Maestría en Ingeniería de Software

Universidad Blas Pascal - Universidad Nacional de la Plata Facultad de informática.

2019

Tesis

Identificación y Clasificación de Patrones de Diseño de Servicios *Web* para mejorar *QoS.*

Director: Dr. Gustavo Rossi
Alumno: Lic. Pablo Aguerre

Índice

1. Introducción

1.1. Motivación y estado del arte

Los servicios *Web* representan el Santo Grial de la computación ya que permiten que cualquier aplicación de *sofwtare* en el mundo pueda comunicarse potencialmente con otra (desarrollada en una tecnología diferente o similar) tal como se señala en [1]; motivo por el cual, hoy en día se puede mencionar que éstos conforman la iniciativa tecnológica más relevante en cuanto a los negocios. En [2] se indica que, en la realidad, la adopción al uso de servicios *Web* no es una alternativa, sino que es una necesidad ya que éstos han cambiado los fundamentos de acuerdo a la manera en que *Information Technology (IT)* es utilizada en los negocios empresariales.

Service Orientated Architecture (SOA) es una arquitectura *loosely-coupled*[1] designada para cumplir con las necesidades de negocio de las organizaciones. En una primera mirada, esta definición parecería ser simplista pero tal como se comenta en [3] una arquitectura orientada a servicios no requiere del uso de servicios *Web*; aunque para la mayoría de las organizaciones éstos serían el enfoque de implementación más simple para cumplir con una arquitectura de este estilo. En el pasado, aplicaciones débilmente acopladas han confiado en otros estilos como *CORBA* y *DCOM*. La definición establecida sirve ya que hace foco en que lo importante no es la arquitectura en si sino en cumplir con las necesidades organizacionales. En términos sencillos, el *SOA* de una organización puede verse como un conjunto de servicios *Web* (u otras tecnologías) disponibles para otra organización. En resumen, pueden existir capacidades de infraestructura comunes como logueo y autenticación, pero para la mayoría el *SOA* de una organización es diferente al *SOA* de cualquier otra.

Los conceptos arquitectónicos de *SOA* no son nuevos – Muchos de estos han evolucionados de ideas originalmente introducidas por *CORBA, DCOM* y otros. A diferencia de estas iniciativas previas, la promesa clave de *SOA* es habilitar ágilmente procesos de negocio a través de estándares abiertos basados en la interoperabilidad. A pesar de que estos estándares son importantes hay que recordar que éstos no son

[1] *Loosely-coupled* significa acoplamiento débil en castellano y en este contexto se refiere a que los cambios en las diferentes capas que conforman *SOA* no conllevarán a modificaciones entrelazadas relevantes y altos niveles de dependencia si estas *layers* son diseñadas de manera adecuada.

arquitectura y que la arquitectura no son implementaciones. Al fin del día, es la implementación de una arquitectura bien diseñada quien va a proveer beneficios de negocio (en *SOA* generalmente mediante la implementación servicios *Web*); no la arquitectura propiamente dicha.

Una variante importante del estilo de arquitectura *SOA* son los microservicios los cuales posibilitan estructurar una aplicación como una colección de servicios débilmente acoplados que promueven la modularidad beneficiando el mantenimiento de los mismos de manera aislada. Los microservicios son de granularidad fina, usan protocolos livianos y se consideran el nuevo enfoque tal como se menciona en [4].

En [5] se indica que los servicios *Web* suelen ser *Application Program Interfaces (APIs)* que pueden ser accedidas dentro de una red (principalmente Internet) y son ejecutados en el sistema que los aloja. Esta definición alberga muchos tipos diferentes de sistemas, pero el caso común de uso se refiere a clientes y servidores que se comunican mediante mensajes *XML* que siguen el estándar *SOAP*; aunque en los últimos años se ha popularizado un estilo de arquitectura de software conocido como *Representational State Transfer (ReST)* el cual ha supuesto una nueva opción de estilo de uso de los servicios *Web* intentando emular al protocolo *HTTP* o protocolos similares mediante la restricción de establecer la interfaz a un conjunto conocido de operaciones estándar (por ejemplo *GET*, *PUT*, …) por lo cual este último se centra más en interactuar con recursos de estado que con mensajes.

Se puede observar a través de la implementación de los estilos arquitectónicos relevantes relacionados a servicios *Web* sobre la base del paradigma *Object Orientated Programming (OOP)* que ofrecer un lenguaje, una arquitectura y una plataforma para resolver problemas no es suficiente. Es innegable que cualquier plataforma de desarrollo necesita de un *layer* de patrones para poder obtener el máximo beneficio optimizando su potencial. Los patrones ingresaron a la escena del desarrollo de *software* después de la publicación de *Design Patterns* en 1995: *Elements of Reusable Object-Oriented Software, the ground breaking book from the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)* y desde la publicación del *GoF book* se han realizado numerosos estudios. En la presente tesis, solamente se hará foco en los patrones de diseño los cuales forman parte de un concepto de ingeniería de *software* para describir soluciones

recurrentes a problemas comunes en el área de diseño de software tal como se indica en [6].

