Instituto de Computación - Facultad de Ingeniería

Universidad de la República

Montevideo, Uruguay

Informe de proyecto de grado

Gestión y Adaptación al Contexto en Plataformas de Integración

Matías Cabrera

Pablo Pirotto

Vanessa Díaz

Docente Supervisor: Ing. Laura González

Docente Supervisor Alterno: Ing. Sebastián Vergara

Julio, 2015

Resumen

La demanda de sistemas de *software* sensible al contexto ha crecido en los últimos años. Este tipo de sistemas aprovechan información del entorno del usuario (localización, idioma, etc.) para anticiparse a sus necesidades y ofrecerles funcionalidades de forma más personalizada.

Por otro lado, los servicios web son una de las tecnologías más utilizadas para implementar Arquitecturas Orientadas a Servicios (SOA). Este tipo de arquitecturas facilita la construcción y diseño de sistemas distribuidos a gran escala. En este marco, los *Enterprise Service Bus* (ESB) son una de las plataformas preferidas para dar soporte a la implementación de una SOA. A su vez, los ESBs están siendo cada vez más empleados junto con tecnologías de *Complex Event Processing* (CEP) para brindar soporte a escenarios donde los eventos tienen un papel primordial.

En este proyecto se realizó la implementación de una plataforma de integración basada en un ESB con un motor CEP que permite construir servicios web sensibles al contexto. El motor CEP utiliza sus capacidades de análisis y procesamiento para detectar situaciones en las que se encuentra el usuario. Ante estas situaciones, el ESB aplica sus capacidades de mediación para adaptar las invocaciones y respuestas de los servicios para hacerlas sensibles al contexto.

En primer lugar, se analizaron varios productos ESB existentes en el mercado, con el fin de seleccionar el producto que mejor se adecuara a las necesidades del proyecto, decidiendo emplear SwitchYard. Como motor CEP se optó por utilizar Drools Fusion.

La plataforma fue diseñada tomando en cuenta las características de estos dos productos. Para la implementación se tuvieron en cuenta requerimientos como la configuración en tiempo de ejecución y la simplicidad de dicha configuración. Por este motivo, se desarrolló una consola de administración que permite configurar cada aspecto necesario para la generación de servicios sensibles al contexto.

En la etapa final del proyecto se definieron casos de estudio para validar el correcto funcionamiento de la plataforma y demostrar que la misma contribuye a la generación de servicios web más personalizados y precisos.

Palabras claves: servicios web, arquitectura orientada a servicios*, enterprise service bus*, procesamiento de eventos complejos, sensibilidad al contexto, adaptación

Contenido

1 Introducción

En este proyecto se realizó una implementación a la solución conceptual planteada en el artículo *"An ESB-Based Infrastructure for Event-Driven Context-Aware Web Services"* [1]. Dicha solución tiene como objetivo generar servicios web sensibles al contexto basándose en las capacidades de mediación de los Enterprise Service Bus (ESB) y las prestaciones de Complex Event Processing (CEP). En este capítulo se presenta el contexto y motivación, objetivos y aportes del proyecto para finalmente mostrar la organización del resto del documento.

1.1 Contexto y Motivación

El software sensible al contexto ha crecido en popularidad, especialmente con el aumento en la cantidad de aplicaciones móviles. Este tipo de soluciones sensibles al contexto responde a la necesidad de mejorar la experiencia del usuario, con datos más precisos y personalizados. Si bien el contexto parece estar fuertemente vinculado a aplicaciones móviles, la creación de *software* sensible al contexto es demandado también por otro tipo de aplicaciones, por ejemplo, aplicaciones de escritorio.

En [2] se destacan tres aspectos importantes del contexto del usuario: dónde se encuentra, con quién se encuentra y qué recursos tiene cerca. Por lo tanto el contexto no solo engloba la posición del usuario, sino también su posición relativa a objetos de interés que también son móviles y cambiantes. Incluye elementos como la iluminación, nivel de ruido, conectividad, costos de comunicación, ancho de banda o posiciones y características de otros usuarios.

Por otro lado, las Arquitecturas Orientadas a Servicios (SOA) emergieron como una solución eficiente para la construcción y diseño de sistemas distribuidos a gran escala. En este enfoque de componentes reutilizables y compartidos, los servicios web aparecen como una de las tecnologías clave, con estándares como Simple Object Access Protocol (SOAP) y Web Service Description Language (WSDL).

Si bien hay excelentes herramientas y *frameworks* para el desarrollo de servicios web, todavía no existen herramientas que permitan el desarrollo de servicios web sensibles al contexto o agregar sensibilidad al contexto a servicios web existentes [1]. Ésta es un área que aún se encuentra bajo investigación.

En particular en [1], se propone una solución conceptual para generar servicios sensibles al contexto. Dicha propuesta plantea utilizar las capacidades de mediación de los ESB y la capacidad de procesamiento que brindan los motores CEP, para detectar situaciones y adaptar la invocación de un servicio de acuerdo a las mismas.

En el esquema de la Figura 1 se muestra una representación de la solución planteada en [1], en la que se observan los componentes principales involucrados en la generación de servicios web sensibles al contexto.

Figura 1 – Esquema de la solución planteada en [1].

La solución propuesta en [1], consiste en construir y exponer servicios sensibles al contexto a través del ESB, partiendo de servicios que pueden no ser sensibles al contexto. Para poder construirlos, el ESB aplica lógica de adaptación utilizando sus capacidades de mediación. Las adaptaciones son aplicadas automáticamente dependiendo de la situación en que se encuentre el usuario que invoca al servicio. Estas situaciones son detectadas en base a la información que llega desde distintas fuentes gracias a las capacidades de CEP. Sin embargo, esta solución está planteada a nivel conceptual y al momento no existe una implementación de la misma.

1.2 Objetivos

El objetivo general de este proyecto es realizar una implementación de la plataforma de integración propuesta en [1] basada en un ESB y un motor CEP específicos, que permita detectar el contexto de ejecución de una invocación a un servicio y adaptarla de acuerdo a dicho contexto.

Para cumplir con el objetivo general se plantean los siguientes objetivos específicos:

- Evaluar y elegir un producto ESB que se adecue a la solución propuesta, analizando distintos productos ESB que existen en el mercado.
- Diseñar una solución y arquitectura basada en los lineamientos propuestos en [1], las prestaciones del ESB seleccionado y el motor CEP sugerido.
- Implementar un prototipo de la solución propuesta utilizando como base la arquitectura diseñada.
- Probar la implementación del prototipo en al menos un caso de estudio concreto que demuestre la factibilidad y el correcto funcionamiento del prototipo.

1.3 Aportes del Proyecto

Los aportes del proyecto son los siguientes:

- Análisis de distintas plataformas ESB de acuerdo a las necesidades del proyecto.
- Diseño de una plataforma de integración dirigida por eventos que permite generar servicios web sensibles al contexto, basado en las características del ESB SwitchYard y el motor CEP Drools Fusion.
- Implementación de una plataforma basada en el diseño propuesto, empleando los patrones de conectividad y mediación que ofrece SwitchYard, y el poder de procesamiento de eventos complejos que brinda Drools Fusion.

  Dicha plataforma cuenta con una consola de administración que permite visualizar y administrar en tiempo de ejecución la lógica de adaptación necesaria para generar servicios sensibles al contexto.
- Desarrollo de casos de prueba que demuestran la factibilidad y aplicabilidad de la plataforma desarrollada.

1.4 Organización del Documento

El resto del documento se organiza de la siguiente manera:

En el capítulo 2 se presenta un marco teórico sobre la sensibilidad al contexto, las Arquitecturas Orientadas a Servicios, los estándares XML, los servicios web, el procesamiento de eventos complejos, las plataformas ESB y sus patrones. Además se describen las partes centrales del artículo propuesto en [1].

En el capítulo 3 se presenta el análisis que se realizó sobre distintos productos ESB, para seleccionar la plataforma que más se adecuara a la problemática planteada y se describen las principales características del ESB seleccionado y del motor CEP que se utilizó.

En el capítulo 4 se describe cómo se resolvió la solución conceptual de la problemática en base a los mecanismos que brindan los productos seleccionados. Se plantea un diseño de arquitectura con sus componentes, y se describe la responsabilidad de cada uno y la interacción entre los mismos.

En el capítulo 5 se detalla la implementación de cada componente de la plataforma mencionando los problemas encontrados a la hora de implementar y cómo fueron resueltos.

En el capítulo 6 se proponen tres casos de estudio. El primero se enfoca en mostrar un ciclo completo de uso de la plataforma, mientras que el resto profundiza en otros aspectos de la misma.

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

2 Marco Teórico

En este capítulo se presentan los conceptos más importantes para la problemática planteada, de manera de brindar una mejor comprensión del documento. En primer lugar, en la Sección 2.1 se profundiza en el concepto de sensibilidad al contexto y en la Sección 2.2 se describen distintas arquitecturas de software como Arquitecturas Orientadas a Servicios (SOA) y *Event Driven Arquitecture* (EDA). Luego, en la Sección 2.3 se describen estándares XML y en la Sección 2.4 conceptos relacionados a los servicios web. Además, en la Sección 2.5 y 2.6 se describen los conceptos *Enterprise Service Bus* (ESB) y Procesamiento de Eventos Complejos (CEP). Por último se describen las partes centrales del artículo "An ESB-Based Infrastructure for Event-Driven Context-Aware Web Services" [1].

2.1 Sensibilidad al Contexto

Tomando como base el artículo [3], en esta sección se presenta uno de los conceptos más importantes en este proyecto, el contexto del usuario. En la sub-sección 2.1.1 se define el contexto y los sistemas sensibles al contexto. Por último en la sub-sección 2.1.2 se establecen distintas clasificaciones del contexto.

2.1.1 Sistemas Sensibles al Contexto

El contexto es toda información que puede ser usada para caracterizar la situación de una entidad. Una entidad es una persona, lugar u objeto que es considerado relevante en la interacción entre el usuario y la aplicación, incluyendo a ambos.

Se dice que un sistema es sensible al contexto, cuando se utiliza la información del contexto del usuario para mejorar la calidad de la interacción entre ambos. Es decir, se aprovecha información como la localización, idioma u otra información del entorno del usuario para anticiparse a sus necesidades y así brindar sistemas más personalizados y de mejor calidad.

Por lo tanto, un sistema es sensible al contexto si utiliza el contexto para brindar mejores servicios al usuario y proveer información más relevante y específica, variando su comportamiento ante las necesidades del cliente. Cabe aclarar, que la información contextual debe ser opcional y el sistema debe poder funcionar sin la misma.

2.1.2 Clasificación del Contexto

Dado que el contexto es particular para cada sistema, hay información que aplica para un sistema y para otro no. Aunque es difícil establecer una clasificación del contexto se pueden distinguir tres grandes categorías [3]:

- **Contexto del dispositivo**: Son las características específicas de los dispositivos del sistema y la comunicación entre ellos, brindando información sobre su estado actual (uso, carga, etc.), su configuración y sus funcionalidades. Por ejemplo: servicios o redes disponibles, tamaño de pantalla, batería disponible, etc.

- **Contexto ambiental**: Condiciones ambientales en la que se encuentran los usuarios y dispositivos, generalmente obtenida a través de sensores (por ejemplo: temperatura, ubicación, etc.).
- **Contexto del usuario**: Son datos especificados por el usuario con respecto a sus preferencias, como por ejemplo datos personales, pasatiempos, etc.

Ser sensible al contexto también implica distinguir qué parte de la aplicación es afectada por él, de este modo se pueden distinguir tres categorías según lo que afecta:

- **Interfaz de usuario**: la forma de representar la información y la forma en que el usuario y la aplicación interactúan puede variar según el contexto; si es un dispositivo táctil o no, si tiene sonidos, etc.
- **Información**: La información que se brinda al usuario puede variar según el contexto. Por ejemplo, si se utiliza la ubicación del usuario es posible filtrar ciertos datos teniendo en cuenta el país donde se encuentra, o si es un dispositivo móvil se puede mostrar menor cantidad de datos.
- **Cambios en la funcionalidad**: El contexto puede determinar cambios en la funcionalidad de la aplicación. Por ejemplo, si se toman en cuenta las preferencias del usuario se puede determinar si un usuario desea pagar un servicio de una forma u otra.

2.2 Arquitecturas Orientada a Servicios y Dirigidas por Eventos

En esta sección se presentan las diferentes arquitecturas de *software* relevantes en la problemática planteada. Primero en la Sección 2.2.1 se describen las Arquitecturas Orientadas a Servicios (SOA) y en la Sección 2.2.2 se describen las Arquitecturas Dirigidas por Eventos (EDA). Por último en la Sección 2.2.3 se presenta una extensión de ambas arquitecturas llamada SOA Dirigida por Eventos (ED-SOA).

2.2.1 Arquitectura Orientada a Servicios (SOA)

Una Arquitectura Orientada a Servicios es una forma lógica de diseñar sistemas de software para proveer servicios a aplicaciones u otros servicios, a través de interfaces que son publicadas y pueden ser descubiertas dinámicamente [4].

Una SOA típicamente puede ser caracterizada por las siguientes propiedades [5]:

- Vista lógica: Un servicio es una vista lógica abstracta de programas o bases de datos, definido en términos de lo que hace.
- Orientado a mensajes: Un servicio está definido formalmente en términos del mensaje intercambiado entre el proveedor del servicio y el que solicita dicho servicio. La estructura interna de los agentes involucrados es abstracta, es decir, no es necesario conocer cómo está implementado un servicio.

- Orientado a su descripción: Un servicio es descrito por *metadata*[1] procesable por máquinas. La semántica del servicio debe estar documentada en esta descripción.
- Granularidad: Los servicios tienden a usar un número reducido de operaciones con mensajes relativamente largos y complejos.
- Orientado a redes: Los servicios están enfocados para que su uso sea a través de redes.
- Independientes de la plataforma: Los mensajes intercambiados son independientes de la plataforma y estandarizados.

2.2.2 Arquitectura Dirigida por Eventos (EDA)

Una arquitectura dirigida por eventos, del inglés Event Driven Arquitecture (EDA), define una metodología para diseñar e implementar aplicaciones y sistemas en donde los eventos son transmitidos entre componentes y servicios desacoplados. Mientras SOA es más apropiado para un intercambio solicitud/respuesta, EDA introduce capacidades de procesamiento asíncrono de larga duración [6].

EDA se basa en publicación y suscripción a eventos, donde el que publica dichos eventos no conoce al suscrito y viceversa, lo único que comparten es la semántica del mensaje [7].

2.2.3 SOA dirigida por Eventos (ED-SOA)

Del inglés Event-Driven SOA (ED-SOA) o también llamado SOA 2.0 es una extensión de SOA que combina lo mejor de dicha arquitectura con las capacidades de manejo de eventos de las arquitecturas EDA. Surge con la necesidad de proveer mecanismos para publicar y consumir eventos en arquitecturas orientadas a servicios [8].

En una ED-SOA, los servicios no requieren conocer la implementación de los protocolos de comunicación, ni tener conocimiento del ruteo de los mensajes a otros servicios. Una fuente de eventos típicamente envía los mensajes a través del mismo middleware de integración, y entonces el middleware publica los mensajes a los servicios suscritos al evento. El evento en sí mismo encapsula una actividad constituyendo una descripción completa de una acción específica [4].

2.3 Estándares XML

En esta sección se describen los estándares XML relevantes para el proyecto. Primero se describe el estándar XML, siguiendo con *XML Path Language (*XPATH) y *Extensible Stylesheet Language Transformations* (XSLT).

[1] Son datos que describen otros datos.

2.3.1 Extensible Markup Language (XML)

Es un lenguaje de marcado para representar información estructurada que utiliza "*tags*" para etiquetar, categorizar y organizar la información. XML no está limitado a un conjunto particular de "*tags*" ya que permite la creación y utilización de nuevos tags según sea necesario. Los documentos XML forman una estructura de árbol, ya que siempre existe un nodo raíz y a partir de éste se definen las hojas del árbol [9].

2.3.2 Extensible Stylesheet Language Transformation (XSLT)

Es un lenguaje para la transformación de documentos XML [10]. Una transformación XSLT describe cómo transformar un árbol origen o la información estructurada de un documento XML. La estructura destino puede ser totalmente diferente a la del origen, donde los elementos del árbol origen pueden ser filtrados y reordenados, permitiendo también agregar estructura arbitraria al mismo.

2.3.3 XML Path Language (XPath)

XPath es un lenguaje para seleccionar y procesar partes de un documento XML [11]. Con el mismo, es posible seleccionar y hacer referencia a texto, elementos, atributos y cualquier otra información contenida dentro de un fichero XML. Para soportar esto, también provee funcionalidades básicas para la manipulación de *strings*, números y *booleans*.

XPath debe su nombre al uso de notación de ruta (*path*) como URLs para navegar a través de la estructura jerárquica del documento XML. En XPath, los ficheros XML son modelados como un árbol de nodos. Existen diferentes tipos de nodos, como nodos de elementos, nodos de atributos y nodos de texto.

La construcción primaria de este lenguaje son las expresiones. Una expresión es evaluada para producir un objeto, y tiene uno de los siguientes tipos básicos:

- conjunto de nodos (colección desordenada de nodos sin duplicados)
- *boolean* (verdadero o falso)
- número (número de punto flotante)
- *string* (cadena de caracteres)

2.4 Servicios Web (WS)

Los servicios web son un conjunto de aplicaciones y tecnologías con capacidad de interoperar en la Web. Estas aplicaciones intercambian datos entre sí con el objetivo de ofrecer servicios. Los proveedores ofrecen sus servicios como procedimientos remotos y los usuarios solicitan un servicio llamando a estos procedimientos a través de la Web [5].

Los servicios web proporcionan mecanismos de comunicación estándar entre aplicaciones que interactúan entre sí para presentar información dinámica al usuario.

Para lograr interoperabilidad y extensibilidad entre aplicaciones, y que al mismo tiempo sea posible su combinación para realizar operaciones complejas, es necesaria una arquitectura de referencia estándar. Tanto OASIS[2] como W3C[3] son comités responsables de su arquitectura y reglamentación [12].

En todo el proceso de comunicación se utilizan una serie de tecnologías que hacen posible esta circulación de información. En la Figura 2 se puede ver el flujo de comunicación donde un proveedor publica un servicio en el directorio de servicios especificándolo con Web Services Description Language (WSDL). Luego el solicitante del servicio lo busca en dicho directorio utilizando Universal Description, Discovery and Integration (UDDI). Por último, al encontrar el servicio deseado, el solicitante invoca mediante Simple Object Access Protocol (SOAP) al proveedor. En las siguientes sub-secciones se presentan cada una de las tecnologías nombradas en el flujo de la Figura 2: WSDL, UDDI y SOAP.

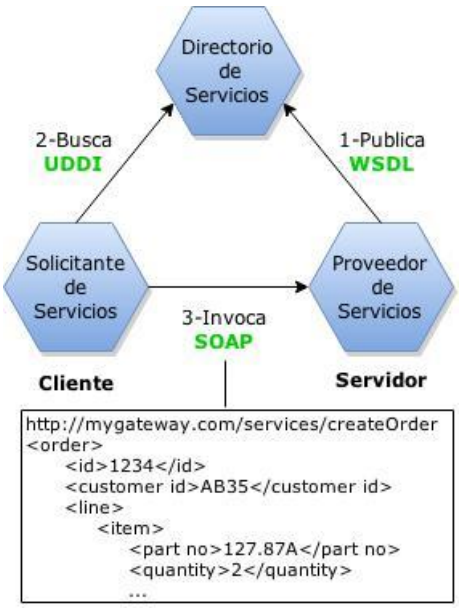


Figura 2 - Flujo de comunicación de servicios web, extraído de [13].

2.4.1 Simple Object Access Protocol (SOAP)

Se trata de un protocolo basado en XML, que permite la interacción entre varios dispositivos y que tiene la capacidad de transmitir información compleja. Los datos pueden ser transmitidos a través de HTTP, SMTP, etc. SOAP especifica el formato de los mensajes. El mensaje SOAP está compuesto por un *envelope* (sobre), cuya estructura está formada por los siguientes elementos: *header* (cabecera) y *body* (cuerpo) [12].

[2] https://www.oasis-open.org/

[3] http://www.w3.org/

2.4.2 Web Services Description Language (WSDL)

WSDL permite que un servicio y un cliente establezcan un acuerdo en lo que se refiere a los detalles de transporte de mensajes y su contenido, a través de un documento procesable por dispositivos. WSDL especifica la sintaxis y los mecanismos de intercambio de mensajes [12].

2.4.3 Universal Description, Discovery and Integration (UDDI)

Es un estándar de OASIS para publicar, categorizar y buscar servicios web. Los registros UDDI permiten clasificar los servicios de acuerdo al tipo de servicio, tipo de negocio o a las relaciones que poseen con otros servicios. Además, define una API que permite buscar, recuperar y publicar servicios [14].

2.4.4 WS Security

WS Security es una extensión de SOAP para proveer integridad y confidencialidad a los mensajes. Además provee un mecanismo para asociar *tokens* de seguridad al contenido del mensaje [15].

