SLA Enforcer

Proyecto de Grado 2014

Supervisor:
Guzmán Llambías
Usuario:
Alejandro Mateo
Tribunal:
Andrea Delgado
Bruno Rienzi
Antonio Mauttone

Estudiantes:
Andrés Pereiro
Nicolás Rodríguez Moré

Resumen

Este proyecto de grado se enmarca en el contexto redes de pagos y cobranzas que interactúan de forma intensiva con servicios de terceros. En una red de pagos y cobranzas existen múltiples locales de pago distribuidos geográficamente en distintos puntos del país, a los cuales concurren usuarios para la realización de un conjunto determinado de trámites (pagos de facturas, cobro de dinero, giros, entre otros).
Los principales desafíos encontrados en este tipo de contexto se basan en la heterogeneidad de los servicios de terceros, destacándose la variabilidad de los tiempos de respuesta, la disponibilidad de los servicios, los *Service Level Agreements* (SLAs) existentes entre los distintos sistemas y el manejo de la consistencia de los datos involucrados. Un SLA es un contrato formal acordado entre dos partes, generalmente un proveedor de servicios y un consumidor de servicios.
Dados los problemas identificados, se realizó un relevamiento de la literatura existente y en base a ello, se planteó una solución que contemplara los requerimientos identificados. La solución propuesta se apoya en primera instancia en un modelo de calidad de servicios para la definición de los SLAs existentes. En segundo lugar, define un algoritmo de scheduler y políticas de ejecución para el procesamiento de las solicitudes teniendo en cuenta los SLAs definidos y la naturaleza transaccional de los servicios involucrados. En tercer lugar, se establecieron mecanismos de tolerancia a fallos y consistencia de datos que garantizan el correcto funcionamiento del negocio frente a fallos inesperados en los servicios y en la red de conectividad.
A nivel de implementación, se desarrolló un prototipo de la solución sobre la plataforma ESB Switchyard que contempla los requerimientos y un conjunto de casos de prueba utilizados para validar algunos aspectos no funcionales del sistema.

Palabras clave: Service Oriented Arquitecture, Modelo de calidad, Service Level Agreement, Enterprise Service Bus, Switchyard, Enterprise Integration Patterns, Camel.

Contenido

1 Introducción 4

1.1 Objetivos 4

1.2 Resultados alcanzados 5

1.3 Organización del documento 5

2 Marco conceptual 6

2.1 Arquitecturas Orientadas a Servicios 6

2.2 Composición de servicios 6

2.3 Consistencia de datos en SOA 9

2.4 Calidad de servicios en SOA 12

2.5 Tecnologías para el desarrollo de SOA 15

2.6 Enterprise Integration Patterns 19

2.7 Herramientas de desarrollo 24

3 Análisis de requerimientos 26

3.1 Descripción del contexto 26

3.2 Identificación de la problemática 27

3.3 Solución a los problemas 27

3.4 Requerimientos 30

3.5 Alcance 34

3.6 Aplicación de conceptos 35

4 Diseño de la solución 38

4.1 Modelo de calidad definido 38

4.2 Arquitectura 40

5 Implementación 55

5.1 Tecnologías 55

5.2 Uso de tecnologías 57

5.3 Aspectos de implementación 59

5.4 Metodología de desarrollo 67

5.5 Pruebas del sistema 67

6 Gestión del proyecto 70

7 Conclusiones y trabajo futuro 71

7.1 Conclusiones 71

7.2 Trabajo Futuro 72

8 Referencias 75
Anexos 79
Anexo A: Mediciones 79
Anexo B: Detalles de los componentes del sistema 81
Anexo C: Pruebas del sistema 85
Anexo D: Configuración del sistema 91
Anexo E: Extensión del sistema 96
Anexo F: Gestión detallada del proyecto 98

1 Introducción

Este proyecto de grado se enmarca en el contexto de una red de pagos y cobranzas que interactúa de forma intensiva con servicios de terceros. En este tipo de redes existen múltiples locales de pago distribuidos geográficamente en distintos puntos del país, a los cuales concurren usuarios para la realización de un conjunto determinado de trámites (pagos de facturas, cobro de dinero, giros, entre otros). Los locales de pago funcionan como terminales y no son capaces de efectivizar por sí solos los distintos trámites que desean realizar los clientes. Para ello se comunican con un servidor central el cuál es el encargado de procesar todos los pedidos provenientes de los locales y hacerlos efectivos invocando a los correspondientes servicios de los sistemas de terceros involucrados. En este contexto altamente distribuido, existen desafíos relativos a tiempos de respuesta, tolerancia a fallos y consistencia de datos que debe tenerse en cuenta por el negocio.

En primer lugar, la interacción de los locales con el servidor central debe hacerse efectiva especificando un tiempo de respuesta máximo, mientras que los servicios de terceros procesan las solicitudes de pago con sus propios tiempos de respuesta. En ese sentido, coordinar el procesamiento de todos los pagos de un cliente frente a los servicios en un tiempo dado es un desafío a resolver. Por otro lado, fallas en la conectividad con el servicio pueden provocar inconsistencias en los datos, por lo que deben ser aplicados mecanismos de tolerancia a fallos que garanticen mantener su consistencia. Asimismo, la indisponibilidad de los servicios tiene un impacto directo en el negocio, por lo que son necesarios mecanismos que permitan reducirlo a su mínima expresión para al funcionamiento normal de la red. Por último, la heterogeneidad en la naturaleza de los servicios no permite la definición de una única solución estándar, por lo que deben ser tenidos en cuenta diferentes escenarios de uso. En este contexto, este proyecto se propone proveer soluciones basadas en middleware de tipo ESB, que permitan resolver los desafíos planteados.

1.1 Objetivos

El objetivo general del proyecto consiste en diseñar e implementar un *framework* basado en ESB que permita contemplar los distintos *Service Level Agreement* (SLA) existentes en el contexto de una red de locales de pagos y cobranzas, así como también proveer mecanismos de tolerancia a fallos para mantener la consistencia de los datos y garantizar el funcionamiento del negocio ante caídas inesperadas de los servicios. Para poder llevar a cabo este objetivo, se definen los siguientes objetivos específicos:

- Analizar la realidad de una red de pagos y cobranzas y profundizar en los desafíos existentes.
- Estudiar la bibliografía en busca de soluciones teóricas a los problemas existentes.
- Definir un modelo de calidad que sirva como base conceptual para establecer los SLA existentes
- Definir un algoritmo de Scheduler, una forma de componer los pedidos, ordenarlos y despacharlos para cumplir con los SLA definidos contemplando los diferentes tipos de servicios que existen en la realidad.
- Proponer soluciones que permitan garantizar la consistencia de datos teniendo en cuenta la naturaleza heterogénea de los servicios que participan en la realidad.
- Diseñar e implementar un prototipo que permita validar la propuesta de solución.
- Acumular experiencia en el uso de Switchyard, Camel y las tecnologías asociadas.

1.2 Resultados alcanzados

El siguiente es un pequeño resumen de los resultados alcanzados durante este proyecto de grado:

- Se desarrolló un modelo de calidad en el cual se contemplara los factores de calidad más relevantes para el sistema.
- Se definieron pautas para mantener la consistencia a las fallas utilizadas en sistemas mediados.
- Se diseñó un algoritmo de Scheduler para el procesamiento de pedidos, teniendo en cuenta los SLA existentes, la disponibilidad de los servicios y los requerimientos de consistencia de datos relevados.
- Se desarrolló una prueba de concepto basada en Switchyard ESB y patrones EIP [38] que implementa el modelo de calidad propuesto, el algoritmo de Scheduler y contempla las pautas de consistencia de datos propuesta.

1.3 Organización del documento

En la sección 2 se presenta el marco conceptual del proyecto, que contiene las definiciones de los conceptos que son referenciados a lo largo del resto del documento.
En la sección 3 se detallan los distintos aspectos relacionados al análisis de la solución, destacándose la descripción detallada del contexto en el que se enmarca la elaboración de este proyecto, el análisis de los requerimientos y la definición del alcance final del proyecto.
La sección 4 incluye el diseño de la solución al problema planteado, definiendo por un lado el modelo de calidad utilizado y por el otro la arquitectura del sistema, a partir de la definición de las vistas lógica, de componentes, de procesos y despliegue.
En la sección 5 se describe la implementación del sistema, que incluye la utilización de las distintas tecnologías, los aspectos relacionados a la implementación, la metodología utilizada y las pruebas del sistema.
En la sección 6 se detalla los aspectos relacionados a la gestión del proyecto.
Por último en la sección 7 se establecen las conclusiones obtenidas y se muestra el trabajo futuro a realizar tomando como base este proyecto.

2 Marco conceptual

En el marco de este proyecto se estudiaron distintos estándares y tecnologías necesarios para el desarrollo y comprensión de la solución que son presentados en este capítulo. La sección 2.1 brinda una introducción a las Arquitecturas Orientadas a Servicios (SOA), que oficia de disparador para las demás secciones. En las secciones 2.2, 2.3, 2.4 y 2.5 respectivamente, se estudian las composiciones de servicios, la consistencia de datos en SOA, la calidad de los servicios en un contexto SOA y las tecnologías necesarias para el desarrollo de este tipo de arquitecturas. Por último en las secciones 2.6 y 2.7 se presentan los Patrones de Integración Empresarial (EIP) más relevantes utilizados en este proyecto y las herramientas de desarrollo utilizadas para la implementación.

2.1 Arquitecturas Orientadas a Servicios

Una Arquitectura Orientada a Servicios (SOA) es un modelo de arquitectura de software para el desarrollo de sistemas distribuidos cuyo fin es mejorar la eficiencia, agilidad y productividad de las distintas funcionalidades de negocio, posicionando a los servicios como medio principal a través de los cuales se representan. Los principales conceptos que engloba una arquitectura SOA son los servicios, las interfaces bien definidas, la interoperabilidad y el bajo acoplamiento [1].
En el marco de SOA, los servicios son entidades de software autónomas, auto-contenidas e independientes de la plataforma, que proveen funcionalidades de negocio, tienen una interfaz pública y pueden ser descubiertas, invocadas y combinadas de forma dinámica [2][3].
Los servicios pueden ser clasificados de la siguiente manera [4]:

- Básicos (o simples): Servicios sin estado, con alto nivel de reutilización que encapsulan el comportamiento y modelo de los datos, asegurando su consistencia.
- Compuestos: Servicios que permiten combinar la funcionalidad provista por varios servicios (simples o compuestos); representando funcionalidades complejas del negocio con o sin estado en la aplicación.

2.2 Composición de servicios

Una composición de servicios es una forma de combinar un conjunto de servicios, ya sean simples o compuestos, para crear procesos de negocio de más alto nivel. Como resultado de una composición se puede obtener una funcionalidad nueva, la cual no sería posible resolver individualmente por los servicios involucrados en la composición. Existen dos tipos de composiciones: estática y dinámica. Una composición estática es generada en tiempo de compilación mientras que una composición dinámica se genera en tiempo de ejecución tras la invocación de un usuario al servicio compuesto [20].
La composición de servicios juega un rol importante en la implementación de SOA proveyendo los siguientes beneficios [21][22]:

- Mejora de la reutilización. La composición de servicio provee un mecanismo natural para la reutilización de servicios existentes.
- Mejora en tiempos de creación de servicios. La existencia de servicios y la posibilidad de combinarlos permite crear nuevos servicios y soluciones con menos tiempo y esfuerzo.
- Mejora en seguridad y auditoría. La composición de servicios representa un único punto de acceso al conjunto de servicios que este utiliza por lo cual provee un

mecanismo simple para aplicar políticas de seguridad y controlar las métricas de acceso a los servicios.

- Menor duplicación. Un mismo servicio que implementa una cierta funcionalidad puede ser reutilizado en múltiples composiciones.
- Mejora al introducir cambios. Dado que un servicio puede ser utilizado en múltiples composiciones, representa un único punto de cambio al tener que introducir mejoras o correcciones.

Hay dos aspectos que hay que tener en cuenta cuando se quiere componer servicios [22]:

- El diseño. Se trata del diseño de una solución basada en servicios ya existentes. Se define la lista de servicios que participan en la composición y la forma en la que interactúan. En el diseño de la composición de servicios no solo se requiere que se defina la interacción entre los servicios sino también un conjunto de componentes y topologías para su implementación. Hay dos enfoques principales para el diseño de la composición de servicios:
    - Enfoque mediado (*Orchestration Model*). Como se muestra en la figura 1, en este enfoque se asume que un único servicio, el mediador, es el que consume el cliente y este servicio es el encargado de interactuar y controlar la ejecución del resto de los servicios que participan en la composición. La implementación de la composición de servicios es tratada como una caja negra hacia el consumidor. El consumidor invoca a la composición y espera hasta que se complete la ejecución para luego utilizar su resultado.

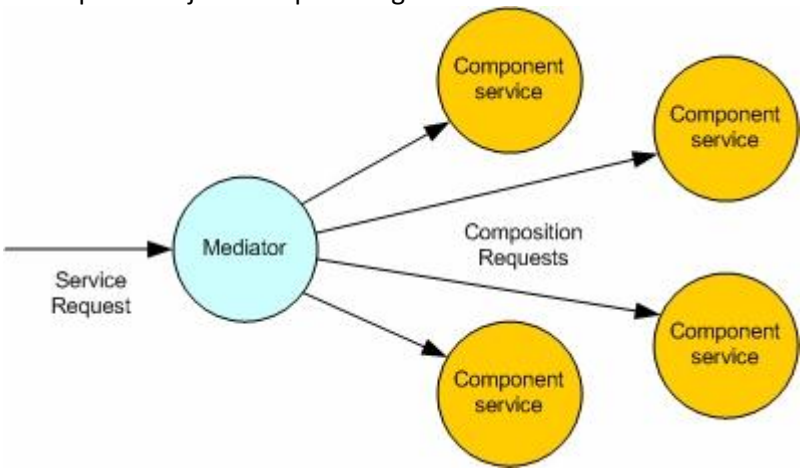

Figura 1: Composición basada en mediación

    - Peer-to-Peer (*Choreography Model*). Como se muestra en la figura 2, en este caso cada servicio participante de la composición puede ejecutar otros servicios. No hay un sistema central que conozca el flujo de invocación.

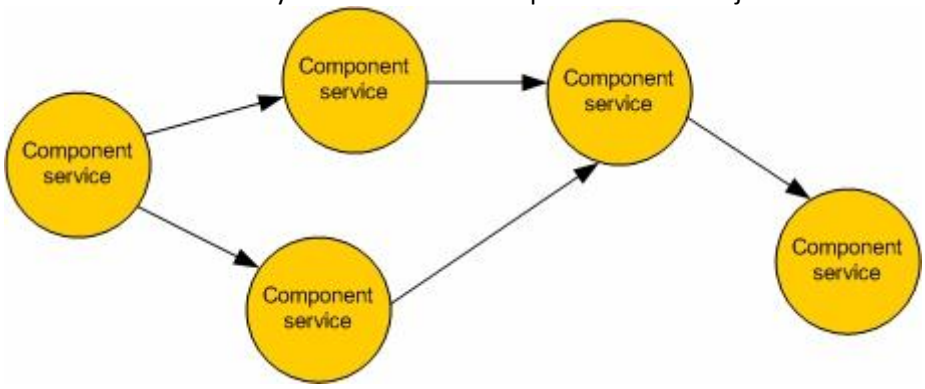

Figura 2: Composición Peer-to-Peer

- La implementación. Este aspecto se ocupa de cómo se logra la coordinación entre los servicios. Las opciones para la implementación de la composición son:

- Programáticamente. En este tipo de implementación se utilizan lenguajes de propósito general para implementar mediación de servicios.
- Eventos, *publish/subscribe*. Este tipo de implementación de composiciones se basa en la interacción entre servicios basada en eventos. Los consumidores del servicio publican eventos en un intermediario del tipo *publish/subscribe (broker)*, el cual es el encargado de entregar estos eventos al proveedor del servicio.
- Orquestación. Este tipo de implementación mejora la implementación programática mediante el uso de lenguaje de orquestación en lugar de lenguaje de programación general para la composición. Esto permite la programación de la composición usando editores visuales, adaptados específicamente para simplificar este tipo de programación. También permite la utilización de la potencia de los motores de orquestación que proporciona capacidades integradas para realizar invocaciones asíncronas, la administración del estado de la ejecución, el apoyo de compensación, etc.

2.2.1 Composición estática y dinámica de servicios

Los servicios web o WS son altamente dinámicos por naturaleza dados los requerimientos cambiantes de la organización. La cantidad de proveedores de servicios está en continuo crecimiento así como la disponibilidad de nuevos servicios que estos exponen. En tal sentido es necesario contar con enfoques que den un alto grado de dinamismo a las composiciones ya que componer servicios de forma manual es un proceso tedioso propenso a errores.
La especificación estática de la composición de servicios no podrá hacer frente a la dinámica del entorno de los servicios. Así, la especificación del proceso de composición de los servicios debe ser generada en tiempo de ejecución en base a la solicitud y ésta debe ser automatizada. Este tipo de enfoque de la composición se llama composición dinámica de servicios y supera las dos limitaciones de enfoque estático a saber, el proceso de especificación composición de servicios manual y estático [23].
La composición dinámica de servicios Web es un proceso de construcción de un nuevo servicio de valor agregado a partir de los servicios disponibles para satisfacer la necesidad funcional del solicitante. Una composición estática se define en el momento del diseño del servicio y no permite cambios en tiempo de ejecución, mientras que una composición dinámica permite adaptarse a cambios en los servicios que la componen y a las necesidades del cliente que utiliza la composición siendo estas necesidades evaluadas con cada petición.
Algunas de las características que se deben de tener en cuenta para la composición dinámica de servicios son [23]:

- Se debe considerar la Calidad de Servicio (QoS) para cada uno de los servicios involucrados.
- Se debe contar con un mecanismo que provea soporte transaccional.
- Garantizar la correctitud de la composición.
- Es necesario encontrar los servicios que serán parte de su composición.
- Permitir adaptarse a los requerimientos del cliente con una mínima intervención.

2.2.2 Relación de los SLA con las composiciones de servicios dinámicos

Un punto importante a tener en cuenta cuando se resuelve qué servicio utilizar en una composición dinámica son las características de QoS de cada servicio. El considerar los aspectos QoS de un servicio cuando se decide cual utilizar en una composición es importante cuando los requerimientos funcionales son satisfechos por más de un servicio. También puede

ser tenido en cuenta el QoS de los servicios de la composición para poder satisfacer las necesidades de QoS del cliente que invoca la composición [24]. QoS y SLA son dos temas analizados en profundidad en la sección 2.4.
En el marco del proyecto esto sería relevante si se contara con varios servicios que cumplieran con la misma funcionalidad y para cumplir con el SLA del pedido el sistema podría invocar a uno de estos servicios dependiendo del QoS medido.

2.3 Consistencia de datos en SOA

En un contexto SOA donde las soluciones son altamente distribuidas existen nuevos elementos a considerar en el desarrollo de un software tales como las transacciones y los SLAs. Las transacciones en SOA son distintas que en los sistemas tradicionales ya que éstas trascienden la frontera de la organización. En esta sección se describe en detalle las transacciones de corta duración y larga duración. Se analiza además los distintos mecanismos de compensación existentes en transacciones de larga duración. Por último, se presenta ACID2 como un conjunto de patrones de diseño de servicios que permiten lograr lo que se denomina como "consistencia eventual", un nuevo paradigma para resolver la consistencia en este contexto.

2.3.1 Transacción de corta duración (atómica).

Una transacción atómica se define como un conjunto de operaciones o tareas que, o se ejecutan todas por completo de manera exitosa, o en caso de que alguna falle se vuelven atrás todos los cambios realizados por las tareas que resultaron exitosas quedando el sistema en su estado inicial [5].
Dicha definición permite que una transacción cumpla con las propiedades ACID [5]:

- *Atomicity* (atomicidad): la transacción se completa de forma exitosa (*commits*) o si falla (*aborts*) todos los efectos son vueltos para atrás (*rollback*).
- *Consistency* (consistencia): Los resultados de una transacción y el estado del sistema luego de ejecutarla siempre son consistentes con las reglas de integridad.
- *Isolation* (aislamiento): Los estados intermedios del sistema producidos mientras una transacción se está ejecutando no pueden ser visibles por otros participantes.
- *Durability* (durabilidad): Salvo en casos de fallas catastróficas los efectos de una transacción exitosa nunca son perdidos y son duraderos en el tiempo.

Las propiedades ACID son aplicables solamente en contextos de transacciones de corta duración, ya que su implementación implica mantener *locks* sobre los distintos recursos involucrados, y no es viable mantenerlos por largos períodos de tiempo.
En los casos que las transacciones atómicas sean distribuidas se utiliza el protocolo Two-Phase Commit para coordinar la ejecución de la transacción. A continuación se da una introducción a dicho protocolo.

2.3.1.1 Two-Phase Commit

El protocolo Two-Phase Commit (2PC) establece los mecanismos de coordinación necesarios para la ejecución de transacciones atómicas distribuidas entre un conjunto de participantes. Para la implementación de este protocolo es necesario que exista un coordinador que gestione su ejecución o cancelación. El protocolo define la utilización de dos fases [6]:

- Fase 1 (*prepare*): el coordinador le pide a los demás participantes de la transacción que indiquen si son capaces de realizar *commit* de la transacción o deben realizar un *rollback*.

- Fase 2 (*commit*): el coordinador decide, en base a la información recabada en la fase 1, si se va a realizar un *commit* o un *rollback* de la transacción. En el caso de que alguno de los participantes no pueda realizar el *commit*, se debe realizar un *rollback*. En caso contrario se puede efectivizar el *commit* de la transacción. El coordinador es quién decide si los participantes deben realizar el *commit* o no, y los participantes le notifican luego de realizada la operación.

En la figura 3 se muestran las dos fases del protocolo, donde intervienen de la transacción dos participantes (A y B) y el coordinador de la transacción (C).

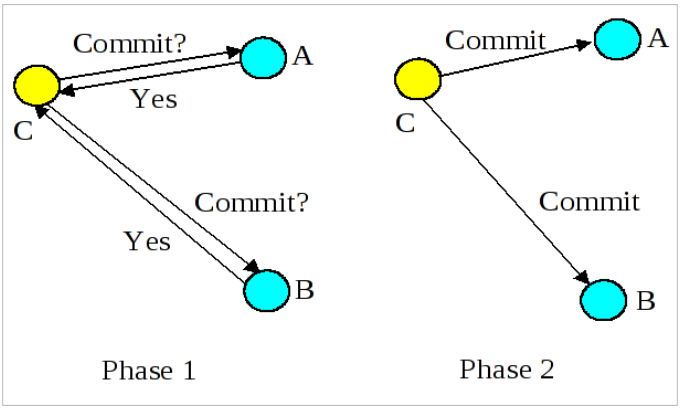

Figura 3: Two Phase Commit

2.3.2 Transacción de larga duración y consistencia eventual

Para entender qué es una transacción de larga duración, es necesario definir qué es el tiempo de vida de una transacción (*Transaction Lifetime*). Éste se define como la mínima cantidad de tiempo por la cual una transacción permanece abierta. De esa forma, una transacción de larga duración puede ser definida como una transacción cuyo *Transaction Lifetime* es largo (horas, días, meses, etc) o indefinido [7].

El problema principal que poseen las transacciones de larga duración es que no pueden ser tratadas de igual forma que las transacciones de corta duración, ya que para poder mantener las propiedades ACID se deberían establecer *locks* por largos períodos de tiempo sobre los distintos recursos involucrados en la transacción y esto volvería inutilizable al sistema. Sin embargo las transacciones de larga duración deben poseer algún mecanismo para mantener consistencia sobre los datos en la mayoría de las situaciones.

La Consistencia Eventual plantea que el sistema de almacenamiento garantice que si no se realizan actualizaciones sobre un objeto, eventualmente todos los accesos al mismo retornarán el valor actualizado por última vez. En el caso de que no ocurran fallas, el tamaño máximo de la ventana de inconsistencia puede ser determinado en función de factores tales como los retrasos en la comunicación o la carga existente en el sistema [8]. Un ejemplo de sistema existente que implementa mecanismos de consistencia eventual es el *Domain Name System* (DNS), en el cual las actualizaciones a un nombre de dominio son llevadas a cabo siguiendo un patrón determinado, y mediante la utilización de cachés distribuidas, eventualmente todos los clientes verán la actualización.

Lograr esa consistencia eventual sería deseable en un contexto SOA que involucre transacciones de larga duración. Para obtener un sistema que garantice una consistencia eventual sobre los datos es necesario contar con algún mecanismo de compensación. Existen dos enfoques posibles: compensación programada y compensación por diseño.

2.3.2.1 Compensación programada

Los distintos servicios involucrados deben poseer implementados mecanismos de compensación, los cuales deben garantizar que ante fallos, los cambios realizados por los servicios involucrados en la composición sean revertidos completamente. De esta forma, cuando ocurre una falla en una transacción de larga duración tenemos un mecanismo para volver atrás todos los cambios realizados. Los mecanismos de compensación pueden ser muy variados, desde proveer implementaciones de servicios para anular cambios realizados en el sistema hasta proveer sistemas que permitan una gestión manual de las fallas [4]. En particular, el lenguaje WSBPEL [9] (un lenguaje para la especificación de composiciones de servicios) posee soporte para compensaciones programadas, ya que permite la definición de manejadores que ejecutan servicios compensatorios ante la ocurrencia de fallas. La figura 4 muestra un ejemplo de proceso el cual consiste en realizar tres llamadas en paralelo a tres servicios distintos, para realizar en simultáneo la reserva de un vuelo, el hotel y un taxi:

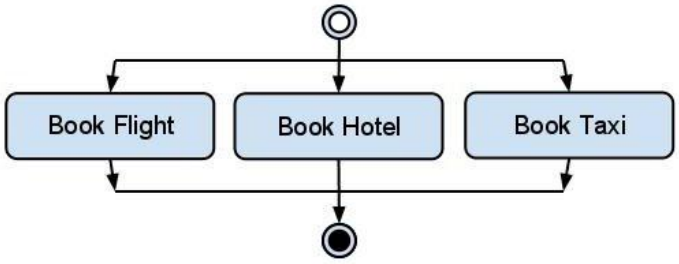

Figura 4: Ejemplo WSBPEL

En el caso de que falle cualquiera de las invocaciones todo el proceso debe ser cancelado por completo. Si los servicios involucrados exponen operaciones para la realización de cancelaciones es posible definir en el proceso un flujo de ejecución ante una falla que programáticamente ejecute las compensaciones correspondientes.

2.3.2.2 Compensación por diseño

La compensación por diseño plantea una solución diferente a la compensación programada. En este caso no se poseen mecanismos de compensación, sino que se diseñan y programan los servicios de manera que no sea necesaria la utilización de transacciones u operaciones de compensación.

Para lograr la consistencia eventual en este contexto es necesario que los servicios cumplan con las siguientes propiedades:

- Asociatividad
- Conmutabilidad
- Idempotencia
- Capacidad Distribuida

Dichas propiedades son conocidas como ACID2, y surgen a partir del principio CALM (Consistency As Logical Monotonicity). Para comprender el principio CALM es necesario primero definir qué es un programa monotónico. Un programa es monotónico si satisface que al agregar elementos a la entrada del mismo solamente se puede aumentar su salida [11]. Por el contrario, un programa no monotónico puede tener que retirar o eliminar elementos de una salida previa si se agregan nuevos elementos a la entrada.

El principio CALM establece que:

- Un programa o fragmento de código lógicamente monotónico distribuido es eventualmente consistente sin la necesidad de proveer ningún tipo de protocolo de coordinación.

- Se puede garantizar la consistencia eventual de cualquier programa protegiendo los fragmentos de código no monotónicos con protocolos de coordinación [11].

En la sección 2.3.3 se describe en detalle las propiedades ACID2 y se brinda un ejemplo de utilización de cada una de ellas.

2.3.3 ACID2

ACID2 es un conjunto de propiedades que flexibilizan la rigidez que poseen las propiedades ACID tradicionales de forma de poder lograr con restricciones razonables una consistencia eventual en un entorno distribuido sobre los datos en un contexto SOA [4].
Dichas propiedades son:

- Asociatividad: los servicios pueden ser combinados indistintamente produciendo el mismo resultado sobre el sistema. A modo de ejemplo, si se desea invocar a los servicios $s_1, s_2\ y\ s_3$, el orden de invocación no va a afectar el resultado final:
$$s_1 \circ (s_2 \circ\ s_3) =\ (s_1 \circ s_2) \circ\ s_3$$
- Conmutatividad: llevando la definición matemática al contexto SOA; si tenemos N servicios a ejecutar, el orden de ejecución de los mismos no va a generar estados diferentes en el sistema. De esta forma se puede combinar las invocaciones procurando optimizar la performance y los tiempos de ejecución [4]. Si tomamos dos servicios $s_1\ y\ s_2$, tenemos:
$$s_1 \circ s_2 =\ s_2 \circ s_1$$
- Idempotencia: La idempotencia garantiza que, ante múltiples invocaciones idénticas a un servicio, no se produzcan efectos no deseados y, en el caso de que no fallen todas las invocaciones, se generarán los mismos efectos sobre la lógica que si dicho servicio se invocara una única vez. En particular la idempotencia garantiza que en el caso de que el servidor falle al enviar la respuesta, el cliente podrá reintentar el pedido sin problema hasta recibir la confirmación del servidor [12]. A modo de ejemplo, tomemos el servicio $s_1$invocado con los parámetros x, tenemos que:
$$s_1(x) =\ s_1(x) \circ s_1(x) = \cdots =\ s_1^n(x)$$
- Capacidad distribuida: Este patrón permite que un servicio se mantenga representado con exactamente el mismo contrato incluso en los casos en los que su implementación no es capaz de manejar los requerimientos de procesamiento que necesita para procesar un determinado pedido. Lo que se busca es lograr que la lógica del servicio sea distribuida de forma de poder brindar mejores prestaciones ante requerimientos de procesamiento más estrictos, separando dicha lógica en múltiples lugares físicos. Para implementarlo es necesario agregar procesamiento intermedio que actúe como link entre la parte central de la lógica y las partes que se encuentran distribuidas [12].

2.4 Calidad de servicios en SOA

A continuación se describen las principales características asociadas a la calidad en servicios y los principales modelos de calidad existentes. Se definen algunos conceptos referentes a la calidad de servicios como *Service Level Agreement* y *Quality of Service* y se describe por qué es necesario contar con un modelo de calidad profundizando en dos de los modelos más populares: S-Cube y WS-QM.

2.4.1 Service Level Agreement

Un *Service Level Agreement* (SLA) es un contrato formal acordado entre dos partes (en un contexto SOA, generalmente un proveedor de servicios y un consumidor de servicios). Los SLA

especifican criterios de responsabilidades, garantías y prioridades de manera de establecer un acuerdo en la calidad del servicio que se brinda. Son utilizados para expresar restricciones y pueden contener políticas de penalidades ante la violación de estos acuerdos.
Es importante destacar que la negociación del SLA entre dos partes es esencial para alcanzar un acuerdo de servicio basado tanto en los requerimientos de Calidad de Servicio (QoS) del cliente como en las capacidades del QoS del proveedor [13]. El hecho de depender de servicios externos requiere la necesidad de garantizar cierta calidad de servicio, la cual es establecida en el contrato.

2.4.2 Calidad en servicios

La Calidad de Servicio (QoS) nos permite definir y evaluar los distintos aspectos no funcionales de un servicio. Con el avance de las arquitecturas SOA en los sistemas informáticos y el rol que puede tener la calidad de un servicio en su ciclo de vida, se le ha dado en los últimos años una mayor importancia a esos aspectos no funcionales. El hecho de contar con información sobre la calidad de un conjunto de servicios disponible permite, entre otras cosas [3]:

- Seleccionar o filtrar servicios que sean funcionalmente equivalentes.
- Componer servicios dinámicamente utilizando servicios que posean mejor calidad al momento de ejecutar la composición.
- Garantizar el cumplimiento del SLA. Dado que se conoce la calidad de un servicio dado, se puede establecer si éste cumple o no con los niveles de calidad previamente establecidos con el proveedor del servicio.

Es importante destacar que las distintas propiedades de un servicio consideradas como aspectos de la calidad del mismo pueden ser clasificadas en dos categorías principales:

- Medibles (por ejemplo: *throughput*, tiempo de respuesta, disponibilidad, etc.).
- Caracterizables (por ejemplo: reputación, seguridad, confianza en el proveedor, etc.).

Una de las principales características del proyecto es la importancia que tiene la relación entre el QoS de cada servicio y el impacto de este en el QoS global de la invocación al sistema. El QoS de la invocación al sistema debe ser cuantificable a los efectos de poder establecer si el sistema puede o no ejecutar el pedido.

2.4.3 Modelos de calidad

Para poder determinar la calidad de los servicios es necesario identificar y clasificar factores o atributos de calidad los cuales determinan qué se quiere medir y cómo se deben medir. Estos factores de calidad se organizan y documentan en modelos de calidad [3].
Dos de los modelos más populares son el OASIS WS-QM [15] y el modelo de calidad definido en el proyecto S-Cube (S-Cube QRM [16] y S-Cube QMM [17]). A continuación se da una breve descripción de ambos. En [3] y [14] se realiza una comparación entre ambos modelos.

2.4.3.1 S-Cube Quality Reference Model / Quality Meta-Model

La definición de un modelo de calidad en el proyecto S-Cube fue realizada en varias etapas, destacándose las dos principales: definición de un conjunto de categorías de calidad y de los atributos asociados a las mismas (Quality Reference Model (QRM) [16]); y definición de los conceptos necesarios para especificar la calidad de un servicio y del SLA asociado (Quality Meta-Model (QMM) [17]).
El proceso que llevó a definir a S-Cube QRM fue el siguiente:

i. Recolección y análisis de distintos modelos de calidad existentes en el mercado. Para cada uno de los modelos considerados se tomó una descripción y se generó además una lista de atributos de calidad asociados al mismo.

ii. Análisis de los atributos de calidad. A partir de todos los atributos de calidad obtenidos en la primera fase, se buscó identificar aquellos que eran relevantes para los servicios y las aplicaciones basadas en servicios.
iii. Se agruparon los distintos atributos de calidad en categorías, permitiendo definir el QRM.

