Instituto de Computación – Facultad de Ingeniería

Universidad de la República

Montevideo, Uruguay

Aplicación de Microservicios sobre una arquitectura SOA con restricciones de calidad de servicio

Informe de Proyecto de Grado

Mikaela Pisani

Romina Miraballes

Nicolás García

Tutor: Guzmán Llambías

Abril 2016

Resumen

La arquitectura Microservicios se ha vuelto popular en los últimos años, donde grandes empresas se han migrado a este tipo de arquitectura como Netflix, eBay y Amazon. Esto se debe a que el alcance de los sistemas de información ha cambiado, anteriormente eran desarrollados para ambientes internos de empresas, pero hoy en día el acceso a los sistemas se ha globalizado a través de internet, con esto se torna vital el soporte de grandes cantidades de usuarios y por lo tanto la escalabilidad. Las arquitecturas anteriores comenzaban a mostrar sus limitaciones, principalmente al momento de escalar y agregar nuevas funcionalidades de forma rápida y ágil, como lo demanda al mercado.

El objetivo del proyecto fue el rediseño de un sistema basado en una arquitectura SOA con fuertes requerimientos de calidad de servicios a una arquitectura de Microservicios. La realidad existente se desarrolló en el marco de un proyecto de grado en el año 2014, cuya implementación estaba basada en un ESB. Para cumplir con el objetivo, se investigó la arquitectura de Microservicios, profundizando sobre los beneficios y desafíos que presenta, así como también las tecnologías existentes en el mercado.

Para logar el rediseño se analizó la realidad, los requerimientos y como se podría mejorar la arquitectura. Esto permitió identificar nuevos requerimientos, para luego definir la arquitectura considerando los problemas a enfrentar al aplicar Microservicios.

A partir del diseño propuesto, se implementó el nuevo sistema, sobre el cual se realizaron pruebas de performance para evaluar el comportamiento ante distintos escenarios, alcanzando resultados favorables.

Palabras claves: Service Oriented Architecture (SOA), Microservicios, Service Level Agreement (SLA), Bounded Context, Tolerancia al particionado, Escalabilidad.

1 Introducción

A lo largo del tiempo, las arquitecturas de los sistemas han evolucionado, buscando solucionar las limitaciones que surgen a medida que se requieren nuevas funcionalidades y requerimientos no funcionales como escalabilidad, concurrencia, consistencia y seguridad.

A fines de los años 90 los sistemas comenzaron a incrementar su dimensión tornándose complejos y a su vez, estaban altamente acoplados debido a que la comunicación se realizaba a través de métodos remotos orientados a objetos. SOA surgió en la década del 2000 para solucionar la integración entre los sistemas, introduciendo varios principios que permiten crear sistemas desacoplados y estructurados en función a servicios.

Con el pasar de los años las aplicaciones continuaron incrementando su dimensión y comenzaron a usarse masivamente tornándose vital el soporte a grandes cantidades de usuarios, debiendo escalar el sistema. Para escalar un sistema SOA estructurado como un "monolítico", se debe replicar el sistema en su totalidad, cuando quizás solo una parte es la que necesita escalar. Ante sistemas de gran dimensión agregar una nueva funcionalidad es costoso ya que es necesario hacer deploy y testing del sistema "monolítico" completo, lo cual no es compatible con la demanda de un mercado competitivo que requiere nuevas funcionalidades de forma rápida y ágil.

Es en este marco donde surge la arquitectura de Microservicios, donde la idea es tener pequeños sistemas con funcionalidades específicas, los cuales trabajan en conjunto para lograr el objetivo del sistema, en contraposición de un único sistema "monolítico". Cada uno de estos subsistemas tiene un desarrollo, deploy y testing independiente, lo cual permite mayor agilidad para la incorporación de nuevas funcionalidades, disminuyendo la salida al mercado. A su vez, estos subsistemas pueden escalar de forma independiente permitiendo adaptarse rápidamente a cambios en la carga de los sistemas.

Esta nueva arquitectura provee varios beneficios, pero presenta nuevos desafíos debido a la naturaleza distribuida, al existir muchos componentes para monitorear, administrar y hacer deploy. También, al existir comunicaciones vía red entre los componentes se generan nuevos puntos de falla, pero el sistema debe comportarse como si las comunicaciones fueran locales, por lo que deben diseñarse siendo tolerantes a este tipo de fallas.

1.1 Objetivos

El objetivo de este proyecto es migrar un sistema Monolítico basado en una arquitectura SOA hacia una arquitectura de Microservicios. Para esto, es necesario estudiar en profundidad la arquitectura de Microservicios, investigando sus beneficios y desafíos que surgen al momento de diseñar. Luego, analizar las tecnologías existentes realizando prototipos para poder evaluarlas y elegir la que mejor se adapte al sistema.

Los objetivos específicos del proyecto son los siguientes:

- Estudiar la arquitectura Microservicios, compararla con la arquitectura SOA e identificar buenas prácticas y nuevos desafíos que se presentan.
- Estudiar las tecnologías de Microservicios y desarrollar prototipos.
- Analizar y rediseñar un sistema basado en una arquitectura SOA a una de Microservicios.
- Implementar el diseño propuesto.

- Realizar pruebas de performance, escalabilidad y tolerancia a fallas del sistema.
- Evaluar la experiencia de trabajar con una arquitectura de Microservicios.

1.2 Organización del documento

En el capítulo 2 se presenta un marco conceptual del estudio realizado, centrándose en los desafíos que presenta una arquitectura de Microservicios.

En el capítulo 3 se realiza un análisis sobre el sistema a rediseñar, describiendo requerimientos y restricciones. Además, se presenta los requerimientos del nuevo sistema y el alcance del proyecto.

En el capítulo 4 se presenta la solución propuesta mostrando la arquitectura, como interactúan los diferentes componentes y las principales decisiones de diseño.

En el capítulo 5 se brindan aspectos relevantes de implementación junto con las pruebas realizadas, en éstas se destacan las de performance sobre las cuales se brinda un análisis.

En el capítulo 6 se detalla la gestión del proyecto.

Finalmente, en el capítulo 7 se exponen las conclusiones obtenidas y posibles trabajos futuros.

2 Marco conceptual

En este capítulo se introducen conceptos y tecnologías para entender el resto del documento. En primer lugar, se presenta una descripción general de la arquitectura orientada a servicios (SOA). Luego, se introduce el concepto de la arquitectura de Microservicios, y se describen sus principales beneficios y desafíos. Finalmente, se comparan las arquitecturas SOA y Microservicios.

2.1 Arquitectura orientada a servicios (SOA)

SOA es un patrón de arquitectura para diseñar sistemas de software y proveer servicios a través de interfaces públicas. [1]

En la orientación de servicios, los servicios son entidades de software autónomas, autocontenidas e independientes de la plataforma. A su vez, proveen funcionalidades de negocio y tienen una interfaz pública; pudiendo ser descubiertos, invocados y combinados entre ellos.

Tradicionalmente en las aplicaciones SOA el software es un *Monolítico*, es decir, se tiene un código dividido en módulos, pero se empaqueta como una única pieza ejecutable. Esto permite tener un *deploy* simple ya que solamente se requiere hacer *deploy* de dicha pieza. De aquí en adelante cuando se haga referencia al término *Monolítico* se estará haciendo referencia a esta definición.

SOA es una arquitectura cliente-servidor, donde el servidor expone sus funcionalidades como servicios que son consumidos por clientes y se comunican mediante protocolos de comunicación, típicamente *http*.

En la teoría, se tendría un registro donde los clientes encontrarían los servicios y el servidor se encargaría de publicarlos, esto se puede observar en la Figura 1. En la práctica solamente se tiene el cliente y el servidor, la información necesaria para invocar los servicios se intercambia por otros medios, como ser e-mail. [2]

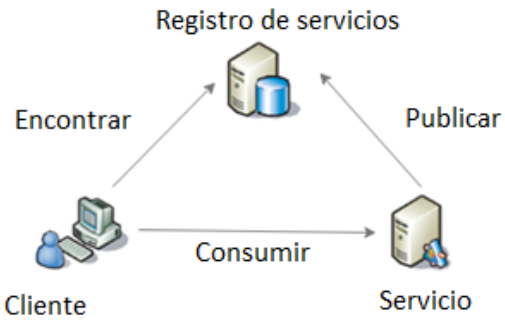

Figura 1 - Comunicación SOA

La arquitectura SOA establece ocho principios que se explican a continuación [1].

Contratos de servicios estandarizados: Este es el principio fundamental que establece que tiene que existir un contrato estandarizado para cada servicio. Los servicios expresan su propósito y funcionalidades a través de su contrato.

Servicios con bajo acoplamiento: Este principio hace referencia a minimizar el nivel de dependencia entre servicios y consumidores, teniendo como resultado el mínimo impacto a la hora de evolucionar.

Abstracción de servicios: Este principio pone énfasis en ocultar los detalles internos del servicio. El servicio debe de ser una caja negra, únicamente definido por su contrato.

Reutilización de servicios: Permite que la lógica de los servicios sea reutilizada, posicionando los servicios como recursos reutilizables.

Autonomía de servicios: Los servicios son autónomos, teniendo alto control sobre su entorno de ejecución.

Servicios sin estado (Stateless): La idea de este principio es diseñar los servicios para recordar el estado solamente cuando es requerido, minimizando el consumo de recursos.

Descubrimiento de servicios: Los servicios se complementan con metadatos comunicativos a través de los cuáles pueden ser descubiertos e interpretados de forma efectiva, fácil de comprender e identificar. Apuntando a aumentar la reutilización y reducir la posibilidad de desarrollar servicios cuyas funcionalidades se superpongan.

Composición de servicios: La idea de este principio es lograr que un servicio pueda formar parte de otros más complejos. Permitiendo la construcción de nuevos servicios a partir de los ya existentes y logrando una mayor agilidad en la construcción.

2.1.1 Arquitectura de referencia

Se presenta una arquitectura de referencia de SOA, la misma cuenta con siete capas que se pueden observar en la Figura 2. [3]

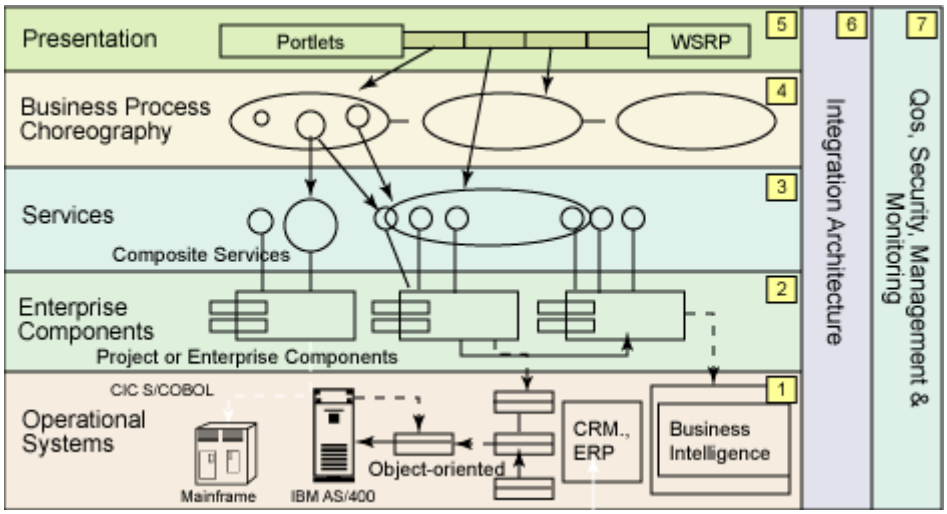

Figura 2 - Arquitectura de referencia [3]

Capa 1: Sistemas operacionales. Está formada por las aplicaciones existentes o legadas. Esto permite reutilizar los sistemas existentes e integrarlos con nuevos sistemas.

Capa 2: Componentes empresariales. Incluye componentes de software, donde cada uno provee la implementación de un servicio, que puede utilizar funcionalidades de varios

sistemas de la capa de Sistemas Operacionales. En esta capa se garantiza la alineación entre la descripción y la implementación del servicio.

Capa 3: Servicios. Esta capa contiene los servicios que provee una arquitectura SOA.

Capa 4: Proceso de negocio. Se combinan los servicios expuestos en la capa de anterior para crear nuevos servicios que dan soporte a los procesos de negocio, aplicando el principio de composición.

Capa 5: Presentación. Se maneja la interacción con el usuario final. Además, permite ofrecer datos y funcionalidades a otras aplicaciones y/o usuarios.

Capa 6: Integración. Se encarga de la integración de servicios utilizando herramientas de *middleware* que permitan la transformación de solicitudes y ruteos de mensajes.

Capa 7: QoS. Esta capa provee las capacidades necesarias para monitorear, administrar y mantener la calidad de los servicios como la seguridad, performance y disponibilidad.

2.2 Arquitectura Microservicios

Esta arquitectura surge debido a limitaciones que presenta la arquitectura SOA Monolítica, por ejemplo, si se quiere agregar nuevas funcionalidades sobre un sistema de este tipo, aumentará su tamaño, lo cual tiene varias implicancias [4]:

- Los desarrolladores no tienen conocimiento del código en su totalidad ya que las dimensiones se tornan inmanejables.
- Un pequeño cambio en la aplicación genera un nuevo *deploy* de todo el sistema.
- Si se requiere escalar, se debe aplicar a todo el sistema.

La idea principal en Microservicios es poner cada funcionalidad en un componente separado, permitiendo un deploy y escalado independiente. La Figura 3 compara una arquitectura SOA con una de Microservicios.

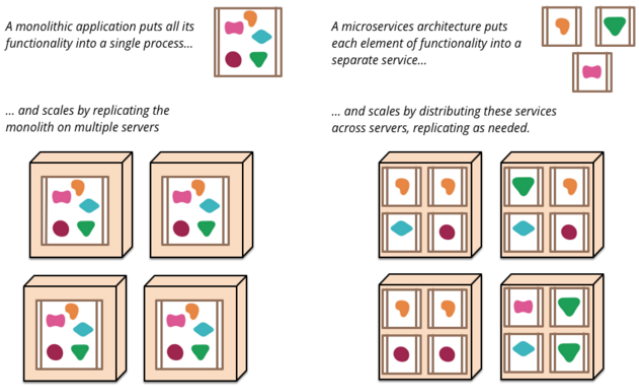

Figura 3 - Monolítico vs Microservicios [5]

Son varias las empresas de renombre que han decidido migrarse a esta arquitectura, claros ejemplos son Netflix, eBay y Amazon. Sin embargo, al ser un concepto relativamente nuevo, no existe una definición estandarizada, es por ello que se presentan diferentes definiciones por algunos referentes en el tema.

Según Microsoft, una arquitectura de Microservicios implica pequeñas piezas de código con su configuración, versionado, *deploy* y escalado independiente. Estas se comunican entre sí sobre protocolos e interfaces bien definidas. A su vez se pueden desarrollar con cualquier tecnología. [6]

Netflix es un vanguardista en el tema, es por tal motivo que se presenta la definición de Adrian Cockcroft quien trabajó como Arquitecto en Netflix y realizó el cambio de la arquitectura *Monolítica* a una basada en Microservicios. Cockcroft define este patrón como una arquitectura orientada a servicios compuesta por elementos con bajo acoplamiento que tienen *"Bounded Contexts"*[1]. [7]

Martin Fowler, escritor de numerosos libros en el desarrollo de software, incluyendo "Refactoring" y "Patterns of Enterprise Application Architecture", a su vez ha escrito varios artículos sobre el tema. Lo define como un enfoque para el desarrollo de una aplicación mediante un conjunto de pequeños servicios, cada uno corriendo en su propio proceso y comunicándose mediante protocolos livianos. Estos servicios son construidos en torno a las funcionalidades del negocio y tienen un *deploy* independiente. Existe un mínimo de gestión centralizada para manejar estos servicios, que pueden estar escritos en diferentes lenguajes de programación y utilizar distintas tecnologías de almacenamiento. [5]

Sam Newman quien escribió el libro "Building Microservices" define de forma breve a los microservicios como servicios pequeños y autónomos que trabajan de forma conjunta. [8]

Como se mencionó anteriormente no existe una definición estandarizada, a su vez las definiciones de los referentes se complementan, es por tal motivo que se creó una definición para este proyecto tomando como base las definiciones anteriores:

Es un patrón de arquitectura que divide al sistema en base a las funcionalidades del negocio, definiendo los "Bounded Context". Esta división determina componentes del sistema que son autónomos, dedicados y cooperan para un fin común, interactuando a través de interfaces bien definidas. Los componentes son implementados eligiendo la tecnología más adecuada para su funcionalidad, éstos se pueden escalar y hacer deploy de forma independiente.

En la Figura 4 se da un ejemplo de este patrón donde se muestran cuatro componentes implementados con diferentes tecnologías y sus interacciones. Cada uno de ellos cumple con una funcionalidad dentro del proceso de negocio. El cliente invoca a un servicio y para responder la solicitud varios componentes cooperan aportando su funcionalidad del negocio.

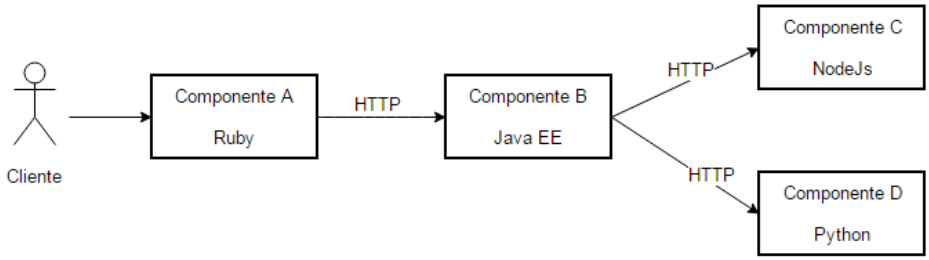

Figura 4 - Ejemplo Arquitectura de Microservicios

[1] En la sección 2.2.1 se describe este concepto.

2.2.1 Bounded Context

Bounded Context es un concepto central en el marco de Microservicios. Un *Bounded Context* cumple una funcionalidad de negocio específica, teniendo límites bien definidos que determinan una interfaz. Los Bounded Context interaccionan entre sí solamente a través de las interfaces y para el exterior se comportan como una caja negra. La idea es que el modelo de cada uno sea independiente y refleje los conceptos de la realidad, respetando los nombres de la misma. [8] [9] [10]

Para comprender mejor este concepto, se aplicará sobre la realidad de una empresa que vende computadoras, la misma tiene dos departamentos, uno encargado de la venta y otro del soporte técnico que arregla defectos en las computadoras.

Cuando un cliente desea arreglar una computadora se comunica con el departamento de soporte, para validar que esa computadora fue comprada en la empresa soporte deberá comunicarse con el departamento de ventas.

Se desea realizar un sistema para esta empresa, es por ello que es necesario identificar los *Bounded Context.*

Se identifican dos funcionalidades de negocio la venta y el soporte técnico, es por ello que se definen dos *Bounded Context:* Ventas y Soporte.

Los *Bounded Context* Ventas y Soporte deben comunicarse para validar que una computadora se compró en la empresa, lo que define una interfaz entre ambos. A su vez, los departamentos de ventas y soporte manejan productos, que se ven reflejados en el modelo de cada *Bounded Context* de la siguiente manera*:*

- En el *Bounded Context* Ventas un Producto contiene un identificador y un precio.
- En el *Bounded Context* Soporte un Producto contiene un identificador, año y número de serie.

Como se puede ver los dos modelos manejan Productos, pero estos hacen referencia a conceptos distintos, lo cual también sucede en la realidad ya que en Ventas solamente interesa el precio de la computadora para venderla, en cambio para arreglarla son necesarios otros datos. En la Figura 5 se muestran los *Bounded Context* con sus modelos*.*

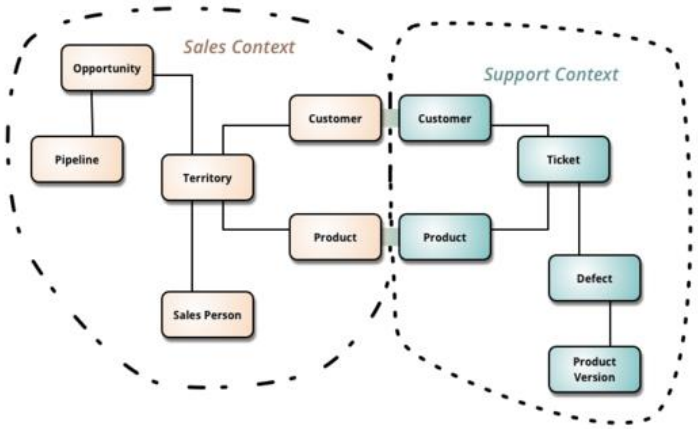

Figura 5 - Ejemplo de Bounded Context [8]

2.2.2 Beneficios principales

Una arquitectura de Microservicios presenta los siguientes beneficios principales: [8]

Heterogeneidad de tecnologías: Se puede utilizar una tecnología diferente en cada componente, utilizando en cada caso la que mejor se ajuste a los requerimientos.

Aislamiento ante fallas (Resilience): Ante una falla en una entidad, el resto del sistema puede seguir funcionando siempre y cuando no se produzca una falla en cascada. En cambio, en un sistema monolítico si una parte falla, el sistema en su totalidad dejará de funcionar. Una falla en cascada ocurre cuando existen invocaciones encadenadas de servicios y una falla, lo que lleva a un efecto dominó en las fallas.

Escalabilidad horizontal y vertical: Se puede escalar horizontalmente replicando componentes para balancear la carga entre los nodos duplicados, sin tener que duplicar todo el sistema. Para escalar verticalmente se añaden recursos al componente que lo requiera sin la necesidad de realizarlo en todo el sistema.

Deploy independiente: Al realizar un pequeño cambio no es necesario realizar el *deploy* de todo el sistema, solamente se realiza sobre el componente correspondiente.

Alineamiento organizacional: La idea es que pequeños grupos trabajan enfocados en diferentes partes del código, logrando una mayor productividad ya que se logra tener un mejor entendimiento del código, al ser de menor tamaño que el monolítico.

Composición de servicios: Permite la reutilización de servicios, a su vez se pueden construir servicios con mayor complejidad a partir de los ya existentes.

Actualización tecnológica: Si se desea cambiar de tecnología o mejorar la implementación de un componente se puede realizar de forma independiente, sin la necesidad de cambiar el resto del sistema.

2.3 Desafíos en Microservicios

Más allá de los beneficios existentes, hay ciertos desafíos a enfrentar para los cuales existen diferentes soluciones, algunos de ellos se describen a continuación.

2.3.1 Comunicación entre servicios

Debido a la naturaleza distribuida de este tipo de arquitectura, es necesario definir el protocolo de comunicación entre cada uno de los servicios. En un mismo ecosistema de Microservicios se pueden aplicar diferentes tipos de comunicación según corresponda, por lo cual el desafío aumenta exponencialmente en función de la cantidad de servicios. Se identifican dos tipos de comunicación, sincrónico y asincrónico:

- En una comunicación sincrónica el cliente envía una solicitud y queda bloqueado hasta recibir una respuesta del servicio, en este momento sabe si la operación se realizó exitosamente o no.
- En una comunicación asincrónica el cliente envía una solicitud y no sabe inmediatamente si la operación fue exitosa una vez que invoca el servicio.

Ejemplos de comunicación sincrónica son *REST* y SOAP utilizando el patrón *request/response* que permiten tener un control sobre las operaciones identificando si se realizaron exitosamente. Mientras que un ejemplo de comunicación asincrónica es una cola

de mensajes, donde un sistema envía un mensaje a una cola y otro lo consume, el sistema que envía el mensaje no espera una respuesta por dicho procesamiento. Este tipo de comunicación permite normalizar la velocidad de procesamiento, debido a que el componente que consume mensajes de la cola, lo hará a medida que tenga recursos disponibles. Además, la cola de mensajes es independiente de los componentes que interactúan con ella, si estos no están disponibles, la cola puede persistir los mensajes hasta que sean consumidos. La Figura 6 muestra un ejemplo de los tipos de comunicación mencionados.

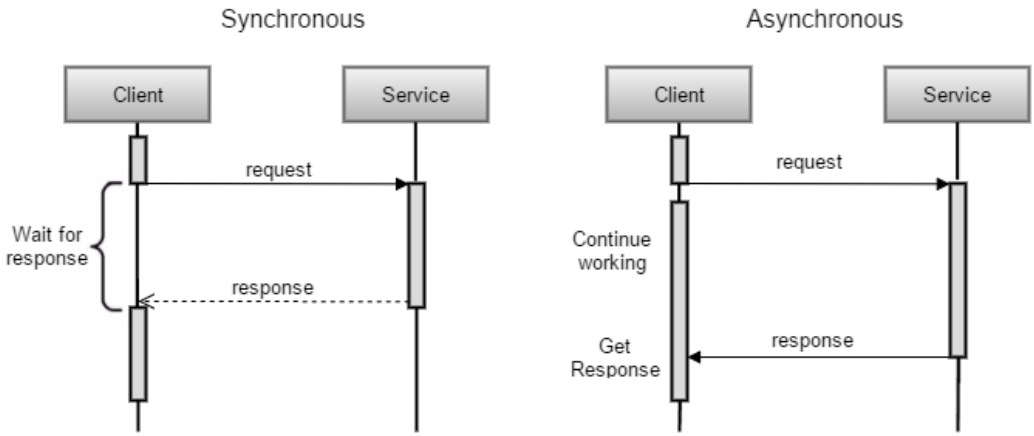

Figura 6 - Sincrónico vs Asincrónico

2.3.2 Modelo de ejecución

Al diseñar una arquitectura de este tipo, lo primero a definir es cómo se comunican los servicios. Una vez definido esto, se requiere definir el flujo que determine la ejecución del proceso de negocio, existen dos maneras de enfrentar este problema:

- Orquestación, en donde una entidad se encarga de manejar el proceso de negocio comunicándose con el resto de las entidades.
- Coreografía, en la cual cada servicio debe saber qué rol cumple en el proceso, sin tener conocimiento del resto del sistema, al recibir un mensaje lo procesa y envía respuestas a quienes corresponda.

A modo de ejemplo, si se tiene un servicio A que es la composición de los servicios B, C y D, se puede utilizar cualquiera de los dos modelos de ejecución.

Si se maneja una solución con orquestación, el servicio A será encargado de llamar a cada uno de los otros servicios, como se puede observar en la Figura 7.

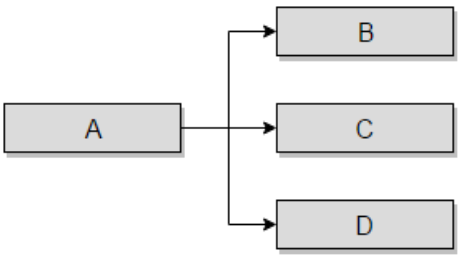

Figura 7 - Ejemplo de orquestación

Este modelo tiene la ventaja que una entidad conoce el proceso de negocio en su totalidad y es más fácil cambiar la lógica de negocio ya que se encuentra centralizada. Como desventaja, si la entidad que maneja la lógica falla, fallará todo el proceso.

Si para el mismo problema se opta por una solución basada en coreografía cada uno de los servicios sabe qué rol cumple en el sistema y reacciona ante eventos. Cada servicio se suscribe al evento de procesar A, cuando este evento ocurre cada uno reacciona según corresponda, en la Figura 8 se muestra este ejemplo.

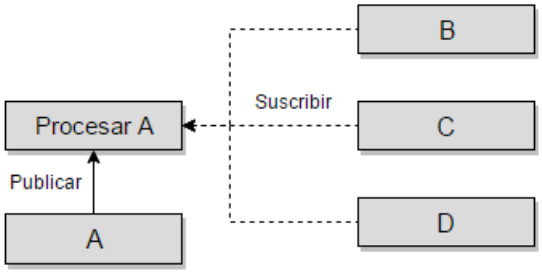

Figura 8 - Ejemplo de coreografía

Este modelo presenta la ventaja de ser desacoplado, si en un futuro el servicio A necesita agregar otra funcionalidad, simplemente se agrega un nuevo servicio que se encargue de ejecutar la nueva funcionalidad sin la necesidad de cambiar ningún otro componente. Posee la desventaja que se pierde la idea global del proceso de negocio, ya que la lógica del mismo se encuentra dividida.

2.3.3 Disponibilidad

En una arquitectura de Microservicios se tienen muchos componentes y para tener disponibilidad en todo el sistema se debe garantizar la misma en cada uno de los ellos. Una forma de solucionar este problema es escalar los componentes de forma independiente.

Para visualizar las formas de escalar un sistema existe el modelo "*Scale Cube*" [11], en la Figura 9 se representan las tres dimensiones en las que se puede escalar un sistema.

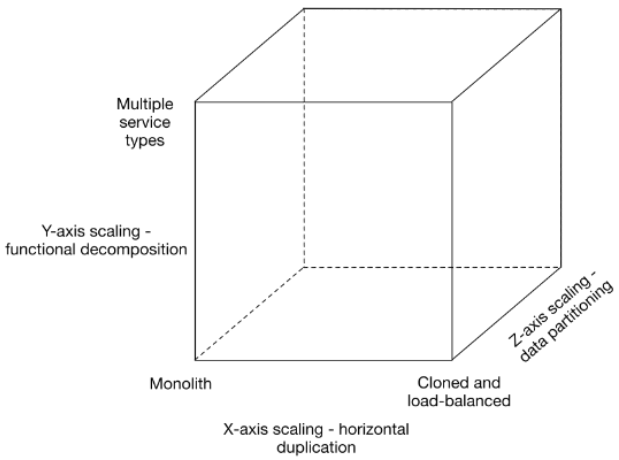

Figura 9 - Scale Cube [11]

Eje X: Este eje implica repartir la carga entre varias instancias de la aplicación, lo que permite escalar horizontalmente. Se reparte la carga aplicando algún algoritmo de balanceo.