Existen tres tipos de *tokens*: Username Token, Binary Security Token y Security Assertion Markup Language (SAML). En particular, el Username Token tiene como objetivo proveer el nombre del usuario. Este *token* puede ser agregado opcionalmente y tiene el formato que se muestra en la Figura 3.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="...">
    <S11:Header>
            ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>Zoe</wsse:Username>
            </wsse:UsernameToken>
        </wsse:Security>
            ...
    </S11:Header>
    ...
</S11:Envelope>
```

Figura 3 – Encabezado Username Token de WS Security

2.5 Enterprise Service Bus (ESB)

En esta sección se presenta el concepto ESB, describiendo sus funcionalidades básicas y los patrones más importantes que debe implementar un ESB en el marco de este proyecto. Por último se resumen las principales características de la plataforma propuesta en el artículo "*Adaptive ESB Infrastructure for Service Based Systems*" [16], base para la solución propuesta en [1].

2.5.1 Definición de ESB

Enterprise Service Bus (ESB) es un conjunto de funcionalidades de infraestructura implementado usando tecnologías de *middleware* diseñado para permitir la

implementación, despliegue y administración de soluciones SOA [13]. Para realizar esto, provee procesamiento distribuido e integración basada en estándares. En particular, un ESB está diseñado para proveer comunicación entre aplicaciones de gran porte y otros componentes. De esta manera, el ESB actúa como medio de transporte y adaptación para permitir la distribución de estos servicios entre aplicaciones que se ejecutan en plataformas heterogéneas y usan diversos formatos de datos [4].

Un ESB combina las funcionalidades de los paradigmas EDA y SOA para simplificar la integración de las unidades de negocio. La invocación a un servicio en un ESB puede ser iniciada por un consumidor o un evento. También soporta comunicación sincrónica o asincrónica entre una o muchas aplicaciones o componentes [6].

En la Figura 4 se muestra una representación de un ESB, actuando como mediador entre distintos componente y aplicaciones.

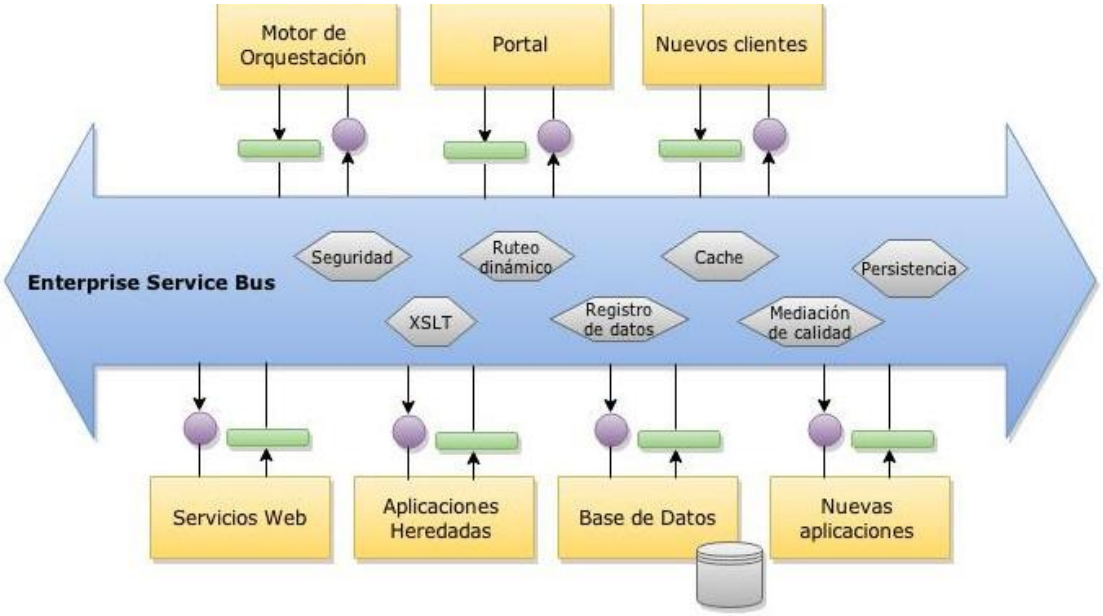


Figura 4 - Enterprise Service Bus, extraído de [17].

2.5.2 Funcionalidades Básicas de un ESB

Dado que existen gran variedad de implementaciones de plataformas ESB, es importante saber cuáles son las funcionalidades básicas que debe tener un ESB para ser considerado como tal. Según lo visto en el capítulo 4 del libro "*Patterns: Implementing an SOA Using an Enterprise Service Bus*" [13] las funcionalidades básicas esperadas de un ESB se resumen en la Tabla 1.

Categoría	Funcionalidad	Motivo
Comunicación	• Ruteo. • Direccionamiento. • Al menos un estilo de mensajería (solicitud/respuesta, etc.). • Al menos un protocolo de transporte.	Proveer transparencia de localización de servicios y soportar la sustitución de servicios.
Integración	• Múltiples estilos de integración y adaptadores. • Transformación de protocolos.	Soportar la integración en ambientes heterogéneos y sustituir servicios.
Interacción de servicios	• Definición de interfaz de servicios • Modelo de servicio de mensajes • Sustitución de la implementación de servicios	Soportar los principios de SOA, abstrayendo el código de la aplicación de los protocolos del servicio y su implementación.
Administración	• Capacidad de administración.	Punto de control sobre el direccionamiento y nombrado de servicios.

Tabla 1 - Funcionalidades básicas de un ESB, extraído de [13].

2.5.3 Patrones ESB

El uso y creación de patrones en la informática es una técnica muy utilizada y productiva ya que los problemas y soluciones tienden a repetirse, en este caso los ESB no son la excepción. Por esto, obtener experiencia de implementaciones de la vida real y abstraer de éstas un conjunto de patrones es de suma utilidad. Partir de un conjunto de soluciones probadas y recomendadas sirve como base para tomar decisiones de arquitectura e implementación, como también para la selección y uso de productos ESB.

Existe una gran variedad de patrones y caracterizaciones diferentes, una de ellas son los *Enterprise Integration Patterns* (EIP), cuyo objetivo es facilitar el diseño e implementación de soluciones de integración. Los EIP son un compilado de 65 patrones documentados en el libro "*Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*" [18]. Estos patrones fueron recabados con el objetivo de generar una guía de diseño tecnológicamente independiente que ayude a desarrolladores y arquitectos a diseñar y desarrollar soluciones de integración robustas [19]. Una gran variedad de plataformas de integración los utilizan, brindando una forma unificada y estandarizada para el diseño de aplicaciones ESB.

Para la solución a la problemática planteada se utilizaron distintos patrones, varios de ellos extraídos de los EIP. A continuación se nombran y se describen los patrones relevantes en el contexto de este proyecto agrupados en dos sub-secciones, patrones de mediación y de conectividad.

2.5.3.1 Patrones de Conectividad

Los patrones de conectividad hacen referencia a estilos de integración de alto nivel para soluciones ESB. En las siguientes sub-secciones se presentan los que se utilizaron en este proyecto.

Servicio Virtual

Este patrón no se encuentra definido explícitamente en los EIP pero vale la pena su inclusión en el contexto de esta problemática. Es un tipo de patrón que provee bajo acoplamiento entre servicios a través del agregado de niveles adicionales de in-dirección [20].

Dentro del grupo de patrones de servicios virtuales, existe el patrón servicio *proxy* simple que es de gran importancia para la plataforma desarrollada. Como se muestra en la Figura 5, este patrón toma un servicio existente y genera un nuevo servicio virtual en el ESB, permitiendo que sea accedido a través de un punto de acceso controlado y de protocolos diferentes a los brindados por el proveedor original. La introducción del ESB como punto de mediación hace posible la introducción de funcionalidades de administración, como por ejemplo: manejo de errores y alertas, control del tráfico y mediciones de rendimiento.

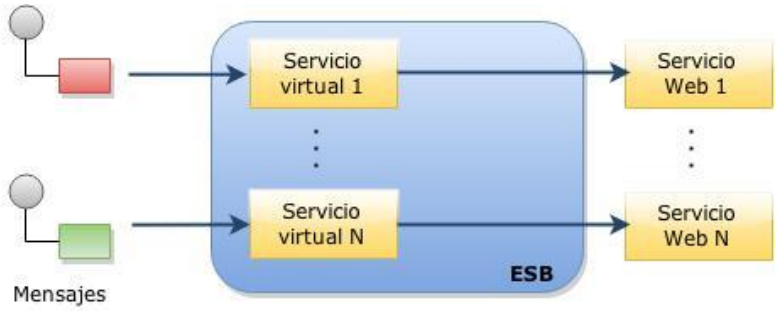


Figura 5 - Servicios virtuales

Gateway

Un *gateway* es parte del *bus* de mensajes o *bus* de servicios y provee funciones que aplican a todos los mensajes entrantes o salientes, es decir, permite aplicar un conjunto común de operaciones de mediación a todos los mensajes. Por ejemplo aplicar funciones de seguridad como autenticación, o auditoria del contenido de los mensajes [20]. En la Figura 6 se muestra cómo el *Gateway* intercepta todos los mensajes que llegan al ESB.

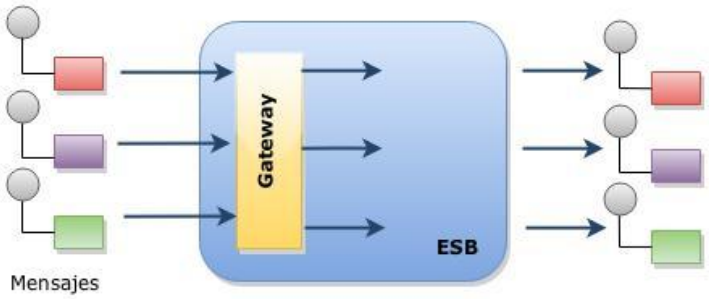


Figura 6 - Patrón Gateway

Consumidor Dirigido por Eventos (*Event-Driven Consumer*)

Con este patrón los mensajes son automáticamente entregados a medida que llegan. Esto también es conocido como un receptor asincrónico ya que el receptor no tiene un hilo corriendo hasta que recibe el mensaje. Es llamado *Event-Driven Consumer* porque el receptor trata al mensaje recibido como un evento que dispara una acción. En la Figura 7 se muestra cómo el consumidor dirigido por eventos reacciona al recibir un mensaje, disparando un evento que desencadena una acción.

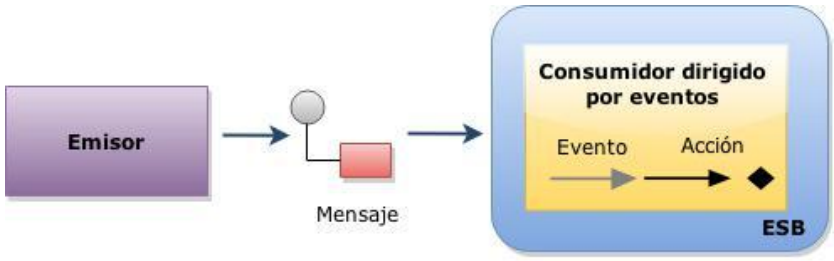


Figura 7 - Consumidor dirigido por eventos.

Mapeo de Mensaje (*Messaging Mapper*)

Este patrón consiste en crear un componente que contiene la lógica de mapeo entre la infraestructura de mensajería y el dominio del mensaje. Como se puede ver en la Figura 8, este componente accede a uno o más objetos del dominio y los convierte en el mensaje requerido por el canal de mensajería. También realiza la operación inversa.

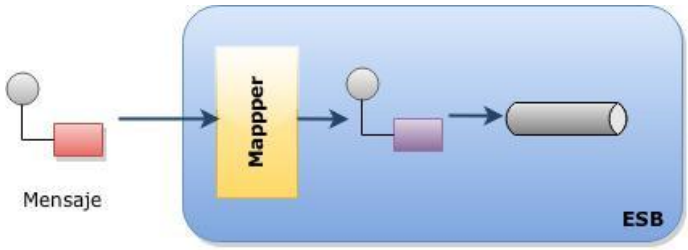


Figura 8 - Mapeo de mensaje.

Polling Consumer

El patrón *Polling Consumer* es usado para aplicaciones que explícitamente hacen una llamada cuando quieren recibir un mensaje. Es también llamado receptor sincrónico, ya que el receptor se bloquea hasta recibir el mensaje. En la Figura 9 se muestra un esquema con la idea general del patrón, donde el *poller* invoca al emisor del mensaje, procesa el mensaje que se le envía y luego vuelve a invocar al emisor para recibir un nuevo mensaje.

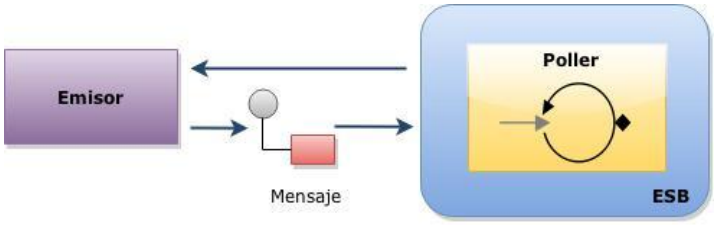


Figura 9 - Polling Consumer.

2.5.3.2 Patrones de Mediación

Los patrones de mediación especifican familias de operaciones que pueden ser realizadas sobre los mensajes que pasan por el ESB. En las siguientes sub-secciones se presentan los que se utilizaron en este proyecto.

Ruteo de Mensajes

- **Ruteo Basado en Contenido (*Content Based Router*)**

El ruteo basado en contenido examina el contenido del mensaje y dirige el mensaje al destinatario correspondiente basándose en información del mismo. El ruteo puede estar basado en diferentes criterios como: la existencia de cierto campo, valor específico de un campo, etc. En la Figura 10 se muestra un diagrama del mismo.

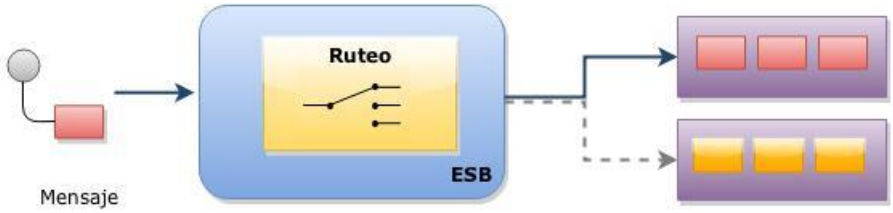


Figura 10 - Ruteo basado en contenido.

- **Ruteo Basado en Itinerario (*Routing Slip*)**

El ruteo basado en itinerario tiene como objetivo dirigir el mensaje a través de una serie de destinatarios en forma consecutiva, dicha serie no está definida en tiempo de diseño y puede variar para cada mensaje. Como primera etapa, se genera un componente especial al comienzo del proceso que computa la lista de pasos requeridos para cada

mensaje. Luego de generada esta lista, se inserta al mensaje y se comienza el proceso dirigiendo el mensaje al primer paso correspondiente. Después de finalizado correctamente, cada paso consulta el itinerario y pasa el mensaje al siguiente destinatario especificado. En la Figura 11 se muestra un esquema del patrón donde se incorpora al mensaje un itinerario indicando que el primer paso es el proceso A y el segundo es el proceso C.

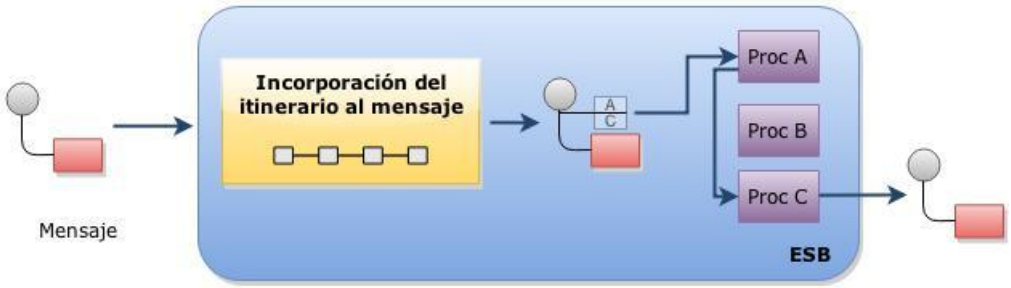


Figura 11 - Ruteo basado en itinerario.

Transformación de Mensajes

- **Enriquecimiento de Contenido (*Content Enrichment*)**

Para enriquecer el contenido de un mensaje se usa información del mismo (ej.: campo clave) para recuperar información desde una fuente externa. Luego de obtenida dicha información, se agrega al mensaje. La información del mensaje original puede ser incluida en el mensaje resultante o puede no ser requerida, dependiendo de las necesidades específicas de la aplicación receptora. En la Figura 12 se muestra una ejemplificación del patrón, donde el enriquecedor toma los mensajes entrantes y les agrega información.

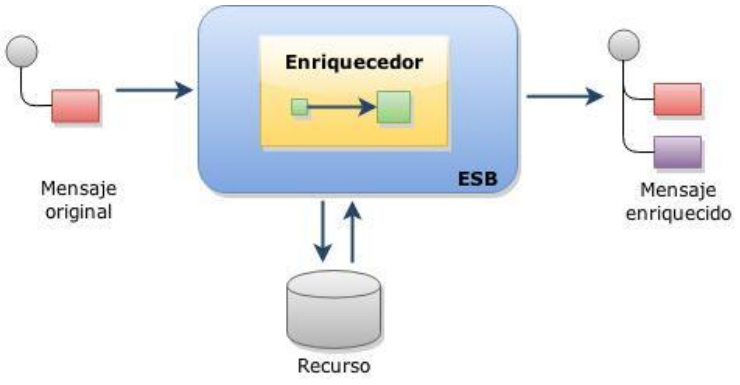


Figura 12 - Enriquecimiento de contexto.

- **Filtro de Contenido (*Content Filter*)**

A diferencia del patrón *Content Enrichment* donde se añade información, este patrón es utilizado para remover datos del mensaje. En la Figura 13 se muestra un esquema de este patrón donde se remueven datos del mensaje.

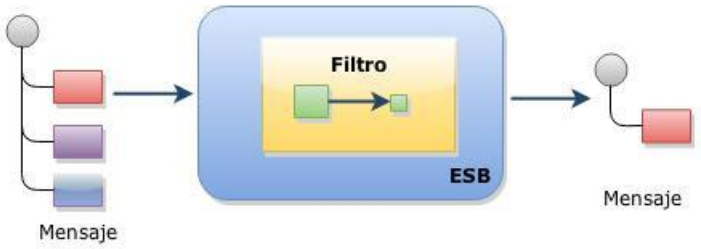


Figura 13 - Filtro de contenido.

2.5.4 ESB Adaptativo

Esta sección describe la plataforma del ESB adaptativo propuesto en [16]. La plataforma se encarga de aplicar adaptaciones dinámica y automáticamente en sistemas basados en servicios.

La solución propuesta supone que los servicios se comunican con mensajes a través del ESB, utilizando los patrones de virtualización de servicios. Para aplicar las adaptaciones en tiempo de ejecución, se interceptan todos los mensajes que llegan al ESB, y si es necesaria una adaptación para el servicio que se está invocando, se envían a través de flujos de adaptación. Estos flujos son construidos con patrones de mediación como transformaciones y ruteos, para ser capaces de aplicar la estrategia de adaptación que sea necesaria, como por ejemplo invocar un servicio equivalente. Para saber si un servicio requiere una adaptación, previamente debe configurarse en la plataforma las directivas de adaptación necesarias para cada servicio. Estas directivas se generan a partir de propiedades de servicios monitoreados y de requerimientos de nivel de servicio. La Figura 14 presenta la plataforma ESB adaptativa con un ejemplo donde el flujo de adaptación consiste en aplicar una transformación antes de la invocación al servicio.

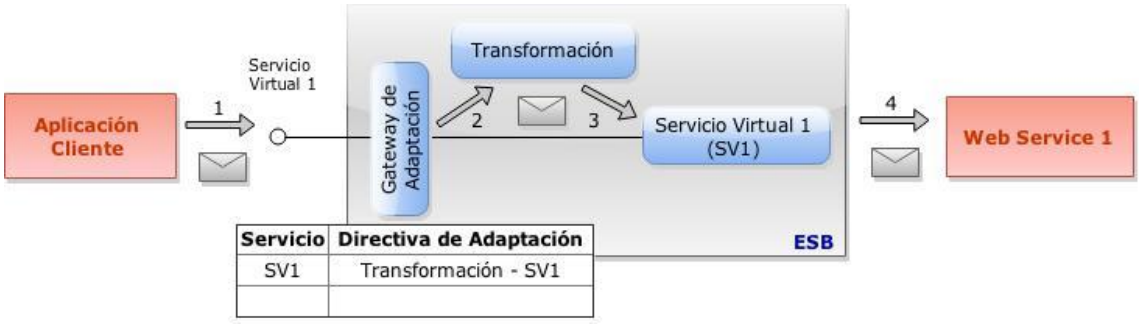


Figura 14 - Plataforma ESB adaptativa, extraído de [16].

Como se puede ver en la Figura 14 el cliente envía un mensaje a través del ESB (1) para invocar al servicio destino. El mensaje es interceptado por el *gateway* de adaptación utilizando el patrón *gateway* del ESB. Si existe una directiva de adaptación configurada para el servicio, el *gateway* de adaptación agrega al mensaje el flujo de adaptación correspondiente y utilizando el patrón ruteo basado en itinerario, dirige el mensaje al primer paso del flujo (2). Luego de aplicarse la transformación, especificada en el

mensaje, el mensaje es dirigido al siguiente paso del flujo (3) que invoca al servicio destino (4).

2.6 Procesamiento de Eventos Complejos (CEP)

Procesamiento de Eventos Complejos, del inglés *Complex Event Processing* (CEP), es una tecnología que provee un conjunto de técnicas para descubrir eventos complejos analizando y correlacionando eventos simples y complejos [21]. Un evento simple ocurre en un punto en el tiempo, es indivisible y atómico, mientras que un evento complejo puede ocurrir en un período de tiempo, representa o denota un conjunto de eventos simples o complejos y contiene un significado más semántico. Alguna de estas técnicas son: detectar la causalidad de eventos, definir relaciones temporales entre eventos, abstraer procesos dirigidos por eventos y detectar patrones de eventos. Por lo tanto CEP permite detectar eventos complejos y significativos, llamados situaciones, e inferir conocimiento útil y valioso para el usuario y el negocio [22]. En el glosario [23] se pueden encontrar todos los conceptos vinculados a CEP.

La ventaja principal del uso de CEP es que los eventos pueden ser identificados y reportados en tiempo real, no como en otros enfoques tradicionales de análisis de eventos, por lo tanto se reduce la demora en la toma de decisiones. En consecuencia, CEP es una tecnología ideal para aplicaciones que:

- Deben responder velozmente ante situaciones que cambian rápido y asincrónicamente.
- Deben reaccionar rápidamente a situaciones inusuales.
- Requieren bajo acoplamiento y deben ser adaptables.

2.7 Event-Driven Integration Platform for Context-Aware Web Services

En esta sección se presenta un resumen sobre la solución planteada en [1], donde las capacidades de las plataformas ESB y CEP permiten la construcción de servicios sensibles al contexto.

2.7.1 Descripción General

La solución propuesta consiste en construir y exponer servicios sensibles al contexto a través del ESB utilizando servicios virtuales, partiendo de servicios que pueden no ser sensibles al contexto. Para poder construirlos, el ESB aplica lógica de adaptación utilizando sus capacidades de mediación. Las adaptaciones son generadas automáticamente dependiendo de la situación en que se encuentre el usuario que invoca al servicio. Para detectar las situaciones, éstas deben ser definidas previamente en el motor CEP como eventos complejos. Luego, serán disparados dependiendo de la información contextual que llega al ESB desde distintas fuentes. La Figura 15 presenta un esquema general de la arquitectura propuesta.

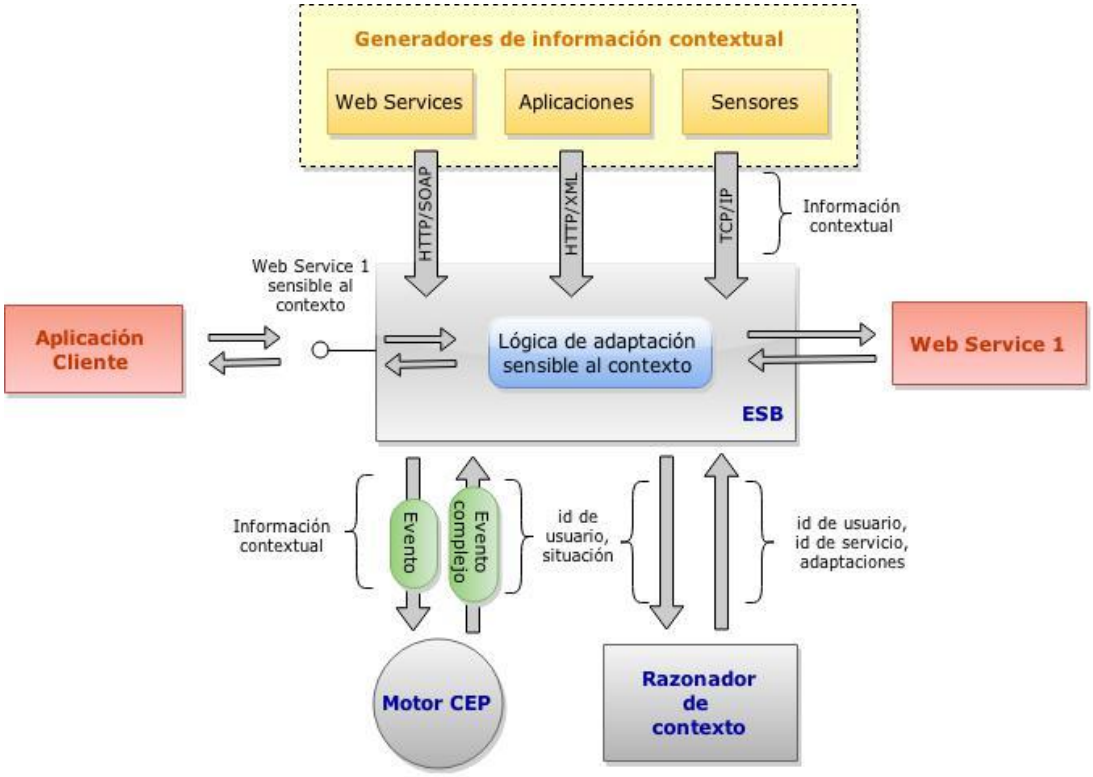


Figura 15 - Arquitectura general, extraído de [1].

Las fuentes de contexto proveen al ESB con diferentes tipos de información contextual en forma de eventos. Por ejemplo, un dispositivo móvil con GPS puede proveer información sobre la localización geográfica del usuario o sensores pueden monitorear el ambiente para proveer información sobre condiciones climáticas. El motor CEP recibe toda esta información a través del ESB y en base a reglas que tiene definidas detecta situaciones complejas en las que los usuarios se encuentran. Por ejemplo, basado en la localización geográfica del usuario y el clima de una ciudad, el motor puede inferir que el usuario se encuentra en una ciudad donde está lloviendo.

El razonador de contexto recibe las situaciones en la que los usuarios se encuentran y en tiempo de ejecución genera las adaptaciones que sean necesarias para cada servicio que se encuentre configurado. Para esto, el razonador de contexto debe tener establecidas las situaciones que afectan cada servicio y las adaptaciones que se deben de aplicar en cada caso. Por ejemplo, si un servicio retorna las atracciones de una ciudad y el usuario se encuentra en una ciudad donde está lloviendo, una adaptación puede ser no retornar las atracciones que sean al aire libre.

Por último, el ESB recibe las adaptaciones requeridas para cada usuario y servicio, y cuando llegue una invocación debe aplicar las adaptaciones correspondientes mediante sus capacidades de medicación. En el ejemplo anterior, consiste en aplicar el patrón de filtrado para quitar de la respuesta las atracciones al aire libre.

La Figura 16 presenta los principales elementos del modelo conceptual de la solución propuesta con los ejemplos antes nombrados.

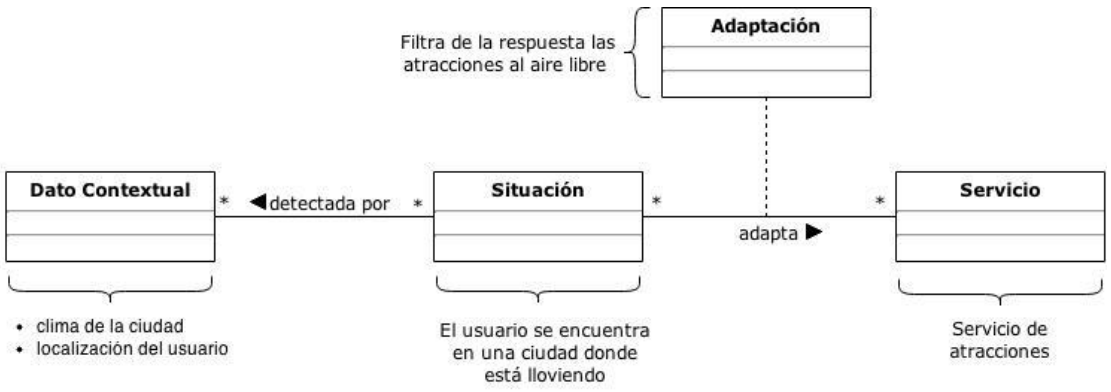


Figura 16 - Modelo conceptual, extraído de [1].

2.7.2 Recepción de Información Contextual

La información contextual es obtenida a través de las fuentes de contexto en forma de eventos, utilizando las capacidades de conectividad del ESB. Los eventos que se reciben se envían al motor CEP para ser procesados.

2.7.3 Detección de Situaciones

Las situaciones son detectadas como eventos complejos por el motor CEP. Para poder especificar situaciones en la que se encuentra el usuario, es necesario definir reglas en el motor CEP para detectar dichas situaciones. Las reglas se pueden basar en información contextual, en otras situaciones detectadas y en funciones auxiliares. Por ejemplo, si se tiene información contextual como la localización del usuario y una función auxiliar que devuelve la ciudad a partir de las coordenadas, se puede detectar que el usuario se encuentra en determinada ciudad. Para esto se debe especificar una regla que dispare dicha situación automáticamente cuando se reciba la localización del usuario utilizando la función auxiliar. Para definir las reglas se pueden utilizar distintos lenguajes, como por ejemplo *Drools Rule Language*[4] (DRL).

2.7.4 Configuración de Adaptaciones

Para poder adaptar los servicios en función de la situación en que se encuentra el usuario, el razonador de contexto debe contar con las siguientes configuraciones:

- La situación que afecta cada servicio.
- Las adaptaciones que deben aplicarse en cada caso.
- El momento a ser aplicadas las adaptaciones, ya sea en la invocación o en la respuesta del servicio.
- La prioridad de las adaptaciones para cada servicio, en caso que haya más de una adaptación configurada para un servicio.

[4] https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch05.html

Cada adaptación debe tener configurada qué patrón de mediación necesita aplicar y la información que utilizará. Por ejemplo, si se detectó que el usuario se encuentra en una ciudad, se puede enriquecer la invocación al servicio agregando la ciudad como un parámetro extra en la invocación, si es que el servicio lo soporta. Para esto puede aplicarse el patrón de enriquecimiento y por lo tanto se debe configurar una transformación XSLT, explicada en la Sección 2.3.2, que agrega la ciudad en el mensaje SOAP. La adaptación no queda totalmente configurada hasta que se detecta la situación para un usuario, en el ejemplo, el XSLT no tendrá cargado la ciudad hasta que se detecte la misma.

2.7.5 Adaptación de Servicios

Luego de la configuración de las adaptaciones para cada servicio y situación, cuando el razonador de contexto recibe información sobre la situación del usuario, genera automáticamente las adaptaciones necesarias para el usuario y situación dados, comunicándolas al ESB. Por ejemplo, si se le informó al razonador de contexto que un usuario se encuentra en determinada ciudad, según el ejemplo de la sección anterior, la adaptación que debe enviar al ESB es un XSLT, donde el parámetro ciudad es la ciudad informada. Suponiendo que se configuró previamente que la situación informada afecta el servicio que retorna las atracciones, el usuario solo va a recibir las atracciones de la ciudad donde se encuentra.

El ESB debe aplicar dinámicamente las adaptaciones que envía el razonador de contexto, para que esto sea posible se utiliza la plataforma del ESB adaptativo descripta en la Sección 2.5.4. La información que recibe el ESB debe ser tratada como directivas de adaptación a aplicarse en la invocación al servicio por parte del usuario. Por lo tanto, la infraestructura del ESB adaptativo debe ser modificada para tener en cuenta la información del usuario en la invocación al servicio y para aplicar las directivas de adaptación. La Figura 17 presenta la infraestructura del ESB adaptativo con las mejoras mencionadas para aplicar adaptaciones sensibles al contexto.

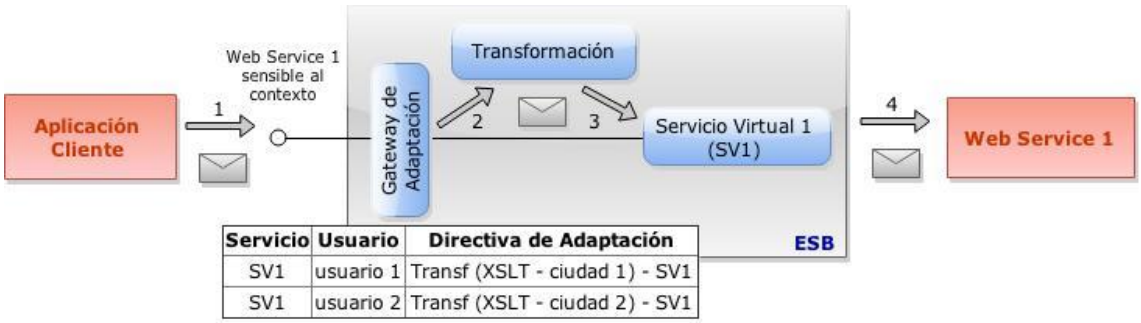


Figura 17 - Adaptaciones sensibles al contexto, extraído de [1] .

3 Selección de Tecnologías a Utilizar

En esta sección se presentan las tecnologías principales utilizadas para la implementación del prototipo. En la Sección 3.1 se explica la elección de Drools Fusion como motor CEP y en la Sección 3.2 se detalla el proceso de elección de la plataforma ESB utilizada: SwitchYard. Luego se describen Drools Fusion y Switchyard, presentando sus componentes principales y funcionalidades relevantes al proyecto.

3.1 Selección del Motor CEP

El componente CEP fue implementado con Drools Fusion como Motor. La elección del producto fue sugerida por los docentes supervisores y además es el producto de referencia utilizado en [1]. Por otro lado, ya se contaba con experiencia utilizando Drools Fusion. Todo esto motivó a que en vez de realizar una evaluación de los distintos productos del mercado, simplemente se validó que con Drools Fusion se pudieran implementar los requerimientos del proyecto.

3.2 Selección del ESB

La selección de la plataforma ESB a utilizar fue una de las etapas más importantes para comenzar la implementación del prototipo. Para esto, se investigaron varios productos actuales para elegir así el que mejor se adaptaba a las necesidades del proyecto.

3.2.1 Proceso de Selección

En esta sección se describe el proceso de selección del producto ESB que se utilizó para la realización del prototipo. Existen tres etapas importantes a destacar: la selección de los productos a evaluar, la evaluación de los mismos y la elección final del producto en base a lo analizado.

Como primera etapa se relevaron los productos ESB existentes. Dado que se debía integrar el ESB con el motor de eventos complejos Drools Fusion, dicha integración fue un aspecto muy importante a tener en cuenta. Otro aspecto de suma importancia fue la implementación de los EIP por parte de los mismos.

Para la selección del producto ESB se tomó en cuenta un conjunto de criterios relevantes para el desarrollo de la plataforma. Estos criterios se enfocaron en la evaluación de las capacidades de mediación y conectividad de las plataformas ESB, teniendo en cuenta los patrones necesarios para este proyecto, como el ruteo basado en itinerario y la transformación de mensajes.

A continuación se describen los criterios utilizados en la evaluación, indicando qué es lo que mide cada punto analizado.

- **Comunidad activa:** Se refiere a la actividad que tienen los foros dedicados al producto en cuestión. El criterio se considera cumplido si el tiempo máximo entre *posts* es de 5 días.

- **Ambiente de desarrollo integrado (IDE):** Evalúa si la plataforma brinda un ambiente de programación fácil de usar y flexible con editor de código, compilador, *debugger* y editor de interfaz gráfica. El IDE puede ser una aplicación independiente o puede ser parte de otra aplicación compatible, como por ejemplo un *plugin*.
- **Facilidad de uso:** Hace referencia a la facilidad con la que se puede interactuar con la plataforma, enfocado al proceso de instalación y a la realización de casos de uso básicos. Se considera cumplido si cuenta con guías de uso rápido las cuales permiten instalar y utilizar las funcionalidades básicas de la plataforma en menos de 2 horas.
- **Documentación:** Implica buena calidad de información disponible sobre las funcionalidades de la plataforma, incluyendo buenos ejemplos de uso de sus características principales.
- **Protocolos de comunicación:** Evalúa si la plataforma soporta múltiples protocolos de comunicación, ya que el prototipo a implementar necesita recibir información contextual de distintas fuentes de contexto con distintos protocolos de comunicación. El criterio se considera cubierto si al menos soporta HTTP, SOAP y REST.
- **Patrones de mediación y conectividad:** Evalúa si la plataforma provee implementaciones de los patrones de conectividad y mediación necesarios para la propuesta.
- **Integración con Drools:** Se refiere a la capacidad de integración con Drools que brinda el producto de manera nativa.

3.2.2 Productos a Evaluar

En esta sección se presenta una breve descripción de las plataformas ESB que fueron elegidas para ser evaluadas. Para la preselección de los productos, una condición requerida fue que los mismos sean de código abierto, haciendo posible realizar modificaciones o extensiones al mismo. Otro aspecto a tener en cuenta, fue la preferencia por tecnologías que se integren fácilmente con Drools Fusion, el motor CEP elegido. En este aspecto, las tecnologías JBoss presentaban una ventaja importante.

En base a estos criterios, los productos elegidos fueron: JBoss Fuse, JBoss Fuse Service Works, WSO2, SwitchYard y el ESB Adaptativo basado en JBoss ESB descrito en la Sección 2.5.4.

3.2.2.1 JBoss Fuse

Es una plataforma de código abierto, de la comunidad JBoss, de tamaño reducido y adaptable que permite una rápida integración entre aplicaciones. Brinda un *framework* de integración basado en patrones ya que utiliza como base Apache Camel, explicado

en la Sección 3.4.3 (ver Figura 18). También puede ser configurado y administrado dinámicamente mientras la plataforma está ejecutando y utiliza como servidor de aplicaciones Apache Karaf[5]. Provee múltiples opciones de conexión como JDBC, HTTP y RMI entre otros. Está integrado a JBoss Developer Studio y cuenta con una consola web para su administración. La versión evaluada fue JBoss Fuse 6.1.0.GA [24].

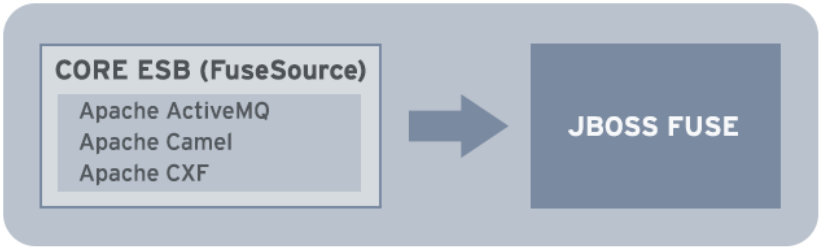


Figura 18 - JBoss Fuse, extraído de [24].

3.2.2.2 JBoss Fuse Service Works

Es una plataforma de código abierto para el diseño, desarrollo e integración de servicios perteneciente a la comunidad JBoss. Es la solución de middleware de Red Hat para integración de aplicaciones, mensajería y SOA. Como se muestra en la Figura 19, combina y utiliza las funcionalidades de JBoss Fuse, SwitchYard (Sección 3.2.2.5) y Overlord[6], ver Figura 19. Corre sobre JBoss Enterprise Application Platform y la versión evaluada fue JBoss Fuse Service Works 6.0.0.GA [25].

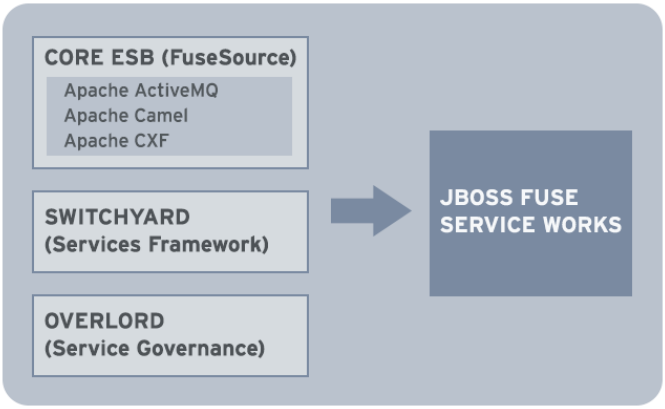


Figura 19 - JBoss Fuse Service Works, extraído de [25].

3.2.2.3 WSO2 ESB

Es un ESB de código abierto rápido, liviano y fácil de usar distribuido mediante la licencia Apache Software License v2.0[7]. Permite a los desarrolladores y administradores del sistema configurar fácilmente rutas de mensajería, transformaciones, programación de

[5] http://karaf.apache.org/
[6] http://www.projectoverlord.io/
[7] http://www.apache.org/licenses/LICENSE-2.0.html

tareas, balanceo de carga, etc. Para casos de integración avanzados también soporta mediación basada en reglas y prioridades, mensajería por eventos y otras técnicas avanzadas. Está diseñado para ser totalmente asincrónico, no bloqueante y basado en *streams*. La versión probada fue la 4.8.1, la última estable al momento de la evaluación [26].

3.2.2.4 ESB Adaptativo (basado en JBoss ESB)

Esta plataforma fue implementada como solución a la propuesta en [27] descripta en la Sección 2.5.4, la misma utiliza como base JBoss ESB. JBoss ESB es un producto de gran madurez que aprovecha tecnologías de la comunidad como el motor de reglas de negocios para el ruteo de mensajes basado en contenido. Utiliza una arquitectura flexible basada en los principios de SOA como el bajo acoplamiento y el pasaje asíncrono de mensajes. La versión utilizada como base fue la 4.11 [28].

Basados en esta plataforma se desarrolló un prototipo que extiende JBoss ESB agregando la capacidad de aplicar ruteo basado en itinerario, además de funcionalidades de administración y monitoreo.

3.2.2.5 SwitchYard

Es un *framework* de desarrollo basado en componentes enfocado en la construcción de servicios y aplicaciones estructuradas y fáciles de mantener, utilizando las mejores prácticas de SOA. Trabaja con Apache Camel para proveer una integración rápida, flexible y con gran poder de conectividad. Pertenece a la comunidad JBoss y es una aplicación de código abierto bajo la licencia de Apache. La versión evaluada fue la 1.1.0, su última versión estable al momento de la evaluación [29].

3.2.2.6 Resumen de Características de los Productos ESB

En la Tabla 2 se muestra un resumen de las características de los productos seleccionados para la evaluación. Para cada producto se tomó la última versión estable al momento de la evaluación, en Junio de 2014.

Características	JBoss Fuse	JBoss Fuse Service Works	WSO2	JBoss ESB (ESB Adaptativo)	SwitchYard
Última versión estable	6.1.0.GA 14/04/2014	6.0.0.GA 30/01/2014	4.8.1 4/02/2014	4.12 23/03/2013	1.1.0.Final 27/11/2013
Licencia	Apache v2.0	Apache v2.0	Apache v2.0	Lesser General Public License v2.1	Apache v2.0
Código abierto	Si	Si	Si	Si	Si
Respaldo	Red Hat	Red Hat	WSO2	Red Hat	Red Hat
Servidores Web	JBoss EAP 6.1, Apache Karaf	JBoss EAP 6.1	Tomcat	JBoss EAP 6.1	JBoss EAP 6.1

Tabla 2 – Resumen de características de los productos ESB.

3.2.3 Evaluación de Productos

Para la evaluación se descartó el ESB Adaptativo ya que está basado en JBoss ESB, una herramienta discontinuada. Esto hace que errores del mismo sean difíciles de corregir y que su comunidad sea cada vez menor, lo que provocó que fuera descartado desde un principio.

En el proceso de evaluación se estableció como objetivo probar cada plataforma a través de casos de uso sencillos. Cada producto fue instalado, y configurado en entornos de prueba para poder evaluar su funcionamiento, facilidad de uso y su documentación.

En esta instancia sobresalieron dos productos, WSO2 ESB y SwitchYard, los cuales presentan una interfaz amigable para su uso y configuración. Para el resto de los productos, JBoss Fuse y JBoss Fuse Service Works, el proceso de prueba fue insatisfactorio. Las pruebas resultaron más complejas ya que implementar casos similares a los de las demás plataformas implicó mayor tiempo y trabajo. Esto se debía a su documentación mal organizada y complejidades propias de la plataforma.

3.2.4 Selección del Producto ESB

En la Tabla 3 se muestran los criterios aplicados a los productos seleccionados para la evaluación. Como se mencionó anteriormente, el producto ESB Adaptativo no fue considerado en esta evaluación ya que JBoss ESB fue descontinuado.

Criterios	JBoss Fuse	JBoss Fuse Service Works	WSO2	SwitchYard
Comunidad activa	No	No	No	Si
Documentación	Aceptable	Aceptable	Buena	Buena
IDE	Aceptable	Aceptable	Muy bueno	Bueno
Facilidad de uso	No	No	Si	Si
Múltiples protocolos de comunicación	Si	Si	Si	Si
Patrones de mediación y conectividad	EIP	EIP	EIP	EIP
Integración con Drools	No	Si	No	Si

Tabla 3 - Criterios de evaluación aplicados.

Si bien los cuatro productos cumplen con los requerimientos a nivel técnico, SwitchYard sobresale en ciertos aspectos como su facilidad de uso, documentación e integración con Drools. Otro aspecto importante es el respaldo de la comunidad JBoss con el que cuenta SwitchYard. En tanto WSO2 ESB cuenta con una consola de administración muy interesante y una buena documentación, pero no cuenta con una integración nativa con Drools y tampoco tiene una comunidad activa, por lo que fue descartado.

Por lo tanto se decidió utilizar SwitchYard como producto ESB ya que cumple la mayor cantidad de criterios establecidos.

3.3 Drools Fusion

JBoss Drools Fusion es un software libre distribuido según licencia Apache que se define como un sistema de gestión de reglas de negocio con un sistema de inferencia de reglas. Actualmente Drools es un proyecto compuesto por muchos sub-proyectos entre los que se encuentra Fusion, la parte orientada a CEP. Para este proyecto se utilizó la versión 6.0.0 Final, la última versión estable de la herramienta al momento de comenzar la implementación.

Las principales características del producto son:

- Disponible solo para Java.
- Soporta operadores temporales, lógicos y aritméticos.
- Soporta ventanas de eventos tanto por cantidad como por tiempo.
- Soporta funciones de agregación en las ventanas.
- Cuenta con un Garbage Collector de eventos.
- Ejecución de código Java directamente en reglas.

- Representación de eventos como Plain Old Java Objects[8] (POJOS).
- Posibilidad de crear o modificar reglas en tiempo de ejecución.

Drools define los eventos como: "Un registro de un cambio de estado significativo en el dominio de una aplicación en un momento de tiempo dado" [30]. Un evento puede ocurrir durante un intervalo de tiempo determinado o en un instante (duración cero).

3.3.1 Drools Rule Language (DRL)

Drools tiene un lenguaje nativo para definir sus reglas, Drools Rule Language (DRL). Es un formato permisivo en términos de puntuación y permite ser extendido para adaptarse a los distintos dominios de los problemas. Para el proyecto se soporta sólo el lenguaje nativo.

En la Figura 20 se muestra la estructura de una regla con sus componentes principales. Los atributos definen algunos comportamientos específicos de la regla y son opcionales. El lado izquierdo de la regla (LHS, del inglés *Left Hand Side*), es la parte condicional y define cuándo debe ejecutarse la parte derecha (RHS, del inglés *Right Hand Side*) que es la acción que se debe realizar, básicamente código Java.