Las distintas categorías que se obtuvieron son [3]:

- Desempeño (*performance*): Agrupa los distintos atributos que permiten especificar qué tan bien se desempeña un servicio. Dos de los atributos que se incluyen en esta categoría son: Tiempo de respuesta (*response time*) y rendimiento (*throughput*).
- Confiabilidad (*dependability*): refiere a la capacidad que posee un servicio de brindar confiabilidad. Los atributos principales dentro de esta categoría son: disponibilidad (*availability*), fiabilidad (*reliability*).
- Seguridad: agrupa los distintos atributos que permiten establecer la seguridad de un servicio, entre los cuales se encuentran autenticación, autorización, confidencialidad, etc.
- Calidad de datos: Engloba aquellos atributos que permiten establecer la calidad de los datos que son pasados por parámetro o devueltos por un servicio (por ejemplo: la política que se posee sobre los datos -*data policy*-).
- Usabilidad: Son los atributos que permiten medir la facilidad de uso de un servicio para un usuario.
- Configuración: Atributos que determinan la forma en la que se configura un servicio para funcionar (por ejemplo, nivel de servicio) o que determinan si el nivel de funcionalidad establecido fue brindado efectivamente por el servicio (por ejemplo, estabilidad y reputación).
- Calidad de red e infraestructura: Se consideran atributos relacionados a la calidad de la infraestructura existente debido a que éstos pueden afectar los distintos parámetros de calidad de un servicio. Dentro de los atributos se encuentran: ancho de banda, calidad del contexto de uso, costo y otros.

Una lista completa de los atributos definidos y el significado de los mismos se puede obtener en [18].

El S-Cube QMM establece un conjunto de entidades (Actores) que son las encargadas de establecer los SLA (por ejemplo: Usuarios, Compañías o Agentes). Dichos Actores participan en la negociación de los servicios (por lo tanto poseen una estrategia de negociación), tomando un determinado rol (Proveedor, Solicitante o Tercero). Los proveedores brindan un servicio con un modelo de costos asociado, en los cuales se mapean atributos de calidad a cada uno de los costos del modelo.

Los distintos servicios poseen Prestaciones de Calidad (*Quality Capabilities*) específicas, que son ofrecidas por los mismos en las especificaciones de calidad (*Quality Specifications*) junto con un período de validez asociado (importante para contextos altamente dinámicos).

Cuando una negociación de servicios es exitosa, las prestaciones de calidad son mapeadas a Niveles de Servicio (*Service Levels*).

La especificación de la calidad de un servicio se hace a través de un conjunto de reglas y restricciones [17].

2.4.3.2 OASIS WS-QM

WS-QM establece un modelo de calidad en el que existen tres componentes principales: los Actores de Calidad, las Actividades de Calidad y los Factores de Calidad en el cual los Actores realizan Actividades, como se muestra en la figura 5, las cuales utilizan ciertos Factores de calidad [15][19].

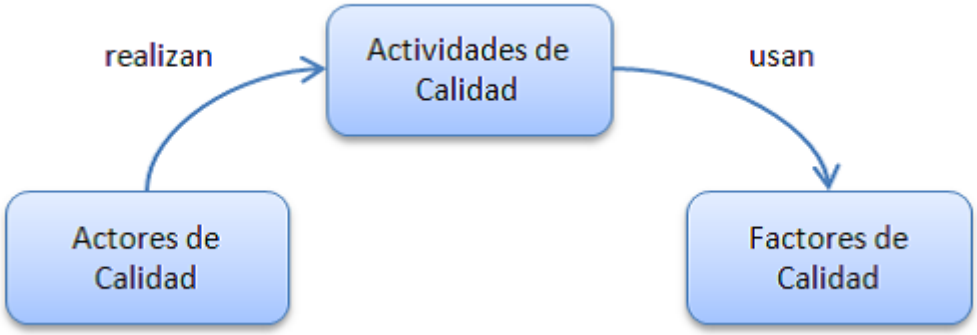

Figura 5: Modelo de Calidad para Web Services de OASIS

Los factores de calidad se dividen en dos grupos principales dependiendo de si tienen una perspectiva del negocio o del sistema [19]:

- Factores asociados a la calidad del negocio: en esta clasificación se encuentran los factores de calidad que apuntan a medir y permitir seleccionar un servicio de acuerdo al valor que éste aporta al negocio. Algunos de los sub-factores que están comprendidos en este grupo son el precio, la reputación del proveedor del servicio y las penalizaciones e incentivos existentes.
- Factores asociados a la calidad del sistema: Se dividen en dos subgrupos, los considerados como una parte invariante del sistema y los variables. El primer subgrupo engloba a factores comprendidos en las áreas de interoperabilidad, procesos de negocio, gestión y seguridad; y son aquellos factores cuyo valor se determina y permanece incambiado tan pronto como se desarrolla el servicio. El segundo subgrupo involucra a los factores relacionados con los acuerdos de niveles servicio y son factores cuyos valores pueden verse modificados dinámicamente en tiempo de ejecución, por ejemplo el tiempo de respuesta o la disponibilidad del servicio.

2.5 Tecnologías para el desarrollo de SOA

En esta sección se detallan las distintas tecnologías necesarias para el desarrollo de SOA. En la sección 2.5.1 se brinda una definición de lo que es un Web Service y en la sección 2.5.2 se presentan los distintos estándares existentes para el manejo de transacciones en Web Services. La definición de Web Service sirve de base para introducir los conceptos de Service Component Arquitecture (SCA) y Enterprise Service Bus (ESB) en las secciones 2.5.3 y 2.5.4.

2.5.1 Web Service

Un *Web Service* es un componente de software diseñado para permitir la interoperabilidad entre dos sistemas que desean interactuar a través de una red. Dicho componente brinda una interfaz que respeta el formato *Web Service Definition Language* (WSDL) y sistemas externos pueden interactuar con él utilizando mensajes con el formato establecido por el protocolo *Simple Object Address Potocol* (SOAP), generalmente transmitidos utilizando un conjunto de estándares web, entre los cuales se encuentran HTTP y XML[25].

WSDL es un formato XML que permite describir la interfaz de un servicio, expuesto a través de la red, como un conjunto de puntos de acceso (*endpoints*) que operan sobre mensajes que contienen información. Las operaciones y los mensajes son descritos de manera abstracta y luego son ligados a un determinado protocolo de red y formato de mensajes (en la actualidad solo se especifica a SOAP 1.1, HTTP GET/POST y MIME) para definir el *endpoint* [26].

SOAP es un estándar para codificar mensajes en formato XML el cual es utilizado por las aplicaciones basadas en web services para comunicarse e inter-operar a través de la red [27].

La arquitectura de referencia de Web Services modela la interacción existente entre tres roles: el proveedor del servicio (Service Provider), la o las agencias de descubrimiento de servicios (Discovery Agencies) y el cliente que desea consumir el servicio (Service Requestor). En la figura 6 se muestra la interacción entre los tres agentes involucrados, donde los proveedores de servicios son responsables de publicar los servicios que exponen en las agencias y los consumidores deben ser capaces de encontrar la descripción del servicio (WSDL) que desean invocar consultando a las agencias. Las distintas operaciones entre los agentes intercambian mensajes SOAP [28].

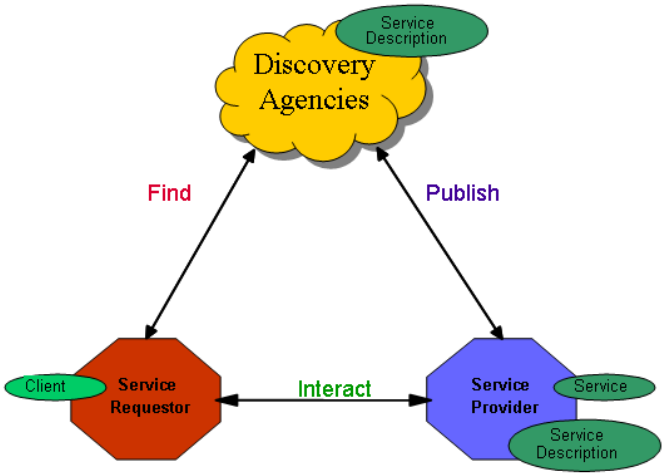

Figura 6. Arquitectura de referencia de Web Services

2.5.2 Estándares para transacciones en *web services*.

Luego de que se logró definir los estándares que rigen la interoperabilidad entre servicios web (SOAP y WSDL); las composiciones de servicios se volvieron una moneda corriente. En particular el grado de complejidad de los procesos de negocio llevó a que en el año 2002 las empresas BEA, IBM y Microsoft definieran un conjunto de especificaciones para el manejo de transacciones en *web services*. Dichas especificaciones fueron WS-Coordination, WS-AtomicTransaction y WS-BusinessActivity.
En el año 2005 los miembros de OASIS decidieron formar un comité (OASIS Web Services Transaction (WS-TX) Technica lCommittee) dedicado a la elaboración de un estándar para el manejo de transacciones en *web services* fuertemente basado en las tres especificaciones mencionadas (dichas especificaciones fueron tomadas como la versión 1.0 del estándar) [29].
A continuación se da una breve introducción a cada uno de los protocolos.

2.5.2.1 WS-Coordination

Este estándar es la definición de un *framework* que permite la coordinación de actividades, mediante el uso de un coordinador y un conjunto de protocolos de coordinación. Los protocolos de coordinación que pueden ser utilizados en este *framework* pueden estar orientados tanto a operaciones de corta duración como a actividades de negocio de larga duración. Dos de los protocolos de coordinación existentes que respetan la especificación establecida en WS-Coordination son WS-AtomicTransaction y WS-BusinessActivity.

2.5.2.2 WS-AtomicTransaction

WS-AtomicTransaction es una especificación que extiende WS-Coordination y permite coordinar actividades o servicios que posean las características de ser transacciones atómicas

distribuidas. Este estándar se basa en el protocolo 2PC descrito en la sección 2.3.1.1 de este documento, orientado a transacciones atómicas. Las transacciones atómicas son transacciones de corta duración, y que en general, requieren un alto nivel de confianza entre los participantes. En este tipo de transacciones, todas las acciones tomadas por los distintos participantes de una transacción antes de realizar un *commit* son tentativas; es decir, que no son ni persistidas ni visibles fuera de la transacción. Cuando se finaliza la transacción, la aplicación involucrada realiza un pedido de confirmación al coordinador que le diga si la transacción fue realizada con éxito o no, siendo el coordinador el encargado de validar contra todos los participantes si tuvieron alguna falla en el procesamiento.
Dependiendo de la respuesta de los participantes, ocurre uno de los siguientes casos [6]:

- Caso 1: Todos los participantes responden afirmativamente. En esta situación, el coordinador realiza *commit* de todas las acciones realizadas, comunicándole a los distintos participantes que las acciones pasen de ser tentativas a efectivamente realizadas; pudiéndose visualizar los cambios realizados desde afuera de la transacción.
- Caso 2: Alguno de los participantes aborta o no da respuesta alguna. En este caso el coordinador ordena a todos los participantes que aborten las distintas acciones realizadas, lo que lleva a que para cualquier actor fuera de la transacción, los cambios realizados nunca hubieran sucedido.

2.5.2.3 WS-BusinessActivity

WS-BusinessActivity es una especificación que extiende WS-Coordination y que provee la definición de dos protocolos de coordinación, que son utilizados para la coordinación de actividades de un proceso de negocio y que aplican lógica de negocio para el manejo de excepciones que puedan ocurrir durante la ejecución de las actividades.
Conceptualmente se diferencia de WS-AtomicTransaction en que todas las acciones y cambios son aplicados inmediatamente y son permanentes, y en caso de la ocurrencia de errores se toman acciones compensatorias para el manejo de los mismos (en vez de poseer mecanismos de *rollback*) [32]. WS-BusinessActivity sigue el modelo de compensación programada descrito en la sección 2.3.2.1 de este documento.
WS-BusinessActivity define dos protocolos de coordinación, que son muy similares entre sí pero difieren en la forma en la que un participante se debe comportar durante la actividad [33]:

- Acuerdo de Negocio con Finalización por parte del Participante: permite a un participante determinar cuándo ha completado su parte en la Actividad de Negocio.
- Acuerdo de Negocio con Finalización por parte del Coordinador: requiere que un participante confíe en que el coordinador de la actividad de negocio le notifique cuando no posee más responsabilidades dentro de la actividad.

Además, WS-BusinessActivity define dos tipos de coordinación posibles los cuales pueden ser utilizados en combinación con cualquiera de los protocolos de coordinación definidos anteriormente. Los tipos de coordinación definidos establecen cómo el protocolo de coordinación debe gestionar los resultados de las transacciones realizadas por los distintos involucrados [33]:

- Atomic Outcome: el coordinador debe indicar a todos los participantes de la actividad que deben cerrar la transacción o realizar una compensación (*rollback*), todos por igual.
- Mixed Outcome: el coordinador debe indicar a todos los participantes de la actividad que deben cerrar la transacción o realizar una compensación, pero puede tomar decisiones diferentes para los distintos participantes.

2.5.3 Enterprise Service Bus

Como vimos en la sección 2.1, los principales conceptos que engloba una arquitectura SOA son los servicios, las interfaces bien definidas, la interoperabilidad y el bajo acoplamiento. El encargado de dar soporte a la implementación de las soluciones basadas en SOA es el *Enterprise Service Bus* (ESB).

Un ESB es una plataforma diseñada para proveer la capa de interconexión entre los servicios que permite sobrellevar los problemas tradicionales asociados a la fiabilidad, escalabilidad y disparidad de mecanismos de comunicación teniendo como objetivos principales el bajo acoplamiento y dividir la lógica en partes que sean fácilmente manejables. En esta arquitectura, los servicios se comunican intercambiando mensajes a través del ESB el cual actúa como mediador y permite que se puedan ejecutar diferentes operaciones sobre los mensajes como transformar, routear, seguridad, etc. [34][35].

Como se puede observar en la figura 7, el ESB interconecta aplicaciones y servicios que utilizan tecnologías y protocolos de comunicación diferentes proveyendo la capacidad de rutear los mensajes a través de servicios y transformar los mensajes de servicios con diferentes estructuras de datos.

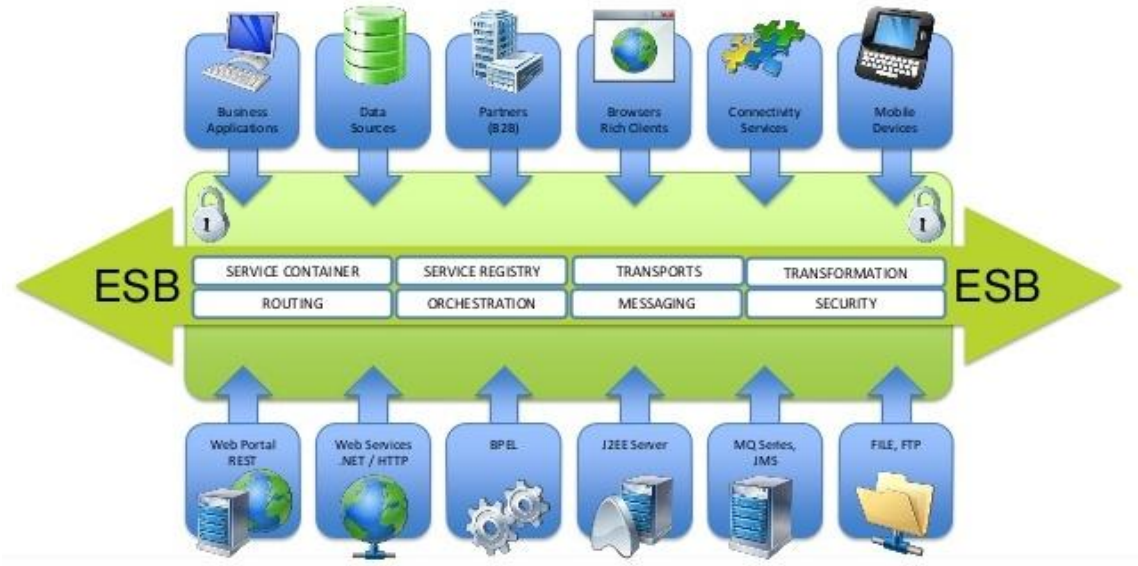

Figura 7: Enterprise Service Bus

2.5.4 SCA

Service Component Architecture (SCA) es un conjunto de especificaciones que describen un modelo para el desarrollo de sistemas y aplicaciones utilizando una arquitectura SOA. La idea base de SCA es que una funcionalidad del negocio puede ser compuesta por una serie de componentes de lógica más específica y sencilla, los cuales son ensamblados para crear soluciones a las necesidades del negocio. Las aplicaciones compuestas (*composites*) pueden ser consumidas como un servicio, exponiéndolas con una interface agnóstica al mecanismo de comunicación que se utilice. A su vez permite que los diferentes servicios declarados en la composición puedan ser consumidos por otros componentes de forma directa sin importar el mecanismo de comunicación utilizados entre estos [36]. En la figura 8 se muestra la arquitectura SCA de referencia. Los elementos que componen SCA son [37]:

- Composite: estructura que permite ensamblar componentes SCA en grupos lógicos. Permite generar composiciones de elementos SCA y puede contener un conjunto de componentes, servicios, referencias y propiedades.

- Componente (component): es el elemento básico de una funcionalidad del negocio en una aplicación SCA. Los componentes proveen y consumen servicios y pueden poseer una implementación determinada. Cada componente puede contener servicios, referencias y propiedades.
- Servicios (Composite Services): Los servicios son utilizados por los componentes para exponer sus funcionalidades como Servicios para que puedan ser utilizadas por otros componentes.
- Referencias (Component References): Las referencias son utilizadas por los componentes para consumir servicios expuestos por otros componentes.
- Propiedades (Properties): Las propiedades son utilizadas por los componentes para configurar valores específicos de su implementación.

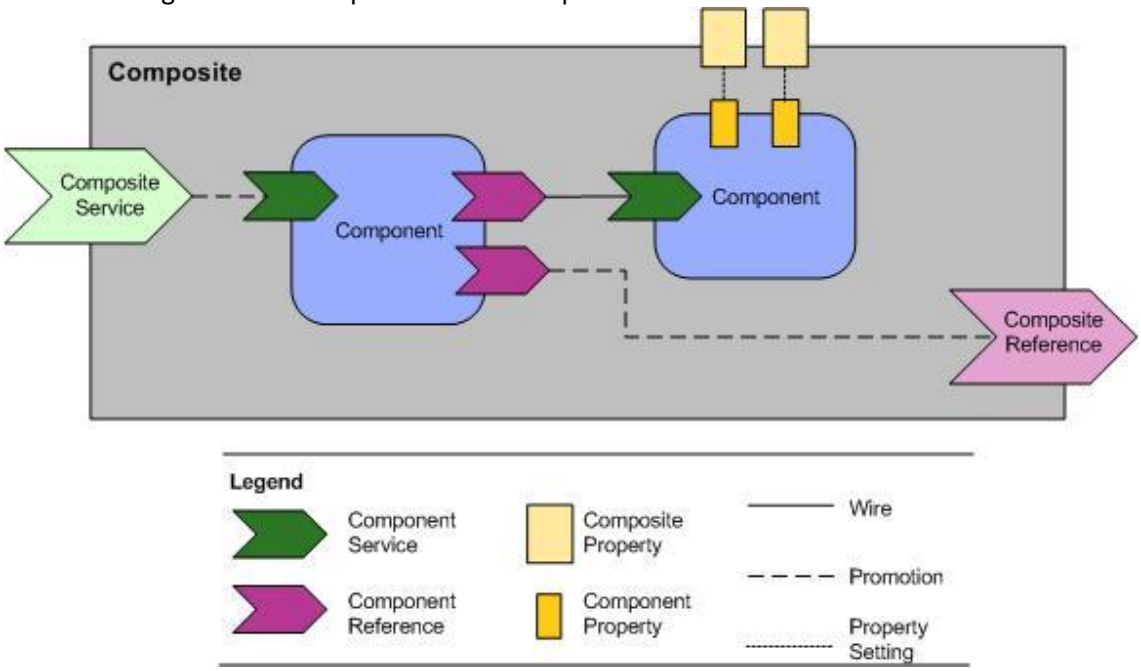

Figura 8: Arquitectura de referencia SCA

2.6 Enterprise Integration Patterns

Los *Enterprise Integration Patterns* (EIP) son un conjunto de patrones de integración de aplicaciones empresariales que son tomados como soluciones a problemas recurrentes en determinados contextos. Poseen la particularidad de ser lo suficientemente abstractos para ser utilizados con la mayoría de las tecnologías de integración existentes y lo suficientemente específicos para guiar tanto a desarrolladores como a diseñadores y arquitectos [38].
A continuación se listan algunos de los patrones más relevantes para el proyecto.

2.6.1 Splitter

El patrón *Splitter* permite separar un mensaje compuesto en un conjunto de mensajes individuales, cada uno conteniendo datos relacionados a cada uno de los ítems del mensaje compuesto original. Este patrón se utiliza para consumir una lista de elementos repetitivos de los cuales cada uno puede ser procesado por separado. Para ello genera un mensaje para cada uno de los ítems [38]. En la figura 9 se muestra el efecto de la utilización de este patrón y en la tabla 1 se detalla la entrada, los parámetros que recibe y la salida esperada.

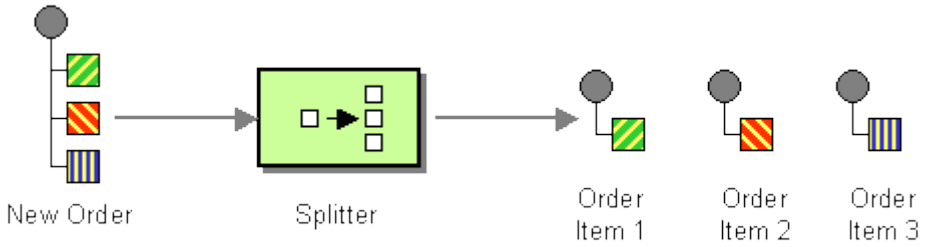

Figura 9: Patrón Splitter

Entrada	Mensaje a procesar
Parámetros:	
splitStrategy	Estrategia para la realización del Split. Permite especificar de qué forma se parte el mensaje en submensajes.
Salida	Mensajes individuales resultantes de aplicar la estrategia de separación al mensaje entrante.

Tabla 1: Entrada, parámetros y salida esperada de patrón Splitter

2.6.2 Aggregator

El patrón *Aggregator* permite recolectar y guardar mensajes individuales hasta el momento en el que se reciba un conjunto completo de mensajes relacionados, luego publica un mensaje único resultado de la composición de los mensajes individuales. Este patrón funciona como un filtro, que recibe un flujo de mensajes e identifica qué mensajes están correlacionados. Cuando se recibió un conjunto completo informa de esto publicando un mensaje único, en la salida del canal [38]. En la figura 10 se observa el comportamiento del patrón y en la tabla 2 se detalla la entrada, los parámetros que recibe y la salida esperada.

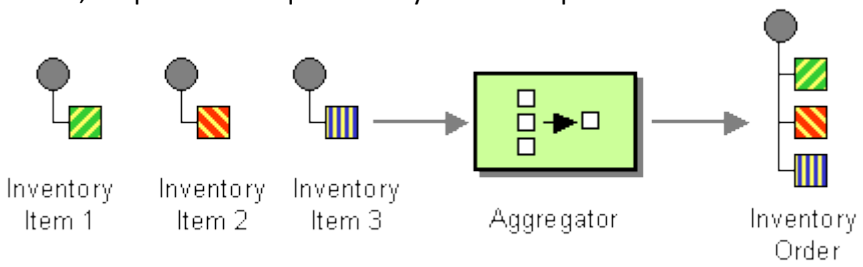

Figura 10: Patrón Aggregator

Entrada	Mensaje a procesar
Parámetros:	
aggregateStrategy	Estrategia para la realización de la agregación de los mensajes. Permite especificar de qué forma se agrupan los mensajes para formar el mensaje compuesto.
correlationId	Identificador que permite correlacionar distintos mensajes para realizar la agregación.
releaseStrategy	Condición para la liberación del mensaje.
Salida	Mensaje compuesto formado a partir de la aplicación de la estrategia de agregación *aggregateStrategy* a los mensajes con el identificador de correlación *correlationId*.

Tabla 2: Entrada, parámetros y salida esperada de patrón Aggregator

2.6.3 Recipient List

El patrón *Recipient List* permite enviar un mensaje a una lista de receptores, brindando la ventaja de que esa lista puede ser generada dinámicamente. Un caso particular de la utilización de este patrón sucede cuando la lista de receptores posee un único elemento, en el cual solamente se explota la posibilidad de establecer dinámicamente quién es el receptor.
Para la implementación del patrón es necesario definir un canal de mensajes para cada uno de los receptores en la lista [38]. La figura 11 muestra el comportamiento del patrón y en la tabla 3 se detalla la entrada, los parámetros que recibe y la salida esperada.

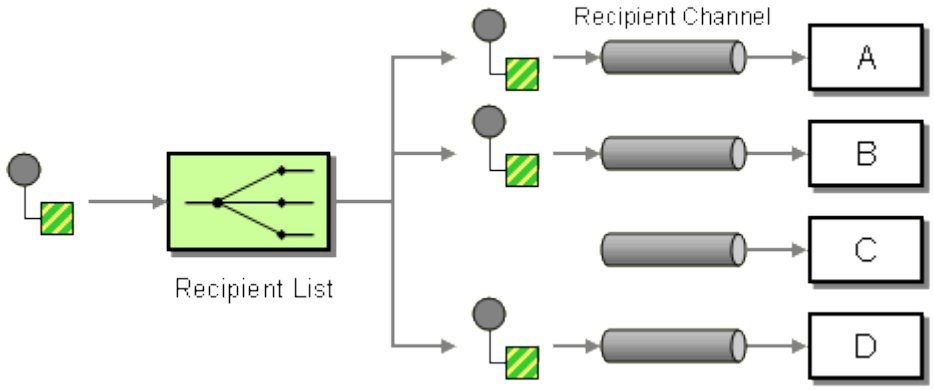

Figura 11: Patrón Recipient List

Entrada	Mensaje a procesar
Parámetros:	
recipients	Lista de receptores a quienes debe enviar el mensaje. La lista puede ser especificada estática o dinámicamente.
Salida	Reenvía el mensaje de entrada a todos los receptores especificados en *recipients*.

Tabla 3: Entrada, parámetros y salida esperada de patrón Recipient List

2.6.4 Message Translator

El patrón *Message Translator* es un filtro que permite traducir un mensaje con un formato "origen" a un formato "destino". Es utilizado principalmente para que aplicaciones que manejan distintos tipos de entidades o interfaces puedan comunicarse entre sí con facilidad. Es importante destacar que este patrón no aplica a un tipo de dato específico, sino que por lo contrario busca ser suficientemente genérico como para aceptar cualquier formato o tipo de datos tanto en el origen como en el destino [38]. La figura 12 muestra el comportamiento del patrón y en la tabla 4 se detalla la entrada, los parámetros que recibe y la salida esperada.

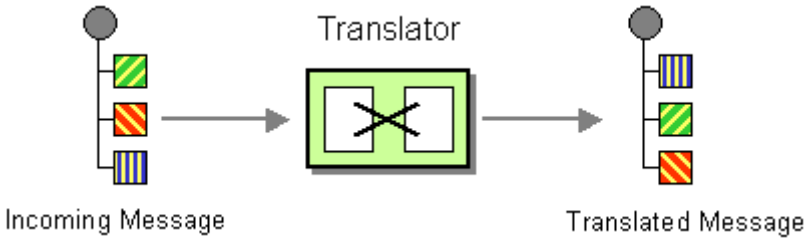

Figura 12: Patrón Message Translator

Entrada	Mensaje a procesar
Parámetros:	
transform	Transformación a ser aplicada sobre el mensaje de entrada.
Salida	Retorna el mensaje resultante de aplicar la transformación *transform* al mensaje recibido.

Tabla 4: Entrada, parámetros y salida esperada de patrón Message Translator

2.6.5 Content Based Router

El patrón *Content Based Router* permite examinar un mensaje y determinar a partir de su contenido el receptor al cuál debe ser redirigido, de una lista de receptores posibles definida estáticamente. El destinatario de un mensaje es único. Los criterios posibles a utilizar para la determinación del receptor pueden ser variados, desde la existencia de un determinado campo en el mensaje hasta valores específicos en campos determinados [38]. La figura 13 muestra el comportamiento del patrón y en la tabla 5 se detalla la entrada, los parámetros que recibe y la salida esperada.

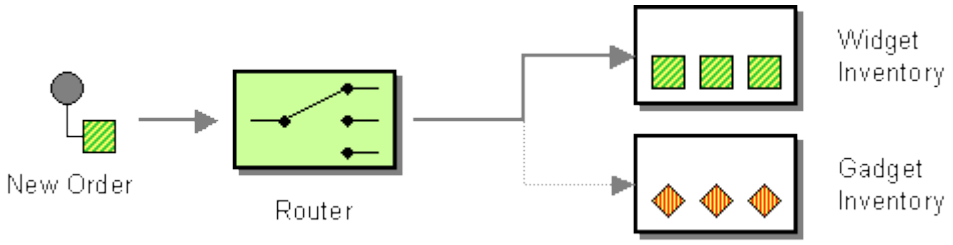

Figura 13: Patrón Content Based Router

Entrada	Mensaje a enviar
Parámetros:	
routingConditions	Lista de condiciones a evaluar para determinar a dónde debe reenviar el mensaje
routingDestinations	Lista de canales correspondiente a los posibles destinos a reenviar el mensaje de acuerdo a las condiciones *routingConditions*
Salida	Reenvía el mensaje al destinatario en *routingDestinations* correspondiente a la condición de *routingConditions* que al evaluar resultó válida.

Tabla 5: Entrada, parámetros y salida esperada de patrón Content Based Router

2.6.6 Composed Message Processor

El patrón Composed Message Processor permite combinar las funcionalidades brindadas por los patrones Splitter, Content Based Router y Aggregator para poder dividir un mensaje compuesto, reenviar cada uno de los sub-mensajes a un determinado *endpoint* y luego combinar las respuestas recibidas en un mensaje compuesto [38]. La figura 14 muestra el comportamiento del patrón y en la tabla 6 se detalla la entrada, los parámetros que recibe y la salida esperada.

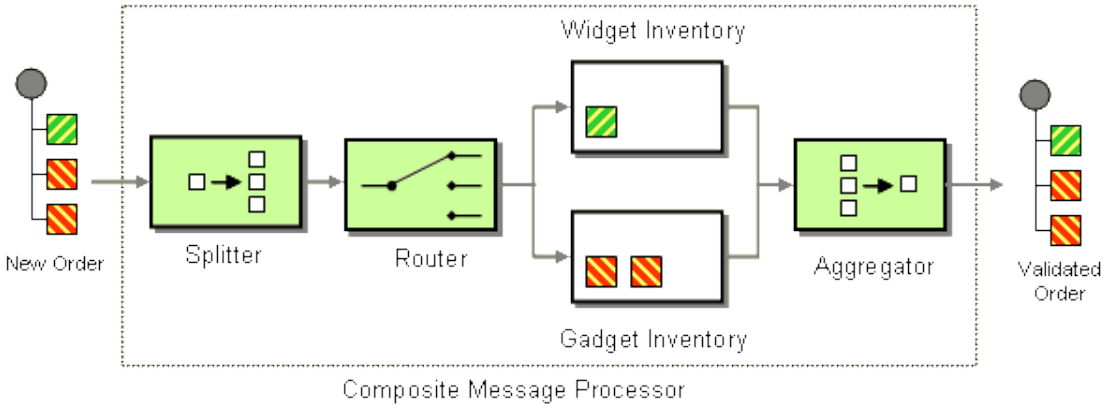

Figura 14: Patrón Composed Message Processor

Entrada	Mensaje compuesto a procesar
Parámetros:	
splitStrategy	Estrategia para la realización del *split*. Permite especificar de qué forma se parte el mensaje en sub-mensajes.
routingConditions	Lista de condiciones a evaluar para determinar a dónde debe reenviar cada uno de los sub-mensajes generados por el *splitter*
routingDestinations	Lista de canales correspondiente a los posibles destinos a reenviar el mensaje de acuerdo a las condiciones *routingConditions*
aggregateStrategy	Estrategia para la realización de la agregación de los mensajes resultantes de la respuesta de cada sub-mensaje enviado por el *router*. Permite especificar de qué forma se agrupan los mensajes para formar el mensaje compuesto.
correlationId	Identificador que permite correlacionar distintos mensajes para realizar la agregación.
releaseStrategy	Condición para la liberación del mensaje.
Salida	Retorna el mensaje resultante de procesar todas las respuestas de los sub-mensajes reenviados por el componente *router* aplicando la estrategia de agregación *aggregateStrategy*

Tabla 6: Entrada, parámetros y salida esperada de patrón Composed Message Processor

2.6.7 Selective Consumer

El patrón Selective Consumer permite que una aplicación que consume mensajes a través de un *Message Channel* pueda seleccionar los mensajes que desea consumir, especificando un criterio determinado a cumplir por los mensajes [38]. La figura 15 muestra el comportamiento del patrón y en la tabla 7 se detalla la entrada, los parámetros que recibe y la salida esperada.

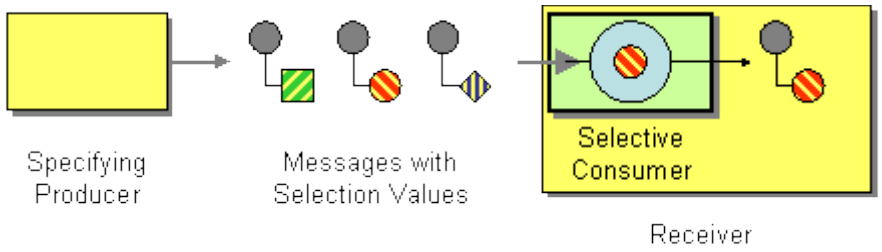

Figura 15: Patrón Selective Consumer

Entrada	Mensajes enviados al MessageChannel del cual se desean consumir mensajes.

Parámetros:	
FilterCriteria	Criterio de filtro para seleccionar únicamente los mensajes que el consumidor desea.
Salida	Mensajes que cumplen el criterio establecido por FilterCriteria.

Tabla 7: Entrada, parámetros y salida esperada de patrón Selective Consumer

2.6.8 Wire Tap

El patrón Wire Tap permite consumir los mensajes enviados a través de un canal sin interferir en el comportamiento de la aplicación que está escuchando los mensajes enviados, utilizando para ello un canal secundario. Para lograr dicho comportamiento el patrón Wire Tap se implementa como un Recipient List, con dos receptores fijos: el canal original como primer receptor y un canal secundario como el segundo. La figura 16 muestra el comportamiento del patrón y en la tabla 8 se detalla la entrada, los parámetros que recibe y la salida esperada.