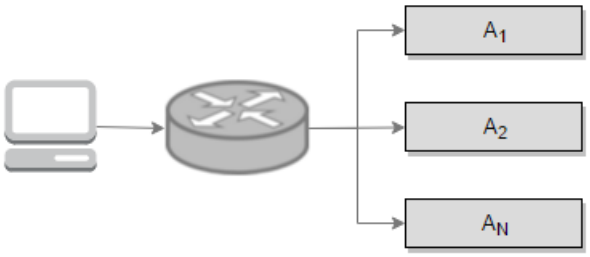

Figura 10 - Eje X

Eje Y: Este eje implica dividir el sistema en varios subsistemas. Si bien no hay una metodología definida, existen algunas estrategias. Una de ellas es dividir por casos de uso o funcionalidades del sistema.

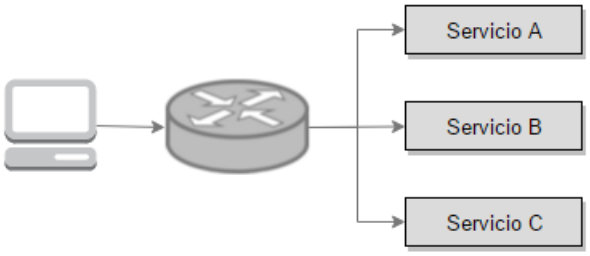

Figura 11 - Eje Y

Eje Z: Para este caso, se divide según los datos de las solicitudes. Un componente del sistema es responsable de realizar el ruteo de los pedidos que se correspondan con cada instancia. Por ejemplo, como se muestra en la Figura 12 el ruteo se realiza de forma alfabética, el servidor que atenderá la solicitud depende del nombre del producto.

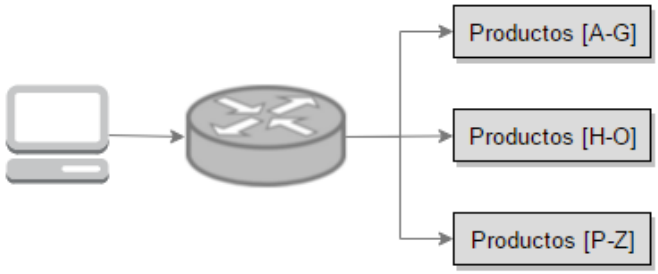

Figura 12 - Eje Z

En Microservicios primero que nada se apunta a la división por funcionalidades (eje Y), además se pueden utilizar las otras dos técnicas para cada uno de los componentes resultantes.

2.3.4 Tolerancia al particionado

Se presenta el problema de la tolerancia al particionado debido a que un sistema basado en una arquitectura de Microservicios debe comportarse como un monolítico, pero este se encuentra particionado en varios componentes. Estos componentes se comunican vía red, lo que puede causar fallas en la comunicación o lo que es peor comunicaciones lentas.

Otro problema que surge del particionado son las fallas en cascada por *timeout*, debido a que una falla puede causar un efecto dominó degradando el sistema en su totalidad, generando demoras excesivas. Para visualizar mejor este problema, se presenta un ejemplo representado en la Figura 13, donde se tienen tres servicios que se invocan de forma sincrónica en cadena y el servicio C está respondiendo lento, lo que provoca que los servicio A y B estén bloqueados durante mucho tiempo, provocando una degradación del sistema en su totalidad.

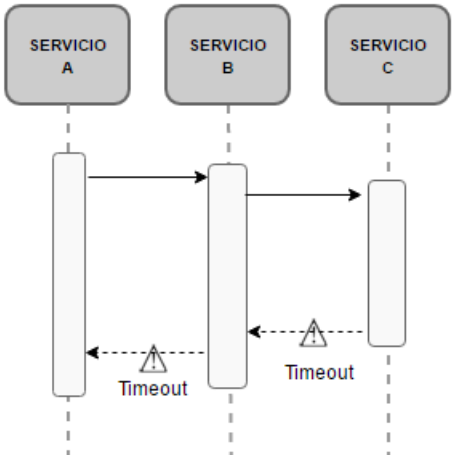

Figura 13 - Ejemplo falla en cascada

Es imposible pensar en un sistema de microservicios y no pensar en mecanismos para que sea tolerante al particionado, es por este motivo que a continuación se presentan mecanismos para crear sistemas tolerantes a las fallas en el particionado.

Time out*:* Previenen esperas infinitas, dado que un servicio puede no contestar por una falla interna o por saturación.

Patrón Retry*:* Aplica una política de reintentos que permite ante fallas temporales de conexión lograr la comunicación deseada.

Circuit Breaker*:* Este patrón se basa en la idea de verificar la disponibilidad de servicios, detectando fallas en la comunicación. Este patrón previene enviar pedidos a sistemas saturados o que probablemente fallen. Con este patrón se evitan los errores en cascada.

Para implementar este patrón se diseña una máquina de estado que se ilustra en la Figura 14.

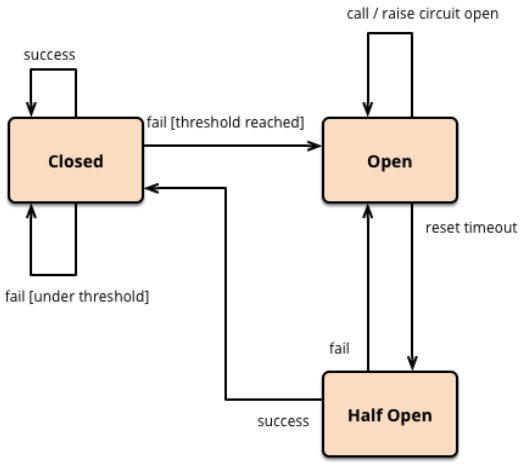

Figura 14 - Circuit Breaker [12]

La máquina de estado tiene los siguientes estados:

- *Closed*: En este estado se permanece mientras el servicio se encuentra disponible. Se cuenta la cantidad de fallas y cuando excede un umbral pasa al estado *Open*.
- *Open*: Se espera cierto tiempo y luego se pasa al estado *Half-Open*. En este estado los pedidos que llegan son rechazados.
- *Half-Open*: Se verifica la ausencia de la falla, en caso que la misma persista se pasa nuevamente al estado *Open*. Si alcanza un número de operaciones exitosas pasa a *Closed*.

BulkHead: Este patrón consiste en separar una parte del sistema para protegerla del resto, aislando el error.

Establece una analogía con un barco, donde si una sección está dañada, cerramos las compuertas del "*Bulkhead*" perdiendo una parte del barco, pero el resto permanece intacto.

Para explicar mejor este patrón se presenta un ejemplo, como se muestra en la Figura 15 donde se tiene un *pool* de conexiones para acceder a dos servicios *Legacy* diferentes, si un sistema *Legacy* comienza a responder de forma lenta, esto va a afectar a las conexiones que se realicen al otro sistema. Aplicando este patrón se separa el pool de conexiones, teniendo uno para cada sistema *Legacy*, y de esta forma si un sistema *Legacy* no funciona correctamente las llamadas al otro no se ven afectadas.

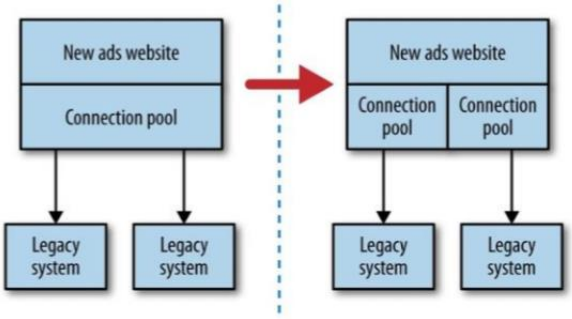

Figura 15 - Ejemplo de BulkHead

Throttler: Este patrón permite limitar la cantidad de mensajes que se envían a un servicio evitando la saturación del mismo.

2.3.5 Consistencia

Un sistema de Microservicios debe ser tolerante al particionado y a su vez por lo general se desea tener alta disponibilidad, pero el Teorema del CAP [8] determina que en un sistema distribuido es imposible garantizar simultáneamente las propiedades de disponibilidad, particionado y consistencia.

Si se quiere tener cierto nivel de consistencia, se puede optar por un tipo de consistencia no tan fuerte como la provista por las transacciones, la consistencia eventual. Este concepto garantiza que, si existen diferentes componentes que comparten datos y se realiza alguna modificación sobre éstos, en algún momento el sistema será capaz de aplicar dichos cambios sobre todos los componentes. El principal problema de la consistencia eventual es que en algún momento del tiempo se pueden obtener datos desactualizados.

Para lograr la consistencia existen diferentes estrategias:

- "*Try again later":* se basa en el reintento ante fallas hasta lograr la operación deseada.
- Compensación: se realizan operaciones de compensación que cancelen las acciones realizadas.

2.3.6 Testing

Las pruebas en una arquitectura de Microservicios se tornan más complejas debido a su naturaleza distribuida. No solo son necesarias pruebas unitarias de los servicios, sino que también se necesitan de integración. En la Figura 16 se muestra la pirámide del *Testing* que se recomienda para una arquitectura de Microservicios. [13]

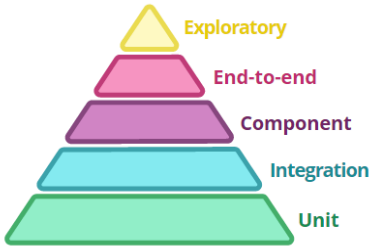

Figura 16 - Pirámide de testing [14]

Pruebas unitarias: Se verifica las funcionalidades del servicio y el cumplimiento del comportamiento deseado.

Pruebas de integración: Se realizan pruebas sobre las interfaces de los servicios, dentro de este tipo de pruebas se tienen:

- *Gateway Integration:* simula errores como *timeout* o caídas de servicios.
- *Persistence integration:* pruebas a nivel de la base de datos para validar el modelo.

Pruebas de componente: Se realizan pruebas sobre un componente simulando los eventos y mensajes que le envían los demás componentes.

End to end: Análisis del sistema entero de un punto a otro. Se verifica que se cumplen los requerimientos.

Prueba exploratoria: Prueba a mano para explorar el sistema.

2.3.7 Deploy, monitoreo y administración

El deploy, monitoreo y administración de un sistema basado en una arquitectura de Microservicios resulta más complejo que si lo comparamos con una monolítica, debido a que se tienen muchos componentes para hacer deploy, monitorear y administrar en comparación a tener sólo uno. Por lo que resulta necesario contar con herramientas que permitan realizar estas actividades de forma centralizada, esto se puede lograr de la siguiente manera:

- Deploy centralizado: mediante herramientas de integración continua que permiten automatizar el proceso, como por ejemplo Jenkins [15].
- Monitoreo centralizado: permite controlar métricas de cada componente y visualizar el correcto funcionamiento del sistema, esto se puede realizar mediante New Relic [16] y Nagios [17].
- Gestión de logs: al tener *logs* distribuidos en varios componentes resulta útil tener una herramienta que permita gestionar de forma centralizada los mismos para lograr una traza de lo que sucede en el sistema, esto puede realizar con Graylog [18] y Logstash [19].

2.3.8 Infraestructura

Los distintos componentes pueden tener diferentes requerimientos de CPU, memoria, sistema operativo y dependencias tecnológicas, por lo que resulta necesario definir la infraestructura para correr todos los componentes del sistema.

La primera opción sería tener un servidor con todos los componentes, pero esto tiene la desventaja que se pueden generar conflictos tecnológicos. Ante este problema se podría tener un servidor por cada componente, pero tal vez no todos los componentes necesiten un host dedicado y se estaría haciendo un uso ineficiente de los recursos lo que generaría un costo elevado.

Esto se podría solucionar utilizando máquinas virtuales (VMs), colocando un componente por VM, logrando el aislamiento deseado. Si el sistema fuera de gran tamaño podría tener ciento o miles de componentes, nuevamente con esta solución se estría haciendo un uso ineficiente de los recursos generando un costo elevado.

Para paliar esto se tienen varios enfoques:

- Muchos componentes por VM: Esto genera los mismos problemas que se tienen en el caso de muchos componentes por host.
- Varias máquinas, cada una con muchas VMs: En este caso se incrementa el costo asociado.

- *Infrastructure as a Service* (IaaS): Brinda una solución basada en virtualización en la nube que permite pagar por las VMs utilizadas. Algunos ejemplos son Amazon Web Services [20], OpenStack [21] y Azure [22].

Otra opción es la utilización de *Containers,* que son mecanismos más livianos que permiten tener muchos *Containers* en una misma máquina física. Esto permite tener un componente independiente por *Container,* logrando que los microservicios estén de forma aislada.

Los *Containers* brindan un mecanismo liviano de virtualización del sistema operativo, permiten mantener de forma aislada una aplicación con los componentes necesarios para ejecutar, empaquetando la aplicación y sus dependencias en un formato de imagen. Esto permite cierta independencia del sistema operativo del host subyacente y la ejecución de varios *Containers* sobre un mismo host, cada uno con la configuración que lo requiera. [23] [24]

Los *Containers* tienen muchas similitudes con las VMs como el aislamiento, pero presentan algunas diferencias que serán explicadas a continuación:

- Comparten el núcleo del sistema operativo.
- Corren como un proceso independiente.
- No requieren incluir un nuevo sistema operativo.
- Solamente carga lo necesario para poder correr la aplicación que se desee.

En la Figura 17 se muestra una imagen que permite visualizar mejor las diferencias.

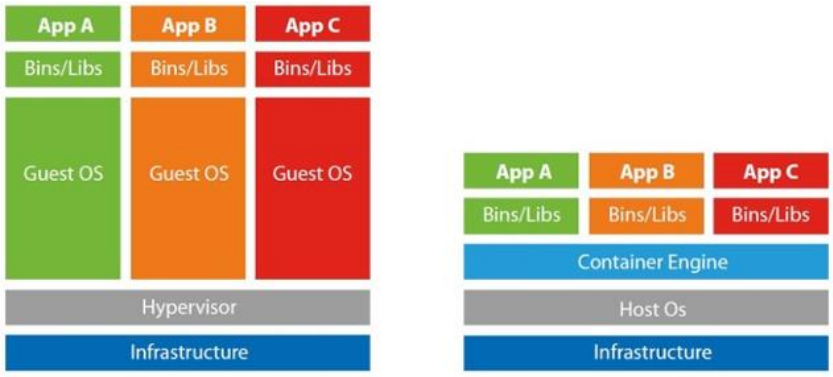

Figura 17 - VMs vs Containers [25]

Como se pude observar la gran diferencia es que en una VM se necesita todo el sistema operativo invitado para correr una aplicación, sin embargo, en un *Container* solo se requiere el *Container Engine,* la aplicación y sus dependencias. El *Container Engine* permite correr y administrar los *Container*s.

Algunas tecnologías IaaS para *Containers* son Tutum [26] y Docker Cloud [27].

2.3.9 Descubrimiento y registro de servicios

En una arquitectura de Microservicios se tienen muchos servicios de forma distribuida y al escalar pueden surgir nuevas direcciones de servicios, resultando complejo si se quiere llevar la correspondencia de servicios con direcciones de forma manual. Es por tal motivo

que se torna necesario contar con métodos que permitan descubrir y registrar los servicios automáticamente, para solucionar este problema existen varios patrones.

En todos los patrones existe un componente denominado *Service Registry* encargado de almacenar las direcciones de los servicios disponibles. [28] [29]

2.3.9.1 Registro de servicios

Self-Registration: Cada instancia es responsable de su registro y baja en el *Service Registry*. Para que el registro no expire algunas implementaciones de este patrón envían un aviso de forma periódica.

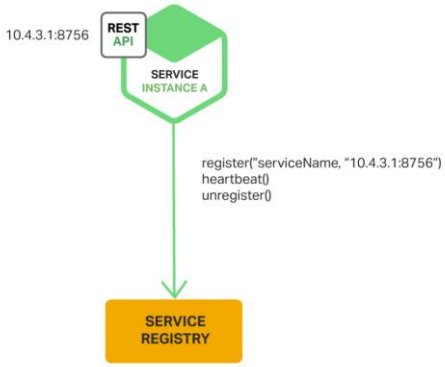

Figura 18 - Self-Registration [28]

Third-Party Registration: En este patrón la responsabilidad del registro de servicios la posee un componente específico. El mismo se encarga de la detección de nuevos servicios, para luego realizar el registro en el *Service Registry*. También se encarga de verificar el estado de los mismos y de esta forma detectar cuando un servicio no está disponible para darlo de baja.

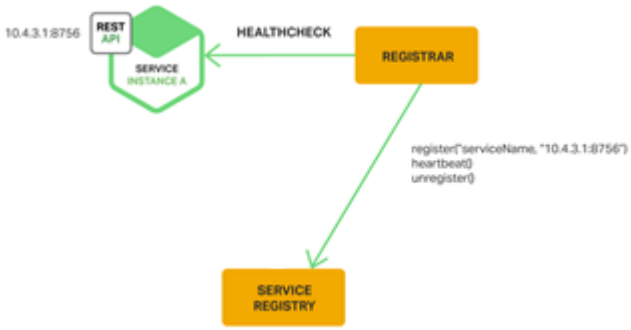

Figura 19 - Third-Party Registration [28]

Las ventajas de un patrón se pueden ver como desventajas del otro. El patrón *Third-Party Registration* tiene la ventaja de no tener que replicar la lógica de registro en cada servicio, pero tiene la desventaja que hay que agregar un componente extra. Mientras que, en el otro patrón, no es necesario contar con un componente adicional para el registro, sin embargo, cada servicio debe implementar la lógica necesaria para la comunicación con el *Service Registry.*

2.3.9.2 Descubrimiento de servicios

Client-Side Discovery: Los servicios son registrados y eliminados en un *Service Registry*. Los clientes consultan el *Service Registry* para conocer dónde se encuentran los servicios a invocar, en caso de existir más de una instancia el cliente debe utilizar un algoritmo de selección.

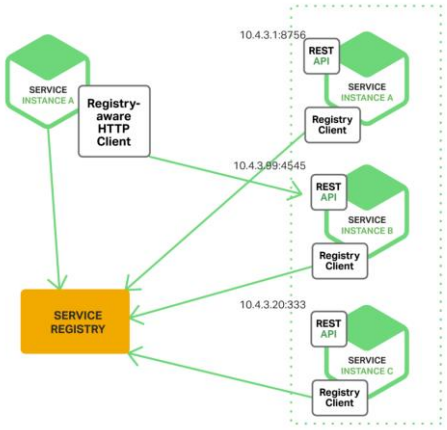

Figura 20 - Client-Side Discovery [28]

Server-Side Discovery*:* Al igual que en el anterior, los servicios son registrados en un *Service Registry.* El cliente invoca un servicio por medio de un *Load Balancer,* éste consulta al *Service Registry* e invoca al servicio requerido.

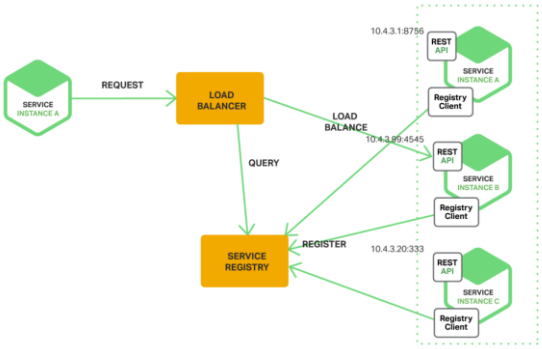

Figura 21 - Server-Side Discovery [28]

Al igual que para los patrones de registro, las ventajas de un patrón se reflejan como desventajas del otro. En *Server-Side Discovery* se tiene la ventaja que la lógica de balanceo no se duplica en todos los servicios. Sin embargo, es necesario agregar un nuevo componente para este fin. En el otro patrón, se puede personalizar el algoritmo de balanceo y no requiere de un componente adicional para ello, pero se debe agregar esta lógica en todos los servicios.

2.4 SOA vs Microservicios

De cierto modo la arquitectura Microservicios se alinea con los principios de SOA, pero brinda una nueva visión en comparación a la arquitectura SOA tradicional monolítica. Como se mencionó anteriormente los servicios en SOA deben ser con contratos estandarizados, de bajo acoplamiento, ocultando los detalles de implementación, siendo reutilizables, autónomos, sin estado, pudiendo ser descubiertos y compuestos. En microservicios estos principios se aplican brindando herramientas para poder cumplirlos.

Para visualizar mejora las diferencias entre dichas arquitecturas, se presenta un cuadro comparativo, donde se exponen algunas diferencias. [30] [31] [32]

	SOA Tradicional Monolítico	**Arquitectura Microservicios**
Protocolo de intercambio típico	SOAP (Simple Object Access Protocol), WS-*	REST
Tecnologías de los servicios	Todos los servicios dentro de la misma aplicación utilizan el mismo lenguaje de programación.	Variadas, específica para cada componente.
Deploy	General sobre todo el sistema.	Deploy independiente para cada componente.
Escalado horizontal y vertical	Sobre todo el sistema.	Sobre cada componente de forma independiente.
Aislamiento ante fallas	El sistema completo se ve afectado.	La falla sólo afecta a un componente.

3 Análisis de requerimientos

En este capítulo se realiza un análisis de la problemática planteada. Esta se basa en rediseñar un sistema monolítico con una arquitectura SOA que se desarrolló en el marco de un proyecto de grado “SLA Enforcer” [33]. Para la comprensión de este sistema se presenta una descripción general, algunos aspectos relevantes de la implementación y restricciones existentes. Finalmente, se presentan los nuevos requerimientos y el alcance final del proyecto.

3.1 Descripción de la realidad

El sistema desarrollado en el marco del proyecto de grado “SLA Enforcer”, de ahora en más denominado *Legacy,* consiste en una red de pagos y cobranzas donde existen múltiples locales de pagos distribuidos. A estos locales concurren usuarios con el fin de realizar un conjunto de trámites (pagos de facturas, giros, cobro de dinero entre otros), de ahora en más pedidos. Para la realización de los pagos el sistema *Legacy* debe invocar a servicios de terceros.

El sistema *Legacy* posee requerimientos de calidad basado en *Service Level Agreement* (SLA). SLA es un contrato entre dos partes, en el cual se especifican las responsabilidades, garantías y prioridades entre éstas. Se utilizan para garantizar la calidad del servicio tanto del proveedor como del cliente.

En la realidad existente se tienen dos tipos de SLA, los cuales son descriptos a continuación y se muestran en la Figura 22:

- SLA Locales de pago (Sistema *Legacy* hacia los locales de pagos)*:* establece que el sistema *Legacy* debe responder en determinado tiempo.
- SLA Servicio Tercero (Servicios de tercero hacia el sistema Legacy): establece que los servicios de terceros deben de responder en determinado tiempo, y que soportan determinada cantidad de pedidos concurrentes.

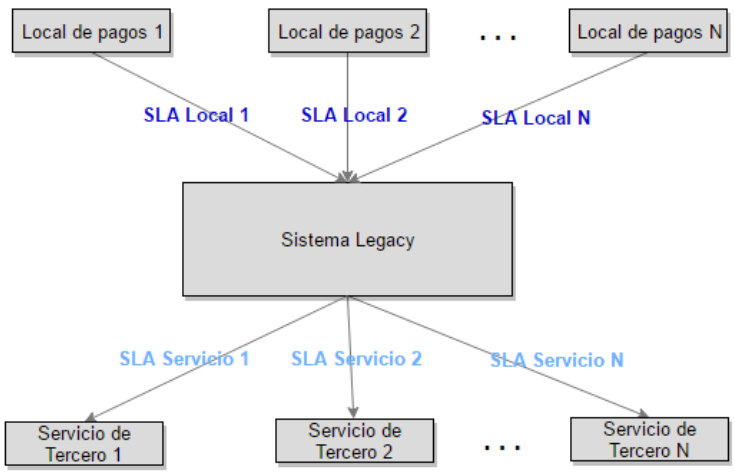

Figura 22 - Interacción del sistema Legacy con los locales de pago y los servicios de terceros

Para el procesamiento de un conjunto de pagos se definen políticas:

- *Best Effort (BE):* Se ejecutan todos los pagos posibles dentro del tiempo acordado.
- *Todo o nada (ToN*): Se ejecutan todos los pagos o ninguno. Si se finaliza el tiempo acordado y no se realizaron todos los pagos, se deben aplicar mecanismos de compensación sobre los pagos ya ejecutados.
- *Dependency on Priority (DoP):* En este caso los pagos tienen prioridades de ejecución, ejecutándose todos los pagos posibles dentro del tiempo acordado, respetando las prioridades.

3.1.1 Aspectos relevantes de la implementación del sistema Legacy

Como se puede observar en la Figura 23 el sistema *Legacy* cuenta con los siguientes módulos:

- CEP: es el encargado de la gestión de los eventos generados por el sistema.
- Monitor: permite el monitoreo del estado del sistema.
- UI: contiene las distintas interfaces de usuario.
- Core: es el componente más importante ya que concentra la lógica de negocio, por este motivo se brindan más detalles sobre algunos módulos del mismo:
    - *Planner:* realiza la validación inicial del pedido y elabora una estrategia de ejecución basándose en la política y estadísticas del sistema.
    - *Coordinator:* maneja las ejecuciones de los pedidos y se encarga de ejecutar las compensaciones cuando el Planner se lo indique.
    - Router: realiza las invocaciones a los servicios de terceros, es el único componente que se comunica con éstos.

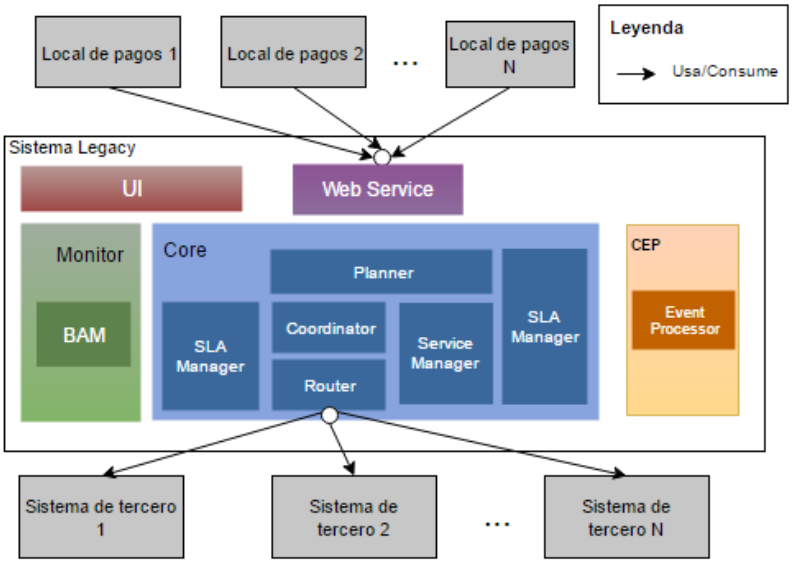

Figura 23 - Vista lógica del sistema Legacy

Para elaborar la estrategia de ejecución el componente *Planner* diseña un grafo de dependencias que determina un plan de ejecución. Un plan posee un conjunto de niveles compuestos por tareas que se ejecutan en paralelo. Cada tarea está compuesta por un conjunto de invocaciones a servicios de terceros, estas invocaciones deben de ser ejecutadas de forma serial. En la Figura 24 se muestra un ejemplo de un plan de ejecución.

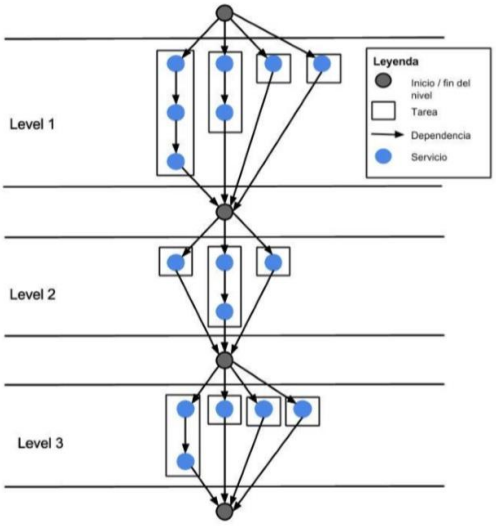

Figura 24 - Plan de ejecución

3.2 Restricciones del sistema Legacy

El sistema *Legacy* presenta ciertas limitaciones que son explicadas a continuación:

Escalabilidad: Si se desea escalar horizontalmente se debe replicar el sistema en su totalidad. Al igual que si se desea escalar verticalmente, es necesario aplicarlo sobre todo el sistema.

Agregar/Modificar SLAs: Si se quiere modificar los SLAs acordados, es necesario modificar el código y hacer un nuevo *deploy* del mismo.

Soporte a nuevos servicios de terceros: Para soportar un nuevo servicio de tercero, se requiere generar una nueva versión del sistema.

Restricción en la comunicación con los servicios de terceros: Se cuenta con la restricción que los sistemas externos deben comunicarse mediante SOAP e implementar la misma interfaz.

Falla del sistema: Si por alguna razón un módulo falla, todo el sistema se verá afectado y en consecuencia quedará inhabilitado hasta que se recomponga todo el sistema.

SLA con servicios de terceros: Cada servicio de tercero restringe la cantidad de pedidos concurrentes, a partir de dicha cantidad los pedidos son rechazados. El sistema existente no toma ninguna acción ante esta restricción.

No se realizan reintentos ante fallas en las compensaciones: El sistema no realiza reintentos si una operación de compensación falla, esto provoca que si una operación de compensación falla el sistema queda inconsistente.

