos, dotando a la plataforma un gran abanico de posibilidades de comunicación.

Si bien no se logró desplegar la plataforma en la nube, el uso de la tecnología de contenedores permitiría este tipo de despliegue sin mayores problemas, ya que una de sus bondades es la portabilidad.

A nivel personal, el desarrollo de este proyecto permitió conocer y experimentar con la arquitectura de microservicios. Se logró comprobar la utilidad de desarrollar cada microservicio con la tecnología que mejor se adapta al problema. Permitió minimizar la complejidad de cada servicio, lo que simplificó el desarrollo, las pruebas y mantenimiento de cada componente.

A pesar de estos beneficios, la coordinación de todos los microservicios es uno de los aspectos más complejos. Esta complejidad se relaciona con la cantidad de microservicios que participan en la plataforma, a mayor cantidad de microservicios, mayor es la complejidad de coordinarlos.

Al utilizar un broker de mensajería en el estilo de ejecución coreografía, se simplifica la comunicación entre los componentes, pero es necesario configurar varias características en cada componente para que el flujo de comunicación sea el deseado. Para el caso de orquestación se vuelve más sencillo ya que el orquestador mantiene la hoja de ruta de ejecución y es quién inicia las comunicaciones con los componentes. De todas formas, podría incurrir en problemas de desempeño si inunda de mensajes a un componente o si mantiene muchas solicitudes instanciadas con otros componentes.

Se verifica que la trazabilidad de las solicitudes es uno de los patrones deseables en todo desarrollo basado en microservicios, ya que encontrar los puntos de fallos en este tipo de arquitectura distribuida es una tarea muy compleja.

La realización de este proyecto permitió comprobar que implementar una plataforma de integración basada en microservicios puede ser muy útil, en especial cuando se deben mantener muchos flujos de integración complejos o que comparten componentes. La arquitectura permite crear componentes con una lógica más sencilla, simples de mantener y capaces de escalar si hay un gran uso de estos. Si bien la comunicación de los componentes puede ser compleja, resulta interesante asumir el desafío para acceder a los beneficios descritos anteriormente.

Para desarrollar flujos de integración simples, puede no resultar apropiado implementar un microservicio por cada componente de integración, ya que el esfuerzo de implementar, coordinar y monitorear el flujo aumenta considerablemente en comparación con un único microservicio que implemente todo el flujo de integración.

Se logró comprobar que los estilos de ejecución (coreografía y orquestación) no presentan grandes diferencias en lo que respecta a desempeño, pero el estilo de coreografía tiene un rendimiento más estable a lo largo del tiempo.

La implementación del estilo coreografía resultó ser más compleja que la orquestación, debido a que en cada componente debe configurarse el flujo de comunicación sobre el broker. La comunicación asincrónica de la coreografía es una ventaja sobre el sincronismo de orquestación, ya que el broker oficia de intermediario, evitando inundar de peticiones al resto de los componentes.

Como no se implementó de manera asincrónica la orquestación, no se puede afirmar que la coreografía asincrónica es el estilo de ejecución ideal para la plataforma de integración (o para microservicios). Además, la elección del estilo de ejecución depende de la cantidad de componentes que integran el flujo de comunicación, ya que resulta más sencillo coordinar microservicios en orquestación. Considerando la solución construida en el caso de estudio con ambos estilos de ejecución, el de coreografía asíncrona es la mejor opción para la comunicación de los microservicios dentro de la plataforma desarrollada.

7.2 Trabajo futuro

En esta sección se presentan algunos aspectos de la plataforma que deben mejorarse o desarrollarse para brindar mejores atributos de calidad al producto.

7.2.1 Gateway de entrada

Uno de los aspectos débiles de la plataforma es la no escalabilidad del conector de entrada. Una posible solución sería agregar un *gateway* al componente disparador de entrada. De esta forma, todos los pedidos del sistema externo inicial deberán canalizarse a través del *gateway*, y este se encarga de balancear los pedidos sobre las distintas instancias del conector de entrada. Se sugiere el uso del componente Spring Cloud Gateway, ya que el hecho de ser un componente de Spring, la integración con los componentes ya desarrollados no debería implicar grandes problemas. Otra opción es el uso del componente Zuul, perteneciente a Spring Cloud Netflix, pero a finales del 2018 entro en modo mantenimiento[55] y ya no se desarrollarán nuevas funcionalidades para el mismo.

7.2.2 Tolerancia a fallos

Dentro de los requerimientos iniciales se pedía que la plataforma fuera tolerante a fallos. Este requerimiento se retira del alcance final porque su implementación retrasaría aún más la finalización del proyecto. Nuevamente, Spring Cloud Netflix cuenta con un componente capaz de resolver este tema. Se trata de Hystrix, pero al igual que Zuul, entro en modo mantenimiento. Spring Cloud sugiere el uso de Resilience4j[56], una librería liviana para tolerancia a fallas, diseñada para Java 8 y programación funcional.

7.2.3 Seguridad: autenticación y autorización

Un aspecto muy importante de una plataforma de integración es la seguridad. Es necesario que todas las peticiones de sistemas externos se sometan a mecanismos de autenticación y autorización, como forma de evitar accesos

[55] https://spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now#spring-cloud-netflix-projects-entering-maintenance-mode

[56] https://github.com/resilience4j/resilience4j

indebidos a la plataforma. Un componente capaz de dotar de lógica de autenticación y autorización es Spring Cloud Security[57].

7.2.4 Archivo único de configuración

Para el despliegue de soluciones es necesario configurar varios aspectos de los componentes de integración, lo que implica revisar y modificar varios archivos de configuración. Para simplificar los despliegues de las soluciones, se podría crear un archivo único de configuración que contenga todos los parámetros y ajustes necesarios para desplegar la solución. De esta forma se centraliza toda la configuración relacionada a la solución siendo más sencillo su administración y despliegue.

7.2.5 Nuevos componentes de integración

En el presente trabajo se desarrollaron los componentes de integración transformador y enriquecedor, que quedan disponibles para ser usados por cualquier solución de integración. Para continuar extendiendo y enriqueciendo la plataforma, se recomienda la creación de otros componentes, por ejemplo: componentes de ruteo, notificación, etc.

7.2.6 Monitoreo

Otro aspecto no abordado en este proyecto es la posibilidad de monitorizar la plataforma de integración. Es crítico conocer el estado y salud de los distintos microservicios que componen la plataforma, ya que a partir de allí se podrán tomar decisiones que mejoren la performance, ya sea escalando ciertos componentes o detectando problemas que impliquen cambios en la lógica de estos. Para las pruebas de performance se utilizó la herramienta OneAgent de Dynatrace, pero al no ser una herramienta *open source* no es viable su uso. En la industria existen soluciones *open source* que permiten realizar esta tarea. Un ejemplo es Checkmk Raw[58], una herramienta basada en Nagios[59] que permite el monitoreo de aplicaciones, servidores, redes *on-premise* o en la nube. Existen

[57] https://spring.io/projects/spring-cloud-security

[58] https://checkmk.com/

[59] https://www.nagios.org/

otras herramientas que combinadas pueden crear un buen sistema de monitoreo. Por ejemplo, se puede usar Prometheus[60] para la extracción de métricas de las distintas aplicaciones y visualizar los datos con Grafana[61].

[60] https://prometheus.io/

[61] https://grafana.com/

Referencias

[1] G. Hohpe y B. Woolf, Enterprise integration patterns: Designing, building, and deploying messaging solutions, Addison-Wesley Professional, 2004.

[2] A. Nebel, «Arquitectura de Microservicios para Plataformas de Integración,» 2019.

[3] T. Bishop y R. Karne, «A survey of Middleware.,» de *Proceedings of the ISCA 18th International Conference Computers and Their Applications*, Honolulu, Hawaii, USA, 2003.

[4] J. Al-Jaroodi, I. Jawhar, A. Al-Dhaheri, F. Al-Abdouli y N. Mohamed, «Security middleware approaches and issues for ubiquitous applications,» *Computers & Mathematics with Applications,* vol. 60, nº 2, pp. 187-197, 2010.

[5] E. Curry, «Message-oriented middleware,» de *Middleware for communications*, 2004, pp. 1-28.

[6] 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: 04 06 2019].

[7] «AMQP Advanced Message Queuing Protocol. Protocol Specification.,» Junio 2006. [En línea]. Available: https://www.rabbitmq.com/resources/specs/amqp0-8.pdf. [Último acceso: 15 Mayo 2019].

[8] L. González y R. Ruggia, «A reference architecture for integration platforms supporting cross-organizational collaboration.,» de *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, Brussels, Belgium, 2015.

[9] D. Chappell, Enterprise Service Bus: Theory in Practice, O'Reilly, 2004.

[10] M. Pezzini y B. J. Lheureux, «Integration platform as a service: moving integration to the cloud.,» Gartner, 2011.

[11] J. Lewis y M. Fowler, «Microservices,» 2014. [En línea]. Available: https://martinfowler.com/articles/microservices.html. [Último acceso: 5 4 2019].

[12] S. Newman, Building microservices: designing fine-grained systems., O'Reilly Media, Inc., 2015.

[13] I. Nadareishvili, R. Mitra, M. McLarty y M. Amundsen, Microservice architecture: aligning principles, practices, and culture., O'Reilly Media, Inc., 2016.

[14] AWS, «¿En qué consisten las operaciones de desarrollo?,» 2019. [En línea]. Available: https://aws.amazon.com/es/devops/what-is-devops. [Último acceso: 10 4 2019].

[15] M. Fowler, «Microservices Prerequisites,» 2014. [En línea]. Available: https://martinfowler.com/bliki/MicroservicePrerequisites.html. [Último acceso: 11 Setiembre 2019].

[16] 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.

[17] C. Richardson, «Pattern: Microservice Architecture,» [En línea]. Available: https://microservices.io/patterns/microservices.html. [Último

acceso: 20 4 2019].

[18] A. Ladera, «¿Qué son los microservicios?,» [En línea]. Available: https://www2.deloitte.com/es/es/pages/technology/articles/microservicios.htm. [Último acceso: 20 4 2019].

[19] «RoboMQ Microservices Platform,» [En línea]. Available: https://www.robomq.io/microservices-platform/. [Último acceso: 27 Abril 2019].

[20] «RoboMQ Integration Flow Designer.,» [En línea]. Available: https://www.robomq.io/integration-flow-designer/. [Último acceso: 27 Abril 2019].

[21] «RoboMQ Key Features,» [En línea]. Available: https://www.robomq.io/docs/. [Último acceso: 27 Abril 2019].

[22] « What is an integration flow?,» Elastic.io, [En línea]. Available: https://docs.elastic.io/getting-started/integration-flow.html. [Último acceso: 20 Abril 2019].

[23] «Elastic.io new challenges with cloud integration,» 2016. [En línea]. Available: https://info.leanix.net/hubfs/Website_Downloads/EA_Connect_Day_Presentations/EAConnectDay2016-Elasticio-Cloud-Integration.pdf. [Último acceso: 20 Abril 2019].

[24] M. Eder, «Hypervisor-vs. container-based virtualization.,» de *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, Munich, Alemania, 2015-2016.

[25] L. Zheng, M. Kihl, L. Qinghua y J. A. Andersson, «Performance Overhead Comparison between Hypervisor and Container based Virtualization,» de *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, Taipei, Taiwan, 2017.

[26] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar y M. Steinder, «Performance Evaluation of Microservices Architectures Using Containers,» de *2015 IEEE 14th International Symposium on Network Computing and Applications*, Cambridge, MA, USA, 2015.

[27] C. Dukatz y S. Parthasarathy, «Application Containers Transcending the private-public cloud frontier,» Accenture, 2015. [En línea]. Available: https://www.accenture.com/_acnmedia/Accenture/Conversion-Assets/DotCom/Documents/Global/PDF/Dualpub_23/Accenture-Application-Containers.pdf. [Último acceso: 20 Abril 2019].

[28] B. Scholl, «Microservices and Containers: Patterns for Scalable Agility in the Age of Digital Disruption,» Agosto 2017. [En línea]. Available: https://www.oracle.com/technetwork/articles/dsl/microservices-containers-pattern-wp.pdf. [Último acceso: 12 Mayo 2019].

[29] A. Cole, S. Gibb, M. Grzejszczak, D. Syer y J. Bryant, «Spring Cloud Sleuth,» [En línea]. Available: https://cloud.spring.io/spring-cloud-sleuth/reference/html/. [Último acceso: Setiembre 2019].