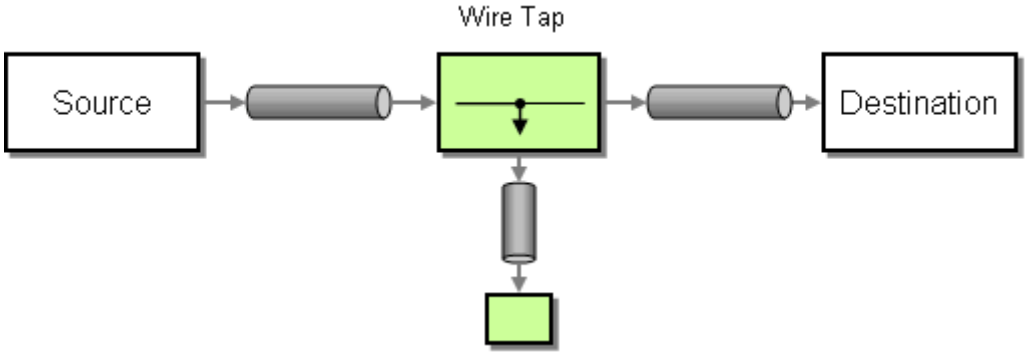

Figura 16: Patrón Wire Tap

Entrada	Mensaje a enviar entre las aplicaciones.
Parámetros:	
secondaryChannel	Canal secundario a dónde se deben enviar los mensajes.
Salida	El mismo mensaje de entrada enviado al destino original del mensaje y al canal secundario.

Tabla 8: Entrada, parámetros y salida esperada de patrón Wire Tap

2.7 Herramientas de desarrollo

Esta sección presenta las principales herramientas de desarrollo SOA evaluadas para ser utilizadas a lo largo del proyecto. Se presentan Switchyard y Camel, las cuales efectivamente fueron utilizadas.

2.7.1 Switchyard

Switchyard es un ESB desarrollado por JBoss, cuyo foco es la construcción de servicios y aplicaciones utilizando los conceptos y mejores prácticas de SOA. Implementa una arquitectura SCA y Utiliza Apache Camel [39] para proveer conectividad y transportes en ejecución de manera simple, flexible y rápida. Posee un modelo de aplicación que permite integrar Apache Camel, Java EE, BPM [40], Drools [41], orquestación y ruteo de servicios para lograr un modelo de desarrollo y ejecución completamente integrado.
En particular, Switchyard ofrece las distintas funcionalidades que ofrecen los ESB tradicionales, complementándolos con un fuerte foco en el desarrollo de servicios y la capacidad de realizar un desarrollo integrado, ejecución y gestión de los distintos componentes del sistema [42].

2.7.2 Camel

Apache Camel ™ es un framework de integración de código abierto, el cual está basado en los Patrones de Integración Empresarial (EIP) existentes. Permite definir ruteos y reglas de mediación utilizando distintos lenguajes (API Java, Spring, Scala DSL, entre otros); y utiliza URIs para trabajar directamente con cualquier tipo de transporte o modelo de mensajería (por ejemplo: HTTP, ActiveMQ, JMS, SCA) [43].

3 Análisis de requerimientos

En la sección anterior vimos los conceptos y tecnologías inherentes al proyecto. En esta sección se describe el análisis realizado de la realidad planteada. En primer lugar se describe el marco contextual del proyecto, la problemática identificada y las soluciones a los problemas. Luego se describen los requerimientos identificados del análisis anterior y el alcance definido para el proyecto, y por último un conjunto de recomendaciones a tener en cuenta y cómo se aplicaron algunos de los conceptos descritos anteriormente en el marco conceptual (sección 2).

3.1 Descripción del contexto

A continuación se realiza un análisis extensivo de la realidad estudiada profundizando en sus aspectos más relevantes.

Este proyecto de grado fue propuesto por un cliente externo y se enmarca en el contexto redes de pagos y cobranzas que interactúan de forma intensiva con servicios de terceros. Una red de pagos y cobranzas funciona de la siguiente manera: por un lado, existen múltiples locales de pago distribuidos geográficamente en distintos puntos del país, a los cuales concurren usuarios para la realización de un conjunto determinado de trámites (pagos de facturas, cobro de dinero, giros y transacciones locales e internacionales, trámites municipales, entre otros). Los locales de pago funcionan como terminales y no son capaces de efectivizar por si solos los distintos trámites que desean realizar los clientes. Para ello se comunican con un servidor central el cuál es el encargado de procesar todos los pedidos provenientes de los locales y efectivizarlos invocando a los correspondientes servicios de los sistemas de terceros involucrados. El sistema debe considerar que en un pedido de un cliente, las invocaciones a un mismo servicio de tercero (ejemplo: pago de factura de UTE) deben ser realizadas de forma serial.

La figura 17 ilustra un ejemplo de comportamiento del sistema, en el que un cliente desea pagar una factura de UTE y una factura de Antel en un local de pagos. Este ejemplo contempla el flujo de ejecución normal del sistema. La serie de pasos para poder efectivizar dichos pagos es la siguiente:

1. El local de pagos genera un pedido al servidor central indicando los datos de las dos facturas a pagar.
2. El servidor central recibe el pedido y ejecuta invocaciones a los dos servicios de terceros involucrados: por un lado al servicio brindado por Antel, para efectivizar el pago de la factura de Antel y por el otro al brindado por UTE.
3. Luego de recibir la confirmación de ambos pedidos, el servidor central responde al local indicando que la realización de los pagos fue efectivizada.
4. Por último, el cliente recibe la notificación (por ejemplo, se imprime la factura correspondiente) de que su pago fue efectivizado.

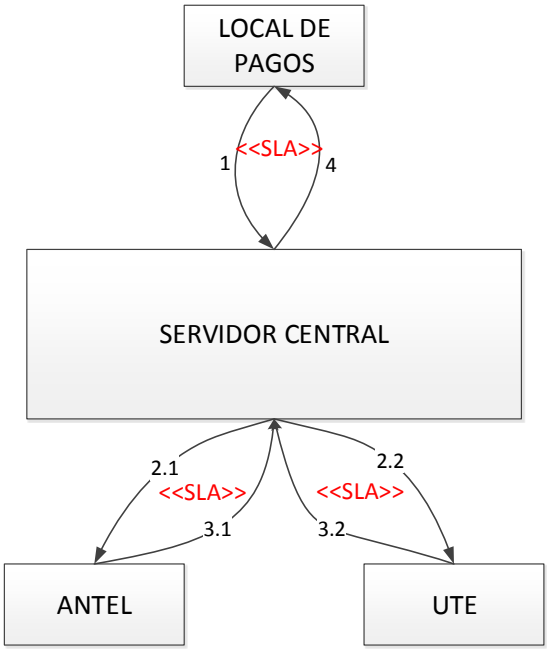

Figura 17: Ejemplo de interacción con el servidor central

En caso de error en alguna de las invocaciones a los servicios de terceros (falla o *timeout*) el cliente recibe el detalle del estado de la ejecución de cada uno de los servicios.
Por otro lado, el sistema central debe responder en un determinado tiempo de respuesta a los locales de pago.

3.2 Identificación de la problemática

La necesidad actual genera un espacio de interacción multi organizacional complejo en donde cada organización provee servicios con diferentes características tanto dentro de los umbrales de la propia organización como con las organizaciones que consumen estos servicios. Este ecosistema de servicios es de índole muy diferente teniendo cada servicio características disímiles.
La heterogeneidad de los servicios ha llevado a que este tipo de interacción sea cada vez más difícil de mantener, causando que los procesos de incorporación de nuevas funcionalidades, o modificación de las ya existentes, sean costosos y generen los siguientes problemas técnicos con un impacto negativo en el negocio:

- Tiempo de respuesta. Los servicios pueden tener momentos del día en el cual son muy consultados haciéndolos indisponibles o con demoras.
- Disponibilidad. Problemas de conectividad con los organismos proveedores de los servicios impiden su invocación.
- SLA. El sistema debe poder ejecutar los pedidos en un tiempo determinado definido entre el local y el sistema central.
- Consistencia de datos:
    - Los problemas de conectividad pueden dejar el sistema inconsistente.
    - Puede ser posible cancelar una invocación a un servicio dado que el tiempo de ejecución llevó más de lo permitido.

3.3 Solución a los problemas

Para los problemas identificados en el punto anterior, se determinaron algunas posibles soluciones:

- Tiempo de respuesta. El sistema debería adecuarse a los cambios en los tiempos de respuesta de los servicios durante el día.
- Conectividad. Ante un fallo en la invocación a un servicio debido a un problema de conectividad, el sistema debería ser capaz poder dejar el estado de los servicios en un estado consistente.
- SLA. Es necesario contemplar los SLA de los servicios involucrados y del propio sistema central con los locales.
- Consistencia de datos. Ante la cancelación en la invocación a un servicio, el sistema debería ser capaz poder dejar el estado de los servicios en un estado consistente.

Dada la realidad planteada y los problemas identificados para este tipo de sistemas, se plantea una solución basada por un lado en la clasificación de los servicios involucrados y por otro lado la definición de políticas de ejecución para la invocación a los servicios.

3.3.1 Clasificación de servicios

En el marco de nuestro proyecto, se planteó el problema de interoperar con servicios con diferentes características. Una de las necesidades del sistema es el de poder decidir el grado de concurrencia en la invocación a los servicios y cómo reaccionar ante fallas. Para poder determinar cuál va a ser la forma en la que se van a invocar los servicios se define una clasificación basada en las necesidades del cliente. En una clasificación más general estarían involucradas otras características que no aplican a esta solución.
Los tipos de servicios identificados son los siguientes:

- Cancelable. Son los servicios que ofrecen un mecanismo para cancelar un pedido que está en curso. La operación de cancelación podría ser invocada en el caso de que la invocación al servicio de un *timeout.*
- Anulable. Son los servicios que ofrecen una operación que permite deshacer los cambios realizados por una invocación al servicio que haya sido finalizada (compensación).
- Simples. Son aquellos servicios que no cuentan con mecanismos para cancelar su ejecución o deshacer una ejecución que haya sido confirmada como exitosa.

Para nuestra realidad los servicios solo podrán ser clasificados de una de las tres formas posibles (no habrán servicios que sean a la vez cancelables y anulables).

3.3.2 Políticas de ejecución

La política de ejecución determina cómo el sistema invocará a los diferentes servicios y cómo reaccionará ante posibles fallos, errores o *timeouts*. A continuación se describen las diferentes políticas definidas.

3.3.2.1 Best Effort (BE).

Se ejecutan todos los servicios del pedido que puedan ser tenidos en cuenta en el tiempo acordado en la invocación al sistema. En caso de que el tiempo estimado sea mayor al del pedido, ser recortarán los servicios que se estima demoran más tiempo hasta poder cumplir con el tiempo acordado. Solo se detiene la ejecución cuando se termina la invocación a todos los servicios o cuando se termina el tiempo de ejecución. Se le comunica al cliente cuáles invocaciones resultaron exitosas, cuáles fallaron y cuáles no se ejecutaron.
A modo de ejemplo, supongamos que un cliente desea realizar el pago de varias facturas en un local de la red de pagos y cobranzas. El hecho de que el pago de alguna de las facturas no pueda efectivizarse no debe ser un impedimento para realizar el pago de las demás, por ende

la mejor política a utilizar en esta situación es BE. En la figura 18 se puede observar el ejemplo mencionado.

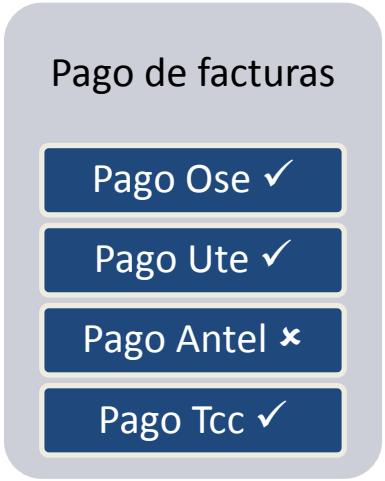

Figura 18: ejemplo Best Effort

3.3.2.2 Todo o Nada (ToN).

Se deben ejecutar todos los servicios del pedido. Si el tiempo estimado para la invocación a todos los servicios es mayor que el tiempo solicitado entonces no se ejecuta ninguna invocación y se responde inmediatamente. Si alguno de ellos falla, se cancelan/anulan todas las invocaciones ya realizadas. Para esta política se restringe el tipo de servicio a ejecutar. Solo pueden ejecutarse servicios anulables y a lo sumo un servicio simple o cancelable en un mismo pedido. Si durante la ejecución del pedido se termina el tiempo solicitado, entonces se procede a anular/cancelar todos los servicios ya ejecutados o en curso según corresponda.
A modo de ejemplo supongamos que un cliente desea realizar dos depósitos en efectivo, a cuentas pertenecientes a diferentes bancos (digamos BBVA e Itaú), pero sólo quiere hacer efectivo los depósitos si se pueden realizar ambos. En caso de que alguno de los depósitos falle, no desea que el otro depósito se efectivice. En la figura 19 se puede observar el ejemplo mencionado.

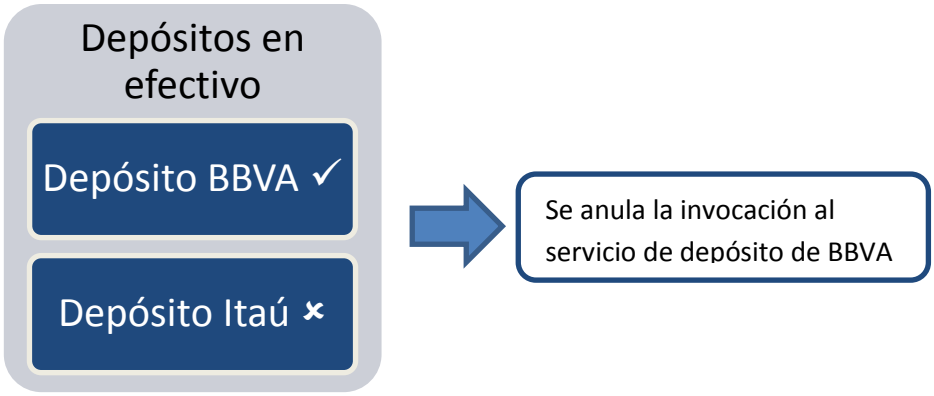

Figura 19: ejemplo Todo o Nada

3.3.2.3 Dependency on Priority (DoP)

Esta política se diferencia con Best Effort en que en el pedido los servicios pueden tener definidas prioridades. Las prioridades son utilizadas para determinar el orden de ejecución de los servicios y al determinar que el pedido no se puede cumplir en el tiempo deseado se

utilizan las prioridades para la poda de servicios. En definitiva Best Effort se puede ver como una DoP en donde todos los servicios tienen la misma prioridad.
A modo de ejemplo, supongamos que un cliente desea realizar un retiro en efectivo de una cuenta y realizar el pago de dos facturas (Ose y Antel) condicionando la realización de los pagos a que pueda realizar el retiro del efectivo. En el caso de que no pueda realizar el retiro en efectivo, los pagos no deben realizarse. En caso contrario, no importa si luego los pagos fallan o no. En la figura 20 se visualiza el ejemplo con los dos casos mencionados.

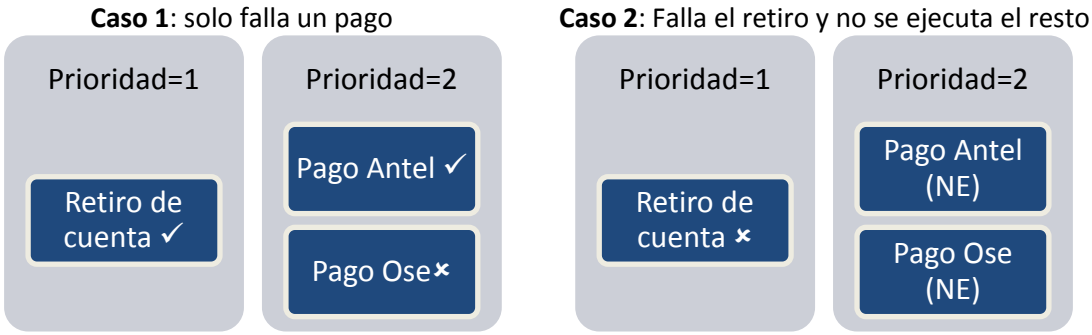

Figura 20: ejemplo Dependency on Priority

3.4 Requerimientos

A continuación se presentan los distintos requerimientos que debe poseer la solución, clasificados según sean funcionales o no funcionales. Su elaboración fue realizada en conjunto con el cliente de este proyecto y validada por éste.

3.4.1 Funcionales

Los requerimientos funcionales son clasificados en distintos subgrupos de acuerdo a la parte del sistema en la que están involucrados.

3.4.1.1 Servicios Finales

Los servicios finales son aquellos servicios brindados por terceros (proveedores) que son consumidos por el sistema (a modo de ejemplo: Antel, OSE, UTE). En la tabla 9 se presenta la lista de requerimientos asociados a los servicios finales.

ID	Requerimiento	Descripción
RQ-SF1	Clasificación de servicios	Los servicios que se ingresen al sistema deben de poder clasificarse según lo descrito en la sección 3.3.1
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	Para cada servicio final, se considerarán las siguientes métricas: • Tiempo de respuesta (response time). • Rendimiento máximo (máximum

		throughput). • Disponibilidad (availability). • Accesibilidad (accessibility). • Nivel de éxito (successability).
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.
RQ-SF7	Otras métricas	Para las demás métricas sólo se tendrá en cuenta la medida en ejecución.

Tabla 9: Requerimientos de servicios finales

3.4.1.2 Servicio de cara al cliente

El servicio de cara al cliente es el servicio brindado por el sistema central hacia los locales que componen la red de pagos y cobranzas. En la tabla 10 se presenta la lista de requerimientos asociados al servicio 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. En otras palabras, el sistema debe de realizar de forma serial las invocaciones a un mismo servicio final por parte de un mismo local.
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. Esta métrica puede ser definida en el sistema o puede ser informada por el cliente en la invocación.
RQ-SC4	Políticas de ejecución	El sistema debe soportar las siguientes políticas de ejecución: • Todo o nada. En caso de falla de algún servicio, detendrán todas las invocaciones pendientes y en curso y revertir en caso que se pueda. • Best effort. Esta política permite invocar todos los servicios programados para la ejecución a pesar que alguno de ellos falle.

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.

Tabla 10: Requerimientos del servicio de cara al cliente

3.4.1.3 Sistema central

En la tabla 11 se presentan los requerimientos funcionales relativos al núcleo del sistema central.

ID	Requerimiento	Descripción
RQ-S1	Reintentos en servicios simples	Dado un servicio de tipo *simple* se podrá definir la política de reintentos ante una falla en su invocación.
RQ-S2	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 deberá resolver el orden en el cual invocar a los servicios y si es posible ejecutarlos en paralelo. Ante una política de "Todo o nada", los servicios simples serán los últimos en ser invocados.
RQ-S3	Cumplimiento del SLA	Para la evaluación del complimiento del SLA con el local, el sistema se basará en la métrica *response time*.
RQ-S4	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. Se deberá tener en cuenta los valores históricos de las métricas en diferentes momentos como ser la hora del día, día de la semana y día en el mes para realizar la validación.
RQ-S5	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	Thread pool por servicio	Se deberá contar con un *thread pool* para la invocación a cada servicio y de esa forma tener control de la cantidad de invocaciones concurrentes al mismo.
RQ-S7	API para ABM de servicios y SLA	El sistema brindará una API que permita dar de alta un nuevo servicio y el SLA correspondiente.

Tabla 11: Requerimientos del núcleo del sistema

3.4.1.4 Interfaces

Engloba los requerimientos funcionales relativos a las distintas interfaces gráficas existentes en el sistema. En la tabla 12 se presenta una lista detallada de los mismos.

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: • A través de un *dashboard* se podrán obtener los distintos valores de las métricas involucradas en el SLA de los servicios. • La información de las métricas se podrá visualizar gráficamente y/o en forma de grilla.
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. Los atributos configurables a los servicios son: • Cantidad máxima de invocaciones concurrentes. • Especificar la transformación a aplicar al Request/Response del servicio. • Clasificación del servicio (simple, anulable o cancelable). • En caso de que corresponda, permitir especificar la forma en la que se debe cancelar o anular una invocación del servicio. • Máximo número de reintentos. • Valores teóricos para las distintas métricas existentes en el SLA.

Tabla 12: Requerimientos de interfaces gráficas

3.4.2 No Funcionales

En la tabla 13 se detallan los requerimientos no funcionales del sistema.

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.

Tabla 13: Requerimientos no funcionales

3.5 Alcance

En esta sección se resume el alcance final de la solución luego de la negociación con el cliente llevada a cabo en las diferentes etapas del proyecto. Se realiza una descripción en alto nivel del sistema y de las principales características con las cuales contará este. Además se revisan los requerimientos inicialmente establecidos, indicándose aquellos que no están considerados en el alcance final del producto y aquellos que sufrieron modificaciones respecto de su definición original.

El sistema provee una interfaz única a las aplicaciones de la organización, la cual es expuesta como *Web Service* SOAP. La respuesta contará con la información del resultado de cada pedido realizado.

Cuando una aplicación invoca al sistema, éste introspecciona los datos con el fin de saber cuáles son los servicios que debe invocar. Sabiendo cuales son los servicios a invocar se calcula el tiempo teórico que le llevaría al sistema realizar las invocaciones a los servicios y se valida que el tiempo esté dentro de los parámetros estipulados por el SLA especificado por el cliente en la invocación (o en su defecto, el del sistema). Los tiempos de respuesta de cada servicio se calculan en *runtime* además de tener un valor teórico definido en cada servicio. A partir de la política establecida se define un plan de ejecución para el pedido, se realiza una estimación del tiempo total de ejecución de dicho plan y se valida la viabilidad de la ejecución del mismo.

En caso de fallas o demoras inesperadas en los servicios, se aplican compensaciones o reversa a los servicios que lo dispongan, dependiendo de la política seleccionada.

En la tabla 14 se detallan los requerimientos que fueron eliminados, los modificados y los nuevos requerimientos agregados. Los requerimientos que permanecieron incambiados no son incluidos.

ID	Estado (M=Modificado, E=Eliminado, N=Nuevo)	Modificaciones (si corresponde)
RQ-SF4	M	Se implementan las métricas tiempo de respuesta, accesibilidad y nivel de éxito.
RQ-SC4	M	Se agrega una nueva política de ejecución, llamada *Dependency on Priority*. Dicha política está descrita en el punto 4.1.3.3.
RQ-SC7	N	Los parámetros de la invocación a cada servicio son especificados en formato clave-valor por el cliente y los correspondientes a las respuestas de igual forma por el sistema.
RQ-S1	E	
RQ-S4	M	Para el cálculo de las métricas se considera únicamente la hora del día y el día de la semana. El cálculo de las métricas se realiza cada un cierto tiempo configurable, en vez de luego de cada invocación.
RQ-S6	M	No se restringe la cantidad de invocaciones a un mismo servicio (gestionando un *thread pool*) pero se establecen los pasos a seguir para implementarlo/configurarlo.
RQ-S8	N	El algoritmo de scheduler debe retroalimentarse con los datos de las métricas calculadas para los servicios que debe ejecutar
RQ-I2	M	Se sustituye el *dashboard* por una interfaz simple que permite visualizar los valores de las métricas de los servicios, el estado del sistema y una gráfica comparativa del tiempo calculado para las ejecuciones y el tiempo real de las mismas.
RQ-I4	M	A la lista de atributos configurables para un servicio se agregan:

		• URLs de servicios de cancelación / compensación. • Tiempo de respuesta máximo para los pedidos y sus correspondientes cancelaciones / compensaciones.

Tabla 14: Alcance final

3.6 Aplicación de conceptos

En las secciones 2.2, 2.3 y 2.4 de este documento se presentan distintos conceptos estudiados durante la etapa de investigación y análisis de este proyecto. Dichos conceptos a medida que avanzó el tiempo fueron aplicados o descartados dependiendo de distintos factores. A continuación se analizan a fondo los mismos y la aplicabilidad que tuvieron en el sistema.

3.6.1.1 Composición de servicios

En la sección 2.2 se presentan los distintos conceptos relativos a las composiciones de servicios, ya sean estáticas o dinámicas. En particular nuestro sistema debe interactuar con servicios provistos por terceros, sin embargo, para resolver cada uno de los componentes de un pedido realizado por un local de pago es necesaria la invocación a un único servicio de terceros. Cada pedido realizado por un local de pagos es en sí una composición de servicios, pero ésta no es definida por el sistema central, sino que viene implícita en el pedido. Esto implica que éste no realiza composiciones de servicios, sino que simplemente las ejecuta utilizando el modelo de mediación.
Por lo tanto el soporte a composiciones de servicios, ya sean estáticas o dinámicas, para la resolución de pedidos de los locales de pago está fuera del alcance de este sistema. En la sección 7.2.8 se realiza el planteo de dar soporte a composiciones de servicios como un trabajo a futuro.

3.6.1.2 Consistencia de datos

Los conceptos presentados son el resultado de una investigación profunda sobre los distintos conceptos relacionados al manejo de transacciones en *Web Services*. Para cada uno de ellos se detalla por qué fueron o no tenidos en cuenta en las etapas posteriores del proyecto:

- **Transacciones de corta duración (atómicas):** Una transacción de corta duración necesita bloquear los distintos recursos involucrados mientras ésta dura, lo que implica que pueda afectar el rendimiento del sistema en aquellos casos que exista mucha concurrencia y los tiempos de demora de las transacciones no sean cortos. En nuestro sistema la atención de pedidos concurrentes es parte inherente de los requerimientos y a priori no se conoce el tiempo que puede demorar la respuesta de un servicio, por lo que contar con un soporte transaccional de este estilo no fue tenido en cuenta.
- **Transacciones de larga duración:** La utilización de transacciones de larga duración efectivamente encaja en el contexto manejado por nuestro sistema. Como se detalla en la sección 3.3.1, existen servicios brindados por sistemas de terceros que deben proveer mecanismos para realizar compensación de las operaciones que no resultaron exitosas. En la sección 2.3.2 se mencionó dos mecanismos para realizar las compensaciones: por diseño y programadas. En nuestro sistema las compensaciones se realizan de forma programada, ya que los servicios que requieren de la utilización de compensaciones exponen una operación para cumplir con esa funcionalidad. De todas formas en el punto Diseño de Web Services se realiza un conjunto de recomendaciones que detalla los beneficios de contar con compensaciones por diseño.

- **Estándares para transacciones en *Web Services*:** En la sección 2.5.2 se brinda una introducción al estándar de manejo de transacciones *WS-Coordination* y a las dos implementaciones existentes hoy en día: *WS-AtomicTransaction* (para manejo de transacciones atómicas) y *WS-BusinessActivity* (para manejo de transacciones de larga duración). La utilización de estos estándares en el sistema para comunicarse con servicios de terceros quedó descartada debido a que el potencial que tienen puede ser aprovechado al máximo en ambientes altamente dinámicos y distribuidos; mientras que nuestro sistema no requiere coordinar actividades de negocio de alta complejidad, simplemente invocar servicios de terceros a medida que sea posible.
- **Diseño de Web Services.** Si bien la mayoría de los servicios con los cuales interactúa una red de pagos y cobranzas son servicios que llevan desarrollados varios años, vemos necesario realizar un conjunto básico de recomendaciones a la hora de diseñar e implementar servicios web. De esta manera, los nuevos servicios que sean incorporados (o aquellos en los que se realice una reingeniería completa) pueden contar con una guía que permita mejorar la consistencia de datos y optimizar la forma en la que interactúan con otros sistemas. Este conjunto de recomendaciones fue elaborado a partir del marco teórico presentado en la sección 2 de este documento:
    - Considerar servicios que sean ACID2. Como se mostró en el marco teórico este tipo de servicios permiten que los sistemas puedan reaccionar mejor ante situaciones de fallos. A modo de ejemplo, servicios idempotentes implican que quién consuma el servicio pueda realizar múltiples invocaciones en caso de la ocurrencia de un *timeout* sin perjuicio de generar un resultado no deseado sobre él sistema.
    - En caso de no implementar servicios que cumplan con ACID2, es importante proveer al menos un mecanismo alternativo que permita deshacer los cambios realizados (compensación).

3.6.1.3 Calidad

Es importante destacar que el modelo que plantea el proyecto S-Cube es muy útil en contextos altamente dinámicos y complejos donde existen potencialmente múltiples implementaciones de servicios que son capaces de resolver un mismo pedido. En este tipo de contexto se le puede sacar provecho a los más de 80 atributos pertenecientes al modelo y a la expresividad del mismo en cuanto a la capacidad de definir acuerdos de servicio de manera dinámica, estableciendo métodos para que los actores involucrados negocien los SLA dinámicamente.
En contraste con dicho contexto, nuestro proyecto está orientado a un caso mucho más acotado, en el cual potencialmente solo pueda existir una única implementación disponible para resolver un servicio. A modo de ejemplo: si un cliente concurre a un local de la red de pagos y cobranzas pretendiendo pagar las facturas de UTE, OSE y ANTEL; con seguridad podemos afirmar que sólo va a existir un único servicio que permita el pago de facturas de UTE, al igual que para OSE y ANTEL siendo cada uno de los mismos brindado por la correspondiente entidad. La potencial capacidad de expresión que posee este modelo no justifica su complejidad de uso en este caso en particular.
Considerando el modelo WS-QM, podemos decir que nuestro modelo de calidad debe estar enfocado a un grupo muy reducido de actores (básicamente serían los usuarios del sistema, el sistema central y los sistemas de terceros) y como actividad principal tenemos la de evaluar si se puede llevar o no a cabo el pedido realizado por el cliente; lo que no justifica una clasificación en Actores y Actividades.
Dadas las características del sistema a construir, decidimos elaborar un modelo de calidad que se adapte a la realidad del negocio mucho más simple que los modelos planteados. Nuestro modelo considera aquellos sub-factores de calidad definidos en WS-QF que nos permiten

obtener una medida del nivel de servicio que se está brindando y además el sub-factor asociado a la integridad de las transacciones involucradas (*TransactionIntegrity*), dejando de lado la definición de actores y actividades en el sistema que presenta WS-QM.

4 Diseño de la solución

En las secciones siguientes se presentan los distintos aspectos comprendidos dentro del diseño del sistema. En la sección 4.1 se define el modelo de calidad utilizado y en la 4.2 se muestran los distintos aspectos de la arquitectura del sistema, clasificados según la vista a la que corresponden. Se presentan la vista lógica, de procesos y de componentes del sistema, lo que permite su comprensión desde múltiples ópticas.

4.1 Modelo de calidad definido

En esta sección se presentan los distintos aspectos relacionados al diseño del modelo de calidad definido para el sistema. Se realiza una descripción del modelo, donde se muestran los factores de calidad considerados (secciones 4.1.1 y 4.1.2) y las dependencias existentes entre ellos (sección 4.1.3).
El modelo de calidad definido debe contemplar dos SLA diferentes:

- El existente entre el Servidor Central y los distintos locales de pagos.
- El existente entre el Servidor Central y los distintos Servicios de Terceros a ser consumidos.

El modelo planteado está basado en un subconjunto de los sub-factores de calidad definidos en WS-QF y dos factores de calidad definidos en el proyecto. En la sección 4.1.1.1 se detallan los sub-factores de calidad tomados de WS-QF mientras que en la sección 4.1.1.2 se define a los factores *Transaction Integrity* y *Overhead* del sistema como los nuevos factores de calidad tomados en cuenta para el modelo. Por último, en la sección 4.1.1.3 se muestran las dependencias existentes entre los distintos factores de calidad considerados en el modelo.
En la figura 21 se observa un diagrama con los dos SLA existentes en el contexto de la red de locales de pagos y cobranzas. A pesar de contar con dos tipos de SLA existentes en el sistema, el modelo de calidad los contempla a ambos.

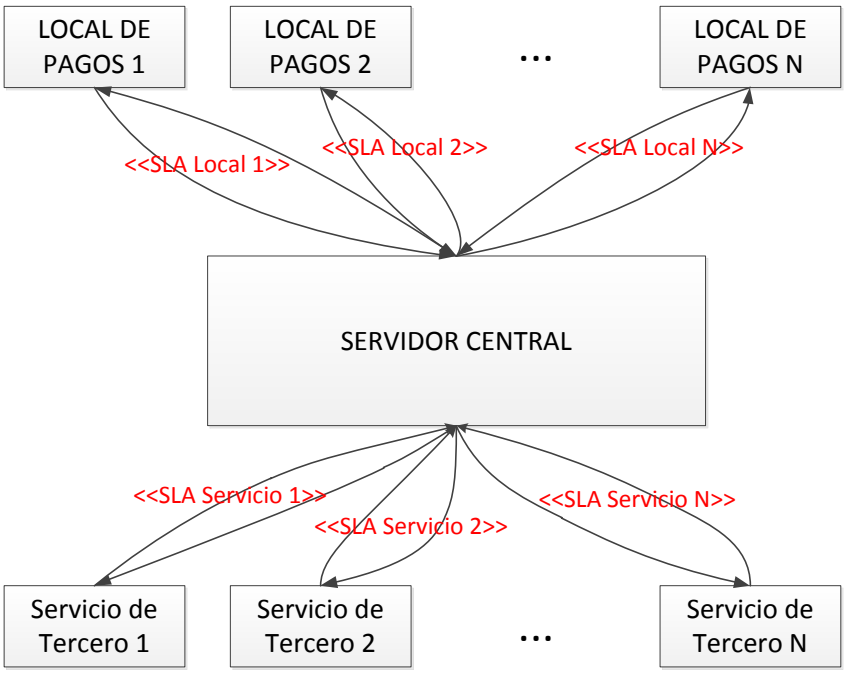

Figura 21: Ejemplo de interacción con el servidor central y los SLA involucrados

En la figura 22 se puede observar un resumen de los factores contemplados en el modelo de calidad definido.