```
1  rule "nombre de la regla"
2      atributos
3      when
4          LHS
5      then
6          RHS
7  end
8
```

Figura 20 - Estructura de una regla.

3.4 SwitchYard

En esta sección se presentan las características de SwitchYard, el producto seleccionado para la implementación de la plataforma. Se describe brevemente el producto y se explican los principales componentes que fueron utilizados. Además se presenta Apache Camel, el motor de ruteo y mediación de SwitchYard.

3.4.1 Descripción

Es un *framework* ligero que provee soporte para el desarrollo, despliegue y administración de aplicaciones SOA. Incorpora Apache Camel, junto con Java EE[9], gestión de procesos de negocios, manejo de reglas y ruteo de mensajes. Funcionalidades como la validación, transformación y políticas están aisladas de la lógica de negocios y son usadas de forma declarativa. Esto asegura consistencia y reduce la duplicación de

[8] Instancia de una clase que no extiende ni implementa nada en particular.

[9] http://www.oracle.com/technetwork/java/javaee/overview/index.html

código, ofreciendo a los desarrolladores una clara visión de la estructura y relaciones de los servicios.

Cuenta con un *plugin* para Eclipse[10] que permite el diseño y la implementación con mayor facilidad.

3.4.2 Componentes Principales

En esta sub-sección se muestran los componentes principales de SwitchYard utilizados en este proyecto. Con el *plugin* de Eclipse los mismos pueden ser usados de manera gráfica como se ve en la Figura 21.

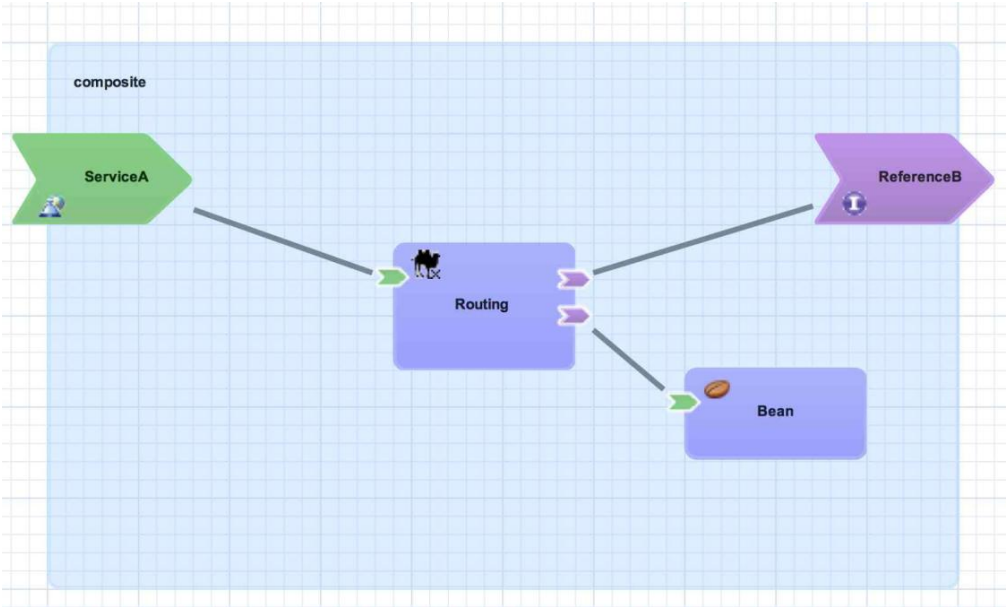


Figura 21 - Ejemplo de aplicación en SwitchYard, extraído de [31].

En el ejemplo de la Figura 21, la aplicación expone un servicio “ServiceA” donde el componente “Routing” recibe los mensajes y dirige a cada uno hacia el *bean* “Bean” o hacia un servicio externo mediante el componente “ReferenceB”.

En la Tabla 4 se muestra la representación gráfica de los distintos componentes de SwtichYard utilizados, con una breve descripción de su función. En el Apéndice 1 se encuentra información más detallada de cada componente.

[10] https://docs.jboss.org/author/display/SWITCHYARD/Installing+Eclipse+Tooling

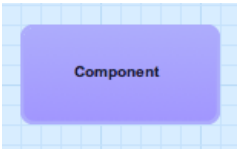
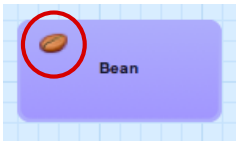
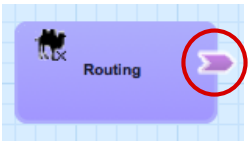
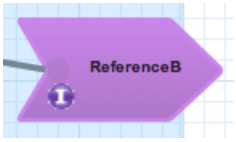
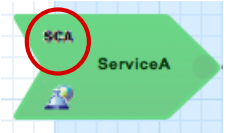
Componente	Representación gráfica	Descripción
Composite		Representa los límites de la aplicación.
Component		Contenedor modular para la lógica de la aplicación.
Bean Implementation		Permite que un Bean consuma o provea servicios mediante anotaciones.
Camel Java Implementation		Permite la utilización de los EIP usando Java DSL[11] con Apache Camel.
Camel XML Implementation		Permite la utilización de los EIP usando XML con Apache Camel.
Component Service		Expone la funcionalidad de una implementación como un servicio.
Composite Service		Representa un servicio que es visible para otras aplicaciones y extiende un *component service*.
Component Reference		Permite a un componente consumir otros servicios.
Composite Reference		Permite conectar un *component reference* con servicios fuera de la aplicación.
Service Binding		Define el método de acceso al *composite service*, en este caso es un *binding* SCA.
Reference Binding		Define el método de acceso a un servicio externo a través de un *composite reference.*

Tabla 4 – Representación gráfica de los componentes SwitchYard.

[11] http://camel.apache.org/dsl.html

3.4.3 Apache Camel

SwitchYard implementa los EIP a través de Apache Camel, por esto es necesario describir qué es Apache Camel[12] y cuáles son sus funcionalidades principales.

Apache Camel es un *framework* de código abierto escrito en Java que se enfoca en hacer la integración entre aplicaciones y componentes más fácil y accesible para desarrolladores [32]. Lo hace brindando:

- implementaciones concretas de los EIP más usados.
- conectividad de una gran variedad de protocolos de transporte y APIs.
- un lenguaje específico del dominio (DSL) fácil de usar para conectar EIPs y los diferentes tipos de comunicación.

En la Figura 22 se muestra cómo se mapean los tres conceptos antes nombrados a Apache Camel.

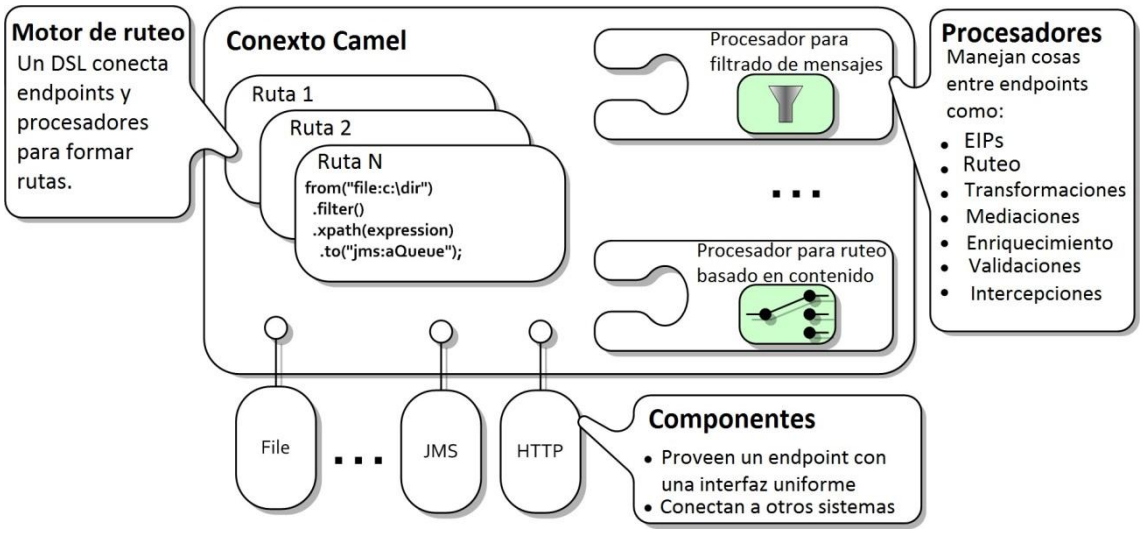


Figura 22 - Arquitectura a alto nivel de Apache Camel, extraído de [32].

Los componentes son el punto de extensión de Camel para agregar conectividad a otros sistemas. Para exponer estos componentes, los mismos proveen una interfaz de *endpoint*. Al utilizar URIs[13], es posible enviar y recibir mensajes a *endpoints* de una manera uniforme.

Los procesadores son usados para manipular y mediar mensajes entre *endpoints*. Todos los EIP están definidos como procesadores o un conjunto de ellos. Para interconectar procesadores y *endpoints*, Camel define múltiples lenguajes específicos del dominio en lenguajes de programación conocidos como Java, Scala, etc.

[12] http://camel.apache.org/

[13] http://www.w3.org/Addressing/

4 Solución Propuesta

En este capítulo se describen las principales características de la plataforma implementada. En la Sección 4.1 se presenta una descripción general de la solución y un diagrama conceptual de la misma. La Sección 4.2 presenta el modelo conceptual de la solución y la Sección 4.3 el esquema general de su arquitectura, basada en la solución propuesta en [1]. Por último, la Sección 4.4 describe los diferentes componentes de la plataforma y la Sección 4.5 la forma en que interactúan dichos componentes.

4.1 Descripción General

En esta sección se presenta una descripción general de la solución desarrollada en este proyecto, que como se mencionó anteriormente implementa la propuesta presentada en [1]. El objetivo de la plataforma es transformar servicios que no consideran el contexto del usuario, en servicios sensibles al contexto. La misma fue realizada utilizando SwitchYard como ESB y Drools Fusion como motor CEP.

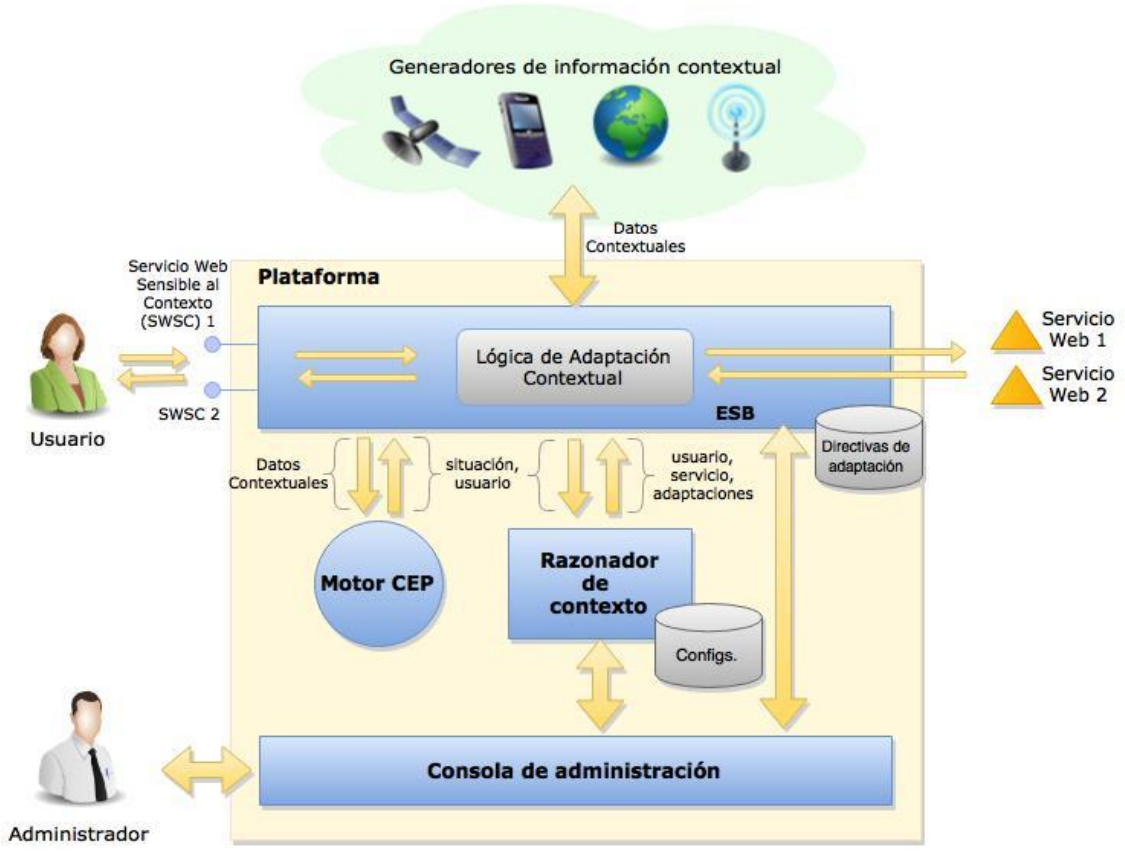


Figura 23 – Diagrama general de la plataforma.

En la Figura 23 se presenta un diagrama general de la plataforma, donde se pueden observar los cuatro componentes principales que la conforman y los agentes externos que interactúan con la misma. Existen dos tipos de usuarios que utilizan la plataforma:

Usuario y Administrador. El usuario interactúa de forma transparente con el ESB invocando los servicios que desea para obtener información adaptada a su contexto. El administrador es el encargado de configurar la plataforma para crear estos servicios.

Debido a la alta complejidad que implica la configuración del sistema, surge la necesidad de contar con un componente que ayude al administrador a gestionarla. Dicho componente se denomina Consola de Administración y permite al administrador configurar todos los aspectos de la plataforma. Esto incluye por ejemplo, configurar reglas que detecten situaciones y adaptaciones que deseen realizarse.

La Consola de Administración fue diseñada para brindarle al administrador una interfaz guiada y altamente configurable. El objetivo de este componente es que el administrador no tenga la necesidad de programar o configurar archivos de manera manual, sino que pueda realizar las configuraciones necesarias desde la consola. Esto también ayuda a evitar posibles errores en la configuración, ya que se realizan sugerencias y validaciones. La parte más compleja de la configuración del sistema es la definición de reglas para detectar situaciones, ya que es código de Drools. Para guiar esta configuración se utilizaron plantillas que sugieren gran parte del código, por lo que el administrador solo debe enfocarse en las condiciones que deben cumplirse para que sea detectada una situación.

Como se muestra en la Figura 23 se almacenan datos en dos componentes: el ESB y el Razonador de Contexto. En el ESB se guardan las directivas de adaptación asegurando su persistencia ante reinicios del sistema y se alimenta con la información enviada por el Razonador de Contexto. Por otro lado, en el Razonador de Contexto se almacena toda la configuración estática del sistema, como reglas, fuentes de contexto, datos contextuales, servicios, situaciones y adaptaciones.

4.2 Modelo Conceptual

En esta sección se presenta el modelo conceptual de la plataforma que abarca las entidades relevantes en la problemática y las relaciones entre las mismas. La Figura 24 presenta dicho modelo, donde se distinguen las entidades existentes en el modelo propuesto en [1] y las nuevas entidades definidas para la implementación de la plataforma.

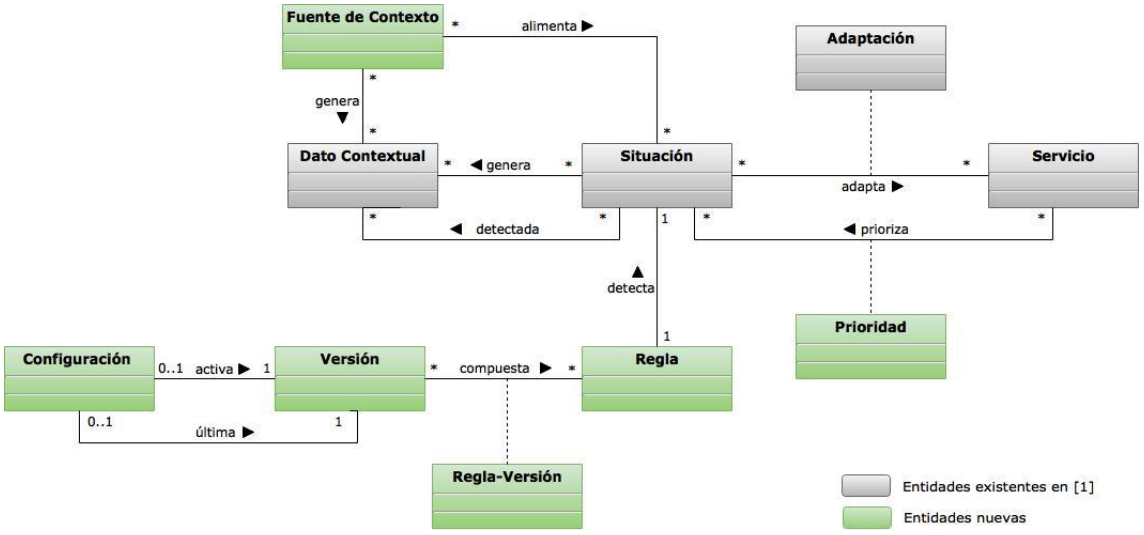


Figura 24 – Modelo conceptual.

En la Figura 24 se observa que las Fuentes de Contexto y las Situaciones generan Datos Contextuales, y que una Situación es detectada a través de los Datos Contextuales brindados por las Fuentes de Contexto. El concepto de Fuente de Contexto es un concepto que no existe en el modelo presentado en [1] pero que responde a la necesidad de identificar de dónde proviene cada Dato Contextual. Las Adaptaciones se definen para una Situación y Servicio determinado. Cada dupla <Situación, Servicio> tiene asociada una Prioridad, determinando qué Adaptaciones aplicar ante Situaciones que involucren un mismo Servicio.

En la parte inferior del diagrama aparecen otras entidades agregadas al modelo y vinculadas a la Regla. El concepto de Regla surge con la necesidad de determinar cómo es detectada una Situación. Una Regla detecta una y solo una Situación y puede estar asociada a muchas Versiones. Para cada asociación entre Regla y Versión existe la entidad Regla-Versión que contiene la regla de Drools correspondiente a cada Versión. Existe otra nueva entidad llamada Configuración que es la encargada de conocer la Versión de reglas activa y la última Versión de reglas creada. El motivo de este versionado de Reglas es que Drools no permite agregar nuevas reglas a las actualmente definidas, en cambio permite definir un nuevo conjunto de reglas y desplegarlas conjuntamente.

4.3 Arquitectura General

En esta sección se presenta la arquitectura general de la plataforma desarrollada, detallando los componentes que la conforman y sus funciones. En la Figura 25 se presenta un esquema de la arquitectura general, mostrando los componentes de la misma.

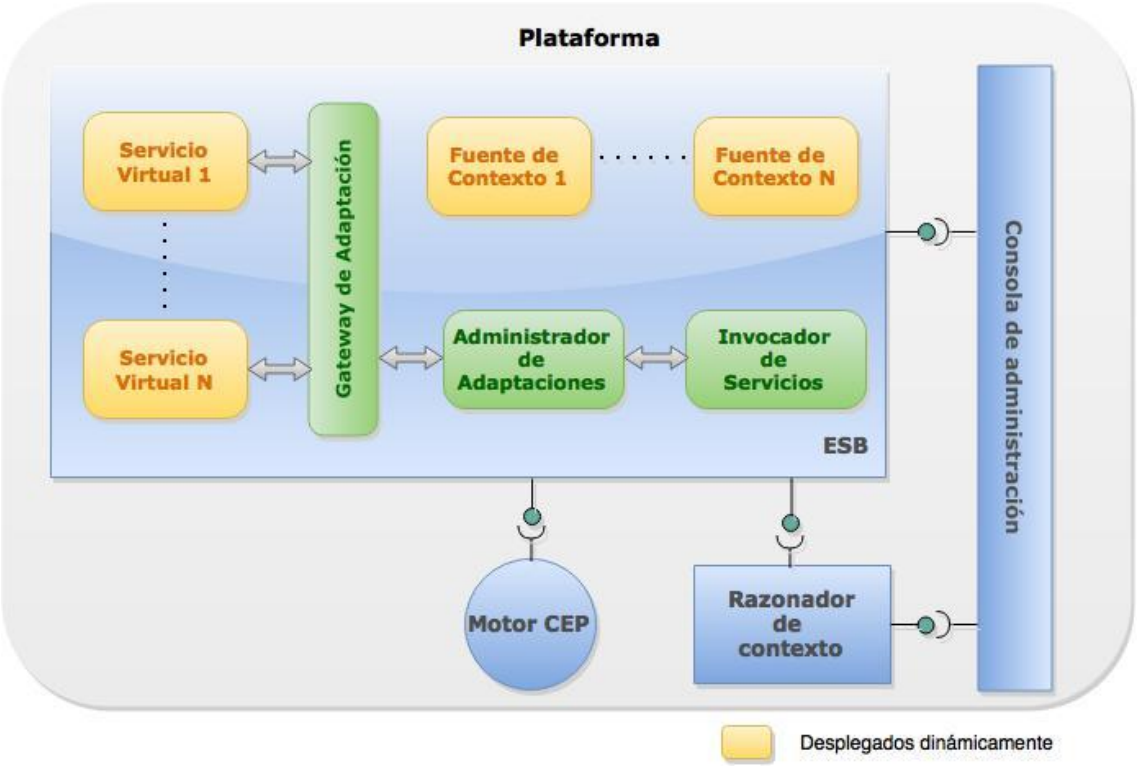


Figura 25 – Arquitectura general de la plataforma.

Los cuatro componentes principales de la plataforma son:

- **Motor CEP:** Se encarga de recibir la información de los distintos generadores de información contextual, detectar eventos complejos y notificarlos al Razonador de Contexto.
- **Razonador de Contexto:** Recibe las notificaciones del Motor CEP e informa al ESB las adaptaciones que se deben aplicar.
- **ESB:** Tiene la lógica de adaptación para generar los servicios sensibles al contexto.
- **Consola de Administración:** Brinda la interfaz gráfica para que el administrador configure y monitoree la plataforma.

Para definir la arquitectura se utilizó como base la definida en [1], añadiendo la Consola de Administración como nuevo componente. Se tomaron algunas decisiones particulares basadas en las características de las tecnologías utilizadas y la necesidad de cubrir los siguientes requerimientos:

- Publicación de nuevos *endpoints* que virtualicen los servicios que se quieren configurar en la plataforma en tiempo de ejecución.
- Posibilidad de especificar nuevas fuentes de contexto con distintos perfiles de comunicación en tiempo de ejecución.
- Soporte de distintos tipos de adaptaciones utilizando patrones de mediación.
- Posibilidad de definir en tiempo de ejecución un conjunto de adaptaciones para las situaciones y servicios.
- Posibilidad de invocar servicios externos que oficien de adaptaciones, siendo un potente punto de extensión.
- Simplicidad para configurar y monitorear el sistema abstrayendo al administrador de la implementación del mismo.
- Configuración de reglas en tiempo de ejecución para detectar nuevas situaciones.
- Persistencia del estado del sistema ante reinicios del mismo.

El ESB está compuesto por cinco subcomponentes, dos de los cuales en realidad son "clases de componentes" ya que son creados a demanda en tiempo de ejecución con diferentes configuraciones, como se detallará en las siguientes secciones. Dichos componentes son los Servicios Virtuales y las Fuentes de Contexto, y fueron diseñados de esta forma debido a ciertas limitantes de SwitchYard.

Para entender las siguientes secciones, es pertinente aclarar algunas particularidades de la solución realizada. Todos los componentes fueron implementados utilizando SwitchYard, a excepción de la Consola de Administración. Esto se debe a que se decidió aprovechar las facilidades para publicar y consumir servicios que ofrece SwitchYard nativamente. Una consecuencia de esto es que el componente denominado ESB sea en realidad un conjunto de subcomponentes SwitchYard, una práctica usual en plataformas implementadas en la mencionada tecnología.

4.4 Componentes

En esta sección se describen los componentes de la plataforma desarrollada, detallando las responsabilidades de los mismos y el motivo al cual responde su diseño. La sección se organiza en subsecciones, una por cada componente.

4.4.1 Fuentes de Contexto

Las Fuentes de Contexto son conectores del componente ESB que permiten recibir información de distintas fuentes externas que utilizan distintos formatos y protocolos de comunicación. Cada una expone un *endpoint* o consulta una fuente externa enviando la información obtenida al Motor CEP, lo cual se define como modos *listener* y *poller*

respectivamente. Estos dos modos aplican dos patrones EIP, el patrón Polling Consumer en las fuentes de tipo *poller* y el patrón Event-Driven Consumer para las de modo *listener*. En la Figura 26 se observa un ejemplo de Fuente de Contexto *poller* que consulta un servicio del estado del tiempo de Montevideo cada un minuto.

Figura 26 – Ejemplo de Fuente de Contexto modo poller.

Debido a que la información de dichas fuentes puede encontrarse en distintos formatos, se realiza un mapeo de los mensajes a un estándar para abstraer al Motor CEP del formato particular de cada mensaje, soportando formatos XML y JSON. Para este mapeo se aplica el patrón Messaging Mapper.

Una Fuente de Contexto es una aplicación SwitchYard que se crea y despliega dinámicamente desde la Consola de Administración a partir de parámetros configurables relacionados a la fuente externa asociada. Esto se debe a que SwitchYard no permite la publicación de *endpoints* dinámicamente, lo que es imprescindible para los requerimientos del proyecto. La forma de solucionar este problema es crear una nueva aplicación que publique un nuevo *endpoint* por cada fuente que se quiera agregar, y ésta envíe el mensaje al Motor CEP.

4.4.2 Motor CEP

Este componente contiene el motor CEP Drools Fusion. Se alimenta de los mensajes enviados por las Fuentes de Contexto y en base a reglas detecta eventos complejos. Un evento complejo detectado por el motor, es una situación que debe ser informada al Razonador de Contexto. En la Figura 27 se muestra un ejemplo simplificado de la regla que detecta la situación “InCity”, que infiere la ciudad donde se encuentra un usuario.