En [7] se menciona que debido al incremento en el número de servicios *Web* disponibles en el mercado, existe un interés creciente en poder proveer metodologías de diseño de servicios que cumplan con especificaciones de calidad de servicio o *Quality of Service (QoS)* además de calidad funcional. Aunque no es propósito de esta tesis indagar sobre las metodologías de diseño existentes está claro que el uso de patrones de diseño es fundamental para poder brindar soluciones sobre conceptos comunes basados en la experiencia y es en la naturaleza del dominio de los servicios *Web* donde éstos aportan su máximo potencial en el cumplimiento de los objetivos de negocio de las empresas.

1.2. Objetivo

El objetivo general de esta propuesta es elaborar un inventario de patrones de diseño relacionado al desarrollo y consumo de servicios *Web* para mejorar *QoS*; por tal motivo, se tendrán en cuenta trabajos realizados en el dominio en cuestión y arquitecturas orientadas a servicio y microservicios de aplicaciones actuales.

En el cumplimiento de dicho propósito primario se destacan las siguientes contribuciones:

- Creación de una fórmula específica para evaluar la calidad de los servicios de manera cualitativa acotada a la aplicación en el área establecida (incluyendo interoperabilidad como principio fundamental en el diseño de los servicios).
- Establecimiento del vínculo entre los servicios *Web* y un compendio de antipatrones fundacionales lo cual sirve para la compresión de la ontología del concepto y la posterior elaboración del inventariado.
- Catálogo de patrones de diseño de servicios *Web* evaluados cualitativamente en base a la fórmula de *QoS* conveniente al dominio lo cual permite mejorar la calidad de los componentes en la nube. Además, este catálogo se clasifica en base a modelos, tecnologías y definiciones de la actualidad de forma comparativa lo cual brinda un aporte sustancial al momento diseñar o actualizar una arquitectura orientada a servicios.
- Identificación y clasificación de patrones de diseño en los servicios de una aplicación líder en la actualidad sobre las cual se aportan mejoras de *QoS* en las *APIs* para

maximizar los beneficios y cubrir posibles brechas existentes.

1.3. Organización de la tesis

Este trabajo está organizado en nueve capítulos tal como se detalla a continuación:

- Capítulo 1 - Introducción y objetivos: Desarrollo de la introducción general, objetivo principal y objetivos específicos.
- Capítulo 2 - Servicios *Web*: Establecimiento del marco referencial teórico relacionado al dominio de servicios *Web*. Definición de servicios *Web*, estilos existentes, tendencia y análisis de componentes sobre arquitecturas orientadas a servicios.
- Capítulo 3 – Trabajos relacionados: Se indaga sobre estudios previos relacionados a servicios *Web y* arquitecturas en la nube, después, se examinan los trabajos vinculados a patrones de diseño, servicios y *QoS*.
- Capítulo 4 - Restricciones de diseño en el domino de servicios *Web*: Análisis respecto a restricciones de diseño asociadas a las arquitecturas orientadas a servicios.
- Capítulo 5 - Patrones y su vínculo con los servicios *Web*: Ontología de patrones y análisis de estudios actuales asociados con patrones de diseño aplicados en el desarrollo y consumo de servicios.
- Capítulo 6 - Concepto de *QoS* y su categorización teniendo en cuenta su vínculo con los servicios *Web*. Relevamiento y concordancia de su categorización teniendo en cuenta documentos de investigación recientes.
- Capítulo 7 – Catálogo de patrones de diseño de servicios para mejorar *QoS*: Categorización de patrones de diseño del dominio en base a los estilos de servicios existentes, principios de diseño de servicios y su relación con la mejora de *QoS*.
- Capítulo 8 - Utilización de patrones de servicios en un sistema referente de la actualidad: Relevamiento y valoración de patrones de diseño asociados al uso de los distintos estilos de servicios *Web* del sistema *Semi-automated Business Research Environment (SABRE)* para brindar una propuesta que permita mejorar *QoS* cubriendo posibles brechas existentes y optimizando los beneficios organizacionales.
- Capítulo 9 - Conclusiones y posibles estudios futuros: Conclusiones generales de la tesis y propuesta de trabajos futuros a realizar sobre temas relacionados al dominio en cuestión.

2. Servicios *Web*

2.1. Definición

La definición más adecuada la establece *W3C*[2] en [8] e indica que un servicio *Web* es un sistema de software diseñado para admitir interoperabilidad respecto a las interacciones máquina en máquina realizadas a través de una red de trabajo. En su versión original, éste posee una interfaz descripta en un formato que puede ser procesado por una máquina siguiendo una especificación formal (*WSDL*) y en su versión minimalista se destaca por seguir especificaciones más livianas (como *JSON)*.

El concepto clave asociado al dominio de los servicios *Web* es la promesa de interoperabilidad. La arquitectura de los servicios *Web* está basada en el envío de mensajes *XML* en formato *SOAP*[3] o *JSON/POX* para el enfoque *ReST*. Los archivos en formato *XML* o *JSON* pueden ser representados como una cadena de caracteres *ASCII*, la cual puede ser transferida sencillamente de máquina en máquina. Las implicaciones de esto son significantes:

- No importa en que el lenguaje de programación fue codificado el *Software* cliente del servicio *Web*.
- El cliente no necesita conocer que si hay un motor *SOAP* o qué procesador se encuentra corriendo del lado del servidor *Web*.
- No importa qué tipo de computadora envía el mensaje o sobre qué tipo de sistema operativo corre la misma.
- No importa la ubicación en el mundo respecto desde donde se envía el mensaje.