El sistema no reacciona ante timeout: Si al realizar un pago el servicio de tercero no responde dentro del tiempo acordado en el SLA y como consecuencia se obtiene un

timeout, no se realiza ninguna acción. En este caso se toma el pago como no se ejecutado, pero puede ocurrir que el mismo se haya concretado en el servicio de tercero.

3.3 Requerimientos del nuevo sistema

Se desea rediseñar el sistema *Legacy* utilizando una arquitectura de Microservicios. Además, el nuevo sistema debe respetar la lógica de negocio del sistema anterior, manteniendo los mismos requerimientos.

Para definir los nuevos requerimientos del sistema es necesario definir el término dinámico, que implica la posibilidad de incorporar y/o cambiar una funcionalidad o parámetro en *runtime* sin *redeploy* ni modificar el código existente. Para agregar una nueva funcionalidad se requiere crear un nuevo componente que la implemente. El cambio o componente se adapta al sistema y se ve reflejado automáticamente.

3.3.1 Principales requerimientos del sistema Legacy

ID	Requerimiento	Descripción
RQL1.	Cumplimiento del SLA con local de pagos	El servicio expuesto por el sistema deberá de respetar el SLA acordado con el local de pagos.
RQL2.	Política de ejecución	Utilizar la política de ejecución informada en cada invocación.

El conjunto total de requerimientos del sistema *Legacy* se encuentra en el Anexo 1.

3.3.2 Requerimientos agregados

Como mejoras, se agregan los siguientes requerimientos que surgen de las restricciones del sistema *Legacy*:

ID	Requerimiento	Descripción
RQM1.	Arquitectura Microservicios	Rediseñar el sistema *Legacy* utilizando la arquitectura Microservicios.
RQM2.	Escalado horizontal	Escalar horizontalmente los componentes de forma dinámica e independiente.
RQM3.	Soportar nuevos servicios de terceros	Soportar nuevos servicios de terceros dinámicamente.
RQM4.	Protocolos de comunicación con servicios de tercero	Soportar distintos protocolos de comunicación con los servicios de terceros (además de SOAP).
RQM5.	Agregar/Modificar SLA	Cambiar el valor de un SLA y agregar un nuevo SLA de los definidos para un servicio de tercero dinámicamente.
RQM6.	Monitoreo del sistema	Monitorear el sistema de forma centralizada.
RQM7.	Cumplimiento SLA con servicio de terceros (cantidad de pagos concurrentes)	Respetar la cantidad de pedidos concurrentes acordados en el SLA con los servicios de tercero.
RQM8.	Reintento ante fallas en las compensaciones	Realizar reintentos cuando las compensaciones fallen, asegurando que la compensación se realice.
RQM9.	Cancelar los pagos	Realizar cancelaciones ante un error de timeout.

3.4 Alcance del proyecto

El alcance del proyecto se definió en conjunto con el tutor del proyecto a partir del objetivo inicial.

El objetivo del proyecto es aplicar microservicios a una arquitectura SOA, como caso de estudio se utilizó el sistema *Legacy*. Para cumplir con el objetivo se debió, en un principio investigar el nuevo estilo de arquitectura Microservicios, sus principales beneficios y desafíos, así como posibles soluciones a los problemas que se introducen. Luego, se pasó a rediseñar el sistema.

Al momento de rediseñar el nuevo sistema se presentó la problemática de que el sistema anterior tenía una dimensión muy grande y estaba desarrollado con tecnologías desconocidas por el grupo. Por este motivo se decidió acotar el alcance y reimplementar una parte del sistema sin utilizar el código fuente del *Legacy*. A su vez el enfoque del proyecto era la investigación de Microservicios para aplicarlo a una realidad, por lo cual perdía sentido invertir esfuerzo en comprender el funcionamiento del código fuente del sistema *Legacy*.

Para cumplir con las mismas funcionalidades que antes, el nuevo sistema tiene que comunicarse con el *Legacy*. Para llevar a cabo la ejecución de un conjunto de pagos, los locales de pagos se comunican con el sistema *Legacy*, luego de cierto procesamiento delega el trabajo al nuevo sistema el cual se encarga de ejecutar un nivel y comunicarse con los servicios de terceros, esto se puede observar en la Figura 25.

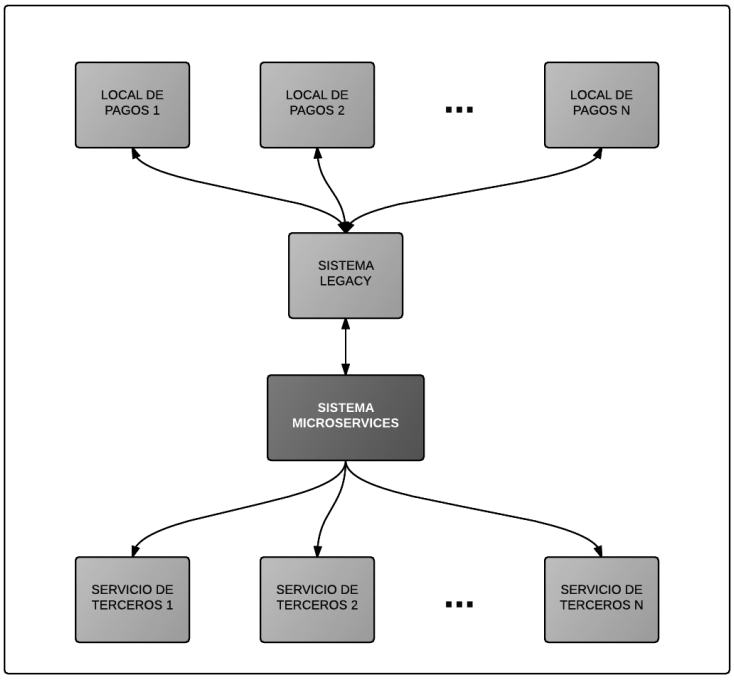

Figura 25 - Comunicación entre sistemas

Este nuevo sistema es capaz de ejecutar un nivel completo, teniendo en cuenta la política de ejecución y aplicando compensaciones en caso de ser necesario. Para que el sistema *Legacy* pueda cumplir con el SLA acordado con los locales de pagos, se define un nuevo SLA entre el sistema *Legacy* y el nuevo sistema. Dicho SLA se recibe en cada nivel a procesar.

Para que el sistema sea capaz de procesar un nivel se rediseñaron los componentes *Coordinator* y *Router*. Al ejecutar un nivel se consideran los mismos mecanismos que se realizan en el sistema *Legacy*, ejecutándose las tareas de un mismo nivel en paralelo, y dentro de cada tarea, los pagos de forma secuencial.

En resumen, los requerimientos del nuevo sistema dentro del alcance son:

ID	Requerimiento	Descripción	Id ref.
RQ1.	Arquitectura Microservicios	Rediseñar los componentes *Coordinator* y *Router* del sistema *Legacy* utilizando la arquitectura Microservicios, respetando la misma lógica de negocio.	RQM1
RQ2.	Ejecutar nivel	Realizar los pagos de un nivel	[Nuevo]
RQ3.	SLA con sistema Legacy	Responder al sistema Legacy dentro del tiempo acordado en el SLA.	[Nuevo]
RQ4.	Consistencia del nivel.	Respetar la consistencia del nivel de acuerdo a la política del mismo. En una política ToN puede implicar realizar compensaciones.	[Nuevo]
RQ5.	SLA con servicio de terceros (cantidad de pagos concurrentes)	Respetar la cantidad de pedidos concurrentes acordados en el SLA con los servicios de tercero.	RQM7
RQ6.	Política de ejecución	Utilizar la política de ejecución informada en cada invocación.	RQL2
RQ7.	Soportar nuevos servicios de terceros	Soportar nuevos servicios de terceros dinámicamente.	RQM3
RQ8.	Protocolos de comunicación con servicios de tercero	Soportar distintos protocolos de comunicación con los servicios de terceros (además de SOAP).	RQM4
RQ9.	Agregar/Modificar SLA	Cambiar el valor de un SLA y agregar un nuevo SLA de los tipos definidos para un servicio de tercero dinámicamente.	RQM5
RQ10.	Reintento ante fallas en las compensaciones	Realizar reintentos cuando las compensaciones fallen, asegurando que la compensación se realice.	RQM8
RQ11.	Cancelar los pagos ante un error de timeout	Realizar cancelaciones ante un error de *timeout* con el servicio de tercero, asegurando la consistencia del sistema.	RQM9
RQ12.	Escalado horizontal	Escalar horizontalmente los componentes de forma dinámica e independiente.	RQM2
RQ13.	Monitoreo del sistema	Monitorear el sistema de forma centralizada.	RQM6

Observación:

Los requerimientos que tienen Id de referencia [Nuevo], surgieron como consecuencia de acotar el alcance, recortando el requerimiento RQM1. Ahora el sistema deberá ejecutar niveles teniendo un que respetar un SLA con el sistema *Legacy*, además deberá tener lógica de compensación de pagos. La política de ejecución *Dependency on Priority* no es utilizada ya que se ejecutan niveles, y esta política determina el orden de los niveles.

4 Diseño de la solución

En este capítulo se describe el diseño de la solución, brindando primero una descripción general presentando las decisiones de diseño y luego siguiendo el modelo de vistas 4+1 de Philippe Kruchten [34]. Finalmente, se exponen aspectos relevantes de la solución.

4.1 Descripción general

El nuevo sistema recibe pedidos del sistema *Legacy*, los procesa y luego le envía el resultado. Los pedidos contienen un nivel de ejecución, la política a aplicar, el tiempo disponible para procesar y un identificador único del nivel.

El procesamiento de los pedidos se lleva a cabo separando el nivel en sus distintas tareas y ejecutándolas de manera concurrente. Estas tareas contienen pagos que se ejecutan de forma serial dentro de una misma tarea, dichos pagos serán concretados realizando invocaciones a los servicios de terceros que corresponda. El nuevo sistema debe respetar el SLA acordado con el sistema *Legacy* referente al tiempo de respuesta.

El nuevo sistema es diseñado aplicando el patrón de arquitectura de Microservicios, atacando los problemas y desafíos que esto implica.

4.2 Decisiones de diseño

Al diseñar una arquitectura basada en Microservicios, son varios los desafíos que se presentan, es por tal motivo que en la presente sección se describen las decisiones de diseño tomadas para cada uno.

4.2.1 Bounded Context

El primer desafío al momento de diseñar una arquitectura de Microservicios es identificar los *Bounded Context*, para ello es necesario estudiar la lógica de negocio y comprenderla. Como el objetivo es rediseñar un sistema, se estudió con mayor detalle aspectos de la arquitectura y diseño para determinar las principales funcionalidades del sistema, las cuales definen los *Bounded Context*.

A continuación, se muestran los *Bounded Context* identificados:

- Level Coordinator: La principal funcionalidad es separar un nivel en diferentes tareas a procesar.
- Task Coordinator: La funcionalidad principal de este *Bounded Context* es ejecutar una tarea de forma serial.
- Connector: Se define un *Bounded Context Connector* por cada tipo de servicio de tercero, el cual tiene la funcionalidad principal de conocer e implementar el protocolo de comunicación hacia el servicio de tercero.
- Consistency Manager: Tiene la funcionalidad de garantizar la consistencia en el sistema, para ello realiza operaciones de compensación.
- Admin: Brinda la funcionalidad de configurar el sistema dinámicamente, agregar nuevos componentes y cambiar parámetros del sistema.

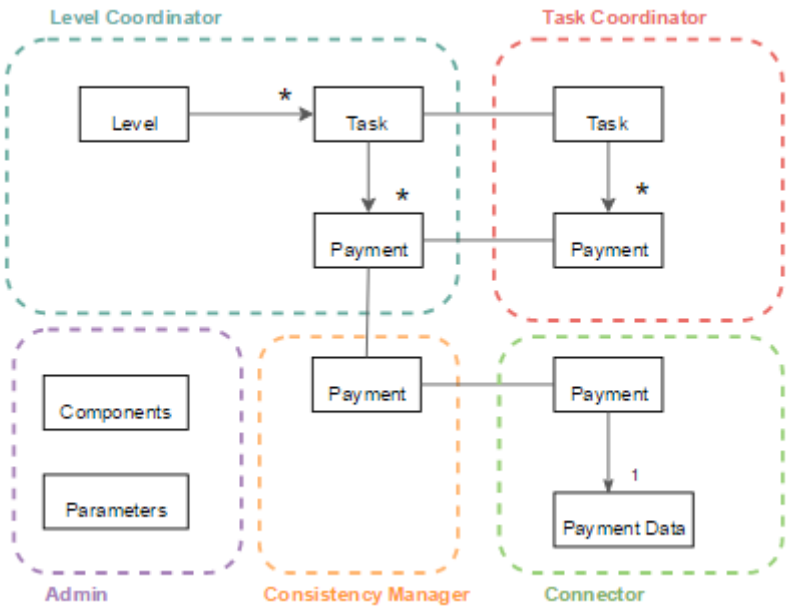

Figura 26 - Bounded Context

En la Figura 26 se puede observar las interacciones entre los *Bounded Context* y los conceptos que maneja cada uno. Si bien en los diferentes *Bounded Context* existen conceptos con el mismo nombre, no hacen referencia a lo mismo. Por ejemplo, en el marco del *Bounded Context Connector* para un *Payment* interesa la información para realizar el pago (por ejemplo, número de cuenta, monto, etc), mientras que en el *Bounded Context Task* no se necesita dicha información.

4.2.2 Comunicación entre servicios

Luego de identificados los *Bounded Context,* la siguiente tarea es definir los tipos de comunicación entre los componentes. Se utilizan los dos tipos de comunicación, sincrónica y asincrónica. En los casos que se necesita obtener el resultado de la invocación para continuar con el procesamiento se eligió una comunicación sincrónica y para los demás casos asincrónica.

Por otro lado, resulta necesario enviar notificaciones de tipo *broadcast* entre algunos de los componentes. Para este caso, se decide utilizar una comunicación orientada a eventos donde un componente publica un mensaje y otros se suscriben al evento.

Para implementar las comunicaciones sincrónicas se decidió utilizar REST debido a que es un protocolo liviano y para las asincrónicas se optó por utilizar una cola de mensajes.

Al utilizar una solución con colas de mensajes, es necesario describir el concepto de los patrones de mensajería utilizados; es por este motivo que aquí se expone cada uno de ellos:

- *Publish/Suscribe:* un mensaje es recibido por todas las instancias que se suscriben al evento. Este patrón se utiliza para el caso que se desea notificar a todas las instancias sobre un evento.

- *Competing Consumer:* permite que múltiples instancias consuman de la misma cola, donde el mensaje es procesado solamente por una de ellas. Este patrón se elige para todas las comunicaciones asincrónicas restantes.

A su vez las colas pueden ser persistentes o no, donde las colas persistentes permiten almacenar y garantizar que no se pierdan mensajes. Cuando se utiliza el patrón *Competing Consumer* las colas son persistentes. En cambio, en el otro patrón no se utilizan colas persistentes, dado que se utiliza con el objetivo de notificación y solamente tiene sentido que el mensaje sea procesado en el momento.

4.2.3 Modelo de ejecución

Habiendo identificado los *Bounded Context* y la comunicación entre los servicios, se procedió a definir el modelo de ejecución. Dado que se tienen varios *Bounded Context,* es necesario definir cómo será la visión global del flujo de ejecución. Para esto se tienen dos modelos, orquestación y coreografía.

Se decide aplicar el patrón de coreografía ya que permite una mayor autonomía en los componentes, logrando que el sistema sea desacoplado. Un requerimiento del sistema es agregar nuevas funcionalidades sin la necesidad de cambiar el resto del sistema y esto resulta más fácil de realizar aplicando coreografía en comparación a tener un orquestador. A su vez, se desea manejar eventos entre los componentes y que los mismos reaccionen ante éstos, lo que es ideal en este tipo de modelo.

4.2.4 Registro de información

Al aplicar el modelo de coreografía, no se tiene una visión global sobre el sistema, por otro lado, es necesario tener información global referente a los niveles, tareas y pagos, con el objetivo de saber si es necesario realizar compensaciones. Es por este motivo que se decide introducir una base de datos para el almacenamiento de esta información.

La primera opción sería tener una base de datos a la cual todos los *Bounded Context* accederían, actualizando la información referente a los pagos, pero esto va en contra de la arquitectura de Microservicios, ya que todos los *Bounded Context* quedarían acoplados al modelo de la base de datos y un cambio afectaría a todos. Por este motivo se decidió que solamente un *Bounded Context* sea el encargado del manejo de la base de datos. Dado que el *Level Coordinator* posee la visión global de los niveles y es el punto de entrada y salida al sistema, se decidió que es el indicado para realizar dicha gestión.

Para modelar una base de datos se cuenta con el modelo relacional y el no relacional. En primera instancia se manejó la opción de tener un modelo no relacional debido a que la información intercambiada es con formato *json*. Por otro lado, el sistema maneja cobranzas lo que requiere cumplir con la consistencia y atomicidad sobre las operaciones realizadas en la base de datos. Por lo cual la decisión se orientó a la utilización del modelo relacional, pudiendo garantizar la propiedad ACID (cumplir con Atomicidad, Consistencia, Aislamiento y Durabilidad) [35], propiedad que no garantizan completamente los modelos no relacionales.

4.2.5 Tolerancia al particionado

En una arquitectura de Microservicios se tiene particionado el sistema, por lo cual es necesario contar con mecanismos que permitan que el sistema sea tolerante ante fallas en la comunicación entre los componentes, con este objetivo se aplican varios patrones:

Patrón Retry: se aplica cuando se realizan las compensaciones de los pagos, debido a que se quiere ser tolerante a fallas en la comunicación y asegurar la ejecución de la operación cumpliendo con el requerimiento RQ10.

Time out: se aplica en todas las comunicaciones sincrónicas para no tener esperas infinitas si un servicio no responde o responde lento.

Bulkhead: se aplica en el *Level Coordinator* y en el *Task Coordinator,* separando el procesamiento de los mensajes referentes a la ejecución de pagos y a notificaciones (por ejemplo, fin de procesamiento). Con esto se logra que, si se tienen muchas notificaciones, no afecten al curso normal del sistema.

Throttler: se decide aplicar este patrón antes de realizar la invocación a los servicios de terceros para cumplir con el requerimiento RQ5 (cantidad de pedidos concurrentes definidos en el SLA).

Circuit Breaker: se aplica para evitar la degradación del sistema ante servicios de terceros que responden de forma lenta. Para ello se agrega un nuevo componente encargado de ejecutar este patrón para cada servicio de terceros, evitando duplicar la lógica referente al patrón en todos los componentes que se requiere.

4.2.6 Disponibilidad

Es deseable que el sistema tenga alta disponibilidad, esto se puede lograr escalando el sistema. Debido al requerimiento RQ12 se requiere escalar cada uno de los componentes de forma independiente.

Dado que se tiene una base de datos con el estado global de los pagos, si se quiere escalar horizontalmente el componente que la utiliza, surge el problema de cómo realizarlo. Una opción sería tener una base de datos por cada instancia, pero esto no es posible porque se quiere tener la información centralizada. Por lo que se decide que las instancias trabajen en modo *cluster*, compartiendo la base de datos.

4.2.7 Consistencia

Por ser un sistema basado en Microservicios es inevitable no tener tolerancia al particionado, pero el sistema también requiere alta disponibilidad. Por el teorema del CAP no se puede tener consistencia, esto es un problema dado que se trata de un sistema de cobranzas.

Para solucionarlo se decidió tener una consistencia eventual sobre el estado de los pagos. Por ejemplo, ante una cancelación de un pago, el estado del mismo para el sistema *Legacy* es no ejecutado, mientras que internamente se debe realiza la compensación. Esto permite brindar una respuesta rápida al sistema *Legacy* mientras que se realiza la cancelación.

4.2.8 Registro y descubrimiento de servicios

Se tienen muchos componentes que requieren escalar dinámicamente, a su vez por el requerimiento RQ7, nuevos componentes deben de ser agregados al sistema de forma dinámica. Esto implica que los componentes se deben dar de alta en el sistema e integrarse de tal forma que puedan ser utilizados sin realizar ningún cambio en el sistema. Por este motivo es necesario contar con mecanismos para el registro y descubrimiento de servicios.

Para evitar la duplicación de la lógica de registro, descubrimiento y balanceo se decide utilizar los patrones *Third-Party Registration y Server-Side Discovery*.

4.3 Modelo 4 + 1

En esta sección se describe el diseño de la solución propuesta, utilizando las vistas 4+1 de Philippe Kruchten. Este modelo permite describir una arquitectura desde distintos puntos de vista, para esto propone 5 vistas: de escenarios, lógica, de componentes físicos, de procesos y de despliegue.

4.3.1 Vista de escenarios

Esta vista se ve representada por los casos de uso de software y sus dependencias. En el sistema se identifican tres actores:

- Sistema *Legacy*: envía la solicitud de ejecución de un nivel.
- Servicios de terceros: dispone de servicios para la ejecución de pagos.
- Administrador: realiza la configuración dinámica del sistema.

A continuación, se muestran los escenarios más relevantes para la arquitectura.

4.3.1.1 Ejecutar nivel

Este escenario representa la invocación del sistema *Legacy* para la ejecución de un nivel. Como se puede apreciar en la Figura 27, de este caso se derivan varios escenarios.

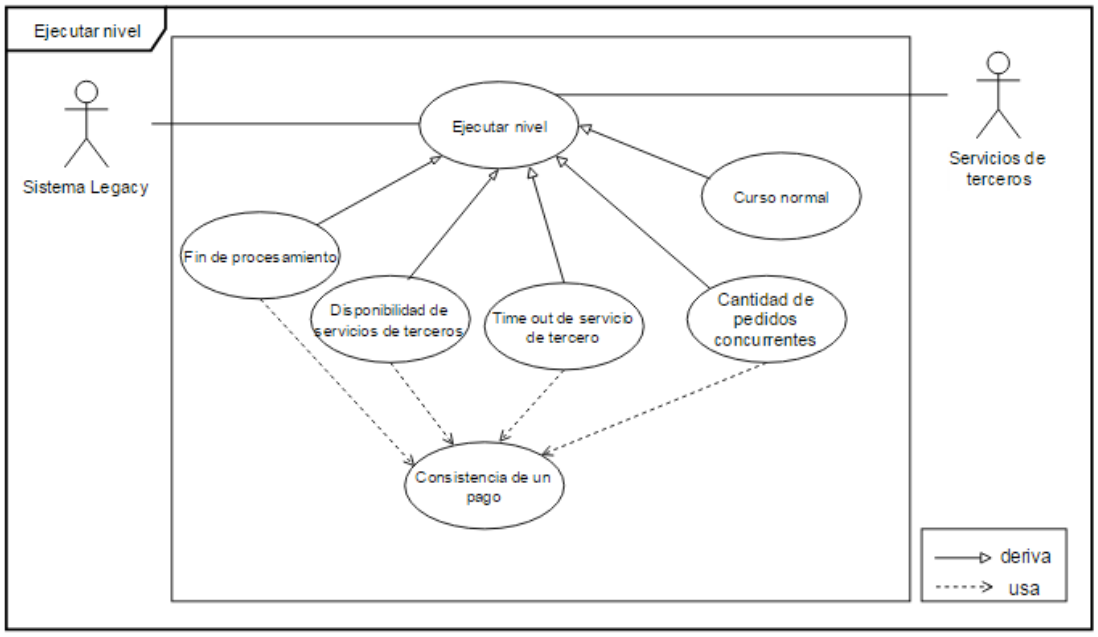

Figura 27 - Escenario Ejecutar nivel

Este escenario es relevante para la arquitectura debido a que a partir de éste se determinan necesarios componentes que cumplan con las siguientes funcionalidades:

- Ejecutar un nivel.
- Ejecutar una tarea.
- Invocar servicios de terceros.
- Mantener la consistencia de los pagos.
- Controlar pedidos concurrentes.
- Reaccionar ante errores en la comunicación con los servicios de terceros, si éstos no están disponibles o si dan *timeout*.
- Detectar que finalizó el fin de procesamiento.

4.3.1.2 Configuración dinámica

Este escenario involucra al administrador cuando realiza cambios sobre el sistema. Las diferentes acciones que puede realizar se pueden observar en la Figura 28.

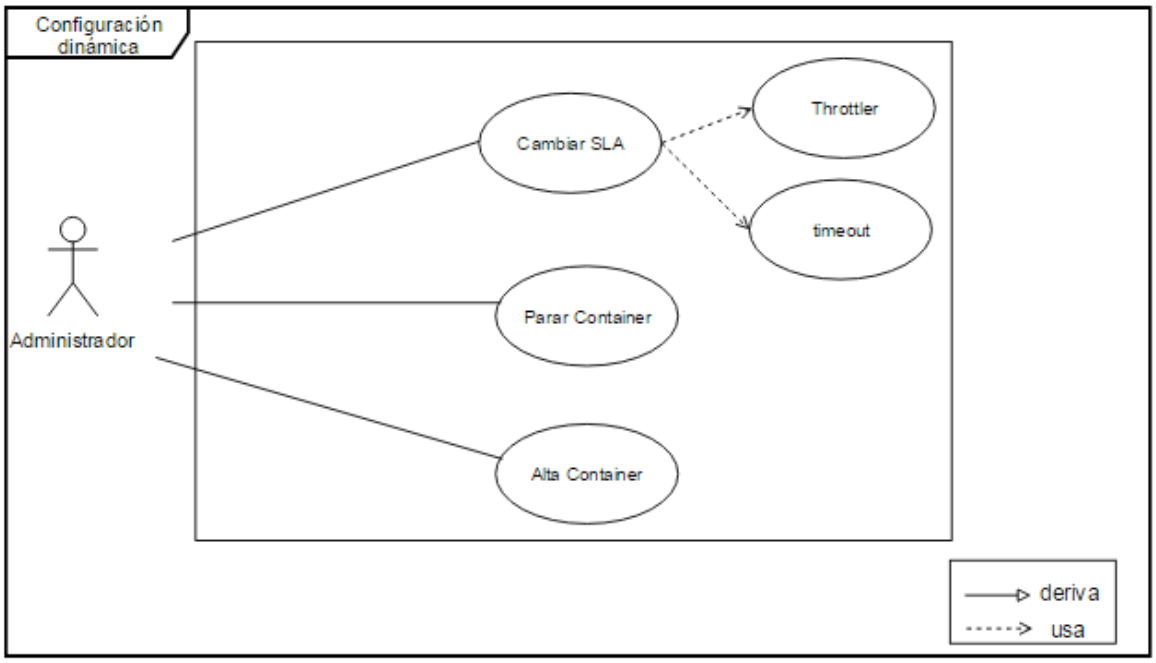

Figura 28 - Escenario configuración dinámica

Este escenario es importante para la arquitectura del sistema porque determina un componente con la funcionalidad de administrar el sistema.

4.3.2 Vista lógica

Los *Bounded Context* identificados en la sección 4.2.1 son mapeados directamente a los componentes de la vista lógica; a su vez se agregan otros que surgen de las decisiones de diseño.

En la Figura 29 se muestra la vista lógica y luego se da una descripción de las funcionalidades de cada componente.

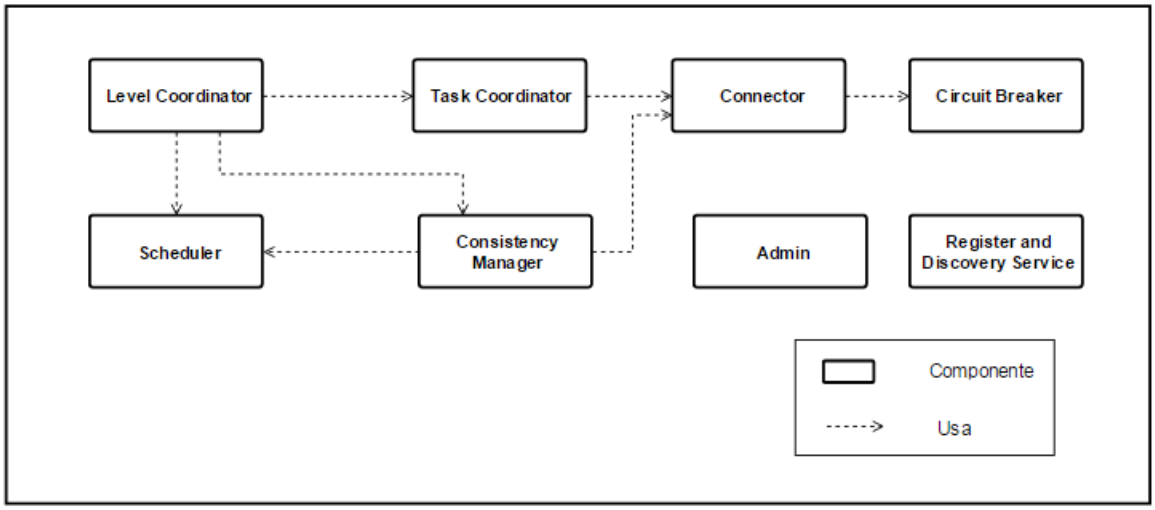

Figura 29 - Vista Lógica

Level Coordinator

Se encarga de recibir un pedido y enviar el resultado al sistema *Legacy*. Se comunica con los siguientes componentes para llevar a cabo la solicitud:

- *Scheduler:* se agenda un evento de fin de procesamiento para ser notificado cunado el tiempo expire.
- *Task Coordinator:* envía cada una de las tareas del pedido.
- *Consistency Manager:* envía los pagos a compensar.

Task Coordinator

Este componente tiene como principal funcionalidad ejecutar una tarea, compuesta por pagos que se deben ejecutar en forma serial. La ejecución de cada pago, se realiza mediante la invocación al *Connector* que corresponda según el servicio de tercero.