```
rule "UserPosition city detection"
when
    userPosition:UserPosition()
then
    String city = getCityFromCoord(userPosition.getLatitude(), userPosition.getLongitude());

    //Se cargan los datos contextuales
    contextualData.put(“city”, city);

    //Se informa al Razonador de Contexto
    situationDetected(userPosition.getUserId(), "InCity", contextualData);
end
```

Figura 27 – Ejemplo de regla en Drools.

Los datos contextuales tienen un papel fundamental en este componente, diferenciándolos en dos categorías: entrada y salida. Los datos contextuales de entrada, son los que provienen directamente de las Fuentes de Contexto y son utilizados para inferir los eventos complejos. En la Figura 27, los datos contextuales de entrada son la latitud y longitud. Por otro lado, los datos contextuales de salida son los que se notifican al Razonador de Contexto como información propia de la situación detectada. Éstos pueden coincidir con datos de entrada, ser calculados por las reglas en base a varios eventos de distintas fuentes, o hasta utilizar fuentes externas para ser obtenidos. Esto se debe a que las reglas contienen código Java embebido, brindando gran potencia para generar los datos de salida. En la Figura 27, se carga la ciudad como información contextual obtenida de una fuente externa que calcula la ciudad a partir de las coordenadas.

De esta forma al Razonador de Contexto se le informa el identificador del usuario, el nombre de la situación e información contextual de salida correspondiente a la situación detectada. Esto se muestra en la Figura 27 invocando a la función "situationDetected()".

4.4.3 Razonador de Contexto

En el Razonador de Contexto están definidas qué adaptaciones deben aplicarse cuando el Motor CEP informa una situación. Las adaptaciones son definidas por el administrador en la Consola de Administración y cada una hace referencia a un patrón de mediación. Éstas no se definen para un usuario en particular, se definen para una situación y un servicio determinado y aplican para cualquier usuario que esté en dicha situación.

Para una situación y un servicio determinado, se encuentra definido en el Razonador de Contexto un conjunto ordenado de adaptaciones, llamado itinerario. En la Tabla 5 se muestra un ejemplo de itinerario donde se puede observar que uno de los pasos es la invocación al servicio destino. Esto determina que las adaptaciones que se configuren antes de la invocación serán aplicadas al *request*, mientras que si son configuradas con un orden mayor a la de la invocación serán aplicadas al *response*.

Situación	Servicio	Tipo de adaptación	Adaptación	Orden	Información contextual
InCityRainig	getAttractions	Enriquecimiento (*Content Enrichment*)	Agregar información contextual (ciudad)	1	Ciudad donde se encuentra el usuario
InCityRainig	getAttractions	Invocación al servicio	No aplica	2	No aplica
InCityRainig	getAttractions	Filtrado (*Content Filter*)	Filtrar atracciones al aire libre	3	No aplica

Tabla 5 – Ejemplo de itinerario configurado en el Razonador de Contexto.

Anteriormente se describió la situación “InCity” que detecta la ciudad donde se encuentra un usuario. En la Tabla 5 se utiliza como ejemplo la situación “InCityRainig” que además implica que en esa ciudad esté lloviendo. Para esta situación y el servicio “getAttractions”, que devuelve atracciones, se encuentra configurado el itinerario de la Tabla 5. Como primer paso se agrega al *request* la ciudad donde se encuentra el usuario, luego se invoca el servicio y finalmente se filtran del *response* las atracciones al aire libre.

Para solucionar cómo agregar la información contextual al mensaje se utilizó un motor de *templates* que permite generar plantillas de transformaciones XSLT. De esta forma, cuando llegan datos contextuales se sustituyen las variables de la plantilla por éstos. Estos datos contextuales son definidos por el administrador y se generan al detectar una situación. Cuando una situación genera datos contextuales, por ejemplo al inferir que un usuario está en determinada ciudad, este dato se sustituye por la variable en la plantilla.

Cuando se notifica una situación, el Razonador de Contexto busca todos los servicios que se ven afectados por dicha situación y obtiene los itinerarios correspondientes. Para cada adaptación de los itinerarios añade la información contextual generada por la situación en caso que corresponda. Una vez que los mismos se encuentran listos para ser aplicados, el Razonador de Contexto notifica al Gateway de Adaptación los itinerarios para cada servicio y usuario. El Gateway de Adaptación necesita conocer el usuario para poder aplicar el itinerario cuando éste invoque un servicio afectado por la situación.

Se decidió que una situación tenga asociada un tiempo de vida, el cuál es especificado por el administrador y se asocia a los itinerarios generados por esta situación. Además a cada itinerario se le asignó una prioridad ya que un servicio puede tener configurado más de un itinerario dependiendo de las situaciones que lo afecten.

También se decidió que este componente tenga la responsabilidad de almacenar todas las configuraciones del sistema.

4.4.4 Servicios Virtuales

Son aplicaciones SwitchYard que se crean y despliegan dinámicamente desde la Consola de Administración a partir del WSDL del servicio que se desea invocar. Se asume que los servicios que se desean extender utilizan el protocolo SOAP para su comunicación. Un Servicio Virtual expone un *endpoint* con el mismo servicio al cual se desea invocar actuando como *proxy de* forma transparente al usuario. De esta manera se aplica el patrón Proxy Simple para cada servicio.

Para cada invocación, se envía el mensaje SOAP al Gateway de Adaptación junto a un encabezado que le indica cuál es el servicio final que se está invocando. En la Figura 28 se puede ver el flujo simplificado que sigue el mensaje.

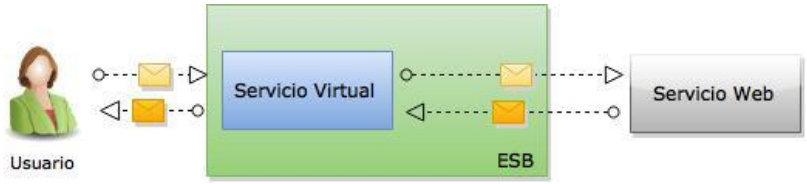


Figura 28 – Flujo del mensaje al invocar un servicio.

Al igual que con las Fuentes de Contexto, los Servicios Virtuales se implementan como aplicaciones independientes ya que SwitchYard no permite publicar *endpoints* dinámicamente, imposibilitando agregar nuevos servicios en tiempo de ejecución.

4.4.5 Gateway de Adaptación

Este componente intercepta todas las invocaciones que llegan a la plataforma por intermedio de los Servicio Virtuales aplicando el patrón Gateway como se observa en la Figura 29.

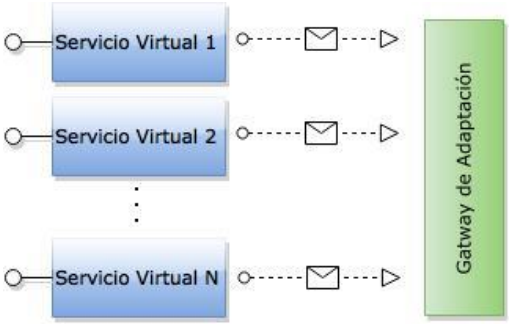


Figura 29 – Gateway de Adaptación interceptando mensajes.

Cuando una invocación llega al Gateway de Adaptación, se chequea si existe algún itinerario para el servicio y usuario correspondiente. Para identificar al usuario se asume que los mensajes SOAP enviados a la plataforma contienen el identificador del usuario mediante el estándar WS-Security.

Si existe un itinerario definido para un servicio y usuario, éste se añade al encabezado del mensaje para que sea utilizado por el Administrador de Adaptaciones para aplicar el ruteo basado en itinerario descrito anteriormente. En este momento queda determinado cuál es el camino que el mensaje debe recorrer en el ESB antes de devolver una respuesta al cliente. Además del itinerario, es necesario añadir al mensaje la información que necesita cada adaptación para ser aplicada. Por ejemplo, el filtrado de contenido necesita el XSLT que debe aplicar para realizar la transformación que hará el filtrado correspondiente.

En la Tabla 6 se muestran ejemplos de itinerarios definidos en el Gateway de Adaptación que se encuentran configurados para un usuario y servicio específicos. Cada adaptación del itinerario tiene la información que necesita, en el ejemplo cada enriquecimiento y filtrado tiene su XSLT asociado.

Usuario	Servicio	Itinerario	Prioridad	Expiración
Juan	getAttractions	1. Enriquecimiento: XSLT que añade la ciudad (Montevideo) 2. Invocación al servicio 3. Filtrado: XSLT que filtra atracciones al aire libre	1	2 horas
Juan	getAttractions	1. Enriquecimiento: XSLT que añade la ciudad (Artigas) 2. Invocación al servicio	2	6 horas
Ana	getAttractions	1. Invocación al servicio 2. Filtrado: XSLT que filtra descripciones largas	1	1 día

Tabla 6 – Ejemplo de itinerarios definidos en el Gateway de Adaptación.

Si no existe un itinerario definido, el componente añade al mensaje un itinerario que tiene como único paso la invocación al servicio. Si por el contrario, existe más de un itinerario definido, se utiliza el que tiene mayor prioridad. Esta prioridad fue añadida a la solución para contemplar este tipo de casos, donde existe más de un itinerario para el mismo servicio y usuario.

Cuando el Razonador de Contexto informa un nuevo itinerario para un servicio y usuario, se verifica si el mismo ya existe. En caso de existir se actualiza toda su información, incluyendo su tiempo de vida. Esta decisión se tomó debido a que cuando se detecta nuevamente una situación sus datos contextuales o la configuración de su itinerario pudo haber cambiado.

Además de lo expuesto, el Gateway de Adaptación tiene a su cargo la responsabilidad de eliminar los itinerarios cuyo tiempo de vida ha expirado. Para esto se definió un proceso que corre cada un minuto chequeando los itinerarios para eliminar los expirados.

Se decidió que los itinerarios fueran persistidos ya que el tiempo de vida de los mismos es indeterminado, pudiendo ser desde segundos hasta años. Por este motivo es deseable que ante reinicios o fallos del sistema no se pierda dicha información.

4.4.6 Administrador de Adaptaciones

El Administrador de Adaptaciones es el encargado de aplicar las adaptaciones que fueron configuradas en el Gateway de Adaptación utilizando el ruteo basado en itinerario y la información necesaria para aplicar cada adaptación. Como se mencionó anteriormente, toda esta información está configurada en el encabezado del mensaje. Las adaptaciones desarrolladas en el prototipo se describen a continuación:

- **Filtrado:** Con un XSLT se define el filtro a realizar a un mensaje, aplicando el patrón Content Filter.

- **Enriquecimiento:** Se define una plantilla que utilizando la información contextual genera un XSLT. Esto se realiza en el Razonador de Contexto, por lo que para el Administrador de Adaptaciones es simplemente una transformación XSLT. De esta manera se aplica el patrón Content Enrichment.
- **Retardo:** Se retarda el mensaje una cierta cantidad de milisegundos, la cual puede ser definida por el administrador en la Consola de Administración o informada por el Motor CEP como información contextual.
- **Ruteo Basado en Contenido:** Esta adaptación aplica el patrón Content Based Router. Utilizando XPath se puede definir una condición para de esta manera, basado en el contenido del mensaje, se decida qué adaptación se debe aplicar. Es la adaptación más compleja, ya que incluye cualquiera de las anteriores dependiendo del contenido del mensaje, incluso otro ruteo basado en contenido. Esto se debe a que el ruteo basado en contenido fue diseñado como un árbol de decisión, donde cada nodo puede ser una adaptación u otro ruteo basado en contenido. A continuación en la Figura 30 se puede observar una ilustración del árbol en el cual el nodo padre del árbol es siempre un XPath condicional.

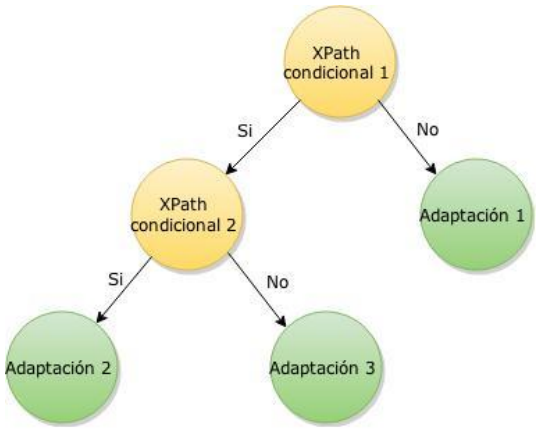


Figura 30 – Árbol utilizado para el ruteo basado en contenido.

- **Invocación Externa:** Se invoca a una URL externa por medio del Invocador de Servicios. Es una herramienta muy potente ya que permite extender las adaptaciones ya predefinidas en tiempo de ejecución por medio de un servicio externo que implemente la adaptación.

4.4.7 Invocador de Servicios

El Invocador de Servicios es el componente que se encarga de invocar los servicios externos a la aplicación. Normalmente uno de los pasos del itinerario es la invocación al servicio destino, y la misma se hace por medio de este componente. Además, por este componente se rutean las adaptaciones de tipo “Invocación Externa”.

Este componente obtiene del itinerario la URL del servicio destino y envía el mensaje a dicho servicio. La respuesta de este servicio es enviada al Administrador de Adaptaciones para que continúe con el siguiente paso del itinerario.

4.4.8 Consola de Administración

La Consola de Administración permite al administrador de la plataforma configurar y monitorear las distintas funcionalidades de la misma.

Brinda interfaces gráficas para:

- Configurar servicios a virtualizar.
- Agregar fuentes de contexto para alimentar el Motor CEP.
- Definir situaciones a detectar con sus reglas Drools correspondientes.
- Definir adaptaciones para los servicios y situaciones configurados.
- Gestión avanzada de reglas definidas, permitiendo su modificación y mejora.
- Monitoreo de la plataforma.

Fue diseñada para que el administrador pueda configurar cada aspecto de la plataforma de manera sencilla y flexible. Cuenta con gran variedad de validaciones para que el administrador no genere inconsistencias en la plataforma.

A continuación se describen en detalle las funcionalidades de la Consola de Administración:

Manejo de Fuentes de Contexto

Permite la creación de fuentes de contexto, los puntos de entrada de información contextual a la plataforma. El administrador puede definir nuevas fuentes a demanda y puede ver qué fuentes hay disponibles con los datos contextuales que se obtienen de cada una. Permite crear los dos tipos de fuente de contexto, *poller* y *listener,* para ambas permite establecer los dos tipos de mensajes que recibe, XML o JSON.

La creación de una nueva fuente de contexto, como se menciona anteriormente en la Sección 4.4.1, implica desplegar una nueva aplicación SwitchYard dinámicamente con los parámetros establecidos. Además esta configuración es mantenida en la base de datos con el objetivo de volver a generar y desplegar una fuente si se desea.

Manejo de Servicios

En esta sección de la consola es posible vincular nuevos servicios a la plataforma. Cuando el administrador vincula un nuevo servicio, la plataforma genera automáticamente un nuevo *endpoint* a partir del WSDL de éste. Esto implica la generación en tiempo de ejecución de una nueva aplicación SwitchYard, un proceso similar a la creación de las Fuentes de Contexto.

Manejo de Situaciones

Esta sección permite definir nuevas situaciones y datos contextuales. Los datos contextuales son los datos que se obtienen de las Fuentes de Contexto y los generados por las situaciones. Las situaciones deben vincularse a las fuentes de contexto ya que se detectan a través de datos generados por éstas.

En la Figura 31 se pueden observar las entidades involucradas al momento de definir una situación.

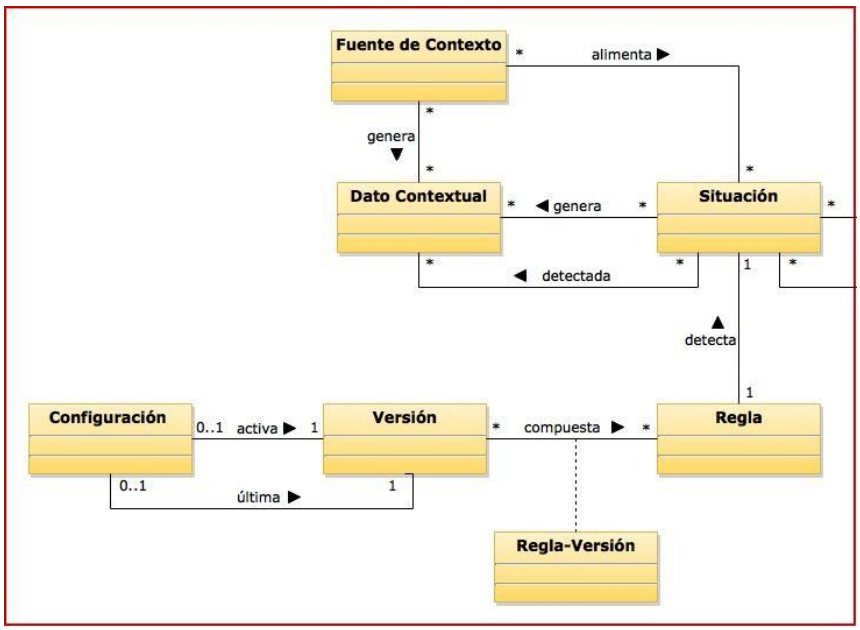


Figura 31 – Entidades involucradas en el manejo de situaciones.

Una situación se define en base a una o varias fuentes de contexto junto a los datos contextuales de cada una. A su vez, una situación puede generar datos contextuales ya que por ejemplo si se infiere que un usuario se encuentra en una ciudad determinada, esta ciudad es el dato contextual generado por la situación.

Al momento de crear una situación, también es necesario definir la regla de Drools Fusion que la detecta. Estas reglas son sugeridas automáticamente a partir de los datos cargados en el proceso de creación de la situación guiando al administrador en dicha tarea. Luego de definida, la regla es validada en busca de errores de compilación o inconsistencias respecto a los datos cargados en el proceso, indicándole al administrador los errores encontrados.

Manejo de Itinerarios

En esta sección se permite definir nuevos itinerarios para los servicios y situaciones disponibles. Un itinerario está constituido por una lista de adaptaciones. Cada adaptación requiere información específica, por ejemplo para la adaptación de enriquecimiento es necesario definir una plantilla donde se define la transformación XSLT y los datos necesarios que serán informados por el Motor CEP.

En la Figura 32 se observan las entidades involucradas para la creación de un itinerario.

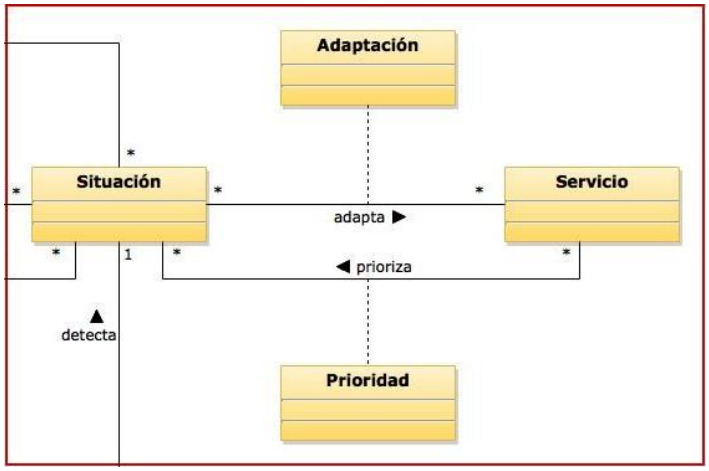


Figura 32 – Entidades involucradas en el manejo de itinerarios.

Manejo de Reglas

Esta sección se utiliza para el manejo avanzado de las reglas y versiones. Se permite crear nuevas versiones modificando las reglas ya existentes, de manera de corregir errores, realizar mejoras o cambios.

También es posible desplegar cualquier versión de reglas definida previamente. Se debe tener en cuenta que al desplegar una versión de reglas, las situaciones asociadas a reglas que no están activas no serán detectadas.

Monitoreo de configuraciones e itinerarios activos

En esta sección se permite visualizar los itinerarios activos para cada usuario y servicio. Estos itinerarios están definidos en el Gateway de Adaptación, y serán aplicados cuando el usuario invoque el servicio. Además es posible visualizar las situaciones junto con sus fuentes de contexto asociadas y los datos contextuales que generan. También se pueden ver los servicios configurados en el Razonador del contexto para los cuales están definidas situaciones e itinerarios.

4.5 Interacción entre Componentes

En esta sección se presenta la interacción entre los componentes y subcomponentes descritos en la sección anterior.

4.5.1 Recepción de Información Contextual

Como se explicó anteriormente, la información contextual proviene de las distintas fuentes externas y se envía al Motor CEP para ser procesada. En la Figura 33 se presenta un diagrama de secuencia donde se obtiene información de un generador de información contextual.

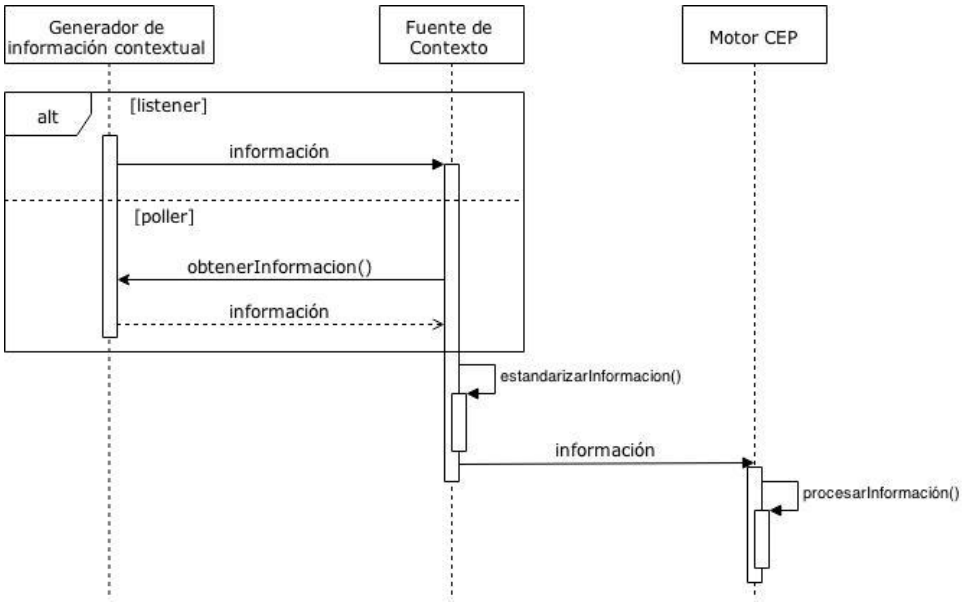


Figura 33 – Recepción de información contextual.

Existen dos posibilidades para obtener información, que la misma llegue de forma asincrónica al ESB por medio de la Fuente de Contexto o que la Fuente de Contexto invoque al generador cada cierto tiempo recibiendo la información en la respuesta.

Luego la Fuente de Contexto estandariza la información recibida para que el Motor CEP pueda abstraerse del formato en que llega y la envía al motor. El Motor CEP recibe la información y la procesa buscando detectar eventos complejos en base a las reglas configuradas.

4.5.2 Detección de Situaciones

Cuando se detecta una situación para un usuario se desencadenan una serie de interacciones entre tres componentes de la plataforma: el Motor CEP, el Razonador de Contexto y el ESB. En la Figura 34 se muestra un diagrama de secuencia detallando dicha interacción.

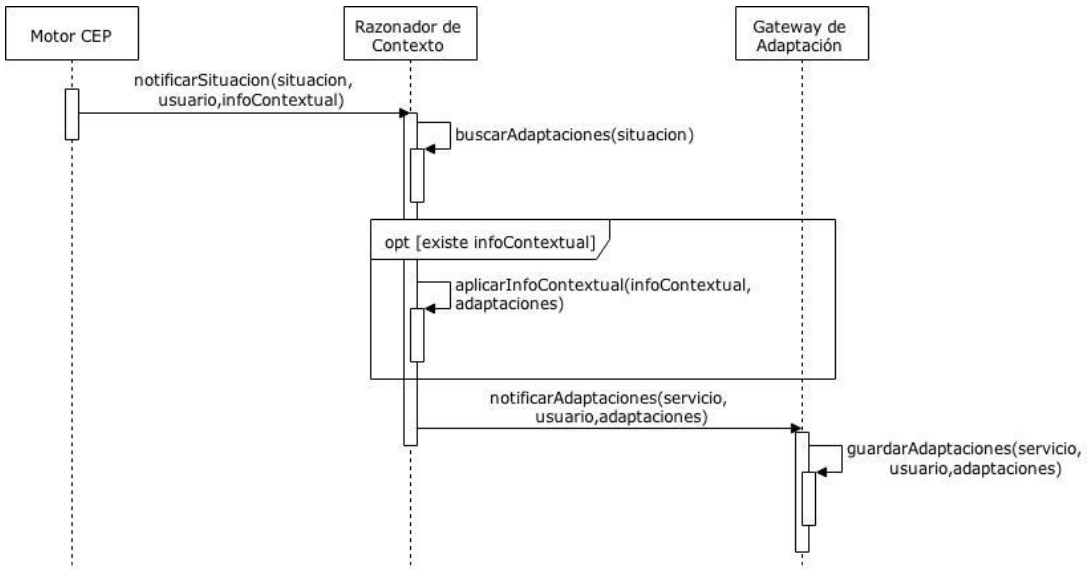


Figura 34 – Interacción ante la detección de una situación para un usuario

Si el Motor CEP detecta una situación para un usuario, éste le notifica al Razonador de Contexto la situación detectada, el usuario correspondiente y posible información contextual inferida por el motor. El Razonador de Contexto recibe la situación y busca los servicios afectados por dicha situación. Para cada servicio y situación busca las adaptaciones configuradas y si el Motor CEP envió información contextual, se aplica dicha información en las adaptaciones que correspondan. De esta forma, el Razonador de contexto arma un itinerario por cada servicio afectado para el usuario en cuestión y se lo notifica al Gateway de Adaptación. Este último componente, guarda para el usuario los itinerarios informados correspondientes a cada servicio, para así poder aplicar las adaptaciones que correspondan cuando llegue una invocación por parte del usuario a alguno de dichos servicios.

4.5.3 Interacción del Usuario

El usuario interactúa con la plataforma al invocar un servicio virtual correspondiente al servicio web que desea consumir. En la Figura 35 se muestra la interacción del usuario con la plataforma desde que invoca al servicio web hasta que se le devuelve una respuesta.

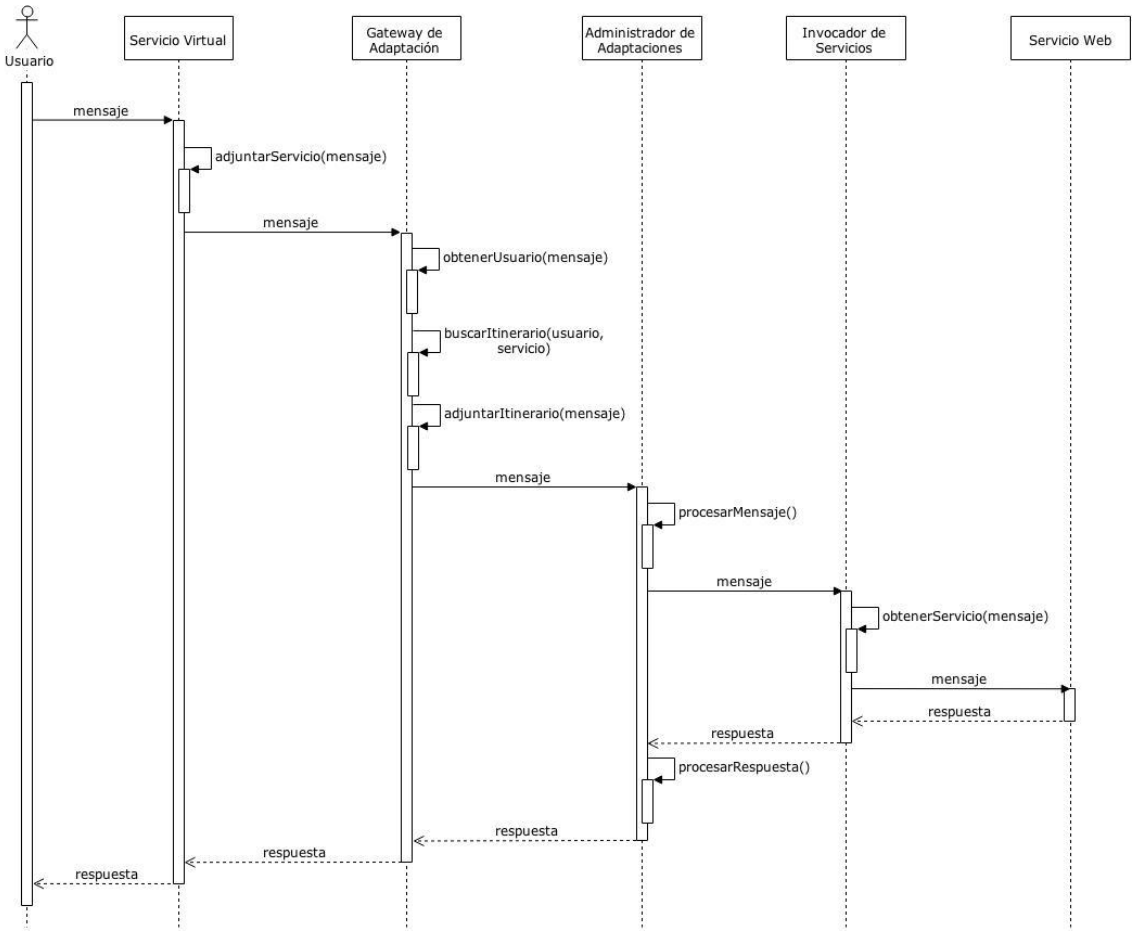


Figura 35 – Interacción del usuario con la plataforma.

Cada Servicio Virtual adjunta en el encabezado del mensaje el identificador del servicio que está siendo invocado para poder ser utilizado por el resto de los componentes. Luego el Gateway de Adaptación obtiene el usuario del cabezal WS Security y el servicio del encabezado agregado por el Servicio Virtual. Para el servicio y el usuario obtenidos, busca si existe algún itinerario configurado, y si es así, lo adjunta al mensaje para ser utilizado por el Administrador de Adaptaciones. Si no existe ningún itinerario, el Gateway arma un itinerario donde el único paso es invocar al servicio web.

Luego el Administrador de Adaptaciones obtiene del mensaje el itinerario y aplica el ruteo basado en itinerario, encaminando al mensaje por las distintas adaptaciones. Cuando uno de los pasos del itinerario es invocar al servicio web, el Invocador de Servicios obtiene del itinerario el servicio que se quiere consumir e invoca el servicio web. Luego, la respuesta es procesada por el Administrador de Adaptaciones aplicando

las adaptaciones que aún se encuentren en el itinerario. Cuando no quedan más pasos del itinerario para realizar, la respuesta es devuelta al usuario.

4.5.4 Interacción del Administrador

El administrador interactúa con la plataforma a través de la Consola de Administración. Como se explicó en las secciones anteriores, el administrador tiene la libertad de configurar en tiempo de ejecución múltiples aspectos de la plataforma. En la Figura 36 se muestra un ejemplo donde el administrador crea un nuevo itinerario.

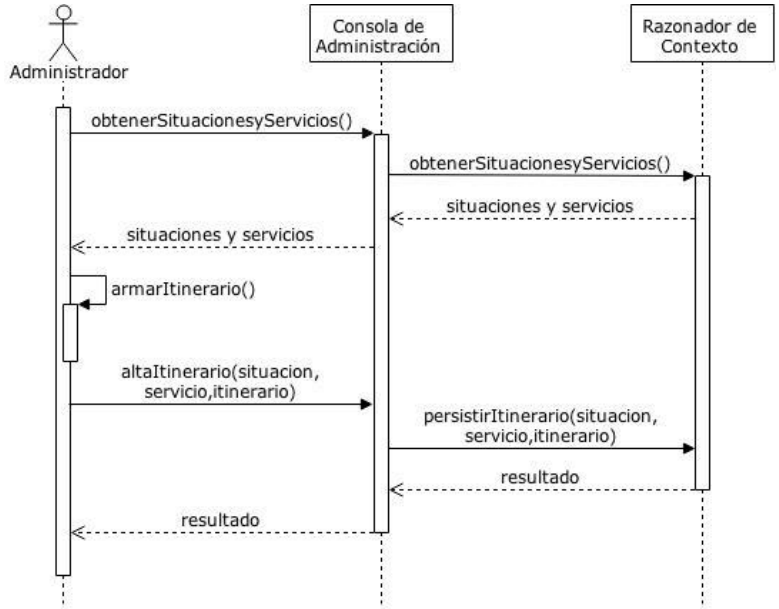


Figura 36 – Interacción del administrador con la Consola de Administración.

Como los itinerarios se configuran para una situación y servicios específicos, para poder dar de alta un nuevo itinerario, el administrador necesita conocer las situaciones y los servicios que se encuentran definidos en la plataforma. Para esto, la Consola de Administración solicita dichos datos al Razonador de Contexto. Luego de conocer la configuración existente, el administrador debe indicar a qué situación y servicio corresponde el itinerario que desea crear. Además se le presentan las posibles adaptaciones que puede elegir para armar el itinerario, donde debe cargar la información necesaria de cada adaptación. Una vez armado dicho itinerario, la Consola de Administración indica al Razonador de Contexto que persista la nueva configuración.

5 Implementación de la Solución

En este capítulo se presentan los distintos aspectos de la implementación de la solución propuesta. En la primera sección se muestra el diagrama de componentes y tecnologías y una descripción de las herramientas utilizadas. La Sección 5.2 presenta los detalles de implementación de los distintos componentes de la plataforma. Por último, la Sección 5.3 describe los problemas encontrados más importantes y que tuvieron impacto directo en el producto final obtenido.

5.1 Herramientas Utilizadas

En esta sección se presenta un diagrama de componentes y tecnologías junto a una breve descripción de las herramientas utilizadas. En la Figura 37 se puede observar el diagrama de componentes de la plataforma y las tecnologías más importantes utilizadas en cada uno.

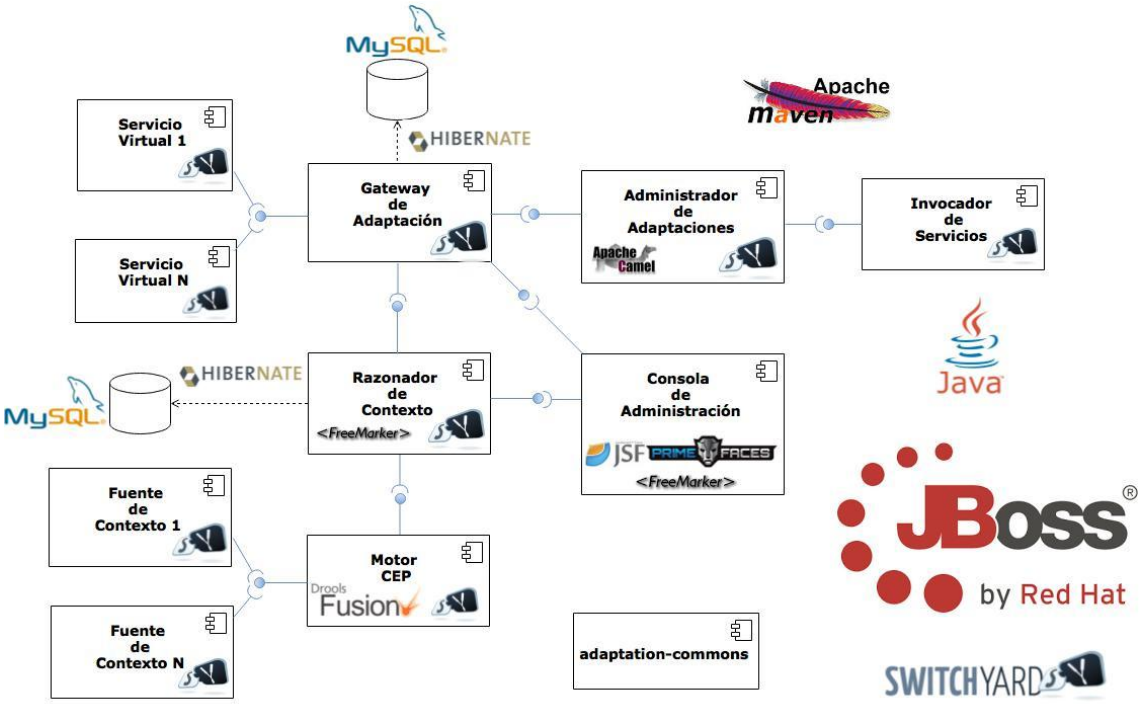


Figura 37 – Diagrama de componentes y tecnologías.

Todos los componentes son independientes, están escritos en lenguaje Java en su versión 1.7 y corren sobre un servidor JBoss EAP 6.1.

A continuación se describen las tecnologías mostradas en el diagrama de la Figura 37.

5.1.1 Apache Maven

Apache Maven [33] es una herramienta que permite la gestión de proyectos de *software*. Maven permite manejar las dependencias del proyecto, compilar, empaquetar y ejecutar *tests.* Un proyecto en Maven se define mediante el archivo

pom.xml, donde se especifican las configuraciones del mismo. En la solución se utilizó para el manejo de dependencias en todos los componentes.

5.1.2 MySQL

MySQL [34] es un motor de base de datos relacional de código abierto. Está basado en un lenguaje de consulta estructurado, del inglés *Structured Query Language* (SQL) y soporta múltiples plataformas. En la solución fue utilizado por los dos componentes que almacenan datos: el Razonador de Contexto y el Gateway de Adaptación. La versión utilizada fue la 5.5.

5.1.3 Hibernate

Hibernate [35] es una herramienta que se encarga del mapeo entre objetos y el modelo de base de datos. También facilita la realización de consultas y la recuperación de datos. La versión utilizada fue la 4.2.0 y fue utilizado por los dos componentes que almacenan datos para realizar el mapeo de entidades y acceder a los datos.

5.1.4 JavaServer Faces (JSF)

JavaServer Faces [36] es un *framework* y estándar utilizado para desarrollar interfaces de usuario del lado del servidor. Está basado en Java y la versión utilizada fue la 2.2. En el prototipo se utilizó para el desarrollo de la Consola de Administración.

Las principales características que proporciona son:

- Definición de interfaces de usuario mediante vistas que agrupan componentes gráficos.
- Conexión de componentes gráficos con datos de la aplicación mediante *beans*.
- Conversión de datos y validación automática de la entrada del usuario.
- Navegación entre vistas.

5.1.5 PrimeFaces

PrimeFaces [37] es una biblioteca de componentes de interfaz de usuario para JSF. Cuenta con una gran variedad de componentes que facilitan la creación de aplicaciones web. Su configuración es sencilla ya que simplemente consiste en agregar un único archivo *jar* como dependencia. La versión utilizada fue la 5.1 y se utilizó en conjunto con JSF para el desarrollo de la Consola de Administración.

5.1.6 FreeMarker

FreeMarker es un motor de plantillas, es decir, una herramienta que permite generar salidas de texto basado en plantillas [38]. Puede ser utilizado en todo tipo de aplicaciones, pero generalmente es usado para generar código, configurar archivos y mails. En el prototipo fue utilizado por la Consola de Administración y el Razonador de Contexto para generar diferentes configuraciones dinámicamente a partir de plantillas.

En la Figura 38 se muestra un ejemplo de cómo funciona FreeMarker, donde se puede ver la plantilla con una variable "*name*" y el objeto Java asociado a la plantilla que tiene el valor de dicha variable. Cuando actúa el motor, éste sustituye la variable de la plantilla por el valor que tiene cargado el objeto, generando la salida correspondiente.

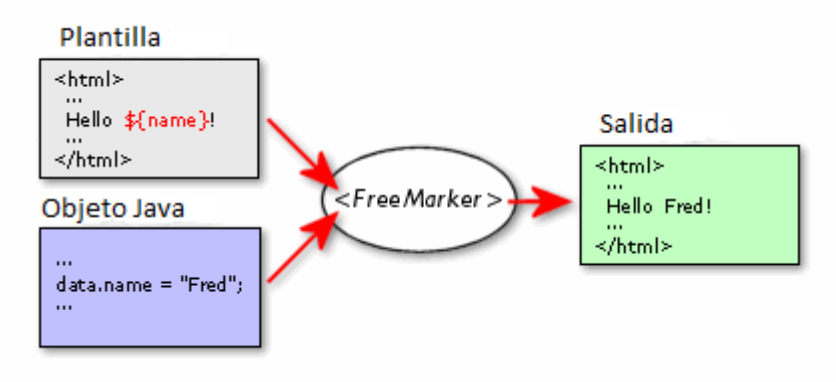


Figura 38 - Ejemplo de funcionamiento de FreeMarker.

5.2 Detalles de Implementación

En el capítulo 4 se describieron las aplicaciones que componen la solución desde el punto de vista de sus responsabilidades orientadas a la problemática planteada. En esta sección se hace foco en la implementación de cada componente.

5.2.1 Adaptation-Commons

Adaptation-Commons es una biblioteca que posee exclusivamente responsabilidad técnica. Está compuesta por las APIs necesarias para la comunicación entre aplicaciones y en general posee *Data Transfer Objects*[14] (DTOs). Fue desarrollada para no generar dependencias innecesarias entre las distintas aplicaciones SwitchYard a la hora de resolver la comunicación entre ellas, y a la vez lograr una comunicación bien definida de una forma sencilla. La biblioteca, además de los DTOs posee definidas constantes que son utilizadas por más de una aplicación, por lo cual se necesita una visibilidad que excede a la aplicación misma.

5.2.2 Fuentes de Contexto

Como se explicó en la Sección 4.4.1, las Fuentes de Contexto son aplicaciones SwitchYard que se crean y despliegan dinámicamente desde la Consola de Administración. Existen dos tipos de Fuentes de Contexto de acuerdo a su modo de obtener la información: *poller* y *listener*. Para explicar mejor la implementación, primero se explica el código de Fuentes de Contexto utilizado de base y luego cómo son generadas dinámicamente a partir de éste.

[14] Objetos que transportan datos, contienen datos relacionados y no contienen lógica de negocio.

5.2.2.1 Fuente de Contexto Listener

La Figura 39 presenta el diseño de este componente con el *plugin* para Eclipse utilizado.

Figura 39- Representación de una Fuente de Contexto Listener.

En la Tabla 7 se describe la responsabilidad de cada componente de una Fuente de Contexto Listener. Se utilizaron números para distinguir los componentes y para hacer más fácil el mapeo entre la representación presentada en la Figura 39 y los componentes descritos en la Tabla 7.

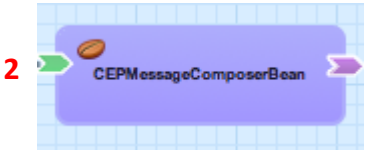
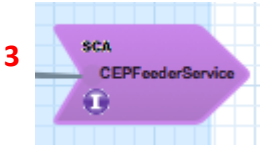
Componente	Representación gráfica	Descripción
CEPMessageComposer		Expone un *endpoint* HTTP mediante un *binding* HTTP.
CEPMessageComposerBean		Estandariza el mensaje que puede estar en formato JSON o XML a un mapa de Java, utilizando las bibliotecas XStream[15] y Jackson[16].
CEPFeederService		Envía al motor CEP el mensaje estandarizado utilizando un *binding* SCA.

Tabla 7 – Componentes de una Fuente de Contexto Listener.

En la Figura 40 se muestra un ejemplo donde se estandariza un mensaje en formato XML, convirtiéndolo a un mapa de Java.

Figura 40 – Estandarización de un mensaje en formato XML.

[15] http://mvnrepository.com/artifact/xstream/xstream

[16] http://mvnrepository.com/artifact/org.codehaus.jackson

Al Motor CEP se le envía el mensaje estandarizado y el nombre del evento. Este nombre es utilizado por las reglas configuradas en el Motor CEP para identificar el origen de la notificación y cómo ésta debe ser interpretada.

5.2.2.2 Fuente de Contexto Poller

La Figura 41 muestra el diseño en SwitchYard de una Fuente de Contexto en su modo *poller*.

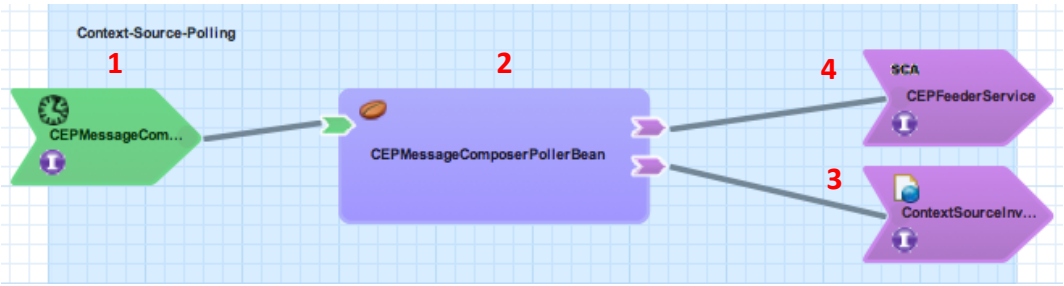


Figura 41 – Representación de una Fuente de Contexto Poller.

En la Tabla 8 se describe la responsabilidad de cada componente de una Fuente de Contexto Poller.

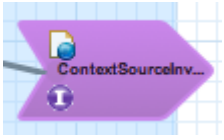
Componente	Representación gráfica	Descripción
CEPMessageComposerPoller	1 	Posee un *binding scheduling* que permite configurar un *cron*[17] para determinar la frecuencia de la invocación.
CEPMessageComposerPollerBean	2 	Estandariza la respuesta de la invocación al igual que en el modo *listener*.
ContextSourceInvoker	3 	Provee un *binding* HTTP para realizar la invocación.
CEPFeederService	4 	Envía al motor CEP el mansaje estandarizado utilizando un *binding* SCA.

Tabla 8 – Componentes de una Fuente de Contexto Poller

[17] http://iie.fing.edu.uy/ense/asign/admunix/cron.htm

5.2.2.3 Generar Fuentes de Contexto Dinámicamente

Para generar las Fuentes de Contexto dinámicamente se crearon dos "plantillas de aplicación", una para cada modo anteriormente descrito, *poller* y *listener*. Las plantillas se crearon a partir de un archivo JAR generado al empaquetar una Fuente de Contexto de ejemplo. A partir de este JAR, se sustituyeron los valores particulares en los archivos del empaquetado por variables de FreeMarker que luego se sustituirán. En el Apéndice 2 se encuentra una descripción más detallada de este proceso.

5.2.3 Motor CEP

En la Figura 42 se muestra el diseño del componente CEP en el *plugin* para Eclipse utilizado.

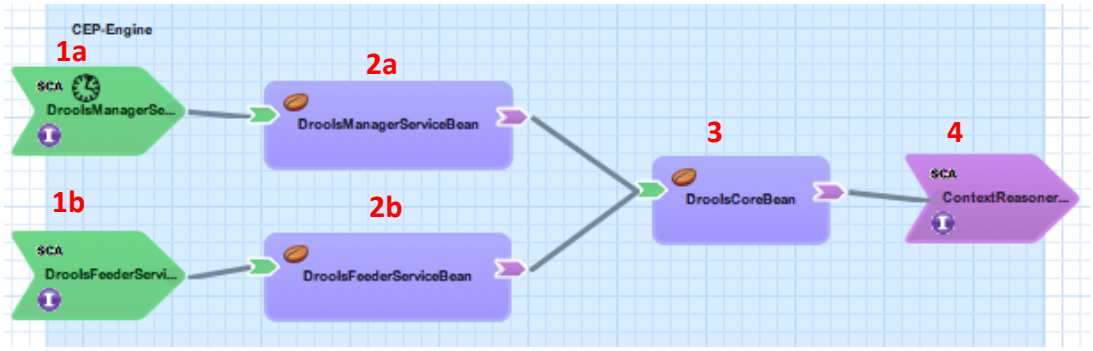


Figura 42 – Representación del componente CEP.

En la Tabla 9 y Tabla 10 se presenta una descripción de cada componente que aparece en la Figura 42.