2.2. Componentes

Para que las transacciones de los servicios *Web* se ejecuten de manera exitosa, los componentes involucrados deben comportarse de manera esperada. A continuación, se sintetizan los componentes principales relacionados a los estilos arquitectónicos descriptos en la próxima sección:

---

[2] *World Wide Web Consortium (W3C)* es un consorcio responsable del desarrollo de tecnologías que brindan interoperabilidad incluyendo especificaciones, guías, software y herramientas. Ésta maneja las especificaciones *SOAP, WSDL, XML, XML Schema* y *HTTP*, entre otras. Ver más en <www.w3.org>

[3] Para el propósito de esta tesis, los mensajes *SOAP* o envío de mensajes *XML* siguiendo la especificación *SOAP* son sinónimos.

XML (Extensible Markup Language): Es un metalenguaje para documentos estructurados que permite describir otros lenguajes de marcado mediante gramáticas descriptas en *XML schemas* (*SOAP* y *WSDL* son derivados de éste). Ver más en: <www.xml.com>

JSON (JavaScript Object Notation) es un formato de archivo estándar, abierto y legible por humanos que se usa para transmitir objetos de datos que consisten en una matriz compuesta por pares de atributos clave-valor, tal como indica [9].

SOAP (Simple Object Access Protocol): Es un protocolo simple basado en *XML* el cual permite a las aplicaciones intercambiar información a través de *HTTP* o *JMS* de forma menos usual. El objetivo final de *SOAP* es especificar el formato de intercambio de mensajes independiente de cualquier tipo de arquitectura de software o hardware. Para más información referirse a: <http://www.w3schools.com/soap/soap_intro.asp>

HTTP (Hypertext Transport Protocol): Es un estándar, usado por los servicios, que fue desarrollado para facilitar la transferencia de *requests* desde un navegador o *browser* hacia un servidor *Web*. Éste define 8 métodos básicos los cuales indican la acción a efectuarse sobre el recurso identificado (*HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS* y *CONNECT*). Ver más en: <http://www.w3.org/Protocols/>

JMS (Java Message Service): Es una *API* que soporta el uso de cola de mensajes creada por *Sun Microsystems*. Ésta permite a los componentes de aplicaciones basados en la plataforma *Java2* crear, enviar, recibir, y leer mensajes mediante una comunicación asíncrona principalmente, aunque también es factible la sincronicidad.
Existen dos modelos de la *API JMS*, los cuales son:
Modelo *Point to Point (P2P)*: Éste cuenta con solo dos participantes, uno que envía el mensaje y otro que lo recibe, asegurando la llegada del mensaje ya que, si el receptor no está disponible para aceptar el mensaje o atenderlo, de cualquier forma, se le envía el mensaje y este se agrega en una cola del tipo *FIFO* (*First Input First Output*) para luego ser recibido según haya entrado.
Modelo *Publish/Subscribe*: Éste cuenta con múltiples participantes, uno que publica temas o eventos y varios que ven estos temas, a diferencia del modelo punto a punto este modelo tiende a tener más de un consumidor.

Ambos modelos pueden ser síncronos mediante el método *receive* y asíncronos por medio de un *MessageListener* aunque la implementación subyacente sigue siendo una cola de mensajes. La especificación *SOAP* sobre *JMS* proporciona un conjunto normalizado de directrices mediante la utilización de un transporte comunicación que permite la interoperabilidad entre las implementaciones de distintos proveedores. Ver más en: <https://www.ibm.com/support/knowledgecenter/cs/SSAW57_8.5.5/com.ibm.websphere.nd.multiplatform.doc/ae/twbs_soapjmstransportwbs.html>

Web Services Description Language (WSDL): Es una especificación que indica como describir una pieza de *software* en términos de las llamadas a los métodos abstractos, independientes del lenguaje de programación el cual los implementa, a los cuales se responde. Es de destacar, que un *WSDL* también posee una sección concreta la cual involucra detalles respecto a como realizar una conexión con el servicio; si un servicio *Web* es accedido usando *HTTP, JMS, FTP* o *SMTP* se especificarán las entradas en esta sección. Finalmente, un documento *WSDL* se considera extensible ya que permite la descripción de sus *endpoints* y mensajes sin tener en cuenta el formato de los mensajes o protocolos de transporte en uso durante la comunicación. Para más información ver: <http://www.w3.org/TR/wsdl>

Swagger ReSTful API Documentation Specification (OpenAPI Specification): Esta generalización puede ser conceptualizada de manera análoga a lo que *WSDL* representa para los servicios *SOAP* en el sentido de que *Swagger* posibilita documentar la definición del servicio *ReSTful* de tal manera que sus clientes se adhieran a esta firma. El proyecto *Swagger-UI* puede utilizar estos archivos para mostrar la *API* y adicionalmente *Swagger-Codegen* para generar clientes de servicios *Web* en varios lenguajes de programación. Las especificaciones *Swagger* se escriben en formatos *JSON Schema* o *Yamel Ain't Markup Lenguage (YAML)* que son lenguajes de datos entendibles por seres humanos con una sintaxis mínima en comparación a *XML*. Más información al respecto se encuentra disponible en: <https://docs.swagger.io/spec.html>