Connector

Este componente es el único que conoce la interfaz de comunicación con el servicio de tercero, permitiendo la realización de los pagos, cancelaciones y anulaciones, previamente realizando las trasformaciones necesarias para la invocación. Se comunica con el *Circuit Breaker* para notificar si la comunicación con el servicio de tercero fue exitosa o no.

Circuit Breaker

Este componente surge para evitar la degradación del sistema ocasionada por servicios que no funcionan correctamente implementando el patrón *Circuit Breaker* para cada tipo de *Connector*, enviando notificaciones de cambio de estado.

Consistency Manager

Se encarga de la compensación de los pagos mediante la invocación al Connector que corresponda, permitiendo que en algún momento el sistema quede consistente. Si alguna compensación falla se agenda un evento a través del *Scheduler* para realizar nuevamente la compensación.

Scheduler

Se agrega ante la necesidad de agendar eventos para su posterior ejecución.

Admin

Este componente permite cambiar los SLA acordados, además permite dar de alta y baja a componentes del sistema.

Register and Discovery Service

Este componente se encarga del registro y descubrimiento de servicios, implementando los patrones *Third-Party Registration* y *Server-Side Discovery*.

4.3.2.1 Vista lógica con el sistema Legacy

Para tener una visión global del sistema, en esta sección se muestran los componentes lógicos del sistema *Legacy* que fueron rediseñados y a qué componentes del nuevo sistema se corresponden, esto se puede observar en la Figura 30.

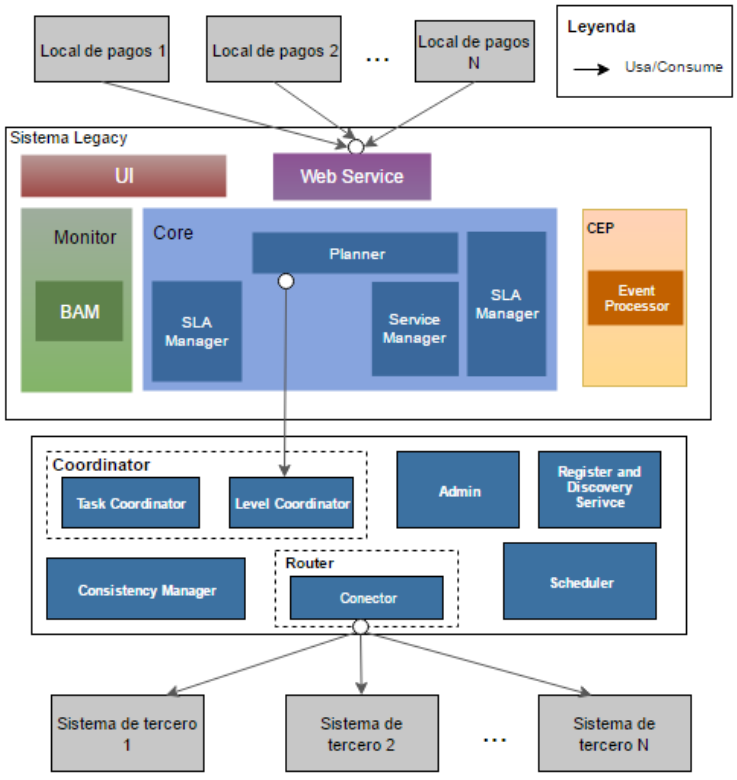

Figura 30 - Vista con el sistema Legacy

Como se puede ver en la figura se reimplementaron los siguientes componentes:

- *Coordinator: S*e corresponde con los componentes Level *Coordinator y Task Coordinator.*
- *Router*: Se corresponde con el *Connector,* pero tiene más funcionalidades, además de enviar los pedidos a los servicios de terceros, se encarga de implementar el protocolo que corresponda y realizar las trasformaciones necesarias para la invocación.

4.3.3 Vista de componentes físicos

Esta vista implica visualizar los componentes de la arquitectura y cómo se comunican entre sí, esto se puede observar en la Figura 31.

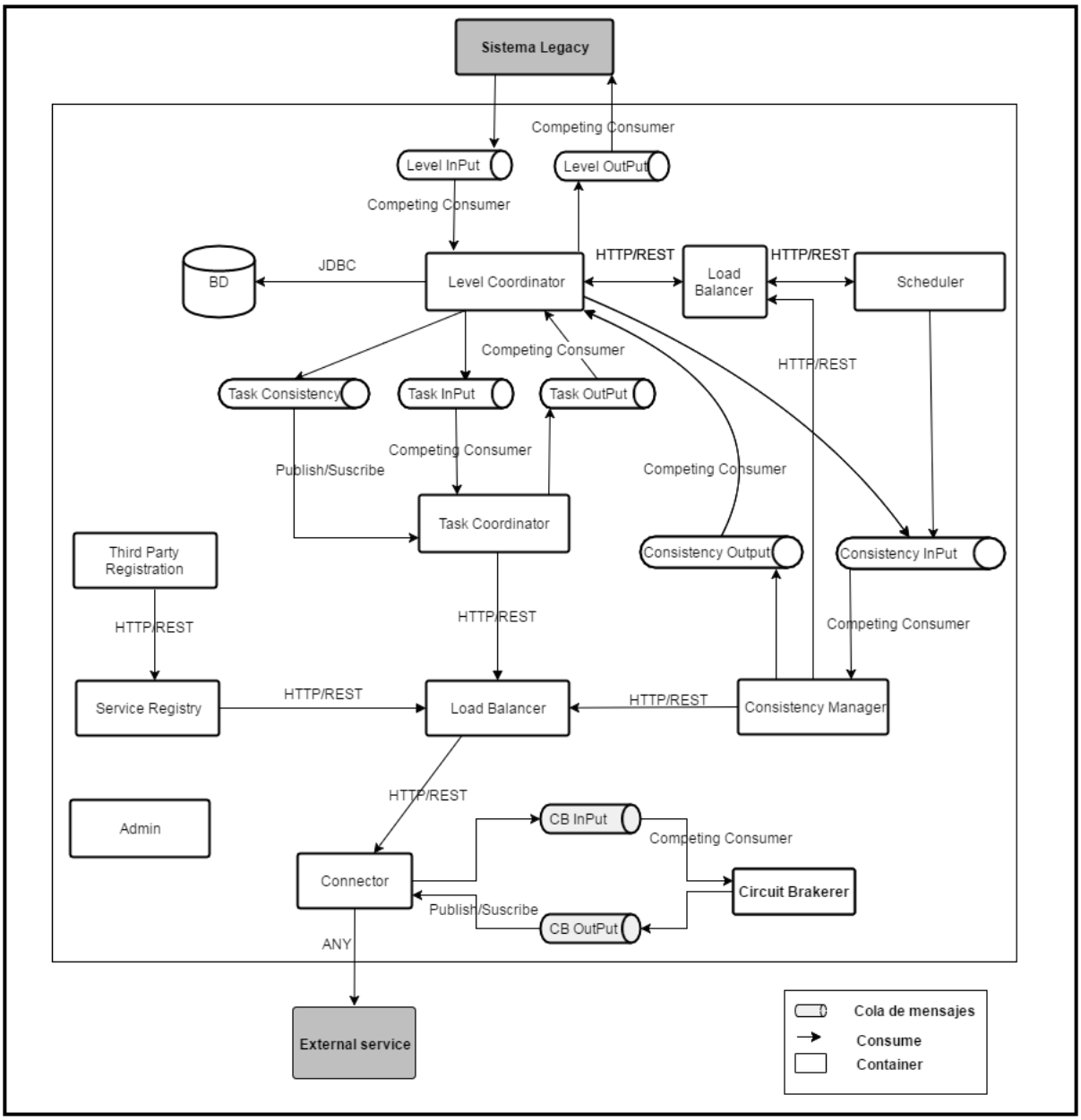

Figura 31 - Vista de componentes físicos

Todos los componentes físicos se corresponden a los lógicos descritos anteriormente, salvo el correspondiente al *Register and Discovery Service*, el cual se divide tres componentes: *Third-Party Registration*, *Service Registry* y *Load Balancer*.

En el Anexo 2 se bridan detalles de cada uno de los componentes y sus comunicaciones.

4.3.4 Vista de procesos

En esta sección se describen los casos de uso principales, los cuales son fundamentales para comprender el funcionamiento del sistema, en el Anexo 3 se describen con más detalles.

4.3.4.1 Curso normal

El flujo normal del sistema consiste en la ejecución de un nivel, donde su procesamiento termina dentro del tiempo acordado con el sistema *Legacy.*

En la Figura 32 se muestra el flujo, donde se puede observar que al final del proceso el *Level Coordinator* consume un mensaje de fin de procesamiento, pero el mismo es descartado debido a que el nivel ya fue completado y se envió un resultado al *Legacy.*

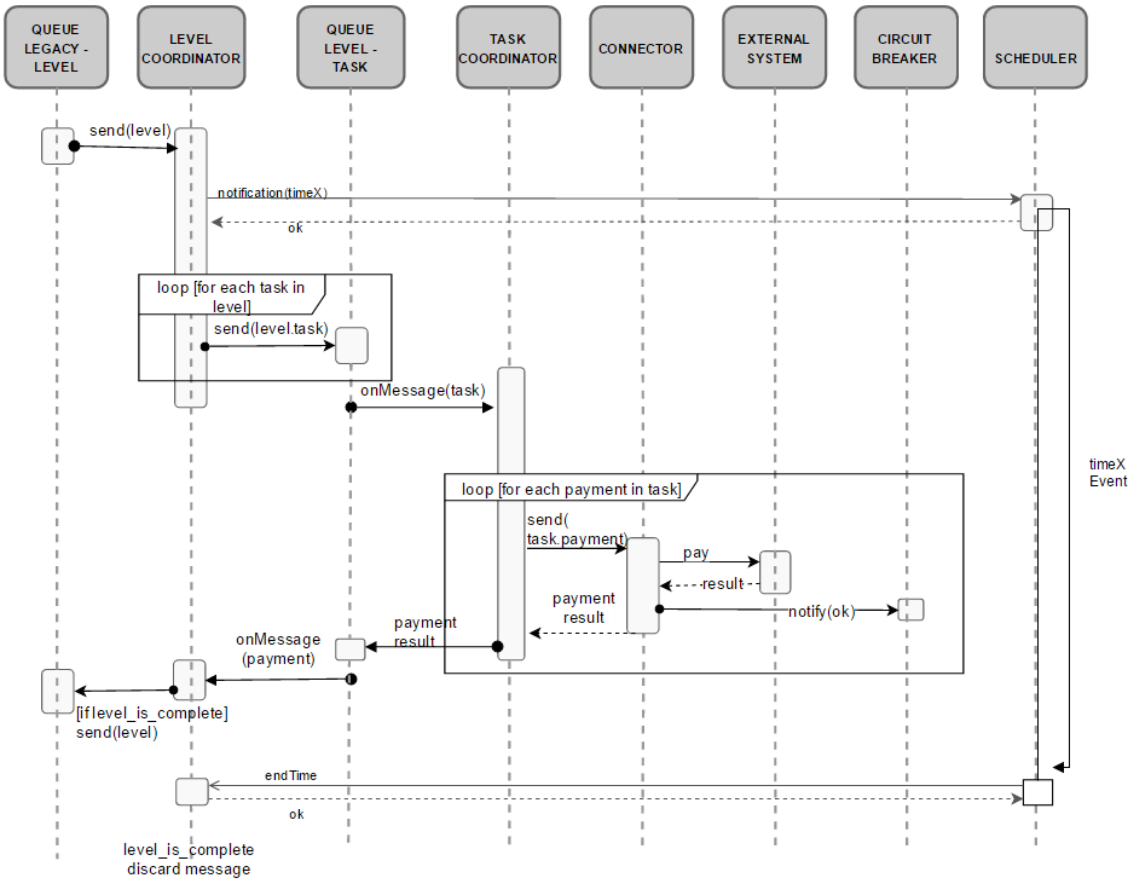

Figura 32 - Curso normal

Nota: En la comunicación hacia el Circuit Breaker hay una cola de mensajes, pero no está en el esquema para mayor claridad.

4.3.4.2 Fin de tiempo de procesamiento

Al finalizar el tiempo de procesamiento, se envía al sistema *Legacy* el estado completo del nivel. Este resultado depende de la política y el estado de cada pago.

Esto genera dos flujos diferentes dependiendo de si la política es BE o ToN, en la Figura 33 solamente se muestra el flujo cuando la política es ToN ya que es el caso más complejo. Al recibir una notificación de fin de procesamiento por parte del *Scheduler*, el *Level Coordinator* responde al *Legacy* que no se pudo ejecutar ningún pago y el sistema realizara las compensaciones que correspondan.

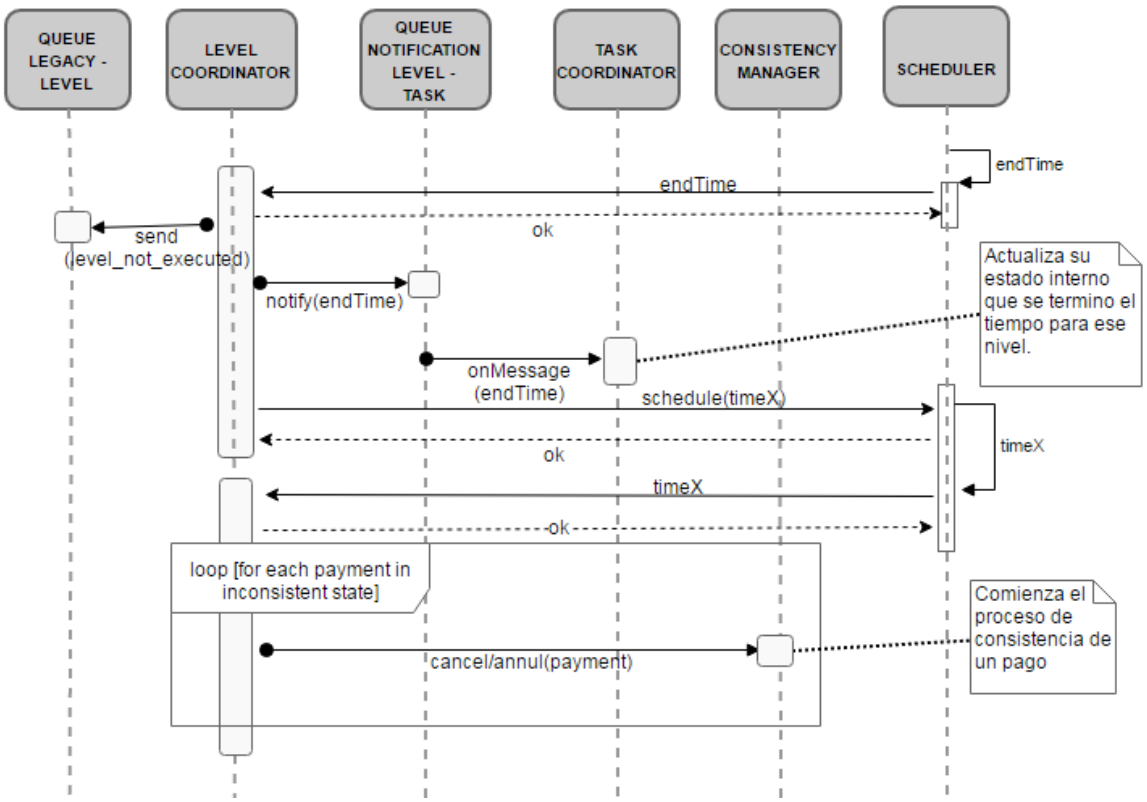

Figura 33 - Fin de tiempo de procesamiento

4.3.4.3 Timeout de Servicios Externos

Se describe el funcionamiento del sistema ante un *timeout* en la comunicación con un servicio de tercero. Al igual que en el proceso anterior se tienen dos flujos dependiendo de la política, se muestra el caso de ToN por ser más complejo.

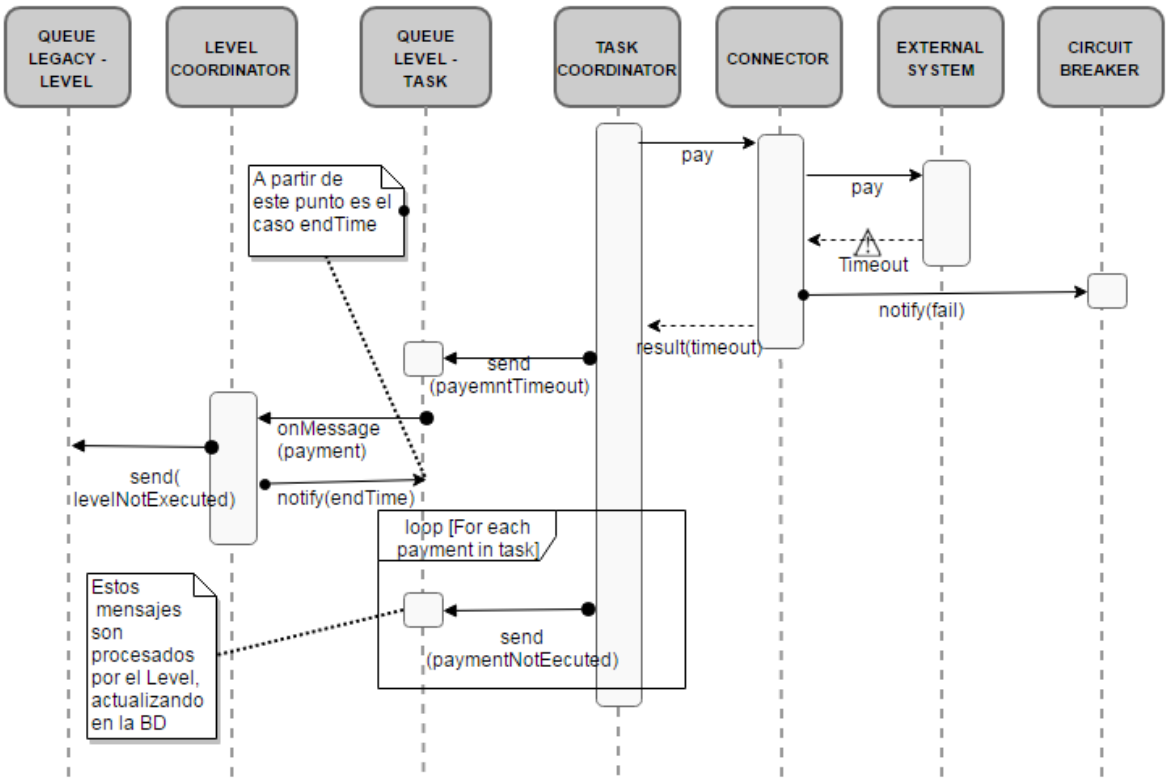

Figura 34 - Time out

4.3.4.4 Cantidad de pedidos concurrentes

Se describe el funcionamiento del sistema ante la superación del umbral establecido en el SLA con los servicios de terceros, referente a la cantidad de pedidos concurrentes. Para este caso se implementa el patrón de *Throttler* con el objetivo de no enviar pedidos que serán rechazados.

Esto se configura para cada tipo de servicio de tercero en el *Load Balancer* para la comunicación entre el *Task Coordinator* y los *Connector.* El control no se realiza en la comunicación entre el *Connector* y el *External Service* dado que se trabaja como caja negra y de esta forma se evita la reimplementación del control en cada *Connector*.

El flujo comienza cuando al invocar un *Connector* el *Task Coordinator* recibe un mensaje de error por *Throttler,* este error lo envía el *Load Balancer* al detectar que hay más pedidos concurrentes de los establecidos en el SLA. El proceso que se desencadena a continuación es similar al caso de error por *timeout,* explicado en la sección anterior, a partir del punto en el cual el *Task Coordinator* recibe un error, con la única diferencia que al recibir un error de *Throttler* el pago no se debe cancelar.

4.3.4.5 Disponibilidad de Servicios Externos

El flujo comienza cuando al invocar un C*onnector* el *Task Coordinator* recibe un mensaje de error por *Circuit Breaker*, el proceso que se desencadena es igual al de cantidad de pedidos concurrentes explicado en la sección anterior. En la Figura 35 se puede observar un ejemplo mediante el cual el Task Coordinator recibe un mensaje de error por *Circuit Breaker*.

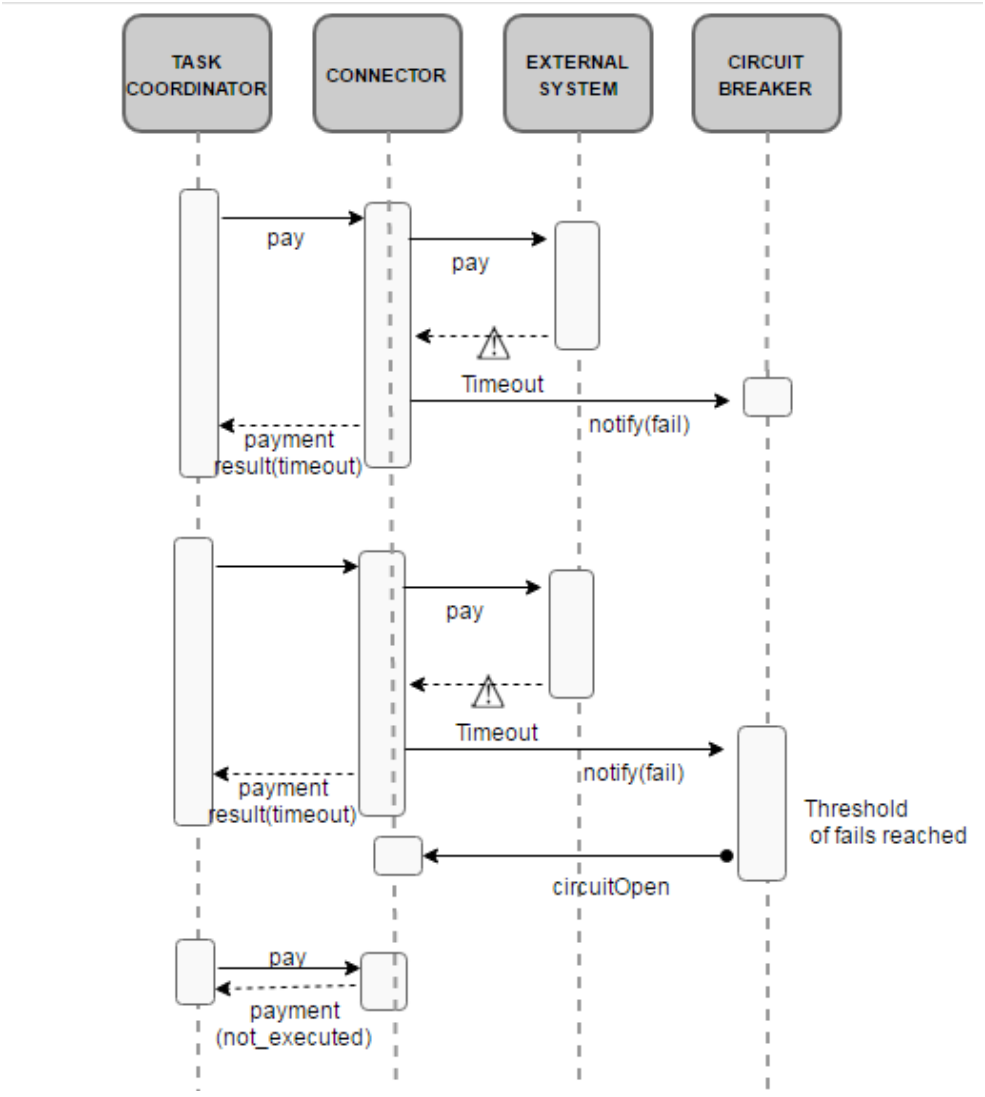

Figura 35 - Disponibilidad de Servicios Externos

4.3.4.6 Consistencia de un pago

A continuación, se describe el proceso de consistencia de un pago, el cual fue mencionado anteriormente, pero se aborda con más detalles.

Este proceso comienza cuando el *Level Coordinator* envía al *Consistency Manager* una operación de compensación (anulación o cancelación). La anulación se da cuando se quiere revertir un pago concretado en un *Sistema Externo*. En cambio, la cancelación se realiza cuando no se sabe el estado del pago, por ejemplo, cuando se recibe un error de *timeout* no se sabe en qué estado está el pago en el *Sistema Externo.*

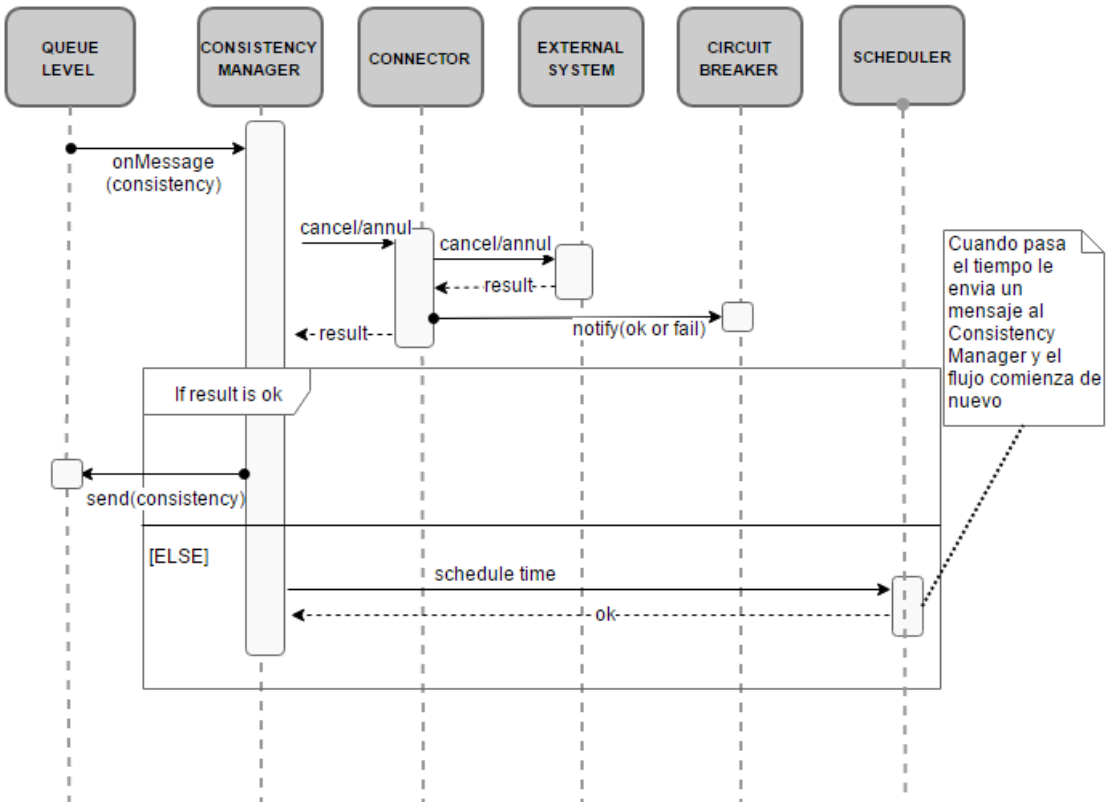

Figura 36 - Consistencia de un pago

4.3.5 Vista de despliegue

En esta vista se muestra cómo son distribuidos los componentes en la arquitectura, esto se puede observar en la Figura 37. Algunas consideraciones para comprender el esquema:

- Cada una de las “cajas” representa un *Container*.
- Las interacciones entre los componentes no se muestran, esto se puede ver en la vista de componentes físicos.
- Se muestran dos *Load Balancer,* pero en realidad se implementó como un único componente, pero es posible realizarlo de forma separada.
- Se dispone de un servidor de cola de mensajes.

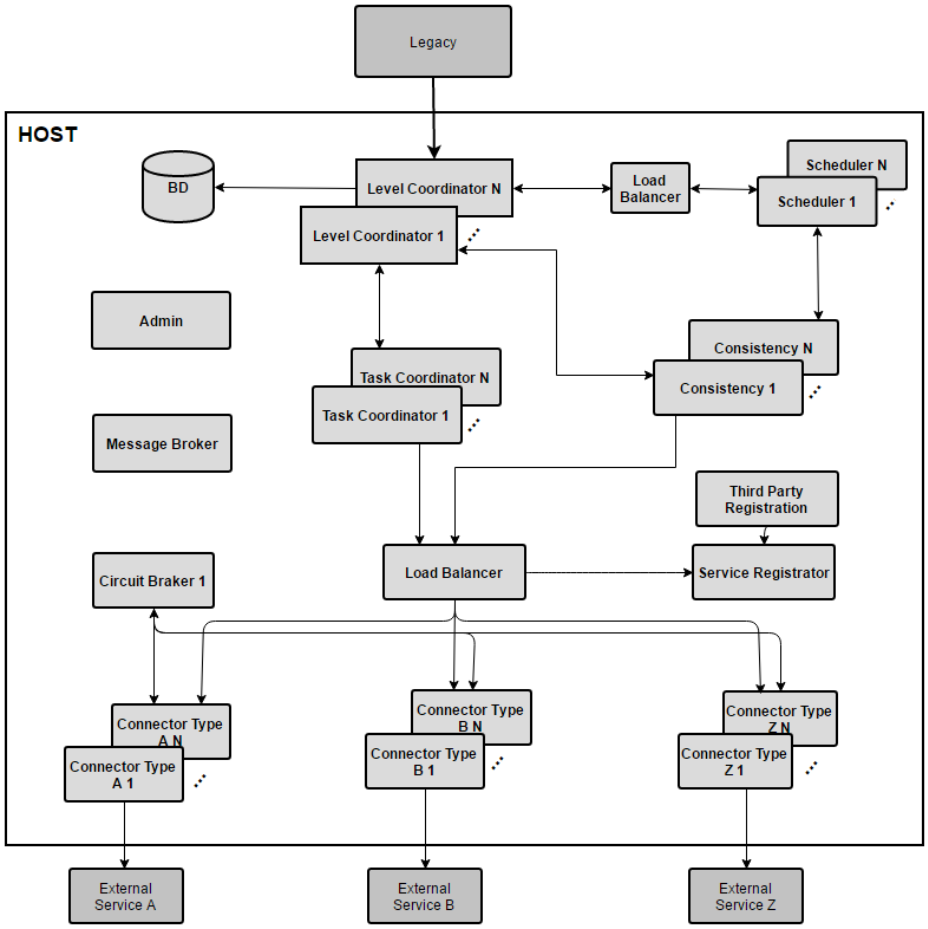

Figura 37 - Vista física

La idea es mostrar que la arquitectura soporta escalar horizontalmente los siguientes componentes: *Level Coordinator, Scheduler, Consistency Manager y Task Coordinator*. En cuanto a los *Connectors* es importante destacar que se pueden tener cualquier cantidad de tipo de *Connectors* y a su vez N instancias de cada tipo. Si bien el sistema está en un único host, es posible dividir los *Container* en cualquier cantidad de host. Para tomar esta decisión hay que evaluar la carga del sistema y realizar la división que mejor se ajuste a la situación.