[30] G. Hohpe y B. Woolf, «Enterprise Integration Patterns. Messaging Patterns,» [En línea]. Available: https://www.enterpriseintegrationpatterns.com/patterns/messaging/. [Último acceso: 10 Junio 2019].

[31] P. B. Kruchten, «The 4+1 View Model of architecture,» *IEEE Software,* vol. 12, nº 6, pp. 42-50, 1995.

Apéndices

A. *Enterprise Integration Patterns* (EIP)

La integración empresarial es demasiado compleja para ser resuelta con un enfoque simple de "libro de recetas". En cambio, los patrones pueden proporcionar una guía documentando el tipo de experiencia que normalmente vive solo en la cabeza de los arquitectos: son soluciones aceptadas para problemas recurrentes dentro de un contexto dado. Los patrones son lo suficientemente abstractos como para aplicarse a la mayoría de las tecnologías de integración, pero lo suficientemente específicos como para proporcionar una guía práctica a los diseñadores y arquitectos. Los patrones también proporcionan un vocabulario para que los desarrolladores describan de manera eficiente su solución.

Los patrones actuales se centran en la mensajería, que forma la base de la mayoría de los otros patrones de integración.

Se conocen aproximadamente 65 patrones de mensajería, organizados como se puede apreciar en la Figura A. 1 [30]:

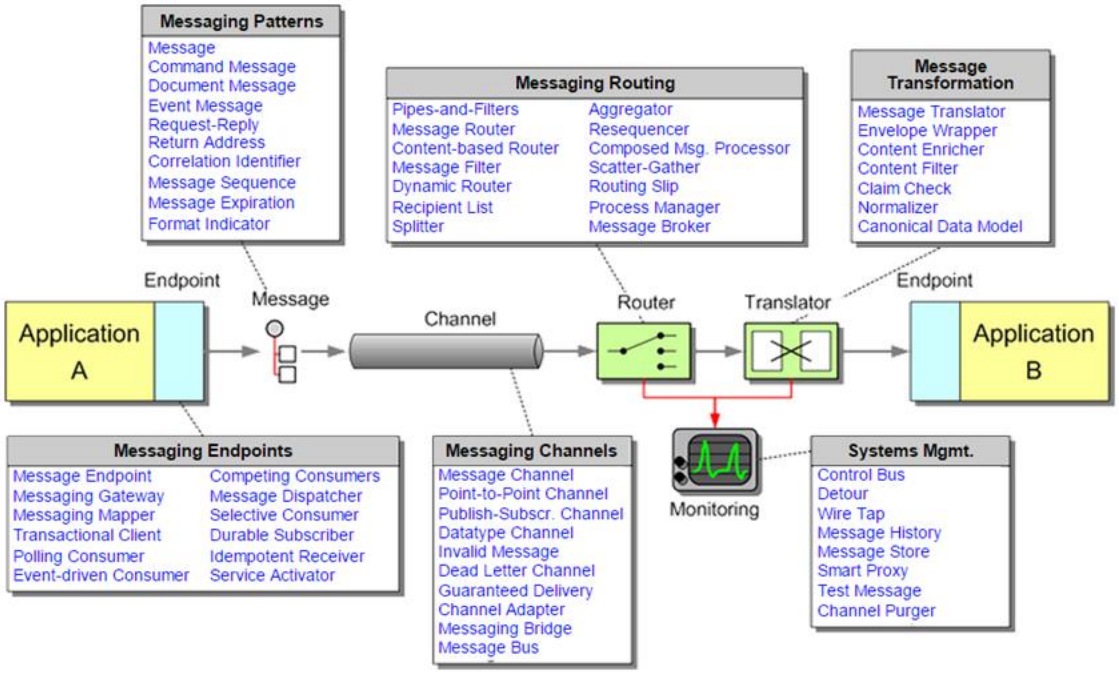

Figura A. 1: Patrones de integración empresariales [30]

Algunos de estos patrones se describen e ilustran a continuación:

- ***Message Channel*:** Conecta dos aplicaciones mediante un sistema de mensajería (Figura A. 2).

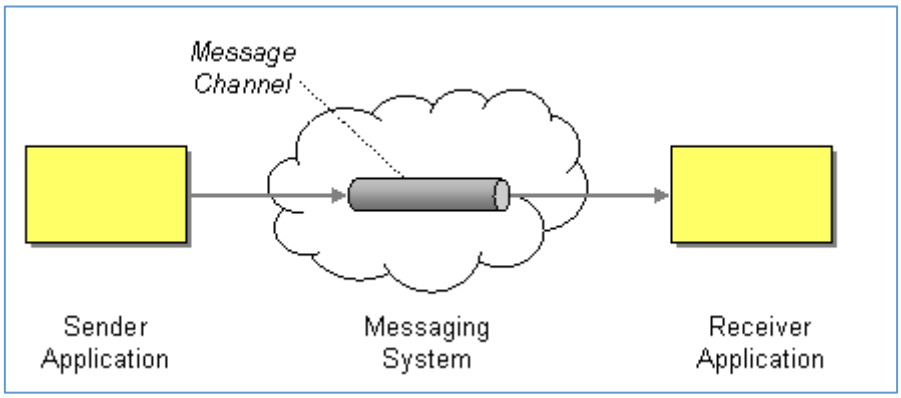

Figura A. 2: Message channel [30]

- ***Routing Slip*:** Adjunta una hoja de ruta a cada mensaje, especificando la secuencia de pasos de procesamiento (Figura A. 3).

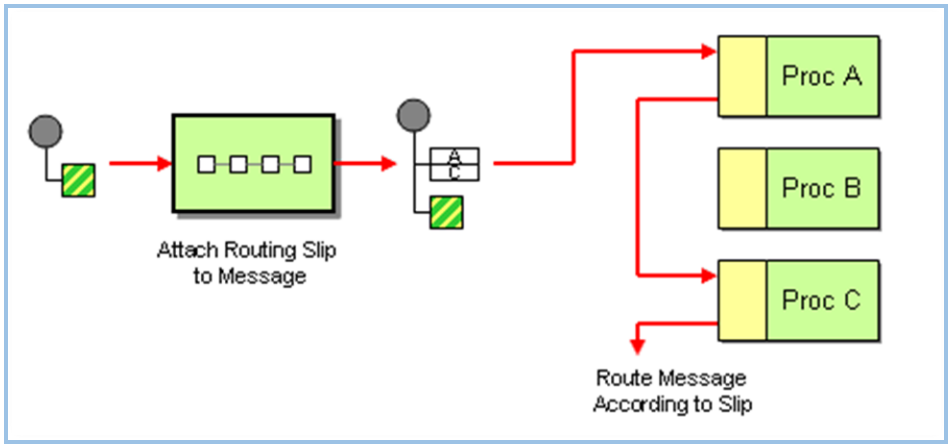

Figura A. 3: Routing Slip [30]

- ***Content-Based Router*:** Examina el contenido de un mensaje para distribuirlo por diferentes canales en función de los datos que componen dicho mensaje (Figura A. 4).

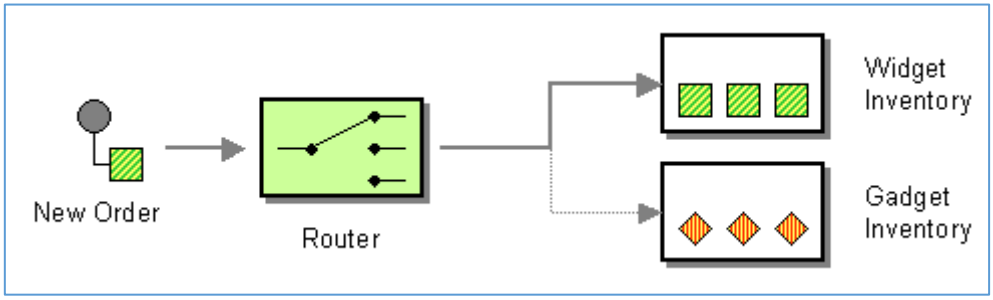

Figura A. 4: Content-based router [30]

- ***Message Translator*:** Transformación de un mensaje en otro, para que pueda ser usado en un contexto diferente (Figura A. 5).

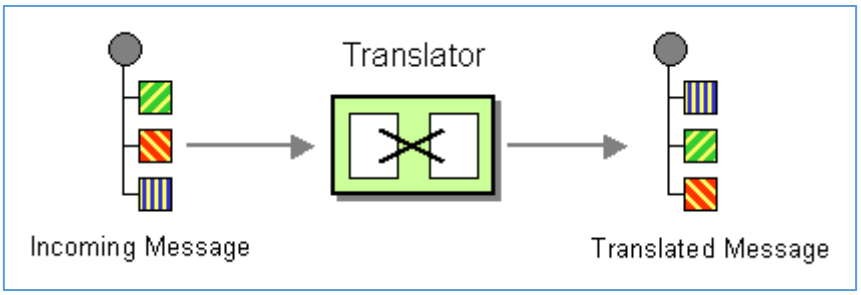

Figura A. 5: Message translator [30]

- ***Content Enricher***: Facilita la comunicación con otro sistema, en el caso de que el mensaje original no tenga disponible todos los datos necesarios (Figura A. 6).

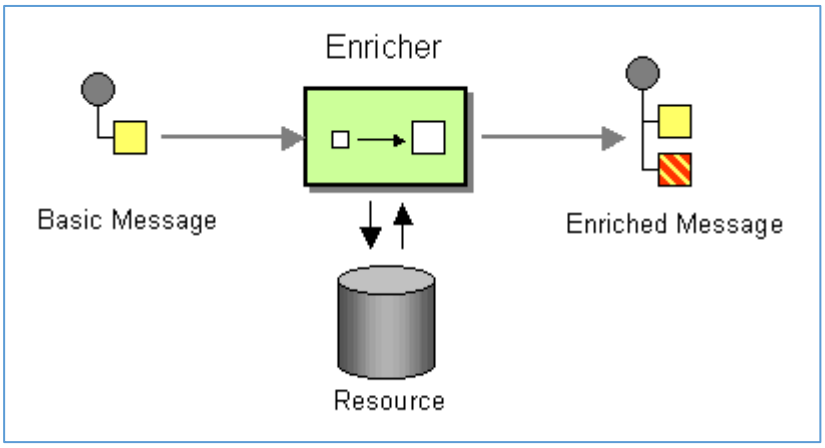

Figura A. 6: Content enricher [30]

B. Arquitectura de referencia

En esta sección se presenta la propuesta de arquitectura de referencia para Plataformas de Integración basadas en microservicios propuesta por Andrés Nebel [2]. La arquitectura se describe usando el modelo de vistas 4+1 de Kruchten [31].

Vista de Casos de Uso

En la Figura B. 1 se presentan los casos de uso más relevantes de la arquitectura de referencia, así como los actores que los ejecutan.

Los administradores son los encargados de crear las soluciones de integración, agregando y vinculando componentes de integración. Los componentes son de dos tipos: Disparador o Acción y deben ser configurados al momento de incluirlos en la SI.

El caso de Ejecución de SI, comienza cuando un componente disparador activa la SI, invocando así al próximo componente para ejecutar la siguiente acción. En este caso, los actores pueden ser tres: un usuario, un sistema externo o la propia PI.