Web Application Description Language (WADL): Es un lenguaje descriptivo en *XML* sobre servicios *Web HTTP* el cual es legible por máquinas. Éste fue creado por *Sun Microsystems* y modela los recursos proporcionados por un servicio y las relaciones entre ellos. Su objetivo es simplificar la reutilización de los servicios *Web* que se basan en la

arquitectura *HTTP* existente de la *Web* (es otro de los equivalentes en *ReSTful* a lo que *WSDL* representa para *SOAP*). Ver más en <https://www.w3.org/Submission/wadl/>

JSON API: Es una especificación, para los servicios de estilo *ReST+JSON,* en la cual se declara como los clientes deben solicitar que se busquen o modifiquen recursos, y como los servidores deben responder a esas solicitudes. Esta individualización está diseñada para minimizar tanto el número de solicitudes como la cantidad de datos transmitidos entre clientes y servidores; lo cual se logra sin comprometer la legibilidad, la flexibilidad o la capacidad de descubrimiento de recursos. Puede decirse que, *JSON API* es una especificación de los servicios *ReST* similar a lo que representa el protocolo *SOAP* para los servicios clásicos. Para más datos ver: <https://jsonapi.org/>

Hypertext Application Language (HAL): Es un lenguaje de formato simple que proporciona una manera consistente y fácil de vincular los recursos de las interfaces de los servicios. La adopción de *HAL* permite que las *APIs* sean explorables, cumpliendo con el principio de auto descubrimiento, lo que hace que la documentación de los servicios sea sencilla. Para más información se recomienda ver el siguiente *link*: <http://stateless.co/hal_specification.html>

Universal Discovery Description Integration (UDDI): Esta especificación describe como un cliente potencial de un servicio *Web* clásico puede aprender respecto a las capacidades de éste y obtener información básica respecto a como realizar un primer contacto con el sitio vinculado. Los tipos de registros en *UDDI* pueden ser públicos, privados o semi-privados: Un directorio público permite que cualquier individuo o sistema en el planeta pueda acceder a la información ingresada en el registro en contraposición a un directorio privado el cual solo existe detrás del *firewall* de una organización sólo accedido por los miembros de la misma y, por último, un registro semi-privado es el enfoque utilizado respecto a limitar la visibilidad de acuerdo a un conjunto definido de *partners*[4]. Para más detalle referirse a <http://uddi.xml.org/technical-notes>

Eureka: Es un servicio de estilo *ReST* el cual es usado normalmente con *Amazon Web*

[4] *Partners* se refiere a organizaciones asociadas al negocio de la organización responsable del directorio semi-privado.

Services (AWS) para el registro y descubrimiento de servicios *Web*. Éste se denomina *Eureka Server* y, también, realiza funciones de balanceo de carga y recuperación o *failover* sobre servicios registrados. Vale la pena remarcar que, la empresa *Netflix* provee un servicio de balanceo de carga sofisticado basado en *Eureka*, teniendo en cuenta factores como el tráfico, uso de recursos y condiciones de error para proporcionar capacidades de recuperación avanzadas. Para más información ver <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>

URI (Uniform Resource Identifier): Es una cadena de caracteres que sirve para identificar de manera unívoca un nombre o recurso (página, servicio, documento, *e-mail*, etc) a través de una red o sistema. Las *URIs* pueden ser clasificadas como localizadores (*URLs),* nombres (*URNs*) o ambas. Un *Uniform Resource Name (URN)* se comporta análogamente como el nombre de una persona, mientras que un *Uniform Resource Locator (URL)* sería la dirección del individuo. En resumen, una *URN* define la identidad de un ítem y la *URL* provee el método para encontrarlo. Para más información se recomienda leer <http://tools.ietf.org/html/rfc6570>

Docker: Es una tecnología diseñada para facilitar la creación, implementación y ejecución de aplicaciones mediante el uso de contenedores. Los contenedores le permiten a un desarrollador empaquetar una aplicación y su infraestructura con todas las partes que necesita, como librerías u otras dependencias, y distribuirla como una sola unidad. Para más detalles referirse a: <https://www.docker.com/>

Kubernates (*k8s*): Es un sistema orquestador de contenedores utilizado para automatizar la implementación, gestión y el escalamiento de aplicaciones empaquetadas en imágenes. Originalmente fue diseñado por *Google* y ahora es mantenido por *Cloud Native Computing Foundation*. Funciona con una gama de tecnologías de contenedores, incluido *Docker*. Los servicios en la nube suelen ofrecen una plataforma o infraestructura basada en *Kubernetes* como servicios *PaaS* o *IaaS*. Finalmente, algunos proveedores, como *AWS*, también proporcionan sus propias distribuciones de marca *Kubernetes*. Ver más información en: <https://cloud.google.com/kubernetes-engine>

2.3. Estilos de servicios

Una arquitectura orientada a servicios no se restringe a un tipo o estilo de servicio tal como se indica en [10]. Las capacidades funcionales y no funcionales de un servicio, y el grado de cumplimiento respecto a los principios *SOA* o de microservicios conforman el criterio de diseño en cuanto al estilo apropiado. Cada uno éstos tienen sus ventajas y desventajas, aunque el estilo clásico se encuentra maduro, el estilo minimalista se visualiza como la tendencia por su simplicidad al no estar acoplado fuertemente al cumplimiento de estándares, y la combinación de ambos suele ser la alternativa más apropiada. La mejor manera de resolver la controversia respecto a la selección del estilo arquitectónico adecuado es poder entender las diferencias de estos tipos más allá de las preferencias de un arquitecto o diseñador.