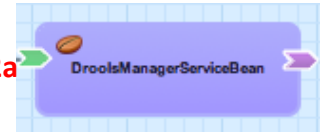
Componente	Representación gráfica	Descripción
DroolsManagerService		• Provee un *binding* SCA para la gestión del motor de reglas. • Posee un *binding scheduling* que invoca automáticamente al servicio que despliega la versión activa cuando se inicia el motor, debido al problema descrito en la Sección 5.3.2.
DroolsManagerServiceBean		Contiene las operaciones para gestionar el motor Drools, permitiendo: • verificar la compilación de una versión. • desplegar una versión en Drools. • inicializar el motor.

Tabla 9 – Componentes del Motor CEP (parte 1).

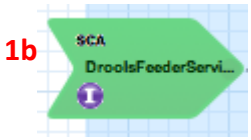
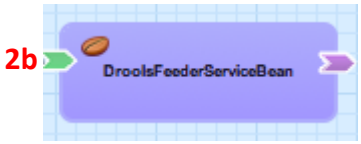
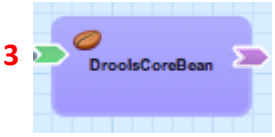
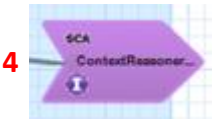
Componente	Representación gráfica	Descripción
DroolsFeederService		Provee un *binding* SCA para recibir eventos desde las Fuentes de Contexto.
DroolsFeederServiceBean		Contiene las operaciones para alimentar el motor Drools.
DroolsCoreBean		Encapsula toda la lógica necesaria para interactuar con Drools Fusion.
ContextReasonerService		Obtiene del Razonador de Contexto las reglas de la versión activa mediante un *binding* SCA.

Tabla 10 – Componentes del Motor CEP (parte 2).

En el Apéndice 3 se describe con más detalle algunos aspectos de la implementación del Motor CEP.

5.2.4 Razonador de Contexto

El Razonador de Contexto es una aplicación SwitchYard con una base de datos MySQL que persiste todas las configuraciones de la plataforma que se realizan desde la Consola de Administración. Esta aplicación se comunica con tres componentes: el Motor CEP, el Gateway de Adaptación y la Consola de Administración. Por este motivo se decidió utilizar SwitchYard para su implementación, ya que los *bindings* SCA facilitan la comunicación con dichos componentes.

En las siguientes sub-secciones se detallan las dos responsabilidades del Razonador de Contexto: la recepción de situaciones notificadas por el Motor CEP y la configuración de funcionalidades de la plataforma.

5.2.4.1 Recepción de Situaciones

Una de las responsabilidades principales del Razonador de Contexto es la recepción de situaciones. En la Figura 43 se presentan los componentes involucrados en dicha recepción.

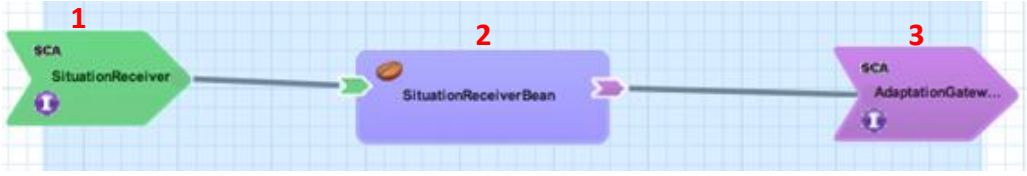


Figura 43 - Componentes involucrados en la recepción de situaciones.

En la Tabla 11 se describen los componentes del Razonador de Contexto mostrados en la Figura 43.

Componente	Representación gráfica	Descripción
SituationReceiver		Expone un *endpoint* con un *binding* SCA para recibir las situaciones detectadas por el Motor CEP.
SituationReceiverBean		Encargado de buscar en la base de datos los servicios afectados por la situación recibida y armar los itinerarios.
AdaptationGatewayService		Envía todos los itinerarios al Gateway de Adaptación.

Tabla 11 – Descripción de componentes involucrados en la recepción de situaciones.

El Razonador de Contexto recibe las situaciones detectadas por el Motor CEP. Cuando se recibe una situación, se arma un itinerario para cada servicio con las adaptaciones previamente configuradas, respetando el orden que se le haya otorgado a cada adaptación. Las adaptaciones que requieren información contextual como el enriquecimiento, utilizan los datos contextuales enviados para remplazar las variables de FreeMarker.

5.2.4.2 Configuración de Funcionalidades

Como se mencionó anteriormente, el Razonador de Contexto permite la configuración de las distintas funcionalidades de la plataforma. Sus funcionalidades principales son:

- Dar de alta servicios, situaciones, fuentes de contexto, datos contextuales e itinerarios.
- Obtener el conjunto de servicios virtuales, situaciones, fuentes de contexto y datos contextuales dados de alta desde la Consola de Administración.
- Obtener todos los servicios, las situaciones que lo afectan y las adaptaciones que hayan sido configuradas para cada servicio y adaptación.
- Obtener las fuentes de contexto y las situaciones junto a los datos contextuales que cada una genera.
- Obtener las reglas que contiene cada versión, crear nuevas versiones con previa compilación en el Motor CEP y desplegar en el motor la versión que se desee.

En el Apéndice 4 se detalla la implementación que permite realizar estas configuraciones.

5.2.5 Servicios Virtuales

Cada Servicio Virtual provee un *endpoint* que expone un nuevo WSDL generado a partir del indicado al momento de crear dicho servicio. En la Figura 44 se muestra la representación de un Servicio Virtual en el editor gráfico de SwitchYard.

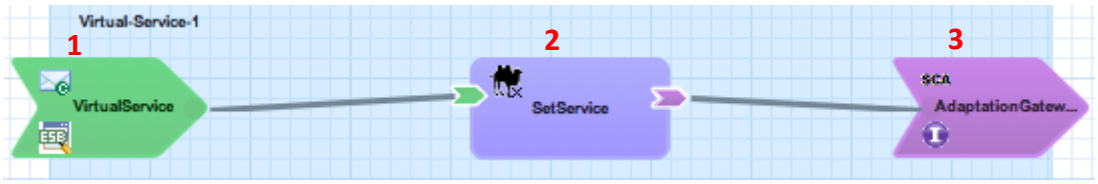


Figura 44 – Representación de un Servicio Virtual.

En la Tabla 12 se describen las responsabilidades de cada componente de un Servicio Virtual.

Componente	Representación gráfica	Descripción
VirtualService		Expone un *endpoint* con un *binding* SOAP.
SetService		Agrega dos encabezados al mensaje: nombre y URL del servicio que está siendo invocado, como se explica en el Apéndice 5Apéndice 2.
AdaptationGatewayService		Envía el mensaje al Gateway de Adaptación mediante un *binding* SCA.

Tabla 12 – Componentes de un Servicio Virtual.

Al igual que las Fuentes de Contexto, los Servicios Virtuales se crean y despliegan dinámicamente. La estrategia utilizada para resolver este problema es exactamente igual a la utilizada para las Fuentes de Contexto. En el Apéndice 2 se encuentra una descripción más detallada de este proceso.

5.2.6 Gateway de Adaptación

El Gateway de Adaptación es una aplicación SwitchYard que tiene una base de datos MySQL donde mantiene los itinerarios activos, por usuario y servicio, notificados por el Razonador de Contexto. Esta aplicación se comunica con múltiples aplicaciones: los Servicios Virtuales, el Administrador de Adaptaciones, el Razonador de Contexto y la Consola de Administración.

En las siguientes sub-secciones se detallan las dos responsabilidades del Gateway de Adaptación: el procesamiento de mensajes enviados por los Servicios Virtuales y la gestión de los itinerarios notificados por el Razonador de Contexto.

5.2.6.1 Procesamiento de Mensajes

Todos los Servicios Virtuales envían sus mensajes al Gateway de Adaptación y éste los procesa en busca de itinerarios que deben ser aplicados. En la Figura 45 se muestran los componentes involucrados en dicho procesamiento.

Figura 45 – Componentes involucrados en el procesamiento de mensajes.

En la Tabla 13 se describen las responsabilidades de los componentes mostrados en la Figura 45.

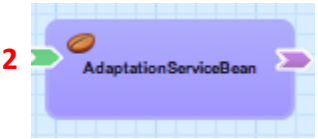
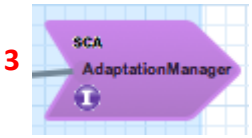
Componente	Representación gráfica	Descripción
AdaptationService		Expone un *endpoint* con un *binding* SCA para recibir mensajes de los distintos Servicios Virtuales.
AdaptationServiceBean		Busca si existe un itinerario configurado para el mensaje y se lo adjunta.
AdaptationManagerService		Envía el mensaje al Administrador de Adaptaciones mediante un *binding* SCA.

Tabla 13 – Descripción de los componentes involucrados en el procesamiento de mensajes.

Cuando llega un mensaje, se busca en éste: el identificador del usuario en el cabezal *Username Token* de WS-Security, el identificador del servicio en el encabezado añadido por el Servicio Virtual como se muestra en el Apéndice 5 y la operación que está siendo invocada por el mensaje. Es necesario tener en cuenta la operación debido a que las adaptaciones no solo están configuradas a nivel de servicios, sino también a nivel de operaciones del servicio SOAP.

Luego de tener identificado el usuario, servicio y operación, se busca en la base de datos del Gateway de Adaptación el itinerario de mayor prioridad no expirado. En caso de encontrar un itinerario, se construye una estructura llamada “AdaptedMessage” que posee el mensaje, el itinerario y una directiva de adaptación, como se puede observar en la Figura 46. El itinerario es utilizado por el ruteo basado en itinerario e indica el camino que debe tomar el mensaje en el Administrador de Adaptaciones. Cada paso del itinerario, indica una adaptación que debe ser aplicada. La directiva de adaptación es

una lista que posee la información necesaria para aplicar la adaptación que corresponde en cada paso.

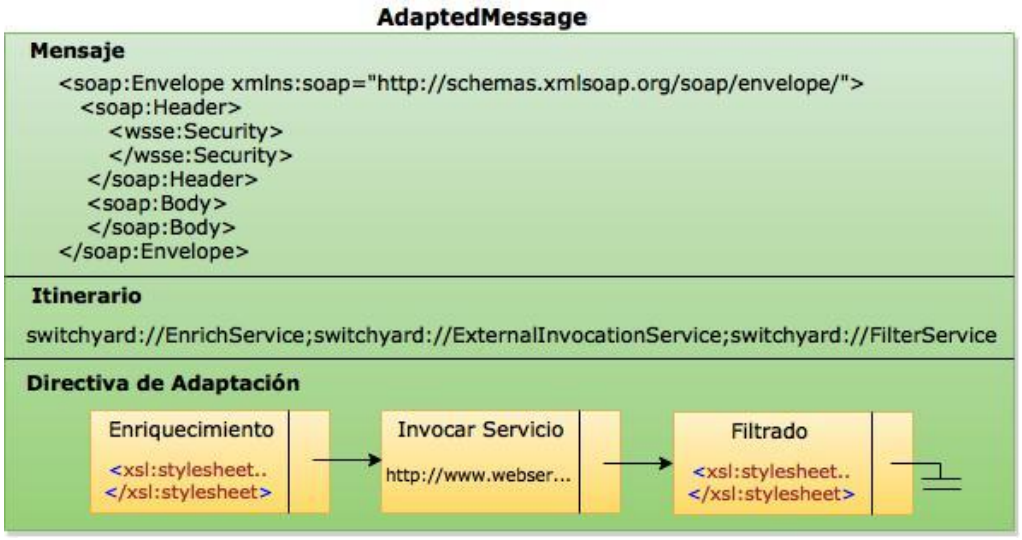


Figura 46 – Composición de la estructura AdaptedMessage.

Si no existe un itinerario configurado en el Gateway, se crea un "AdaptedMessage" con un itinerario de un único paso: invocar al servicio destino.

5.2.6.2 Gestión de Itinerarios

Otra de las responsabilidades del Gateway de Adaptación es la gestión de itinerarios. Estos itinerarios son notificados por el Razonador de Contexto y necesitan ser purgados una vez que hayan expirado. En la Figura 47 se muestran los componentes involucrados en dicha gestión.

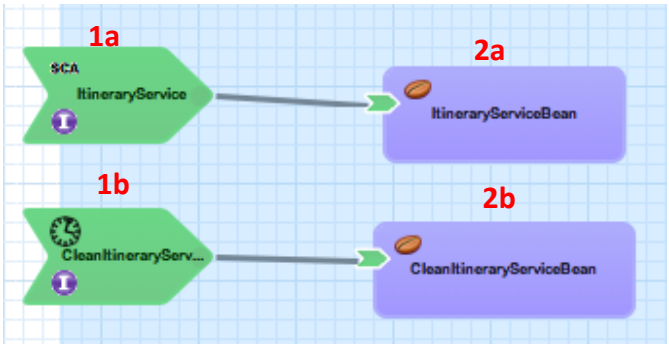


Figura 47 – Componentes involucrados en la gestión de itinerarios.

En la Tabla 14 se detallan las responsabilidades de los componentes mostrados en la Figura 47.

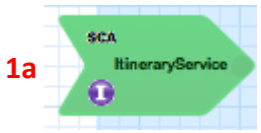
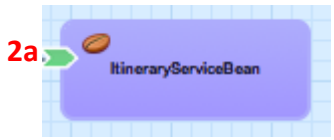
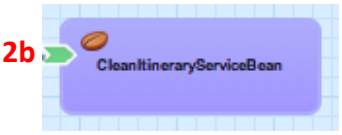
Componente	Representación gráfica	Descripción
ItineraryService		Expone un *endpoint* con un *binding* SCA para: • Recibir itinerarios notificados por el Razonador de Contexto. • Brindar a la Consola de Administración todos los itinerarios activos configurados.
ItineraryServiceBean		• Persiste todos los itinerarios recibidos. • Obtiene de la base de datos los itinerarios activos.
CleanItineraryService		Posee un *binding scheduling* con un cron que corre cada un minuto invocando al *bean* "CleanItineraryServiceBean".
CleanItineraryServiceBean		Elimina itinerarios cuya fecha de expiración sea menor a la fecha actual.

Tabla 14 – Descripción de los componentes involucrados en la gestión de itinerarios.

5.2.7 Administrador de Adaptaciones

El Administrador de Adaptaciones es una aplicación SwitchYard que utiliza Apache Camel para la implementación de los EIP. Este componente se comunica con el Gateway de Adaptación, que es quien le envía el mensaje con su itinerario, y con el Invocador de Servicios para realizar invocaciones externas. En la Figura 48 se muestra la representación del Administrador de Adaptaciones en el editor gráfico de SwitchYard.

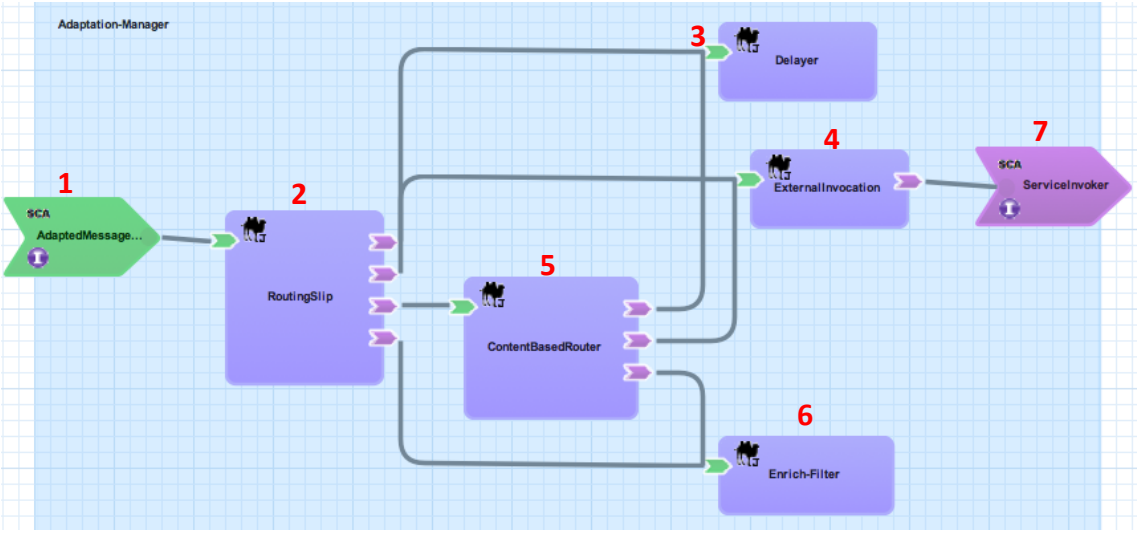


Figura 48 – Representación del Administrador de Adaptaciones.

En la Tabla 15 se detallan las responsabilidades de cada componente mostrado en la Figura 48.

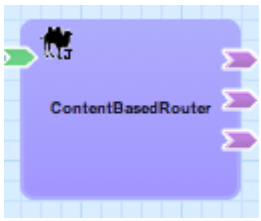
Componente	Representación gráfica	Descripción
AdaptedMessageService	1 	Expone un *endpoint* con un *binding* SCA para recibir los mensajes enviados por el Gateway de Adaptación.
RoutingSlip	2 	Aplica el ruteo basado en itinerario, detallado en el Apéndice 6, utilizando el itinerario cargado en el "AdaptedMessage".
Delayer	3 	Retrasa el envío de mensajes la cantidad de milisegundos especificada en la directiva de adaptación.
External Invocation	4 	Envía el mensaje al componente "ServiceInvoker", tanto para llamar al servicio destino del mensaje como para llamar a un servicio que actué de adaptación.
ContentBasedRouter	5 	Aplica el ruteo basado en contenido detallado en el Apéndice 6.
Enrich-Filter	6 	Aplica transformaciones XSLT tanto para añadir información al mensaje, como para filtrar datos del mismo.
ServiceInvoker	7 	Envía el mensaje al Invocador de Servicios mediante un *binding* SCA.

Tabla 15 – Componentes del Administrador de Adaptaciones.

5.2.8 Invocador de Servicios

El Invocador de Servicios es un componente SwitchYard encargado de realizar la invocación al servicio web indicado por el Administrador de Adaptaciones. En la Figura 49 se muestra la representación del Invocador de Servicios en el editor gráfico de SwitchYard.

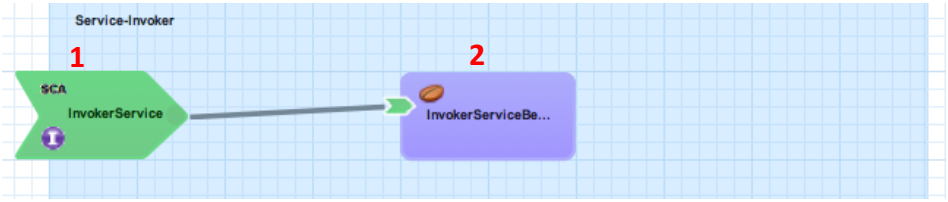


Figura 49 – Representación del Invocador de Servicios.

En la Tabla 16 se describen las responsabilidades de cada componente del Invocador de Servicios.

Componente	Representación gráfica	Descripción
InvokerService		Expone un *binding* SCA para recibir del Administrador de Adaptaciones el mensaje y la URL a invocar.
InvokerServiceBean		Realiza el POST HTTP a la URL indicada y la respuesta del servicio es devuelta al Administrador de Adaptaciones.

Tabla 16 – Componentes del Invocador de Servicios.

5.2.9 Consola de Administración

La Consola de Administración fue desarrollada utilizando el *framework* Java Server Faces y componentes de PrimeFaces. Como se explica anteriormente, la Consola de Administración no fue implementada como una aplicación SwitchYard debido a que presenta problemas de compatibilidad para generar proyectos web como se detalla en la Sección 5.3.5. En consecuencia, fue implementada como un proyecto web y la comunicación con los demás componentes SwitchYard se realiza a través del cliente de invocación remota "RemoteInvoker"[18] que brinda SwitchYard.

En las siguientes secciones se detallan las diferentes funciones que brinda la consola, agrupadas de acuerdo a su disposición.

[18] https://docs.jboss.org/author/display/SWITCHYARD/Remote+Invoker

5.2.9.1 Manejo de Fuentes de Contexto

Comprende dos funcionalidades vinculadas a las Fuentes de Contexto, crearlas y visualizarlas. Permite crear nuevas Fuentes de Contexto definiendo su nombre, tipo y el formato de mensaje. En caso que el tipo de fuente sea *poller*, es necesario especificar adicionalmente un *cron*, y la URL a la cual consultar. Los tipos de mensaje soportados para las fuentes de contexto son XML o JSON.

Además es posible visualizar los datos contextuales asociados a cada Fuente de Contexto como se observa en la Figura 50. Cada Fuente de Contexto tiene un conjunto de datos definidos por el administrador de manera de ver qué datos genera cada una. Los datos contextuales se vinculan a la fuente cuando se crea una situación.

Fuentes de Contexto	
Nombre	Tipo
UserLocation	Fuente de Contexto - listener
latitude	Dato Contextual
longitude	Dato Contextual
userId	Dato Contextual
CityRaining	Fuente de Contexto - polling
raining	Dato Contextual
city	Dato Contextual

Figura 50 – Fuentes de Contexto y sus Datos Contextuales.

5.2.9.2 Manejo de Servicios

Esta pantalla permite vincular nuevos servicios a la plataforma definiendo el nombre y la URL del servicio. Una vez seleccionada la URL del servicio, la plataforma obtiene del WSDL la lista las operaciones disponibles pudiendo seleccionar el conjunto de operaciones que se deseen. Para obtener estos datos se utilizó la biblioteca predic8.wsdl[19] que cuenta con un *parser* para analizar los WSDL.

5.2.9.3 Manejo de Situaciones

En esta sección es posible definir las situaciones y sus datos contextuales asociados. Para que la creación de las situaciones sea más sencilla se tiene una serie de tres pantallas. En la primera se define el nombre de la situación, una descripción de la misma y su duración en milisegundos. En la segunda se establece qué fuentes de contexto alimentan la situación, asociando también las fuentes de contexto con los datos contextuales de entrada, así como los datos contextuales de salida que genera la situación. En la Figura 51 se muestra cómo se debe seleccionar la fuente de contexto de entrada vinculando además los datos contextuales que genera, en este caso latitud, longitud y usuario. Además se define la ciudad como dato de salida de la regla.

[19] http://www.membrane-soa.org/soa-model-doc/1.2/manipulate-wsdl-java-api.htm

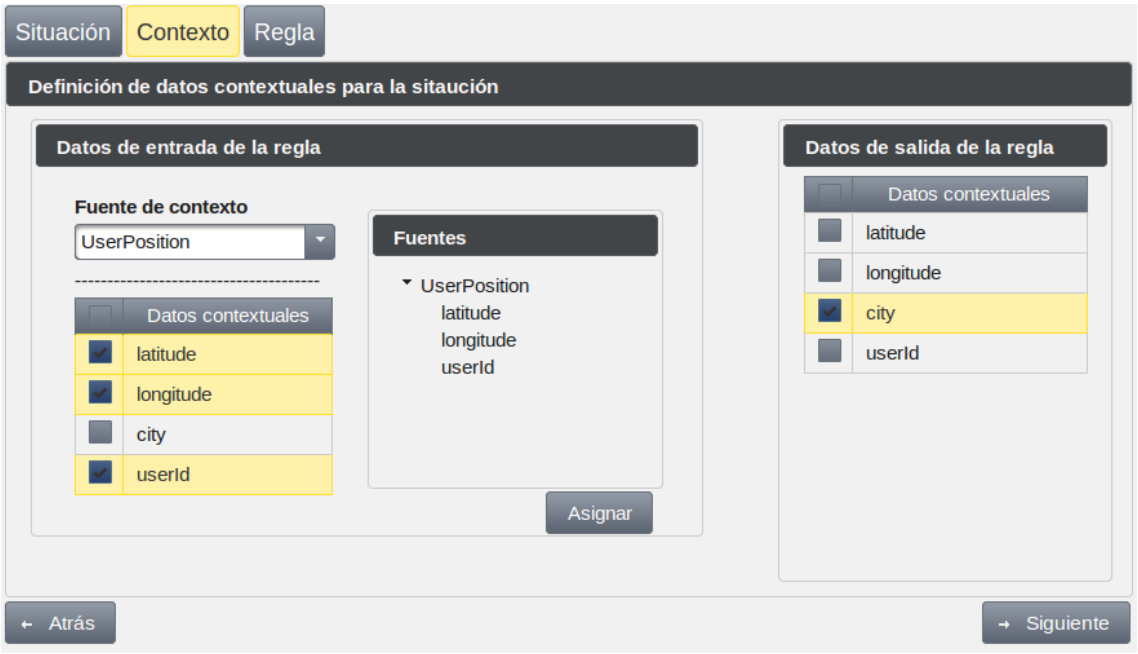


Figura 51 – Pantalla de definición de datos contextuales para una situación.

En la tercer y última pantalla se debe cargar el archivo con las reglas de Drools necesarias para detectar la situación que se está definiendo. Para cargar la regla y ayudar a que la configuración del sistema sea más intuitiva, se genera una sugerencia de regla creada a partir de una plantilla FreeMarker y los datos que se fueron cargando en las pantallas anteriores. En el Apéndice 7 se detalla la generación de dicha regla. De esta forma el administrador cuenta con un borrador de la regla, que muestra cómo obtener los datos de entrada para utilizarlos y cómo informar al Razonador de Contexto los datos de salida.

5.2.9.4 Manejo de Itinerarios

En esta pantalla es posible definir los itinerarios para los distintos servicios y situaciones. Lo primero a definir es el servicio, que se selecciona de la lista de servicios configurados. Una vez seleccionado el servicio, se despliegan las situaciones asociadas a este servicio. El administrador debe elegir una situación y asignar una prioridad para el itinerario a crear.

Luego de especificados el servicio, situación y prioridad, se agregan las distintas adaptaciones, definiendo para cada una la configuración correspondiente. Para cada tipo de adaptación se tiene una validación específica:

- **Delay**: Se verifica sea un número entero positivo.
- **Filter**: Utilizando la función "newTemplates" de la biblioteca javax.xml.transform.TransformerFactory se genera una plantilla XSLT a partir del String cargado, si la plantilla no es válida la función genera una excepción con un mensaje con las causas del error. Dicha información se muestra al usuario para facilitar su corrección.

- **Enrich**: Primero se verifica que las variables de FreeMarker en la plantilla sean las especificadas al momento del alta de la situación, ya que cada situación puede tener un conjunto de información contextual de salida que debe corresponder con el conjunto de variables de la plantilla FreeMarker.
- **External Transformation**: Se valida que cumpla el formato básico de una URL.
- **Content Based Router**: Para definir esta adaptación el usuario debe crear un árbol de condiciones donde cada nodo es un XPath booleano o una adaptación de las anteriores. Cuando se define el árbol, se realiza una validación nodo por nodo. En caso que el nodo sea un XPath, se valida con la función "compile" de la biblioteca javax.xml.xpath, que lanza una excepción si el mismo no es válido. Si el nodo es una adaptación como las descritas anteriormente, se le aplica la validación correspondiente de esa adaptación.

5.2.9.5 Manejo Avanzado de Reglas

En esta sección es posible visualizar y modificar las reglas del Motor CEP. Al momento de generar las versiones se valida que el conjunto de reglas sea correcto. Es posible hacer modificaciones sobre cualquier versión de reglas y generar una nueva versión a partir de esta modificación. Sin embargo, no es posible agregar nuevas reglas desde esta pantalla, ya que las reglas están asociadas a una situación y se crean al momento de crear las mismas. Un punto importante a destacar es que cuando se genera una nueva versión, ya sea en esta pantalla o cuando se da de alta una nueva situación, la misma no es desplegada automáticamente, permitiendo al administrador realizar configuraciones sin necesidad de activarlas instantáneamente. La única forma de desplegar una nueva versión en el motor es desde esta sección.

En la Figura 52 se puede ver el conjunto de reglas de la versión 1.3.1 seleccionada, y se ve que la versión activa es la 1.2.2.

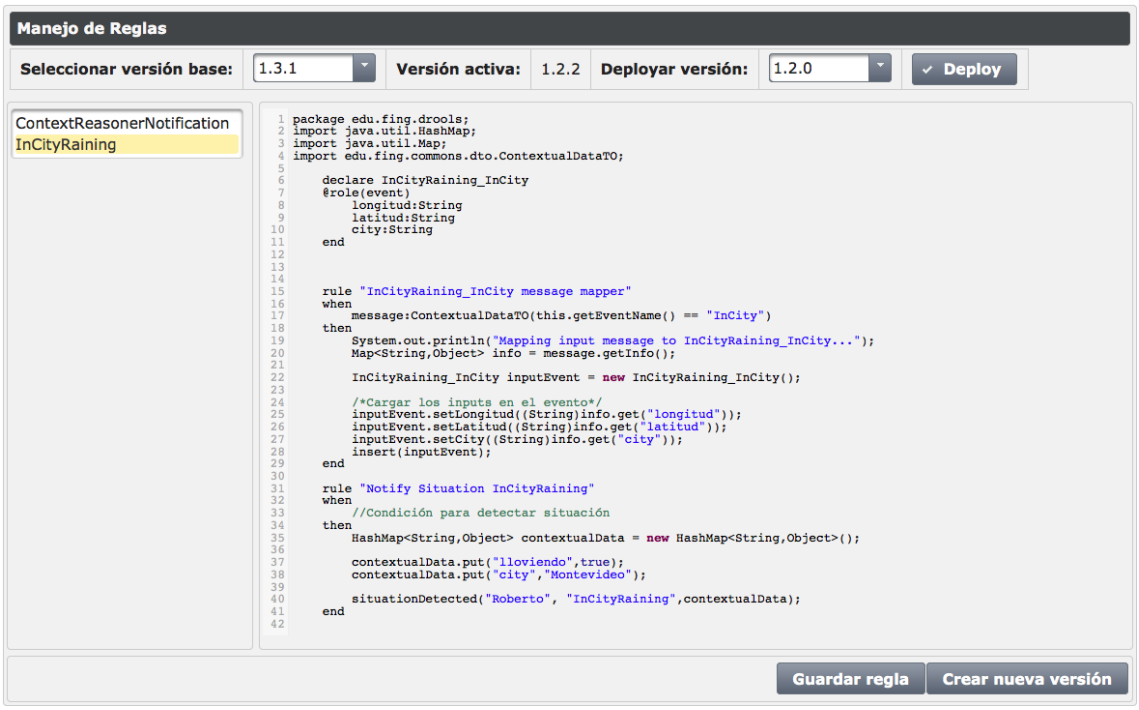


Figura 52 – Pantalla del manejo de reglas.

5.2.9.6 Monitoreo

Visualizar las distintas configuraciones de la plataforma es de gran importancia para el administrador por lo que se brindan tres pantallas enfocadas en visualizar el estado actual del sistema. La primera pantalla permite visualizar los itinerarios configurados, la segunda permite ver los itinerarios activos y la última las situaciones activas junto con las reglas y sus datos contextuales.

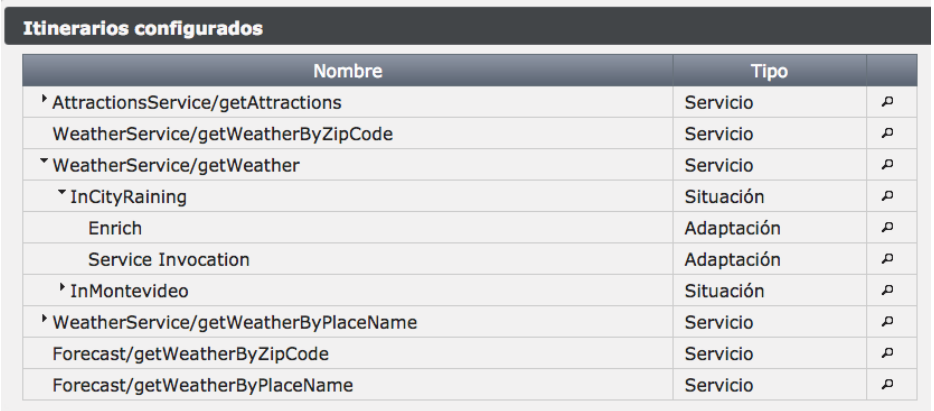


Figura 53 – Pantalla de itinerarios configurados.

En la Figura 53 se muestran los itinerarios configurados para los distintos servicios y situaciones que son consultados al Razonador de Contexto. Haciendo clic en el icono de la lupa se muestran más detalles, por ejemplo en caso de una adaptación *Enrich* la plantilla correspondiente.

Figura 54 – Pantalla de itinerarios activos.

Para obtener los datos de los itinerarios activos que se ven en la Figura 54, la Consola de Administración consulta al Gateway de Adaptación, que es el que tiene todos los itinerarios a aplicar.

Figura 55 – Pantalla de situaciones activas y su información contextual.

En la Figura 55 se pueden ver las situaciones activas junto a su información de entrada y salida, y la regla que la detecta. En este último caso se decide mostrar las reglas y situaciones asociadas a la versión de reglas activa, ya que pueden existir situaciones para las cuáles no existen reglas desplegadas en el motor CEP. Esto implica que si se crea una nueva situación y la versión generada no se ha desplegado, la situación no es mostrada en esta pantalla.

5.3 Problemas encontrados

En la siguiente sección se describen los problemas encontrados en el desarrollo de la plataforma. Si bien no fueron los únicos problemas encontrados, fueron los más relevantes debido al tiempo que se dedicó a resolverlos y a su impacto en la solución.

5.3.1 Actualización de sesiones Drools al actualizar reglas

Según la documentación de Drools, desplegar una nueva versión actualiza la versión en todas las sesiones ya creadas como demuestra el test unitario "testKJarUpgradeSameSession" de [39]. Sin embargo, esto no funciona cuando Drools se encuentra funcionando en modo *stream*, es decir, cuando se configura para ser utilizado como Motor de Procesamiento de Eventos Complejos (Drools Fusion). Esto

mismo fue corroborado en una discusión en GoolgleGroups, donde se discute sobre el problema con un desarrollador que utiliza Drools y un desarrollador de Drools, llegando a la conclusión de que era un bug de la versión 6.0.0 Final. La discusión de GoogleGroups se puede encontrar en [40].

La otra versión final (6.1.0 Final) disponible al momento de la implementación tampoco soluciona el inconveniente y versiones anteriores tenían problemas similares como se muestra en [41]. Por lo que esto, junto con el análisis del impacto realizado en el Apéndice 3, motivó continuar con la versión 6.0.0 Final que se estaba utilizando.

5.3.2 Ciclo de vida SwitchYard separado del ciclo de vida CDI

El ciclo de vida de los *beans* CDI[20] está separado del ciclo de vida de SwitchYard. Esto genera que no se puedan utilizar elementos de SwitchYard en un método anotado como @PostConstruct[21], ya que pueden existir elementos de SwitchYard que no estén listos para ser utilizados. Este problema puede verificarse en el Bugzilla en el hilo [42] y obligó a usar distintas estrategias para poder inicializar algunos *beans* que requerían una inicialización dependiente de elementos de SwitchYard.

5.3.3 Configuraciones de *endpoints* en tiempo de ejecución

SwitchYard posee grandes limitantes a la hora de configurar *endpoints*, ya sea para invocar un *endpoint* o publicar un nuevo servicio. Cuando se crea un servicio, el mismo no puede definirse como un *endpoint* genérico (definido por una expresión), sino que debe ser fijo. Esto obliga a tener que crear nuevas aplicaciones para configurar un nuevo *endpoint* si no se quiere tener *downtime*.

Además, SwitchYard tampoco permite invocar una URL que no haya sido definida en tiempo de compilación en su archivo de configuración, por lo que no pueden definirse en tiempo de ejecución nuevas invocaciones, algo que resulta crítico para la plataforma. Esto puede observarse en las referencias [43], [44] y [45] donde se crean solicitudes para que estas limitantes puedan ser configuradas de una manera más flexible. Este inconveniente obligó a desarrollar el componente Invocador de Servicios.

5.3.4 Optimizaciones de Binding SCA

Un *binding* SCA es una forma sencilla de comunicar dos aplicaciones SwitchYard. Una invocación de este tipo tiene algunas optimizaciones que son muy buenas. Un ejemplo de estas optimizaciones es que cuando se realiza una invocación por un *binding* SCA a otra aplicación que se ejecuta en la misma JVM. La invocación se hace igual que como si se invocara un método propio, lo que se puede verificar en [46] o utilizando el *debugger* de Eclipse. Sin embargo, esta invocación puede traer problemas, como trae cuando se accede a recursos que dependen del *classpath* de la aplicación misma. Esto llevó a que no se pueda utilizar una invocación con un *binding* SCA para realizar acciones sobre el

[20] CDI (Context Dependency Injection) es una tecnología Java para resolver la inyección de dependencias.
[21] https://docs.oracle.com/javaee/6/tutorial/doc/gmgkd.html

Motor CEP desde el Razonador de Contexto, ya que al hacerlo, Drools no encuentra algunas bibliotecas.

Para solucionar esto, se decidió utilizar el cliente “RemoteInvoker” para así evitar esta optimización que generaba los inconvenientes mencionados.

5.3.5 Creación de Proyecto Web SwitchYard

Según su documentación [47], SwitchYard permite la creación de proyectos web, lo que permite añadir las facilidades de interconexión que brinda SwitchYard. Sin embargo, no fue posible generar proyectos con dicha funcionalidad ya que existen problemas para su configuración. Inicialmente no es posible crear el empaquetado WAR asociado a proyectos web siguiendo la guía existente. Buscando información en foros se pudo corregir dicho problema y generar el WAR correspondiente siguiendo la recomendación del foro [48].

No obstante, pese a que el proyecto puede ser desplegado, los servicios SwitchYard no se inicializan impidiendo que puedan utilizarse. Por este motivo se decidió no utilizar SwitchYard para la creación de la Consola de Administración.

6 Casos de Estudio

En este capítulo se presentan los casos de estudio realizados, enfocados en validar la solución planteada y mostrar el funcionamiento de la plataforma.

6.1 Objetivo

El objetivo de los casos de estudio es validar el correcto funcionamiento de la plataforma y servir de ejemplo para entender cómo se realizan las configuraciones. Para realizar los casos de estudio se tomaron ideas de lo propuesto en [49], un artículo que consiste en el desarrollo de aplicaciones móviles basada en el concepto de *Smart Tourism*. Dicho artículo se basa en el conocimiento del momento actual del tráfico y de las ofertas turísticas de la ciudad en que se encuentre el turista.

Los casos de estudio realizados para la validación de la plataforma se basaron en las ofertas turísticas de Uruguay, teniendo en cuenta el contexto del turista. Para conocer dicho contexto se utilizaron fuentes de contexto que permitieran conocer la ciudad donde se encuentra y el clima de las distintas ciudades de Uruguay.

El primer caso consiste en un ciclo completo de uso del prototipo, desde la configuración de la plataforma hasta el uso por parte de un usuario, en este caso un turista. El segundo y tercer caso añaden complejidad al primero enfocándose en cubrir otros aspectos de la solución desarrollada.

6.2 Caso de Estudio 1

En este caso de estudio se desea contextualizar un servicio con una operación que devuelve una lista de atracciones turísticas de Uruguay. Esta operación recibe opcionalmente un parámetro con el nombre de la ciudad para obtener las atracciones de dicha ciudad. Cuando se detecte que el turista se encuentra en determinada ciudad, se quiere enriquecer el *request* con dicha ciudad para obtener solamente aquellas atracciones que sean de la ciudad donde se encuentra. Para esto es necesario configurar en la plataforma el servicio, la fuente de contexto que brinda la posición del usuario, la regla que detecta que el usuario se encuentra en una ciudad y el itinerario necesario para enriquecer el mensaje.

Todo este proceso de configuración tiene una complejidad elevada por lo que se trata de mostrar detalladamente cada paso.

6.2.1 Configuración

Para poder definir el conjunto de adaptaciones que se desean aplicar a un mensaje, es necesario definir previamente el servicio y la situación. Además, antes de definir una situación es necesario definir las fuentes de contexto que alimentarán dicha situación y los datos contextuales que utilizará. A continuación se detallan los pasos a seguir:

primero se define el servicio, luego la situación con sus fuentes de contexto y por último las adaptaciones.

6.2.1.1 Configuración del Servicio

Como primer paso se configura el servicio que se quiere contextualizar, en este caso "AttractionsService", con la operación "getAttractions" que devuelve las atracciones turísticas de Uruguay como se explicó anteriormente. En la Figura 56 se puede ver la pantalla para dar de alta el servicio, donde se especifica únicamente la URL del servicio y a partir de ésta se cargan los datos. Adicionalmente deben seleccionarse las operaciones del servicio. La URL del servicio virtual que se muestra en la Figura 56 es autogenerada e indica la URL de publicación del nuevo servicio.

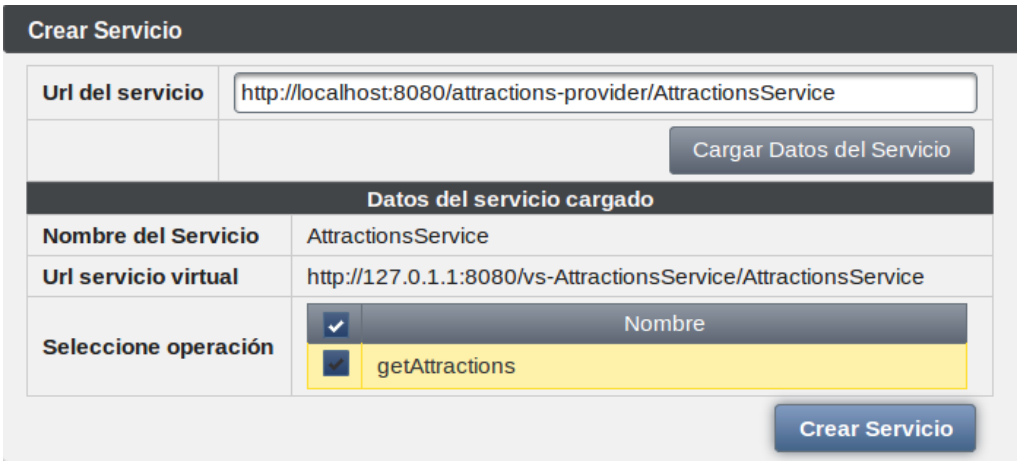


Figura 56 – Pantalla de creación de un servicio.

6.2.1.2 Configuración de la Situación

Luego para poder dar de alta la situación que indica la ciudad donde se encuentra un usuario, es necesario configurar las fuentes de contexto que alimentan al Motor CEP. En este caso se define una fuente de contexto que escucha la posición de los usuarios. En la Figura 57 se puede ver cómo se define la fuente de contexto "UserPosition". En este caso se especifica XML como formato de mensaje y *listener* como tipo de fuente.

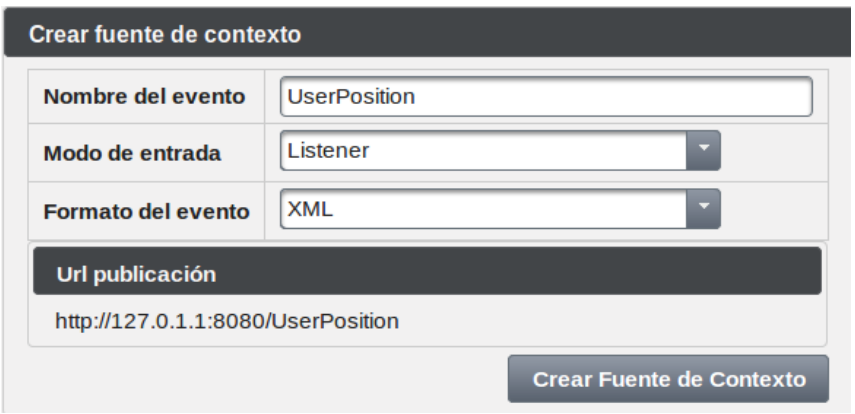


Figura 57 – Pantalla de creación de una fuente de contexto listener.

Una vez que se configuró la fuente de contexto, es necesario definir los datos contextuales que genera dicha fuente para poder utilizarlos al momento de crear la situación. En este caso, la fuente "UserPosition" genera los datos contextuales: latitud, longitud e identificador del usuario. En la Figura 58 se puede ver la pantalla donde se definen estos datos y donde se está dando de alta el dato "city" que será generado como resultado de la situación.

Figura 58 – Pantalla de creación de datos contextuales.

Ahora que ya se encuentran definidos los datos contextuales, se debe configurar la situación "InCity" que indicará que un usuario se encuentra en determinada ciudad. La configuración de las situaciones consiste de tres pantallas. En la primera se especifica el nombre, descripción y la duración de la situación como se puede ver en la Figura 59.

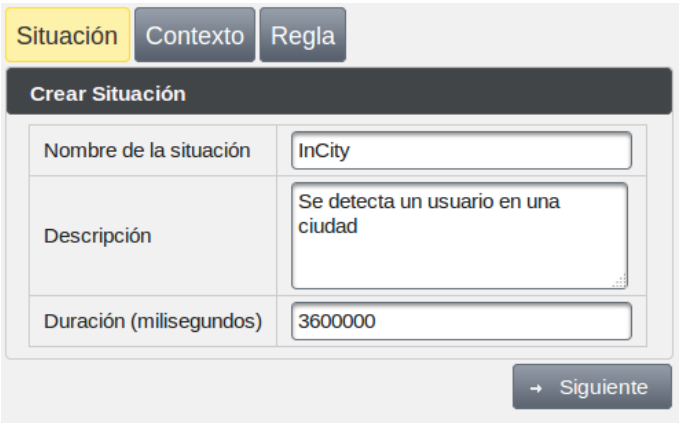


Figura 59 – Pantalla de creación de una situación.

En el segundo paso de la configuración de una situación se deben seleccionar las fuentes de contexto que alimentan la situación, junto con los datos contextuales, tanto para cada fuente como los generados por la situación. En la Figura 60 se observa cómo se cargan estos datos, con la fuente de contexto y los datos contextuales cargados anteriormente.

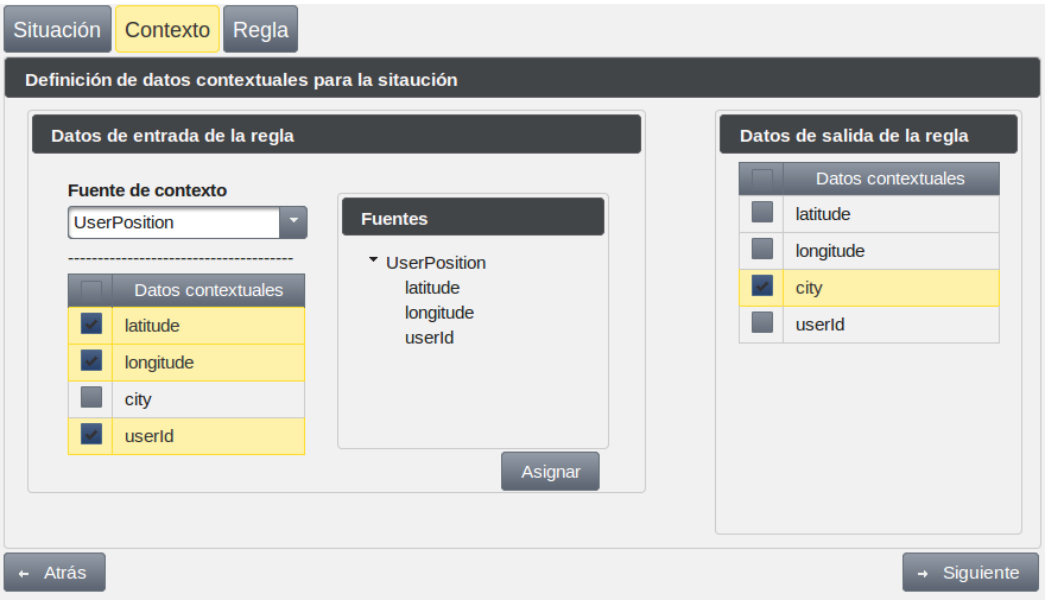


Figura 60 – Pantalla de definición de datos contextuales para una situación.

Por último se define la regla para detectar la situación que es generada automáticamente con los datos cargados anteriormente. En la Figura 61 se muestra la sugerencia de regla para generar la regla que detecta la situación "InCity", donde se puede observar cómo el código autogenerado utiliza la fuente de contexto y los datos contextuales definidos en la pantalla anterior.