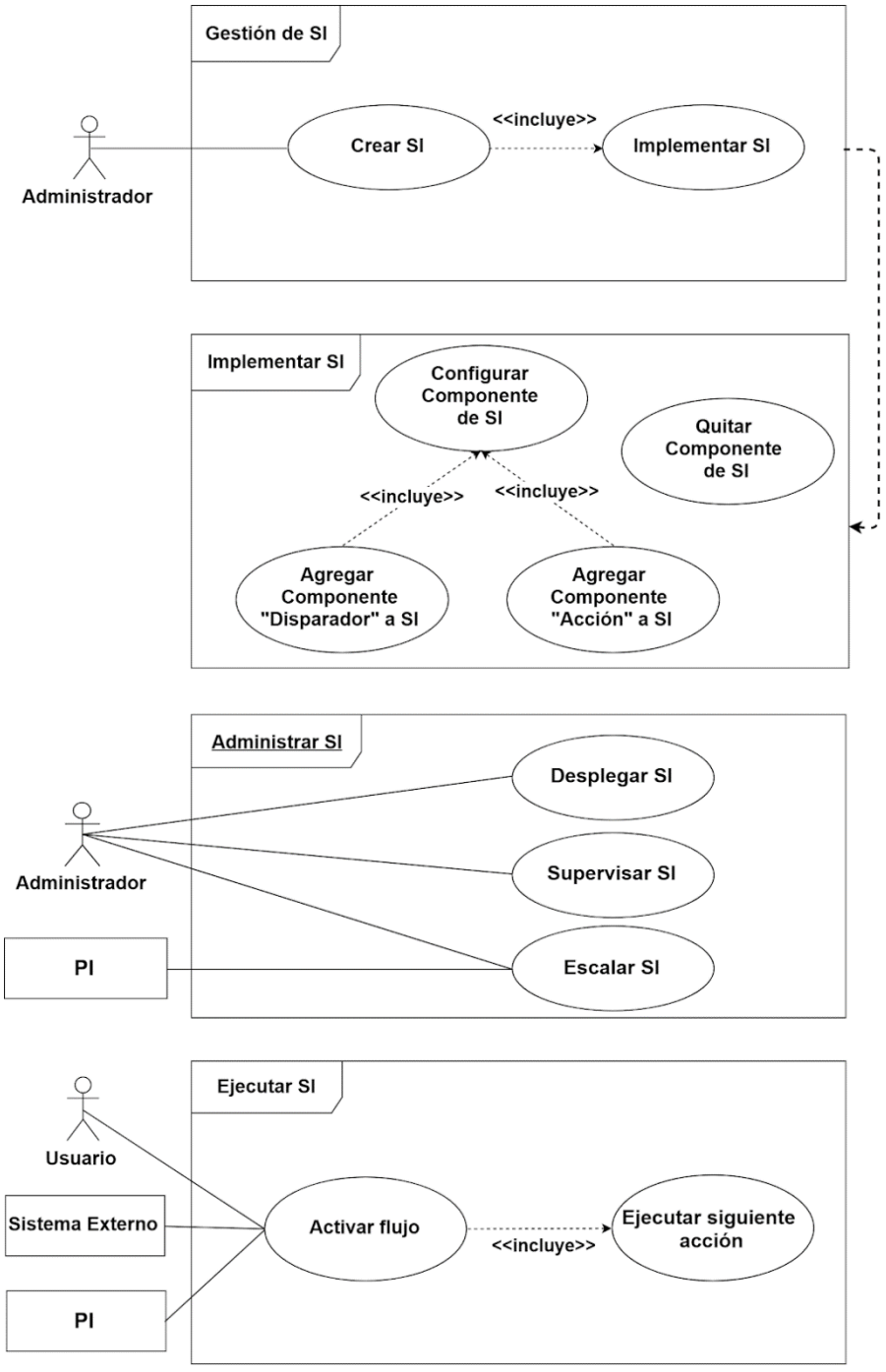

Figura B. 1: Vista de casos de uso relevantes [2]

Vista Lógica

La siguiente vista se construye a partir de los casos de uso descritos en la parte anterior y el desglose se basa en el principio de responsabilidad única y los conceptos de separación de entidades propuestos por Richardson en [17].

En la Figura B. 2 se muestra un diagrama de bajo nivel de las entidades y su relacionamiento cuando el tipo de coordinación es coreografía.

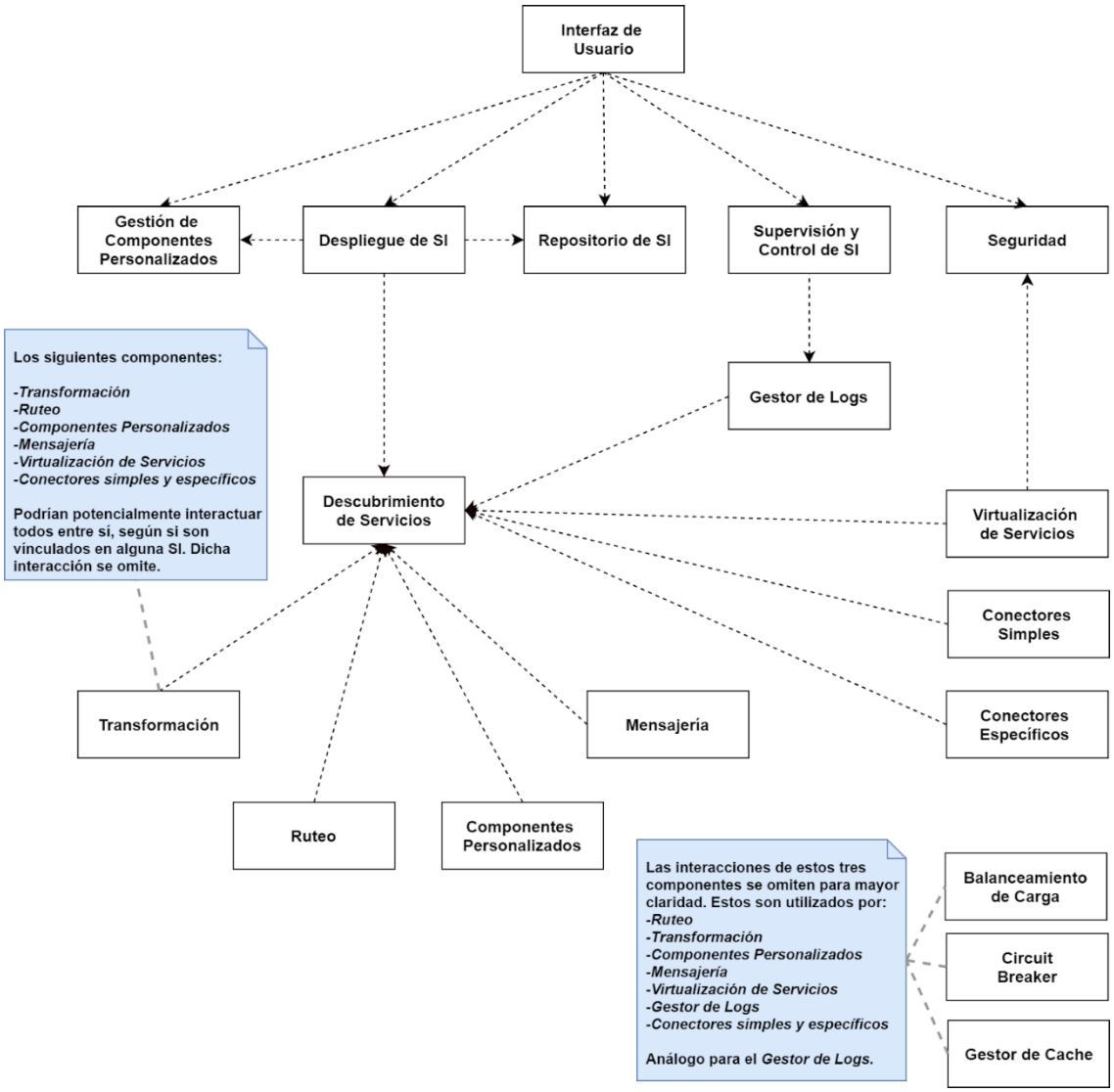

Figura B. 2: Vista lógica de bajo nivel [2]

La entidad *Despliegue de SI* es la encargada de instanciar y desplegar las SI desde el *Repositorio de SI*. En el caso que el tipo de coordinación sea coreografía, la entidad *Despliegue de SI* proporciona a cada componente la información del próximo componente a invocar. En el caso de orquestación, la entidad *Despliegue de SI* transmite la información a la entidad *Orquestador*, esta interacción se puede observar en la Figura B. 3.

Las entidades del tipo Conector son las encargadas de resolver la conectividad con los sistemas a integrar. Existen dos tipos, simples y específicos. Los simples permiten la conectividad mediante protocolos estándares, como pueden ser HTTP o SMTP. Los específicos utilizan protocolos no estándares o que requieren una interacción especial, como el protocolo SAP *Remote Function Call*.

Otra entidad importante es el Gestor de Logs, que se encarga de recolectar e indizar la información de ejecución de los componentes. De esta forma es posible supervisar la ejecución de las SI.

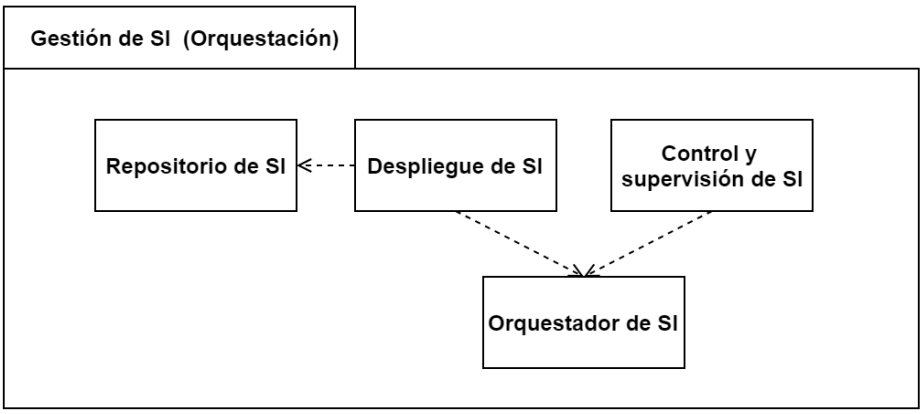

Figura B. 3: Desglose del subsistema Gestión de SI en orquestación [2]

Vista de Desarrollo

La Figura B. 4 presenta la vista de desarrollo, que describe la modularización en subsistemas, servicios y componentes, y detalla la organización estática en módulos del sistema.

El diagrama consta de dos partes importantes. El subsistema *PI-Módulo Central,* es el encargado de realizar todas las actividades de gestión de la PI (gestión, supervisión y despliegue de SI). Este subsistema se diseña bajo una arquitectura monolítica, ya que, por encargarse de casos de uso administrativos, es usado de forma poco frecuente, demandando pocos recursos en comparación con los componentes que dan soporte a las SI.
La segunda parte de la vista se compone de un conjunto de servicios y microservicios, encargados de la ejecución y supervisión de las SI.

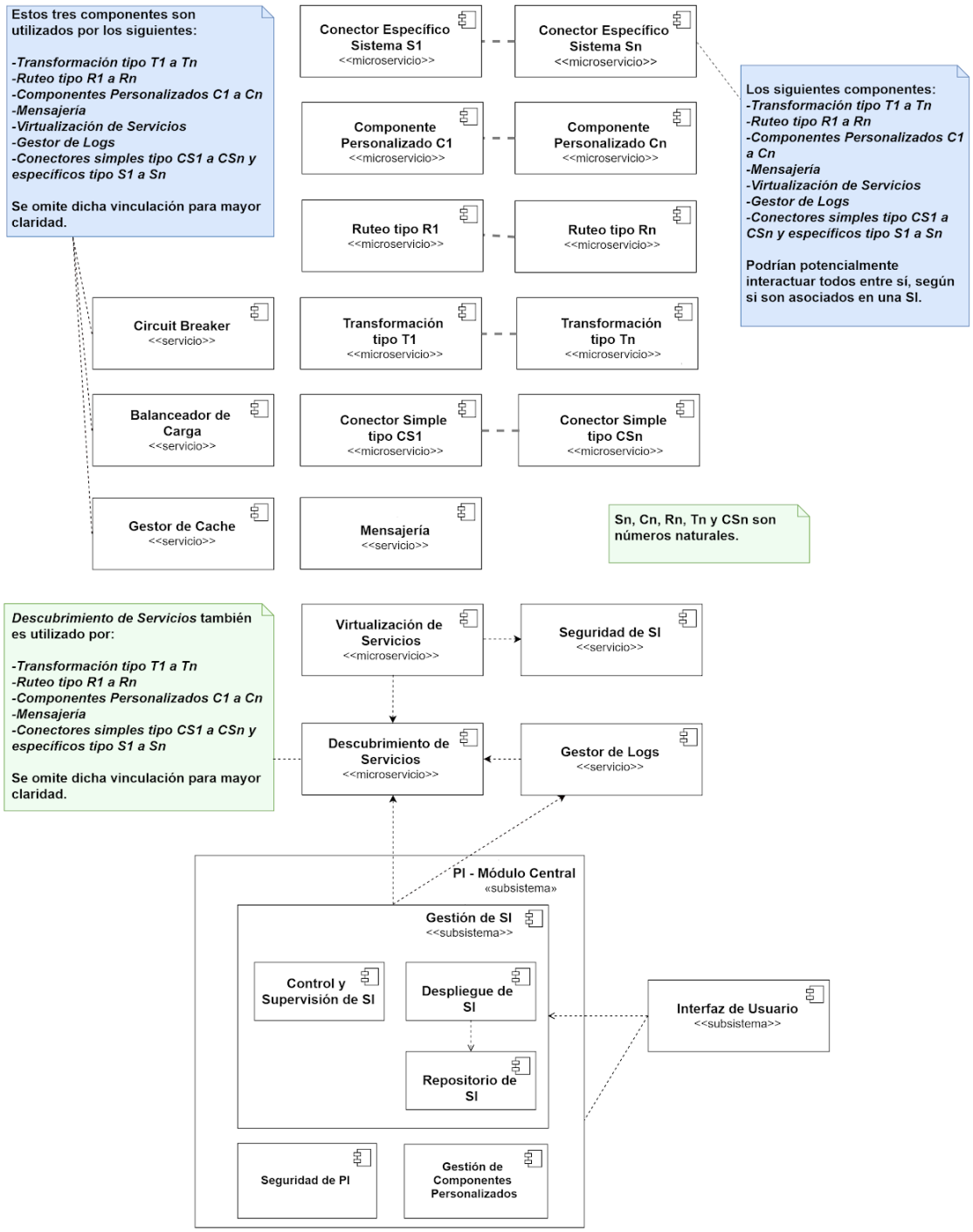

Figura B. 4: Vista de desarrollo [2]

Vista de Procesos

En esta vista solo se analiza la forma de ejecución de la SI, ya que es el caso más relevante para este proyecto.