2.3.1. Mensajes *XML* siguiendo la especificación *SOAP*

El estilo clásico, el cual se hace énfasis en la sección 2.1 debido al beneficio de interoperabilidad, es responsable de exponer las funcionalidades principales a los consumidores comúnmente descriptas como *RPCs (Remote Pocedure Calls)*. Éste debe cumplir con ciertos estándares conocidos como *WS-** o Pila de protocolos de servicios *Web*. Los tópicos de seguridad, manejo y comunicación de protocolos pueden extienden la lista de estándares *WS-** que definen y soportan este estilo maduro. En definitiva, se pueden obtener altos niveles de interoperabilidad respecto al intercambio de mensajes cuando los consumidores adoptan la pila de estándares *WS-** (ver figura 1).

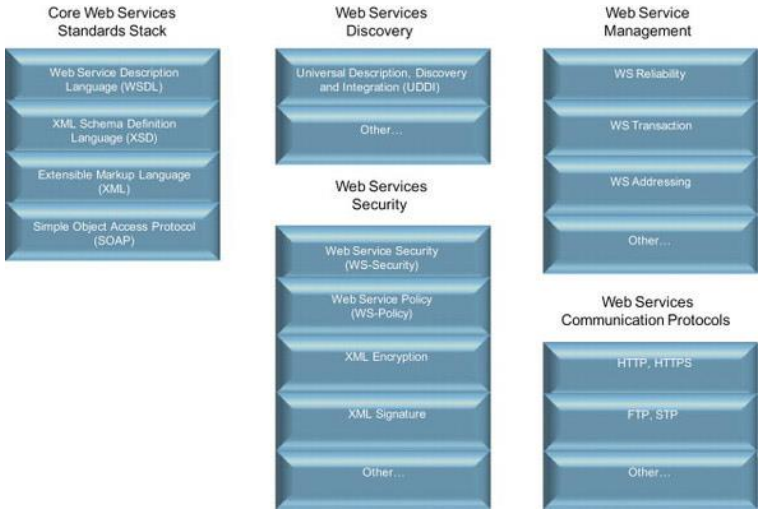


Figura 1: Pila de estándares de servicios *Web*. Fuente [11].

Los 4 estándares principales o *Core Web Service Standards Stack* fueron resumidos en la sección 2.2 y a continuación se presenta la interrelación desde un enfoque actualizado

respecto a las versiones actuales haciendo énfasis en la especificación *WSDL* más estable: Documento *WSDL*: Actualmente existen dos especificaciones primarias en juego. La primera, *WSDL* 1.1, es ampliamente reconocida e implementada a través de la mayoría de las herramientas y tecnologías [12]. La segunda, *WSDL* 2.0, es la revisión más reciente de la especificación *WSDL* 1.1 e incluye un número valioso de mejoras y extensiones [13]. En este momento, la especificación *WSDL* 1.1 es usada e implementada en la mayoría de las organizaciones, aunque la versión *WSDL* 2.0 es la recomendada.

Un *WSDL* definido para un servicio puede también hacer referencia o definir intrínsecamente en el mismo el contenido de una interfaz granular usando *XML* Schema <types/>. Una buena manera de pensar respecto a un *XML Schema* se refiere a que es un lenguaje de metadatos restringidos para los mensajes *XML*. Como parte de la pila de protocolos de servicios *Web*, un *XML Schema* es usado para definir y restringir el contenido de un documento *XML*. *XML Schema Definition Language (XSD)* es una especificación *W3C* ampliamente reconocida e implementada [14].

El mensaje que es intercambiado entre los colaboradores *SOA* (clientes y servicios) es un derivado de las definiciones establecidas en el *XML Schema* sobre la interfaz del servicio. Para los servicios *Web,* el *XML Schema* que define estos mensajes se encuentra referenciado o embebido dentro de un documento *WSDL*. Por lo tanto, *XML* se usa tanto para la descripción del *XML Schema* como para la interfaz estructural *WSDL* posibilitando la comunicación a los consumidores y los servicios independientemente de las tecnologías de desarrollo.

La especificación *SOAP* sigue el paradigma de envoltura o *envelope* el cual es un análogo al papel de envoltura de una carta la cual se usa para enviar cartas o documentos através del servicio postal de correos. El correo postal incluye la dirección del destino (*endpoint* del servicio), y usualmente el tipo de servicio postal a utilizar como si es por normal o por aire (análogo al protocolo del servicio y el *binding*). Dentro del *envelope* se encuentra la carta o documento a enviar análogos a la información establecida dentro del *envelope* de los *XML requests*. El *SOAP envelope* posee un encabezado o *header* (<soap:header/>) el cual pude contener información descriptiva de la dirección e información de entrega, y el cuerpo o *body* (<soap:body/>) que contiene el mensaje *XML* codificado. En caso de un error, *SOAP* posee una sección (<soap:fault/>) la cual posee información del error a retornar. También se puede referir al contenido del *SOAP body* como el "*payload"*[5].

[5] *Payload* significa en castellano carga útil ya que dentro del *body* del mensaje *XML* se encuentra la información relevante a enviar.