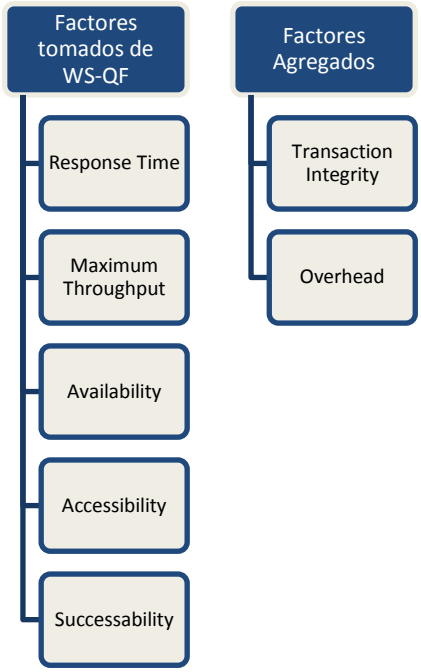

Figura 22: Modelo de calidad definido

4.1.1 Factores de calidad tomados de WS-QF

Dada la relevancia que poseen en el contexto de este proyecto, el modelo de calidad está definido en gran parte a partir de los sub-factores de calidad establecidos en WS-QF que están considerados dentro del factor de calidad Service Level Measurement (Medida del Nivel de Servicio).
Los sub-factores de calidad definidos en WS-QF que son considerados dentro de la Medida del Nivel de Servicio son los siguientes:

- *response time* (tiempo de respuesta)
- *máximum throughput* (rendimiento máximo)
- *availability* (disponibilidad)
- *accessibility* (accesibilidad)
- *successability* (nivel de éxito)

Estos factores nos permiten evaluar cuantitativamente la respuesta de un conjunto de servicios en ejecución ante los consumidores. En particular, podemos:

- Para el caso del sub-factor *response time*, generar alertas preventivas sobre el cumplimiento de los distintos SLA involucrados (aquellos generados con proveedores del sistema y el establecido con los locales de pago).
- Para los restantes sub-factores, generar alertas sobre posibles fallas que puedan estar ocurriendo en el sistema y detectar cuando un servicio brindado por un proveedor no está funcionando correctamente.

En el Anexo A se puede encontrar el detalle con la definición y el método de cálculo para la métrica de todos los factores mencionados.

4.1.2 Factores de calidad agregados

Dada la necesidad de tener un mayor control sobre el cálculo de los tiempos involucrados en una solicitud es que se ve la necesidad de definir un nuevo factor, llamado Overhead, que

ayude a tales efectos. También existe la necesidad de contemplar la capacidad de manejar mecanismos de compensación en el caso de la utilización de la política ToN, por lo que se contextualizó la utilización del factor de calidad *Transaction Integrity* definido en WS-QF.
A continuación se detalla la utilización de ambos factores.

4.1.2.1 Transaction Integrity

En el modelo WS-QualityFactor se define el factor *Transaction Integrity* para casos de transacciones de larga duración, el factor *Transaction Integrity* refiere a la capacidad que pueda tener el servicio para asegurar una consistencia de los sistemas participantes y de brindar mecanismos de compensación para aquellas transacciones que resultan fallidas.
En el contexto de este proyecto se considera este factor para el caso de la utilización de la política ToN, ya que es el único caso en el que el sistema debe contemplar la posibilidad de tener que realizar compensaciones ante la falla de la invocación de alguno de los servicios involucrados.

4.1.2.2 Overhead

Si un sistema necesita invocar un servicio y esta invocación la realiza a través de otro sistema, entonces el tiempo de ejecución agregado por este sistema intermedio se considera tiempo adicional. Se definió el factor de calidad Overhead como el tiempo consumido por el sistema intermedio, desde que es invocado, para preparar la invocación a un servicio y dar una respuesta.
Como vimos anteriormente, el Response Time es la suma de tres parámetros: ClientLatency, NetworkLatency y ServerLatency. El cliente al hacer un pedido al sistema tiene un Response Time con el sistema. El ServerLatency de la invocación al sistema estaría compuesto de la siguiente forma:

$$\text{ServerLatency} = \text{Overhead} + \sum_{i=1}^{N} RT_i \quad siendo \quad \begin{matrix} N = número\ de\ servicios \\ RT_i = response\ time\ del\ servicio\ i \end{matrix}$$

Este factor de calidad solo aplica a servicios compuestos o escenarios mediados como por ejemplo un middleware tipo ESB.
En la sección 5.3.2.5 se explica en detalle cómo es utilizado este factor para la estimación del tiempo de ejecución de un pedido al sistema.

4.1.3 Dependencia entre factores

En el análisis realizado al modelo de calidad con el fin de seleccionar cuales son los factores de calidad más relevantes para este proyecto, se analizó también si existe o no alguna relación entre los factores que conforman el modelo de calidad. De este análisis se desprende que existe una única dependencia entre los factores anteriormente mencionados. A medida que aumenta el tiempo de respuesta (*ResponseTime*) de un servicio, disminuye el rendimiento máximo (*MaximumThroughput*) del mismo.

4.2 Arquitectura

A continuación se describe la arquitectura del sistema mediante la utilización de las distintas vistas (lógica, componentes, despliegue y procesos) y se realiza una descripción de alto nivel de la estructura de la aplicación.

4.2.1 Vista lógica

En la figura 23 se muestra la arquitectura modular y en capas aplicada al diseño de la aplicación y cómo ésta interactúa con los distintos locales de pago y los servicios finales. Se puede observar que la interacción entre los locales de pago y el sistema desarrollado se realiza únicamente a través del módulo Web Service, único punto de acceso para los locales. Por el otro lado, también se puede observar que el único componente que interactúa con los distintos servicios finales es el Router.

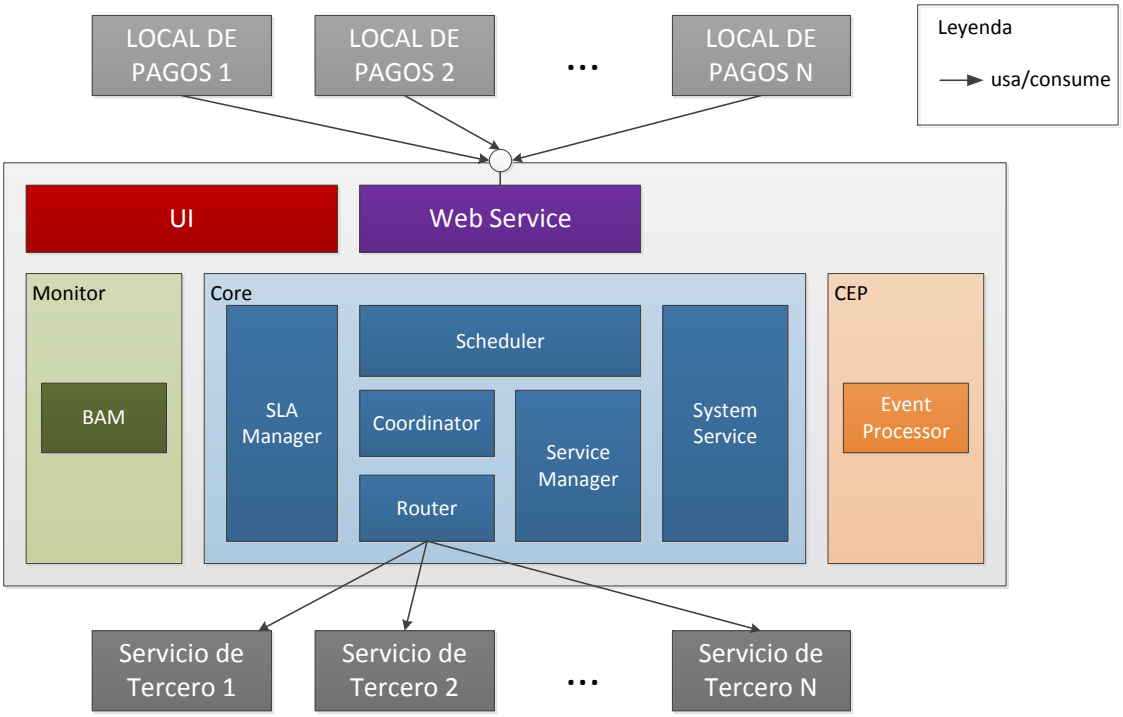

Figura 23: Vista lógica del sistema

Los distintos módulos presentados están definidos de acuerdo a la función que cumplen en el sistema:

- **UI**: Engloba las distintas interfaces de usuario existentes. Dentro de este módulo se incluye la interfaz que permite establecer los servicios de terceros con los que se interactúa y la visualización del estado general del sistema.
- **Web Service**: Módulo que encapsula la publicación del servicio único brindado a los distintos locales que utilizan el sistema. La interfaz expuesta sigue el protocolo WSDL.
- **Core**: Es el núcleo del sistema. Engloba los módulos principales encargados del procesamiento completo de los pedidos realizados por los locales de pagos. Los módulos pertenecientes al Core son:
    - SLA Manager: encargado de la gestión de los SLA de los servicios de terceros.
    - Scheduler: elabora la mejor estrategia (plan) para la ejecución del pedido del cliente. Obtiene los datos de los distintos servicios de terceros mediante la invocación al módulo ServiceManager y realiza la ejecución del pedido mediante la invocación al módulo Coordinator.
    - Router: módulo que realiza la interacción directa con los servicios de terceros.
    - Coordinator: módulo encargado de gestionar las distintas ejecuciones de pedidos a los servicios de terceros a través de la invocación al módulo Router. Además gestiona la ejecución de cancelaciones y compensaciones en caso que sea necesario.

- ServiceManager: gestiona los distintos datos correspondientes a los servicios de terceros (por ejemplo: tiempo de respuesta teórico, URL, tipo de servicio, etc.).
- SystemService: almacena parámetros globales al sistema, independientes de los pedidos que se encuentren en ejecución.

- **Monitor**: Engloba los módulos que permiten el monitoreo del estado general del sistema. Actualmente posee un único módulo, BAM (*Business Activity Monitoring*) que se encarga del monitoreo propiamente dicho.
- **CEP** (*Complex Event Processing*): Módulos que se encargan del procesamiento de los distintos eventos generados en el sistema. Al igual que Monitor, posee un único módulo, EventProcessor.

4.2.2 Vista de componentes

La figura 24 muestra el diagrama de componentes del sistema. A continuación se realiza una breve descripción de cada componente (agrupados por paquete). Es importante destacar que los componentes están agrupados en paquetes de acuerdo a la funcionalidad que cumplen en el sistema.

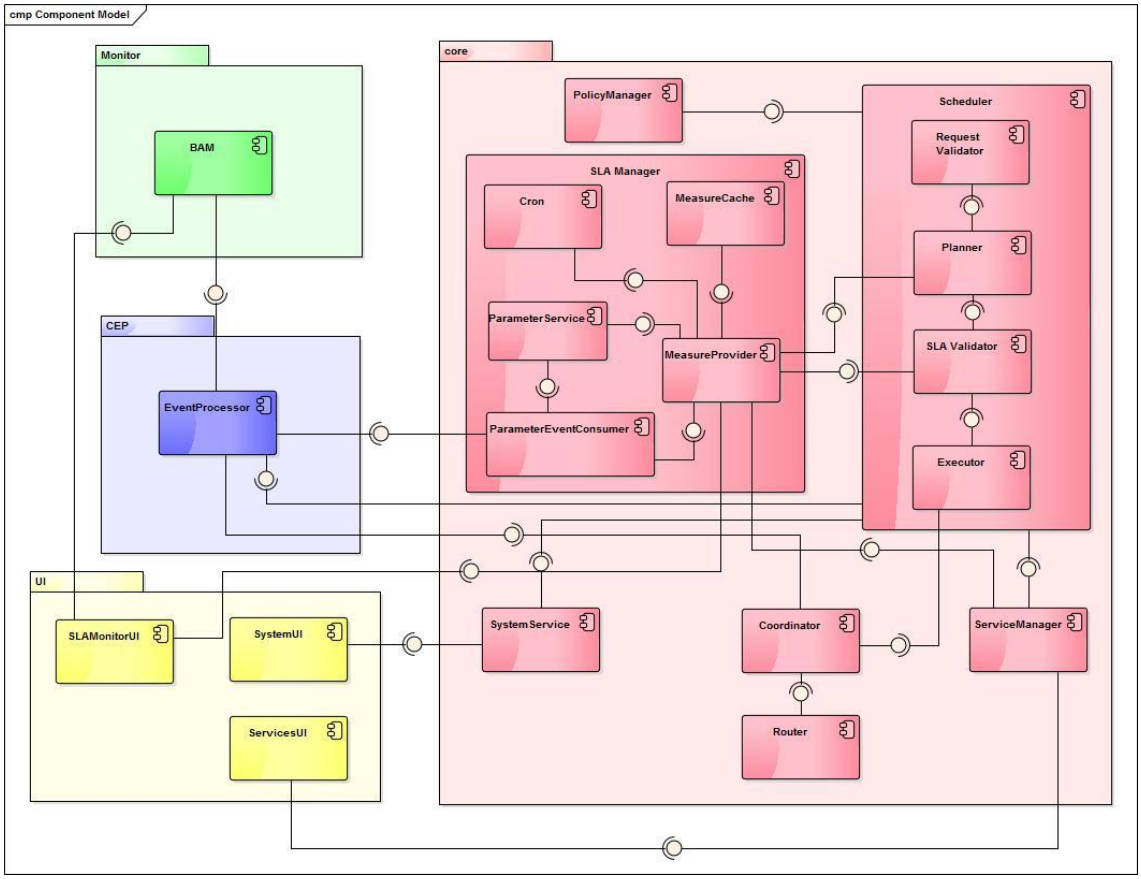

Figura 24: Diagrama de componentes del sistema

4.2.2.1 Paquete Core

Incluye a todos los componentes que forman parte del Core del sistema. A continuación se detalla cada uno de ellos.

Scheduler

Es el encargado de procesar, validar y ejecutar los pedidos realizados por los clientes de acuerdo a la política seleccionada, el SLA definido y el estado de los servicios.
Dada su complejidad, está dividido en cuatro sub-componentes cuyas funciones se detallan en el anexo B1.

SLA Manager

Este componente es el encargado de la gestión de los distintos SLA de los servicios involucrados en el sistema. La gestión de dichos SLA implica un conjunto grande de funcionalidades, las cuales van desde la recepción y el procesamiento de eventos, manejos de caché para lograr accesos a los datos (con la velocidad que demanda el sistema) hasta tareas automáticas que permitan determinar cuándo se deben calcular los valores de las métricas para los distintos períodos de tiempo considerados.
Las funciones principales del componente SLAManager son:

- **Gestionar los datos del modelo de calidad para cada uno de los servicios involucrados y su evolución en el tiempo.** Debe permitir la eliminación de datos históricos que ya no sean relevantes para el modelo (a modo de ejemplo, la eliminación de datos con fecha de creación anterior a una fecha determinada); lo que permite que los valores para los distintos factores de calidad estén actualizados. Además, debe poseer procesos automáticos que se encarguen de dicha actualización.
- **Brindar servicios a los demás componentes para la obtención de los valores de las distintas medidas consideradas en el modelo de calidad.** Esto implica poseer una interfaz de consulta a través de la cual se pueda conocer el valor de cada una de las métricas consideradas en el modelo, para cada uno de los servicios existentes. Se debe garantizar que los datos brindados estén actualizados y que en caso de no estarlo (o directamente, no poseer dato alguno) se indique el correspondiente valor teórico existente.

Dada la complejidad de este componente, se lo divide en componentes más específicos los cuales están detallados en el anexo B2.

Coordinator

El componente Coordinator es el encargado de resolver la ejecución de cada uno de los niveles del plan de ejecución generado por el componente Scheduler, interactuando con el subcomponente PlanExecutor. Dicha interacción se realiza de forma asincrónica, y los detalles de su implementación se encuentran en la sección 5.3.2.1. Es importante destacar que el componente Coordinator no posee la capacidad de invocar a los servicios finales, pero sí es capaz de resolver la ejecución completa del nivel, delegando la invocación a dichos servicios en el componente Router.
A modo de resumen las principales funcionalidades del componente Coordinator son:

- Control de tiempos de demoras de invocaciones, invocaciones exitosas, fallas y cantidad de invocaciones por unidad de tiempo.
- Generación de eventos en el sistema para los casos antes mencionados.
- Posibilidad de aplicar transformaciones tanto en el Request como en el Response para cada servicio que se debe invocar.
- Gestionar la invocación a los servicios de terceros y el correspondiente resultado, mediante la invocación al componente Router.

En el anexo B3 se presenta en formato resumido las dependencias existentes del componente Coordinator para con los demás componentes del sistema.

Router

El componente Router posee la función de realizar invocaciones a servicios finales, sin poseer manejo de la lógica complementaria al servicio (plan, nivel, tarea, etc). Es importante destacar

que este componente al realizar una invocación realiza el control del *timeout,* dato comunicado por el componente Coordinator al llevar a cabo las ejecuciones de los servicios del nivel que está procesando. El único componente que puede interactuar con el Router es el Coordinator.
El componente Router no invoca ninguna funcionalidad de otros componentes del sistema.

Policy Manager

Gestiona las distintas políticas existentes para la validación y ejecución de un plan. En un principio habrá tres políticas posibles (ToN, BE, DoP) pero está previsto que posea una interfaz lo suficientemente genérica y extensible de manera de facilitar la incorporación de nuevas políticas en el futuro. En el anexo D1 se detalla los pasos a seguir para el agregado de una nueva política al sistema.
El componente PolicyManager no invoca ninguna funcionalidad de otros componentes del sistema.

ServiceManager

Componente encargado de gestionar los servicios de terceros que se encuentran disponibles en el sistema. Brinda un conjunto de servicios que permiten las operaciones básicas sobre los recursos de tipo servicio (alta, baja y modificación). El componente Scheduler utiliza al componente ServiceManager a la hora de armado del plan de ejecución, ya que necesita obtener las URL de los servicios involucrados y las URI de los servicios de transformación.
El componente ServiceManager no invoca ninguna funcionalidad de otros componentes del sistema.

SystemService

Componente que gestiona parámetros globales correspondientes al servicio brindado hacia los locales de pago. Por ejemplo, establece cuál es el tiempo máximo de respuesta para dicho servicio en los casos en los que el pedido carece de ese valor.
El componente SystemService no invoca ninguna funcionalidad de otros componentes del sistema.

4.2.2.2 Paquete CEP

En este paquete se abstrae la lógica de procesamiento de los eventos generados por el sistema. Actualmente el único componente perteneciente al paquete CEP (Complex Event Processing) es el EventProcessor.

EventProcessor

El componente EventProcessor es el encargado del procesamiento de los eventos generados en el sistema. El procesamiento de los eventos puede consistir en reenviar el evento a los consumidores registrados y/o en la generación de nuevos eventos a partir de los consumidos.
Funcionalidades principales:

- Consumir eventos generados en el sistema.
- Reenviar eventos a los correspondientes consumidores.
- Generación de nuevos eventos a partir de los recibidos.

En el anexo B4 se presenta en formato resumido las dependencias existentes del componente EventProcessor para con los demás componentes del sistema.

4.2.2.3 Paquete Monitor

Engloba a los componentes encargados de la gestión de la información generada a partir de los eventos ocurridos en el sistema con especial foco en el manejo de la ocurrencia de fallos. Actualmente está formado por un único componente (BAM).

BAM

El componente BAM (*Business Activity Monitoring*) permite el análisis y gestión de los eventos generados en el sistema y la generación de alertas ante la ocurrencia de eventos inesperados o fallas. Además brinda una interfaz para poder consultar el histórico de fallas ocurridas.
El componente BAM no invoca ninguna funcionalidad de otros componentes del sistema.

4.2.2.4 Paquete UI

El paquete UI engloba a los distintos componentes que forman parte de la interfaz gráfica del sistema. A continuación se detalla cada uno de ellos.

ServicesUI

Interfaz gráfica que permite al usuario gestionar (alta, baja y modificación) los servicios de terceros que van a estar disponibles en el sistema.
En el anexo B5 se presenta en formato resumido las dependencias existentes del componente ServicesUI para con los demás componentes del sistema.

SLAMonitorUI

Interfaz gráfica que permite consultar las distintas fallas ocurridas y analizar las informaciones referentes a las métricas y el historial de las mismas para los distintos SLA existentes en el sistema.
En el anexo B6 se presenta en formato resumido las dependencias existentes del componente SLAMonitorUI para con los demás componentes del sistema.

SystemUI

Componente gráfico que permite al usuario del sistema la invocación de las distintas funcionalidades provistas por el componente SystemService, perteneciente al paquete Core del sistema.
En el anexo B7 se presenta en formato resumido las dependencias existentes del componente SystemUI para con los demás componentes del sistema.

4.2.3 Vista de procesos

Los principales procesos existentes en el sistema son:

- Cálculo de las medidas para tiempo de respuesta.
- Automatización del cálculo de las medidas.
- Procesamiento y ejecución de los pedidos realizados por los clientes (Recepción de Solicitud).
- Construcción y ejecución del plan.
- Ejecución de un nivel.
- Procesamiento de los eventos generados en el sistema.

En la tabla 15 se muestra un resumen de las relaciones existentes entre los procesos anteriormente mencionados.

Proceso	Relacionado con	Detalles
Cálculo automático de las medidas	Cálculo de medidas para tiempo de respuesta	Automatización del cálculo de las medidas
Recepción de solicitud	Construcción y ejecución de un plan	Parte del procesamiento del pedido realizado por un local de pagos implica la ejecución de un plan de trabajo
Construcción y ejecución del plan	Ejecución de un nivel	Parte de la ejecución del plan implica la ejecución de cada uno de los niveles que lo componen
Procesamiento de eventos	Cálculo de medidas para tiempo de respuesta	Los eventos que sean procesados por el sistema son luego utilizados para el cálculo de las correspondientes medidas

Tabla 15: Relaciones entre los principales procesos del sistema

A continuación se muestran en detalle cada uno de los procesos anteriormente mencionados.

4.2.3.1 Cálculo de medidas para tiempo de respuesta

Disclaim: Métrica y medida son conceptos diferentes, sin embargo en el resto del documento se utilizan indistintamente.
La viabilidad de poder invocar a un servicio cumpliendo con el SLA especificado refiere a que el Tiempo de Respuesta medido para el servicio sea menor o igual al exigido.
Sean $t_1, t_2, \dots, t_{n-1}, t_n$ los tiempos de respuesta ordenados de forma creciente (por duración) obtenidos para un período de tiempo y un servicio determinados. Se evaluaron dos alternativas para el cálculo de del valor de dicha métrica:

- Opción 1: Utilización del promedio de los tiempos de respuesta. El promedio del conjunto de tiempos de respuesta obtenidos para un período dado se define como:

$$\bar{X} = \frac{\sum_{i=1}^{n} t_i}{n}$$

- Opción 2: Utilización de la mediana de los tiempos de respuesta. La mediana del conjunto de tiempos de respuesta obtenidos para un período dado se define como:

$$M_{ed} = \begin{cases} t_i, & i = \dfrac{n+1}{2}; n \bmod 2 \neq 0 \\ \dfrac{t_i + t_{i+1}}{2}, & i = \dfrac{n}{2}; n \bmod 2 = 0 \end{cases}$$

La principal ventaja que ofrece la utilización de la mediana en vez del promedio es que evita la distorsión estadística de los valores atípicos, es decir, que ante la ocurrencia de que el tiempo de respuesta de un servicio en una invocación difiera mucho del resto de las invocaciones, la mediana resultante se verá poco afectada. Esto motivó la elección de la segunda opción para el cálculo del valor de la métrica *Response Time*.
Para calcular el valor de la métrica *Response Time* se tiene en cuenta el momento del día, el día de la semana y los valores históricos en un rango de tiempo cercano a ese momento.

4.2.3.2 Cálculo automático de las métricas

En la figura 25 se muestra mediante la utilización de un diagrama de secuencia cómo las métricas del sistema son calculadas cada un cierto tiempo configurable. Es importante destacar que se descartó la posibilidad de que los valores de las métricas fueran calculados en tiempo real, sino que se utilizó este mecanismo dado que permite evitar la degradación del

sistema por el procesamiento requerido en el cálculo de las métricas. El componente Cron (perteneciente al SLAManager) es el encargado de controlar la ejecución del cálculo efectivo de las métricas.

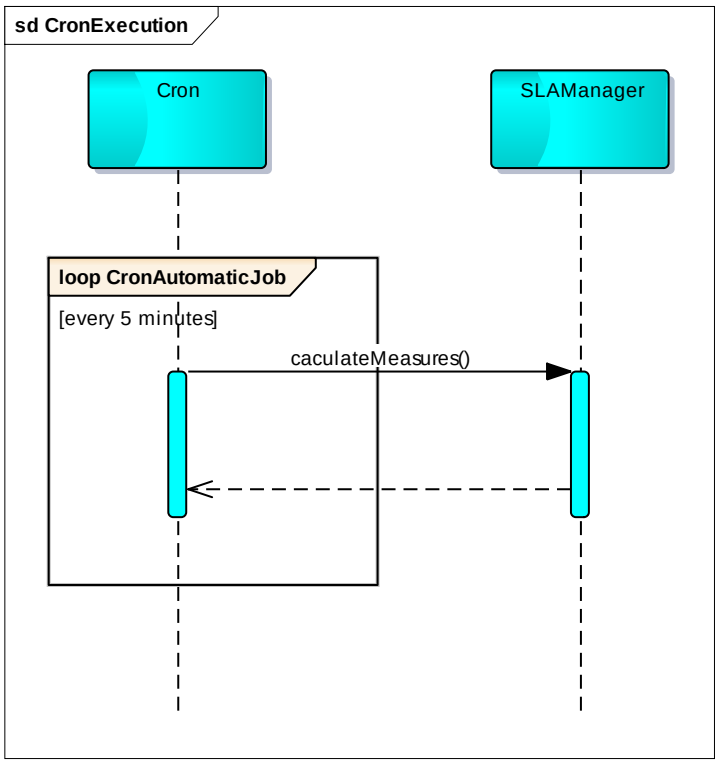

Figura 25: Diagrama de secuencia para la ejecución automática del Cron.

4.2.3.3 Recepción y procesamiento de solicitud

Toda invocación al sistema es procesada por el componente Scheduler. Como se detalla en el anexo B1 este componente está compuesto por otros cuatro componentes (RequestValidator, Planner, SLAValidator y Excecutor), de manera de lograr encapsular las distintas tareas que implican procesar los pedidos de los clientes en componentes diferentes. Al proceso que implica el procesamiento completo de dichos pedidos lo llamamos Algoritmo de Scheduler. En la figura 26 se puede observar un diagrama que representa las diferentes fases ejecutadas por el algoritmo para validar la viabilidad de la ejecución del pedido, armar un plan de trabajo y ejecutarlo.

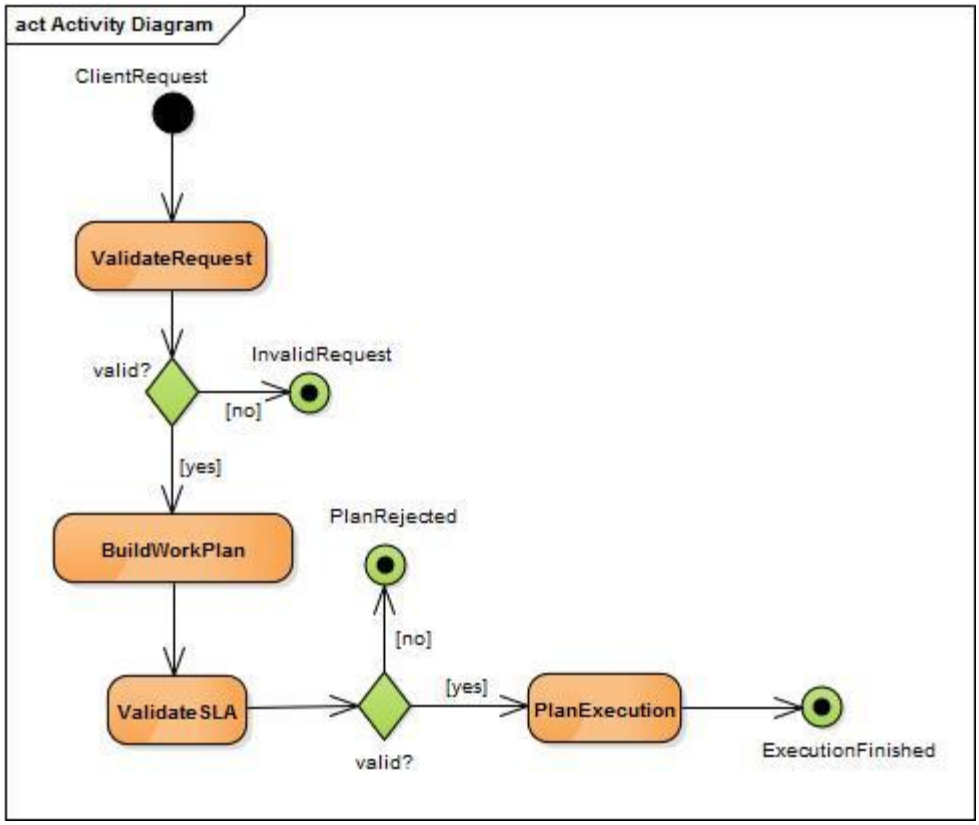

Figura 26: Diagrama de actividad del algoritmo de scheduler

Validación del Request (ValidateRequest)

Se valida la política a emplear, el SLA propuesto, la estructura del mensaje en general y se realizan validaciones de tipos de datos.

Se verifica que los tipos de servicios puedan ejecutarse según la política elegida. Se delega a la política especificada en el pedido, o en su defecto la política global del sistema, la validación de los tipos de los distintos servicios involucrados.

Construcción del Plan de Trabajo (BuildWorkPlan)

En esta fase se crea el plan de trabajo delegando a cada política de ejecución su construcción. El plan de trabajo es una estructura de grafo utilizada para modelar la forma en que se deben de invocar los servicios definiendo orden y concurrencia. Este plan es construido por la política seleccionada tomando en cuenta factores como la clasificación del servicio o si define prioridades. Este plan luego será el *input* para el motor de ejecución. En la sección 4.2.3.4 se describe en detalle la estructura del plan y cómo cada política de ejecución determina su composición.

Validación del SLA (ValidateSLA)

En esta etapa se valida que el plan de trabajo cumpla con el SLA. Dado el plan de ejecución, se ponderan las tareas de acuerdo al valor histórico del costo de la invocación a los servicios o al valor teórico y se suma el *overhead* calculado por el sistema. Se calcula el camino de costo máximo del grafo que representa al plan de trabajo y se utiliza como valor teórico para la ejecución de todo el plan.

Si el tiempo calculado es menor al establecido por el local de pagos para procesar todo el *request* se ejecuta todo el plan. Si el tiempo calculado es mayor al SLA establecido para procesar todo el *request* entonces queda en manos de cada política de ejecución el resolver si puede ejecutar o deben de realizarse cambios en el plan de trabajo. En la sección 4.2.3.4 se

describen las políticas y cómo estas afectan a la ejecución de un plan en los casos antes mencionados.

Ejecución del plan (PlanExecution)

La ejecución del plan consiste en recorrer la estructura del plan de trabajo, creado y validado por las etapas anteriores, y pasarle cada nivel a un componente más especializado (*Coordinator*) quien es el responsable de su ejecución. En la siguiente sección se describe con más detalle esta interacción entre componentes.

4.2.3.4 Construcción y ejecución del plan

En la figura 27 se muestra cómo se ejecuta el procesamiento de un pedido realizado por un cliente. En particular, se destaca que el pedido del cliente es respondido de forma sincrónica por el sistema, mientras que luego de generar el plan, lo ejecuta de manera asincrónica. Además se muestra cómo ocurre asincrónicamente la generación de eventos.

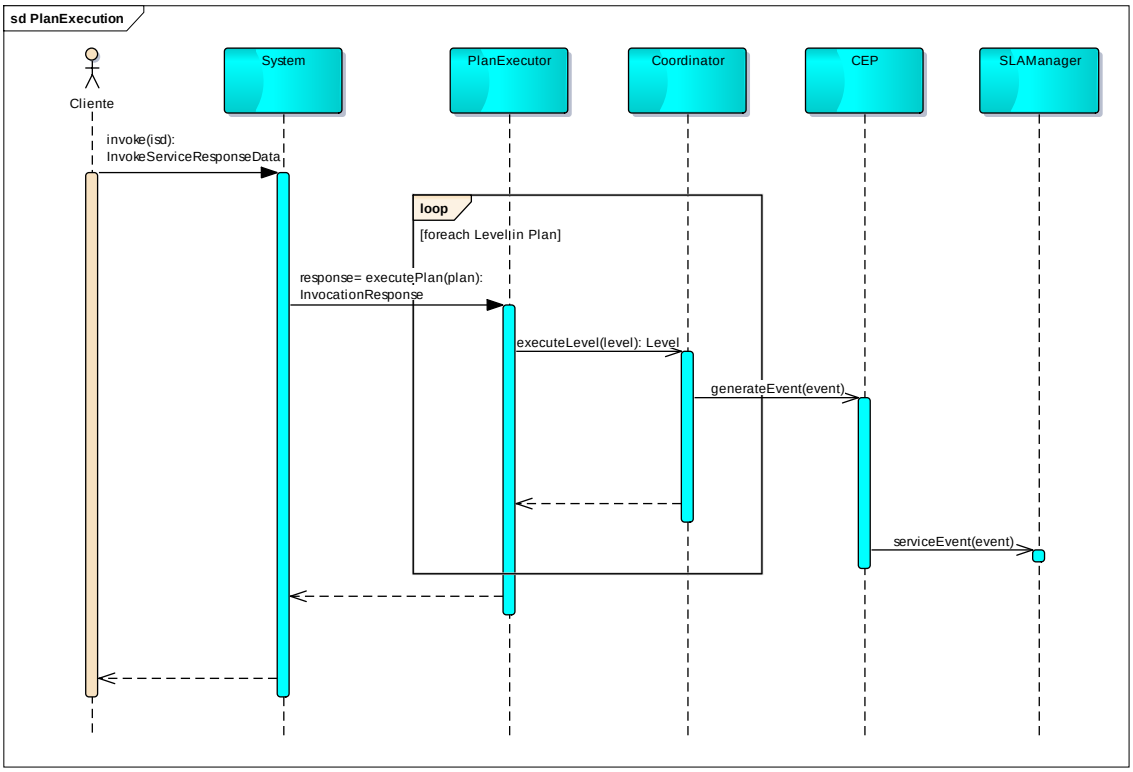

Figura 27: Diagrama de secuencia de un pedido al sistema

Plan de trabajo

Se definió una estructura llamada Plan de Trabajo que permitiera modelar la ejecución de los pedidos de los clientes, construyéndose un plan para cada pedido realizado. La estructura del plan determina el orden de invocación de los distintos servicios de terceros especificados en el pedido.