En la Figura B. 5 se presenta un diagrama de actividad para la ejecución coreográfica de una SI genérica que integra dos sistemas. Datos de un *Sistema J*

son sometidos a una serie de acciones de integración para luego ser enviados al *Sistema K* mediante un conector específico. Cada componente obtiene de su configuración el próximo componente a invocar y mediante el servicio de *Descubrimiento de Servicios* determina su ubicación.

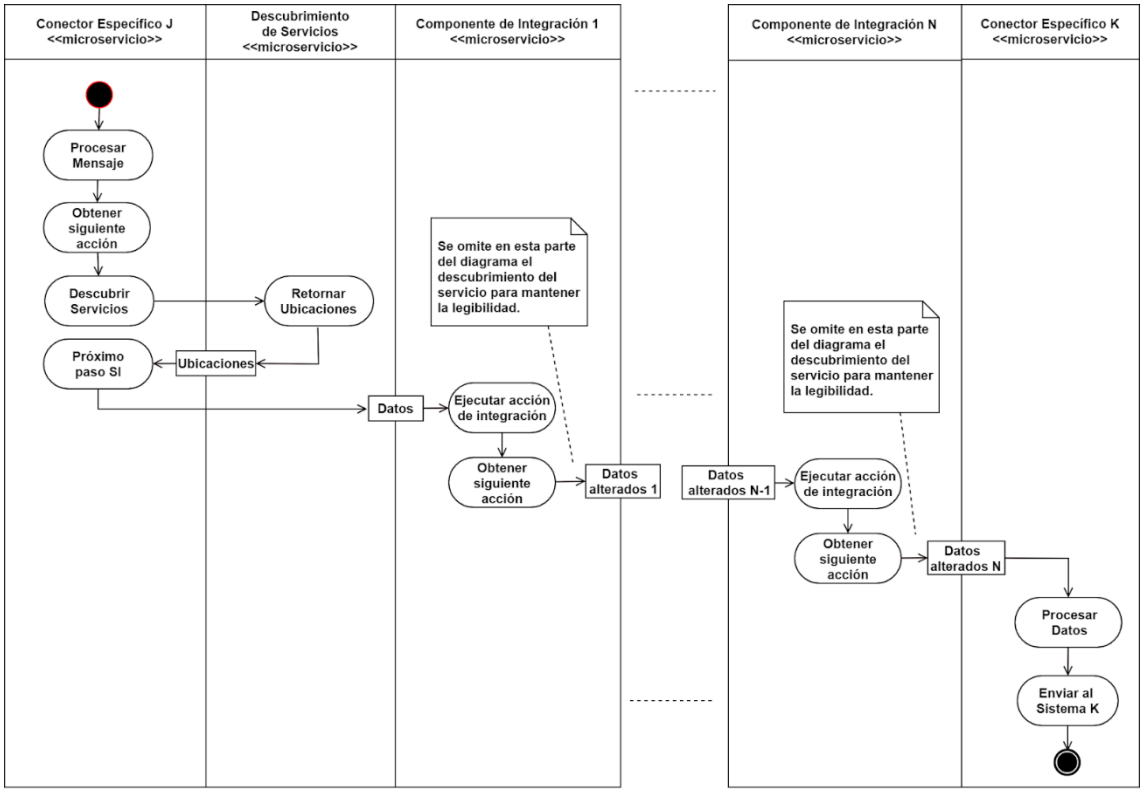

Figura B. 5: Vista de Procesos - Ejecución de SI que integra dos sistemas (one-way) [2]

Vista de Despliegue

Para aprovechar los beneficios de la arquitectura respecto a la escalabilidad, tolerancia a fallas y mantenibilidad, la PI debe desplegarse en una plataforma de contenedores.

En la Figura B. 6 se observa el despliegue de los componentes identificados en las vistas anteriores.

La correspondencia entre los contenedores y los componentes es de uno a uno, siguiendo el patrón de un servicio por contenedor [17]; mientras que el subsistema responsable de la administración de la PI se despliega de forma conjunta sobre un único contenedor.

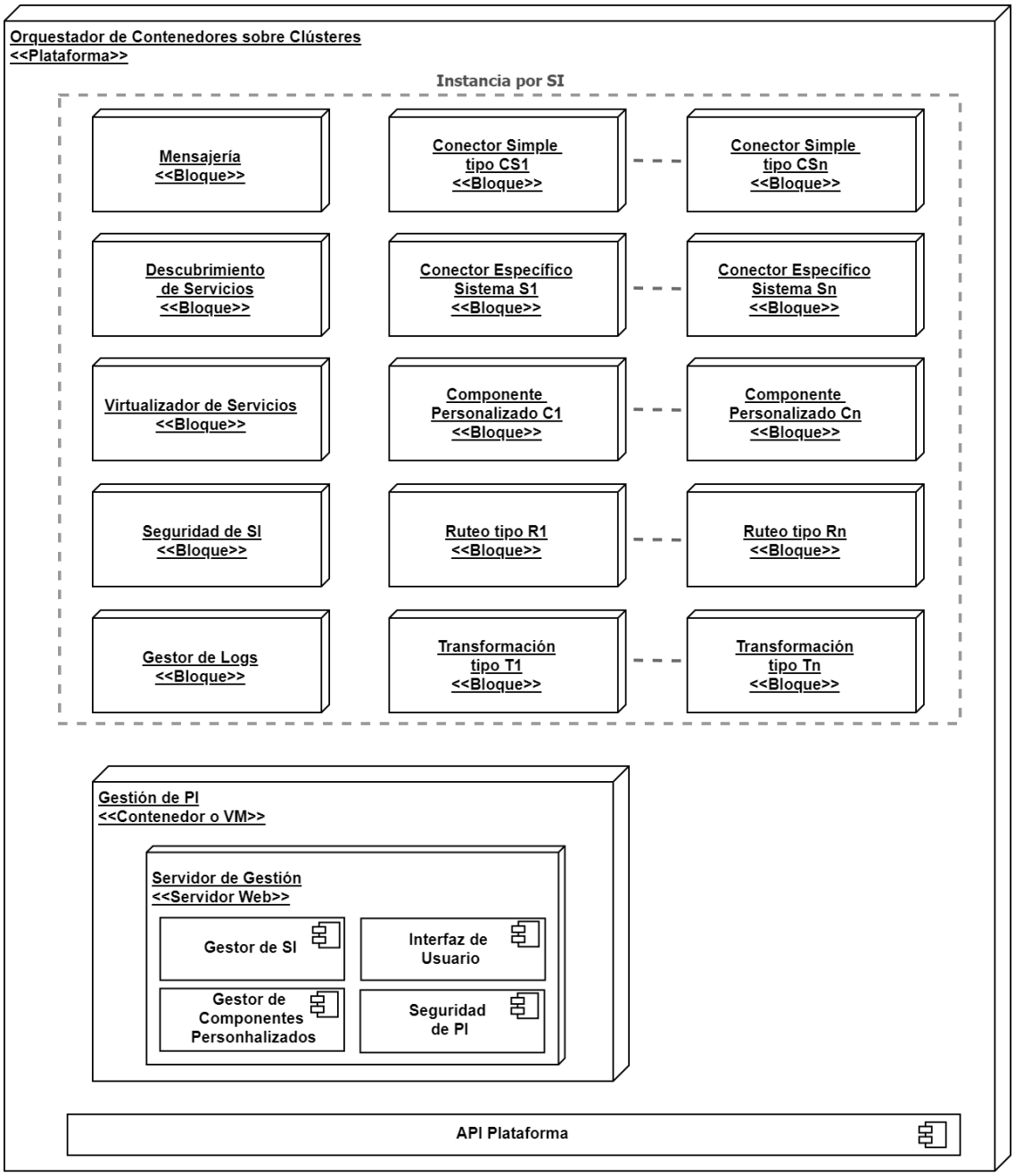

Figura B. 6: Vista de despliegue [2]

C. Configuración de componentes de integración

En esta sección se presenta, en la Tabla C. 1, la configuración del componente de integración Transformación JSON a XML utilizado en el caso de estudio presentado en la Sección 5.3. El objetivo es presentar como se componen los archivos de configuraciones de los microservicios que representan los componentes en la PI.

```
spring:
  rabbitmq:
    host: broker
    port: 5672
    username: guest
    password: guest
  cloud:
    stream:
      bindings:
        transformacionSubscribableChannel:
          destination: conectorEntradaMessages
          group: transformacionjsonxmlQueue
          consumer:
            maxAttempts: 1
        transformacionMessagesChannel:
          destination: transformacionjsonxmlMessages
          consumer:
            maxAttempts: 1
        transformacionMessagesChannelErrores:
          destination: replayMessages
          consumer:
            maxAttempts: 1

logging:
  config: ${configserver.discovered.uri}/logback/default/master/logback-spring-
docker.xml
```

Tabla C. 1: archivo de configuración de microservicio Transformación JSON a XML

A continuación, se detallan los parámetros que contiene al archivo de configuración del componente presentado en la Tabla C. 1.

Los parámetros bajo el nombre *spring.rabbitmq* hacen referencia a información del broker de mensajería de RabbitMQ.

Las propiedades presentes con nombre *spring.cloud.stream.bindings* refieren a las colas y *echanges* que utilizará el componente en soluciones que ejecuten con coreografía.

El componente del ejemplo posee una cola, la cual está asociada el canal con nombre *transformacionSubscribableChannel*. El mismo, bajo la propiedad *destination*, tiene el nombre del *exchange* que enviará los mensajes (*conectorEntradaMessage*s). Por otro lado, bajo la propiedad *group*, tiene el nombre de la cola que recibirá los mensajes del *exchange* anterior (transformacionjsonxmlQueue).

El componente posee dos *exchanges (transformacionjsonxmlMessages* y *replayMessages)*, los cuales están asociados a los canales *transformacionMessagesChannel* y *transformacionMessagesChannelErrores* respectivamente.

En Tabla C. 2 se presenta la configuración del componente orquestador utilizado en el caso de estudio presentado en la Sección 5.3.

```
logging:
  config:${configserver.discovered.uri}/logback/default/master/logback-spring-docker.xml
solucion:
  s2:
    itinerario: transformacionjsonxml,enriquecedor,conectorSalida,transformacionxmljson
```

Tabla C. 2: archivo de configuración de microservicio Orquestador

La propiedad principal de este componente es *solución.s2.itinerario*, donde se almacena el itinerario de los componentes de integración, con el orden en que el orquestador debe invocarlos en la solución de integración identificada como s2. En el itinerario se encuentran los nombres con los que son registrados los microservicios en Eureka.

D. Despliegue de plataforma y solución de integración

Como se detalla en la Sección 4.1, la plataforma cuenta con dos clases de componentes, los que pertenecen a las soluciones de integración y los que brindan servicios a los componentes de las soluciones. Por este motivo es que el despliegue se realiza en dos fases: en primer lugar, se despliegan todos los componentes que brindan servicios dentro de la plataforma. Una vez iniciados estos, se despliegan los componentes relacionados a las soluciones de integración.