Fundamentalmente, un servicio *Web* puede ser considerado como un proveedor de funcionalidades de entrega, manipulación e intercambio de datos bajo un contexto determinado. Los valores de la información se encuentran encapsulados dentro de *XML tags*, que a su vez se encuentran encapsuladas dentro del *SOAP body* el cual, como se mencionó está envuelto dentro del *envelope* usado para intercambiar mensajes entre los colaboradores.

Los servicios *Web* pueden ser diseñados para ocultar información estructural subyacente como así también especificaciones únicas de una tecnología de implementación en particular. Esta capacidad adicional de *SOA* y los servicios *Web* brinda ayuda respecto a la solución de problemas de integración empresarial. El consumo de aplicaciones no requiere conocimiento específico alguno respecto a las fuentes de información o implementaciones técnicas usadas en la exposición del servicio. Los consumidores simplemente hacen un pedido de la información que necesitan y utilizan solo el contenido de la respuesta obtenida.

Otra de las ventajas de los servicios *Web* que siguen este estilo tradicional es la información de entrega. Información de entrega se refiere al proceso de adquirir información a través uno o más sistemas, racionalizando esta información en una vista individual, transformando los datos de manera consistente y entregando la información a más de un cliente.

Para los conceptos vinculados a información de entrega e integración empresarial, se puede observar que las ventajas asociadas al uso de servicios *Web* son primariamente una combinación de interoperabilidad entre los colaboradores resultante del apego a las especificaciones *WS*-* y un soporte tecnológico de alta escala. El cumplimiento de la pila de protocolos de los servicios *Web* permite un alto grado de interoperabilidad. Los servicios son ampliamente ubicuos en este sentido ya que todos los *vendors*[6] tecnológicos de gran relevancia proveen de alguna manera soporte relacionado a servicios *Web* y los estándar *WS*-*.

Algunos pueden mencionar que la complejidad respecto a la pila de protocolos de servicios *Web* puede devenir en latencia operativa resultante la cual negará (o al menos reducirá) el valor de los servicios pero, al igual que en cualquier tecnología, existen ventajas y desventajas a tener en cuenta.

[6] *Vendors* significa vendedores en castellano haciendo alusión a las empresas proveedoras de tecnología.

2.3.2. *ReST y ReSTful*[7]

Los servicios *Web* que conforman el estilo de arquitectura de *software ReST* se denominan *ReSTful (RWS)* y fueron introducidos por primera vez en la tesis doctoral del Dr. Roy Fielding [15]. Una interpretación común a lo que el Dr. Fielding define como servicios de estilo *ReST* es que éstos son una imitación de las operaciones básicas existentes en la *World Wide Web* basada en protocolos de comunicación, tales como *HTTP*. Los servicios *ReSTful* se centran en identificar y proveer recursos, en lugar del intercambio de transacciones individuales (una página *Web HTML* es un ejemplo fácil de entender considerada como recurso en este contexto). En esencia, éste se basa en la noción de un recurso identificado, el tipo de recurso, y el conjunto básico de comandos *HTTP*. Estos comandos *HTTP* también son similares a las actividades *CRUD* (crear, leer, actualizar y eliminar). *ReST* también define el concepto de "estado" como aquel siendo transferido de un participante a otro y por lo tanto se considera que con este estilo los servicios no poseen estado en sí mismos, sino que lo transfieren (en el ejemplo de la página *Web* el estado se transfiere desde el solicitante de la página *HTML* al servidor). Todo lo requerido para poder completar la interacción se provee en el *request*, y el resultado se expone en la *response.*

Tabla 1: Actividades *CRUD* comparadas con los comandos *HTTP*. Fuente propia.

Actividades *CRUD*	Comandos *http*
Crear	*POST*
Leer	*GET*
Actualizar	*PUT*
Eliminar	*DELETE*

En [16] se define un modelo de maduración de servicios de estilo *ReST* dividiendo los elementos principales de este enfoque en tres niveles (ver figura 2).

Nivel 0: Los servicios tienen una sola *URI* y utilizan un solo método *HTTP* (normalmente *POST*), ignorando efectivamente el resto de los verbos. De manera similar, los servicios basados en *XML-RPC* que envían datos como *Plain Old XML (POX)*.

Nivel 1: Estos servicios emplean múltiples *URIs* de recursos, pero solo un único verbo

[7] Más allá de que técnicamente referirse a *ReST* implica arquitectura mientras que *ReSTful* son los servicios *Web*, es común en la jerga mencionar *ReST y ReSTful* de manera análoga, aunque el uso correcto es "servicio *ReSTful* o servicio de estilo *ReST"* (en las referencias a trabajos o artículos esto no se va a modificar por temas de autoría, pero en definiciones propias de ésta tesis se intentará indicar de manera adecuada la especificación correcta cuando sea el caso).

HTTP (generalmente *POST*). Son mejores que el nivel 0 debido a que cada recurso se identifica por separado mediante una *URI* única.

Nivel 2: Tales servicios admiten varios de los verbos *HTTP* en cada recurso expuesto: *POST*, *GET*, *PUT* y *DELETE*. Aquí, el estado de los recursos, los cuales normalmente representan entidades comerciales, se puede transferir a través de la red. Los servicios de este nivel albergan numerosos recursos disponibles en múltiples *URIs* y el diseñador de servicios espera que la gente se esfuerce por dominar las *API*s, generalmente leyendo la documentación suministrada *(Swagger)*.