4.4 Aspectos relevantes de la arquitectura

En esta sección se destacan los aspectos más importantes de la arquitectura. Algunos son inherentes por ser una arquitectura basada en Microservicios y otros son particulares.

4.4.1 Tolerancia al particionado

La tolerancia al particionado es un aspecto muy importante en Microservicios y la arquitectura se diseñó con este objetivo. Se utilizaron los siguientes patrones para lograr la tolerancia a fallas en la comunicación:

Circuit Breaker: Si los servicios externos están saturados, es deseable no enviar más pedidos, evitando la espera cuando es probable que de *timeout* y mejorando la performance global del sistema.

Esta implementación se aísla en un único componente, evitando la duplicación del código en todos los *Connectors*. Dado que se pueden escalar horizontalmente los *Connectors*, tener el *Circuit Breaker* en un componente separado permite que la máquina de estados se ejecute de forma global para todos los del mismo tipo. Si se implementara el *Circuit Breaker* en cada *Connector* y se contara con dos instancias del mismo tipo, cada una de ellas tendría su propia visión de la máquina de estados.

Time out*:* Para prevenir esperas infinitas se implementó este patrón en cada una de las invocaciones sincrónicas.

Throttler: Mediante la aplicación de este patrón se evita la invocación a los sistemas de terceros cuando se supera el umbral de cantidad de pedidos concurrentes establecido en el SLA.

4.4.2 Escalabilidad

La arquitectura fue diseñada para que pueda ser escalable en sentido horizontal y vertical de forma independiente en cada componente. A su vez, es de destacar que la escalabilidad horizontal se puede realizar de forma dinámica, configurándolo desde el componente *Admin*.

4.4.3 Disponibilidad

Al poder escalar horizontalmente cada componente, se pueden tener varias instancias. De esta forma si alguna no está disponible, el trabajo podrá ser realizado por otra instancia, permitiendo tener alta disponibilidad.

4.4.4 Extensibilidad

Un requerimiento relevante de la arquitectura es la flexibilidad en el soporte a nuevos servicios de terceros.

La solución propuesta permite agregar dinámicamente un *Connector* que se comunique con el nuevo servicio de tercero, debiendo implementar únicamente la lógica correspondiente a la invocación. Automáticamente el sistema soportará este nuevo tipo de servicio de tercero, sin la necesidad de modificar ningún otro componente ni dar de baja el sistema. También se puede configurar de forma dinámica los diferentes SLA con los servicios de tercero a través de la interfaz que expone el *Admin*.

4.4.5 Consistencia eventual

Como se mencionó en la sección 4.2.7 el sistema cuenta con una consistencia eventual, teniendo un componente dedicado a las operaciones de compensación sobre el cual se implementan reintentos ante fallas. De esta forma se garantiza que en algún momento cada nivel estará en un estado consistente.

5 Implementación

En este capítulo se describe las tecnologías utilizadas, aspectos relevantes de la implementación y por último las pruebas realizadas.

5.1 Tecnologías

Se describen las tecnologías utilizadas para la implementación de la solución, entrando en detalle en aquellas que se consideran relevantes para la comprensión del sistema.

5.1.1 Containers

Al investigar sobre las tecnologías utilizadas para la creación y administración de *Containers*, Docker [36] fue la tecnología de vanguardia mencionada en todas las referencias encontradas.

Docker es una plataforma que permite crear, correr y manejar *Containers*, creando una red de comunicación. Para cada *Container* se puede configurar la cantidad de recursos como CPU y memoria.

Un *Container* se crea a partir de una imagen, la cual se crea mediante un archivo denominado *Dockerfile*, en el cual se especifican las configuraciones que requiera la aplicación a correr.

Existe un repositorio donde se pueden almacenar y descargar imágenes, denominado *Docker Hub*. El proyecto en su totalidad esta en este repositorio, lo que permite descargar las imágenes y tener el proyecto funcionando, sin la necesidad de lidiar con temas de versiones de las tecnologías ni compilar cada componente. Los pasos para la configuración se encuentran en el Anexo 10.

Docker corre como un demonio en el host, cuando comienza su ejecución se crea una interfaz *Ethernet* virtual denominada docker0 [37].Todos los *Containers* se conectan por defecto a esta subred virtual, lo que permite establecer la comunicación con el host y con los demás *Containers*. El demonio de *Docker* escucha por defecto en un *socket*, al cual pueden enviarse comandos para la administración de las imágenes y *Containers*.

5.1.2 Lenguaje de programación

Como lenguaje de programación se utilizó Java en todos los componentes, ya que el sistema Legacy lo utiliza y a su vez, ningún componente requirió el uso de otro leguaje.

5.1.3 Servidor de aplicaciones

Se utiliza Wildfly [38] como servidor de aplicaciones ya que existe una imagen oficial de esta tecnología en Docker. Además, el equipo contaba con experiencia en el manejo de ésta.

5.1.4 Base de datos

Se utiliza MySQL [39] como manejador de base de datos, dado que el equipo ya tenía experiencia en la utilización de la herramienta y cumple con el fin requerido. Para abstraerse

del manejo de la base de datos, se utilizó Hibernate [40], un framework para establecer la correspondencia entre las clases de Java y las tablas de la base de datos.

5.1.5 Comunicación sincrónica

Para este tipo de comunicación se utilizó el protocolo REST sobre HTTP con el patrón *request/response*, ya que es un método liviano de comunicación, esto va a favor de la arquitectura de Microservicios.

5.1.6 Comunicación asincrónica

Como *broker* de mensajería se utilizó RabbitMQ [41], dado que se dispone de una imagen oficial en Docker de *RabbitMQ*. Para la creación y configuración de los *listeners* que reciben en las diferentes colas se utilizó *Spring Framework* [42].

Se manejó la posibilidad de utilizar un modelo de eventos para este tipo de comunicación mediante *Akka*. Luego de investigar la tecnología se decidió descartarla, debido a que se debería utilizar en todos los componentes y el sistema quedaría acoplado al uso de esta tecnología, esto va contra los principios de la arquitectura de Microservicios; a su vez, esta tecnología tiene una gran curva de aprendizaje.

5.1.7 Registro y descubrimiento de servicios

Para implementar los patrones *Third-Party Registration* y *Server-Side Discovery*, se procedió a investigar sobre las tecnologías existentes. Se encontraron cuatro tecnologías principales que son descriptas a continuación:

- *ZooKeeper*: es la tecnología más antigua de este tipo, teniendo como gran ventaja su madurez en el mercado, pero como contraparte es una tecnología compleja que tiene una curva de aprendizaje considerable. [43].
- *Etcd*: Permite un almacenamiento consistente distribuido del tipo clave/valor. Es mejor opción que *ZooKeeper*, ya que es más simple de configurar, pero no existe mucha documentación. [44]
- *Eureka*: Esta tecnología tiene el respaldo de ser desarrollada y utilizada por un referente en el tema, *Netflix*, pero la gran desventaja es la poca documentación. A su vez está enfocada para ser utilizada en la nube de *Amazon*. [45]
- *Consul*: Es una herramienta para el descubrimiento y configuración de servicios. Provee cuatro funcionalidades principales: Descubrimiento de servicios, almacenamiento del tipo clave/valor, *health checking* y *multi datacenter*. [46]

Se decide utilizar *Consul* debido a que existe más documentación en comparación a las demás tecnologías y se integra con *Docker*. A su vez, cuenta con una interfaz gráfica que permite la administración.

Se utilizan otras herramientas que integradas a *Consul* permiten realizar las actividades de registro y balanceo, estas se explican a continuación:

- **Registrator:** Implementa el patrón *Third-Party Registration*, para ello registra los *Dockers* y sus servicios en *Consul*, detectando la creación de nuevos *Dockers*. [47].
- **Consul-template:** Permite crear archivos con valores obtenidos en Consul, los mismos son actualizados dinámicamente ante cambios [48]. Para la creación de los

archivos se utiliza el formato *Go-template* [49] con una extensión propia para acceder a parámetros de Consul.

- **Nginx:** Es un balanceador de carga y para configurarlo dinámicamente se utilizó *Consul- template* [50].

En la Figura 38 se puede observar cómo interactúan estas tecnologías. *Registrator* detecta la creación de nuevos *Docker*s y sus servicios para luego registrarlos en *Consul*. *Consul-template* reacciona ante los cambios en *Consul*, esto se refleja en la configuración de *Nginx*, el cual es utilizado para realizar el balanceo entre los diferentes servicios.

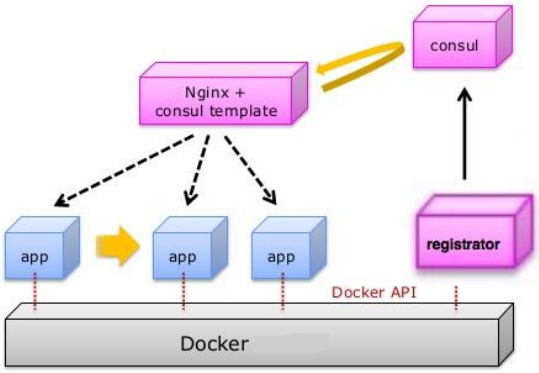

Figura 38 – Interacción ente Nginx-Consul y registrator

5.1.8 Gestión de dependencias

Para la gestión de dependencias se utilizó *Apache Maven* [51]. Este es un software que permite crear, gestionar proyectos y contiene la información centralizada en un archivo del tipo *Project Object Model (POM).* Dentro de sus funcionalidades más importantes están compilación, empaquetado y gestión de dependencias. En la solución se utilizó para compilar y empaquetar los distintos proyectos en los componentes, además de gestionar las dependencias.

5.1.9 Control de versiones y repositorio

Para el control de versiones se utilizó Git, es un sistema distribuido de control de código fuente o Source Code Management(SCM) que permite auditar, controlar, deshacer, administrar y manejar diferentes versiones sobre el código fuente. Esto fue utilizado para gestionar el código de todos los componentes [52]. Como repositorio de código se utilizó BitBucket [53] debido a que es privado y se encuentra en la nube y como repositorio de imágenes de *Docker* se utilizó *Docker Hub.*

5.1.10 Monitoreo

Para el monitoreo se utilizaron dos herramientas que se explican a continuación:

- *NewRelic*: Es un servicio de monitoreo de aplicaciones web/móviles y servidores, brinda un completo informe sobre el comportamiento del sistema en una línea de tiempo. Esta herramienta se integró con *Docker* para obtener estadísticas del uso de CPU y memoria de cada *Container*. En el Anexo 9 se explica cómo se realiza la configuración. [16]

- *DockerUI*: Es una interfaz gráfica que se integra con *Docker* mediante la cual se pueden visualizar y administrar los *Container*s que se tienen en el sistema. [54]

5.1.11 Testing

Para las distintas etapas de *testing* del sistema se utilizaron diferentes herramientas que son explicadas a continuación:

- *Junit*: Es un *framework* para la ejecución de test unitarios en java. Fue utilizado para evaluar el comportamiento de métodos complejos, así como también, permitió verificar el correcto funcionamiento del sistema ante cambios en el código. [55]
- *Postman*: Es una herramienta que permite probar servicios que exponen una API REST. Fue utilizada en las pruebas de integración para invocar directamente a los diferentes servicios, permitiendo realizar pruebas puntuales; así como también para ejecutar las pruebas end to end.
- *RabbitMQ*: La interfaz web de *RabbitMQ* brinda una opción para enviar un mensaje directamente a una cola de mensajes. Esto permitió la simulación de los componentes que envían a la cola, pudiendo comprobar el comportamiento del sistema desde ese punto.
- *SoapUI*: Es una aplicación que permite probar de forma ágil *web services* SOAP y REST. Permite la realización de test unitarios, de performance y de carga. Se pueden crear grupos de test, configurando la cantidad de hilos que los ejecutan y el tiempo de duración del mismo. Esta herramienta se utilizó para probar los servicios SOAP y realizar las pruebas de performance [56].

5.2 Aspectos relevantes de implementación

En esta sección se describen características de la implementación que se consideran relevantes para entender el comportamiento del sistema.

5.2.1 Level Coordinator

El *Level Coordinator* es el encargado de recibir los niveles y enviar los resultados al sistema *Legacy*. A su vez se encarga de persistir los datos y enviar a consistir los pagos cuando sea necesario.

5.2.1.1 Estados

Para representar los estados de un nivel, se modelan dos máquinas de estados, una relacionada a los estados que ve el sistema *Legacy* y otra para los internos.

En la Figura 39 se muestra la máquina de estados que ve el sistema *Legacy*. Los estados posibles son *PROCESSING, ERROR y OK*, los dos últimos son finales.

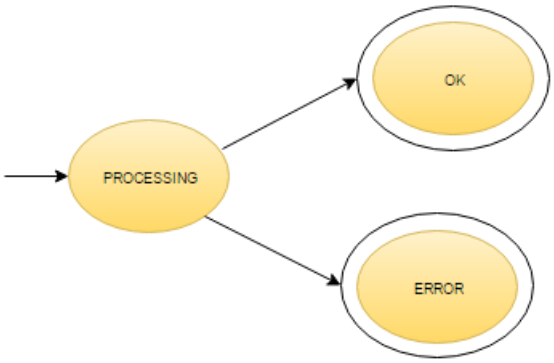

Figura 39 - Máquina de estados para estados del sistema Legacy

Al ingresar un nivel al sistema, comienza en el estado *PROCESSING* hasta que se envía el resultado. El resultado enviado hacia el sistema *Legacy* dependerá del estado de los pagos y la política. La siguiente tabla muestra el estado final enviado en función de la política y los resultados de los pagos del nivel.

Política\Estado final	OK	ERROR
ToN	Todos los pagos OK	Al menos 1 pago ERROR
BE	Al menos 1 pago OK	Todos los pagos ERROR

Debido a que se deben realizar compensaciones, resulta necesario modelar internamente de otra forma los niveles. Los posibles estados internos son:

- *PROCESSING*: los pagos del nivel están siendo procesados.
- *EVENTUAL_CONSISTENCY*: existen pagos que deben ser compensados.
- *CONSISTENCY*: todos los pagos del nivel se encuentran en estado consistente.

La Figura 40 muestra la máquina de estados para los estados internos:

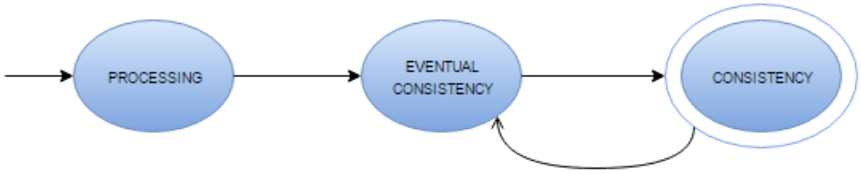

Figura 40 - Máquina de estados para estado internos

Como se observa, se puede pasar del estado *CONSISTENCY* a *EVENTUAL_CONSISTENCY,* este caso ocurre cuando el nivel está en un estado consistente y llega desde un *Task Coordinator* un resultado diferente al enviado hacia al *Legacy.*

Del mismo modo que se tienen estados externos e internos para el nivel, se cuenta con estados externos e internos para los pagos. Esto permite enviar el resultado de un pago como no ejecutado e internamente realizar la compensación.

En la Figura 41 se observa los estados de los pagos internos cuando el estado del nivel es *PROCESSING.* Como se puede observar es posible pasar del estado *PROCESSED_OK* a *TO_ANNUL,* esto puede ocurrir en el siguiente escenario: se tiene un nivel con política ToN y un pago falla, entonces los pagos ejecutados correctamente deben de ser anulados.

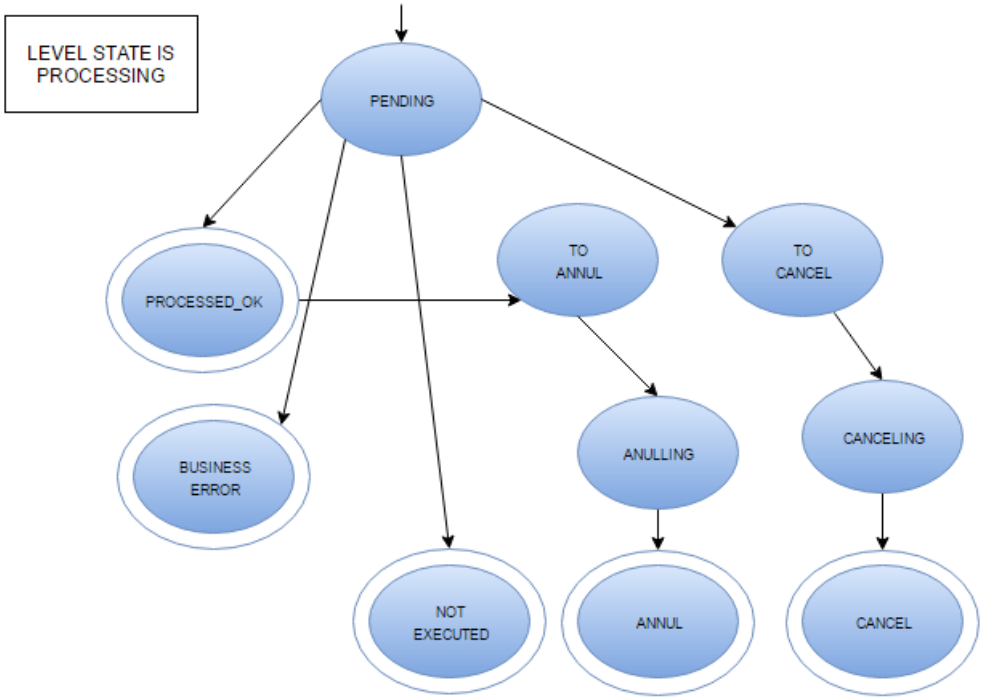

Figura 41 - Estados de los pagos con nivel PROCESSING

Luego de enviar el resultado al *Legacy*, el estado del nivel puede ser, *CONSISTENCY* o *EVENTUAL_CONSISTENCY.* A su vez el resultado de los pagos que se envían pueden corresponder a diferentes estados internos, esta correspondencia se muestra en la Figura 42. Se puede observar que tanto para *PROCESED_OK* y *BUSINESS_ERROR* la correspondencia es uno a uno, mientras que, si se envió *NOT_EXECUTED* el estado interno tiene varias posibilidades, esto refleja la consistencia eventual del sistema, donde se envía *NOT_EXECUTED,* pero internamente se están realizando operaciones de compensación.

Es de destacar que, si se termina el tiempo de procesamiento y hay pagos en el estado *PENDING,* al no saber que ocurrió con los mismos el sistema los cancelará. Esto permite que el sistema soporte la caída de un Task Coordinator, porque si no se recibió el resultado que estaba procesando el sistema se encargará de mantener la consistencia.

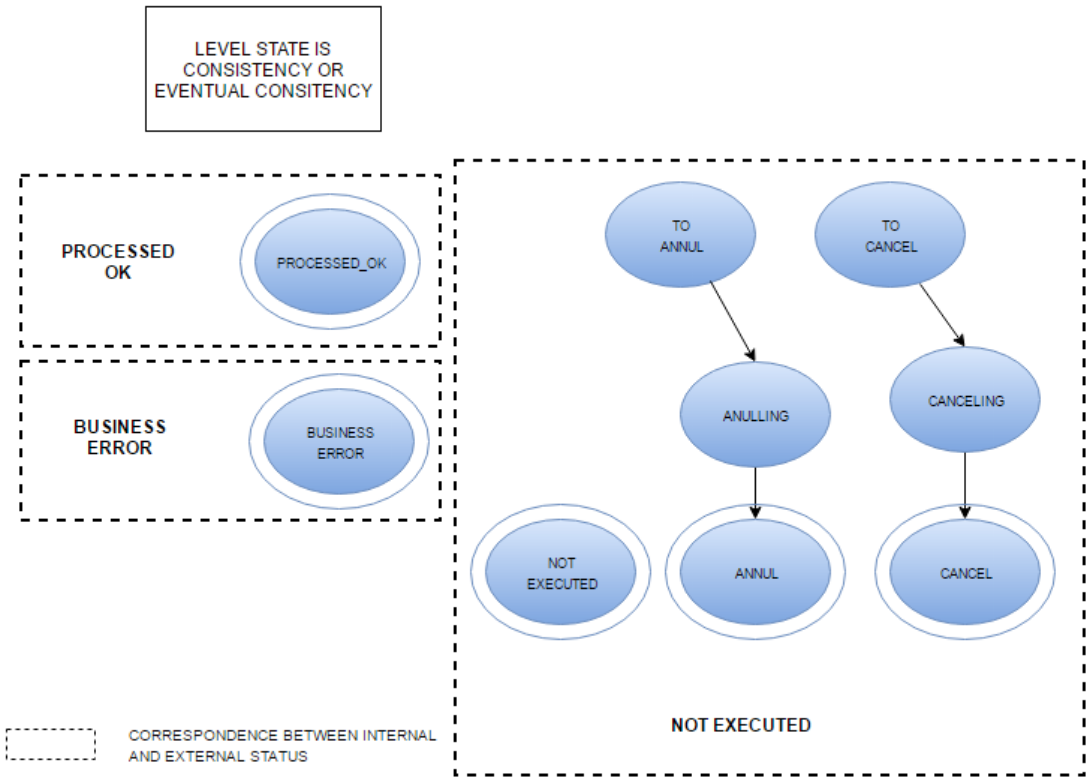

Figura 42 - Estados de los pagos con nivel CONCISTENCY o EVENTUAL_CONSISTENCY

El Anexo 4 presenta un ejemplo para comprender los estados del nivel y los pagos.

5.2.1.2 Generación de identificadores

Para poder identificar las diferentes compensaciones enviadas hacia el *Consistency Manager,* el *Level Coordinator* asocia cada compensación a un identificador único generado internamente, pudiendo tener de esta forma una traza sobre cada una de las compensaciones realizadas.

5.2.1.3 Workers

Para cada funcionalidad el *Level Coordinator* posee un *pool* de hilos los cuales reciben trabajos a procesar, denominados *workers*. Esto permite que cada *worker* procese distintas actividades, además de no bloquear el resto de las operaciones que se encuentran procesando en el componente. A continuación, se describen los *workers* para el *Level Coordinator*:

- Level Worker: Se encarga de procesar un nivel, dividirlo en las tareas y enviarlo al *Task Coordinator*. Además, asocia el modo de ejecución de la tarea según la política recibida: si la política es ToN, se mapea con el modo de ejecución *Stop on Error* (SoE), en otro caso es *Continue on Error* (CoE).
- Result Worker: Se encarga de procesar los resultados de los pagos enviados por el *Task Coordinator*. A su vez, verifica si el nivel está listo para enviar la respuesta al sistema *Legacy* y delegar al *Consistency Worker* en caso que sea necesario.
- Consistency Worker: Se encarga de enviar y recibir los mensajes al *Consistency Manager*.

- End Time Worker: Se encarga de recibir la notificación de fin de tiempo de procesamiento y realizar las acciones correspondientes, como por ejemplo enviar la notificación de fin de tiempo de procesamiento a los *Task Coordinator* en caso que el nivel se esté ejecutando.

5.2.1.4 Base de datos

En la base de datos se almacena la información relevante del nivel y cada uno de sus pagos. Se tienen tres entidades, cuyo modelo se puede observar en la Figura 43. La descripción de cada campo de las tablas se realiza en el Anexo 5.

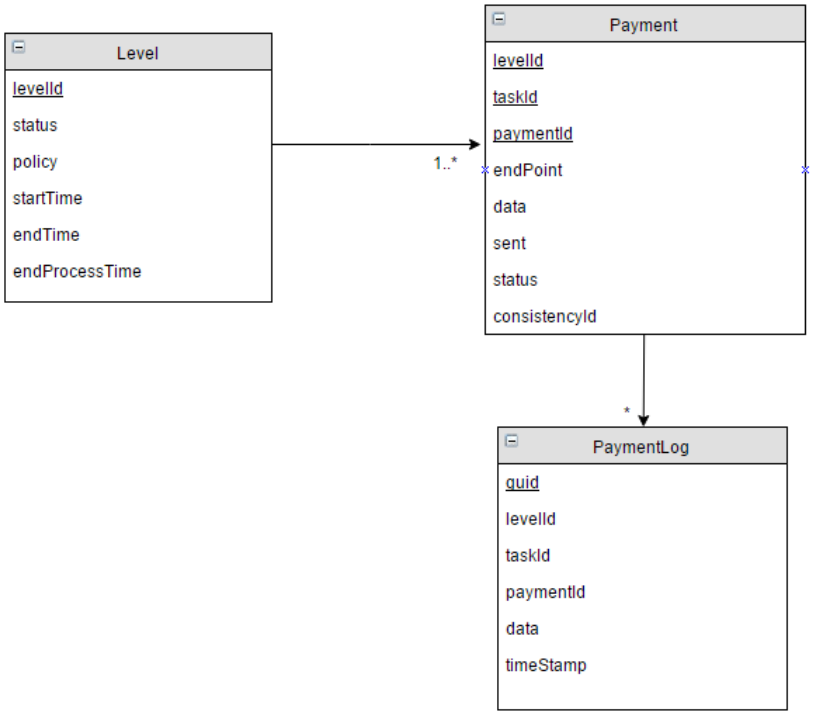

Figura 43 - Modelo de la base de datos

5.2.2 Task Coordinator

En esta sección se describen los aspectos relevantes de la implementación del componente *Task Coordinator*.

5.2.2.1 Modos de ejecución

Existen dos modos de ejecución, *Stop on Error(SoE)* y *Continue on Error(CoE)*. Si el modo de ejecución es SoE alcanza con que un pago dé error para que se detenga la ejecución de la tarea y se envíe el resultado al *Level Coordinator*. En el caso de CoE no se detiene la ejecución de la tarea si falla un pago.

5.2.2.2 Workers

Al igual que el *Level Coordinator*, el *Task Coordinator* posee un w*orker* por cada funcionalidad, los *workers* implementados son:

- Task Worker: Se encarga de procesar una tarea ejecutando sus pagos de forma serial, enviando los resultados de los pagos al *Level Coordinator* a medida que se obtienen.
- End Time Worker: Recibe eventos de fin de tiempo de procesamiento y modifica el estado de la cache para el nivel recibido.

5.2.2.3 Cache

Este componente implementa una cache interna que mantiene el estado de los niveles que está procesando, esta información es compartida y actualizada por los hilos de ejecución. Es consultada antes de ejecutar cada pago con el objetivo de validar si se puede realizar la invocación, la invocación no se puede realizar si el estado del nivel fue cambiado por el *End Time Worker* al estado correspondiente a la finalización del tiempo de procesamiento.

En el Anexo 6 se describe con mayor detalle el proceso que siguen las tareas dentro del *Task Coordinator* y los estados de la cache.

5.2.3 Connector

Este componente se encarga de la comunicación con el sistema externo, es el único que conoce el protocolo de comunicación y la interfaz. A su vez se encarga de transformar el mensaje recibido al formato que el sistema externo espera. Al iniciar su ejecución el *Connector* se registra en el *Circuit Breaker* para configurar los parámetros del mismo.

Los Connector implementan una interfaz la cual contiene los métodos *pay*, *cancel* y *annul*, además tiene un módulo que se encarga de la comunicación con el *Circuit Breaker*.

Si se desea comunicar con un nuevo sistema externo se debe implementar un nuevo *Connector* partiendo del *Connector Template*, el cual contiene la comunicación resuelta con el *Circuit Breaker*. Además, contiene los servicios REST definidos para la comunicación con los *Task Coordinator*. Lo único a implementar es la invocación al servicio externo y las transformaciones de los datos que se requieran.

5.3 Testing

En esta sección se describirán los tipos de pruebas que se realizaron sobre el sistema.

5.3.1 Testing funcional

- *Pruebas unitarias*: se realizaron pruebas unitarias con *JUnit* en las funcionalidades de la base de datos en el *Level Coordinator*.
- *Pruebas de integración*: se realizaron pruebas en la comunicación de los componentes para detectar defectos en las interfaces, simulando que los servicios respondan lento o no estén disponibles.
- *Pruebas de componentes*: se realizaron pruebas simulando el punto de entrada a los componentes, validando el flujo a partir de este punto. A partir de estas pruebas se detectó un defecto, el cual consistía en cargar la configuración de las colas de mensajes cada vez que se quería mandar uno. Para solucionarlo se creó una clase *Singleton* que carga la configuración cuando inicia el componente. Luego de este cambio se detectó un descenso importante en el tiempo de ejecución de los pagos.