Para crear contenedores sobre Docker es necesario especificar una imagen de base. Dicha imagen debe tener el mínimo software requerido por el contenedor, por ejemplo, para desplegar una aplicación Java, es necesario crear una imagen con el software base (por lo general alguna distribución Linux), instalar Java y copiar la aplicación. Luego solo debe iniciarse el contenedor especificando la ejecución de la aplicación.

```
FROM alpine:latest
RUN apk add --no-cache openjdk8
RUN apk add --no-cache bash
COPY wait-for.sh /opt/lib/wait-for-it.sh
RUN chmod +x /opt/lib/wait-for-it.sh
```

Figura D. 1: Archivo de configuración base para los microservicios

Para facilitar la creación de las imágenes, se pueden especificar archivos de configuración con los pasos a seguir. Luego solo deben crearse los contenedores a partir de esos archivos. Se creó un archivo de configuración con el software de base que debe contener cada microservicio implementado. En la Figura D. 1 se puede ver el archivo de configuración base. En el mismo se instala una versión mínima de Alpine Linux, se instala la JDK 8, se instala bash y se copia un script utilizado para sincronizar el inicio de los servicios. Para generar la imagen base se ejecuta el siguiente comando:

```
docker build --file=alpine_base --tag=alpine-jdk:base --rm=true .
```

Docker permite crear redes donde los contenedores puedan comunicarse, por lo que se decide que todos los servicios de la plataforma estén en una red común y que los componentes pertenecientes a una solución de integración estén en su

propia red, aislados del resto de las soluciones, pero permitiendo la comunicación a la red de servicios.

Despliegue fase 1 – servicios de la plataforma

Los componentes que ofrecen algún servicio a las soluciones de integración son el broker de mensajería, el gestor de logs, el servicio de descubrimiento y el servidor de configuración. A continuación, se detalla cómo se despliegan estos componentes.

El primer paso es crear la red que contendrá a estos servicios. El comando para hacerlo es:

```
docker network create -d bridge plataforma_comunes
```

Para crear el contenedor de RabbitMQ se usa una imagen disponible en el repositorio Docker Hub[62]. Los comandos para crear y levantar el contenedor son los siguientes:

- Creación de imagen:
  `docker pull rabbitmq:3.7.9-management-alpine`
- Crear y ejecutar contenedor a partir de la imagen anterior:
  `docker run -d --name broker -p 15672:15672 rabbitmq:3.7.9-management-alpine`
- Agregar contenedor a red de servicios:
  `docker network connect plataforma_comunes broker`

Para crear el contenedor de Graylog, se siguieron las notas de instalación en la documentación oficial[63]. Allí sugiere el uso de la herramienta *docker-compose* (permite coordinar la ejecución de múltiples contenedores) y provee un archivo de configuración para iniciar toda la instalación. En la Figura D. 2 se puede ver el archivo de configuración y allí se especifica la creación de 3 contenedores, uno para la base de datos MongDB[64], otro para Elasticsearch[65] y por último uno exclusivo para la aplicación Graylog.

---

[62] https://hub.docker.com/_/rabbitmq

[63] https://docs.graylog.org/en/2.5/pages/installation/docker.html

[64] https://www.mongodb.com/

```
version: '2.2'
services:
  mongo:
    container_name: mongo
    image: mongo:3
    networks:
      - plataforma_comunes
  elasticsearch:
    container_name: elasticsearch
    image: docker.elastic.co/elasticsearch/elasticsearch:6.5.4
    environment:
      - http.host=0.0.0.0
      - xpack.security.enabled=false
    ulimits:
      memlock:
        soft: -1
        hard: -1
    mem_limit: 1g
    memswap_limit: 1g
    networks:
      - plataforma_comunes
  graylog:
    container_name: graylog
    image: graylog/graylog:2.5
    environment:
      # CHANGE ME (must be at least 16 characters)!
      - GRAYLOG_PASSWORD_SECRET=somepasswordpepper
      # Password: admin
      -
GRAYLOG_ROOT_PASSWORD_SHA2=8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2
ab448a918
      - GRAYLOG_WEB_ENDPOINT_URI=http://127.0.0.1:9100/api
    networks:
      - plataforma_comunes
    depends_on:
      - mongo
      - elasticsearch
    ports:
      # Graylog web interface and REST API
      - 9000:9000
      - 514:514
      - 514:514/udp
      # GELF TCP
      - 12201:12201
      # GELF UDP
      - 12201:12201/udp
networks:
    plataforma_comunes:
        external: true
```

Figura D. 2: Archivo de configuración de Graylog para usar con docker-compose

El comando para crear los contenedores a partir del archivo de configuración *graylog_compose.yml*:

```
docker-compose -p graylog -f graylog_compose.yml up --build -d
```

[65] https://www.elastic.co/es/

Para iniciar y detener el servicio una vez creados los contenedores, se ejecutan los siguientes comandos:

```
docker-compose -p graylog -f graylog_compose.yml start

docker-compose -p graylog -f graylog_compose.yml stop
```

Para generar los contenedores relacionados al servicio de descubrimiento y servidor de configuración se decide utilizar otro archivo de configuración como el usado con Graylog. Esto permite que el inicio de los contenedores se haga en un orden preestablecido, primero el servicio de descubrimiento y luego el servidor de configuración. También se configura que esos contenedores pertenezcan a la red de servicios.

El comando para generar los contenedores a partir del archivo de configuración *comunes_compose.yml* es:

```
docker-compose -p pi_comunes -f comunes_compose.yml up --build -d
```

Para iniciar y detener el servicio una vez creados los contenedores:

```
docker-compose -p pi_comunes -f comunes_compose.yml start

docker-compose -p pi_comunes -f comunes_compose.yml stop
```

Despliegue fase 2 – soluciones de integración

Para desplegar las soluciones de integración, se vuelve a utilizar un archivo de configuración que será usado con *docker-compose*. La estructura es similar a la vista en Figura D. 2, donde se define un contenedor por cada componente de la solución. La mayor diferencia se produce en la definición de las redes. Todos los contenedores tienen acceso a dos redes: la red de servicios de la plataforma y la red de la solución de integración. Esta última se define en el mismo archivo de configuración, de modo que cuando se baje una solución se elimine esa red.

```
version: '2.2'
services:
    orquestador:
        container_name: orquestador
        build:
            context: .
            dockerfile: orquestador
        image: orquestador:latest
        environment:
            SPRING_APPLICATION_JSON: '{
            "spring.profiles.active":"docker",
            "spring.cloud.config.discovery.service-id": "configserver"
            }'
        command: ["-c","/opt/lib/wait-for-it.sh configserver:8888 && sleep 2 &&
/usr/bin/java  -Xmx225m -Xms110m -jar /opt/lib/orquestador.jar"]
        entrypoint: ["/bin/sh"]
        mem_limit: 450m
        memswap_limit: 450m
        expose:
            - "8100"
        ports:
            - "8100:8100"
        networks:
            - plataforma_comunes
            - network_s1
…
…
…
networks:
    plataforma_comunes:
        external: true
    network_s1:
        name: network_s1
        driver: bridge
```

Figura D. 3 Archivo de configuración de una solución de integración para usar con docker-compose[66]

En la Figura D. 3 se puede ver la definición de los contenedores de una solución de integración, en particular la definición del contenedor orquestador y las redes disponibles para la solución. En el parámetro *command* es donde se lanza la ejecución de la aplicación previo chequeo de la disponibilidad del servicio de configuración.

El siguiente comando se ejecuta para generar los contenedores a partir del archivo de configuración *s1_compose.yml*:

```
docker-compose -p solucion_s1 -f s1_compose.yml up --build -d
```

Para iniciar y detener el servicio una vez creados los contenedores, se tienen los comandos:

[66] Se omite la definición de algunos contenedores para hacer la imagen más legible

```
docker-compose -p solucion_s1 -f s1_compose.yml start

docker-compose -p solucion_s1 -f s1_compose.yml stop
```

Todos los archivos de docker utilizados en el proyecto se encuentran en el repositorio Gitlab[67] de Facultad de Ingeniería.

[67] https://gitlab.fing.edu.uy/open-lins/microservices-platform

E. Pruebas funcionales

En esta sección se presentan en la Tabla E. 1 las pruebas y los resultados de estas, ejecutadas sobre la PI, específicamente sobre la Solución de Integración creada para el Caso de Estudio presentado en el capítulo 5.3.

Caso de Prueba	**Tipo Solución**	**Campos Entrada**			**Resultado Éxito**	**Resultado Error**
1: Razon Social Arcos Dorados	Coreografía	**Header 1 (content-type)**	text/plain	**Header**	200 OK	200 OK
		header 2 (idsol)	1			
		header 3 (idMensaje)	101			
		Body	{ "proyectograd o": { "razonSocial": "Arcos Dorados" } }	**Parte del Body**	..... "ns2:codigoRespuesta": 0, "ns2:Empresa": { "ns2:localidad": "Montevideo", "ns2:nombre": "McDonald\u2019s", "ns2:rut": 12323332, "ns2:direccion": "18 de Julio y Ejido", "ns2:razonSocial": "Arcos Dorados" }	Error de procesamiento! Consulte al administrador de la plataforma.
2: Razon Social PepsiCo	Orquestación	**Header 1 (content-type)**	text/plain	**Header**	200 OK	200 OK
		header 2 (idsol)	2			
		header 3 (idMensaje)	102			
		Body	{ "proyectograd o": { "razonSocial": "PepsiCo" } }	**Parte del Body**	..... "ns2:codigoRespuesta": 0, "ns2:Empresa": { "ns2:localidad": "Montevideo", "ns2:nombre": "McDonald\u2019s", "ns2:rut": 12323332, "ns2:direccion": "18 de Julio y Ejido", "ns2:razonSocial": "Arcos Dorados" }	Error de procesamiento! Consulte al administrador de la plataforma.

3: Razon Social Vacia	Orquestación	**Header 1 (content-type)**	text/plain	**Header**	-	200 OK
		header 2 (idsol)	2			
		header 3 (idMensaje)	103			

		Body	{
			"proyectograd o": {
			"razonSocial": ""
			}
			}
				Parte del Body	-	Error de procesamiento! Consulte al administrador de la plataforma.

4: Body Vacio	Coreografía	**Header 1 (content-type)**	text/plain	**Header**	-	406 Not Acceptable
		header 2 (idsol)	1			
		header 3 (idMensaje)	104			
		Body		**Parte del Body**	-	
5: Header idsol Vacio	Orquestación	**Header 1 (content-type)**	text/plain	**Header**	-	406 Not Acceptable
		header 2 (idsol)				
		header 3 (idMensaje)	105			

		Body	{
			"proyectograd o": {
			"razonSocial": "PepsiCo"
			}
			}
				Parte del Body	-	

6: Header content-type Vacio	Orquestación	**Header 1 (content-type)**		**Header**	-	415 Unsupported Media Type Not Acceptable
		header 2 (idsol)	2			
		header 3 (idMensaje)	106			

		Body	{
			"proyectograd o":
			{
			"razonSocial": "PepsiCo"
			}
			}
				Parte del Body	-	

Tabla E. 1: Casos de pruebas funcionales

Los casos de prueba 1 y 2 muestran los posibles resultados para las razones sociales consultadas en cada caso: resultado exitoso con la información de la empresa y resultado de error. Se diferencian por el *body* del mensaje, ya que para ambos casos el código de respuesta es el mismo (*200 OK*).

Para el caso de prueba 3 se consulta con el campo razón social vacío en el cuerpo del mensaje JSON. No existe respuesta exitosa para esta casuística, siempre muestra el mensaje de error.

El caso de prueba 4 consulta con el *body* vacío del *request*, lo que implica que la PI siempre responda con error. Para este caso el código de error es *406 Not Acceptable.*

La prueba numero 5 realiza la consulta con el *header idSol* vacío. Este *header* contiene el identificador de la solución que se desea ejecutar por porte del cliente solicitante. Al igual que en la prueba 4, la PI responde con error *406 Not Acceptable.*

Por último, el caso de prueba 6 hace la consulta con el *header content-type* vacío. La respuesta de la PI es de error *415 Unsupported Media Type Not Acceptable*, lo que implica que es obligatorio enviar el campo *content type* al invocar a una Solución de Integración, con el valor *text/plain*.