Nivel 3: *Hypermedia as the Engine of Application State (HATEOAS)* es el nivel más maduro en el cual las respuestas a las solicitudes del cliente contienen controles de hipermedia que les ayudan a decidir cuál es la siguiente acción que se puede tomar (por ejemplo, controles de paginación de resultados o vínculos a recursos). Este nivel fomenta el descubrimiento sencillo del servicio proporcionando respuestas auto explicativas además de brindar múltiples recursos y verbos.

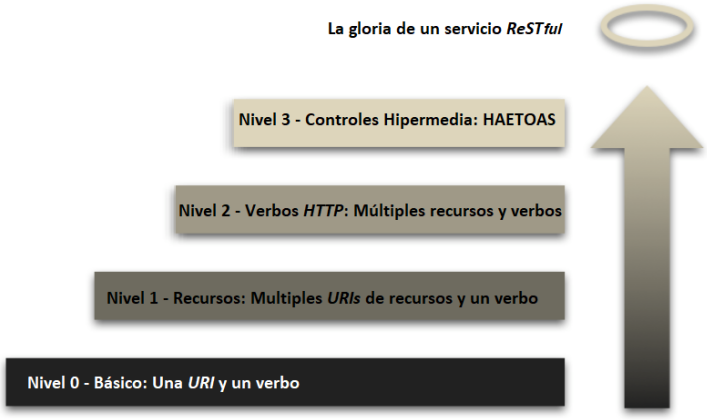


Figura 2: *Richardson Maturity Model (RMM)*. Fuente propia en base a [16].

[17] menciona que una interfase *ReST* es diseñada de manera eficiente para la transferencia de datos logrando así optimización en el uso común de la *Web*. En este tipo de estilo arquitectónico simplificado no hay necesidad del cumplimiento de las especificaciones *WSDL, XML Schema* y *SOAP*. El conjunto mínimo de estándares necesarios para solicitar, adquirir, retornar, y renderizar la información ya está integrado en la infraestructura subyacente del navegador, *HTTP* y la *Web*. La interacción es bastante simple y fácil de entender, aunque en una arquitectura *SOA*, el modelo *ReST* no está exento de limitaciones. Como un ejemplo, se describe la importancia de una "interfaz uniforme" destacando que ésta degrada la eficiencia, ya que la información se transfiere

en un formato normalizado en lugar de uno que es específico a las necesidades de una aplicación. La capacidad de mantener la coherencia es de valor para las interacciones de los servicios ya que promueve el principio de interoperabilidad.

Independientemente, el *ReST* es un modelo sencillo, limpio, y bien probado. No se requiere un conjunto importante de normas complejas y se observan ventajas de rendimiento sobre los servicios *Web* basados en estándares, aunque su inclusión en el mundo empresarial lo vincula a la necesidad imperativa de seguir especificaciones como *Swagger*.

2.4. Otras clasificaciones de servicios *Web*

A continuación, una síntesis de otras clasificaciones de servicios *Web,* que no abarcan los estilos antes mencionados de modo directo, pero valen la pena considerar en el marco de esta tesis.

En [18] se define una clasificación de las *APIs* de los servicios*,* basada en los modelos de abstracción de datos y las acciones que se permiten, denominada *Amundsen Maturity Model (AMM)*. Cuanto más alto sea el nivel, más se desvincula la interfaz de los modelos internos de las implementaciones.

Nivel 0 – Centrado en la base de datos: En este nivel no hay abstracción entre la capa de persistencia del servicio (por ejemplo, la base de datos) y el cliente. Esto significa que cualquier modelo de datos que se encuentre en la base de datos es lo que el cliente obtiene y, por ende, no se puede cambiar la base de datos sin romper el cliente. La manera más fácil de evidenciar esta categoría es observando los resultados, es decir, cuando se obtienen datos en un formato crudo, como un documento *MongoDB*[8] o en un atributo genérico, llamado *output,* que posea la respuesta *SQL*[9] directa.

Nivel 1 – Centrado en objetos: En este nivel se le presenta al cliente los objetos de implementación, por lo tanto, se puede modificar la base de datos sin afectar a los clientes, pero no así la implementación del servicio (*middleware framework*). Los servicios *Web* de estilo clásico (*WS-*)* de la capa intermedia o *middleware*, se incluyen en esta categoría,

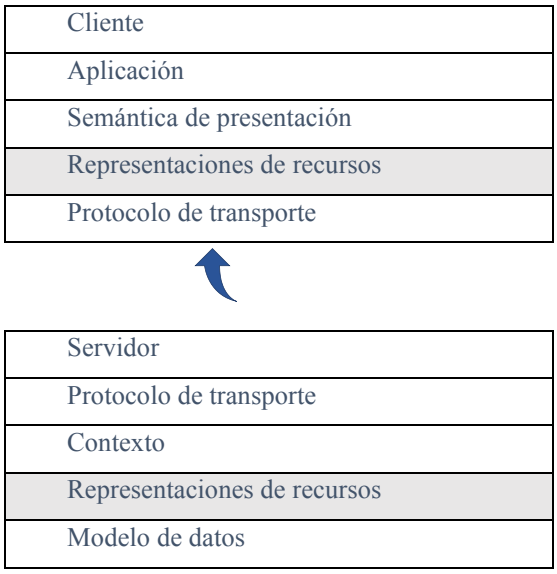
[8] *MongoDB* es una de las bases de datos de estilo documental más usadas en el mercado. Para más información ver: <https://www.mongodb.com/>.