Un plan se compone de un conjunto ordenado de niveles, que determinan una dependencia de ejecución entre sí para que luego el encargado de ejecutar el plan ejecute cada nivel de forma serial. Cada nivel está compuesto por un conjunto de tareas. Las tareas pertenecientes a un mismo nivel poseen la característica de que serán ejecutadas concurrentemente. Por último, cada tarea está compuesta por un conjunto ordenado de invocaciones a servicios que cumplen la condición de que serán ejecutadas de forma serial.

En la figura 28 se muestra un ejemplo de la estructura de un plan de trabajo de tres niveles.

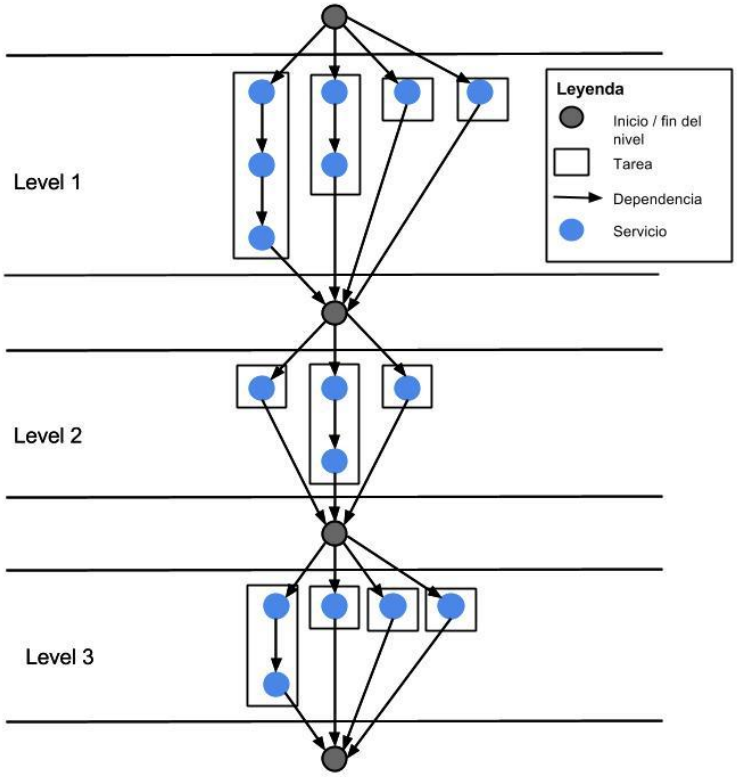

Figura 28: Ejemplo de la estructura de un plan de trabajo

La estructura propuesta para el plan de trabajo es lo suficientemente genérica como para no poner restricciones al momento de su construcción. Uno de los requerimientos planteados en la sección 3.4.1.2 establece que dentro de un mismo pedido las invocaciones a un mismo servicio de terceros deben ser ejecutadas de forma serial, sin embargo la estructura del plan no pone restricción alguna sobre las ejecuciones a realizar. Por lo tanto cualquier restricción que se quiera modelar es delegada a cada política en el momento de construcción del plan.

Construcción de un plan de trabajo

Para la construcción del plan de trabajo, se descompone el pedido (*request*) identificando cada uno de los servicios finales a los cuales es necesario invocar, se los clasifica y se arma un plan de ejecución de acuerdo a las características de cada servicio involucrado y a la política seleccionada.

Se descompone el pedido en servicios $S_1, S_2, \dots, S_n$ y a cada servicio se le asocia una tarea. Las tareas se agrupan por nivel y la política determina si existen dependencias entre los servicios y se agregan o no en una misma tarea.

La creación del plan y la organización de los servicios en diferentes tareas y niveles están delegadas a cada política de ejecución y son estas políticas las encargadas de modelar el plan.

Si la política es ToN el plan constará como máximo con dos niveles de ejecución, a lo sumo con un servicio simple o cancelable (en el nivel dos del plan) y el resto de los servicios serán anulables (nivel uno). El servicio simple o cancelable, si es que hubiera uno, se ejecutará último. Esta restricción se impone debido a que si un servicio fallara, la política obliga a volver a atrás todas las invocaciones realizadas y por esa razón estas deben ser anulables y solo la última puede ser no anulable.

En caso que luego de calcular el costo total de ejecución del plan diera mayor que el tiempo de respuesta definido por el local en el SLA, no se ejecutará el plan. En caso de que una de las

invocaciones diera error o no se pudiera invocar todos los servicios en el tiempo establecido se procederá a ejecutar las compensaciones de los servicios antes invocados.
Si la política es BE el plan constará de un único nivel. En caso que luego de calcular el costo total de ejecución del plan diera mayor que el establecido en el pedido, se realizará una poda del plan a los efectos de quedarse con la mayor cantidad de servicios a invocar en el tiempo establecido. Esta política intentará ejecutar todo lo que se pueda durante el tiempo previsto. En caso de que se llegue a consumir el tiempo establecido se cortará la ejecución, no se ejecutarán más invocaciones a servicios y se procederá a devolver los resultados ya conseguidos.
Si la política es DoP el plan puede contener varios niveles y cada nivel contendrá los servicios con la misma prioridad. Esta política tiene el mismo comportamiento que BE salvo que la estructura del plan, como ya se comentó, puede tener varios niveles. Si se ejecuta una pedido al sistema en donde todos los servicios tienen la misma prioridad, entonces DoP se va a comportar de la misma forma que un BE.

Ejecución del plan de trabajo

Para la ejecución de un plan se toma cada uno de los niveles que lo componen y se ejecutan de forma serial. La ejecución de un nivel consiste en invocar al componente *Coordinator* pasándole como parámetro en la invocación el nivel que este debe de ejecutar. Para cada nivel se invoca al *Coordinator* y se procesa la respuesta, y en caso de ser un procesamiento exitoso se procede a ejecutar el siguiente nivel repitiendo esto hasta ejecutar el último de los niveles. En caso de errores en la ejecución se pueden aplicar compensaciones dependiendo de la política empleada. Luego de terminar con la ejecución del plan, se procesan todas las respuestas y se arma la respuesta *(response)* que se va a devolver al cliente.

4.2.3.5 Coordinador – ejecución de un nivel

Otro de los procesos clave que existe en nuestro sistema es la ejecución de un nivel por parte del coordinador. La ejecución de un plan puede resumirse a la tarea de invocar al componente Coordinator tantas veces como niveles posea el plan.
A continuación se describen los distintos pasos que implica la ejecución de un nivel mediante la utilización de los Patrones de Integración Empresarial (EIP). Es importante destacar que el diseño mostrado en la figura 29 corresponde al diseño teórico realizado previo a la implementación del sistema, y a pesar de que es estructuralmente similar, la implementación estuvo sujeta a restricciones impuestas por la tecnología. En la sección 5.3.2.2 se puede ver en detalle la implementación del componente Coordinator.

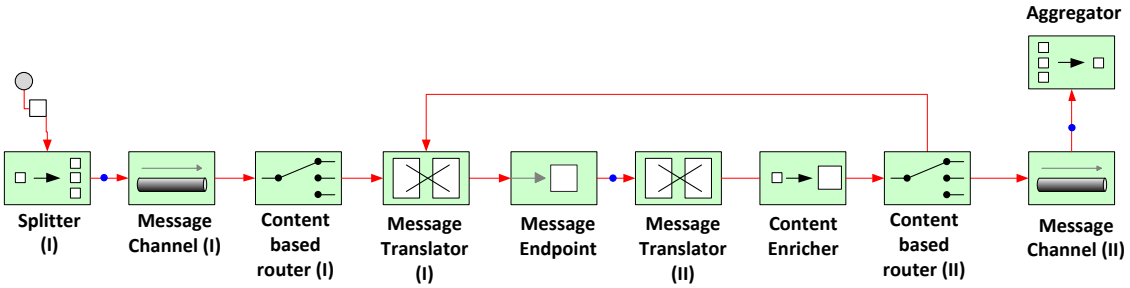

Figura 29: Diseño teórico de la ejecución de un nivel

Se utiliza el patrón *Splitter* para descomponer el nivel en todas las tareas de manera de que puedan ser ejecutadas de forma concurrente, generándose un conjunto de mensajes que son enviados a la cola de mensajes *Message Channel(I)*. Un componente de tipo *Content Based Router(I)* es el encargado de procesar dichos mensajes y examinar el contenido de los mismos para determinar cuál es el servicio final que deben invocar (una o más veces, pero siempre el mismo servicio).

Previo a la invocación de cada servicio final se aplica una transformación (*Message Translator(I)*) a los datos (en caso de que haya sido especificada), luego se realiza la invocación propiamente dicha utilizando el componente *Message Endpoint* y se aplica una nueva transformación a la respuesta obtenida (*Message Translator(II)*). Además, se utiliza el patrón *Content Enricher* para agregar información complementaria al mensaje en caso de que sea necesario (por ejemplo, registrar la demora de la ejecución del servicio). Cuando finaliza el llamado a un servicio, el componente *Content Based Router(II)* analiza si aún quedan servicios por invocar dentro de dicha tarea: en caso afirmativo se repite la ejecución con el siguiente pedido y en caso contrario se envía un mensaje con el resultado de la ejecución de la tarea a la cola de mensajes *Message Channel (II)*.
Por último los mensajes se vuelven a agrupar utilizando el patrón *Agreggator*.
Dadas las características de nuestro sistema y la importancia que tiene el poder obtener datos relativos al procesamiento y ejecución de cada pedido (por ejemplo: tiempos de demora, tiempos de procesamiento, éxito o falla de invocaciones a servicios, etc.), se utilizan componentes de tipo *Wire Tap* en distintos puntos del flujo de la ejecución para la generación de dichos eventos. Dichos componentes no aparecen en el diagrama para mantener la claridad en el mismo, pero sí están indicadas sus ubicaciones mediante la utilización de puntos azules.

4.2.3.6 Procesamiento de eventos

Como ya se ha descrito anteriormente, el sistema debe ser capaz de estimar el tiempo de ejecución de un pedido. Por esta razón es que existe la necesidad de obtener información sobre el tiempo utilizado para efectuar las diferentes actividades que realiza el sistema. Estas actividades son:

- procesamiento del request
- creación de un plan de trabajo
- validación del SLA
- ejecución del plan de trabajo
- invocación a cada servicio final
- generación de la respuesta

Esta información es utilizada para calcular el Overhead del sistema al realizar algunas de las tareas antes mencionadas y para obtener el tiempo que demora en responder cada servicio final.
A los efectos de que el sistema pueda recabar esta información es que se define una arquitectura EDA para la cual contamos con un bus de comunicación, el propio sistema que es quien produce los eventos y el o los consumidores que se registran para procesar estos eventos. La generación y el consumo de los eventos se realizan de forma asincrónica a los efectos que el procesamiento de los eventos no afecte directamente la ejecución de una invocación al sistema. En esta arquitectura el sistema va generando eventos para registrar el momento y la duración en la que se ejecutan las diferentes actividades. Los eventos son procesados por el componente CEP que los analiza y reenvía al canal que corresponda para que luego, un consumidor más especializado, guarde esta información. El componente CEP también genera eventos de servicio a partir de eventos de sistema.
Se identificaron dos tipos de eventos:

- Eventos de sistema. Son eventos que registran la ejecución de ciertas actividades del sistema. Durante una invocación al sistema se generan una secuencia de eventos que quedan correlacionados y de esta forma pueden ser trazados y utilizados para analizar una invocación. Este tipo de evento es consumido por el componente BAM.
- Eventos de servicio. Son eventos que describen la interacción con cada servicio final. Este tipo de evento genera la información que alimenta a los parámetros utilizados para el cálculo de las métricas procesados por el componente SLAManager.

Durante la fase de ejecución se invoca al Coordinador el cual dispara diferentes eventos de servicio antes y luego de invocar a cada servicio final. Los tipos de eventos de servicio son:

- ACK_MESSAGE. Se dispara cuando un servicio se ejecutó de forma exitosa.
- RESPONSE_MESSAGE. Se dispara cuando el servicio invocado responde.
- REQUESTED_MESSAGE. Se dispara cuando un servicio es invocado.
- RESPONSE_TIME. Se dispara cuando un servicio se ejecutó de forma exitosa y se agrega en la información del evento el Response Time del servicio.

Para el cálculo de las métricas se utilizan los datos de los parámetros recogidos por el sistema. Dado que los valores de las distintas métricas pueden variar en el tiempo, el sistema calcula las métricas siempre considerando un período de tiempo determinado.
A medida que el sistema es utilizado son generados nuevos eventos que inyectan más datos a los parámetros. El cálculo de las métricas solo tiene efecto sobre los servicios y métricas que dependen de parámetros para los cuales se procesaron nuevos datos.
El sistema cuenta con una cache persistente que es la encargada de mantener el valor de las distintas métricas de un servicio en un período de tiempo determinado.

4.2.4 Vista de despliegue

En la figura 30 se muestra un diagrama de despliegue el cual permite visualizar físicamente cómo están distribuidos los distintos componentes del sistema y cómo éstos interactúan con los agentes externos.

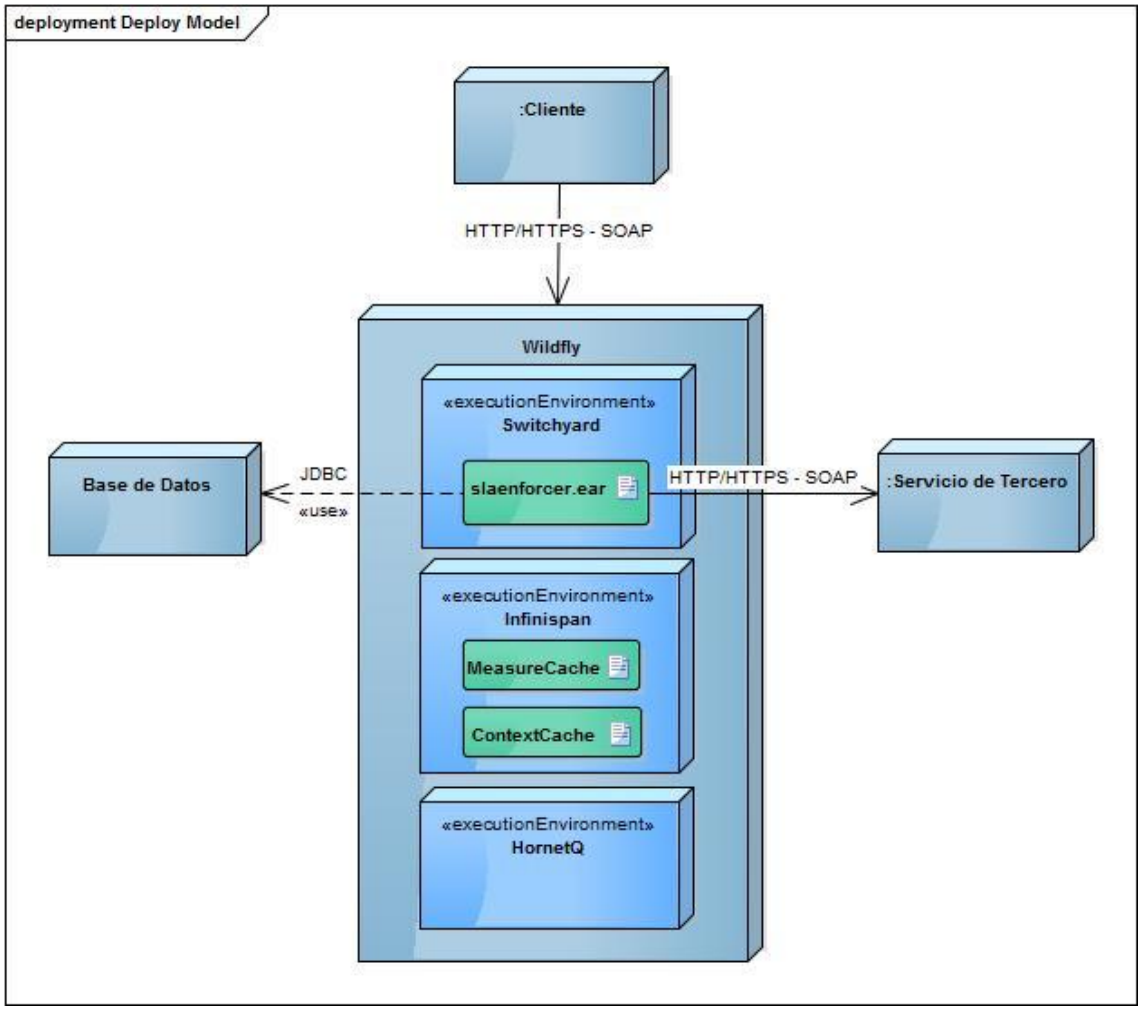

Figura 30: Diagrama de despliegue del sistema

Nuestro sistema ejecuta dentro de un servidor central Wildfly (en la imagen se muestra como un único nodo, pero éste puede ejecutar en un servidor compuesto por varios nodos, dependiendo de las necesidades de procesamiento del cliente). La aplicación resultante se empaqueta como un archivo .EAR (slaenforcer.ear) que ejecuta en el ambiente de ejecución Switchyard, que es la plataforma ESB sobre la cual está montado nuestro sistema. Además existe un ambiente de ejecución brindado por Infinispan donde están alojadas las dos cachés del sistema: MeasureCache y ContextCache. MeasureCache es el repositorio donde se almacenan los distintos datos de las métricas definidas y ContextCache es una caché utilizada para el manejo del contexto de la ejecución de un plan. Además existe un ambiente de ejecución HornetQ que brinda soporte a colas de mensajes utilizadas por la solución.
El componente slaenforcer.ear es la implementación completa de los distintos componentes mostrados en la sección 4.2.2. Interactúa con una base de datos utilizando el estándar JDBC, y con los servicios de terceros y locales que componen la red de pagos y cobranzas mediante la utilización de los protocolos HTTP/HTTPS y SOAP.

4.2.5 Características relevantes de la arquitectura

Desde el inicio se consideró diseñar e implementar el sistema de forma que éste tuviese en consideración aquellos aspectos que consideramos fundamentales para una aplicación con estas características. A continuación se describen en detalle las mismas.

4.2.5.1 Extensibilidad

El sistema implementa las funciones y los algoritmos fundamentales con la visión de dejar encapsulado en ellos todo aquello que sean mecanismos inherentes a la solución, y todo lo que esté relacionado a la toma de decisiones se prevé que pueda ser intervenido mediante la implementación de APIs.
Como principales ejemplos, se pueden tomar la implementación de las políticas del sistema, las métricas existentes y los servicios de terceros. En el anexo E se especifica las distintas interfaces que es necesario implementar en cada caso.

4.2.5.2 Escalabilidad

En la medida que el sistema se vuelva una pieza fundamental en la comunicación entre una organización y los proveedores de servicios externos vemos como característica fundamental del sistema el que este sea escalable. Para poder dotar al sistema con esta característica es que se tomaron las siguientes decisiones:

- Uso de mensajería como mecanismo de comunicación entre algunos de los componentes. Se utiliza para la invocación concurrente a los servicios finales y como canal para el soporte a los eventos. En el caso de que la solución no sea implantada sobre un clúster, los canales pueden configurarse como no persistentes, y de esa forma minimizar el *overhead* en este tipo de ambientes. En caso contrario, se debe tener en cuenta que dichos canales deben ser persistentes.
- Se deduce del punto anterior que la cantidad de consumidores de los mensajes debe ser configurable a los efectos de poder incrementar o decrementar la cantidad de clientes concurrentes que puedan consumir de un mismo canal.
- Uso de cache distribuida para almacenar el cálculo de las métricas y como mecanismo para almacenar datos que son de uso constante y evitar así ir al repositorio de datos lo cual puede significar un mayor tiempo de acceso. En la sección 5.3.2.4 se explica en detalle la solución implementada.

5 Implementación

Esta sección detalla los distintos aspectos de la implementación del sistema. En principio se describen las diferentes tecnologías utilizadas en la implementación, así como las herramientas de desarrollo y testing, y luego se explica cómo fueron utilizadas algunas de las tecnologías más relevantes. En las siguientes secciones se explican detalles del desarrollo como decisiones tomadas durante la implementación y cómo se implementó algunos de los componentes y funcionalidades del diseño. Por último se describen las pruebas realizadas al sistema y la metodología utilizada para la implementación.

5.1 Tecnologías

La implementación del sistema fue realizada teniendo en cuenta los requerimientos funcionales y no funcionales especificados en la sección 3.4. Uno de los dos requerimientos no funcionales establecidos por el cliente fue utilizar Switchyard como plataforma ESB sobre la cual desarrollar el sistema. Debido a eso se consideró clave la utilización de componentes brindados por Switchyard siempre y cuando fuera viable a lo largo de todo el sistema.
A continuación se describe la utilización dentro del sistema de cada una de las tecnologías / *frameworks* / servidores involucrados en el desarrollo.

5.1.1 Wildfly

Se utilizó la versión 8.1.0.Final de Wildfly [44] como servidor de aplicaciones sobre el cual se ejecuta la aplicación. No solo forma parte de los requerimientos no funcionales establecidos por el cliente sino que también es considerado un servidor flexible, liviano y que permite escalar con facilidad.
Wildfly soporta el estándar JEE7 [45], por lo que se buscó la utilización de las distintas ventajas que posee a lo largo de todo el sistema. A continuación se detalla qué componentes de dicho estándar fueron utilizados y de qué manera. Además se hace referencia a la implementación de caché brindada por Wildfly.

5.1.1.1 Context and Dependency Injection (CDI)

Se utilizó CDI [46] para:

- La inyección de dependencias entre distintos componentes del sistema. Mediante la utilización de interfaces y *qualifiers* [47] se logró mantener independencia entre los componentes, ocultando las distintas implementaciones de los componentes inyectados. Un ejemplo de la utilización de CDI con *qualifiers* puede verse en la utilización de las políticas por parte del componente PolicyManager, que es inyectado en la implementación del componente Scheduler.
- La utilización de *Interceptors* e *Interceptor Bindings* [47]. Se mantiene la lógica implementada por el interceptor fuera de los lugares donde éste sea utilizado. En particular son utilizados para la generación de eventos en los distintos componentes del sistema.

5.1.1.2 EJB 3.1

Se utilizaron Enterprise Java Beans (EJBs) [48] de tipo *Stateless* para la implementación de la lógica de negocio de los distintos componentes del sistema que requieren el acceso tanto a la base de datos como a la caché del sistema.

5.1.1.3 JMS / HornetQ

Java Message Service (JMS) [49] es el estándar JEE para mensajería y HornetQ [50] la implementación de JMS que posee Wildfly. Se utilizó colas de mensajes a lo largo del sistema, destacándose tres puntos críticos:

- Comunicación asincrónica entre los componentes PlanExecutor y Coordinator.
- Generación de eventos y procesamiento asincrónico de los mismos.
- Paralelización de ejecución de tareas en el componente Coordinator (en combinación con el patrón *Splitter*).

5.1.1.4 JPA / Hibernate

Java Persistence API (JPA) [51] es el estándar JEE para persistencia. JBoss provee Hibernate [52] como implementación de JPA. Se utilizó Hibernate para el acceso a los datos almacenados en la base de datos.

5.1.1.5 Infinispan

Infinispan [53] es un almacenamiento de datos de tipo clave/valor escalable y con alta disponibilidad. El propósito de Infinispan es exponer una estructura de datos distribuida y concurrente. Puede ser utilizada como memoria caché distribuida así como una almacenamiento clave / valor NoSQL o como base de datos de objetos.
Las distintas caché del sistema se implementaron sobre Infinispan. En las secciones 5.3.2.1 y 5.3.2.4 se detallan los distintos usos que se le dio en el sistema.

5.1.1.6 JSF

Para la implementación de la interfaz gráfica se utilizó JSF [54]. Esta tecnología es parte de la especificación JEE7 [55] que provee el servidor de aplicaciones.

5.1.2 Switchyard

En la sección 2.7.1 se incluye una descripción de Switchyard. Esta sección pretende mostrar la utilización de los distintos componentes de Switchyard utilizados dentro del sistema. Dentro de los componentes que brinda Switchyard destacamos la utilización de Service (para declarar un servicio Switchyard) y Reference (para invocar un servicio Switchyard) [56].

5.1.2.1 Apache Camel

En la sección 2.7.2 se brinda una introducción a Apache Camel. Camel está integrado a Switchard, y en particular se utilizó como motor de ruteo en la implementación del componente Coordinator.

5.1.3 Rich Faces

Para la implementación de la interfaz gráfica también se utilizó RichFaces [57], un *framework* que integra a JSF otros componentes gráficos más avanzados y provee soporte Ajax. En particular se utilizaron las gráficas para visualizar los tiempos de ejecución de un plan.

5.1.4 Soap UI

El software SoapUI [58] fue utilizado para la realización de las distintas pruebas del sistema. En particular se implementó los servicios *mock* y los distintos *tests*, tanto de funcionamiento como de carga.

5.1.5 Apache Maven

Se utilizó Apache Maven [59] para la gestión de dependencias, compilación e integración de Switchyard con el proyecto. Esto permitió mejoras significativas en los tiempos de desarrollo y en el correcto empaquetamiento de la aplicación.

5.1.6 JBoss Developer Studio

Dado que el sistema debía ser desarrollado en Java y Switchyard se decidió utilizar JBoss Developer Studio (JBDS) [60] como IDE basado en Eclipse. Este IDE provee *plugins* para la implementación de composiciones Switchyard e integración con Maven y SVN.

5.1.7 SVN

Como herramienta para control de versiones se utilizó SVN [61].

5.2 Uso de tecnologías

En esta sección se comenta cómo fueron utilizadas en el proyecto. El proyecto fue desarrollado sobre Switchyard como ESB y Camel como motor de ruteo. Estos componentes vienen integrados al servidor de aplicaciones Wildfly el cual se compone de varios sistemas dentro de los cuales se encuentra Infinispan utilizado como cache distribuida. Otro sistema que fue evaluado es Overlord el cual no fue utilizado y se explica por qué fue dejado de lado en el proyecto.

5.2.1 Switchyard

Los principales usos que se le dio a Switchyard fueron:

- Componer servicios a partir de la interacción entre distintos componentes.
- Exponer y/o consumir servicios, haciendo transparente hacia los componentes de la composición el mecanismo de comunicación, delegando su resolución al framework.
- Ofrece diferentes mecanismos para implementar un componente entre los que se encuentran: clases Java, Camel (XML/DSL), Drools, etc.
- IDE con interfaz gráfica amigable que permite una rápida interpretación y edición de la composición.
- Integración con CDI lo cual permite que los componentes Switchyard puedan ser inyectados en JavaBeans utilizando CDI.

- Integración con Camel como motor para transformaciones y ruteo.

El principal problema encontrado al utilizar Switchyard fue el grado de integración con Camel. Camel provee una gran cantidad de parámetros por los cuales se puede configurar sus componentes en runtime, mientras que utilizándolo a través de Switchyard esto no es posible aún, lo cual obligó en algunos casos a desestimar el uso de algunas funcionalidades que este provee.
El uso de Switchyard resultó intuitivo pero no tiene el grado de madurez esperado para un sistema de estas características ya que hace difícil poder sacarle provecho a Camel que es su principal componente. Además también se encontró que la documentación no se muy buena y los ejemplos son demasiado simples.

5.2.2 Apache Camel

Las principales características de Apache Camel como solución de integración de componentes son:

- Implementación de los Enterprise Integration Patterns (EIP): permite el desarrollo de soluciones mediante un lenguaje declarativo, utilizando patrones de diseño estándar. Pensar los problemas en función de patrones de integración empresarial permite subdividirlos en problemas de menor complejidad, utilizando mecanismos estándares.
- Ruteo utilizando Camel URIs: Camel permite realizar ruteo de mensajes mediante la utilización de Camel URIs. El ruteo puede implicar la interacción con múltiples conectores o la realización de transformaciones sobre el mensaje.

Camel es uno de los componentes de ruteo más utilizados dentro de los ESB Open Source por lo que se puede observar su continuo desarrollo y mejoras. La documentación es poco desarrollada pero al ser tan popular es fácil encontrar ejemplos y soluciones en la comunidad.

5.2.3 Infinispan

Infinispan fue utilizada de dos formas diferentes:

- Como cache distribuida para el manejo del contexto de ejecución de un nivel del plan, en donde se almacena el estado de ejecución.
- Como almacenamiento de datos NoSQL persistente y replicado. Este repositorio de datos es utilizado para almacenar los datos de las métricas calculadas. El almacén de datos puede ser configurado para que los datos permanezcan en memoria y se persistan a archivo, manteniendo en memoria un máximo de datos y con una política de desalojo tipo LRU. Con estas características se asegura la no volatilidad de los datos, el rápido acceso a los mismos, la disponibilidad de los mismos en los diferentes nodos del cluster y poder contener el uso de memoria.

Se pudo apreciar como un producto versátil, con buena documentación y fácil de configurar.

5.2.4 Wildfly

Wildfly fue utilizado como servidor de aplicaciones dando soporte a diferentes características utilizadas por el sistema:

- Servidor de aplicaciones web. La interfaz gráfica del sistema está basada en JSF y es desplegada dentro del contenedor de Wildfly.
- Utilización de mensajería asincrónica basada en JMS para la comunicación entre diferentes componentes del sistema. Los más relevantes son:
    - En el Coordinador para la ejecución de los niveles del plan y paralelizar la invocación a los servicios.
    - En la generación y procesamiento de los eventos del sistema.

- Acceso a los datos almacenados en base de datos utilizando JPA para abstraer la solución del manejador de base de datos a utilizar.
- Manejo transaccional de los datos.
- Soporte para conexión a base de datos mediante un pool de conexiones permitiendo un uso más eficiente de este recurso.
- Como contenedor para aplicaciones basadas en Switchyard.

5.2.5 Overlord

Una de las características del sistema es la generación de métricas, en particular para el tiempo de respuesta de los servicios finales y para el overhead del sistema. Luego de definir la arquitectura y hacer el diseño del sistema investigamos si existía algún componente de software que tuviera esta capacidad y se pudiera adaptar a nuestra solución. El sistema que más se adapta esta solución es el proyecto Overlord [88] de JBoss. Este sistema tiene un componente llamado Runtime Governance que se encarga de recolectar los response time de los servicios y tiene la capacidad de generar alertas escribiendo las reglas utilizando Drools. Si bien vimos que la arquitectura del este componente es similar a la nuestra, nos exigía tener que adaptar los servicios escritos en Switchyard a una forma que pudieran ser usados luego por Overlord para la generación de los eventos para los tiempos de respuesta. O sea para poder utilizar Overlord teníamos que escribir los servicios en Switchyard con una estructura diferente a la diseñada para que puedan ser tomadas las métricas por Overlord. Otra característica de nuestro sistema es que se pueden calcular otras métricas además del response time, funcionalidad que no encontramos en Overlord. El cálculo del overhead del sistema es complejo y no vimos que Overlord nos ayudara para tener esta información.
Por estas razones y dado el tiempo que implicaría el estudio a fondo de Overlord para poder adaptarlo a nuestra realidad y como este no es un componente exigido en los requerimientos del sistema es que decidimos dejarlo de lado. De todas formas nos parece que podría ser muy útil su estudio.

5.3 Aspectos de implementación

En esta sección se presentan los distintos aspectos relacionados a la implementación del sistema. En la sección 5.3.1 se presentan las decisiones de implementación tomadas y en la sección 5.3.2 se muestran aspectos de la implementación relativos al diseño detallado de la solución.

5.3.1 Decisiones

En esta sección se detallan las distintas decisiones tomadas en el momento de la implementación.

5.3.1.1 Timeout dinámico para la invocación a servicios

La invocación a los servicios de terceros se realiza estableciendo siempre un *timeout* para la misma. Dicho *timeout* es calculado dinámicamente utilizando la siguiente fórmula:

$$timeout = executionAvailableTime - (currentTime - executionStartTime)$$

Dónde:

- executionAvailableTime: es el tiempo máximo disponible para la ejecución de todo el pedido (es especificado por el cliente o se toma el tiempo por defecto del sistema).
- currentTime: tiempo actual.

- executionStartTime: momento en el que comenzó la ejecución del pedido.

5.3.1.2 Timeout para cancelación y compensación

Se decidió que era necesario que los servicios de terceros para las operaciones de compensación y cancelación contaran con un *timeout* para manejar los casos en los que éstos fallan. Es potestad del administrador del sistema definir dichos *timeouts*. Dichos timeouts deben ser especificados mediante la interfaz web al momento de dar de alta un servicio.

5.3.1.3 Invocación de servicios de transformación de request / response

Como se describe en la sección 5.3.2.2, la invocación de los servicios que implementan las transformaciones de los parámetros para la invocación y posterior procesamiento de los servicios de terceros se implementó utilizando el componente *Recipient List* de Camel. Dicho componente permite la especificación dinámica de cualquier URI que cumpla con alguno de los formatos de URI Camel. En este caso las transformaciones fueron implementadas utilizando el lenguaje XQuery [62], aprovechando además que Camel posee integrado el procesador de XQuery Saxon [63].
Los posibles formatos que deben tener las URI de las transformaciones son los siguientes:

- Opción 1:
  Xquery:file:[xquery.file.path]
  Siendo *[xquery.file.path]* la ruta absoluta del archivo que contiene la XQuery de la transformación.
- Opción 2:
  Xquery:file:${jboss.server.data.dir}\[xquery.local.file.path]
  Siendo:
  - *${jboss.server.data.dir}* una *System Property* configurada por el propio Wildfly que indica la dirección absoluta en *File System* del directorio data dentro de la instalación de Wildfly.
  - *[xquery.local.file.path]* la ruta relativa al directorio *data* de Wildfly del archivo que contiene la XQuery de la transformación.

Es posible sustituir la utilización de XQuery para realizar las transformaciones por cualquier otro componente aceptado por Camel. Para ello basta con modificar los registros de la base de datos de los servicios de terceros existentes colocando la nueva URI en el campo correspondiente a las transformaciones.

5.3.1.4 Caché de segundo nivel - Hibernate

Con el motivo de mejorar la performance de la aplicación, se configuró la caché de segundo nivel de Hibernate. Esta cache mantiene una copia de los datos en memoria dando la posibilidad que acceder a ellos de forma más rápida. Tanto la información de los servicios como la configuración global del sistema son datos que se consultan constantemente, pero que cambian de forma muy ocasional por lo que justifica el evitar tener que ir a la base cada vez que se necesita acceder a esta información para garantizar una mejor performance.
A continuación se detallan los cambios realizados en la aplicación para efectivizar el uso de dicha caché.

- En la configuración de Hibernate (archivo persistence.xml, proyecto core) fue necesario agregar las propiedades descritas en el Anexo D9.
- Las entidades que consideramos que debían estar dentro de la caché fueron anotadas con la *Annotation@Cacheable*.