F. Pruebas de rendimiento

En la Sección 5.3.3 se analizó la primera prueba de rendimiento, correspondiente a 100 usuarios ejecutando 1000 consultas con estilo de ejecución orquestación. En este apéndice se estudiarán el resto de las pruebas definidas.

Continuando en el orden definido, se analiza el caso número dos: 100 usuarios ejecutando 1000 consultas con estilo coreografía.

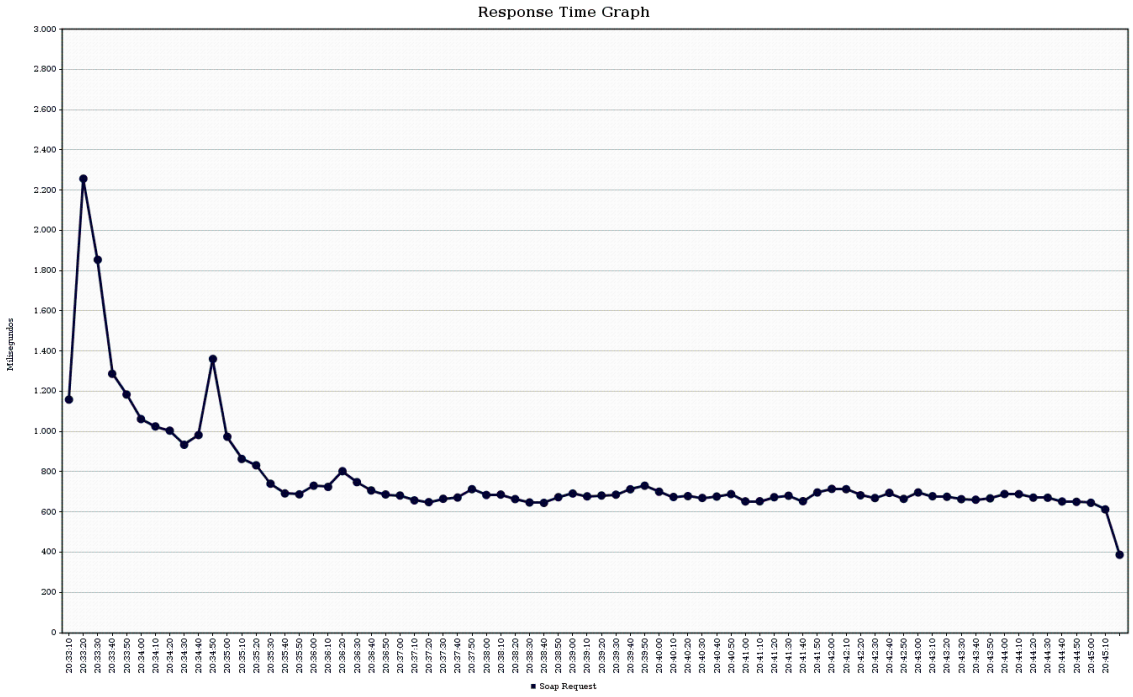

Figura F. 1: Tiempos de respuesta promedio para la prueba dos.

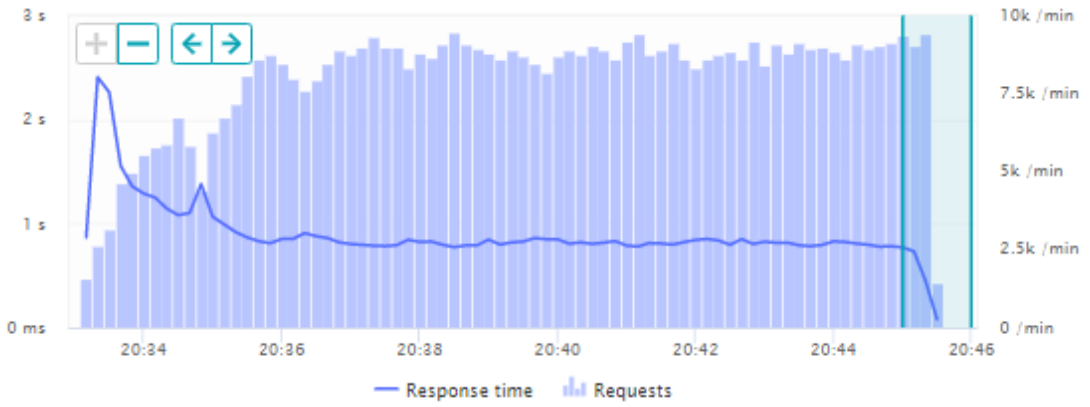

Figura F. 2: Mediana del tiempo de respuesta y cantidad de peticiones a lo largo del tiempo para la prueba dos.

En la Figura F. 1 se aprecia una gráfica con los tiempos promedios de respuesta a lo largo del tiempo y en la Figura F. 2 se observa la cantidad de solicitudes por minuto y la mediana del tiempo de respuesta. Al igual que en el caso de orquestación, al comienzo los tiempos de respuesta se encuentran por encima de 1 segundo y a medida que transcurre la prueba, bajan al orden de los 750 milisegundos. A diferencia del caso de orquestación, hacia el final de la prueba los tiempos de respuesta se mantienen estables. También se puede observar como el tiempo de respuesta y la cantidad de solicitudes atendidas por minuto se mantiene más estable, observación que ya se advirtió al analizar la desviación estándar de los tiempos de respuesta.

Name	Response time median	Failure rate	Requests ▾
RabbitMQ Queue Listener conectorentrada.jar	0 µs	-	4.76k /min
RabbitMQ Queue Listener transformacion.jar	0 µs	-	4.76k /min
AmqpInboundChannelAdapter$Listener conectorentrada.jar	674 µs	0 %	4.76k /min
AmqpInboundChannelAdapter$Listener transformacion.jar	951 µs	0 %	4.76k /min
conectorEntrada-1 conectorentrada.jar	816 ms	0 %	4.76k /min
RabbitMQ Queue Listener conectorsalida.jar	0 µs	-	3.81k /min
RabbitMQ Queue Listener enriquecedor.jar	0 µs	-	3.81k /min
AmqpInboundChannelAdapter$Listener enriquecedor.jar	1.18 ms	0 %	3.81k /min
AmqpInboundChannelAdapter$Listener conectorsalida.jar	6.39 ms	0 %	3.81k /min
AmqpInboundChannelAdapter$Listener transformacion.jar	1.06 ms	0 %	3.11k /min
RabbitMQ Queue Listener transformacion.jar	0 µs	-	3.11k /min
EmpresasEndpoint clientefinalsoap.jar	1.37 ms	0 %	3.11k /min

Figura F.1: Tiempos de respuesta por componente para la prueba dos.

En la Figura F.1 se pueden observar los tiempos de respuesta de los componentes, así como la cantidad de mensajes procesados por las colas en el

broker de mensajería. En los primeros lugares se pueden apreciar las colas y canales del conector de entrada y transformador, ambos con una buena tasa de procesamiento por minuto.

A diferencia del caso de orquestación, el manejo de memoria en la JVM de los componentes tuvo un comportamiento normal, como se aprecia en la Figura F.2. El componente conector de entrada, que había dado problemas en la prueba anterior, no presenta dificultades de memoria en el espacio *Old Gen* y de suspensión de Java.

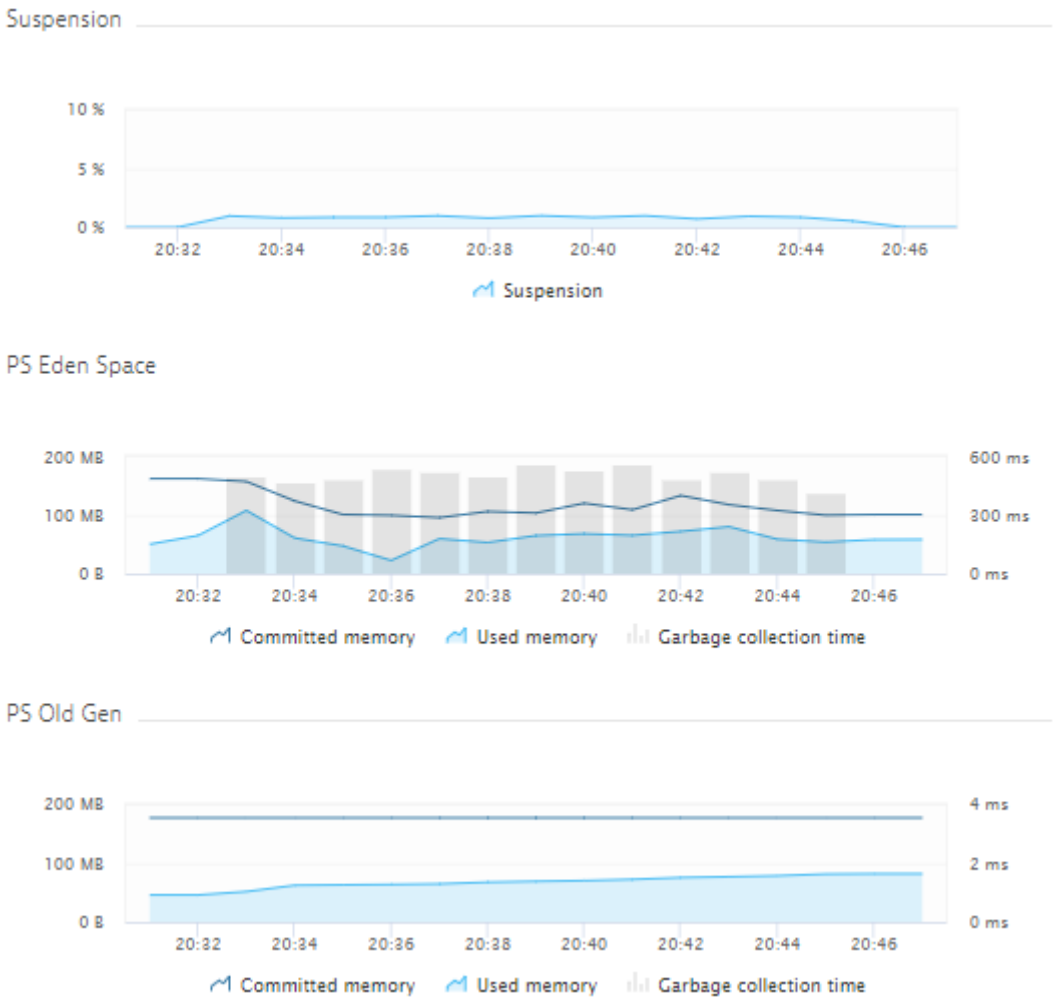

Figura F.2: Memoria de la JVM conector de entrada.

La prueba número tres, corresponde a 100 usuarios ejecutando 1000 consultas con ambos estilos de ejecución.

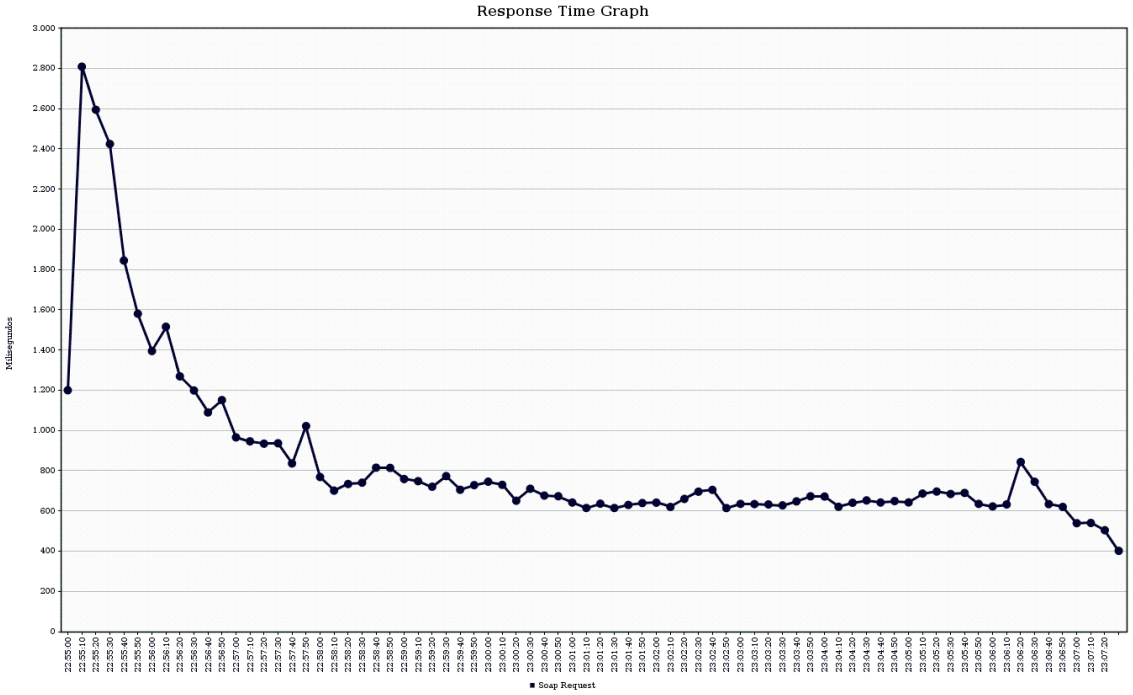

Figura F.3: Tiempos de respuesta promedio para la prueba tres.