```
package edu.fing.drools;
import java.util.HashMap;
import java.util.Map;
import edu.fing.commons.dto.ContextualDataTO;
/**
* Se definen los eventos que llegan desde las Fuentes de Contexto.
* Si un evento es utilizado por mas de una fuente, se debe definir solo una vez, o de lo contrario utilizar distintos nombres para los eventos.
**/
    declare InCity_UserPosition
    @role(event)
     userId:String
     longitude:String
     latitude:String
    end

    rule "InCity_UserPosition message mapper"
    when
        message:ContextualDataTO(this.getEventName() == "UserPosition")
    then
        System.out.println("Mapping input message to InCity_UserPosition...");
     Map<String,Object> info = message.getInfo();

     /*La variable donde se carga la fuente de contexto DEBE llamarse inputEvent*/
     InCity_UserPosition inputEvent = new InCity_UserPosition();

     /*Cargar los inputs en el evento. Se valida el uso de los datos contextuales*/
     inputEvent.setUserId((String)info.get("userId"));
     inputEvent.setLongitude((String)info.get("longitude"));
     inputEvent.setLatitude((String)info.get("latitude"));
     insert(inputEvent);
    end

    rule "Notify Situation InCity"
    when
     //Condicion para detectar situación
    then
        HashMap<String,Object> contextualData = new HashMap<String,Object>();
        /*
        Cargar hash de data contextual
        - El nombre de la variable DEBE ser contextualData
        - Las claves del hash DEBEN ser strings "entre comillas"
        - Se valida el uso de datos contextuales
        */
     contextualData.put("city",/*VALOR_city*/);
        /*Sustituir USER_ID por el id del usuario de tipo String*/
     situationDetected(/*USER_ID*/, "InCity",contextualData);
    end
```

Figura 61 – Pantalla de definición de la regla que detecta una situación.

En el Apéndice 8 se detalla la implementación de la regla “InCity”. Para finalizar la creación se debe elegir el nombre de la versión donde se incluirá esta nueva regla, que se agrega a la última versión generada.

Finalmente para que la configuración tenga efecto, se debe desplegar la nueva versión en la pantalla de Manejo Avanzado de Reglas descrita en la Sección 5.2.9.5.

6.2.1.3 Configuración del itinerario

Como último paso de la configuración, luego de haber configurado el servicio y la situación, se define el itinerario para dicho servicio y situación. En este caso, como se quiere agregar al *request* la ciudad detectada en la situación, el itinerario consta de dos pasos: un enriquecimiento con su correspondiente transformación XSLT y la invocación a “getAttractions”. En la Figura 62 se puede observar la pantalla de creación del itinerario, donde se indica el servicio, la situación y la prioridad del itinerario.

Figura 62 – Pantalla de creación de un itinerario.

Además, en la pantalla de la Figura 62 se definen las adaptaciones de cada paso del itinerario. En el caso del *Content Enrichment* es necesario cargar la plantilla de la transformación XSLT. Para este caso de estudio, la plantilla utilizada se puede observar en la Figura 63.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:p="urn:edu.fing.context.management:test-case:1.0" version="2.0">
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>
  <xsl:template match="p:getAttractions">
    <xsl:copy copy-namespaces="no">
      <xsl:apply-templates select="@* | *" copy-namespaces="no" />
      <city>${city}</city>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Figura 63 – Plantilla de la transformación XSLT para el Content Enrichment.

6.2.2 Resultados

Ahora que la plataforma se encuentra configurada, cuando se detecte la situación "InCity" para un usuario se aplicará la directiva de adaptación.

A continuación, en la parte izquierda de la Figura 64 se puede observar la respuesta original del servicio "getAttractions" antes de ser contextualizado, donde se obtienen las atracciones de distintas ciudades. En la parte derecha de la Figura 64 se muestra la respuesta del servicio contextualizado luego de detectarse la situación "InCity". En el ejemplo se detectó que el usuario se encontraba en "Montevideo", por lo que se aplicó el enriquecimiento del *request* con dicha ciudad, obteniendo como respuesta únicamente las atracciones de Montevideo.

Figura 64 – Comparación de respuestas.

6.3 Caso de Estudio 2

Este caso de estudio utiliza como base el caso de estudio anterior agregándole más complejidad. Lo que se complejiza es la situación que se quiere detectar, identificando la ciudad donde se encuentra el usuario y además si en dicha ciudad está lloviendo. El servicio que se utiliza en este caso de estudio es el mismo que en el caso anterior, devolviendo una lista de atracciones turísticas donde cada una indica si está o no está al aire libre. Con esta nueva situación se desea aplicar una directiva de adaptación que además de agregar al *request* la ciudad donde se encuentra el usuario, filtre del *response* las atracciones que sean al aire libre.

6.3.1 Configuración

Para detectar esta nueva situación llamada “InCityRaining”, primero se debe crear una nueva fuente de contexto que permita conocer el estado del tiempo de distintas ciudades. Para obtener dicha información se invoca a un servicio de la API que provee *openweathermap.org* [50] que recupera los estados del tiempo de los 19 departamentos de Uruguay a partir de los identificadores de los mismos. Se decidió utilizar departamentos en vez de ciudades para simplificar el caso de estudio. En la Figura 65 se muestra cómo se crea la fuente de contexto “CityWeather” con formato JSON y en modo *polling* que consultará al servicio cada un minuto.

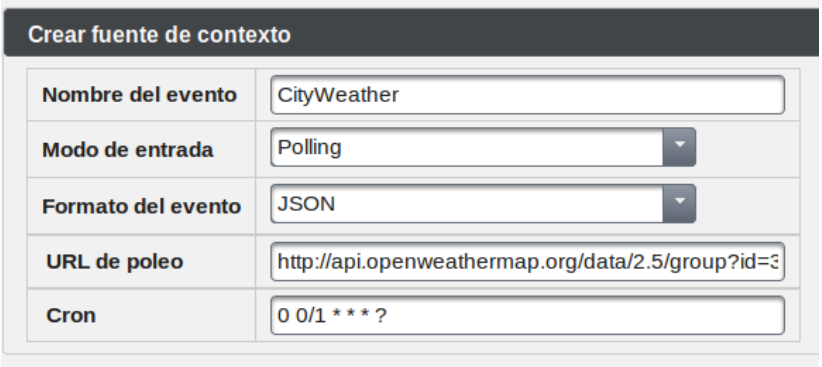


Figura 65 – Pantalla de creación de una fuente de contexto polling.

Como datos contextuales de esta fuente de contexto se agregaron *name* y *main* que corresponden al nombre de la ciudad y el estado del tiempo respectivamente. Estos valores no fueron elegidos sino que son los provistos por el servicio [50].

Luego de configurada la nueva fuente de contexto y sus datos contextuales, se configura la situación de igual manera que en la Sección 6.2.1.2. El dato contextual que genera esta situación es la ciudad, ya que indica que en dicha ciudad está lloviendo. En el Apéndice 8 se encuentra la implementación de la regla que detecta la situación "InCityRaining".

Por último, se configura un nuevo itinerario para esta nueva situación y el mismo servicio. Como existe más de un itinerario para el mismo servicio, se debe configurar la prioridad de esta nueva situación con respecto a la anterior. En este caso, cuando llegue una invocación y se haya detectado la situación "InCityRaining" se desea aplicar el nuevo itinerario, por lo tanto debe ser configurado con mayor prioridad que el itinerario para "InCity".

El nuevo itinerario consta de tres pasos: un enriquecimiento como en el caso de estudio anterior, la invocación al servicio y un filtrado que remueve de la respuesta las atracciones al aire libre. En la Figura 66 se muestra la transformación XSLT utilizada para el filtrado definido en el itinerario.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:output method="xml" indent="yes" />
  <xsl:template match="@* | node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>
  <xsl:template match="getAttractionsResponse/attraction[outside='true']" />
</xsl:stylesheet>
```

Figura 66 – Transformación XSLT para el Content Filter.

6.3.2 Resultados

Ahora que la plataforma se encuentra configurada, cuando se detecte la nueva situación "InCityRaining" para un usuario se aplicará la directiva de adaptación.

A continuación, en la Figura 67 se puede observar la respuesta de "getAttractions" contextualizada para la situación "InCity" y luego para "InCityRaining". Para este último caso se observa que fueron filtradas las atracciones turísticas de Montevideo que se encuentran al aire libre.

Figura 67 – Comparación de respuestas contextualizadas.

6.4 Caso de estudio 3

Este caso de estudio plantea definir un itinerario alternativo para la misma situación y servicio descritos en la Sección 6.2.