5.3.1.5 Utilización de ServiceId en el pedido

Cuando un cliente realiza una invocación al sistema, esta invocación puede contar con varios pedidos de invocaciones a servicios de terceros. Cuando el sistema responde, el cliente debe poder identificar el resultado de la invocación a cada servicio con el que él solicitó invocar inicialmente. Si todos los servicio son diferentes el cliente no tendría problemas en identificar las respuestas ya que podría trazarla por el nombre del servicios, pero cuando una invocación al sistema está compuesta por varias invocaciones a un mismo servicio final, entonces debemos de proveer un mecanismo en el cual el cliente sea capaz de trazar el pedido de invocación a un servicio con su respectiva respuesta. Por esta razón es que se agregó un identificador del servicio el cual debe ser único por servicio en cada invocación al sistema. Este identificador se pasa como un dato más del servicio conteniendo un valor entero único a nivel del pedido. El sistema no hace controles a este identificador delegando al cliente la correcta invocación al sistema.
Como caso de ejemplo se muestra en la figura 31 una invocación al sistema dentro del cual hay dos pedidos de invocación al mismo servicio. La forma de vincular el pedido y su respuesta está dada por el serviceId como se puede ver marcado en dicha figura.

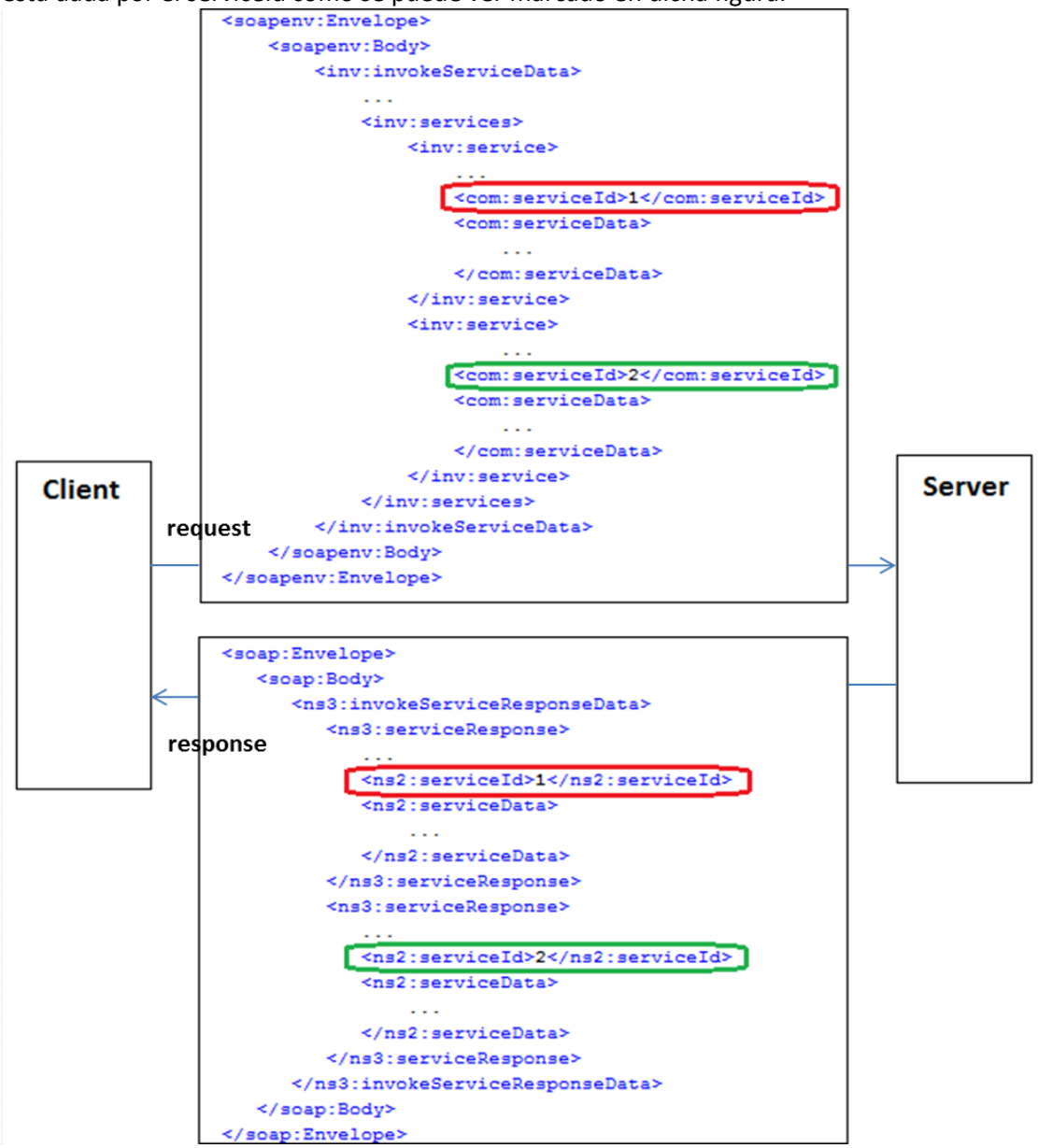

Figura 31: Ejemplo de request – response entre el cliente y el servidor

5.3.1.6 Utilización de HttpComponents y no utilización de camel-http4

Inicialmente el componente Router fue implementado utilizando componentes Switchyard. Las ventajas de tomar dicho camino radicaban en la integración que posee Switchyard con Camel y el soporte existente a componentes de tipo Camel URI. Además, fue necesaria la instalación del módulo camel-http4 en Wildfly, el cual provee soporte para la invocación de recursos externos vía HTTP [64]. Esto nos permitía asociar a una referencia Camel la URL del servicio de tercero a invocar y obtener la respuesta del mismo, utilizando un código más limpio.
A medida de que se realizaron pruebas con mayor profundidad se detectó un bug en la integración de Switchyard con el componente Camel URI. Dicho bug consiste en que al especificar parámetros estáticamente en la URL del componente Camel URI, la sustitución del valor de los mismos en tiempo de ejecución por el valor que efectivamente deberían tener no era realizada. En particular, el parámetro de nuestro interés era *httpClient.soTimeout* el cual permite especificar un *timeout* sobre la conexión creada. Es importante destacar que la no inclusión del parámetro de manera estática implicaba que éste no fuera considerado por más que fuese especificado dinámicamente.
Dado el bug encontrado se decidió cambiar la implementación del componente Router por una que no utilizara componentes Switchyard/Camel; para ello se utilizó la librería HttpComponents de Apache [65]. Para la especificación del *timeout* se agregaron los parámetros CONNECTION_TIMEOUT y SO_TIMEOUT definidos en la clase CoreConnectionPNames [66].

5.3.2 Diseño detallado

A continuación se detallan aspectos de la implementación del software que son considerados como parte del diseño detallado de la solución.

5.3.2.1 Contexto de la ejecución de un plan.

La comunicación entre el componente PlanExecutor y el componente Coordinator es asincrónica como se explicó en la sección 4.2.3.4, por lo tanto necesitamos definir un mecanismo para que ambos componentes puedan intercambiar la información relativa al contexto de ejecución. Para ello se decidió implementar una cache de datos de tipo clave/valor distribuida.
Wildfly ya posee soporte de caches de datos mediante la integración de Infinispan.
Es importante destacar que existen otros mecanismos para compartir memoria entre dos hilos de ejecución diferentes, pero el sistema fue diseñado de manera que pueda ser escalable. Si en un futuro se desea ejecutar la aplicación en un clúster de servidores es necesario que la implementación del mecanismo de comunicación entre los hilos sea mediante un mecanismo que escale de la misma manera que el resto de la aplicación (característica que ya posee Infinispan).
El contexto compartido por los componentes PlanExecutor y Coordinator es utilizado para compartir información sobre los planes que están siendo ejecutados:

- Notificar al componente PlanExecutor que hubo un error en la ejecución de un servicio en el nivel que está siendo ejecutado, para un determinado plan.
- Notificar al componente PlanExecutor que la invocación a un servicio en el nivel que está siendo ejecutado dio *timeout*, para un determinado plan.
- Notificar al componente Coordinator que la ejecución del nivel actual para un determinado plan corresponde a una ejecución de compensación.
- Obtener por parte del Coordinador si le resta tiempo disponible para ejecutar resto del nivel que está ejecutando.

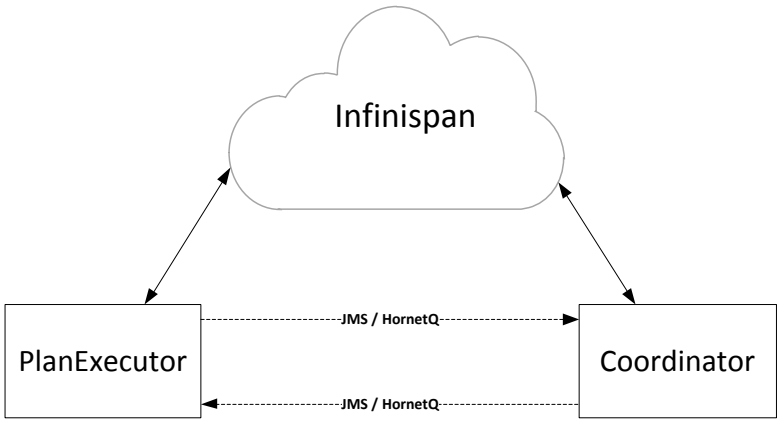

Figura 32: Ejemplo de interacción entre PlanExecutor y el Coodinator

Como se muestra en la figura 32, el mecanismo de comunicación es el siguiente:

El PlanExecutor toma el primer nivel del plan y lo coloca en una cola JMS para que el Coordinador lo ejecute. El PlanExecutor queda esperando la respuesta en otra cola utilizando un *selector* JMS (patrón Selective Consumer) con el identificador del nivel.

El Coordinador ejecuta el nivel, actualiza el contexto (caché Infinispan) y pone el resultado de la ejecución del plan en una cola JMS para que el PlanExecutor tome la respuesta.

Una vez que el PlanExecutor recibe la respuesta se fija en el contexto si la ejecución fue exitosa o hubieron errores. En caso que no hubiera errores se procede a ejecutar el siguiente nivel.

La razón por la cual se utiliza el contexto y no se pasa toda la información en el mensaje es porque el PlanExecutor, al quedar esperando por la respuesta del Coordinador, queda bloqueado. Para que no haya un bloqueo indefinido, el bloqueo del PlanExecutor tiene un *timeout* mayor al tiempo total estimado para la ejecución del plan. En caso de que la espera se termine por *timeout* se revisa el contexto para detectar si se produjeron errores y en caso de que no se hayan producido errores se da una nueva oportunidad para esperar el resultado.

Nunca se debería de dar el caso en el que el Coordinador no devuelva un resultado, pero si se produjera un error inesperado, el Coordinador va a marcar en el contexto el estado de error y el PlanExecutor va a liberar la ejecución así evitando que el sistema entre en un estado de degradación paulatina al quedar hilos colgados.

5.3.2.2 Implementación del componente Coordinator

En la sección 4.2.3.5 se presentó el diseño teórico del componente Coordinator. En la figura 33 se presenta el diseño final resultante, concebido en etapas tempranas de la implementación con el motivo de aprovechar al máximo las ventajas y minimizar los posibles inconvenientes de utilizar Apache Camel.

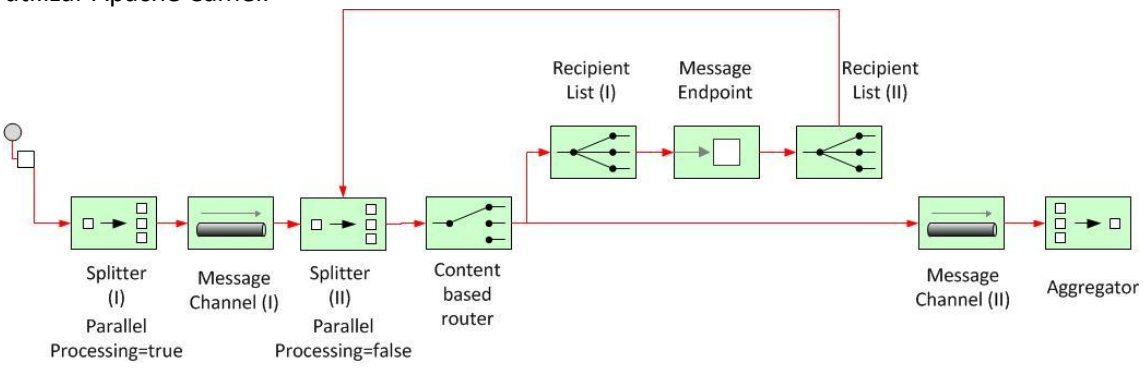

Figura 33: Diseño final de la ejecución de un nivel

Cambios realizados sobre el diseño original:

- Se sustituyen los componentes *Message Translator* por componentes de *tipo Recipient List*. Este cambio fue realizado debido a que la implementación de Camel del componente *MessageTranslator* no acepta la posibilidad de configurar la transformación a utilizar de manera dinámica. Por otro lado, el componente *Recipient List* permite la configuración dinámica de un conjunto de URIs (receptores) a las cuales se les envía una copia exacta del mensaje (en nuestro caso, dicha lista siempre posee un único elemento). Esto además trae consigo un beneficio adicional: las posibles URI a ser especificadas pueden ser cualquiera de las aceptadas por Camel; lo que permite pensar en las transformaciones como servicios genéricos cuya implementación actual es una transformación XQuery (mediante la URI Camel *xquery:XQueryResource*) pero que potencialmente puedan ser sustituidos en el futuro por cualquier servicio que la implemente.
- Se remueve uno de los componentes de tipo *Content Based Router* y se sustituye por la utilización de un componente de tipo *Splitter* (Splitter(II)), en la cual se divide la tarea a procesar en las distintas invocaciones a servicios que la componen. La implementación de Camel del componente *Splitter* posee una propiedad booleana *"paralellProcessing"* que permite configurar al componente para que realice el procesamiento de los mensajes de manera paralela o serial. Colocando el valor de dicha propiedad en *false* logramos el comportamiento deseado: la ejecución de los servicios dentro de una misma tarea se realiza de forma serial.
  Es importante destacar que también fue evaluada la posibilidad de utilizar el componente *Routing Slip* para el procesamiento serial de los servicios de una tarea, pero esto fue descartado ya que dicho componente permite que un mensaje pase por un conjunto establecido de pasos de procesamiento intermedios y en nuestra solución necesitábamos que el mensaje fuera diferente en cada "paso intermedio" (el servicio a ejecutar varía en cada paso).
- El componente *Content Based Router* ahora se encarga de verificar en cada iteración del *Splitter(II)* que aún le quede tiempo restante para procesar el siguiente servicio dentro de la tarea. En caso de no disponer de suficiente tiempo finaliza la ejecución enviando el mensaje a través del componente *Message Channel (II)*.

La implementación del componente Coordinator utilizando los patrones anteriormente mencionados permite la paralelización de la ejecución de las tareas de un nivel. Las tareas de un nivel son ejecutadas en paralelo mediante la utilización de un componente *Splitter*, un componente *MessageChannel* y un *Aggregator* (componente que se encarga de unir nuevamente la tarea en un nivel). Dicha paralelización podría haber sido implementada utilizando únicamente un *Splitter* y un *Aggregator*, pero se decidió incorporar a la misma un *MessageChannel* para poder brindarle al sistema una mayor escalabilidad y control sobre dicha ejecución. Para lograr mayor escalabilidad se podría contar con un ambiente distribuido, en el cual existan múltiples clientes consumiendo del Message Channel.
Además, la implementación actual permite que en un futuro se puedan agregar Message *Channels* nuevos, colocados de forma serial al actual, brindándole al sistema las siguientes ventajas:

- Realizar distintos tipos de procesamiento de las tareas dependiendo del *Message Channel* en el que sean colocadas.
- Establecer un límite de utilización de ciertos recursos, como la invocación a un determinado servicio de tercero, mediante la configuración de *Throttling* en ciertos *Message Channels* [67]. Limitar de esa manera la utilización de las colas de mensajes permite indirectamente crear un *thread pool* sobre el acceso a los servicios de terceros.

5.3.2.3 Cron

Buscando utilizar en lo posible componentes Switchyard (tanto por requerimientos del sistema como por simplicidad de uso) el componente Cron está implementado como un servicio Switchyard que utiliza un *binding* Camel para planificación (*scheduling)*. Para implementar tareas de tipo *scheduling*, Camel posee un componente CamelQuartz [68] el cual utiliza el planificador Quartz [69] como base para brindar dicha funcionalidad. La ventaja en la utilización de Quartz como motor para la planificación de tareas es la posibilidad de configurar el periodo de ejecución mediante la especificación de una CronExpression [70], las cuales poseen un alto poder de expresividad.

5.3.2.4 Gestión de las métricas del sistema.

Las métricas calculadas deben de ser almacenadas en un repositorio no volátil para poder ser consultadas durante la ejecución. Para esto se decidió no utilizar base de datos relacional y sí utilizar un mecanismo de persistencia en disco pero que utilice cache en memoria para mejorar el tiempo acceso a la información. También debe contar con replicación a otros nodos cuando se configure en clúster.
Para esta solución se utilizó Infinispan como una base de datos en memoria persistente para el almacenamiento de los valores calculados para las distintas métricas existentes en el sistema. Esto nos permite contar con un acceso rápido a los mismos manteniendo la posibilidad de que éstos sean persistidos en disco.

5.3.2.5 Cálculo del overhead del sistema

Una de las características principales del sistema es el poder estimar el tiempo que va a demorar en responder a un pedido. Para esto se tienen en cuenta el tiempo de respuesta de cada servicio y el tiempo adicional que agrega el propio sistema al ejecutar.
Desde el punto de vista del cliente que invoca al sistema, el tiempo que el sistema consume en la ejecución es el parámetro ServerLatency. Este parámetro es el que involucra al sistema y debe respetar el tiempo establecido por el cliente en la invocación. Como vimos en la sección 4.1.2.2 parte del ServerLatency está compuesto por el Overhead del sistema.
Desde el punto de vista del sistema, la ejecución de un pedido involucra varias actividades como:

- validar el pedido
- armar un plan de ejecución
- validar si es posible la ejecución
- ejecutar el plan
- armar la respuesta

Para el sistema, el ServerLatency resulta de la suma de la duración de cada una de estas actividades. Al validar si se puede cumplir con el tiempo acordado en el punto 3, las actividades anteriores ya ocurrieron por lo cual ya se conoce el tiempo que demoraron en ejecutar, pero se debe estimar el tiempo que llevará la ejecución de las siguientes actividades para validar si es posible la ejecución del plan. Por simplificación del sistema vamos a despreciar el tiempo que demora el sistema en armar la respuesta, por lo que el sistema debe estimar el tiempo que se demora en la ejecución del plan.
Para calcular el tiempo que demorará el sistema en ejecutar un pedido se estima cuánto es el tiempo que el sistema demorará en ejecutar el plan de trabajo.
Para calcular el tiempo de ejecución de un plan de trabajo se lo descompone en los tiempos de ejecución de cada uno de sus componentes, así es que tenemos el tiempo de ejecutar un plan, el tiempo de ejecutar un nivel, y el tiempo de ejecutar una tarea. El tiempo de ejecución de

una tarea es el tiempo que demora el Coordinador en ejecutarla menos el tiempo de respuesta al invocar al servicio final. Este tiempo es calculado por el Coordinador y pasado como resultado de la ejecución del nivel al PlanExecutor. El componente PlanExecutor es quien calcula el tiempo de ejecución de cada nivel y de todo el plan. Cuando termina la ejecución del plan, el PlanExecutor genera un evento de sistema con toda la información de ejecución del plan incluidos los tiempos antes mencionados. El componente CEP es quien procesa los eventos generados por el sistema y si la ejecución fue exitosa genera tres nuevos eventos de servicio. Cada uno de estos eventos alimenta el parámetro utilizado para el cálculo de la métrica de Overhead. Luego en cada invocación al sistema se utiliza la métrica de Overhead del plan, de cada nivel y de tarea para estimar el tiempo que demorará el sistema en la ejecución.
En particular hay que destacar que para el cálculo de la métrica de Overhead se toman los últimos 100 datos para que el valor sea lo más cercano a la realidad reciente y para el valor de esta métrica se utiliza el cálculo de la mediana.
En resumen, el cálculo de la métrica Overhead para la ejecución de un plan puede expresarse de la siguiente manera:

$$\text{totalOverhead} = \text{planOverhead} + (\text{levelOverhead} + \text{taskOverHead}) * \text{levelsCount}$$

5.3.2.6 Utilización de CEP, parámetros y métricas

Como ya hemos descripto anteriormente, el componente CEP es el encargado de procesar los eventos generados por el sistema. En particular, todos los eventos del tipo servicio son ruteados hacia el canal adecuado en el cual está registrado el SLAManager. Dentro del SLAManager hay un componente especializado que consume este tipo de eventos y los persiste en base de datos. Cada evento procesado alimenta los parámetros que luego son utilizados para calcular las métricas. Cada métrica se calcula en función de un momento del día y de los parámetros utilizados en su cálculo.

5.3.2.7 Parámetros Dirty

Dado la relación que existe entre las métricas, los parámetros y la fecha, podemos establecer que al generarse un evento y ser procesado por el SLAManager, y agregar un nuevo dato a los parámetros, las métricas que dependan de este parámetro y hayan sido calculadas en esa fecha, quedaron desactualizadas. Para evitar el recálculo continuo cada vez que llega un nuevo evento se decidió que el recálculo de las métricas es ejecutado periódicamente por el componente Cron el cual fue previamente descrito. Al ejecutar un recálculo es necesario conocer qué métricas necesitan ser recalculadas y así minimizar el impacto en el sistema. Para poder identificar cuáles son las métricas que deben ser recalculadas es que se implementó el concepto de parámetro *dirty*.
Cuando el SLAManager procesa un evento, además de persistirlo en la base de datos, lo persiste en una cache en la cual se guarda el nombre y la fecha en la cual se generó el parámetro.
El recálculo de las métricas inspecciona esta cache de parámetros *dirty* y si hay métricas que dependan de este parámetro las calcula.

5.3.2.8 Base de datos

Como se explicó en los puntos anteriores, se utilizó base de datos relacional para guardar la información global del sistema, los servicios, los parámetros y los eventos del sistema.
Las tablas utilizadas a tales efectos son:

- Event, EventData: Estas tablas son utilizadas para almacenar la información de los eventos generados por el sistema.
- GlobalData: Esta tabla es utilizada para almacenar parámetros de configuración globales del sistema.
- ParameterData: Esta tabla es utilizada para almacenar los datos generados por los eventos de los servicios que sirven para alimentar a las métricas.
- Service: Esta tabla es utilizada para almacenar la información relativa a cada servicio de tercero (tipo de servicio, nombre, URL de acceso, etc.).

5.4 Metodología de desarrollo

La implementación del sistema siguió un modelo iterativo incremental, debido a que por un lado en cada iteración se generaron componentes nuevos del sistema y por el otro se mejoraron los que ya estaban implementados.
Previo a definir las pruebas a realizar sobre el sistema, se estableció que era necesaria la consideración de los siguientes tipos de pruebas:

- Pruebas unitarias: los componentes del sistema fueron probados unitariamente a medida que fueron desarrollados.
- Pruebas funcionales: a medida que se desarrollaron los distintos componentes se validó que éstos cumplieran con las especificaciones funcionales del sistema.
- Pruebas de integración: a medida que se completó el desarrollo de varios componentes se realizaron pruebas de integración que permitieran validar la interacción entre ellos.
- Pruebas de funcionamiento del sistema: se definió un conjunto base de pruebas que consideraran los distintos tipos de políticas y flujos de ejecución existentes el cual fue implementado utilizando el software SoapUI. Estas pruebas permitieron validar el funcionamiento completo del sistema. Dichas pruebas están detalladas en el anexo C2.
- Pruebas de performance y carga: Finalizadas las pruebas del sistema se implementó un conjunto de pruebas de carga, las cuales permitieron medir la escalabilidad del sistema. Dichas pruebas están detalladas en la sección 5.5.3.

5.5 Pruebas del sistema

Dejando de lado las pruebas unitarias, funcionales y de integración, que no actúan sobre el sistema en su conjunto, se realizó dos tipos de pruebas:

- Pruebas de funcionamiento del sistema: focalizada en la validación del funcionamiento general.
- Pruebas de performance y carga: orientada a evaluar el funcionamiento y reacción ante cargas de trabajo.

A continuación se detallan los servicios mock que fueron utilizados, las pruebas realizadas y se muestran los resultados obtenidos luego de su ejecución.

5.5.1 Servicios mock utilizados para las pruebas

Se definió un conjunto base de servicios de terceros que fueran capaces de cubrir los distintos tipos de ejecuciones que puedan ocurrir en el sistema. Para su implementación se definieron servicios *mock* que son simulaciones de los servicios reales.
En la tabla 16 se presenta un resumen de la clasificación de los servicios desarrollados y el tipo de resultado a obtener en la respuesta. Los posibles resultados de la invocación a un servicio son los siguientes:

- Ejecución exitosa
- Falla por *timeout*
- Falla debido a un error en el servidor (código de respuesta 500)

	Ejecución exitosa	Falla por *timeout*	Falla por error en servidor
SIMPLE	8	1	1
ANULABLE	4	0	1
CANCELABLE	2	2	1

Tabla 16: Tipos de servicios mock

Es importante destacar que con el motivo de poder evaluar mejor el funcionamiento del sistema y poder analizar distintos aspectos como las podas o los tiempos de los distintos *overheads* existentes, se decidió que cada servicio de prueba tuviese un tiempo de ejecución fijo.
En el anexo C1 se realiza un listado de los 20 servicios definidos en el sistema, indicando para cada uno el tipo de servicio que es (simple, anulable o cancelable), el tiempo de respuesta del servicio y el tipo de resultado que va a devolver (exitoso, *timeout* o falla).

5.5.2 Pruebas de funcionamiento

Se definió un conjunto de pruebas de evaluación de funcionamiento global del sistema. Dicho conjunto permite validar el funcionamiento del mismo considerando las distintas políticas existentes y los posibles resultados de distintas ejecuciones. En el anexo C2 se muestran en detalle los distintos casos implementados clasificados por la política que utilizan. Además se brinda una descripción del caso, los servicios involucrados y su agrupación en niveles y el resultado esperado de la ejecución.

5.5.3 Pruebas de carga

Para poder evaluar la performance y escalabilidad de nuestro sistema se decidió implementar diversas pruebas de carga. Las mismas buscan:

- Establecer la capacidad de nuestro sistema de reaccionar a múltiples pedidos concurrentes, proviniendo de distintos locales de pago.
- Medir el *overhead* existente y su evolución, a medida que se aumenta la carga en el sistema.

Para todas las pruebas de carga se utilizaron servicios que tuvieran una demora en la respuesta de dos segundos, de manera de poder realizar una mejor evaluación del funcionamiento del sistema. Los detalles correspondientes a las distintas pruebas de carga realizadas se encuentran en el anexo C3.

5.5.4 Resultados obtenidos

A través de los resúmenes de las ejecuciones realizadas sobre las pruebas de carga presentadas en la sección 5.5.3 y detalladas en el anexo C3, se observan a primera vista dos sucesos relevantes:

- Para baja cantidad de hilos la variación promedio es mínima.
- Para alta cantidad de hilos esta aumenta significativamente.

Analizando los distintos casos en baja cantidad de hilos, podemos ver que:

- En el Caso 1 podemos observar que:
  - En 1 hilo: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{131}{1} = 131$ ejecuciones por hilo
  - En 3 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{388}{3} = 129{,}3$ ejecuciones por hilo
  - En 5 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{638}{5} = 127{,}6$ ejecuciones por hilo
- En el Caso 2 podemos observar que:
  - En 1 hilo: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{28}{1} = 28$ ejecuciones por hilo
  - En 3 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{82}{3} = 27{,}3$ ejecuciones por hilo
  - En 5 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{132}{5} = 26{,}4$ ejecuciones por hilo
  - En 10 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{252}{10} = 25{,}3$ ejecuciones por hilo
- En el Caso 3 podemos observar que:
  - En 1 hilo: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{114}{1} = 114$ ejecuciones por hilo
  - En 3 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{322}{3} = 107{,}3$ ejecuciones por hilo
- En el Caso 4 podemos observar que:
  - En 1 hilo: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{26}{1} = 26$ ejecuciones por hilo
  - En 3 hilos: $\frac{\#ejecuciones}{\#de\ hilos} = \frac{73}{3} = 24{,}3$ ejecuciones por hilo

Si se analizan solamente estos datos se podría establecer que nuestro sistema es escalable, debido a que la cantidad de ejecuciones por hilo no disminuye significativamente a medida que se aumenta la cantidad de hilos.

Por otro lado, si tomamos las ejecuciones con mayor cantidad de hilos se puede comprobar a simple vista que no se cumple la regla anteriormente mencionada; por lo que se decidió indagar más en los motivos por los que el sistema reacciona de esa manera.

Se utilizó la herramienta JVisualVM [71][72] para realizar un monitoreo del uso de los distintos recursos del sistema tomando como contexto la ejecución del Caso 4 utilizando 5 hilos de ejecución. Es importante aclarar que en el caso 4 debido a que los pedidos de los clientes son de cinco servicios distintos, por cada pedido van a ser creados hilos de ejecución que permitan la ejecución en paralelo de las tareas del nivel correspondiente; por lo que potencialmente el sistema debe soportar la ejecución de 25 hilos simultáneamente.

Mediante el análisis de un *thread dump* [73] de la ejecución, se constató que la cantidad de hilos de ejecución y la cantidad de hilos de procesos se corresponden a los que teóricamente tendrían que existir, y que por ende nuestro sistema es capaz de escalar correctamente. Sin embargo las demoras se dan debido al hardware utilizado, el cual no es capaz de soportar tanta cantidad de hilos de ejecución simultáneos sin realizar mayores sacrificios de performance.

6 Gestión del proyecto

Esta sección presenta los distintos aspectos relevantes a la gestión del proyecto. En la estimación original, el proyecto estuvo pensado para ser realizado en aproximadamente ocho meses de trabajo. Debido a motivos de diversa índole la duración del mismo se extendió a un año. En la figura 34 se muestra un diagrama Gantt con la planificación inicial del proyecto, mientras que en la figura 35 se muestra la duración real de las distintas fases.

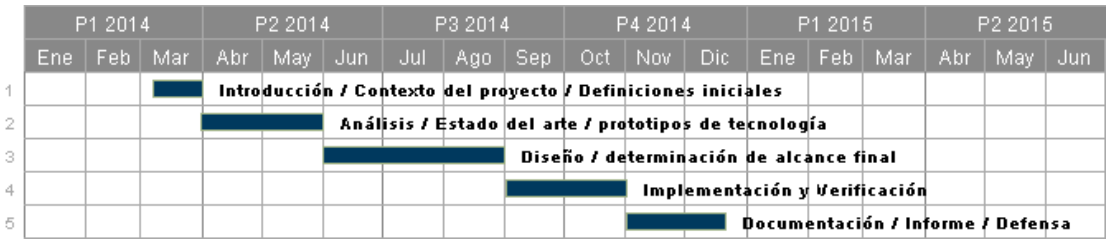

Figura 34: Planificación inicial del proyecto

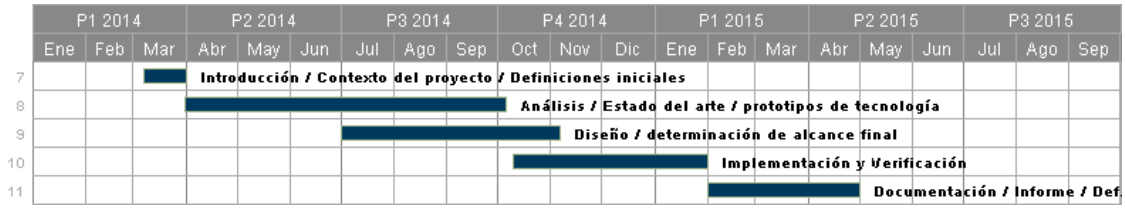

Figura 35: Duración del proyecto

Existieron varias causantes de la desviación en los tiempos del proyecto, dentro de las cuales podemos enumerar a:

- Subestimación de tiempos.
- Profundización en investigación.
- Utilización de tecnologías nuevas, con documentación acotada.
- Licencias.

En el anexo F se encuentra la especificación detallada de todas las fases que componen la elaboración del proyecto.

7 Conclusiones y trabajo futuro

En esta sección se detallan las conclusiones obtenidas y se muestran los posibles trabajos futuros a realizar tomando como base este proyecto.

7.1 Conclusiones

El objetivo del proyecto consistió en diseñar e implementar un *framework* basado en ESB que permita contemplar los distintos *Service Level Agreement* (SLA) existentes en el contexto de una red de locales de pagos y cobranzas, así como también proveer mecanismos de tolerancia a fallos para mantener la consistencia de los datos y garantizar el funcionamiento del negocio ante caídas inesperadas de los servicios

Inicialmente se realizó un análisis de los requerimientos del sistema a desarrollar y se definió el alcance final, el cual fue validado con el cliente. Con esta realidad planteada se hizo un estudio de los servicios involucrados, cómo son invocados y cómo deben formar parte de la composición, dando lugar a: 1) Una clasificación de servicios según el contexto de invocación; 2) Políticas de ejecución de servicios teniendo en cuenta el contexto transaccional de ejecución; y 3) un algoritmo de Scheduler como forma de componer los pedidos a los servicios, ordenarlos y despacharlos para cumplir con los SLA definidos, contemplando la clasificación de servicios y políticas de ejecución previamente definidas.

En el marco de una solución SOA existen diversos estándares y tecnologías que pueden ser aplicadas para la interacción entre servicios ayudando a solucionar y/o mitigar algunos de los problemas planteados en estos ambientes. Dadas las características del proyecto, ahondamos los siguientes: 1) La necesidad de contar con mecanismos que garanticen la consistencia de datos frente a fallos y que involucren diferentes tipos de servicios de diferentes proveedores; 2) Definir un modelo de calidad en el cual poder establecer los factores más relevantes para el proyecto y en el cual poder basar los SLA de los servicios; y 3) Aspectos fundamentales a tener en cuenta en una composición de servicios.