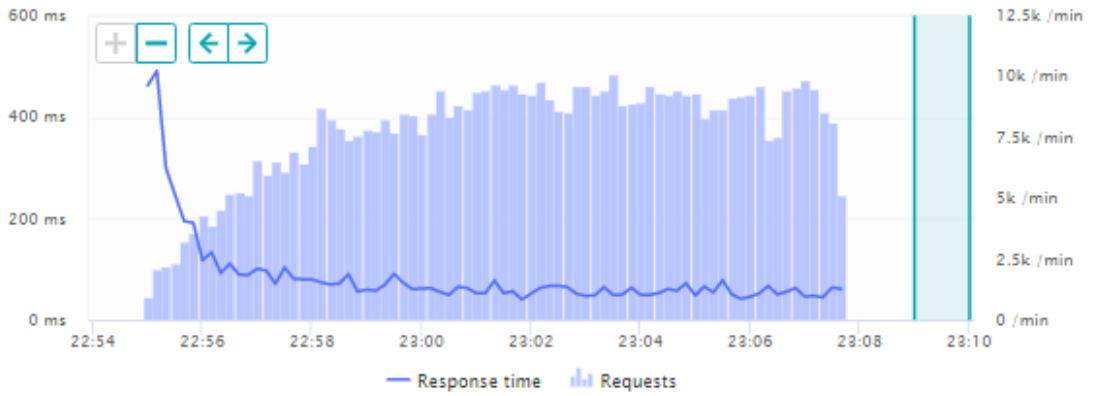

Figura F.4: Mediana del tiempo de respuesta y cantidad de peticiones a lo largo del tiempo para la prueba tres.

En la Figura F.3 se observa un gráfico con los tiempos promedio de respuesta. Al igual que las dos primeras pruebas, los tiempos no varían mucho. En la Figura F.4 se aprecia la mediana de los tiempos de respuesta y la cantidad de solicitudes atendidas a lo largo de la prueba. En este caso, se aprecia que la tasa de solicitudes atendidas es muy parecida a las anteriores pruebas, pero no así la mediana. Iniciada la prueba, la mediana se ubica por debajo de los 200 milisegundos, muy por debajo del promedio de la prueba, que se encuentra en el entorno de los 700 milisegundos. Esto nos lleva a analizar la distribución de las solicitudes en relación con el tiempo de respuesta.

Figura F.5: Distribución de solicitudes por tiempo de respuesta.

En la Figura F.5, se aprecia la distribución de las solicitudes según el tiempo de respuesta. Se observa que de las 100.000 solicitudes que recibió la plataforma, 59.000 se resolvieron en tiempos menores a 182 milisegundos, pero unas 40.000 en tiempos mayores a 1 segundo.

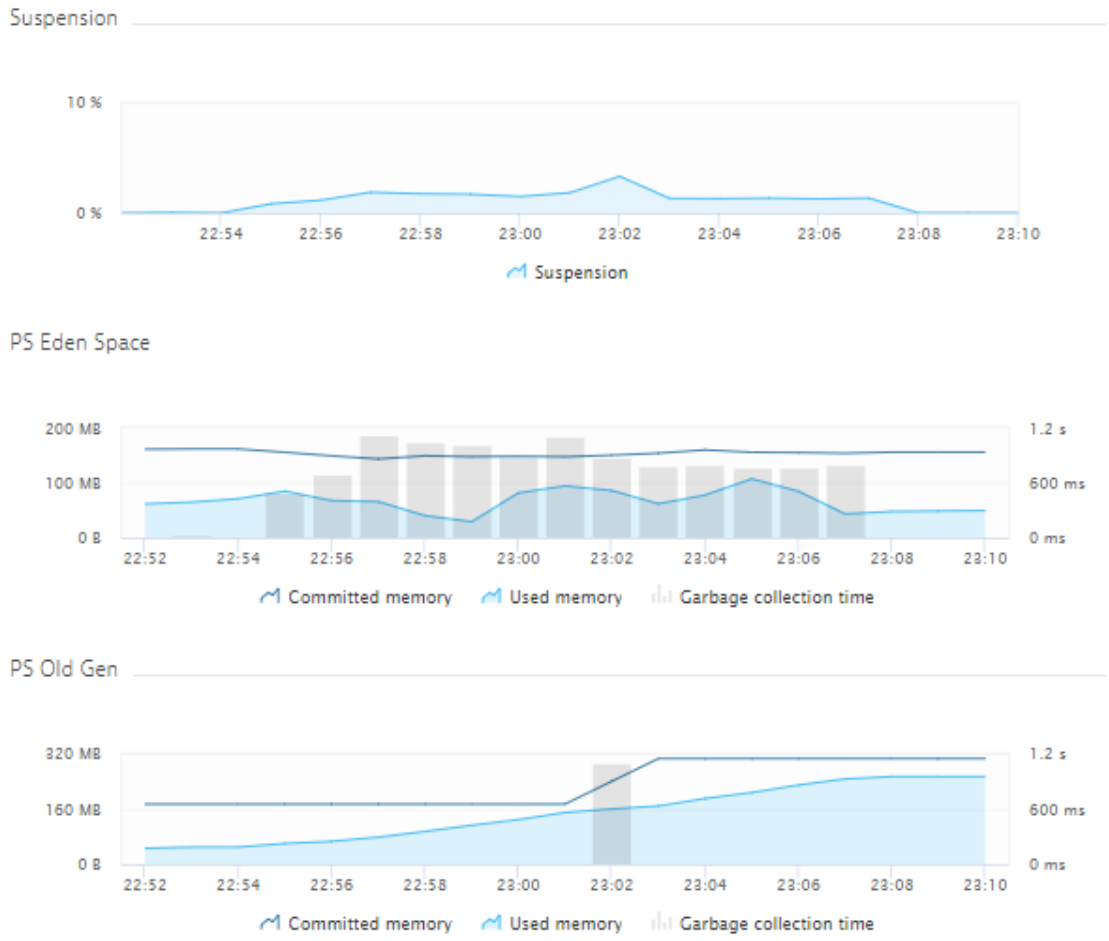

Figura F.6: Memoria de la JVM conector de entrada.

Para encontrar la causa de esta distribución se siguió el mismo camino que en la prueba uno, analizando el conector de entrada. En la Figura F.6 se observa la memoria utilizada por la JVM del conector de entrada durante la prueba. En este caso no se observaron problemas con la memoria. Si bien se encuentra una ejecución del *full garbage collector,* se aprecia que no incide en el funcionamiento del conector, ya que no se generan grandes tiempos de suspensión.

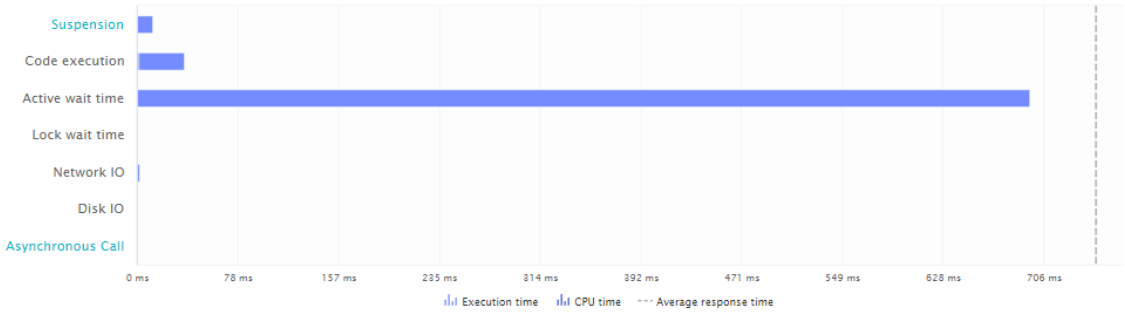

Figura F.7: Composición del tiempo de respuesta.

Se analizó la distribución del tiempo de respuesta dentro del conector. En la Figura F.7 se muestra cómo se compone el tiempo de respuesta de las solicitudes, y la mayor parte del tiempo están en espera. Para detectar donde se produce la espera, se estudiaron los tiempos de respuesta de los puntos de acceso (brindados por una funcionalidad de Dynatrace que permite establecer una relación entre los métodos invocados con el tiempo de espera).

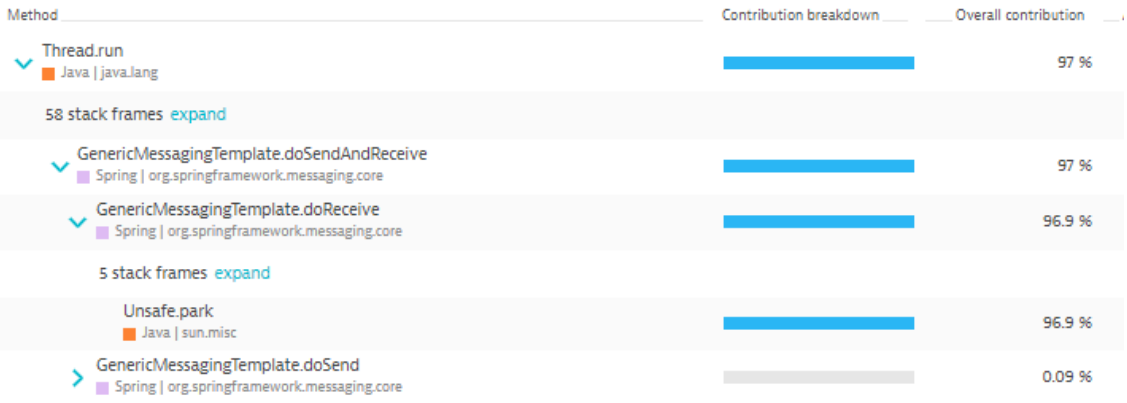

Figura F.8: Tiempos de respuesta de los puntos de acceso.

En la Figura F.8, se destaca que el 96.9% del tiempo de espera es dado por el método *Unsafe.park*. Este método es invocado por librerías internas de Java, por lo general usado en el manejo de los *thread*. Por lo tanto, si el problema es

dado por algún tipo de concurrencia, tal vez sea recomendable escalar el componente y evaluar el comportamiento.

La última prueba de rendimiento corresponde a 300 usuarios ejecutando soluciones de integración con ambos estilos de comunicación. Para este caso no se estableció un límite en la cantidad de consultas realizadas.

Analizando la Tabla 5.1, rápidamente se determina un aumento en el tiempo medio de respuesta en relación con las tres primeras pruebas, sin embargo, el rendimiento por segundo se mantiene en un valor similar al resto. Por lo tanto, se podría afirmar que la cantidad de usuarios y peticiones saturó la plataforma, provocando que los tiempos medios fueran mayores a los 2 segundos.