Para este caso de estudio se cuenta con dos nuevos servicios provistos por las intendencias de Montevideo y Maldonado. Estos servicios poseen la misma estructura que el servicio "getAttractions" y devuelven mayor cantidad de atracciones ubicadas en el departamento de la intendencia correspondiente con información más detallada y actualizada. Por este motivo para Montevideo y Maldonado se quiere utilizar preferentemente estos servicios.

Para modelar este comportamiento primero se enriquece al mensaje con la ciudad donde se encuentra el usuario y en base a esta ciudad se decide a que servicio se debe invocar.

6.4.1 Configuración

Para utilizar los servicios que proveen las intendencias, se utiliza el ruteo basado en contenido luego de enriquecer al *request* con la ciudad. Este ruteo utiliza la ciudad donde se encuentra el usuario para decidir si invocar al servicio de la intendencia que

corresponda o al servicio original para el caso de las intendencias que no proveen dicho servicio.

De esta manera, el nuevo itinerario consta de dos pasos: el enriquecimiento y el ruteo basado en contenido que dirigirá al mensaje al servicio que corresponda. En la Figura 68 se puede observar cómo se crea el ruteo basado en contenido al momento de crear el itinerario.

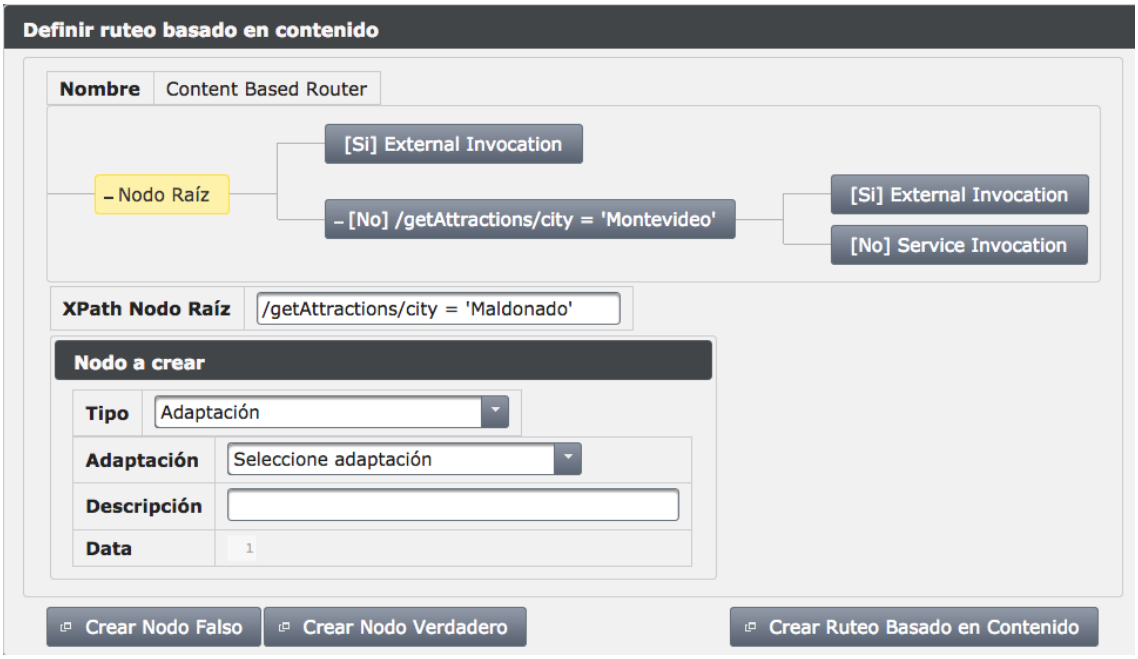


Figura 68 – Pantalla de definición del ruteo basado en contenido.

En la Figura 68 se puede observar que el nodo raíz es un XPath que verifica si el *request* contiene la ciudad Maldonado. En caso de contenerla se aplica la adaptación *External Invocation* que invocará al servicio de la intendencia de Maldonado y en caso contrario evalúa el XPath que verifica si el *request* contiene la ciudad Montevideo. En caso de contenerla se invocará al servicio de la intendencia de Montevideo y en caso contrario se realiza la invocación a “getAttractions”.

6.4.2 Resultados

Luego de generar el nuevo itinerario para la situación “InCity”, la respuesta al usuario será distinta al primer caso de estudio. Como el usuario se encuentra en Montevideo y se aplicó el enriquecimiento como primer paso del itinerario, el ruteo basado en contenido dirige el mensaje al servicio de la intendencia de Montevideo. En la Figura 69 se muestra la comparación entre la respuesta obtenida con el itinerario configurado en el primer caso de estudio y el nuevo itinerario.

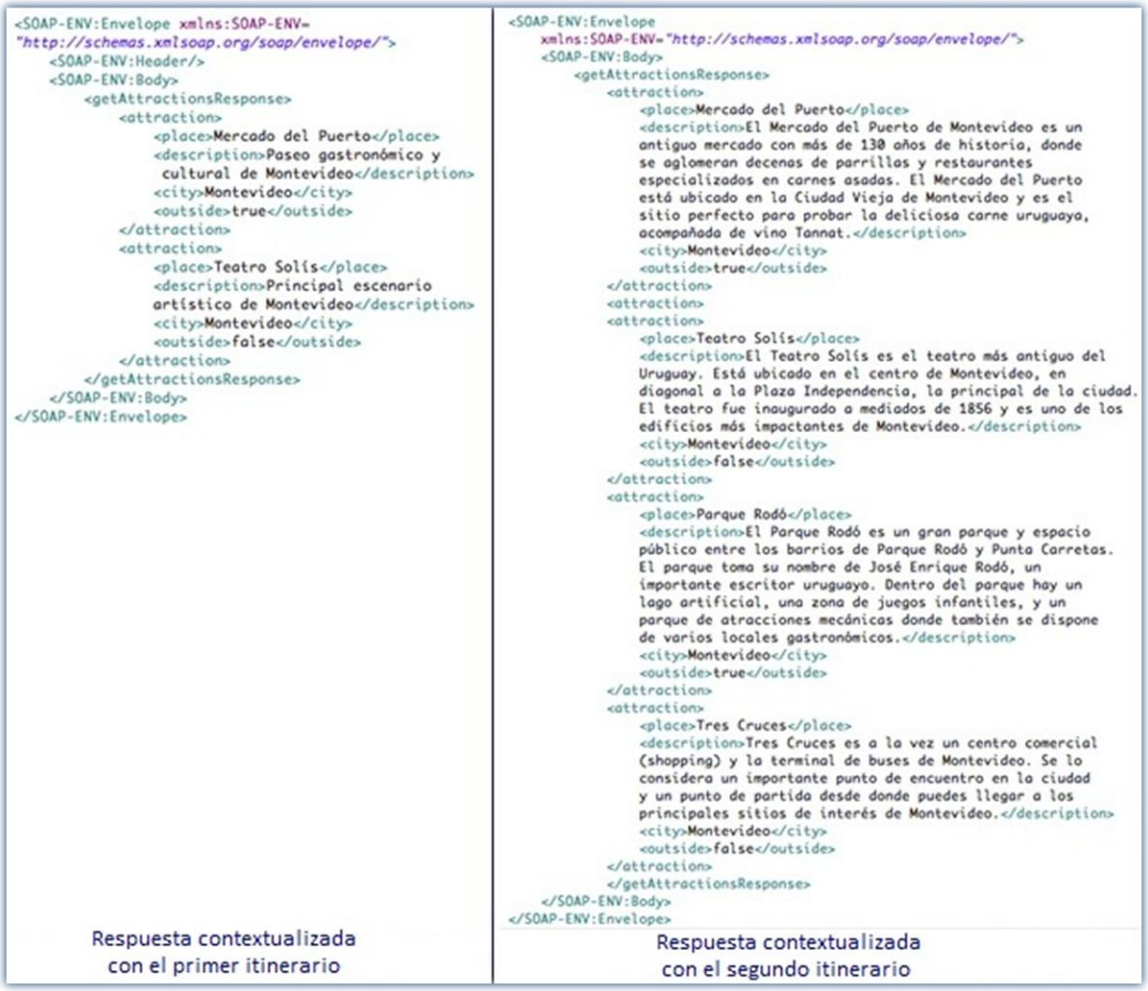


Figura 69 – Comparación de respuestas con distintos itinerarios.

7 Conclusiones y Trabajo a Futuro

Este capítulo presenta en la Sección 7.1 las conclusiones del trabajo realizado y en la Sección 7.2 posibles mejoras que podrían ser realizadas como trabajo a futuro.

7.1 Conclusiones

El objetivo general del proyecto fue la implementación de una plataforma de integración que permita detectar el contexto de invocación de un servicio y adaptarlo de acuerdo al mismo. En particular, la plataforma se implementó utilizando como base el ESB SwitchYard y el motor CEP Drools Fusion.

La primera etapa consistió en realizar evaluaciones de los productos ESB del mercado con el objetivo de elegir el producto que mejor se adaptaba a los requerimientos del proyecto. Dichas evaluaciones permitieron seleccionar a SwitchYard como producto base para el desarrollo de la plataforma debido a sus capacidades de mediación, editor gráfico y sus buenos tutoriales, muy útiles para comenzar a utilizar el producto. También se evaluó el motor CEP Drools Fusion sugerido, de manera de comprobar si cumplía con los requerimientos del proyecto.

Tomando como base los productos seleccionados se diseñó e implementó la solución. Se incorporaron algunos conceptos a la solución planteada debido a características de los productos elegidos y a particularidades surgidas en la implementación. Uno de los conceptos incorporados a la solución fue el versionado de reglas, que surge debido al modo que Drools Fusion maneja las reglas.

Entre los componentes desarrollados se encuentra la Consola de Administración. Su fin es brindar una interfaz gráfica que permita definir y monitorear las herramientas necesarias para que el administrador pueda realizar la configuración de los servicios a contextualizar en tiempo de ejecución. Se buscó que dicho proceso de configuración fuera lo más sencillo y guiado posible, por lo que se incluyó una gran cantidad de validaciones y algunas sugerencias como la regla que se propone como guía al momento de definir una situación.

Luego de haber utilizado SwitchYard para el desarrollo se concluyó que es un producto con mucho potencial pero que aún no tiene el grado de madurez esperado en una plataforma empresarial. Si bien resuelve nativamente muchos de los problemas planteados, tiene limitantes como la falta de ejemplos de uso complejos documentados, además de ciertos *bugs* encontrados. Sumado a esto, la mayoría de sus funcionalidades no están diseñadas para ser configuradas en tiempo de ejecución, lo que provocó que el desarrollo de esta plataforma en particular fuera más complejo.

Con respecto a Drools Fusion se puede decir que es una plataforma con gran potencia para definir reglas para la detección de eventos complejos. Sin embargo la definición de estas reglas resulta compleja y para nada trivial. Se encontraron algunos *bugs* que no se encontraban reportados y afectaban a varias versiones. Además en la documentación

existen ejemplos desactualizados y que no funcionan en la versión correspondiente. Estos motivos hacen dudar del grado de madurez del producto, que aparentaba ser mayor.

Finalmente se realizaron casos de estudio para validar el correcto funcionamiento de la plataforma y demostrar que la misma contribuye a la generación de servicios web más personalizados y precisos. Si bien estos casos son acotados, permiten demostrar el potencial de la herramienta desarrollada y servir de ejemplo de configuración del sistema.
Como conclusión general del trabajo realizado se puede decir que la implementación de una plataforma de integración basada en *"An ESB-Based Infrastructure for Event-Driven Context-Aware Web Services"* es factible, y su uso en servicios web reales es beneficioso.

7.2 Trabajo a Futuro

A continuación se mencionan algunos puntos interesantes que fueron identificados como posibles mejoras.

- **Combinación de itinerarios**

Actualmente sólo es posible aplicar un único itinerario para un servicio (el de mayor prioridad), sin importar si existe más de una situación que aplique para ese servicio y usuario. Una mejora de esto sería permitir la combinación de itinerarios en determinadas condiciones, donde los itinerarios sean considerados combinables. Actualmente, este comportamiento puede ser modelado creando distintos itinerarios para las distintas combinaciones de situaciones y ajustando las prioridades, pero se vuelve dificultoso si la cantidad de itinerarios a gestionar crece.

- **Alimentar el Motor CEP con invocaciones de servicios**

Otra funcionalidad a agregar sería la posibilidad de alimentar el Motor CEP con los *requests* de los servicios. De esta manera, cuando se invoque a un servicio, el mensaje podría ser enviado al Motor CEP asincrónicamente para servirle como una nueva fuente de contexto. Con estos mensajes como eventos en el Motor CEP se brinda al administrador mayor poder a la hora de definir situaciones. Por ejemplo, si el request tuviera información del idioma de preferencia del usuario, ésta podría utilizarse como dato contextual de una situación.

- **Posibilidad de definir *mappers* para las integraciones con servicios externos**

Actualmente, si se cuenta con servicios similares al servicio que se está contextualizando pero con un formato de mensajes distinto, no es posible invocarlos, salvo que se puedan adaptar con transformaciones XSLT. Por ejemplo si se tiene un servicio que recibe y devuelve la misma información pero en formato JSON, el mismo no puede ser invocado. Para solucionar este problema, sería deseable permitir que el administrador pueda

definir *mappers* al momento de definir la adaptación *External Invocation* y así transformar el formato del mensaje.

Estos *mappers* también podrían definirse para Fuentes de Contexto simplificando el procesamiento que se realiza en las reglas del Motor CEP cuando la estructura del mensaje es compleja.

- **Definir nuevos tipos de adaptaciones**

Otro aspecto a implementar en el futuro podría ser la definición de nuevos tipos de adaptaciones, como la adaptación que implemente el patrón *Scatter-Gather* definido en los EIP. Este patrón permite enviar un mensaje a varios destinatarios y definir cómo combinar las respuestas para crear una respuesta única. Definir nuevas adaptaciones podría también implicar un desarrollo en la Consola de Administración.

Un nivel más avanzado de esta mejora sería que las adaptaciones puedan ser definidas dinámicamente.

- **Editor de reglas gráfico**

La definición de las reglas para la detección de situaciones es una de las partes más complejas del uso de la plataforma. Dicha definición se hace en base a reglas de Drools que son difíciles de definir para usuarios sin conocimiento del mismo.

En el siguiente el artículo: *"A model-driven approach for facilitating user-friendly design of complex event patterns*" [51] se presenta una herramienta que permite a través de interfaces gráficas definir patrones para la detección de eventos complejos de una manera mucho más guiada y sencilla. La idea es muy interesante y permite que usuarios no expertos puedan definir situaciones complejas. Siguiendo la propuesta planteada en este artículo, una posible mejora sería implementar una interfaz gráfica para la definición de las reglas similar a la planteada en [51].

También se podría aprovechar la interfaz gráfica planteada en [51], que es independiente del motor CEP, e implementar el módulo que genere código Drools.

Otra opción sería incluir en el sistema la posibilidad de definir Domain Specific Languages [30], los cuales se utilizan en Drools con la finalidad de ayudar a usuarios no técnicos a definir las reglas.

- **Manejo avanzado de fuentes de contexto y servicios virtuales.**

Una funcionalidad interesante a añadir a la plataforma es la administración de las fuentes de contexto y servicios virtuales. Si bien actualmente la configuración particular de cada uno es almacenada en la base de datos, no es posible modificar o eliminar lo que ya fue creado. Esta mejora no implica gran desarrollo ya que toda la información necesaria ya está almacenada.

- **Utilizar roles de usuario.**

Otra posible mejora es añadir el concepto de rol para el usuario que utiliza los servicios contextualizados, de esta manera es posible definir situaciones para un tipo específico de usuario. Esto haría posible detectar situaciones no sólo para un usuario específico sino para una categoría de usuarios, pudiendo definir contextos que no estén relacionados con un usuario en particular sino con una situación que afecta a muchos de ellos.

- **Pruebas de performance**

Un posible trabajo a futuro sería realizar pruebas de *performance* de la plataforma evaluando el *overhead* generado por las acciones de adaptación de la misma. Si bien no era uno de los objetivos planteados, es interesante obtener estos datos para realizar posibles optimizaciones en caso que el *overhead* sea demasiado grande. Las mejoras podrían ser la inclusión de mecanismos de c*ache* o el almacenamiento en memoria de las directivas de adaptación del Gateway de Adaptación.

8 Referencias

[1] L. González y G. Ortiz, «An ESB-Based Infrastructure for Event-Driven Context-Aware Web Services,» de *3rd International Workshop on Adaptive Services for the Future Internet*, Malaga, España, 2013.

[2] B. N. Schilit, N. Adams y R. Want, «Context-Aware Computing Applications,» *IEEE Workshop on Mobile Computing Systems and Applications,* pp. 85-90, 1994.

[3] A. García de Prado y G. Ortiz, «Context-Aware Services: A Survey on Current Proposals,» *The Third International Conferences on Advanced Service Computing,* pp. 104-109, 2011.

[4] D. Georgakopoulos y . M. P. Papazoglou, «Service-Oriented Computing,» 2008. [En línea]. Available: http://mitpress.mit.edu/sites/default/files/titles/content/9780262072960_sch_0001.pdf. [Último acceso: Julio 2015].

[5] «Web Services Architecture,» February 2004. [En línea]. Available: http://www.w3.org/TR/ws-arch/. [Último acceso: Marzo 2015].

[6] J.-L. Maréchaux, «Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus,» Marzo 2006. [En línea]. Available: https://www.ibm.com/developerworks/library/ws-soa-eda-esb/. [Último acceso: Abril 2015].

[7] J. v. Hoof, «How EDA extends SOA and why it is important,» 2006. [En línea]. Available: http://jack.vanhoof.soa.eda.googlepages.com/How_EDA_extends_SOA_and_why_it_is_important_-_Jack_van_Hoof_-_v6.0_-_2006.pdf. [Último acceso: Julio 2015].

[8] B. Sosinsky, Cloud Computing Bible, John Wiley & Sons., 2011.

[9] «Extensible Markup Language (XML),» [En línea]. Available: http://www.w3.org/XML/. [Último acceso: Marzo 2015].

[10] «XSL Transformations (XSLT),» Noviembre 1999. [En línea]. Available: http://www.w3.org/TR/xslt. [Último acceso: Mayo 2015].

[11] «XML Path Language,» Noviembre 1999. [En línea]. Available: http://www.w3.org/TR/xpath/. [Último acceso: Mayo 2015].

[12] «Guía Breve de Servicios Web,» [En línea]. Available: http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb. [Último acceso: Marzo 2015].

[13] M. Keen, A. Acharya, S. Bishop, A. Hopkins, S. Milinski, C. Nott, R. Robinson, J. Adams y P. Verschueren, «Patterns: Implementing an SOA Using an Enterprise Service Bus,» 2004. [En línea]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf. [Último acceso: Julio 2015].

[14] «UDDI Spec TC,» [En línea]. Available: http://www.uddi.org/pubs/uddi-v3.0.2-20041019.htm. [Último acceso: Marzo 2015].

[15] «Web Services Security: SOAP Message Security 1.0,» 2004. [En línea]. Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf. [Último acceso: Marzo 2015].

[16] L. González y R. Ruggia, «Adaptive ESB Infrastructure for Service Based Systems,» de *Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solutions*, IGI Global, 2012.

[17] C. H´erault, G. Thomas y P. Lalanda, «Mediation and Enterprise Service Bus: A position paper,» *In Proceedings of the First International Workshop,* 2005.

[18] G. Hohpe y B. Woolf, Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions, Boston: Addison-Wesley, 2004.

[19] «Enterprise Integration Patterns,» [En línea]. Available: http://www.eaipatterns.com/. [Último acceso: Abril 2015].

[20] H. M. Wylie y P. Lambros, «Enterprise Connectivity Patterns: Implementing integration solutions with IBM's Enterprise Service Bus products,» 2009. [En línea]. Available: http://www.ibm.com/developerworks/library/ws-enterpriseconnectivitypatterns/. [Último acceso: Abril 2015].

[21] D. C. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Boston: Addison-Wesley, 2002.

[22] J. Boubeta-Puig, G. Ortiz y I. Medina-Bulo, «An Approach of Early Disease Detection using CEP and SOA,» de *The Third International Conferences on Advanced Service Computing*, 2011.

[23] D. Luckham y W. R. Schulte, «Event Processing Technical Society,» 2011. [En línea]. Available: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf. [Último acceso: Abril 2015].

[24] «Red Hat JBoss Fuse,» [En línea]. Available: http://www.jboss.org/products/fuse/overview/. [Último acceso: Julio 2014].

[25] «Red Hat JBoss Fuse Service Works,» [En línea]. Available: http://www.jboss.org/products/fsw/overview/. [Último acceso: Julio 2014].

[26] «WSO2 Enterprise Service Bus,» [En línea]. Available: http://wso2.com/products/enterprise-service-bus/. [Último acceso: Julio 2014].

[27] L. González, J. L. Laborde, M. Galnares, M. Fenoglio y R. Rugia, «An Adaptive Enterprise Service Bus Infrastructure for Service Based Systems,» de *1st Workshop on Pervasive Analytical Service Clouds for the Enterprise and Beyond*, Berlín, Alemania, 2013.

[28] «JBoss ESB,» [En línea]. Available: http://jbossesb.jboss.org. [Último acceso: Julio 2014].

[29] «SwitchYard,» [En línea]. Available: http://switchyard.jboss.org/. [Último acceso: Julio 2014].

[30] «Drools Documentation,» [En línea]. Available: http://docs.jboss.org/drools/release/6.0.0.Final/drools-docs/html_single/index.htm. [Último acceso: Abril 2015].

[31] «Project Documentation Editor - SwitchYard 1.1,» [En línea]. Available: https://docs.jboss.org/author/display/SWITCHYARD11/Home. [Último acceso: Mayo 2015].

[32] J. Anstey, «Open Source Integration with Apache Camel and How Fuse IDE Can Help,» [En línea]. Available: http://java.dzone.com/articles/open-source-integration-apache. [Último acceso: Mayo 2015].

[33] «Apache Maven,» [En línea]. Available: https://maven.apache.org/. [Último acceso: Julio 2015].

[34] «MySQL,» [En línea]. Available: https://www.mysql.com/. [Último acceso: Julio 2015].

[35] «Hibernate,» [En línea]. Available: http://hibernate.org/orm/. [Último acceso: Julio 2015].

[36] «JavaServer Faces Technology,» [En línea]. Available: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. [Último acceso: Julio 2015].

[37] «PrimeFaces - Ultimate JSF Framework,» [En línea]. Available: http://primefaces.org/. [Último acceso: Julio 2015].

[38] «FreeMarker,» [En línea]. Available: http://freemarker.org/. [Último acceso: Abril 2015].

[39] «GitHub IncrementalCompilationTest,» [En línea]. Available: https://github.com/droolsjbpm/drools/blob/master/drools-compiler/src/test/java/org/drools/compiler/integrationtests/IncrementalCompilationTest.java. [Último acceso: Octubre 2014].

[40] «Google Groups Forum Existing KieBase,» [En línea]. Available: https://groups.google.com/forum/?utm_medium=email&utm_source=footer#!msg/drools-usage/txDq9Zv--ds/D5uKGNNCDDcJ. [Último acceso: Octubre 2014].

[41] «Google Groups Forum KnowledgeAgent,» [En línea]. Available: https://groups.google.com/forum/#!topic/drools-usage/LwFrP8TYlD4. [Último acceso: Octubre 2014].

[42] «RedHat Bugzilla,» Marzo 2015. [En línea]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=1086295.

[43] «Issue SWITCHYARD-2448,» [En línea]. Available: https://issues.jboss.org/browse/SWITCHYARD-2448. [Último acceso: Junio 2015].

[44] «Issue SWITCHYARD-1997,» [En línea]. Available: https://issues.jboss.org/browse/SWITCHYARD-1997. [Último acceso: Junio 2015].

[45] «Issue SWITCHYARD-1916,» [En línea]. Available: https://issues.jboss.org/browse/SWITCHYARD-1916. [Último acceso: Junio 2015].

[46] «Issue SWITCHYARD-2135,» [En línea]. Available: https://issues.jboss.org/browse/SWITCHYARD-2135. [Último acceso: Junio 2015].

[47] «SwitchYard Deployment,» [En línea]. Available: https://docs.jboss.org/author/display/SWITCHYARD/Deployment. [Último acceso: Junio 2015].

[48] «JBossDeveloper Forums,» [En línea]. Available: https://developer.jboss.org/message/731117. [Último acceso: Junio 2015].

[49] A. M. G. López, «Casos de Estudio Smart Tourism,» Julio 2014.

[50] «OpenWeatherMap - Weather API,» [En línea]. Available: http://api.openweathermap.org/data/2.5/group?id=3441575,3443758,3440714,3442727,3441243,3440789,3440034,3440781,3443173,3440055,3443025,3442587,3440639,3442585,3443413,3442007,3441894,3440777,3439781&units=metric. [Último acceso: Julio 2015].

[51] J. Boubeta-Puig, G. Ortiz y I. Medina-Bulo, «A model-driven approach for facilitating user-friendly design of complex event patterns,» *Expert Systems with Applications: An International Journal archive,* vol. 41, nº 2, pp. 445-456, Febrero 2014.

[52] «GitHub CommonTestMethodBase,» [En línea]. Available: https://github.com/droolsjbpm/drools/blob/master/drools-compiler/src/test/java/org/drools/compiler/CommonTestMethodBase.java. [Último acceso: Octubre 2014].

[53] «Google Maps Geocoding API,» [En línea]. Available: https://developers.google.com/maps/documentation/geocoding/?hl=es. [Último acceso: Julio 2015].

Apéndice 1. Componentes SwitchYard

En este apéndice se muestran los componentes principales que se pueden definir en SwitchYard y una breve descripción de los mismos, extraído de [31]. Con el *plugin* de Eclipse los mismos pueden ser usados de manera gráfica como se ve en la Figura 70.

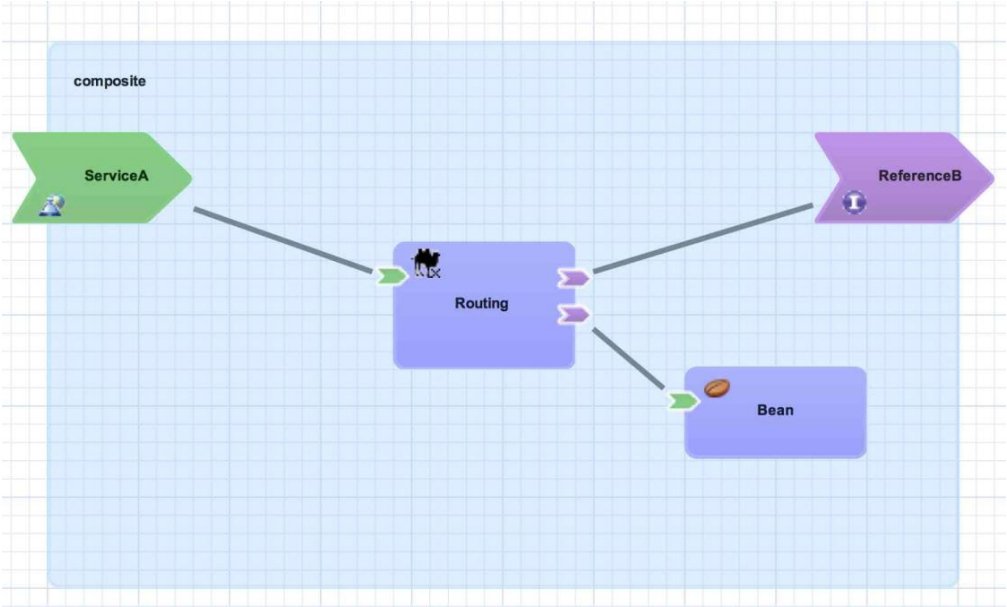


Figura 70 - Ejemplo de aplicación en SwitchYard.

Composición (*Composite*)

Es el rectángulo azul que engloba todos los componentes, que se define como se muestra en la Figura 71 y representa los límites de la aplicación. Una aplicación SwitchYard consiste en una única composición que tiene un nombre y un espacio de nombres destino. El valor del espacio de nombres es importante ya que permite que los nombres definidos dentro de la aplicación sean únicos.

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
</sca:composite>
```

Figura 71 - Definición de una composición.

Componente (*Component*)

Es un contenedor modular para la lógica de la aplicación, mostrado como un rectángulo azul en la Figura 70 y que se define como se muestra en la Figura 72. Cada componente consiste en un servicio de componente o ninguno, en muchas referencias de componentes o ninguna y en una implementación. Los servicios y las referencias permiten al componente interactuar con otros componentes, mientras que la implementación provee la lógica para generar y/o consumir servicios.

```
<sca:component name="Routing">
</sca:component>
```

Figura 72 - Definición de un componente.

Implementación

Actúa como el "cerebro" de un componente y define la implementación de la lógica de la aplicación. Existen varias opciones de implementación:

- *Bean*[22]: permite que un CDI Bean consuma o provea servicios mediante anotaciones.
- Camel: permite la utilización de los EIP usando XML o Java DSL[23] con Apache Camel.
- Reglas: utiliza Drools para la toma de decisiones.

Estas implementaciones son privadas a cada componente, de forma que la interacción con otros componentes o servicios externos es a través de servicios de componentes o de referencias de componentes. En la Figura 73 se muestra un ejemplo de definición de una implementación de un componente con Camel.

```
<sca:component name="Routing">
   <camel:implementation.camel>
      <camel:xml path="RoutingService.xml"/>
   </camel:implementation.camel>
</sca:component>
```

Figura 73 - Definición de la implementación de un componente.

Servicio de componente (*Component Service*)

Es usado para exponer la funcionalidad de una implementación como un servicio. Se representa como una flecha verde a la izquierda de cada componente como se puede ver en la Figura 70 y se define como se muestra en la Figura 74. Todos los servicios tienen un contrato, que puede ser una interfaz Java, WSDL o un conjunto de *datatypes.* También son privados a la aplicación, es decir, que solo pueden ser usados por otros componentes de la aplicación. Si es necesario exponer un *component service*, el mismo puede ser extendido a un *composite service.*

[22] http://searchsoa.techtarget.com/definition/JavaBeans

[23] http://camel.apache.org/dsl.html

```
<sca:component name="Routing">
   <camel:implementation.camel>
      <camel:xml path="route.xml"/>
   </camel:implementation.camel>
   <sca:service name="ServiceA">
      <sca:interface.java interface="org.example.ServiceA"/>
   </sca:service>
</sca:component>
```

Figura 74 - Definición de un servicio de un componente

Servicio de composición (*Composite Service*)

Representa un servicio de una aplicación que es visible para otras aplicaciones. Se da cuando se extiende un servicio de un componte. El nombre e interfaz de la composición puede ser diferente a la del componente. Si la interfaz o el contrato difieren, pueden ser necesarias transformaciones para mapear los tipos de cada uno. Estos servicios se representan como una flecha verde en el borde izquierdo de la delimitación de la aplicación como se muestra en la Figura 70 y se define como se muestra en la Figura 75.

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
   <sca:service name="ServiceA" promote="Routing/ServiceA">
      <sca:interface.wsdl interface="ServiceA.wsdl#wsdl.porttype(ServiceAPortType)"/>
   </sca:service>
</sca:composite>
```

Figura 75 - Definición de un composite service.

Referencia de componente (*Component Reference*)

Una referencia permite a un componente consumir otros servicios. Puede ser cableado a otro servicio disponible de otro componente de la misma aplicación. De forma similar a los servicios de componentes, todas las referencias tienen un contrato que permite a un componente invocar servicios sin conocer su implementación. Se representa como una flecha violeta a la derecha de un componente como se muestra en la Figura 70, donde también se puede ver el cableado a un servicio ofrecido por un Bean. Un ejemplo de definición del mismo se muestra en la Figura 76.

```
<sca:component name="Routing">
   <camel:implementation.camel>
      <camel:xml path="route.xml"/>
   </camel:implementation.camel>
   <sca:service name="ServiceA">
      <sca:interface.java interface="org.example.ServiceA"/>
   </sca:service>
   <sca:reference name="ServiceC">
      <sca:interface.java interface="org.example.ServiceC"/>
   </sca:reference>
</sca:component>
```

Figura 76 - Definición de una referencia a un componente.

Referencia de composición (*Composite Reference*)

Un *composite reference* permite conectar una referencia de componente con servicios fuera de la aplicación. De manera similar a los servicios de una composición, los *bindings* son utilizados por los *composite reference* para especificar el método de comunicación para invocar servicios externos. Estas referencias se representan como una flecha violeta en el borde derecho de la delimitación de la aplicación como se muestra en la Figura 70 y se define como muestra la Figura 77.