En paralelo al análisis de los requerimientos, se investigó y profundizó sobre cada uno de estos puntos que forman parte del marco conceptual descrito en este documento, llegando a poner en práctica algunos de ellos o generando una serie de recomendaciones para su posterior implementación. En particular en cuanto a la consistencia de datos se describió una serie de recomendaciones a tener en cuenta para la implementación de los servicios. En el caso de manejo de los SLA, se definió un modelo de calidad el cual se incorporó a la implementación de la solución propuesta. En el caso de la composición de servicios, sirvió para entender mejor el contexto en el cual debe ejecutar la solución, descartándose su uso donde a priori parecía aplicable.

Como resultado del proyecto, fue posible cumplir con los objetivos inicialmente planteados. Se diseñó e implementó la solución basada en Switchyard, la cual contempla los requerimientos principales y los objetivos del proyecto destacando como características principales una arquitectura extensible y escalable. A nivel de implementación, se evaluó el uso de Switchyard, Camel y las tecnologías asociadas dando lugar a las siguientes dos conclusiones: 1) Switchyard es una tecnología prometedora, pero que aún le falta maduración. La documentación existente no es suficiente, lo que genera en ocasiones la pérdida de tiempo innecesaria solucionando problemas ocurridos; 2) Camel es una tecnología madura en el mercado, estable y probada que permite desarrollar soluciones utilizando patrones estándares (EIP) y con

capacidad de ruteo para comunicar componentes desarrollados con distintos tipos de tecnologías, mediante la utilización de Camel URI.

Por último, se realizaron pruebas de carga en el sistema dando resultados positivos, donde la principal limitación se debió al hardware utilizado.

7.2 Trabajo Futuro

Se identificó un conjunto importante de mejoras posibles a realizar en el sistema, las cuales no fueron posibles de implementar debido a restricciones de alcance y tiempo del proyecto. A continuación se detallan las más importantes.

7.2.1 Incorporación de un motor de reglas

En sistemas críticos es importante la detección en tiempo y forma de problemas o fallas existentes y la generación de alertas de distinto tipo en tiempo real. Esto permite una mejor gestión de las soluciones de software y con una correcta utilización determina mejoras notables en los tiempos de reacción de los equipos encargados de monitorear los sistemas en que se encuentran en producción.
Como trabajo futuro se propone evaluar la posibilidad de incorporar un motor de reglas de negocio para el procesamiento de los eventos generados en el sistema, sustituyendo al componente CEP implementado. En particular sugerimos la evaluación de JBoss Drools [40], el cual posee soporte en Switchyard [74].

7.2.2 Impacto del uso de Switchyard ESB y comparación con otras plataformas

Es importante analizar el impacto del uso de Switchyard ESB en este tipo de soluciones. Switchyard ESB posee la característica de no solo ser una plataforma ESB, sino que también busca posicionarse como un producto que permite acompañar el desarrollo de las soluciones formando parte del proceso, además de la gestión y el monitoreo de servicios.
Además, consideramos necesaria la realización de un análisis comparativo entre la utilización de Switchyard y los demás ESB disponibles en el mercado. Algunos de los factores que consideramos clave a evaluar son los siguientes:

- Performance y escalabilidad.
- Interoperabilidad.
- Monitoreo en tiempo de ejecución.
- Facilidad de uso.
- Integración con el proceso de desarrollo.

7.2.3 Clusterización

El sistema fue diseñado e implementado para que pueda ser desplegado en un clúster de Switchyard y de esta forma poder escalar. No se llegó a hacer las pruebas necesarias para validar su funcionamiento en clúster. Algunas de las condiciones necesarias para poder instalar el clúster es el de contar con hardware adecuado y un componente que sirva como único punto de entrada exponiendo la interface del sistema y que haga el balanceo a los diferentes nodos del clúster. Parte de las pruebas a futuro son validar que el sistema esté correctamente clusterizado, que todos sus componentes repliquen sus estados adecuadamente y el balanceo de carga sea el correcto. También verificar que el sistema escala al agregar nuevos nodos al

clúster y cuál es el *overhead* que se agrega en esta configuración. Los componentes que se deberían verificar son: cache Infinispan, colas JMS, componentes Switchyard.

7.2.4 Visualización y usabilidad

Actualmente el sistema cuenta con una interfaz que permite:

- gestionar los servicios de terceros existentes
- visualizar los valores de las métricas
- visualizar los eventos generados en el sistema
- visualizar gráficamente la comparación del tiempo efectivo de la ejecución de un plan con el tiempo estimado.

En los requerimientos originales se planteó la posibilidad de implementar un *dashboard* que permitiera una mejor visualización y análisis de las métricas generadas en el sistema y de su evolución histórica. Dicho requerimiento quedó fuera del alcance final del proyecto.
Además, las interfaces implementadas no fueron pensadas para ser utilizadas en ambiente de producción, donde la usabilidad de las mismas y la posibilidad de tener diferentes visualizaciones para evaluar los distintos sucesos ocurridos en el sistema no fueron tomadas en cuenta.
Por ende se propone como trabajo futuro la mejora de las interfaces de manera de que tengan en cuenta mecanismos que permitan analizar mejor la información y ser personalizables según el usuario que esté utilizando el sistema.

7.2.5 Incorporación de métricas

El modelo de calidad presentado en la sección 4.1 contempla la utilización de cinco factores de calidad asociados a la calidad del servicio. Sin embargo el alcance de nuestra solución establece que solo se implementó las métricas tiempo de respuesta (*response time*), accesibilidad (*accessibility*) y nivel de éxito (*successability*).
Se plantea como trabajo futuro la implementación de las dos métricas restantes, rendimiento máximo (*maximum throughput*) y disponibilidad (*availability*) al sistema.

7.2.6 SLA determinado por múltiples métricas

En los requerimientos funcionales se estableció que tanto el SLA existente entre el sistema y los clientes como el SLA existente entre el sistema y los servicios de terceros debían contemplar como factor de cumplimiento al valor de la métrica *Response Time*.
Se plantea como trabajo futuro la posibilidad de establecer contratos de servicios tanto con los sistemas de terceros como con los clientes de la plataforma, que puedan contemplar una o más métricas de las definidas en el sistema.

7.2.7 Reacción del sistema ante fallo en compensación o cancelación

En los requerimientos funcionales y en común acuerdo con el cliente, se estableció que quedaba fuera del alcance actual del sistema la reacción de este ante la ocurrencia de un fallo en la ejecución de una compensación o una cancelación.
Se plantea por ende como trabajo futuro, la definición y posterior implementación de la reacción que debe tener el sistema ante la ocurrencia de un evento de estas características.

7.2.8 Composición dinámica de servicios

Las características de la implementación de este sistema lo hacen suficientemente genérico como para adaptarse a otros tipos de soluciones más allá del contexto de una red de pagos y cobranzas.
Sin embargo, en otros sistemas puede ocurrir que existan múltiples implementaciones de un mismo servicio, las cuales funcionen de manera independiente entre sí. En ese contexto vemos necesaria la incorporación de un motor de decisiones que permita la determinación de qué implementación de un determinado servicio conviene invocar, basándose en características como el tiempo de respuesta o la cantidad de pedidos por unidad de tiempo que éste acepta. Dicho motor de decisiones debe actuar en tiempo real, permitiendo así que los servicios sean resueltos dinámicamente y que incluso pueda ser necesaria la composición de múltiples servicios para la resolución de un pedido.

7.2.9 Evaluación de JBoss Overlord

Como describimos en la sección 5.2.1 se evaluó Overlord Runtime Governance como alternativa para la generación y procesamiento de eventos del sistema para implementar las métricas. Creemos que es un componente muy interesante para hacer una evaluación más a fondo a los efectos de entender hasta qué punto se puede adaptar. Algunas de las características que entendemos que deben de ser analizadas son:

- Cuál es la estructura que debe tener una composición Switchyard para que sus servicios generen los eventos o sean filtrados.
- Generación de métricas. Además de Response Time qué otra métrica puede ser generada.
- Cómo poder generar los eventos por API si no fuera por exponer un servicio Switchyard.
- Si se puede adaptar la estructura del evento y correlacionar eventos pertenecientes a una misma invocación que pase por varios servicios.
- En nuestro sistema tomamos como métrica de *Response Time* calculando la mediana de los datos recogidos durante un período de tiempo y luego hacemos el cálculo periódicamente. Se podría evaluar cómo se podría implementar esta funcionalidad con Overlord.

7.2.10 Monitoreo y generación de alertas

Para un sistema de estas características, nos parece importante contar con herramientas que ayuden a conocer en *runtime* como se está comportando el sistema. Un dato que puede darnos esta información es la diferencia entre el tiempo estimado y el tiempo de ejecución de un plan. Este dato debería de tender a cero y se podrían definir bandas para las cuales generar alertas con diferentes niveles de severidad de acuerdo a cuanto se desvíe del cero. Para esto se podrían definir, en el componente CEP, reglas que permitan hacer esta evaluación y generar eventos del tipo alertas para que sean consumidos por un componente que los persista y/o dispare notificaciones tales como mandar mail.
Entendemos que este dato es importante porque nos puede dar una visión de que el sistema no se adapta correctamente y una de las razones para que esto ocurra es que el sistema se esté degradando porque no puede escalar en cantidad de hilos concurrentes.
Entendemos también que se podría agregar componentes que ayuden en la búsqueda y en la visualización de la información generada por el componente BAM. Una alternativa a evaluar es usar Elasticsearch [75] y Kibana [76].

8 Referencias

[1] Thomas Erl, *SOA: principles of service design*. Prentice Hall, 28-Jul-2007.
[2] Dimitrios Georgakopoulos and Michael P. Papazoglou, *Service-oriented computing*. The MIT Press, 2009.
[3] Laura González, *Plataforma ESB Adaptativa para Sistemas Basados en Servicios*. Tesis de Maestría en Informática, Programa de Desarrollo de las Ciencias Básicas - Universidad de la República, 2011.
[4] Pablo Garbusi, *Consistencia de datos en SOA*. Seminario del Laboratorio de Integración de Sistemas (LINS) – Universidad de la República, 23-dic-2013.
[5] *Transactions Overview Guide*. [Online]. http://docs.jboss.org/jbosstm/5.0.0.Final/guides/transactions_overview_guide/index.html [Accedido: 17/03/2015].
[6] *Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2*. [Online]. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html [Accedido: 17/03/2015].
[7] *Long-Running Transactions in SOA*. [Online]. http://soa.sys-con.com/node/318438 [Accedido: 17/03/2015].
[8] *Eventually Consistent - Revisited*. [Online]. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html [Accedido: 17/03/2015].
[9] *OASIS Web Services Business Process Execution Language (WSBPEL) TC*. [Online]. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel [Accedido: 17/03/2015].
[10] *Writing a BPEL compensation handler*. [Online]. http://ddweerasiri.blogspot.com/2011/09/writing-bpel-compensation-handler.html [Accedido: 17/03/2015].
[11] *CALM: consistency as logical monotonicity*. [Online]. http://www.bloom-lang.net/calm/ [Accedido: 17/03/2015].
[12]*Design Patterns*. [Online]. http://soapatterns.org/design_patterns/overview [Accedido: 17/03/2015].
[13]Nicolai M. Josuttis, *SOA In Practice: The Art of Distributed System Design*. O'Reilly ,2007.
[14]Andreas Goeb and Klaus Lochmann, *A Software Quality Model for SOA.* 4-Sep-2011.
[15] Eunju Kim and Youngkon Lee, *OASIS Quality Model for Web Services.* [Online]. http://www.oasis-open.org/committees/download.php/15910/WSQM-ver-2.0.doc. [Accedido: 17/03/2015].
[16] CITY, CNR, FBK, INRIA, et al. *Quality Reference Model for SBA.* 16-Mar-2008.
[17] CITY, FBK, INRIA, Polimi,et al. *Initial Concepts for Specifying End-to-End Quality Characteristics and Negotiating SLAs.* 16-Jun-2009.
[18] *Quality Attributes (of S-Cube Quality Reference Model)*. [Online]. http://www.s-cube-network.eu/km/qrm [Accedido: 17/03/2015].
[19]*Web Services Quality Factors Version 1.0*. [Online]. http://docs.oasis-open.org/wsqm/WS-Quality-Factors/v1.0/cs01/WS-Quality-Factors-v1.0-cs01.html [Accedido: 17/03/2015].
[20] Yashpalsinh Jadeja, Kirit Modi, and Ankur Goswami, *Context Based Dynamic Web Services Composition Approaches: a Comparative Study*. International Journal of Information and Education Technology, Vol. 2, No. 2, Apr-2012.
[21] Thomas Cottenier and Tzilla Elrad, *Executable Choreography Processes with Aspect-Sensitive Services.* Paper, Computer Science Department - Illinois Institute of Technology, 2005.
[22] *Service Composition*. [Online]. http://www.infoq.com/articles/lublinsky-soa-composition [Accedido: 17/03/2015].

[23] Ravi Khadka and Brahmananda Sapkota, *An Evaluation of Dynamic Web Service Composition Approaches*. 4th International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing - ACT4SOC 2010, Atenas – Grecia, 23-Jul-2010.
[24] *Toward a pattern language for Service-Oriented Architecture and Integration, Part 2: Service composition*. [Online]. http://www.ibm.com/developerworks/webservices/library/ws-soa-soi2 [Accedido: 17/03/2015].
[25] *Web Services Architecture*. [Online]. http://www.w3.org/TR/ws-arch [Accedido: 17/03/2015].
[26] *Web Services Description Language (WSDL) 1.1*. [Online]. http://www.w3.org/TR/wsdl [Accedido: 17/03/2015].
[27] *SOAP (Simple Object Access Protocol) definition*. [Online]. http://searchsoa.techtarget.com/definition/SOAP [Accedido: 17/03/2015].
[28] *Web Services Architecture*. [Online]. http://www.w3.org/TR/2002/WD-ws-arch-20021114 [Accedido: 17/03/2015].
[29] *OASIS Members Form Committee to Advance Web Services Transaction (WS-TX) Standards*. [Online]. https://www.oasis-open.org/news/pr/oasis-members-form-committee-to-advance-web-services-transaction-ws-tx-standards [Accedido: 17/03/2015].
[30] *OASIS Members Form Web Services Transaction (WS-TX) Technical Committee.* [Online]. http://xml.coverpages.org/ni2005-10-03-a.html [Accedido: 17/03/2015].
[31] *Web Services Coordination (WS-Coordination) Version 1.2.* [Online]. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html [Accedido: 17/03/2015].
[32] *Web Services Business Activity(WS-BusinessActivity) Version 1.2.* [Online]. http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html [Accedido: 17/03/2015].
[33] *Long-Running Transactions with WS-BusinessActivity*. [Online]. http://www.servicetechspecs.com/ws-businessactivity [Accedido: 17/03/2015].
[34] Schmidt et al., *The Enterprise Service Bus: Making service-oriented architecture real*. IBM Systems Journal, vol 44, no 4, 2005.
[35] M. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, *Service-Oriented Computing: State of the Art and Research Challenges.* IEEE Computer Society, 2007.
[36] *Service Component Architecture (SCA)*. [Online]. http://www.oasis-opencsa.org/book/export/html/21 [Accedido: 17/03/2015].
[37] *OASIS Service Component Architecture / Assembly (SCA-Assembly) TC.* [Online]. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sca-assembly [Accedido: 17/03/2015].
[38] Gregor Hohpe and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 20-Oct-2003.
[39] *Apache Camel*. [Online]. http://camel.apache.org/ [Accedido: 17/03/2015].
[40] *JBoss Drools*. [Online]. http://www.drools.org/ [Accedido: 17/03/2015].
[41] *Object Management Group - Business Process Model and Notation*. [Online]. http://www.bpmn.org/ [Accedido: 17/03/2015].
[42] *Switchyard*. [Online]. http://switchyard.jboss.org/ [Accedido: 17/03/2015].
[43] *What Is Camel?*. [Online]. http://camel.apache.org/what-is-camel.html [Accedido: 17/03/2015].
[44] Wildfly. [Online]. http://wildfly.org/ [Accedido: 17/03/2015].
[45] *Java Platform, Enterprise Edition (Java EE) 7*. [Online]. http://docs.oracle.com/javaee/7/ [Accedido: 17/03/2015].
[46] *Contexts & Dependecy Injection for Java*. [Online]. http://www.cdi-spec.org/ [Accedido: 17/03/2015].
[47] *Contexts and Dependency Injection for the Java EE platform*. [Online]. http://docs.jboss.org/cdi/spec/1.2/cdi-spec.html [Accedido: 17/03/2015].

[48] *JSR-000318 Enterprise JavaBeansTM 3.1*. [Online]. https://jcp.org/aboutJava/communityprocess/final/jsr318/ [Accedido: 17/03/2015].
[49] *JMS 2.0 Final Release*. [Online]. https://java.net/projects/jms-spec/pages/JMS20FinalRelease [Accedido: 17/03/2015].
[50] *HornetQ*. [Online]. http://hornetq.jboss.org/ [Accedido: 17/03/2015].
[51] *Java Persistence API*. [Online]. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html [Accedido: 17/03/2015].
[52] *Hibernate*. [Online]. http://hibernate.org/ [Accedido: 17/03/2015].
[53] *Infinispan*. [Online]. http://infinispan.org/ [Accedido: 17/03/2015].
[54] *JSR-000344 JavaServerTM Faces 2.2* [Online]. https://www.jcp.org/aboutJava/communityprocess/final/jsr344/index.html [Accedido: 17/03/2015].
[55] *JavaServer Faces Technology*. [Online]. https://docs.oracle.com/javaee/7/tutorial/jsf-intro.htm#BNAPH [Accedido: 17/03/2015].
[56] *Switchyard Service Implementations*. [Online]. https://docs.jboss.org/author/display/SWITCHYARD/Service+Implementations [Accedido: 17/03/2015].
[57] *Richfaces*. [Online]. http://richfaces.jboss.org/ [Accedido: 17/03/2015].
[58] *SoapUI* [Online]. http://www.soapui.org/ [Accedido: 17/03/2015].
[59] *Apache Maven* [Online]. http://maven.apache.org/ [Accedido: 17/03/2015].
[60] *Red Hat JBoss Developer Studio* [Online]. http://www.jboss.org/products/devstudio/overview/ [Accedido: 17/03/2015].
[61] *Apache Subversion* [Online]. http://subversion.apache.org/ [Accedido: 17/03/2015].
[62] *XQuery 1.0: An XML Query Language (Second Edition).* [Online]. http://www.w3.org/TR/xquery/ [Accedido: 17/03/2015].
[63] *SAXON: The XSLT and XQuery Processor*. [Online]. http://saxon.sourceforge.net/ [Accedido: 17/03/2015].
[64] *Apache Camel Http4 Component*. [Online]. http://camel.apache.org/http4.html [Accedido: 17/03/2015].
[65] *Apache HttpComponents*. [Online]. http://hc.apache.org/ [Accedido: 17/03/2015].
[66] *Apache HttpComponents: Interface CoreConnectionPNames.* [Online]. http://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/org/apache/http/params/CoreConnectionPNames.html [Accedido: 17/03/2015].
[67] *Switchyard Throttling Configuration.* [Online]. https://docs.jboss.org/author/display/SWITCHYARD/Throttling [Accedido: 17/03/2015].
[68] *Apache Camel: Quartz Component.* [Online]. http://camel.apache.org/quartz.html [Accedido: 17/03/2015].
[69] *Quartz Job Scheduler.* [Online]. http://www.quartz-scheduler.org/ [Accedido: 17/03/2015].
[70] *CronTrigger Tutorial*. [Online]. http://www.quartz-scheduler.org/documentation/quartz-1.x/tutorials/crontrigger [Accedido: 17/03/2015].
[71] *Jvisualvm*. [Online]. http://docs.oracle.com/javase/8/docs/technotes/tools/windows/jvisualvm.html [Accedido: 17/03/2015].
[72] *VisualVM.* [Online]. http://visualvm.java.net/ [Accedido: 17/03/2015].
[73] *Using Thread Dumps*.[Online]. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/using_threaddumps.html [Accedido: 17/03/2015].
[74] *Switchyard Rules*. [Online]. https://docs.jboss.org/author/display/SWITCHYARD/Rules [Accedido: 17/03/2015].

[75]Elastic [Online]. https://www.elastic.co/products/elasticsearch [Accedido:17/03/2015].
[76] Kibana [Online]. https://www.elastic.co/products/kibana [Accedido: 17/03/2015].

Anexos

Anexo A: Mediciones

En este anexo se detalla la definición y el método de cálculo utilizado para los factores de calidad que fueron tomados de WS-QF e incorporados en el modelo de calidad definido.

Response Time (tiempo de respuesta)

El tiempo de respuesta es el tiempo que transcurre desde que se envía un pedido hasta el momento en el que se recibe una respuesta. Los parámetros que componen este factor son:

- Latencia del Cliente (ClientLatency): es la suma de los tiempos de demora que posee el cliente al procesar y enviar un pedido y al recibir y procesar nuevamente una respuesta.
- Latencia de la red (NetworkLatency): es el tiempo que lleva transmitir el pedido desde el cliente al servidor más el tiempo que se demora en transmitir la respuesta correspondiente.
- Latencia del Servidor (ServerLatency): es el tiempo en el que el servicio demora en recibir el pedido que llegó al servidor, más el tiempo de procesamiento del mismo y el tiempo que le lleva colocar la respuesta en la red.

El tiempo de respuesta se calcula realizando la suma de las distintas latencias involucradas:

$$\text{Response Time} = \text{ClientLatency} + \text{NetworkLatency} + \text{ServerLatency}$$

Para los dos tipos de SLA existente el tiempo de respuesta se calcula de la misma forma, es decir, mediante la suma de las distintas latencias involucradas. Sin embargo, en el caso de la latencia del servidor, sólo es posible mostrar un método de cálculo si tomamos como SLA el existente entre los locales de pago y el sistema central, ya que en los servicios de terceros el método de cálculo es parte inherente de su implementación.

MaximumThroughput (rendimiento máximo)

El rendimiento máximo se define como el máximo número de pedidos que el proveedor puede procesar en un determinado período de tiempo. Los parámetros que componen este factor son:

- #dePedidosResueltosEnElTiempoEstablecido: es el número de pedidos resueltos por el servicio en un periodo de tiempo determinado.
- TiempoEstablecido: tiempo considerado para medir la cantidad de pedidos.

El rendimiento máximo se calcula de la siguiente manera:

$$\text{maximumThroughput} = max\left(\frac{\#\text{dePedidosResueltosEnElTiempoEstablecido}}{\text{TiempoEstablecido}}\right)$$

Availability (disponibilidad)

La disponibilidad es una medida que permite representar el grado en el que un servicio se encuentra disponible y operando. Se define como la proporción de tiempo que un servicio se encuentra operando. Los parámetros que componen a este factor son:

- *DownTime*: refiere al tiempo que el servicio estuvo caído.
- *MeasuredTime*: refiere al período de tiempo que se consideró para la medida.

La disponibilidad se calcula de la siguiente manera:

$$Availability = 1 - \frac{\text{DownTime}}{\text{MeasuredTime}}$$

Accessibility (accesibilidad)

La accesibilidad representa la probabilidad de que el servicio sea accesible estando el sistema activo y disponible. Los parámetros que componen a este factor son:

- *NumberOfAckMessages:* refiere a la cantidad de respuestas exitosas que se recibieron del sistema en un período de tiempo.
- *NumberOfRequestedMessages:* refiere a la cantidad de pedidos enviados.

La disponibilidad se calcula como:

$$\text{Accessibility} = \frac{\text{NumberOfAckMessages}}{\text{NumberOfRequestedMessages}}$$

Successability (nivel de éxito)

El nivel de éxito se define como la probabilidad de devolver una respuesta a un servicio web que fue exitosamente procesado. Los parámetros que componen este factor son:

- NumberOfResponseMessages: es la cantidad de respuestas que llegaron al cliente.
- NumberOfRequestedMessages: es la cantidad de pedidos que el cliente realizó.

El nivel de éxito entonces se define como el cociente entre los parámetros antes mencionados:

$$\text{Successability} = \frac{\text{NumberOfResponseMessages}}{\text{NumberOfRequestedMessages}}$$

Anexo B: Detalles de los componentes del sistema

En este anexo se detallan distintos componentes del sistema y se realiza una formalización de las dependencias existentes entre los mismos.

Anexo B1: Scheduler

El componente Scheduler está dividido en los siguientes cuatro sub-componentes:

- **RequestValidator:** Componente que permite validar el formato y el contenido del pedido realizado por el cliente. Además valida que las características de los servicios que se desea invocar sean coherentes con la política de ejecución seleccionada (ToN, BE, DoP).
- **Planner:** A partir de un pedido validado, elabora un plan de ejecución para la resolución del pedido.
- **SLA Validator:** Permite validar la ejecución teórica de un plan contra el SLA establecido con el cliente. Utiliza los servicios brindados por el componente SLAManager, para determinar la viabilidad de dicha ejecución. El SLAManager provee la funcionalidad de devolver el valor de una medida para un servicio determinado. Dependiendo de la política seleccionada, el componente aplica predicados de poda (BE y DoP) o indica que la ejecución no es viable (ToN).
- **Executor:** Gestiona la ejecución de un plan. Para ello invoca al componente Coordinator tantas veces como niveles de ejecución posea el plan.

A continuación se presenta en formato resumido las dependencias existentes del componente Scheduler y sus sub-componentes para con los demás componentes del sistema.

Componente	Utiliza	Motivo
Scheduler	PolicyManager	Validación de un pedido y generación de un plan basado en la política especificada.
Scheduler	ServiceManager	Obtención de los datos correspondientes a los servicios de terceros especificados en el pedido a procesar.
Scheduler	EventProcessor	Generación de eventos.
Scheduler	SystemService	Obtención de datos globales del sistema (por ejemplo: *timeout* y política por defecto).
Planner	MeasureProvider	Obtención del tiempo de respuesta para cada servicio a invocar.
SLAValidator	MeasureProvider	Obtención del tiempo de respuesta para cada servicio a invocar.
RequestValidator	Planner	Generar plan de ejecución
Planner	SLAValidator	Realizar la validación del SLA
SLAValidator	Executor	Ejecutar el plan
Executor	Coordinator	Ejecución de un nivel del plan

Anexo B2: SLA Manager

El componente SLA Manager está dividido en los siguientes sub-componentes:

- **ParameterEventConsumer:** Componente encargado de consumir los distintos eventos enviados a través del componente EventProcessor y almacenar los datos relevantes de los mismos. Estos eventos contienen los datos de los parámetros.

- **MeasureProvider:** Encargado de hacer efectivo el cálculo de los valores de las métricas (para los distintos períodos del día) correspondientes a los distintos servicios de terceros registrados en el sistema. Cuando el componente Cron determina que es necesario recalcular una cierta métrica, éste invoca los servicios provistos por el componente MeasureProvider. Además, este componente debe brindar dos API diferentes hacia los demás componentes del sistema: una que permita realizar las operaciones básicas (alta, baja, modificación) sobre las medidas consideradas y otra que permita obtener los valores de las distintas medidas consideradas en el modelo de calidad en un determinado período para un servicio dado. Es importante destacar que para poder brindar la última API mencionada utiliza los servicios internos del componente MeasureCache.
- **MeasureCache:** Es un componente interno cuya funcionalidad principal es dar un soporte de caché para el componente MeasureProvider. Esto se debe a que dicho componente efectivamente posee la capacidad de calcular los valores de las medidas pero no de almacenarlos.
- **ParameterService:** Es un componente interno del SLAManager que se encarga de gestionar los datos de los parámetros que son utilizados para el cálculo de las métricas. Internamente brinda dichos servicios a los componentes:
    - El componente MeasureProvider invoca a los servicios de ParameterService para obtener los datos de los parámetros que son utilizados en el cálculo de las métricas.
    - El ParameterEventConsumer lo utiliza para almacenar los datos de los parámetros.
- **Cron:** Posee procesos automáticos que deciden cuándo se deben recalcular los valores de las métricas para los distintos servicios de terceros registrados en el sistema.

A continuación se presenta en formato resumido las dependencias existentes del componente SLAManager y sus sub-componentes para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
MeasureProvider	ServiceManager	Obtención de los detalles de los servicios
MeasureProvider	ParameterService	Obtención de parámetros utilizados en el cálculo de las métricas
MeasureProvider	MeasureCache	Soporte de caché
Cron	MeasureProvider	Cálculo automático de medidas
ParameterEventConsumer	ParameterService	Almacenamiento de datos de los parámetros
ParameterEventConsumer	MeasureProvider	Aviso de nuevos datos de un parámetro

El principal valor de este componente es el de obtener, guardar y proveer un API de consulta sobre el comportamiento de los servicios. Para esto es necesario guardar la información que provee el sistema para poder calcular las medidas de un servicio. Además, vemos necesario tener en cuenta el momento *(TimeStamp)* en el que se produjo el dato para poder correlacionarlo con el momento del día.
Para lograr dicha funcionalidad se necesita la siguiente información:

- Identificador del Servicio.
- Parámetro. El parámetro determina a qué corresponde el dato almacenado. Se consideran cuatro tipos de parámetros posibles:
    - AckMessage. El dato corresponde a que el servicio respondió ya sea error o éxito. El servicio no está fuera de línea.
    - ResponseMessage. El servicio dio una respuesta exitosa.

    - RequestedMessage. El servicio fue invocado o se intentó invocar sin importar el resultado obtenido.
    - ResponseTime. El tiempo que demoró el servicio en responder de forma exitosa.
- Valor
- *timeStamp*

Los parámetros están directamente relacionados al cálculo de las distintas métricas que se necesitan obtener, por ende es necesario guardar para cada métrica existente la lista de parámetros de los que depende.
Está previsto que el sistema sea suficientemente extensible como para agregar una nueva métrica con facilidad, simplemente implementando una interfaz común.

Anexo B3: Coordinator

A continuación presenta en formato resumido las dependencias existentes del componente Coordinator para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
Coordinator	Router	Ejecución de un servicio final
Coordinator	EventProcessor	Generación de eventos

Anexo B4: EventProcessor

A continuación se presenta en formato resumido las dependencias existentes del componente EventProcessor para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
EventProcessor	ParameterEventConsumer	Reenvío de eventos generados en el sistema y creados por el componente EventProcessor
EventProcessor	BAM	Reenvío de eventos generados en el sistema y creados por el componente EventProcessor

Anexo B5: ServicesUI

A continuación se presenta en formato resumido las dependencias existentes del componente ServicesUI para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
ServicesUI	ServiceManager	Alta, baja y modificación de servicios finales registrados en el sistema.

Anexo B6: SLAMonitorUI

A continuación se presenta en formato resumido las dependencias existentes del componente SLAMonitorUI para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
SLAMonitorUI	BAM	Consumo de los eventos procesados

		por el módulo BAM para despliegue en la interfaz gráfica
SLAMonitorUI	MeasureProvider	Visualización del historial de las métricas

Anexo B7: SystemUI

A continuación se presenta en formato resumido las dependencias existentes del componente SystemUI para con los demás componentes del sistema.

Componente	**Utiliza**	**Motivo**
SystemUI	SystemService	Modificación de los distintos parámetros globales del sistema.

Anexo C: Pruebas del sistema

Anexo C1: Servicios Mock

A continuación se enumeran los distintos servicios mock utilizados en las pruebas del sistema y las principales características de los mismos.

ID	Nombre	Tipo (Anulable=A, Cancelable=C, Simple=S)	Tiempo Ejecución (en segundos, NC= no corresponde)	Resultado (Success=S, Timeout=T, Error=E)
1	S_PagoUteService	S	1	S
2	S_PagoOseService	S	2	S
3	S_PagoAntelService	S	3	S
4	S_AgendaTramitesGubService	S	4	S
5	S_AutomovilClubReservaService	S	NC	T
6	S_TiempostEnvioService	S	NC	E
7	A_TombolaService	A	1	S
8	A_GiroAbitabService	A	2	S
9	A_RedUtsCompraService	A	3	S
10	A_DebitoAlfaBrouService	A	4	S
11	A_WesternUnionTransfService	A	NC	E
12	C_CanjeAbisService	C	2	S
13	C_AbiPrestamoService	C	3	S
14	C_CuentaAbitabService	C	NC	T
15	C_PagosDgiService	C	NC	T
16	C_PagoTccService	C	NC	E
17	S_GiroHSBCService	S	2	S
18	S_GiroBBVAService	S	2	S
19	S_GiroABNService	S	2	S
20	S_GiroITAUService	S	2	S

Anexo C2: Tests de funcionamiento del sistema

Este anexo muestra en detalle los casos de prueba implementados para validar el correcto funcionamiento del sistema. Éstos están agrupados de acuerdo a la política seleccionada por el cliente. Además se realiza una descripción del caso de uso, los servicios utilizados (agrupados en niveles) y el resultado esperado de la ejecución de cada caso.
Para identificar mejor el tipo de servicio involucrado, se coloca entre paréntesis la inicial del mismo (A=Anulable, C=Cancelable y S=Simple)

Política Todo o Nada

ID Caso	Descripción	Servicios	Resultado esperado
TON_1	Un servicio simple y el resto anulable. Todos ejecutan correctamente, sin falla o timeout.	Nivel 1: 7(A),8(A),9(A), 10(A) Nivel 2: 1(S)	Ejecución exitosa
TON_2	Unservicio simple y el resto anulable. Todos ejecutan correctamente, excepto el simple, que falla.	Nivel 1: 7(A),8(A),9(A), 10(A) Nivel 2: 6(S)	Ejecución no exitosa. Se ejecutan correctamente los servicios anulables, pero cuando el servicio simple falla, se realiza el rollback de todos los anulables.
TON_3	Un servicio cancelable y el resto anulable. El cancelable falla por timeout. El resto ejecuta exitosamente.	Nivel 1: 7(A),8(A),9(A), 10(A) Nivel 2: 14(C)	Ejecución no exitosa. Se ejecutan correctamente los anulables, pero cuando el cancelable da timeout, se realiza la cancelación del mismo y el rollback de los anulables
TON_4	Un servicio simple y el resto anulable. El timeout del pedido es menor a la suma de la duración del servicio simple y el que demora más de los anulables.	Nivel 1: 8(A),9(A) Nivel 2: 4(S)	No se ejecuta el plan, ya que no se va a cumplir con el SLA establecido.
TON_5	Uno simple y el resto anulable. Falla uno de los anulables.	Nivel 1: 8(A),9(A), 10(A), 11(A) Nivel 2: 4(S)	Ejecución no exitosa. Se ejecuta solamente el primer nivel del plan. Cuando se detecta el fallo del servicio anulable se activa el mecanismo de rollback.
TON_6	Dos servicios simples. No se puede armar un plan válido para esta política.	4(S), 4(S)	No se ejecuta el plan, debido a que falla la validación.