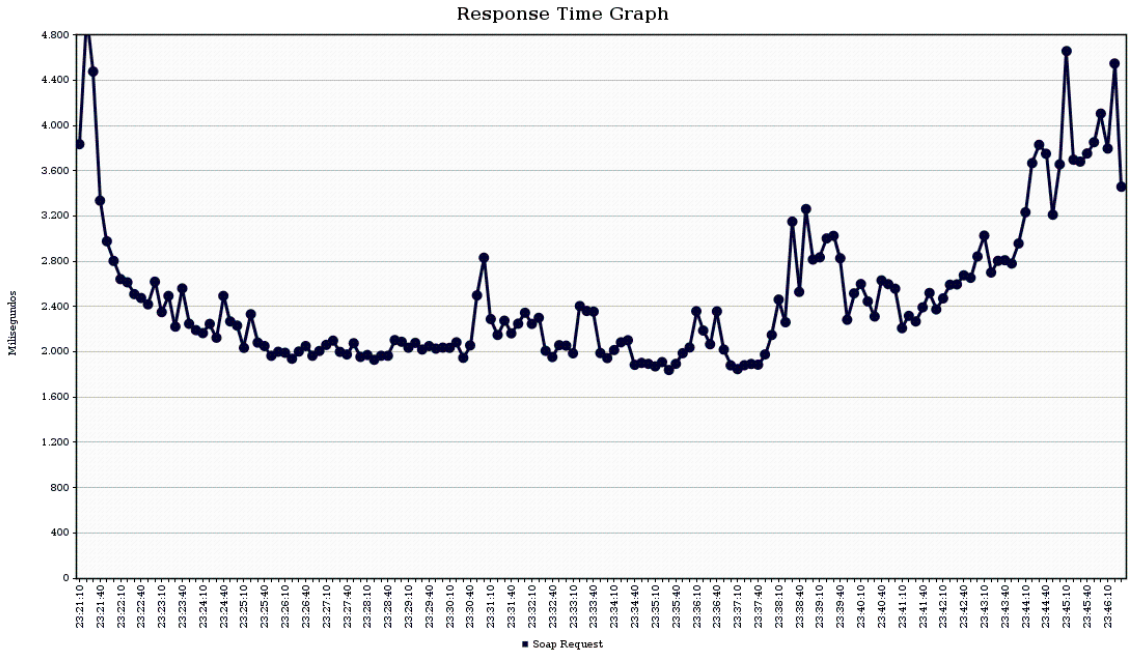

Figura F.9: Tiempos de respuesta promedio para la prueba cuatro.

En la Figura F.9 se observa la evolución del tiempo de respuesta a lo largo de la prueba. Pasados los primeros minutos, el tiempo de respuesta se estabiliza en el entorno de los 2 segundos, pero hacia el final de la prueba los tiempos aumentan drásticamente, derivando en la finalización de esta.

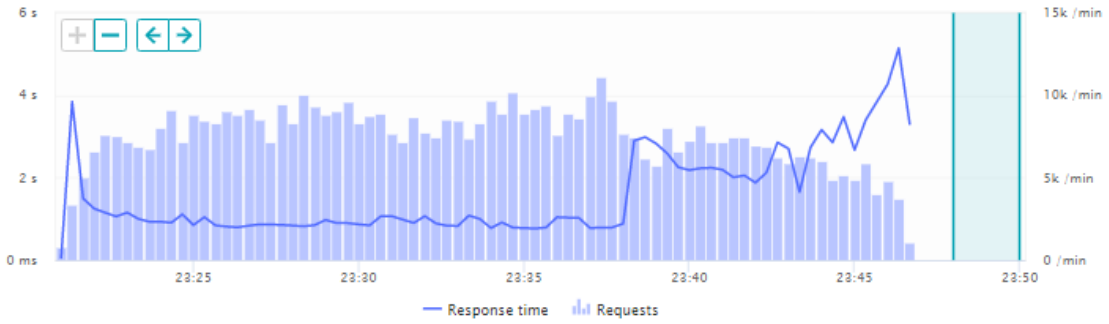

Figura F.10: Mediana del tiempo de respuesta y cantidad de peticiones a lo largo del tiempo para la prueba cuatro.

En la Figura F.10 se observa que la mediana nuevamente se encuentra por debajo del promedio, mientras que la cantidad de solicitudes atendidas por minuto es similar al resto de las pruebas.

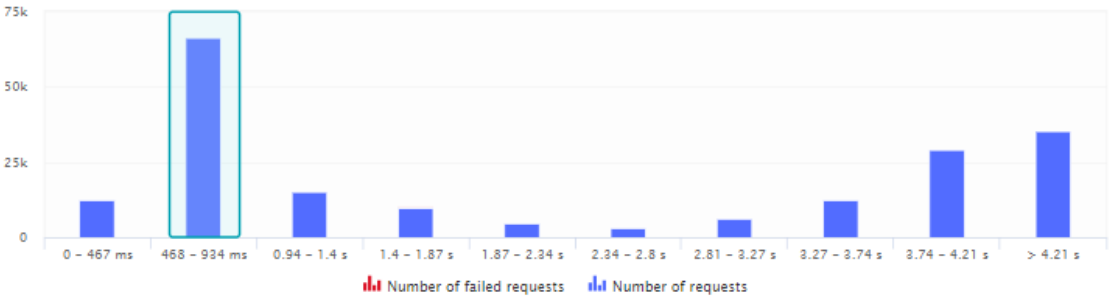

Figura F.11: Distribución de solicitudes por tiempo de respuesta.

En la Figura F.11 se aprecia como se distribuyen las solicitudes según el tiempo de respuesta. Se observa un comportamiento similar al caso anterior, donde la mayor cantidad de solicitudes se encuentran en los extremos de la gráfica. Se tienen 66.000 solicitudes que se resuelven entre 468 y 934 milisegundos y unas 13.000 en menos de 468 ms. En el extremo más lento, se tienen 76.000 peticiones que se resuelven en tiempos mayores a 3.27 segundos (esto se indica por los 3 valores de la derecha de la gráfica).

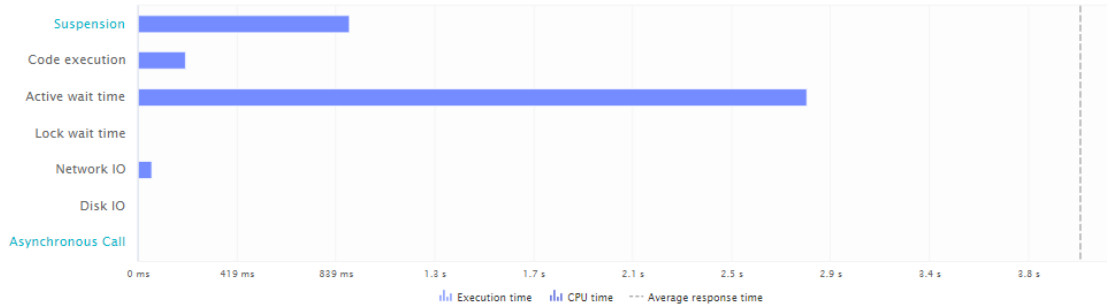

Figura F.12: Composición del tiempo de respuesta.

En la Figura F.12 se observa la composición del tiempo de respuesta en el conector de entrada. Al igual que en la prueba anterior, las esperas son la mayor contribución al tiempo de respuesta de las solicitudes más lentas, en este caso del entorno de 2.8 segundos. Nuevamente el causante de estas demoras se relaciona al método *Unsafe.park*, esto se observa en la Figura F.13.

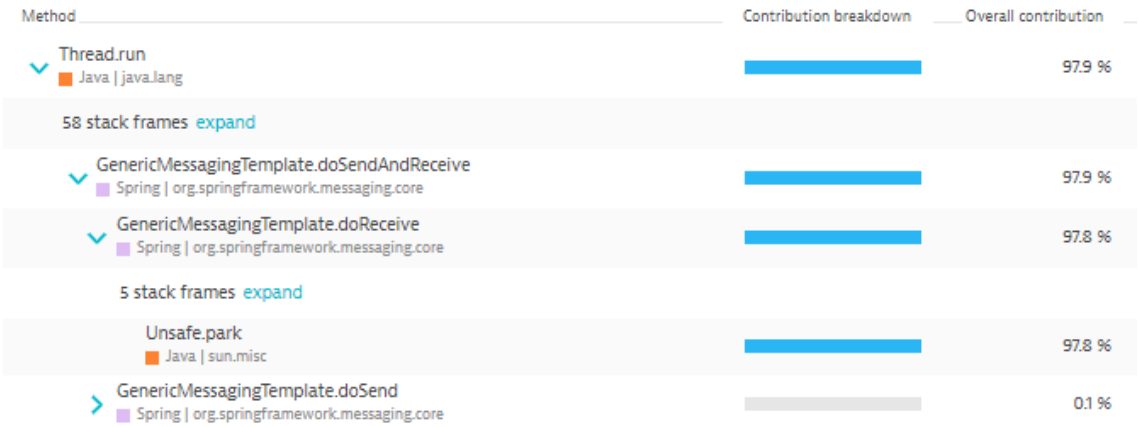

Figura F.13: Tiempos de respuesta de los puntos de acceso.

Con relación al resto de los componentes de la solución, se puede observar en la Figura F.14 como se mantienen las tasas de respuesta, situándose en valores superiores a las 3.000 solicitudes por minuto.

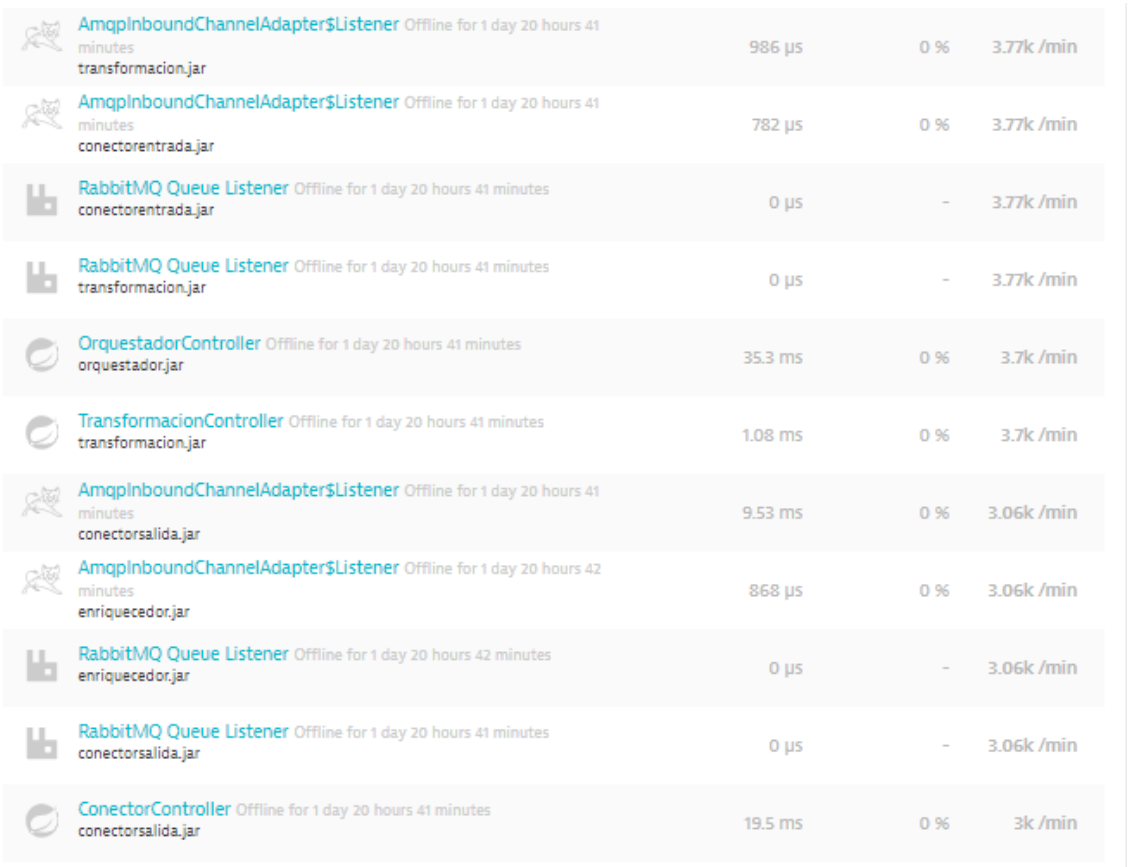

Figura F.14: Tiempos de respuesta por componente

Con estas pruebas se logró comprobar los límites de la plataforma. Se encontraron problemas en el conector de entrada cuando recibe muchas peticiones por segundo. Al ser el punto de entrada a la plataforma, es uno de los componentes que mayor trabajo realiza y resulta fundamental monitorear su carga de trabajo.

En todas las pruebas se obtuvieron valores similares para el rendimiento, entre 125 y 130 peticiones por segundo. Esta medida establece un límite en el rendimiento de cada componente, por lo tanto, es la medida a partir de la cual se debe comenzar a escalar los componentes.