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
   <sca:reference name="ReferenceB" multiplicity="0..1" promote="Routing/ServiceB">
      <sca:interface.java interface="org.example.ServiceB"/>
   </sca:reference>
</sca:composite>
```

Figura 77 - Definición de una referencia a una composición.

Service Binding

Un *service binding* se utiliza para definir el método de acceso al *composite service.* Un *composite service* puede tener múltiples *bindings*, permitiendo a un servicio ser accedido de diferentes formas. Por lo general, representan adaptadores de transporte o de protocolos como por ejemplo SOAP, JMS, REST. En la Figura 78 se muestran los diferentes tipos de *bindings* que soporta SwitchYard y su representación como un ícono sobre los servicios. En la Figura 79 se puede ver como se define un *binding* SOAP donde se debe de especificar el WSDL que se publicará.

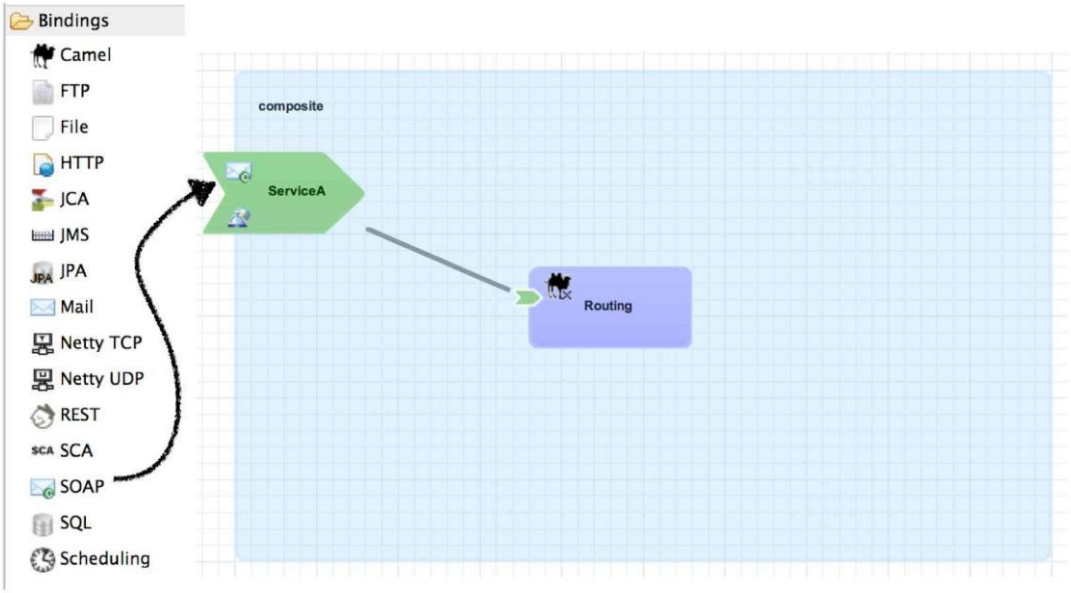


Figura 78 - Representación de un binding.

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
   <sca:service name="ServiceA" promote="Routing/ServiceA">
      <sca:interface.wsdl interface="ServiceA.wsdl#wsdl.porttype(ServiceAPortType)"/>
      <soap:binding.soap>
        <soap:wsdl>ServiceA.wsdl</soap:wsdl>
      </soap:binding.soap>
   </sca:service>
</sca:composite>
```

Figura 79 - Definición de un binding.

Otro tipo de *binding* es el SCA que permite interconectar servicios entre aplicaciones que se encuentren en el mismo entorno de ejecución, brindando un medio de comunicación entre servicios SwitchYard y clientes SwitchYard. Estos *bindings* facilitan la comunicación entre aplicaciones SwitchYard, pudiendo ser usados para vincular una referencia de composición de una aplicación a un servicio de composición de otra aplicación. También provee un *endpoint* de invocación remota para clientes externos permitiendo que aplicaciones externas se comuniquen con aplicaciones SwitchYard.

Reference Binding

Un *reference binding* es utilizado para definir el método de acceso a un servicio externo a través de un *composite reference.* A diferencia de un *service binding,* sólo puede existir un *binding* por cada *composite reference.*

Apéndice 2. Generación dinámica de Fuentes de Contexto y Servicios Virtuales

En este apéndice se explica cómo se resolvió la generación de Fuentes de Contexto y Servicios Virtuales en tiempo de ejecución. Debido a que las Fuentes de Contexto y los Servicios Virtuales se resolvieron de la misma manera, se explica tomando solamente la generación de una Fuente de Contexto en modo *poller*.

Generación Dinámica de Fuentes de Contexto

Para generar las Fuentes de Contexto dinámicamente se crearon dos "plantillas de aplicación", una para cada modo anteriormente descrito, *poller* y *listener*. Las plantillas se crearon a partir de un archivo JAR generado al empaquetar una Fuente de Contexto de ejemplo. A partir de este JAR, se sustituyeron los valores particulares (url, nombre de evento, etc) en los archivos del empaquetado por variables de FreeMarker. De los archivos que componen el JAR solo tres son plantillas FreeMarker: el archivo de propiedades, el XML de configuración de SwitchYard y el pom de Maven. De esta manera, a partir de la plantilla y los valores particulares elegidos por el administrador en la Consola de Administración que reemplazan las variables Freemarker, se crea una nueva Fuente de Contexto. El proceso consiste en instanciar las plantillas, generar el archivo JAR y desplegar la nueva Fuente de Contexto en el servidor JBoss. Para gestionar los archivos JAR se utilizaron bibliotecas del paquete java.util.zip disponible en Java 7.

El componente encargado de gestionar la creación de las nuevas Fuentes de Contexto es la Consola de Administración.

La Figura 80 muestra cómo es el la plantilla de Freemarker para generar el XML de SwitchYard para una Fuente de Contexto en modo *poller*, donde se pueden observar que las variables utilizadas son ${eventName}, ${cron} y ${url}.

```
<sy:switchyard xmlns:bean="urn:switchyard-component-bean:config:1.1"
  xmlns:http="urn:switchyard-component-http:config:1.1"
  xmlns:quartz="urn:switchyard-component-camel-quartz:config:1.1"
  xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
  xmlns:sy="urn:switchyard-config:switchyard:1.1" name="${eventName}"
  targetNamespace="urn:edu.fing.context.management:${eventName}:1.0">
  <sca:composite name="${eventName}"
    targetNamespace="urn:edu.fing.context.management:${eventName}:1.0">
    <sca:component name="CEPMessageComposerPollerBean">
      <bean:implementation.bean class="edu.fing.context.source.polling.CEPMessageComposerPollerBean"/>
      <sca:service name="CEPMessageComposerPoller">
        <sca:interface.java interface="edu.fing.context.source.polling.CEPMessageComposerPoller"/>
      </sca:service>
      <sca:reference name="CEPFeederService">
        <sca:interface.java interface="edu.fing.context.source.polling.CEPFeederService"/>
      </sca:reference>
      <sca:reference name="ContextSourceInvoker">
        <sca:interface.java interface="edu.fing.context.source.polling.ContextSourceInvoker"/>
      </sca:reference>
    </sca:component>
    <sca:service name="CEPMessageComposerPoller"
      promote="CEPMessageComposerPollerBean/CEPMessageComposerPoller">
      <sca:interface.java interface="edu.fing.context.source.polling.CEPMessageComposerPoller"/>
      <quartz:binding.quartz name="Context-Source-Polling">
        <sy:operationSelector operationName="poll"/>
        <quartz:name>Context-Source-Polling</quartz:name>
        <quartz:cron>${cron}</quartz:cron>
      </quartz:binding.quartz>
    </sca:service>
    <sca:reference name="CEPFeederService" multiplicity="0..1"
      promote="CEPMessageComposerPollerBean/CEPFeederService">
      <sca:interface.java interface="edu.fing.context.source.polling.CEPFeederService"/>
      <sca:binding.sca sy:target="ESBServiceDrools"
        sy:targetNamespace="urn:edu.fing.context.management:cep-engine:1.0" name="sca"/>
    </sca:reference>
    <sca:reference name="ContextSourceInvoker" multiplicity="0..1"
      promote="CEPMessageComposerPollerBean/ContextSourceInvoker">
      <sca:interface.java interface="edu.fing.context.source.polling.ContextSourceInvoker"/>
      <http:binding.http name="http1">
        <http:address>${url}</http:address>
        <http:method>GET</http:method>
      </http:binding.http>
    </sca:reference>
  </sca:composite>
</sy:switchyard>
```

Figura 80 – Plantilla de una Fuente de Contexto Poller.

La Figura 81 muestra el archivo de propiedades donde se configuran otras de las variables de Freemarker para generar la nueva Fuente de Contexto.

```
modeConverter=JSON
url=http\://httpbin.org/get
eventName=USER_LOCATION
```

Figura 81 – Archivo de propiedades.

Por último, en el archivo pom.xml se configuran como variables Freemarker parte del *artifactId* y el *name,* como se muestra en la Figura 82.

```
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 Http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>edu.fing.context.management</groupId>
    <version>0.0.1-SNAPSHOT</version>
    <artifactId>context-source-polling-${eventName}</artifactId>
     <name>edu.fing.context.management:${eventName}</name>
```

Figura 82 – Plantilla del archivo pom.xml

Apéndice 3. Encapsulamiento de la Gestión de Drools Fusion

En el Motor CEP se exponen dos endpoints con dos interfaces distintas, una para gestionar el Motor CEP y otra para alimentarlo. Ambas interfaces son implementadas por sus respectivos *beans*, los cuales referencian al *bean* "DroolsCoreBean" que contiene el código referente a Drools Fusion. Esta clase tiene dos atributos estáticos que se utilizan para gestionar Drools como se muestra en la Figura 83.

```
private static KieSession kSession = null;
private static KieServices kServices = null;
```

Figura 83 – Atributos para gestionar Drools.

Un objeto de tipo "KieServices" es mantenido de forma estática en la clase simplemente para verificar la inicialización del motor. El atributo se inicia como *null* cuando se crea la instancia de la clase, y deja de serlo cuando se inicializa Drools por medio de una invocación al servicio "initializeDroolsContext()".

El "KieSession" es la vía para interactuar con el motor. Una "KieSession" es creada cuando se inicializa el motor y es mantenida por la aplicación a lo largo de su ejecución. Cuando se despliega una nueva versión de reglas, la sesión es sustituida por una nueva, creada a partir de una "KieBase" que tiene las nuevas reglas. Según la documentación de Drools, desplegar una nueva versión actualiza la versión en todas las sesiones ya creadas, pero esto no funciona cuando se configura Drools en modo *stream.* Esto fue un inconveniente importante y se explica en la Sección 5.3.1.

El impacto de este *bug* en la solución es que cada vez que se despliega una nueva versión de reglas, todos los eventos que estaban activos en la memoria se pierden. Si bien esto puede parecer una falta importante, no tiene un impacto tan grande debido a dos características de la plataforma y del tipo de eventos que maneja el motor:

- Los eventos que llegan al Motor CEP, usualmente son informados reiteradas veces y tienen una validez relativamente corta. Un ejemplo de esto es una posición del usuario.
- Desplegar nuevas versiones no debería ser hecho con demasiada frecuencia.

Por esto, y lo expuesto en la Sección 5.3.1, donde no se encontró una solución completa, se utilizó la versión 6.0.0 Final.

Para realizar la gestión de versiones en Drools se creó una clase "DroolsUtils" en la cual se utiliza parte del código publicado en [52] que es parte de los *tests* unitarios de Drools. Este código es de gran ayuda ya que encapsula la lógica para crear y desplegar los JAR con las versiones de Drools.

Para evitar problemas de concurrencia, se utilizó una instancia de la clase "ReentrantReadWriteLock" del paquete java.util.concurrent. Cuando una versión se está desplegando, se bloquean los eventos entrantes y se desbloquean cuando la misma

se termina de desplegar. A su vez, este bloqueo permite que no se despliegue una nueva versión mientras otra aún no ha terminado de desplegarse, pero sí permite insertar muchos eventos al mismo tiempo. En otras palabras, cuando se despliega una versión se realiza un bloqueo "escritor" mientras que cuando se inserta un evento se hace un bloqueo "lector".

Funciones de Utilidad

El Motor CEP tiene la responsabilidad de informar al Razonador de Contexto las situaciones detectadas, las cuales para el componente son eventos complejos. Dichos eventos son detectados en las reglas, por lo que se hace necesario definir una forma estructurada para que el administrador indique cuándo una situación fue detectada. Para esto, se definió una función Drools que al ser invocada, notifica al Razonador de Contexto la nueva situación. La función se llama "situationDetected" y recibe el identificador del usuario, el nombre de la situación y los datos contextuales.

Además, se implementaron algunas funciones de utilidad para ayudar al administrador a realizar invocaciones externas desde las reglas. Las mismas se encuentran en la clase "CepUtils" y se pueden observar en la Figura 84.

```
//Invoca mediante http-get la url recibida, retorna la respuesta como string
public static String httpGet(String url);

//Conviente un json en formato String a un Map
public static Map <String, Object> jsonToMap(String json);

//Conviente un xml en formato String a un Map
public static Map <String, Object> xmlToMap(String xml);

//Elimina acentos del String recibido
public static String stripAccents(String str);
```

Figura 84 – Funciones de utilidad en "CepUtils".

Apéndice 4. Configuración de Funcionalidades en el Razonador de Contexto

En este apéndice se muestra el detalle de implementación de la parte encargada de realizar las configuraciones en el Razonador de Contexto.

En la Figura 85 se muestran los componentes del Razonador de Contexto que permiten la configuración de las distintas funcionalidades.

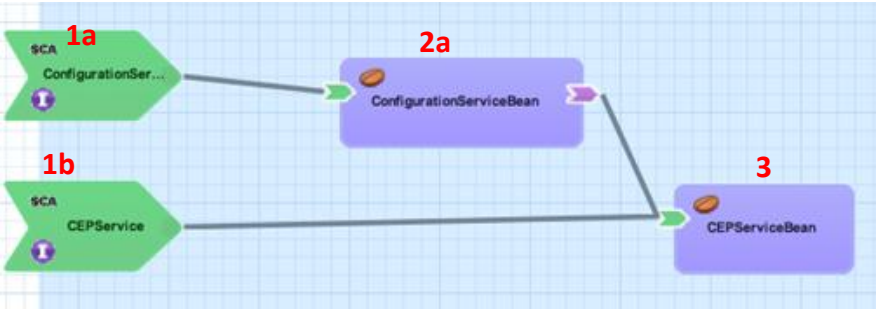


Figura 85 – Componentes involucrados en la configuración de funcionalidades.

En la Tabla 17 se describen las responsabilidades de los componentes mostrados en la Figura 85.

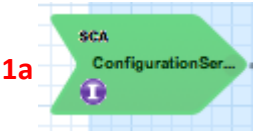
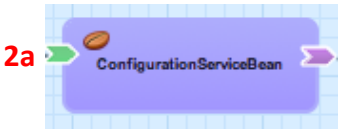
Componente	Representación gráfica	Descripción
ConfigurationService		Expone un *endpoint* con un *binding* SCA para la Consola de Administración que permite la manipulación de los datos que administra el Razonador.
ConfigurationServiceBean		Realiza consultas y persiste configuraciones en la base de datos.
CEPService		Expone un *endpoint* con un *binding* SCA para que la Consola de Administración manipule las reglas.
CepServiceBean		Encargado de: • Manipular el versionado de reglas. • Desplegar versiones y verificar la compilación de reglas invocando remotamente al Motor CEP.

Tabla 17 – Descripción de los componentes involucrados en la configuración de funcionalidades.

Como se puede observar en la Figura 85, el *bean* "ConfigurationServiceBean" utiliza el *bean* "CEPServiceBean" para las operaciones que realizan lógica sobre las reglas. Para poder realizar esto, SwitchYard permite a un *bean* consumir un servicio inyectando directamente una referencia al servicio que se desea. En la Figura 86 se puede observar cómo el *bean* "ConfigurationServiceBean" inyecta la interfaz "CepService" utilizada cada vez que se crea una situación, ya que al dar de alta una situación se crea la regla que la detecta. Esta regla es validada por el Motor CEP antes de ser persistida. Para validarla se utiliza una invocación remota al Motor CEP, no un *binding* SCA como en los casos anteriores debido al problema mencionado en la Sección 5.3.4. Luego de ser validada la regla, se crea una nueva versión de reglas que contiene todas las reglas de la última versión y la nueva que se está creando.

```
@Service(ConfigurationService.class)
public class ConfigurationServiceBean implements ConfigurationService {

    @Inject
    @Reference
    private CEPService cepService;
```

Figura 86 – Consumo de un servicio.

Apéndice 5. Manejo de Encabezados de SwitchYard

En este apéndice se muestra el manejo de los encabezados de SwitchYard. Estos encabezados fueron utilizados para transportar el nombre y URL de los servicios desde los Servicios Virtuales al Gateway de Adaptación.

Manejo de encabezados en los Servicios Virtuales

Cuando llega un mensaje a un Servicio Virtual, la ruta Camel "SetService" presente en dicho componente, le agrega dos encabezados al mensaje con información del servicio. Uno de ellos es el nombre del servicio que está siendo invocado de manera de poder consultar en el Gateway de Adaptación si el servicio tiene algún itinerario configurado. El otro encabezado es la URL del servicio, de manera que el Invocador de Servicios conozca qué URL debe invocar. En la Figura 87 se puede observar cómo se añaden los encabezados "serviceUrl" y "serviceName" utilizando Camel.

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="switchyard://VirtualService"/>
        <setHeader headerName="serviceUrl">
            <simple>http://service.com/attractions-provider/AttractionsService</simple>
        </setHeader>
        <setHeader headerName="serviceName">
            <simple>AttractionsService</simple>
        </setHeader>
        <log message="Virtual Service - message received: ${body}"/>
        <to uri="switchyard://AdaptationGatewayService"/>
        <log message="Virtual Service - response received: ${body}"/>
    </route>
</routes>
```

Figura 87 – Ruta Camel "SetService"

Manejo de encabezados en el Gateway de Adaptación

Para obtener los encabezados añadidos por los Servicios Virtuales, los *beans* permiten acceder al contexto de intercambio de SwitchYard. Este contexto representa una instancia de invocación a un servicio, y provee un "conducto" para los mensajes que viajan como parte de dicha invocación. Inyectando dicho contexto en un *bean* es posible obtener y añadir propiedades para cada invocación. En la Figura 88 se muestra como el *bean* "AdaptationServiceBean" inyecta el contexto con la anotación "@Inject" y accede al identificador y URL del servicio, que fueron añadidos previamente por el Servicio Virtual correspondiente.

```
@Service(AdaptationService.class)
public class AdaptationServiceBean implements AdaptationService {

    @Inject
    private Context context;

    @Override
    public String setItinerary(String message) {

        String serviceName = this.context.getPropertyValue("serviceName");
        String serviceUrl = this.context.getPropertyValue("serviceUrl");
```

Figura 88 - Uso del Contexto de SwitchYard.

Apéndice 6. Detalles de Implementación del Administrador de Adaptaciones

En este apéndice se detalla el ruteo basado en itinerario que es el orquestador del mensaje dirigiéndolo por las adaptaciones que deben aplicarse. Además se describen detalles de las adaptaciones profundizando en el ruteo basado en contenido debido a su complejidad.

Ruteo Basado en Itinerario

Cuando el Gateway de Adaptación envía el "AdaptedMessage" al Administrador de Adaptaciones, éste es procesado por el componente "RoutingSlip" que es el encargado de ejecutar el ruteo basado en itinerario. En la Figura 89 se puede observar la implementación de dicho componente.

```
public class RoutingSlip extends RouteBuilder {

    public void configure() {

        from("switchyard://RoutingService").process(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                AdaptedMessage adaptedMessage = exchange.getIn().getBody(AdaptedMessage.class);
                exchange.getIn().setBody(adaptedMessage.getMessage());
                exchange.getIn().setHeader("adaptationDirective", adaptedMessage.getAdaptations());
                exchange.getIn().setHeader("itinerary", adaptedMessage.getItinerary());
            }
        }).log("Received message for 'RoutingSlip' : ${body}").routingSlip().header("itinerary")
            .log("Finish routing");
    }
}
```

Figura 89 – Componente "RoutingSlip".

En la implementación de este componente se utiliza el contenedor de mensaje de Apache Camel llamado "Exchange" que mantiene información durante todo el ciclo de ruteo del mensaje. Con el fin de aplicar las adaptaciones, el componente "RoutingSlip" configura el "Exchange" para ser utilizado por el resto de las adaptaciones de la siguiente manera:

- *header*: contiene la directiva de adaptación para el resto de las adaptaciones.
- *body*: contiene el mensaje.

Además, este componente añade otro *header* con el itinerario, para que sea utilizado por el método "routingSlip()" de Apache Camel que aplica el ruteo basado en itinerario, como se observa en la Figura 89.

Detalles de las Adaptaciones

Se implementaron cuatro componentes Camel que aplican la lógica de las adaptaciones ofrecidas en la Consola de Administración. Para el prototipo se decidió implementar enriquecimiento, filtrado, ruteo basado en contenido, invocación externa y retardador de mensajes. Sin embargo, como trabajo a futuro pueden ofrecerse más adaptaciones de la antes mencionadas.

Estos componentes utilizan el “Exchange” de Camel para obtener la información necesaria para aplicar la adaptación que corresponda. Esta información se encuentra en el *header* “adaptationDirective” que tiene la lista de adaptaciones, como se muestra en la Figura 89. Cada componente obtiene el primer elemento de dicha lista, utiliza la información y la elimina de la lista. De esta manera cada componente puede acceder al primer elemento sabiendo que contiene su información y no la de otra adaptación.

Ruteo Basado en Contenido

Esta adaptación es implementada por el componente “ContentBasedRouter” y permite el ruteo hacia cualquier adaptación que se desee. Este componente utiliza un árbol binario que es tomado de la lista de adaptaciones, donde cada nodo de decisión es un XPath condicional y las hojas contienen la información para aplicar la adaptación.

Para saber que adaptación aplicar primero se evalúa el XPath de la raíz, si éste es verdadero se realiza la recursión sobre el sub-árbol izquierdo, sino se realiza sobre el sub-árbol derecho. En cada paso de la recursión se chequea si es un XPath o una adaptación, si es un XPath se evalúa y se realiza la recursión al igual que en el nodo raíz, y si es una adaptación se termina la recursión habiendo encontrado la adaptación que debe aplicarse.

Luego de encontrar la adaptación, el mensaje es enviado al componente correspondiente a la adaptación. Para esto se utiliza el patrón *Recipient List* de los EIP que permite rutear un mensaje dinámicamente, pudiendo indicar en tiempo de ejecución el destino del mensaje.

Apéndice 7. Generación de Reglas en la Consola de Administración

En este apéndice se muestra la plantilla usada para la generación de la regla sugerida para la creación de una situación.

En la Figura 90 se puede ver la plantilla utilizada para generar la regla, que además de los datos contextuales recibe el nombre de la situación y los nombres de los eventos asociados a las fuentes de contexto. También se pueden observar comentarios que ayudan al administrador a entender algunas restricciones que se solicitan.

```
package edu.fing.drools;
import java.util.HashMap;
import java.util.Map;
import edu.fing.commons.dto.ContextualDataTO;
/**
* Se definen los eventos que llegan desde las Fuentes de Contexto.
* Si un evento es utilizado por mas de una fuente, se debe definir solo una vez, o de lo contrario utilizar distintos nombres.
**/
<#list rule.mappedContextData as contextSourceTO>
    declare ${rule.situationName}_${contextSourceTO.eventName}
    @role(event)
    <#list contextSourceTO.contextData as contextDatum>
     ${contextDatum}:String
    </#list>
    end
</#list>

<#list rule.mappedContextData as contextSourceTO>
    rule "${rule.situationName}_${contextSourceTO.eventName} message mapper"
    when
        message:ContextualDataTO(this.getEventName() == "${contextSourceTO.eventName}")
    then
        System.out.println("Mapping input message to ${rule.situationName}_${contextSourceTO.eventName}...");
     Map<String,Object> info = message.getInfo();

     /*La variable donde se carga la fuente de contexto DEBE llamarse inputEvent*/
     ${rule.situationName}_${contextSourceTO.eventName} inputEvent = new ${rule.situationName}_${contextSourceTO.eventName}();

     /*Cargar los inputs en el evento*/
    <#list contextSourceTO.contextData as contextDatum>
     inputEvent.set${contextDatum?cap_first}((String)info.get("${contextDatum}"));
    </#list>
     insert(inputEvent);
    end
</#list>

    rule "Notify Situation ${rule.situationName}"
    when
     //Condicion para detectar situación
    then
        HashMap<String,Object> contextualData = new HashMap<String,Object>();
        /*
        Cargar hash de data contextual
        -El nombre de la variable DEBE ser contextualData
        -Las claves del hash deben ser strings "entre comillas"
        */
    <#list rule.selectedOutputData as outputDatum>
     contextualData.put("${outputDatum}",/*VALOR_${outputDatum}*/);
    </#list>
        /*Sustituir USER_ID por el id del usuario de tipo String*/
     situationDetected(/*USER_ID*/, "${rule.situationName}",contextualData);
    end
```

Figura 90 – Plantilla FreeMarker para generar una regla.

Antes de generar la regla y la situación, se realiza una validación y en caso que sea correcta se pregunta al administrador el número de versión de reglas en la cual incluirá la regla actual. Si la regla no es válida, se despliega un mensaje de error con los errores detectados permitiendo que se corrijan. La validación involucra evaluar que todos los datos contextuales de entrada y salida declarados sean utilizados, y que la nueva regla compile correctamente.

Apéndice 8. Reglas Drools Definidas en los Casos de Uso

En este apéndice se presentan las reglas utilizadas para los casos de uso del Capítulo 6. Se utilizaron dos servicios externos: uno que obtiene la ciudad en la que se encuentra un usuario a partir de sus coordenadas geográficas y otro que obtiene el estado del tiempo de las ciudades interesadas. Para simplificar la solución se decidió utilizar los departamentos de Uruguay, ya que se requiere estandarizar los nombres de los lugares y resulta demasiado complejo hacerlo por ciudad.

Reglas para situación "InCity"

Esta regla detecta cuándo un usuario se encuentra en una ciudad a partir de sus coordenadas geográficas. Para obtener la ciudad se utilizó un servicio de Google de *reverse geocoding* [53] que recibe como parámetros en la URL la latitud y longitud.

Cada vez que llega una nueva actualización de la posición del usuario, se invoca a la URL con la nueva posición del usuario y se inserta o actualiza el evento "InCity", que representa la ciudad donde se encuentra un usuario.

A continuación, en la Figura 91 se presenta el código de la regla que detecta la situación "InCity".

```
package edu.fing.drools;

import java.util.HashMap;
import java.util.Map;
import java.util.List;
import edu.fing.commons.dto.ContextualDataTO;
import edu.fing.cep.engine.utils.CepUtils;
/**
* Se definen los eventos que llegan desde las Fuentes de Contexto.
* Si un evento es utilizado por mas de una fuente, se debe definir solo una vez, o de lo
contrario utilizar distintos nombres.
**/
   declare InCity_UserPosition
   @role(event)
        latitude:String
        userId:String
        longitude:String
   end

   declare InCity
   @role(event)
        city:String
        userId:String
   end
```

```
rule "InCity_user_position message mapper"

  when
    message:ContextualDataTO(this.getEventName() == "UserPosition")
  then
    System.out.println("Mapping input message to InCity_UserPosition...");
    Map<String,Object> info = message.getInfo();
    /*La variable donde se carga la fuente de contexto DEBE llamarse inputEvent*/
    InCity_UserPosition inputEvent = new InCity_UserPosition();
    /*Cargar los inputs en el evento*/
    inputEvent.setLatitude((String)info.get("latitude"));
    inputEvent.setLongitude((String)info.get("longitude"));
    inputEvent.setUserId((String)info.get("userId"));
    insert(inputEvent);
  end
  rule "InCity_UserPosition city detection insert"
  when
    userPosition:InCity_UserPosition()
    not InCity(this.getUserId() == userPosition.getUserId())
  then
    System.out.println("Checking if the user is in a valid city...");
    String city = detectCity(userPosition);
    if (city != null){
      InCity inCity = new InCity();
      inCity.setCity(city);
      inCity.setUserId(userPosition.getUserId());
      insert(inCity);
    }
  end
  rule "InCity_UserPosition city detection update"
  no-loop true
  when
    userPosition:InCity_UserPosition()
    inCity:InCity(this.getUserId() == userPosition.getUserId())
  then
    System.out.println("Checking if the user is in a valid city...");
    String city = detectCity(userPosition);
    if (city != null){
        System.out.println("DEPARTAMENTO: '" + city + "'");
      modify(inCity){setCity(city)};
    }
  end
  rule "Notify Situation InCity"
  when
      //Condicion para detectar situación
    inCity:InCity()
  then
    /*
    Cargar hash de data contextual
    -El nombre de la variable DEBE ser contextualData
    -Las claves del hash deben ser strings "entre comillas"
    */
    HashMap<String,Object> contextualData = new HashMap<String,Object>();
    contextualData.put("city",inCity.getCity());
    situationDetected(inCity.getUserId(), "InCity",contextualData);
  end
```

```
function String detectCity(InCity_UserPosition userPosition) {

      String response = CepUtils.httpGet(
          "https://maps.googleapis.com/maps/api/geocode/json?latlng=" +
          userPosition.getLatitude() + "," + userPosition.getLongitude() +
          "&result_type=administrative_area_level_1&" +
          "key=AIzaSyDBV2XLwr5AmhhymFzXQWq_HKXPOV7MQ2A&lang=en");
      Map<String,Object> responseMap = CepUtils.jsonToMap(response);
      List list = ((List)responseMap.get("results"));
      if (!list.isEmpty()){
        Map place = (Map) ((List)responseMap.get("results")).get(0);
        String city = ((String)place.get("formatted_address"))
          .replace("Departamento de","")
          .replace("Department","")
          .replace(", Uruguay","")
          .trim();
        city = CepUtils.stripAccents(city.toLowerCase());
        return city;
      }
      return null;
  }
```

Figura 91 – Regla para detectar la situación "InCity".

Reglas para situación "InCityRaining"

Esta regla detecta cuándo un usuario se encuentra en una ciudad en la cual está lloviendo. Para su definición se utiliza el evento complejo "InCity" definido anteriormente. A partir de la ciudad obtenida y de la información de la Fuente de Contexto que consulta los estados del tiempo de los 19 departamentos periódicamente, se detecta si en esa ciudad está lloviendo.

Cuando se recibe una nueva actualización, se verifica si se tiene información para ese departamento y se decide si se debe actualizar la información de los eventos existentes o insertar nuevos.

A continuación, en la Figura 92 se presenta el código de la regla que detecta la situación "InCityRaining".

```
package edu.fing.drools;

 import java.util.HashMap;
 import java.util.Map;
 import edu.fing.commons.dto.ContextualDataTO;

declare CityWeatherUpdate
 @role(event)
 weather:String
 city:String
 end

declare InCityRaining
 @role(event)
 userId:String
 city:String
 end

 declare CityWeather
 @role(event)
 raining:boolean
 city:String
 end

rule "CityWeather message mapper"
when
   message:ContextualDataTO(this.getEventName() == "CityWeather")
then
   System.out.println("Mapping input message to CityWeatherUpdate...");
   Map<String,Object> info = message.getInfo();
   List list = ((List) info.get("list"));
   if (!list.isEmpty()) {
      for (Object object : list) {
         Map cityInfo = (Map) object;
         String cityWeather = ((String) ((Map) ((List)
cityInfo.get("weather")).get(0)).get("main"));
         String cityName = (String) cityInfo.get("name");
         cityName = cityName.replace("Departamento de","")
                  .replace("Department","")
                  .replace(", Uruguay","")
                  .trim();
         cityName = CepUtils.stripAccents(cityName.toLowerCase());
         CityWeatherUpdate inputEvent = new CityWeatherUpdate();
         inputEvent.setCity(cityName);
         inputEvent.setWeather(cityWeather);
         insert(inputEvent);
      }
   }
end
```

```
rule "CityWeather insert detection"

when
   cwu: CityWeatherUpdate()
   not CityWeather( this.getCity() == cwu.getCity() )
then
   CityWeather cityWeather = new CityWeather();
   cityWeather.setCity(cwu.getCity());
   boolean isRaining = cwu.getWeather().equals("Rain") || cwu.getWeather().equals("Shower
rain") ||
      cwu.getWeather().equals("Thunderstorm") || cwu.getWeather().equals("Snow");
   cityWeather.setRaining(isRaining);
   insert(cityWeather);
   System.out.println("CityWheather inserted. " + cwu.getCity() + " " + cwu.getWeather() + " " +
isRaining);
end

rule "CityWeather update detection"
no-loop true
when
   cwu: CityWeatherUpdate()
   cityWeather: CityWeather( this.getCity() == cwu.getCity() )
then
   boolean isRaining = cwu.getWeather().equals("Rain") || cwu.getWeather().equals("Shower
rain") ||
      cwu.getWeather().equals("Thunderstorm") || cwu.getWeather().equals("Snow");
   modify(cityWeather){ setRaining(isRaining) };
   System.out.println("CityWheather modified. " + cwu.getCity() + " " + cwu.getWeather() + " "
+   isRaining);
end

rule "Notify Situation InCityRaining"
when
   //Condicion para detectar situación
   inCity: InCity()
   cityWeather: CityWeather( this.getCity() == inCity.getCity(), this.isRaining() )
then
   HashMap<String,Object> contextualData = new HashMap<String,Object>();
   System.out.println("Complex event detected. InCityRaining!!!"+ inCity.getCity() + " " +
cityWeather.isRaining());
   /*
   Cargar hash de data contextual
   -El nombre de la variable DEBE ser contextualData
   -Las claves del hash deben ser strings "entre comillas"
   */
   contextualData.put("city",inCity.getCity());
   /*Sustituir USER_ID por el id del usuario de tipo String*/
   situationDetected(inCity.getUserId(), "InCityRaining", contextualData);
end
```

Figura 92 - Regla para detectar la situación "InCityRaining".