Política BestEffort

ID Caso	Descripción	Servicios	Resultado esperado
BE_1	Todos simples. Ninguno falla.	Nivel 1: 1(S),2(S),3(S),4(S)	Ejecución exitosa.
BE_2	Todos simples. Un simple falla.	Nivel 1: 2(S),3(S),4(S),6(S)	Ejecución exitosa de los servicios 2,3 y 4. Se debe indicar que el servicio 6 falló en la ejecución.
BE_3	Todos simples. Un simple da timeout.	Nivel 1: 2(S),3(S),4(S),5(S)	Ejecución exitosa de los servicios 2,3 y 4. Se debe indicar que el servicio 5 dio timeout.
BE_4	Todos simples. El timeout del pedido es menor a la duración del servicio 4.	Nivel 1: 1(S),2(S),3(S),4(S)	Ejecución exitosa de los servicios 1,2 y 3. El servicio 4 debe aparecer como no ejecutado, ya que el algoritmo de poda no lo debe de haber colocado en el plan.
BE_5	Varios servicios con múltiples invocaciones. Ninguno falla.	Nivel 1: 1(S)x2, 4(S)x3, 9(A)x2	Ejecución exitosa. El tiempo que debería demorar la ejecución debe ser similar a la duración de la tarea más larga (cantidad de invocaciones del servicio por la duración del servicio).
BE_6	Varios servicios con múltiples invocaciones. Ninguno falla. El timeout especificado en la invocación debe ser menor al tiempo total de ejecución de esos servicios.	Nivel 1: S: 1(2); 4(3) A: 9(2)	Por poda, se eliminan los llamados a servicios de las tareas con mayor duración, de forma de cumplir con el timeout establecido. El resto de los servicios se ejecuta exitosamente.

Política Dependency on Priority.

ID Caso	Descripción	Servicios	Resultado esperado
DOP_1	Tres niveles. Distintos tipos de servicios (simples, anulables, cancelables). Se ejecuta correctamente.	Nivel 1: 10(A),13(C) Nivel 2: 8(A),2(S) Nivel 3: 1(S),2(S),7(S)	Ejecución exitosa.
DOP_2	Tres niveles. Distintos tipos de servicios (simples, anulables, cancelables). Falla el servicio 11 en el nivel 2.	Nivel 1: 10(A),13(C) Nivel 2: 11(A),2(S)	Ejecución exitosa del nivel 1 y todo el nivel 2 (excepto el servicio 11). El nivel 3 no se ejecuta.

		Nivel 3: 1(S),2(S),7(S)	
DOP_3	Tres niveles. Distintos tipos de servicios (simples, anulables, cancelables). El timeout del pedido es menor al tiempo estimado de ejecución del plan, de forma que sea mayor al tiempo de ejecución del nivel 1 y menor a la suma del 1 y 2.	Nivel 1: 10(A),13(C) Nivel 2: 9(A),2(S),4(S) Nivel 3: 1(S),2(S)	El nivel 3 y parte del nivel 2 no deben ejecutarse, ya que no estarán en el plan porque van a ser filtrados por la poda. El nivel 1 y el resto del nivel 2 se ejecutan correctamente.

Anexo C3: Pruebas de carga

Los servicios utilizados para las pruebas de carga fueron los siguientes:

- S_GiroHSBCService
- S_GiroBBVAService
- S_GiroABNService
- S_GiroITAUService
- S_PagoOseService

Los detalles y características específicas de cada uno de ellos se pueden encontrar en los anexos A1 y A2.
Las pruebas de carga fueron ejecutadas mediante la utilización de dos computadoras conectadas a través de una red local. Las características de las mismas son las siguientes:

- Computadora 1:
  - Tipo: Notebook
  - Procesador: Intel core i5-4200M, 2.5GHz (2 núcleos reales, 4 virtuales)
  - RAM: 8GB
  - Velocidad de disco: 7200rpm
  - Sistema operativo: Windows 7 64bits
  - Utilización: Sobre esta máquina se montó el servidor Wildfly (versión 8.2.0.Final) con el software desarrollado. Además se modificaron los argumentos de memoria de la JVM con los siguientes valores:
    - -Xms4096M
    - -Xmx4096M
- Computadora 2:
  - Tipo: PC de escritorio
  - Procesador: Intel core 2 duo E8400, 3.GHz (2 núcleos reales)
  - RAM: 8GB
  - Velocidad de disco: 7200rpm
  - Sistema operativo: Windows 7 64bits
  - Utilización: Sobre esta máquina se montó el software Soap UI con los servicios mock utilizados.

A continuación se detallan las pruebas de carga realizadas y los resultados obtenidos en su ejecución.

Caso 1: Una invocación a un único servicio

Este caso contempla la ejecución de pedidos de clientes que solo poseen una invocación a un único servicio. El plan de ejecución generado posee un solo nivel, con una única tarea, que posee una única invocación al servicio.
Se realizaron ejecuciones para uno, tres, cinco, diez y veinte clientes simultáneos y los resultados obtenidos son los siguientes:

# de hilos	Tiempo de ejecución (min)	Cantidad de ejecuciones	Tiempo mínimo (ms)	Tiempo máximo (ms)	Tiempo promedio (ms)
1	5	131	2233	2611	2304
3	5	388	2221	3362	2316
5	5	638	2185	4140	2344
10	5	968	2288	6670	3087
20	5	1424	2980	8199	4186

Caso 2: Múltiples invocaciones a un único servicio

Los pedidos de los clientes están compuestos por cinco invocaciones a un mismo servicio. El plan de ejecución generado al igual que el anterior, posee un solo nivel con una única tarea, con la diferencia de que la tarea posee cinco invocaciones al mismo servicio.
Se realizaron ejecuciones para uno, tres, cinco, diez, veinte y veinticinco clientes simultáneos y los resultados obtenidos son los siguientes:

# de hilos	Tiempo de ejecución (min)	Cantidad de ejecuciones	Tiempo mínimo (ms)	Tiempo máximo (ms)	Tiempo promedio (ms)
1	5	28	10688	11319	10829
3	5	82	10683	14098	11010
5	5	132	10694	14034	11116
10	5	252	10740	22138	11734
20	5	346	12418	34932	16889
25	5	370	12648	27018	19668

Caso 3: Una invocación a múltiples servicios.

Cada pedido de un cliente está compuesto por cinco invocaciones a servicios, con la condición de que las mismas son todas a diferentes servicios. De esta forma el plan generado se compone de un único nivel con cinco tareas, cada una de ellas con una invocación a un servicio.
Se realizaron ejecuciones para uno, tres y cinco clientes simultáneos y los resultados obtenidos son los siguientes:

# de hilos	Tiempo de ejecución (min)	Cantidad de ejecuciones	Tiempo mínimo (ms)	Tiempo máximo (ms)	Tiempo promedio (ms)
1	5	114	2519	3862	2652
3	5	322	2487	6285	2792
5	5	380	2796	7955	3925

Caso 4: Múltiples invocaciones a múltiples servicios.

Cada pedido de un cliente interactúa con cinco servicios de terceros distintos, cinco veces con cada uno (en total, 25 invocaciones a servicios de terceros).
El plan generado posee un único nivel con cinco tareas, y cada tarea posee cinco invocaciones a servicios.
Se realizaron ejecuciones para uno, tres y cinco clientes simultáneos y los resultados obtenidos son los siguientes:

# de hilos	Tiempo de ejecución (min)	Cantidad de ejecuciones	Tiempo mínimo (ms)	Tiempo máximo (ms)	Tiempo promedio (ms)
1	5	26	11527	12143	11708
3	5	73	11601	15811	12147
5	5	79	12045	25080	18704

Anexo D: Configuración del sistema

Los anexos comprendidos en esta sección explican los distintos aspectos de la configuración del sistema.

Anexo D1: Armado de ambiente de desarrollo

Los pasos para armar el ambiente de desarrollo son los siguientes:

1) Descargar e instalar Java 7: http://www.oracle.com/technetwork/es/java/javase/downloads/jdk7-downloads-1880260.html
2) Descargar e instalar Maven 3.x:
    a. Sitio de descarga: http://maven.apache.org/download.cgi
    b. Instalación: http://maven.apache.org/download.cgi#Installation
3) Descargar e instalar JBoss Developer Studio 7.0: https://www.jboss.org/download-manager/file/jboss-devstudio-7.1.1.GA-standalone_universal-standalone_jar.jar
4) Instalar plugin de Eclipse Tooling en JBoss Developer Studio, siguiendo las instrucciones: https://docs.jboss.org/author/display/SWITCHYARD/Installing+Eclipse+Tooling
5) Descargar servidor Wildfly 8.1.0.Final y descomprimir en la carpeta %DEVELOP%: http://download.jboss.org/wildfly/8.1.0.Final/wildfly-8.1.0.Final.zip
6) Descargar Switchyard versión 2.0.Alpha2: http://www.redhat.com/j/elqNow/elqRedir.htm?ref=http://downloads.jboss.org/switchyard/releases/v2.0.Alpha2/switchyard-2.0.0.Alpha2-WildFly.zip
7) Instalar Switchyard siguiendo las instrucciones de la sección “Distribution Zip” https://docs.jboss.org/author/display/SWITCHYARD/Installing+SwitchYard

Anexo D2: Proyectos java

La solución está compuesta por 12 proyectos Maven. A continuación se detalla cada uno de ellos, las funcionalidades que implementan y el tipo de proyecto.

- slaenforcer-parent:
    - Tipo de proyecto: pom.
    - Descripción: proyecto Maven padre. Su función es declarar las dependencias a librerías externas y las versiones utilizadas de las mismas. Además posee los archivos de configuración que se necesitan para que el sistema funcione en Wildfly/Switchyard.
- slaenforcer-ear:
    - Tipo de proyecto: ear
    - Descripción: proyecto Maven cuya función es la de declarar los módulos que serán incluidos en la aplicación y generar el archivo EAR con el empaquetamiento de la aplicación y sus dependencias externas.
- scheduler:
    - Tipo de proyecto: jar (switchyard).
    - Descripción: proyecto Maven que posee la implementación completa del componente Scheduler y todos sus subcomponentes.
- coordinator:
    - Tipo de proyecto: jar (switchyard).

    - Descripción: proyecto Maven que posee la implementación de los componentes Coordinator y Router. Implementa toda la lógica de ruteo, *rollbacks* y cancelaciones.
- core:
    - Tipo de proyecto: jar.
    - Descripción: proyecto Maven que posee la implementación de distintos componentes relacionados a la lógica de negocio, entre los cuales se encuentran:
        - Implementación de los EJB que manejan los distintos componentes que acceden a la base de datos.
        - Implementaciones de las políticas BE y ToN (no se incluye DoP, está implementada en el proyecto custom para mostrar extensibilidad) y del componente PolicyManager.
        - Implementación del componente que gestiona la caché que almacena los datos referentes al contexto de la ejecución de un plan.
- common:
    - Tipo de proyecto: jar.
    - Descripción: proyecto Maven que posee los esquemas (archivos XSD) que modelan el servicio expuesto al cliente, el plan generado por el componente Scheduler y los eventos ocurridos en el sistema. Además posee la configuración de un *plugin* Maven (jaxb2-maven-plugin) que genera código automáticamente a partir de dichos esquemas y el código autogenerado.
- slamanager:
    - Tipo de proyecto: jar (switchyard).
    - Descripción: proyecto Maven que posee la implementación del componente SLAManager y todos sus subcomponentes.
- bam:
    - Tipo: jar.
    - Descripción: proyecto Maven que posee la implementación del componente BAM, el consumo de los eventos de sistema generados y su persistencia en la base de datos.
- cep:
    - Tipo: jar (switchyard).
    - Descripción: proyecto Maven que posee la implementación del componente CEP, el cual procesa los eventos generados por el sistema y los rutea a donde corresponda. Además genera nuevos eventos a partir de los procesados.
- slaenforcer-ui:
    - Tipo: war.
    - Descripción: proyecto Maven que posee la implementación de las distintas interfaces gráficas del sistema.
- custom:
    - Tipo: jar (switchyard)
    - Descripción: Proyecto que posee la implementación de la política DoP. El mismo es utilizado para extender el sistema.
- mock-tests:
    - Tipo: pom.
    - Descripción: proyecto Maven que posee la especificación de todos los servicios de prueba que simulan servicios de terceros implementados, junto a sus correspondientes transformaciones y esquemas.

Anexo D3: Compilación y *deploy*

Previo a la compilación y *deploy* del sistema es necesario realizar los siguientes pasos la primera vez que se desee levantar el sistema:

1) Sustituir el archivo:
   %DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\configuration\standalone-full-ha.xml
   Por el archivo:
   slaenforcer-parent/src/main/resources/install/standalone-full-ha.xml
2) Sustituir el archivo:
   %DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\bin\standalone.conf.bat
   Por el archivo:
   slaenforcer-parent/src/main/resources/install/standalone.conf.bat
3) Copiar el archivo:
   slaenforcer-parent/src/main/resources/install/SlaEnforcer.properties
   Y pegarlo en:
   %DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\data\slaenforcer

Para poder compilar y ejecutar el sistema, es necesario seguir los siguientes pasos:

1) Compilar los proyectos ejecutando el comando (desde línea de comandos):
   mvn clean package
   Tener en cuenta que para ejecutar dicho comando es necesario estar posicionado en el proyecto **slaenforcer-parent.**
   NOTA: en el caso de que además se desee que se regeneren las clases autogeneradas, agregar al comando anterior la opción *-Pgenerate*
2) Copiar el archivo EAR generado (slaenforcer.ear) del directorio **target** del proyecto **slaenforcer-ear** y pegarlo en el directorio **standalone/deployments** de la instalación del servidor Wildfly.
3) Levantar el servidor Wildfly ejecutando el siguiente comando desde la carpeta **bin** dentro de la instalación:
   standalone.bat –c standalone-full-ha.xml –b {direccion_ip_local}

Anexo D4: Configuración de datasource

La configuración del *datasource* utilizado se realiza agregando en el archivo:
%DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\configuration\standalone-full-ha.xml
Bajo la sección **datasources**, del subsistema **datasources** el siguiente fragmento XML:

```
<datasource jndi-name="java:jboss/datasources/SLAEnforcerDS" pool-
name="SLAEnforcer" enabled="true">
<connection-
url>jdbc:h2:file:${jboss.server.data.dir}/h2/slaenforcer;mvcc=true</connectio
n-url>
<driver>h2</driver>
<pool>
      <min-pool-size>20</min-pool-size>
      <max-pool-size>50</max-pool-size>
      <prefill>true</prefill>
</pool>
<security>
<user-name>slaenforcer</user-name>
<password>slaenforcer</password>
</security>
</datasource>
```

Es importante destacar que el archivo **standalone-full-ha.xml** entregado ya posee esta configuración.

Anexo D5: Configuración de colas

Las colas utilizadas por los distintos componentes del sistema se configuran agregando en el archivo:
%DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\configuration\standalone-full-ha.xml
Bajo la sección **hornetq-server/jms-destinations** dentro del subsistema **messaging** el siguiente fragmento XML:

```
<jms-queue name="TasksProcessingQueue">
<entry name="java:/jms/TasksProcessingQueue"/>
<durable>true</durable>
</jms-queue>
<jms-queue name="CoordinatorResponseQueue">
<entry name="java:/jms/CoordinatorResponseQueue"/>
<durable>true</durable>
</jms-queue>
<jms-queue name="EventQueue">
<entry name="java:/jms/EventQueue"/>
<durable>true</durable>
</jms-queue>
<jms-queue name="SystemEventQueue">
<entry name="java:/jms/SystemEventQueue"/>
<durable>true</durable>
</jms-queue>
<jms-queue name="SLAEventQueue">
<entry name="java:/jms/SLAEventQueue"/>
<durable>true</durable>
</jms-queue>
```

Anexo D6: Configuración de cache

La configuración de la caché Infinispan utilizada en el sistema se realiza agregando en el archivo:
%DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\configuration\standalone-full-ha.xml
Bajo el subsistema infinispan el siguiente fragmento XML:

```
<cache-container name="slaenforcer" default-cache="default" jndi-
name="java:/jboss/infinispan/container/slaenforcer" start="EAGER"
aliases="slaenforcer">
<transport lock-timeout="60000"/>
<replicated-cache name="default" batching="true" mode="SYNC">
<locking isolation="REPEATABLE_READ"/>
</replicated-cache>
<replicated-cache name="measureCache" batching="true" mode="SYNC">
<locking isolation="REPEATABLE_READ"/>
<file-store purge="false"/>
</replicated-cache>
</cache-container>
```

Anexo D7: Properties del sistema

El archivo:
%DEVELOP%\wildfly-8.1.0.Final-proyecto\standalone\data\slaenforcer\SlaEnforcer.properties
Posee la configuración de las distintas *properties* del sistema.
A continuación se detalla el uso que se le da a cada una de ellas y cómo influyen en el comportamiento del sistema:

- **slaenforcer.meassure.cache.data.notRecalculateOn**: Tiempo en milisegundos durante el cual si una cierta métrica fue recalculada, ésta no se vuelve a calcular a pesar de que tenga parámetros *dirty*.
- **slaenforcer.meassure.cache.dirty.cleanAfter**: Tiempo en milisegundos que indica el máximo tiempo que puede permanecer un parámetro marcado como *dirty* sin que éste haya sido utilizado para los recálculos. En caso de que transcurra dicho tiempo, ese parámetro es borrado de la caché.
- **slaenforcer.defaultTotalOvehead**: Tiempo en milisegundos asignado como *overhead* de la ejecución de un plan en los casos en que no existe la métrica *overhead* para el período de tiempo seleccionado.

Anexo D8: Modificación de intervalo de ejecución del Cron

Actualmente, para modificar el intervalo de tiempo tras el cual se ejecuta el componente Cron es necesario modificar la *cronExpression* especificada en el servicio Switchyard **MeasureCron** que se encuentra dentro del archivo *switchyard.xml* dentro del proyecto *slamanager*.

Anexo D9: Configuración de Hibernate

A continuación se detalla la configuración de las distintas *properties* de Hibernate utilizadas:

```
<propertyname="hibernate.cache.use_second_level_cache"value="true"/>
<propertyname="hibernate.cache.use_query_cache"value="true"/>
<propertyname="hibernate.cache.region.factory_class"value="org.jboss.as.
jpa.hibernate4.infinispan.InfinispanRegionFactory"/>
<propertyname="hibernate.cache.infinispan.cachemanager"value="java:jboss
/infinispan/container/hibernate"/>
<propertyname="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
```

Anexo E: Extensión del sistema

Los anexos siguientes refieren a los mecanismos existentes e implementaciones necesarias para extender distintas funcionalidades del sistema.

Anexo E1: Agregado de nueva política

Para agregar una nueva política de ejecución es necesario agregar en el enumerado *uy.edu.fing.slaenforcer.common.invokeservice.request.PolicyType* un nuevo valor dado que el valor del enumerado es el que luego se usará como nombre en la invocación al sistema. Como este enumerado es generado a partir de los *schemas* que definen algunas de las estructuras del sistema, hay que modificar el archivo InvokeServiceData.xsd y correr Maven con el perfil *generate* para volverlas a generar. La implementación de la nueva política debe implementar al interface *uy.edu.fing.slaenforcer.core.policy.api.Policy* cumpliendo la condición de ser *thread safe*. Para que la nueva política pueda ser encontrada por el sistema tiene que estar anotada con el qualifier uy.edu.fing.slaenforcer.core.policy.api.PolicyQualifier.
A modo de ejemplo se muestra cómo se implementa la Policy DoP:

```
@PolicyQualifier("DEPENDENCY_ON_PRIORITY")
publicclass DependencyOnPriorityPolicy implements Policy {
…
	public String getType() {
		return PolicyType.DEPENDENCY_ON_PRIORITY.value();
	}
}
```

Anexo E2: Agregado de nuevos parámetros para el cálculo de métricas

Para agregar un nuevo parámetro se debe agregar un nuevo valor en el enumerado *uy.edu.fing.slaenforcer.slamanager.dao.entity.ParameterType*.

Anexo E3: Agregado de nuevos servicios de terceros

Para agregar un nuevo servicio el sistema provee, en su interfaz gráfica, un formulario para dar de alta, modificar o dar de baja un servicio. Los campos del formulario *web* son los siguientes:

- Nombre: Identificador del servicio en el *request* de los pedidos que llegan a través del *Web Service* expuesto por el sistema.
- Endpoint: URL en la cual está expuesta el servicio de terceros que se desea invocar.
- Response Time: Tiempo de respuesta teórico para el servicio.
- Service Type: Tipo de servicio (SIMPLE / NULLABLE / CANCELABLE).
- Service Request Transform: URI en la cual está expuesta la transformación del *request* del servicio.
- Service Response Transform:URI en la cual está expuesta la transformación del *response* del servicio.
- Nullable Service Request Transform:URI en la cual está expuesta la transformación del *request* del servicio de anulación.
- Nullable Service Response Transform: URI en la cual está expuesta la transformación del *response* del servicio de anulación.
- Nullable Endpoint: URL en la cual está expuesta el servicio de anulación correspondiente al servicio de terceros que se desea invocar.
- Nullable Response Time: Timeout utilizado para el servicio de anulación.

- Cancel Service Request Transform: URI en la cual está expuesta la transformación del *request* del servicio de cancelación.
- Cancel Service Response Transform: URI en la cual está expuesta la transformación del *response* del servicio decancelación.
- Cancel Response Time:Timeout utilizado para el servicio de cancelación.
- Cancel Endpoint:URL en la cual está expuesta el servicio de cancelación correspondiente al servicio de terceros que se desea invocar.

Anexo E4: Cómo agregar nuevas métricas

Para agregar una nueva métrica en el sistema es necesario agregar en el enumerado *uy.edu.fing.slaenforcer.core.sla.api.MeasureType* un nuevo valor.
La implementación de la nueva métrica debe implementar la interface *uy.edu.fing.slaenforcer.slamanager.measure.api.MeasureManager* cumpliendo la condición de ser *thread safe*. Para que la nueva medida pueda ser encontrada por el sistema tiene que estar anotada con el *qualifieruy.edu.fing.slaenforcer.slamanager.measure.api.MeasureQualifier*.
A modo de ejemplo se muestra cómo se implementa la medida *Response Time*:

```
@MeasureQualifier("RESPONSE_TIME")
publicclass ResponseTimeMeasureManager implements MeasureManager {

…

	@Override
	public MeasureType getType() {
		return MeasureType.RESPONSE_TIME;
	}
}
```

Anexo F: Gestión detallada del proyecto

En este anexo se muestran los pormenores de la gestión del proyecto, indicando cada una de las actividades realizadas. El nivel de detalle es el resultado de contar con actas de cada una de las reuniones mantenidas con el docente. Para poder comprender mejor la evolución de la gestión del proyecto, los distintos hechos fueron agrupados de acuerdo al mes en el que ocurrieron.

Fecha de inicio: 07/03/2014
Fecha de fin: 31/03/2015

Marzo-Abril / 2014:
A partir de la descripción inicial del proyecto se decidió interiorizarse en la plataforma ESB Switchyard y en poder evaluar la factibilidad de que el mismo ejecutara sobre un servidor Wildfly versión 8.1.0.Final. La versión de Switchyard utilizada para realizar pruebas de ambiente y de tecnologías fue la 2.0.0.Alpha2 (última disponible en ese momento).
Además se definió el ambiente de desarrollo y las principales herramientas a utilizar:

- JBoss Developer Studio (versión 7.0)
- Maven 3.x
- java 1.7
- SVN como repositorio para control de cambios.
- Se configuró los distintos repositorios Maven a utilizar.

El 26 marzo se llevó a cabo la reunión inicial con el cliente, donde éste realizó el *kickoff* inicial del proyecto. El cliente mostró la realidad del problema y por qué se quería utilizar Switchyard como plataforma ESB. Esto nos permitió relevar y generar un primer esbozo de los requerimientos del sistema. Además, se estableció el siguiente cronograma inicial en común acuerdo con el docente:

- Abril, Mayo: Estudio del estado del arte en ESB, SLAs en composición de servicios y transacciones de larga duración. Definición de requerimientos del *framework* y definición de alcance inicial. Prototipos de la tecnología JBoss Switchyard.
- Junio, Agosto: Diseño, definición de casos de uso relevantes para la arquitectura y realización de prototipos para validar la arquitectura. Evaluación de avance. Revisión de alcance y definición de alcance final.
- Setiembre, Octubre: Implementación de funcionalidades definidas en el alcance.
- Noviembre, Diciembre: Documentación, informe final y defensa del proyecto.

Mayo / 2014:
A comienzos de mayo se presentó los avances en investigación obtenidos hasta el momento y se planteó las dudas existentes sobre los requerimientos relevados.Se estableció además una primera entrega intermedia a realizar que incluyera la clasificación que tendrían los servicios, una descripción de los distintos SLA asociados y las políticas de ejecución del sistema. La misma fue entregada a mediados del mes de mayo.

Junio / 2014:
A comienzos del mes de junio se realizó una segunda entrega intermedia correspondiente a la etapa uno del proyecto, que incluyó distintos aspectos del marco teórico del proyecto:

- SLA en composiciones de servicios
- Transacciones
- Composiciones de servicios
- Composiciones dinámicas

- Transacciones en *Web Services*
- ACID2

A fines de junio se decidió reestructurar el cronograma, planteando como objetivos inmediatos la finalización de la investigación sobre transacciones, la evaluación de WS-Coordination como protocolo a utilizar para gestionar transacciones de larga duración y la definición inicial del modelo de calidad del sistema.

Julio / 2014:

El 7/7 se entregó un documento conteniendo la definición de nuestro modelo de calidad.
El 15/ 7 se realizó una nueva reunión con el docente, en la que se evaluó el modelo de calidad propuesto y se estableció las prioridades sobre temas pendientes y mejoras a realizar sobre los documentos ya entregados:

- Realizar mejoras en la justificación del modelo de calidad elegido.
- Realizar documento de alcance del proyecto y requerimientos.
- Diseño
- Prototipo
- Evaluar si el factor de calidad *Transaction Integrity*se incluiría como métrica.
- Ver otros que ayuden a definido en qué orden correr los servicios.
- Definir Factores, sub-factores y métricas utilizados en nuestro modelo de calidad.
- Cerrar documento de transacciones.

El 30/07 se estableció un cronograma de entregas para el mes de agosto, consistente en:

- Alcance del proyecto y requerimientos para presentar al cliente.
- Documento ajustado de Transacciones, Modelo de calidad y Composición de servicios.
- Definir algoritmo de Scheduler (primera aproximación).

Agosto / 2014:

El 22/8 se realizó una nueva reunión, donde se entregó los documentos de modelo de calidad final y una primera versión del documento de requerimientos y alcance.
El 27/8 se entregó una versión mejorada del documento de requerimientos y alcance y se estableció un cronograma para finalizar las tareas pendientes previo a la fecha de la presentación intermedia del proyecto:

- 27/08:
    - Definición y entrega de documento de requerimientos.
- 31/08:
    - Revisión general de los documentos.
    - Finalizar modelo de calidad, agregando una evaluación de la dependencia de los distintos factores de calidad entre sí.
    - Finalizar documento de transacciones.
    - Finalizar documento de SLA en composición de servicios y en servicios dinámicos.
- 07/09:
    - Definición de algoritmo de Scheduler (flujos).
    - Evaluación de Switchyard. Viabilidad.
    - Diseño de la solución y servicio de cara al cliente.
- 14/09:
    - Implementación del prototipo.
    - Preparación de la presentación.

El 31/08 se cumplió con lo establecido, entregándose versiones finales o corregidas de los documentos:

- Algoritmo de Scheduler
- Calidad en SOA y modelode calidad

- Clasificación de servicios
- Descripción del proyecto
- Requerimientos y alcance
- SLA en composición de servicios
- Transacciones en *Web Services*

Setiembre / 2014:

El 10/09 se realizó una nueva reunión con el motivo de:

- Mostrar avance sobre el diseño del sistema y evacuar dudas sobre el mismo.
- Elaborar la presentación intermedia.

El 23/09 se realizó una presentación de diseño y arquitectura del sistema al cliente. En la misma se plantearon las siguientes mejoras:

- Realizar una descripción de más bajo nivel del componente SLAManager.
- Detallar en qué partes del proceso se van a disparar eventos.
- Evaluar cambios propuestos en el diagrama EIP (ejecución de un nivel).
- Documentar las decisiones de arquitectura tomadas como, uso de mensajería y cache distribuida para soportar clusterización y HA (Escalabilidad). También la extensibilidad por el uso del PolicyManager.
- Implementar prototipo de algunos componentes que representen la implementación de algunos patrones de EIP para evaluar mejor Switchyard y su funcionamiento.
- Realizar mejorar el diagrama de capas.

En esta reunión se decidió comenzar a realizar prototipos para evaluar viabilidad de arquitectura y tecnologías propuestas.

Octubre / 2014:

El 4/10 se realizó otra entrega la cual consistió en la mejora de los documentos antes presentados incorporando el *feedback* obtenido en la reunión con el cliente. Los documentos modificados y reentrenados fueron:

- Algoritmo de Scheduler
- Arquitectura
- Calidad en SOA y modelo de calidad
- SLA manager
- SLA en composición de servicios

Además, se realizó la entrega de la primera versión de la presentación intermedia del proyecto.

El 06/10 se realizó una nueva reunión con el motivo de evaluar los puntos entregados. Los documentos tenían las mejoras planteadas por el cliente en la reunión del 23. Además se planteó el cronograma para el resto del proyecto y se definió iteraciones cada 15 días para mostrar avances en implementación del proyecto.

El 10/10 comienza laimplementación del sistema, realizándose prototipos en Switchyard de:

- Algoritmo de Scheduler
- Evaluación de Camel para implementar patrones de EIP. Inicios del Coordinador.

El 14/10 se realizó la presentación intermedia del proyecto. Luego de finalizada la misma se planteó el siguiente conjunto de mejoras/sugerencias:

- La posibilidad de contar con una nueva política que considerara varios niveles, y de esa forma mostrar mejor la generalidad de la ejecución de un plan. Se planteó establecer prioridades en los servicios para determinar los niveles del plan.
- Tomar la definición de referencia de Composición Dinámica de Servicios y mostrar cómo el software que estamos implementando se aleja de dicha definición, por lo que no aplica directamente el estado del arte en el tema.
- Definir formalmente el SLA del sistema.

- Definir el umbral de uso de los datos para los atributos de calidad.
- Formalizar el uso de los EIP, indicando los datos de entrada/salida para cada patrón utilizado, estrategias de correlación y liberación de datos.
- Realizar mejoras en la notación eliminando los componentes *Wiretap* y revisar uso de itinerario o composición de *Splitters*.
- Agregar métrica al modelo de calidad y granularidad de la métrica.

Noviembre / 2014:
El 5/11 se realizó una nueva reunión con el tutor que permitió mostrar los avances y las modificaciones realizadas luego de la presentación. Además se mostró la implementación de los componentes Scheduler y Coordinator funcionando por separado.
Estado de la implementación al 25/11:

- Componente *Scheduler* y las tres políticas (ToN, BE y DoP) implementados.
- Componente *Coordinator* con implementación avanzada. utilizando Switchyard y Camel. Se evaluó diferentes alternativas como ser la utilización del patrón *RoutingSlip* y la implementación de una invocación http dinámica.
- La configuración de los servicios y configuración global son tomadas desde la base de datos y se configuraron las entidades para que utilicen cache de segundo nivel de Hibernate.
- Se utiliza Infinispan para mantener el estado de la ejecución de un plan y saber cuándo detenerla en caso de falla.
- En el componente *Coordinator*se resolvió utilizar XQuery para las transformaciones de*request* y *response* (se descartó XSLT debido a la existencia de un bug generado a partir de la incompatibilidad en la versión de dicho componente Switchyard con la brindada por Wildfly). Un dato no menor es que las URI de las transformaciones también son generadas dinámicamente y se utiliza *File System*para alojar las transformaciones.
- Se implementaron varios servicios *mock* para simular los servicios de terceros.

Se mantiene como pendiente la generación y el procesamiento de eventos y la implementación del SLAManager.

Diciembre / 2014:
El 9/12 fue presentada una primera versión del sistema completo integrado funcionando. Se mostró la ejecución de casos de uso simples utilizando políticas BE y ToN, sin contar con compensaciones. Además, se pudo mostrar la creación y la ejecución de un plan, y los tiempos consumidos por los diferentes componentes del sistema.Dentro de la lista de pendientes se redefinió el alcance y se establecieron prioridades en la misma, resultando:

- Dentro del alcance:
    - *Fault Handlers* y mecanismos para ejecutar las compensaciones en caso que corresponda. Cancelación. (prioridad 1)
    - Implementación de SLAManager. (prioridad 2)
    - UI para visualización del estado del sistema. (prioridad 3)
    - Componente de cálculo del *overhead*. (prioridad 4)
- Fuera del alcance:
    - Implementación de CEP y evaluar alternativas a Drools.
    - UI de ingreso de servicios y configuración del sistema.

Restaba además la realización de pruebas de carga en el sistema.
Avance al 12/01 presentado al tutor:
De lo establecido en la reunión del 9/12, se avanzó con:

- *Fault Handlers* y mecanismos para ejecutar las compensaciones en caso que corresponda.

- Cancelación. Tomamos como caso de cancelación el *timeout* de un servicio y en ese caso ejecutamos la cancelación.
- Implementación de SLAManager.
- UI de ingreso de servicios, configuración del sistema y visualización de la cache con los datos de las métricas. (Se decidió incluirla en el alcance, ya que brindaba un plus al sistema).
- Servicios de prueba

Se mantuvo como pendiente:

- UI para visualización del estado del sistema.
- Componente de cálculo del *overhead*.
- hacer pruebas de carga.

El componente CEP se mantuvo fuera del alcance.

Enero / 2015:

El día 22/01 se realizó una última reunión en la etapa de implementación con el docente para realizar una muestra del sistema funcionando completamente. Se descartó la posibilidad de utilizar Drools en el componente CEP pero de todas formas de implementó una versión que permite el procesamiento de los eventos del sistema, de manera manual. Se mostró la UI de ingreso de servicios y configuración del sistema y permaneció pendiente la realización de pruebas de carga. Además se definió la estructura de la tesis.

Se dio por finalizada la etapa de implementación el día 31/01, utilizándose los días previos para la implementación de las pruebas de carga y la corrección de errores detectados en el sistema.

Febrero – Marzo – Abril / 2015:

Se dedicó los meses de febrero, marzo y abril a generar la documentación restante del proyecto y a la elaboración de la presentación del mismo.