- *Pruebas end to end*: se realizaron pruebas del sistema en su totalidad, para los diferentes procesos identificados. En el Anexo 7 se brinda mayor detalle de estas pruebas.

5.3.2 Test de performance

En la presente sección se describen los test de performance que se realizaron sobre el sistema implementado, además las pruebas que se realizaron para validar la tolerancia a fallas en la comunicación con los servicios externos.

Debido a la naturaleza asincrónica de la solución se dificultó la realización de las pruebas de performance, esto se debe a que es más complejo controlar la cantidad de ejecuciones en paralelo que se tienen dentro del sistema.

Los objetivos de las pruebas son:

- Encontrar la máxima capacidad de procesamiento de niveles de forma concurrente.
- Validar que al escalar el sistema se tiene mayor capacidad de procesamiento.
- Validar la tolerancia a fallas del sistema.

5.3.2.1 Datos generales de las pruebas

En todas las pruebas se cuenta con las siguientes configuraciones,

Servicios de Terceros

- BPS, interfaz REST
- UTE, interfaz SOAP
- OSE, interfaz SOAP

Cantidad de instancias

- Connector: 3 uno por cada servicio externo
- Los demás componentes con una instancia

Infraestructura

Las pruebas fueron realizadas utilizando un host para el despliegue de todo el sistema y otro con las mismas características que oficia de cliente. Dichas computadoras tienen las siguientes características:

- Procesador Intel Core i7-2600 CPU @ 3.40GHz × 8
- Memoria RAM 16 GiB
- Disco 1 Tb

El host cliente invoca al sistema mediante SoapUI. Se monitoreo el uso de CPU y memoria de este host, los cuales se mantuvieron estables y en valores normales. La Figura 44 muestra gráficamente como es el despliegue total del sistema y cómo interactúan.

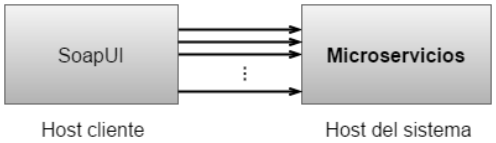

Figura 44 - Despliegue del sistema

5.3.2.2 Test de carga

En esta prueba se desea encontrar la máxima cantidad de pedidos concurrentes que puede soportar el sistema. Para ello se busca encontrar la máxima cantidad de hilos que envían pedidos al sistema de forma concurrente, respetando el tiempo de procesamiento acordado en el SLA. Para encontrar esta cantidad máxima se aumentó la cantidad de hilos concurrentes hasta que el sistema no pueda procesar los pagos en el tiempo de procesamiento establecido (5 segundos).

La cantidad máxima de pagos concurrentes en el sistema fue de 110, utilizando 22 hilos clientes que enviaron pedidos cada un segundo durante cinco minutos. Se procesaron un total de 6600 pagos.

Los tiempos de procesamiento fueron los siguientes:

Tiempo máximo	Tiempo mínimo	Tiempo promedio	Mediana
0:00:04.83	0:00:00.80	0:00:01.81	0:00:01.85

Se puede apreciar que el tiempo promedio y la media se mantienen bajos, mientras que el tiempo máximo está muy cerca del tiempo de procesamiento acordado en el SLA (5 s).

Se definen tres franjas de tiempo en función de las métricas obtenidas, y se grafica la cantidad de pagos en cada franja, las franjas son:

- Entre tiempo mínimo y promedio
- Entre tiempo promedio y mediana

Entre tiempo mediana y máximo

Figura 45 muestra la cantidad de pagos que se encuentran en cada franja de tiempo. En este caso la mayoría de los pagos se procesaron entre la media y el tiempo máximo, esto es un indicio que el sistema se encuentra procesando cerca de su límite de procesamiento.

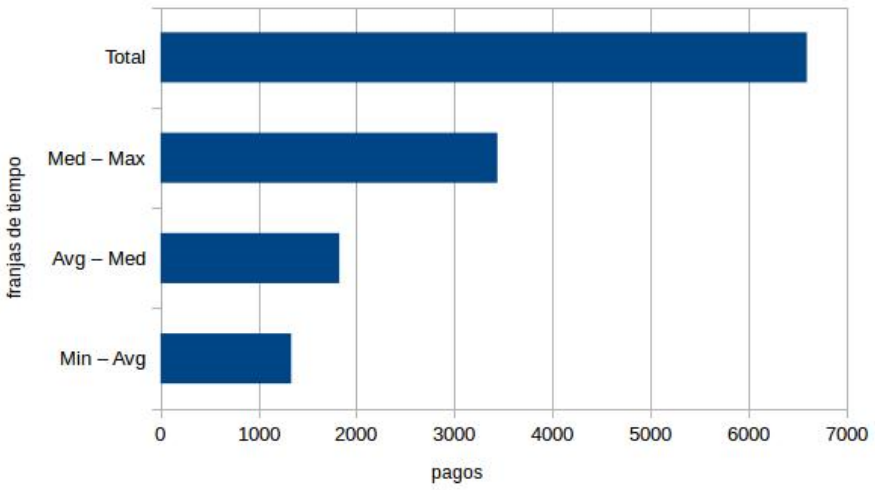

Figura 45 - Cantidad de pagos por franja

En la Figura 46 muestra la cantidad de pagos en procesamiento a lo largo del tiempo. En este caso la cantidad de pagos concurrentes en el sistema oscila entre 60 y 100 en la mayor parte del tiempo, teniendo algunos picos mayores a 100 y menores a 60.

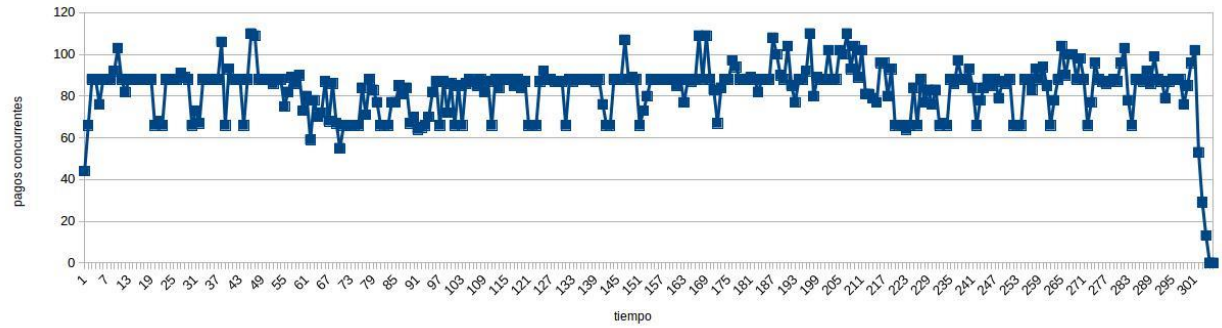

Figura 46 - Pagos concurrentes

La Figura 47 muestra el top 5 de consumo CPU y memoria de los componentes. Dicho consumo de CPU no alcanza el 30% del total y la memoria es casi constante.

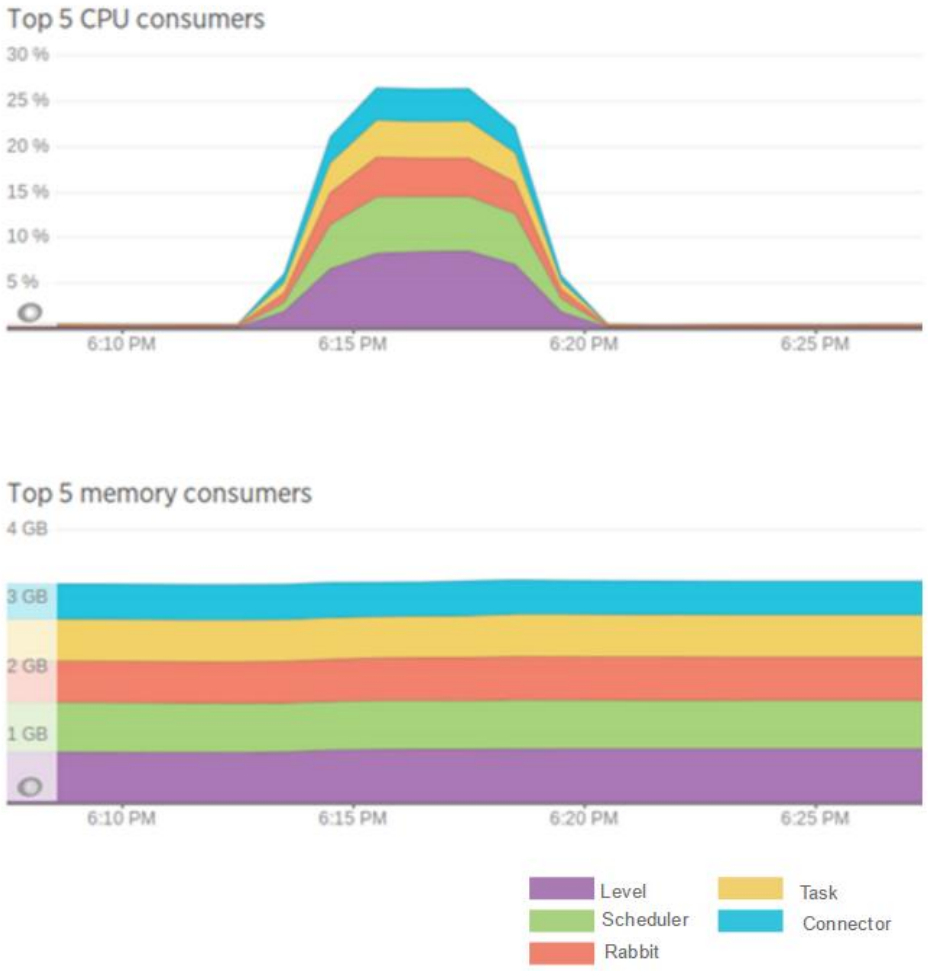

Figura 47 - Top 5 de recursos

Debido a que el *Level Coordinator* es el componente que consume más CPU y tiene mayor procesamiento ya que accede a la base de datos, envía la respuesta al sistema *Legacy* y contiene la lógica de la compensación de los pagos, se decidió replicar dicho componente para validar que éste es el cuello de botella del sistema. Luego de replicarlo se realizaron las pruebas aumentando la cantidad de hilos a 24, los resultados de estas pruebas se encuentran en el Anexo 8. Se procesaron todos los pedidos en un tiempo máximo de 3.15 segundos lo cual significa que el sistema tiene aún mayor capacidad de procesamiento.

Se concluye que el cuello de botella es el *Level Coordinator*, replicando dicho componente el sistema tiene mayor capacidad de procesamiento.

5.3.2.3 Test de tolerancia a fallas

En este caso se muestra cómo el sistema reacciona ante fallas en la comunicación con un servicio externo. Para ello se simula que el sistema externo OSE tarda en responder los pagos, cancelaciones y anulaciones. Esta demora puede ser debido a que la red no está funcionando bien o que el sistema externo está tardando en responder.

Se tomaron datos en dos escenarios, el primero, dejando inactivo el componente *Circuit Breaker* y el segundo, utilizándolo. Luego se compararon los resultados de tiempos obtenidos.

Para simular que el servicio OSE está respondiendo lento, se configuró la instancia de *Connector* que se comunica con el mismo con un *timeout* de un minuto, y el servicio externo OSE con un tiempo de respuesta de dos minutos. Esto tendrá como consecuencia que todos los pagos enviados a dicho servicio darán *timeout*, por lo que el *Connector* enviará al *Circuit Breaker* la notificación de falla del servicio. Los tiempos de procesamiento de UTE y BPS se mantienen en 1 y 1.25 segundos respectivamente.

En esta prueba se utilizaron 10 hilos los cuales cada uno ejecuta tres pagos, uno para cada servicio externo (UTE, OSE, ANTEL), cada 35 segundos.

En la Figura 48 se observa el tiempo de procesamiento para los pagos, sin utilizar el componente Circuit Breaker.

Como era de esperar, todos los pagos hacia el servicio OSE tardan un minuto en responder, por otro lado, los pedidos hacia los servicios externos UTE y BPS tardan en el entorno del segundo en ser procesados. A partir de estos resultados se puede afirmar que la demora del servicio OSE no afecta el procesamiento de los demás servicios.

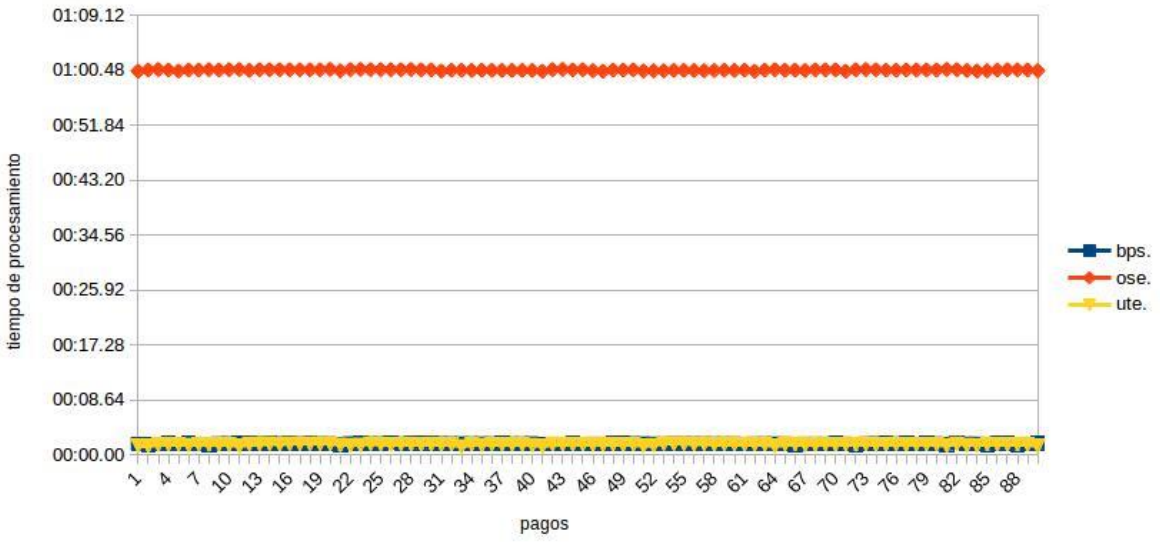

Figura 48 - Tiempo de procesamiento de pagos

Para la prueba utilizando el componente *Circuit Breaker*, este se configuró que con 5 fallas continuas se abre el circuito, luego de abierto el circuito permanecerá en este estado por 60 segundos. Recordar que este componente implementa el patrón *Circuit Breaker* por cada servicio externo. La Figura 49 muestra el tiempo de ejecución de los pagos para esta prueba. En la misma se puede observar la utilidad del componente Circuit Breaker en la diferencia del tiempo de ejecución cuando el circuito está abierto y cerrado. Cuando se encuentra abierto, el Connector responde inmediatamente ya que no realiza la invocación al servicio externo, en cambio cuando se encuentra cerrado, el Connector realiza la invocación al servicio externo demorando un minuto en responder. Al igual que en la prueba anterior las invocaciones a los servicios UTE y BPS no se ven afectas por el estado del circuito del servicio OSE.

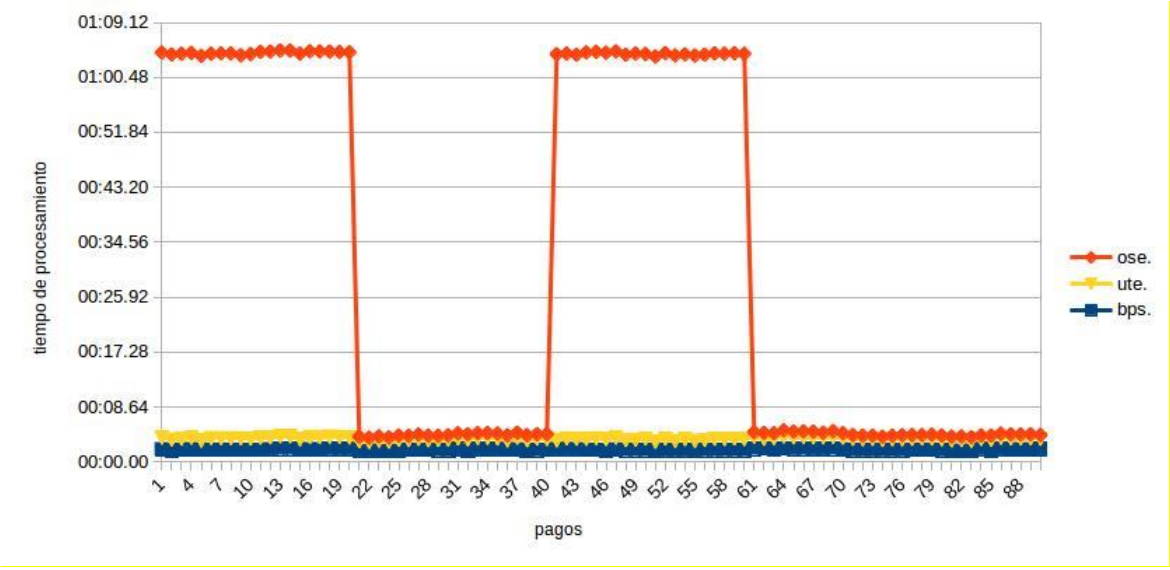

Figura 49 - Tiempo de procesamiento de pagos

En este caso de prueba se pudo demostrar que utilizando el componente Circuit Breaker se logra detectar fallas en la comunicación con los sistemas externos y así no realizar invocaciones que probablemente fallen, logrando así un menor tiempo de respuesta en estos casos.

6 Gestión de proyecto

En este capítulo se describe el proceso que se siguió durante la realización del proyecto, el mismo se divide en varias fases:

Fase 1

- **Marco teórico**: Estudio de la arquitectura SOA y sus principios; por otro lado, la arquitectura de Microservicios, los beneficios y desafíos que presenta.
- **Documentación**: Documentación de lo estudiado.

Fase 2

- **Estudio de la realidad**: Se estudió la realidad del caso de estudio de referencia sobre el cual se quiere aplicar el patrón de arquitectura de Microservicios. También se analizó la arquitectura y el diseño de la solución del sistema *Legacy*. Luego, se intentó comprender su código fuente con el objetivo de reutilizarlo al momento del rediseño.
- **Requerimientos**: Identificación de los requerimientos del sistema y las restricciones que presenta, definiendo el primer alcance del sistema.
- **Documentación**: Se realizó una documentación sobre las principales características del sistema *Legacy,* requerimientos y restricciones.

Fase 3

- **Diseño:** A partir del alcance definido, se realizó el primer diseño.
- **Prototipos:** Realización de prototipos para mitigar los riesgos en etapas tempranas y comprender el funcionamiento de las tecnologías.
- **Documentación:** Se documentó el diseño y lo investigado sobre las tecnologías.

Fase 4

- **Diseño:** Modificación del diseño en base a los prototipos realizados.
- **Prototipos:** Se realizó un prototipo integrando los componentes de la arquitectura, utilizando los prototipos anteriores.
- **Documentación:** Se documentó la nueva arquitectura y se realizó una presentación del avance del proyecto.

Fase 5

- **Requerimientos:** Se incorporaron nuevos requerimientos en cuanto a la consistencia, definiendo el alcance final.
- **Diseño:** Se agregaron componentes para cumplir con esta nueva funcionalidad.
- **Prototipos:** Se realizaron prototipos con los nuevos requerimientos, integrándolo al prototipo anterior.
- **Documentación:** Se actualizó la documentación con los cambios realizados.

Fase 6

- **Prototipos**: Se evalúa la opción de utilizar eventos para la comunicación entre los componentes con *Akka*.
- **Diseño:** Se decide no utilizar *Akka* y se hacen los ajustes finales del diseño.
- **Documentación:** Actualización de la documentación con los ajustes.

Fase 7

- **Implementación y testing**: Se procedió a realizar la implementación, utilizando metodologías ágiles para el desarrollo, realizando el testing de los componentes y su comunicación en etapas tempranas. Al finalizar la implantación se realizó t*est end to end*.
- **Pruebas de performance**: Se diseñó e implementó las pruebas de performance sobre el sistema, observando el comportamiento en cada una.
- **Documentación**: Se procedió a realizar la documentación final del proyecto.

Al comienzo del proyecto se utilizó el modelo en cascada, dado que resultaba necesario finalizar una fase para definir la siguiente. Al comenzar con la implementación de los prototipos se comenzó a utilizar el modelo de *Scrum*, siguiendo una estrategia de desarrollo incremental, esto se puede observar a partir de la fase 3 en el diagrama de Gantt de la Figura 50.

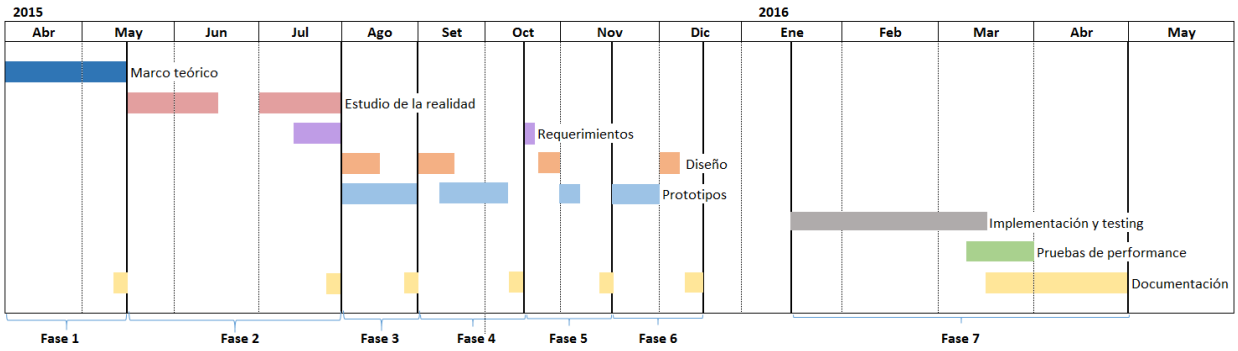

Figura 50 - Diagrama de Gantt

7 Conclusiones

El objetivo del proyecto fue migrar un sistema Monolítico basado en una arquitectura SOA hacia una de Microservicios. Para cumplir este objetivo se investigaron los principales beneficios y desafíos generados al tener una arquitectura de este tipo junto con posibles soluciones. El estudio de esta nueva arquitectura no generó gran dificultad, ya que muchos de los nuevos conceptos están alineados con los principios de SOA.

Luego se realizó un estudio exhaustivo del sistema a rediseñar para comprender la realidad, lo cual no presentó mayores dificultades. Al momento de analizar los detalles de la implementación se empezó a complejizar el entendimiento del sistema, así como también la preparación del ambiente de desarrollo, sumado a que estaba desarrollado con tecnologías desconocidas por el grupo. La inversión de tiempo en esta etapa fue considerable sin obtener buenos resultados. Se concluye a partir de esta experiencia que, si se quiere rediseñar un sistema reutilizando el código original, es necesario tener un buen conocimiento sobre el diseño y tecnologías de implementación. Para ello, es importante haber participado en el proceso, tener contacto directo con algún integrante o en caso contrario, contar con mucho tiempo para el análisis.

Debido a los problemas presentados al comprender la implementación y tecnologías del sistema original, se decidió no utilizar el código fuente con el objetivo de no perder el foco en la aplicación de Microservicios sobre el sistema. Al aplicar Microservicios surgieron nuevos requerimientos, algunos inherentes a esta arquitectura y otros a partir de las restricciones identificadas en el sistema original. Estos requerimientos fueron realizados en su totalidad, pero se debió acotar el alcance de la migración total del sistema.

A nivel de selección de tecnologías, se encontró que existen varias de ellas que facilitan el desarrollo, resolviendo algunos de los desafíos que presenta la arquitectura de Microservicios. A pesar de ser tecnologías novedosas, se encuentran estables, con buena documentación y una comunidad que las respalda. Estas tecnologías fueron evaluadas de forma positiva cumpliendo con el fin del proyecto, pero hay que considerar que tienen una gran curva de aprendizaje, y al ser muchas, el tiempo invertido en la fase de prototipos fue mayor al esperado.

Para realizar el nuevo diseño, se identificaron los *Bounded Context* y los problemas inherentes de una arquitectura de Microservicios, para luego definir las soluciones y la nueva arquitectura.

En el proceso de diseño se tuvieron cuatro fases, donde en las primeras tres se realizaron prototipos incrementales, tratando de mitigar los riesgos de las nuevas tecnologías y formando una base de la arquitectura. En la última fase se evaluó la utilización de eventos mediante la tecnología *Akka,* pero esta opción fue descarta dado la curva de aprendizaje que posee y la etapa del proyecto en la que se estaba. A su vez, todos los componentes quedarían acoplados a esta tecnología y esto va contra los principios de Microservicios.

Al comenzar el proceso de implementación se contaba con una base de los componentes de la arquitectura ya integrados y con conocimiento sobre las tecnologías, lo que permitió focalizarse en su desarrollo.

Al tener una arquitectura distribuida, resultó útil testear en primer lugar la lógica interna de los componentes, para luego emular los mensajes que ingresaban. Finalmente se realizó una integración progresiva, acotando los factores de falla en comparación a realizar una integración masiva. Se puede concluir que al tener una arquitectura de Microservicios es importante realizar primero el testing de cada componente por separado y luego una

integración progresiva, permitiendo detectar errores en etapas tempranas donde el tamaño del sistema es menor.

En lo referente al testing del sistema, se realizaron los *test end to end*, donde se probaron los procesos identificados que incluyen, simulación de caídas, demoras en responder y exceso de la cantidad de pedidos establecidos en el SLA. En este proceso fue complicado llevar la traza sobre lo que estaba ocurriendo en el sistema y saber si el comportamiento era el esperado. Esto es debido a la naturaleza distribuida y no tener un punto central de *logs*, por lo que se decidió agregar en la base de datos la traza de los pagos.

Por último, se realizaron pruebas para evaluar el comportamiento ante fallas en la comunicación y de performance. Esto permitió detectar cuellos de botella y saber la capacidad de procesamiento. De estas pruebas se obtuvieron resultados positivos, destacando que al escalar el *Level Coordinator* el sistema tiene mayor capacidad de procesamiento. A su vez los mecanismos para detectar fallas en la comunicación con los sistemas externos resultaron útiles, ya que se obtuvieron mejores tiempos de respuesta en comparación a cuando no se utilizaron.

Como conclusión general de la arquitectura de Microservicios y de la experiencia adquirida en este proyecto se puede decir que se presenta como una buena opción al momento de elegir una arquitectura para un sistema de gran escala. Si se trata de un sistema pequeño quizás agrega más complejidad que los beneficios que le aporta, pero se puede pensar la arquitectura de manera que si el sistema en un futuro incrementa su tamaño y requiere mayor soporte de usuarios el proceso de migración sea más sencillo y directo.

7.1 Trabajo Futuro

Si bien el objetivo del proyecto se considera alcanzado, se detectan mejoras y desafíos pendientes.

Gestión centralizada de logs: Al tener los *logs* distribuidos en todo el sistema resulta complejo seguir la traza de lo que ocurre en el mismo. Lo que se realizó en el proyecto fue la implementación de una librería para la estandarización de los logs, todos los componentes utilizan esta librería. Se propone como mejora tener una gestión centraliza de *logs* mediante alguna herramienta como *Logstash,* el tener incorporada la liberaría de estandarización de logs facilita la utilización de esta herramienta.

Monitoreo: Si bien se cuenta con el monitoreo centralizado sobre memoria y CPU, se detecta como debilidad en el sistema que si ocurre una falla no se notifica el suceso, para esto se propone utilizar herramientas de monitoreo de alertas como *Nagios.*

Interfaz gráfica de administración: Si bien se cuenta con los servicios necesarios para la administración, un administrador tendría que invocar a cada servicio, introduciendo errores en las invocaciones y a su vez esto no es amigable. Por lo cual se propone una interfaz gráfica que consuma los servicios del componente *Admin*.

Deploy automático: Al tener muchos componentes para realizar el *deploy* resulta costoso el tener que realizarlo de forma independiente, por lo que se propone realizar un *deploy* de forma automática y centralizada, utilizando alguna herramienta de integración continua como *Jenkins.*

Tolerancia a fallas en el Circuit Breaker: En la solución actual no existe un mecanismo para que se recupere el estado de los servicios externos. Se propone como solución que los

componentes *Circuit Breaker* trabajen en modo clúster donde compartirían la información, por ejemplo, en una *cache.*

Seguridad: Debido al alcance del proyecto este tema no se atacó, se propone que se incluyan mecanismos de seguridad en todos los componentes del sistema y en la comunicación.

Migración: Debido al alcance del proyecto no se realizó la migración de todo el sistema, se propone como trabajo futuro migrar todo el resto del sistema *Legacy* a Microservicios.

8 Bibliografía

[1] Service Orientation [En línea]. Available: http://serviceorientation.com/serviceorientation/index. [Último acceso: 16 Marzo 2016].

[2] IBM Web Services [En línea]. Available: http://www.ibm.com/developerworks/webservices/tutorials/ws-soacert1/section2.html. [Último acceso: 2016 Abril 14].

[3] IBM [En línea]. Available: https://www.ibm.com/developerworks/library/ws-soa-design1/. [Último acceso: 14 Marzo 2016].

[4] Introduction to Microservices. Nginx [En línea]. Available: https://www.nginx.com/blog/introduction-to-microservices/. [Último acceso: 14 Marzo 2016].

[5] Microservices [En línea]. Available: http://martinfowler.com/articles/microservices.html. [Último acceso: 14 Marzo 2016].

[6] InfoQ [En línea]. Available: http://www.infoq.com/news/2015/04/azure-service-fabric. [Último acceso: 14 Marzo 2016].

[7] Adopting Microservices at Netflix [En línea]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. [Último acceso: 14 Marzo 2016].

[8] S. Newman, Building Microservices.

[9] The Bounded Context Explained [En línea]. Available: http://blog.sapiensworks.com/post/2012/04/17/DDD-The-Bounded-Context-Explained.aspx. [Último acceso: 14 Marzo 2016].

[10] Bounded Context [En línea]. Available: http://martinfowler.com/bliki/BoundedContext.html. [Último acceso: 14 Marzo 2016].

[11] M. L. A. &. M. T. Fisher, The art of Scalability..

[12] Circuit Breaker [En línea]. Available: http://martinfowler.com/bliki/CircuitBreaker.html. [Último acceso: 14 Marzo 2016].

[13] Testing Strategies in a Microservice Architecture [En línea]. Available: http://martinfowler.com/articles/microservice-testing. [Último acceso: 14 Marzo 2016].

[14] Testing Strategies in a Microservice Architecture [En línea]. Available: http://martinfowler.com/articles/microservice-testing/#conclusion-test-pyramid. [Último acceso: 09 Abril 2016].

[15] Jenkins [En línea]. Available: https://jenkins.io/. [Último acceso: 09 Abril 2016].

[16] New Relic [En línea]. Available: http://newrelic.com/. [Último acceso: 09 Abril 2016].

[17] Nagios [En línea]. Available: https://www.nagios.org/. [Último acceso: 09 Abril 2016].

[18] Graylog [En línea]. Available: https://www.graylog.org/. [Último acceso: 09 Abril 2016].

[19] Logstash [En línea]. Available: https://www.elastic.co/products/logstash. [Último acceso: 09 Abril 2016].

[20] Amazon Web Services [En línea]. Available: https://aws.amazon.com. [Último acceso: 09 Abril 2016].

[21] OpenStack [En línea]. Available: https://www.openstack.org/. [Último acceso: 09 Abril 2016].

[22] Azure [En línea]. Available: https://azure.microsoft.com/es-es/. [Último acceso: 09 Abril 2016].

[23] About Linux Containers [En línea]. Available: https://docs.oracle.com/cd/E37670_01/E37355/html/ol_about_containers.html.. [Último acceso: 09 Abril 2016].

[24] Insights containers [En línea]. Available: https://www.redhat.com/en/insights/containers.. [Último acceso: 09 Abril 2016].

[25] Containerisation vs Virtualisation [En línea]. Available: http://blog.serverspace.co.uk/containerisation-vs-virtualisation-whats-the-difference. [Último acceso: 09 Abril 2016].

[26] Tutum [En línea]. Available: https://www.tutum.co/. [Último acceso: 09 Abril 2016].

[27] Docker Cloud [En línea]. Available: https://cloud.docker.com/. [Último acceso: 09 Abril 2016].

[28] Service Discovery in a Microservices Architecture [En línea]. Available: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. [Último acceso: 14 Marzo 2016].

[29] A pattern language for microservices [En línea]. Available: http://microservices.io/patterns/index.htm. [Último acceso: 14 Marzo 2016].

[30] PWC [En línea]. Available: http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.html. [Último acceso: 14 Marzo 2016].

[31] The Great Debate: Microservices vs SOA [En línea]. Available: http://www.tibco.com/blog/2015/07/24/the-great-debate-microservices-vs-soa/. [Último acceso: 14 Marzo 2016].

[32] Microservices and SOA [En línea]. Available: http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html. [Último acceso: 14 Marzo 2016].

[33] N. R. M. Andrés Pereiro, SLA Enforcer, proyecto de grado Facultad de Ingeniería UDELAR. 2014.

[34] The “4+1” View [En línea]. Available: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf. [Último acceso:

24 Marzo 2016].

[35] Elmasri-Navathe, Fundamentals of Database Systems.

[36] Docker [En línea]. Available: https://www.docker.com/. [Último acceso: 04 Abril 2016].

[37] Docker [En línea]. Available: https://docs.docker.com/v1.5/articles/networking/. [Último acceso: 04 Abril 2016].

[38] WildFly [En línea]. Available: http://wildfly.org/about/. [Último acceso: 13 Abril 2016].

[39] MySql [En línea]. Available: https://www.mysql.com/. [Último acceso: 04 Abril 2016].

[40] Hibernate [En línea]. Available: http://hibernate.org/. [Último acceso: 04 Abril 2016].

[41] Rabbitmq [En línea]. Available: https://www.rabbitmq.com/. [Último acceso: 04 Abril 2016].

[42] Spring framework [En línea]. Available: https://projects.spring.io/spring-framework/. [Último acceso: 04 Abril 2016].

[43] Zookeeper [En línea]. Available: https://zookeeper.apache.org/. [Último acceso: 04 Abril 2016].

[44] Etcd [En línea]. Available: https://github.com/coreos/etcd. [Último acceso: 04 Abril 2016].

[45] Eureka [En línea]. Available: https://github.com/Netflix/eureka. [Último acceso: 04 Abril 2016].

[46] Consul [En línea]. Available: https://www.consul.io/intro/index.html. [Último acceso: 04 Abril 2016].

[47] Registrator [En línea]. Available: https://github.com/gliderlabs/registrator. [Último acceso: 04 Abril 2016].

[48] Consul-template [En línea]. Available: https://github.com/hashicorp/consul-template. [Último acceso: 04 Abril 2016].

[49] The Go Programming Language [En línea]. Available: https://golang.org/pkg/text/template/. [Último acceso: 04 Abril 2016].

[50] Nginx [En línea]. Available: https://www.nginx.com/. [Último acceso: 04 Abril 2016].

[51] Apache maven [En línea]. Available: https://maven.apache.org/. [Último acceso: 04 Abril 2016].

[52] Git [En línea]. Available: https://git-scm.com/. [Último acceso: 04 Abril 2016].

[53] Bitbucket [En línea]. Available: https://bitbucket.org/. [Último acceso: 09 Abril 2016].

[54] DockerUi [En línea]. Available: https://github.com/felixgborrego/docker-ui-chrome-app. [Último acceso: 04 Abril 2016].

[55] Junit [En línea]. Available: http://junit.org/junit4/. [Último acceso: 04 Abril 2016].

[56] SoapUi [En línea]. Available: https://www.soapui.org/ . [Último acceso: 04 Abril 2016].

[57] Microservice Design Patterns [En línea]. Available: https://dzone.com/articles/microservice-design-patterns. [Último acceso: 14 Marzo 2016].

[58] Postman [En línea]. Available: http://www.getpostman.com/. [Último acceso: 04 Abril 2016].

[59] Introducción al Middleware [En línea]. Available: https://eva.fing.edu.uy/pluginfile.php/68246/mod_resource/content/3/03%20-%20Conceptos%20Preliminares%20-%20SOA.pdf. [Último acceso: 09 Abril 2016].

Anexo 1. Requerimientos del sistema Legacy

Los requerimientos completos del sistema *Legacy* fueron divididos en cinco grupos que se detallan a continuación:

Servicios Finales

ID	Requerimiento	Descripción
RQ-SF1	Clasificación de servicios	Los servicios que se ingresan al sistema deben de poder clasificarse según si son simples, cancelables o anulables.
RQ-SF2	Servicios sincrónicos	Los servicios serán request-response sincrónicos. Al invocar un servicio, el sistema permanece a la espera de una respuesta
RQ-SF3	Servicios SOAP sobre HTTP o HTTPS	Los servicios finales configurados en el sistema respetarán el estándar para Web Services SOAP, utilizando como canal de comunicación http o https.
RQ-SF4	Métricas para los servicios	Se implementan las métricas tiempo de respuesta, accesibilidad y nivel de éxito.
RQ-SF5	SLA Basado en Response Time	La métrica Tiempo de respuesta (response time) será utilizada para la evaluación del cumplimiento o no del SLA correspondiente.
RQ-SF6	Response Time con valores teóricos y en runtime	Para la métrica response time, se considerarán dos valores: • Medida teórica: refiere a los valores teóricos establecidos en los contratos. • Medida en ejecución (o runtime): refiere a los valores de la métrica obtenidos en cada invocación del servicio.

Servicios de cara al cliente

ID	Requerimiento	Descripción
RQ-SC1	Único Web Service hacia locales de pago	Se define un único Web Service con una operación a la cual se la invocará con la información necesaria para poder realizar varios pedidos hacia diferentes servicios de terceros. El servicio devolverá el estado de los pedidos realizados.
RQ-SC2	Invocación serial de un mismo servicio para un mismo local en un pedido	Cada local contará con un identificador único el cual deberá ser utilizado en cada invocación a los efectos de controlar que no haya más de una transacción activa por local a un mismo servicio final.
RQ-SC3	SLA con tiempo de respuesta como métrica	El servicio de cara al cliente expuesto por el sistema deberá de contar con un SLA acordado con el local que considere como métrica el tiempo de respuesta de una invocación.
RQ-SC4	Políticas de ejecución	El sistema debe soportar las siguientes políticas de ejecución: • Todo o nada. • Best effort.

RQ-SC5	Política global configurable	El sistema debe permitir configurar una política global la cual será utilizada únicamente en el caso en el que el cliente no especifique la política a utilizar en el pedido.
RQ-SC6	Especificar política por invocación	La política utilizada también puede ser informada en cada invocación. Si en la invocación se especifica la política a utilizar, esta toma prioridad sobre la definida a nivel global.
RQ-SC7	Parámetros de servicios	Los parámetros de la invocación y respuesta a cada servicio son especificados en formato clave-valor.

Sistema central

ID	Requerimiento	Descripción
RQ-S1	Algoritmo de Scheduler	El sistema implementará un algoritmo de scheduler para la resolución de cómo y en qué orden ejecutar la invocación a los servicios finales relacionados a una invocación del cliente. Este algoritmo debe retroalimentarse con los datos de las métricas calculadas.
RQ-S2	Cumplimiento de SLA	Para la evaluación del complimiento del SLA con el local, el sistema se basará en la métrica response time.
RQ-S3	Cálculo de métricas	En cada invocación a un servicio final se calcula el valor de las métricas. Este dato es utilizado para validar la factibilidad del cumplimiento o no del SLA para el pedido en curso.
RQ-S4	Histórico de métricas	Los valores calculados de las métricas se almacenan para mantener un histórico y son utilizados para validar si se puede cumplir el SLA del pedido antes realizar alguna invocación.
RQ-S6	API para ABM de servicios y SLA	El sistema brindará una API que permita dar de alta un nuevo servicio y el SLA correspondiente

Interfaces

ID	Requerimiento	Descripción
RQ-I1	Interfaces en ambiente web	Todas las interfaces del sistema serán desarrolladas en ambiente web.
RQ-I2	Interfaz para monitoreo de SLAs	Se dispondrá de una interfaz que permita el monitoreo del estado de los SLA correspondientes a los distintos servicios finales involucrados en el sistema.
RQ-I3	Interfaz de configuración de parámetros globales	Desarrollo de una interfaz web que permita la configuración las siguientes características globales del sistema: • Política de ejecución (todo o nada,

		best effort). • Establecer el tiempo máximo por defecto de espera del servicio de cara al cliente.
RQ-I4	Interfaz de administración de servicios finales	El sistema contará con una interfaz a través de la cual se podrán administrar los servicios finales pudiendo dar de alta, baja y/o modificar los mismos.

No funcionales

ID	Requerimiento	Descripción
RQ-NF1	Distribución de recursos	Correcta distribución de los recursos de hardware/software disponibles
RQ-NF2	Tecnologías	El sistema deberá de ser desarrollado en Java sobre Wildfly y Switchyard.

Anexo 2. Detalle de la vista de componentes físicos

Register and Discovery Service

El componente lógico encargado del registro y descubrimiento de servicios se divide en tres componentes físicos:

- *Third-Party Registration:* Encargado del registro de los servicios en el *Service Registry.*
- *Service Registry:* Contiene la información y parámetros de configuración de los servicios.
- *Load Balancer:* Realiza la distribución de la carga entre las diferentes instancias del mismo servicio. El *Load Balancer* consulta al *Service Registry* para obtener los nodos a balancear.

Se utilizan herramientas ya existentes para la implementación de estos componentes. La comunicación entre éstos se realiza de forma sincrónica mediante HTTP REST.

Level Coordinator

Este componente establece distintos tipos de comunicación con diferentes componentes:

- *Legacy*: El *Level Coordinator* es el punto de entrada al nuevo sistema, el sistema *Legacy* se comunica con éste para la ejecución de un nivel. La comunicación es asincrónica por medio de colas de mensajes. Para poder escalar horizontalmente, el *Level Coordinator* implementa el patrón *Competing Consumer* sobre la cola de mensajes *Level InPut*, pudiendo tener varias instancias de este componente consumiendo mensajes. Finalmente, cuando culmina el procesamiento de un nivel envía a la cola *Level OutPut* el resultado.

- *Scheduler*: En este caso la comunicación es sincrónica vía HTTP REST. Esto es debido a que el *Level Coordinator* envía al *Scheduler* un mensaje para ser notificado dentro de un determinado período de tiempo y necesita saber si la solicitud se realizó con éxito.

- *Task Coordinator - Task Input*: El *Task Coordinator* implementa el *patrón Competing Consumer* sobre la cola de mensajes Task Input para recibir las tareas a ejecutar; esto permite tener múltiples instancias ejecutando en paralelo diferentes tareas.

- *Task Coordinator - Task Consistency*: El *Level Coordinator* envía una notificación de fin de procesamiento a la cola Task Consistency. Este mensaje será recibido por todos los *Task Coordinator*, debido a que éstos implementan el patrón *Publish*/*Suscribe*; permitiendo que el *Level Coordinator* no tenga que esperar una respuesta. Si la comunicación fuera sincrónica, debería esperar la confirmación de todos los *Task Coordinator*.

- *Consistency Manager*: La comunicación con el *Consistency Manager* se realiza de forma asincrónica, por medio de colas de mensajes. El *Consistency Manager* recibe por la cola *Consistency InPut* pagos que es necesario cancelar o anular para que el nivel quede consistente. Debido a que el sistema mantiene consistencia eventual, no hay necesidad que el *Level Coordinator* espere una respuesta inmediata. Cuando el *Consistency Manager* tenga un resultado lo enviará a la cola *Consistency OutPut.*

Task Coordinator

Se comunica con el *Connector* para realizar la ejecución de los pagos de una tarea y con el *Level Coordinator* para enviar los resultados de los mismos. A continuación, se describen estas comunicaciones:

- *Connector:* La comunicación con el *Connector* se realiza de forma sincrónica vía HTTP REST, debido a que el *Task Coordinator* debe ejecutar los pagos de forma serial y necesita saber el resultado de un pago para ejecutar el siguiente. Para balancear la carga entre los diferentes *Connectors* de un mismo tipo se utiliza un *Load Balancer.*

- *Level Coordinator - Task Output:* El *Level Coordinator* consume mensajes de una cola implementando el patrón *Competing Consumer*. Esto permite al *Task Coordinator* enviar los resultados intermedios de cada pago. Si la comunicación fuera sincrónica, no se contaría con los resultados intermedios de los pagos, impidiendo tener un resultado parcial del nivel.

Connector

Se tiene al menos un componente de este tipo por cada servicio de tercero, el mismo se comunica con:

- External Service: La comunicación con el servicio de tercero puede ser implementada mediante el protocolo que imponga el mismo. La única restricción se encuentra en que el *Connector* tiene que exponer una API REST con los servicios pay, annul y *cancel*. Cada *Connector* implementa estos servicios según corresponda realizando las transformaciones necesarias para establecer la comunicación con el servicio de tercero.

- *Circuit Breaker*: Esta comunicación se realiza asincrónicamente por medio de colas de mensajes. El *Circuit Breaker* recibe notificaciones respecto a si la comunicación

se pudo establecer de forma correcta con los servicios de tercero, realizando su procesamiento a medida que sea posible.

Circuit Breaker

El *Circuit Breaker* recibe mensajes por parte de los *Connectors* y envía notificaciones cuando detecta un cambio de estado del servicio de tercero. Las notificaciones se realizan de manera asincrónica mediante cola de mensajes implementando el patrón *Publish/Suscribe*, lo cual permite que cada *Connector* se suscriba al *topic* de su tipo para recibir notificaciones de cambio de estado.

Consistency Manager

Este componente es el encargado de la ejecución de la compensación de los pagos, para lograrlo se comunica con los siguientes componentes:

- *Connector*: Invoca al conector para la cancelación y anulación de los pagos. Al igual que el *Task Coordinator* esta comunicación se realiza por medio de un *Load Balancer*.
- *Level Coordinator*: Se comunica con el *Level Coordinato*r para notificar el resultado de los pagos ya procesados. Esta comunicación es asincrónica a través de una cola de mensajes persistente, de este modo se logra que el *Level Coordinator* consuma los mensajes de la misma a medida que pueda y garantizar que si por alguna razón el mismo no está disponible, los mensajes serán persistidos.
- *Scheduler*: En caso que un pago falle, se agenda un evento en el *Scheduler* para procesar el pago más adelante, esta comunicación se realiza sincrónica, reintentando en caso de falla, con el objetivo de asegurarse que llegue a destino.

Scheduler

Los tipos de eventos que agenda el *Scheduler* pueden ser enviados por HTTP REST o cola de mensajes, según la entrada que requiera el servicio al cual se notifica. En el caso del *Level Coordinator,* se realiza por HTTP REST y para el *Consistency Manager* mediante cola de mensajes.

Base de datos

Las diferentes instancias de *Level Coordinator* trabajan en forma de *cluster* realizando inserciones y consultas sobre la misma base de datos a través de JDBC.

Admin

Este componente se comunica con el *Service Registry* para la configuración de parámetros referentes a los SLA de los servicios de terceros.

Anexo 3. Detalle de los procesos

Se presentan los procesos con más detalles, mostrando nuevamente la imagen en los mismos para mayor claridad.

Curso Normal

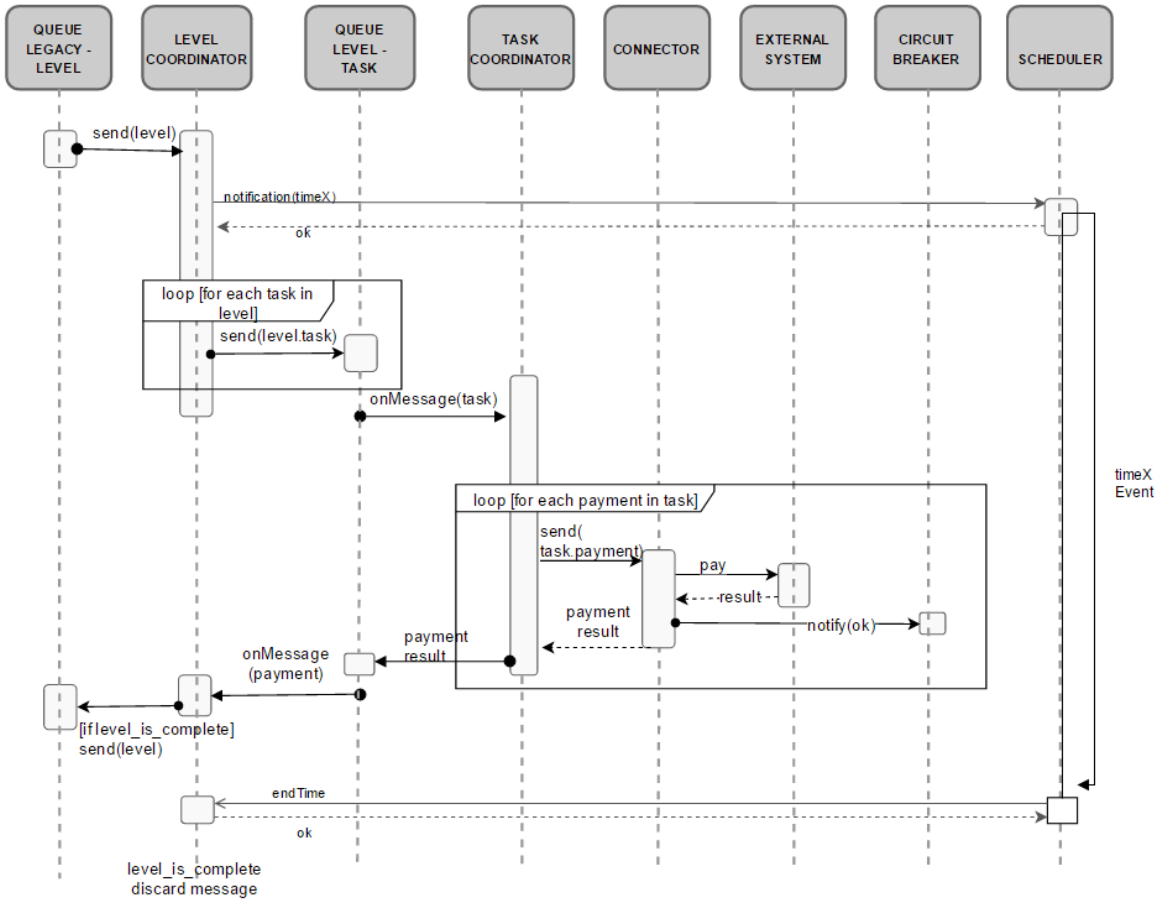

Figura 51 - Curso Normal

1. El *Level Coordinator* recibe un nivel de ejecución y ejecuta las siguientes acciones:
    a. Obtiene el tiempo de fin procesamiento y envía al *Scheduler* un mensaje para ser notificado en dicho tiempo.
    b. Divide el nivel en tareas y envía a la cola *Task Input* cada una de ellas.
2. El *Task Coordinator* obtiene una tarea de la cola y ejecuta los pagos de forma serial.
    a. La ejecución de cada pago consiste en invocar al *Connector* correspondiente.
        i. El *Connector* realiza la transformación necesaria e invoca al sistema externo.
        ii. Luego envía una notificación al *Circuit Breaker* indicando si la comunicación fue exitosa.
3. El *Task Coordinator* envía el resultado de cada pago a la cola de respuesta del Level *Coordinator.*
4. El *Level Coordinator* al recibir un resultado, verifica si el nivel se completó, en cuyo caso envía el resultado del nivel completo hacia la cola de respuesta del sistema *Legacy.*

Fin de tiempo de procesamiento

A continuación, se detalla el caso de fin de procesamiento cuando se tiene una política ToN.

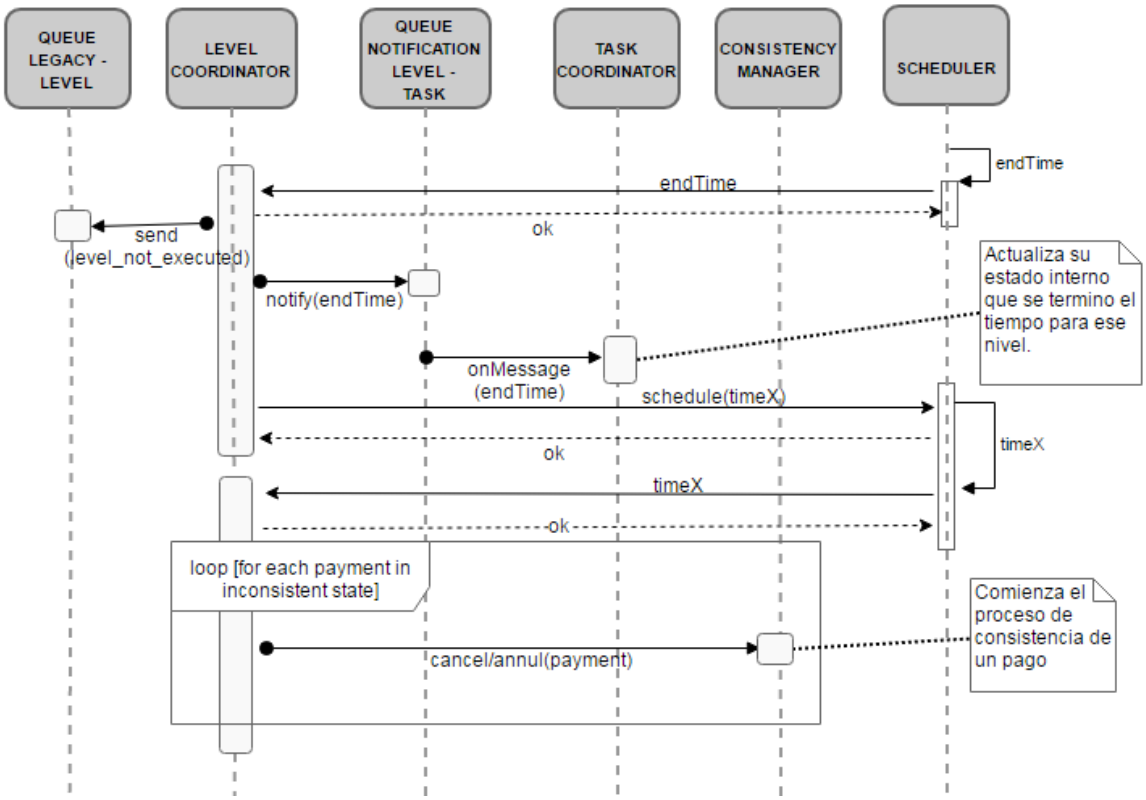

Figura 52 - Fin de tiempo de procesamiento

1. El fin de procesamiento es notificado por parte del *Scheduler* hacia el *Level Coordinator.*
2. Al llegar una notificación de fin de procesamiento, el *Level Coordinator* envía el resultado del nivel hacia la cola de respuesta del sistema *Legacy,* indicando cada uno de los pagos como no ejecutados.
3. Luego de enviar el resultado, envía una notificación de fin de procesamiento a todos los *Task Coordinator* a través de la cola de notificación. Cada uno de ellos realizara lo siguiente:
    a. Actualizar el estado del nivel en caso de estar procesando pagos de éste.
    b. Al finalizar la ejecución de un pago en curso y detectar que el nivel está en el estado fin de tiempo de procesamiento, envía hacia la cola de respuesta el resultado obtenido para dicho pago y los siguientes pagos como no ejecutados.
4. El *Level Coordinator* envía un mensaje al *Scheduler* para agendar la posterior anulación y/o cancelación de pagos.
5. Luego de pasado el tiempo agendado, el *Scheduler* envía un mensaje al *Level Coordinator*, indicando una notificación de consistencia.
6. El *Level Coordinator* al recibir la notificación mencionada, verifica si hay pagos que se encuentran en estado inconsistente y si es necesario ejecutar la anulación o

cancelación de los mismos. Para cada pago que sea necesario realizar dicho procedimiento, se envía a la cola del *Consistency Manager*

7. El *Consistency Manager* procesa el mensaje e invoca al *Connector* correspondiente, al recibir un resultado válido envía a la cola de respuesta del *Level Coordinator.*

TimeOut de servicios externos

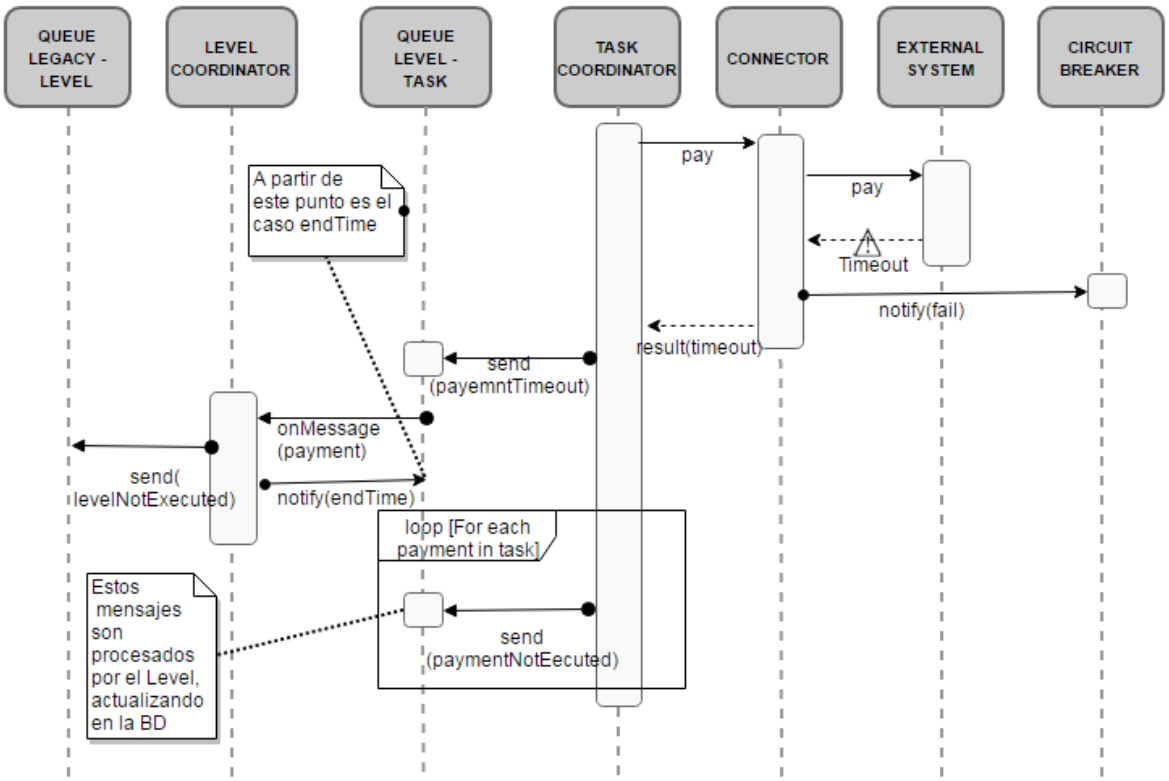

Figura 53 - Timeout

Este proceso comienza en el *Connector*, ya es que el encargado de establecer la comunicación con el servicio de tercero. La secuencia de pasos consiste en:

1. El *Connector* responde al *Task Coordinator* un error de *timeout*.
2. Luego notifica al *Circuit Breaker* que la comunicación no fue exitosa con el servicio de tercero.
3. El *Task Coordinator* envía al *Level Coordinator* el resultado del pago con error de *timeout* y retorna los pagos restantes de la tarea con estado de no ejecución.
4. El *Level Coordinator* al recibir el resultado del pago con error toma las siguientes acciones:
    a. Envía el resultado del pedido al sistema *Legacy*, con el estado de los pagos como no ejecutados y el estado del nivel con error.
    b. Envía fin de procesamiento a los *Task Coordinator* y se procede igual que el caso de fin de procesamiento descripto anteriormente a partir del punto 3.

Consistencia de un pago

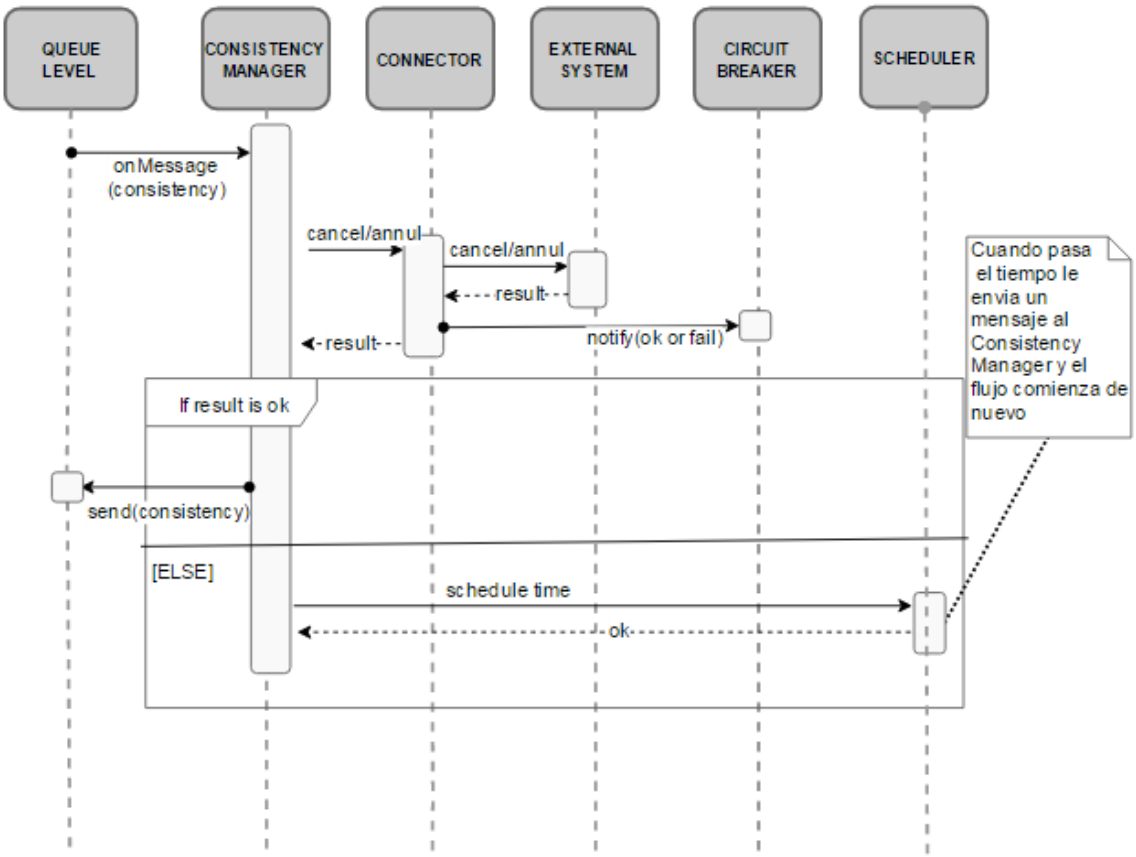

Figura 54 - Consistencia de un pago

1. Realiza la invocación al *Sistema Externo* mediante el *Connector* correspondiente.
2. Existen dos alternativas para el resultado del pago:
    a. Resultado exitoso, entonces lo envía al *Level Coordinator.*
    b. Resultado no exitoso, entonces el *Consistency Manager* agenda en el *Scheduler* la compensación para una nueva invocación. Al recibir la notificación del *Scheduler* vuelve al paso 1.
3. El *Level Coordinator* al recibir este mensaje actualiza el estado del pago en la base de datos. Luego, verifica si el nivel se encuentra en estado consistente para actualizar el estado.

Anexo 4. Flujo de estados internos de pagos

A continuación, se expone un ejemplo para comprender mejor los cambios de estados en cada una de las máquinas de estados y la relación que existe entre los estados del nivel y sus pagos.

Ejemplo: Se tiene un nivel con una política ToN, el cual posee 2 pagos en una tarea. El primero de éstos se procesa correctamente, pero el segundo no puede realizarse debido a un error de negocio. El resultado del nivel enviado hacia el sistema Legacy será el siguiente:

- estado del nivel: *ERROR*
- estado del primer pago: *NOT_EXECUTED*
- estado del segundo pago: *NOT_EXECUTED*

Si bien en este caso los estados externos de los pagos son "no ejecutados", internamente el primer pago se ejecutó correctamente, debiendo aplicar una compensación para lograr la consistencia con lo enviado hacia el *Legacy*. Para lograr esto, el *Level Coordinator* cambia el estado de dicho pago a *TO_ANNUL* para que sea anulado, se cambia el estado interno del nivel a *EVENTUAL_CONSISTENCY* para indicar que se están realizando compensaciones sobre ese nivel.

Cuando se realice la anulación correctamente mediante la invocación al *Consistency Manager* el nivel quedará consistente pudiendo cambiarlo al estado *CONSISTENCY*.

Anexo 5. Modelo de base de datos

El significado de cada campo de las tablas en la base de datos se muestra a continuación:

Tabla Level	**Significado**
levelId	Identificador del nivel
status	Estado del nivel: *PROCESSING*, *EVENTUAL_CONSISTENCY*, *CONSISTENCY.*
policy	Política: ToN, BE, DoP.
startTime	Fecha en la cual el *Level Coordinator* comienza el procesamiento del nivel.
endTime	Fecha en la cual se envía el resultado hacia el *Legacy.*
endProcessTime	Fecha de fin de procesamiento.

Tabla Payment	**Significado**
levelId	Identificador del nivel al cual pertenece el pago.
taskId	Identificador de la tarea a la cual pertenece el pago.
paymentId	Identificador del pago.
endPoint	Nombre del servicio de tercero a invocar.
data	Información necesaria para realizar el pago en formato json.
sent	Estado enviado al *Legacy.*
status	Estado actual
consistencyId	Último identificador de consistencia utilizado para el pago.

Tabla PaymentLog	**Significado**
guid	Identificador generado para la consistencia.
levelId	Identificador del nivel al cual pertenece el pago.
taskId	Identificador de la tarea a la cual pertenece el pago.
paymentId	Identificador del pago.
Data	Información específica del resultado del pago.
timeStamp	Fecha de inserción.

Anexo 6. Uso de cache

El *Task Coordinator* cuenta con una *cache* para manejar el estado del nivel, se utiliza por los diferentes hilos que contiene a modo de poder compartir esta información.

Se implementó mediante una clase *Singleton* que contiene un mapa con la información de cada nivel que el componente está ejecutando. Esta información está formada por el estado del nivel y las tareas en ejecución.

Cuando llega una notificación de fin de procesamiento, el *Worker* encargado actualiza el estado del nivel en la cache a *END_OF_TIME.*

Antes de la ejecución de cada pago, se consulta el estado del nivel correspondiente en la *cache*, si el estado se corresponde con *END_OF_TIME* no es ejecutado, enviando este pago y los que le siguen como no ejecutados al *Level Coordinator*.

Una vez que finaliza la ejecución de las tareas de un nivel, la información referente al mismo es eliminada de la *cache*.

Anexo 7. Test end to end

A continuación, se describen los test realizados para verificar el correcto funcionamiento del sistema:

- **Curso normal**: Para el curso normal se enviaron un conjunto de pagos que requieren diferentes servicios de terceros, utilizando los dos modos de ejecución. En todos los casos el resultado esperado corresponde a que los pagos fueron procesados correctamente.
- **Error de negocio**: En este caso el conjunto de pagos seleccionado, además de requerir servicio de terceros diferentes, consiste en que el resultado de sea el correspondiente a error de negocio, variando la política. Comprobando que en el caso que se estuviera en una política *ToN* y un pago fallara por un error de negocio, el resto de los pagos no se ejecutaran.
- **Timeout**: Para probar el caso de *timeout* se simuló que los servicios de terceros demoran en contestar. En este caso los pagos que dieran *timeout* deberían ser cancelados.
- **Throttler**: Para poder visualizar correctamente el comportamiento del sistema en este caso, se utilizaron pagos configurados de tal forma que el procesamiento en el servicio de tercero demore. De esta forma si el Throttler estaba configurado en una cantidad de pedidos concurrentes n, al enviar el pedido n + 1 se debe obtener error. Estas pruebas fueron realizadas cambiando la política, si corresponde a *ToN* los pagos que se ejecutaron correctamente deben anularse.
- **Servicio de tercero no disponible**: se enviaron un conjunto de pagos dentro de los cuales algunos corresponden a un servicio de tercero que no está disponible, cambiando la política de ejecución, esperando resultados diferentes en cada caso. Si corresponde a una política *ToN* los pagos que se ejecutaron correctamente deben anularse.
- **Fin de procesamiento**: Para realizar las pruebas de fin de procesamiento se asocia al mensaje un tiempo de procesamiento pequeño y con niveles que contienen varias tareas. Verificando que la notificación es recibida por todos los *Task Coordinator* y que cada uno finaliza su procesamiento, enviando el resultado de los pagos restante como no ejecutados.
- **Circuit Breaker**: Para realizar pruebas del correcto funcionamiento del *Circuit Breaker,* se enviaron pagos hacia un servicio que no estaba disponible hasta llegar al punto donde el circuito se abre, verificando que en este caso el error recibido sea por *Circuit Breaker*.

- **Concurrencia y consistencia**: Las pruebas de *end to end* relacionadas a la consistencia fueron realizadas con dos instancias de *Level Coordinator,* enviando varios niveles en diferentes condiciones, verificando que luego de un tiempo cada nivel quedaba en un estado consistente.

- **Cambio en SLA**: Se cambian dinámicamente los parámetros del SLA, verificando que estos cambios se realicen en *Consul* y probando en cada caso que el sistema se comporta correctamente, realizando las pruebas de *Throttler* y *timeout* mencionadas anteriormente.

- **Nuevo servicio de tercero**: En primer lugar, se envía un pedido que corresponde al servicio de tercero a crear y se verifica que de error. Luego, a través del *Admin,* se da de alta y se envía un pedido, comprobando que responde correctamente.

Anexo 8. Test de performance

A continuación, se brinda información complementaria de las pruebas de performance.

Datos generales

- Los componentes que tienen como punto de entrada una cola de mensajes tienen quince hilos para el procesamiento.
- Para la obtención de las medidas de uso de CPU y memoria se utilizó la herramienta *NewRelic* [16].
- Las pruebas fueron realizadas repetidas veces con el fin de verificar que los resultados obtenidos no fueron una excepción.
- Todas las pruebas fueron ejecutadas durante 5 minutos.

Test de carga

Cada hilo envía un nivel por segundo, estos niveles contienen una tarea con un pago. Los pagos tienen un tiempo de procesamiento de cinco segundos, y el conjunto de pagos se distribuyen de la siguiente manera:

- UTE 40%
- BPS 30%
- OSE 30%

En este caso el tiempo de procesamiento de los servicios externos es fijo y su demora es:

- UTE 1 segundo
- OSE 0.5 segundos
- BPS 1.25 segundos

Para cada prueba se obtuvieron las siguientes métricas del conjunto de pagos:

- Tiempo máximo
- Tiempo mínimo
- Tiempo promedio
- Media

A continuación, se muestran los resultados obtenidos con 15 y 22 hilos. A partir de 23 hilos el sistema comienza a degradarse y los pagos no se procesan en el tiempo establecido.

Para cada prueba se grafican la cantidad de pagos en procesamiento dentro del sistema en función del tiempo. Por otro lado, se grafica la cantidad de pagos en cada franja definida en la sección 5.3.2.2.

Resultados con 15 hilos

En este caso se procesan un total de 4500 pagos. Todos los pagos fueron procesados dentro del tiempo establecido.

El siguiente cuadro muestran las métricas obtenidas, se puede apreciar que el tiempo máximo de procesamiento de un pago es 2.94 segundos, esto es un indicio que el sistema tiene capacidad para procesar más pagos.

Tiempo máximo	Tiempo mínimo	Tiempo promedio	Mediana
0:00:02.94	0:00:00.74	0:00:01.64	0:00:01.69

En la Figura 55 se muestra la distribución de los pagos en las distintas franjas de tiempos. Como se puede observar, la mayoría de los pagos se encuentran en la franja de tiempo entre el promedio y la media, este es otro indicio para decir que el sistema es capaz de procesar una mayor cantidad de pedidos.

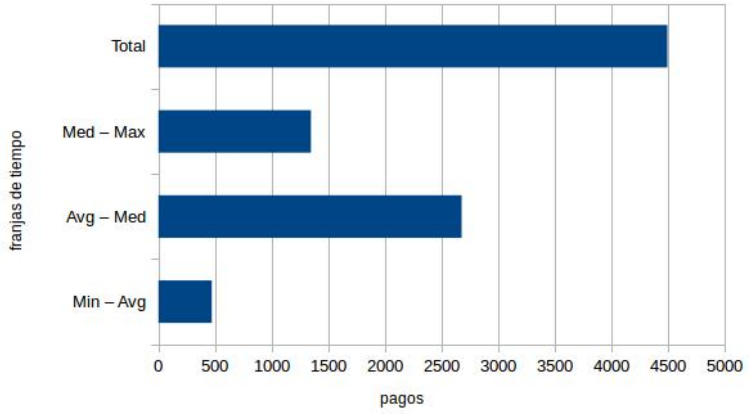

Figura 55 - Cantidad de pagos por franja

En la Figura 56 se grafica la cantidad de pagos concurrentes a lo largo del tiempo, ésta oscila entre 40 y 70, teniendo algunos valores mayores a 70. La razón de esta oscilación puede ser debido a varios factores, en un ambiente de ejecución los tiempos de procesamiento no son constantes presentándose pequeñas variaciones, a su vez el tiempo de procesamiento varía según el servicio externo. Esto genera que en ciertos momentos el sistema esté más saturado, con lo que demora más en responder y como consecuencia se tienen mayor cantidad de pedidos en simultáneo.

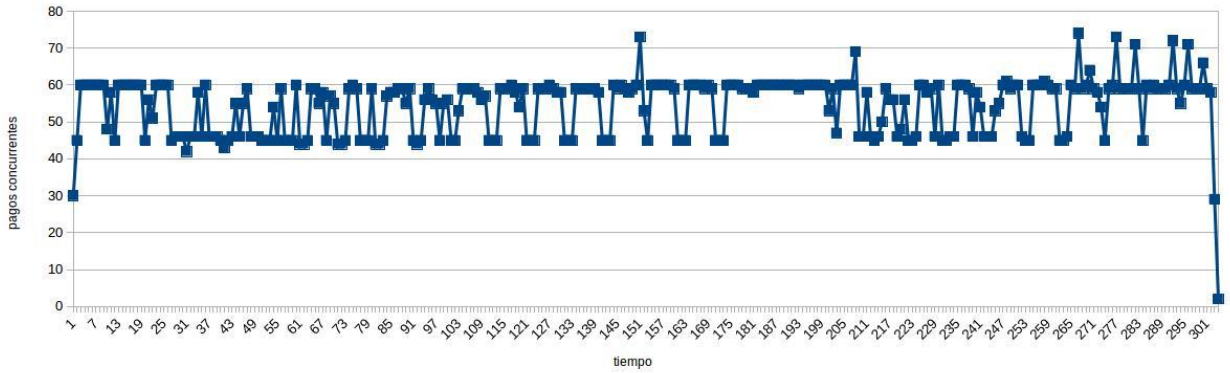

Figura 56 - Pagos concurrentes

Durante la prueba se controlaron los recursos utilizados, el uso de CPU fue menor a un 20% y la memoria consumida no varió. A partir de estos resultados y las métricas obtenidas se concluye que el sistema tiene capacidad para mayor procesamiento de pedidos, por lo que se aumentó la cantidad de hilos y como consecuencia la cantidad de pedidos concurrentes que se procesan en el sistema.

Resultados con 24 hilos

A partir de la prueba realizada con 22 hilos con una instancia de *Level Coordinator,* punto en el cual se obtuvo la máxima cantidad de pedidos concurrentes en el sistema respetando el SLA acordado, se pasa a ejecutar pruebas con dos instancias de este componente para verificar que está siendo el cuello de botella en el sistema.

A continuación, se exponen los resultados obtenidos al ejecutar las pruebas con 24 hilos y dos instancias de *Level Coordinator.*

Teniendo una cantidad de hilos mayor, se logró tener un resultado exitoso en cuanto al SLA acordado, procesando 7200 pagos en 5 minutos. Como se puede ver en los tiempos presentados a continuación, se obtuvo un tiempo máximo menor al obtenido en el caso con 22 hilos, por lo cual se confirma la hipótesis inicial sobre el cuello de botella.

Tiempo máximo	Tiempo mínimo	Tiempo promedio	Mediana
0:00:03:15	0:00:00:78	0:00:01:22	0:00:01:60

La Figura 57 muestra la distribución de los pagos en las tres franjas definidas en la sección 5.3.2.2, se puede apreciar que la mayoría de los pagos se encuentran en la franja de tiempo Med – Max. Esto se debe a que dicha en franja trascurre mayor cantidad de tiempo, sumado a que el tiempo máximo de procesamiento es relativamente bajo.

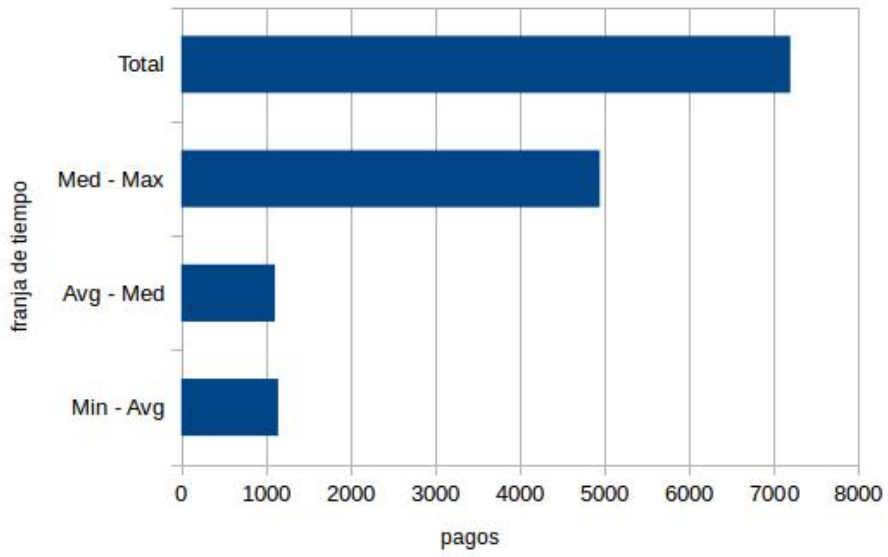

Figura 57 - Cantidad de pagos por franja

En la Figura 58 se muestra la cantidad de niveles concurrentes a lo largo del tiempo, teniendo oscilaciones entre los 60 y 120. Esta cantidad aumentó en comparación a la prueba con 22 hilos.

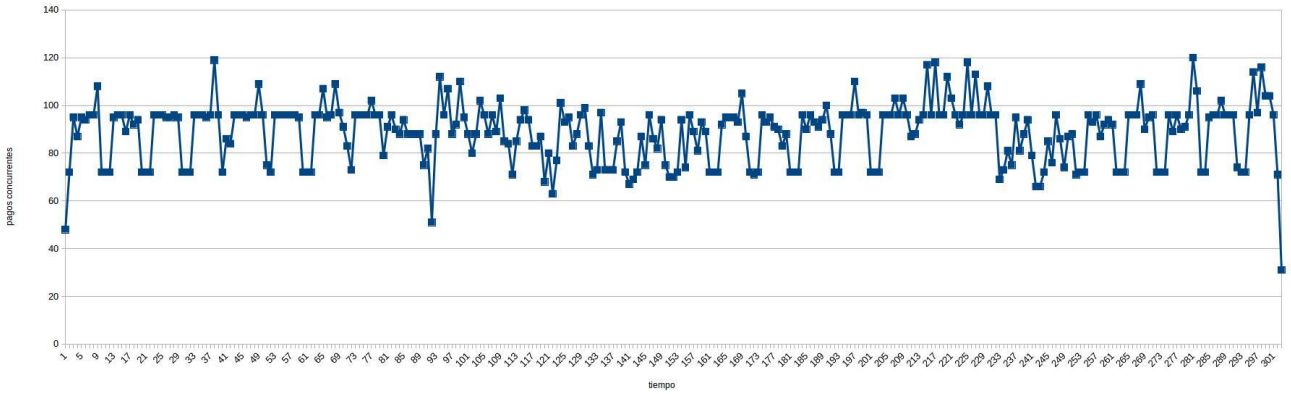

Figura 58 - Pagos concurrentes

En la Figura 59 se pueden observar dos ejecuciones de esta prueba, donde la CPU estuvo al rededor del treinta por ciento en ambas pruebas y la memoria se mantuvo casi constante.

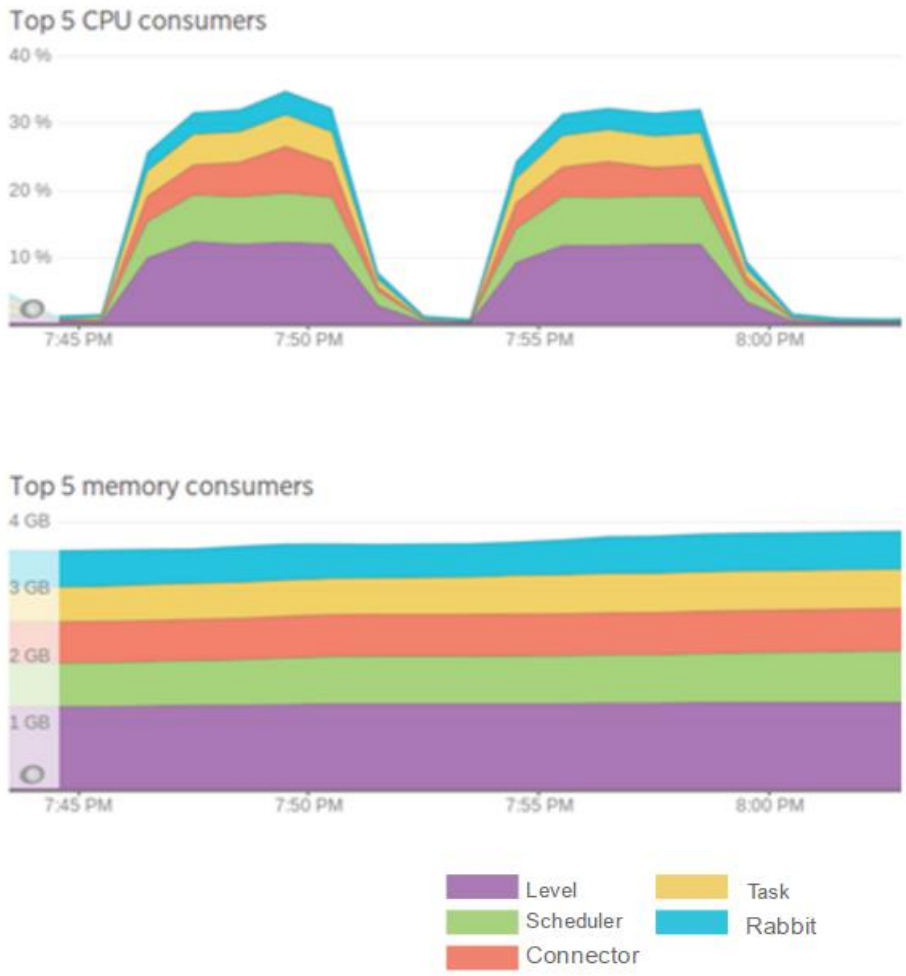

Figura 59 - Consumo de CPU y memoria

Anexo 9. Configuración del Ambiente de desarrollo

Para configurar el ambiente de desarrollo se requiere instalar varias tecnologías, a continuación, se presentan los comandos necesarios. Cuando se hace referencia a `$DIR` indica que es la ruta absoluta donde se encuentra el proyecto. La guía de configuración es para Ubuntu 14.04.

1. Java 8

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
sudo apt-get install oracle-java8-set-default
```

Referencia: http://tecadmin.net/install-oracle-java-8-jdk-8-ubuntu-via-ppa/

2. Docker

Instalación	sudo apt-get update sudo apt-get install docker.io
Ejecución del demonio	sudo service docker start
Prueba	sudo docker run hello-world
Para poder ejecutar docker sin sudo, agregar el usuario "USERNAME" al grupo	sudo gpasswd -a ${USERNAME} docker sudo service docker restart Reiniciar la sesión

Referencia: https://docs.docker.com/engine/installation/linux/ubuntulinux/

2.1. Configurar IP

Ejecutar ifconfig y verificar que la ip de docker0 sea 172.17.42.1. En el caso que sea diferente ejecutar lo siguiente, partiendo que $OLD_IP es la ip de docker0:

```
ip addr del $OLD_IP/16 dev docker0
ip addr add 192.168.5.1/24 dev docker0
ip link set dev docker0 up
sudo service docker restart
```

3. Maven

```
sudo apt-add-repository ppa:andrei-pozolotin/maven3
sudo apt-get update
sudo apt-get install maven3
```

4. Compilación

```
sh $DIR/scripts/clean.sh
sh $DIR/scripts/compile.sh
```

5. Script para bajar imágenes y correr containers

Para correr todas las imágenes ir al directorio del proyecto y correr el script, puede requerir correrlo como *sudo*.

```
sh $DIR/scripts/create_containers.sh
```

6. Parar o iniciar Containers

En el directorio `$DIR/scripts` existen dos archivos *stop.sh* y *start.sh* para parar e iniciar los *Containers* respectivamente.

7. Administración, monitoreo y testing

7.1. DockerUI

Es una herramienta para el manejo de las imágenes y contenedores de *Docker.*
Instalar la herramienta, bajando la aplicación de Chrome.

Editar el archivo: `/etc/default/docker agregando:`

```
DOCKER_OPTS='-H tcp://0.0.0.0:2375 -H
unix:///var/run/docker.sock'
```

Reiniciar docker: `sudo service docker restart`
Prueba en el navegador: http://localhost:2375/_ping y se debe de recibir OK

Refernecia:https://chrome.google.com/webstore/detail/simple-docker-ui/jfaelnolkgonnjdlkfokjadedkacbnib?hl=en

7.2. New Relic
Se debe ejecutar los siguientes comandos:

```
echo 'deb http://apt.newrelic.com/debian/ newrelic non-free' | sudo
tee /etc/apt/sources.list.d/newrelic.list

wget -O- https://download.newrelic.com/548C16BF.gpg | sudo apt-key
add -

apt-get update

apt-get install newrelic-sysmond

nrsysmond-config --set
license_key=5886440b64759333e0983f44c4bfa6f06a591acd

/etc/init.d/newrelic-sysmond start

sudo usermod -a -G docker newrelic

sudo service docker restart

/etc/init.d/newrelic-sysmond restart

cd $DIR/scripts

sh start.sh
```

Referencia:https://docs.newrelic.com/docs/servers/new-relic-servers-linux/installation-configuration/servers-installation-ubuntu-debian#apt

7.3. Postman

Mediante esta herramienta se ejecutaron los diferentes casos de prueba, probando los distintos escenarios. Es una aplicación de google Chrome.
Se dispone de un usuario con los casos de prueba realizados:
Usuario: proyectomicroservice@gmail.com
Clave: proyecto2016

Además, se cuenta con un proyecto que puede ser importado desde el *Postman*, este se encuentra en `$DIR/test`
Referencia:
https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en

7.4. SoapUI

Esta herramienta es necesaria para ejecutar los test de performance. Se cuenta con un proyecto que contiene las pruebas de performance realizadas, este se encuentra en `$DIR/test`

Referencia: `https://www.soapui.org/`

Anexo 10. Configuración del ambiente de deploy

Este ambiente no contiene el código fuente de los componentes, éste está empaquetado en cada imagen descargada. El repositorio utilizado para el manejo de las imágenes es *Docker Hub*, el cual puede ser actualizado desde el ambiente de desarrollo generando una nueva versión, subiéndola al repositorio.

Los datos para acceder al repositorio son:

Usuario: proyectomicroservices2016

Clave: proyecto2016

Mail: proyectomicroservice@gmail.com

Para crear este ambiente es necesario tener instalado *Docker*, luego se debe ejecutar los siguientes comandos utilizando los datos anteriores:

```
docker login
```

```
sh $DIR/scripts/install.sh
```

Con el ultimo se descargan las imágenes y se corren los *Containers*, dejando el ambiente listo para ejecutar pruebas.