[9] *Standard Query Language (SQL)* es el lenguaje estándar para manipulación, guardado y recuperación de información utilizado en las bases de datos relacionales como *Postgres, SQL Server y Oracle.* Para más detalle ver: <https://www.w3schools.com/sql/>.

como así también *Restify Mongoose*[10] es un modelo en los de estilo *ReST*. Para identificar esta categoría, la persona que realiza la evaluación debe poder distinguir si los datos, según lo aceptado y expuesto por la *API*, están relacionados con la implementación del servicio, ya que, si es así es una mala señal, la cual indica que no será posible la evolución de la *API* independiente de la implementación.

Nivel 2 – Centrado en recursos: En este nivel se logra desacoplar la interfaz del servicio de la implementación y los modelos de persistencia mediante un modelo de abstracción basado en recursos. Esto se puede lograr utilizando el patrón representante o *representor pattern*[11], donde los modelos de datos internos se reemplazan con representaciones de recursos antes de ser compartidos a través de la capa de transporte. Esta categoría se evidencia, normalmente, por las operaciones *CRUD* asociadas a recursos en forma de *URIs*.

Tabla 2: Patrón representante. Fuente propia basada en [19].



Nivel 3 – Centrado en acciones de producto que se pueden realizar: En el último nivel, se piensa en las *APIs* de los servicios como parte de un producto, por lo cual, éstas se modelan de la mano con las historias de usuario del producto o *product owner*, lo que resulta en operaciones expuestas que reflejan las necesidades de los consumidores. Con las interfaces de los servicios diseñadas de esta forma, a los clientes ya no se les pide que echen un vistazo a los datos y descubran qué pueden hacer con ellos, sino que, se les

[10] *Restify Mongoose* es una tecnología de servicios estilo *ReST* que se utiliza para abstraer los modelos de la base de datos *MongoDB* en forma de recursos. Para más datos ver: <https://github.com/saintedlama/restify-mongoose>.

[11] El *representor pattern* se basa en la idea de que existen recursos únicos, independientes de los *media types* y los modelos de datos de dominio, por lo tanto, mediante la creación programática de representaciones asociadas a recursos, la generación de solicitudes y respuestas de mensajes hipermedia se simplifica significativamente. Para más información ver: [19].

presenta directamente las acciones que pueden realizar. Finalmente, en esta categoría, se proporcionan las acciones disponibles en tiempo de ejecución (estilo *ReST* con controles *hypermedia*) junto a los *media types*[12] necesarios (como *application/vnd.restful + json* de la especificación *JSON API*).

En [20] se menciona otra clasificación de servicios *Web,* basada en la manera de desarrollar los servicios y los componentes físicos que se necesitan:

Software como servicio o *Software as a Service* (*SaaS*) es un modelo de distribución de software en el que un proveedor externo aloja aplicaciones y las pone a disposición de los clientes a través de *Internet*. Dependiendo del nivel de acuerdo del servicio o *Service Level Agreement* (*SLA*), los datos del cliente pueden almacenarse localmente, en la nube o ambos. También, las organizaciones pueden integrar aplicaciones *SaaS* con otro *software* utilizando interfaces de servicios (*APIs*). Existen aplicaciones *SaaS* en tecnologías comerciales fundamentales, como el correo electrónico, la gestión de ventas, recursos humanos, facturación y colaboración. Los principales proveedores de *SaaS* son *Salesforce*, *Oracle*, *SAP*, *Intuit* y *Microsoft*.

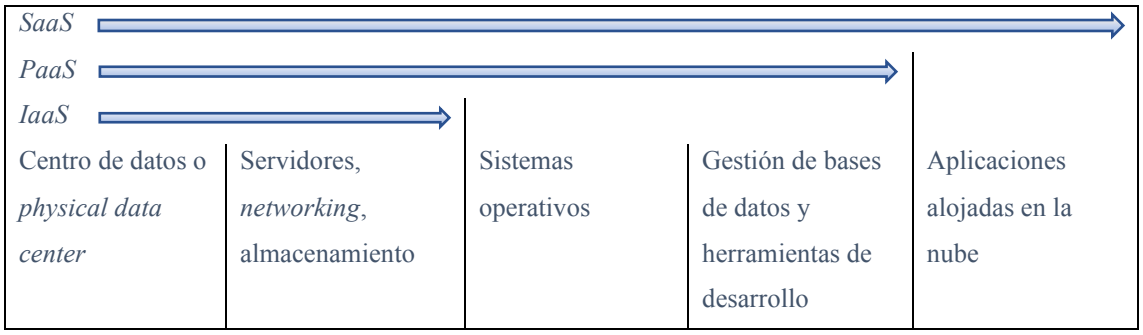
En el modelo de Plataforma como servicio o *Platform as a Service* (*PaaS*), los desarrolladores esencialmente rentan todo lo que necesitan para construir una aplicación, confiando en proveedores en la nube (como *Amazon* o *Google*) en cuanto a las herramientas de desarrollo, la infraestructura y los sistemas operativos. *PaaS* simplifica enormemente el desarrollo de servicios *Web*; desde la perspectiva del desarrollador, ya que toda la administración del *backend* se lleva a cabo entre bastidores. Las principales ofertas que brindan los proveedores de *PaaS* son: Herramientas de desarrollo, *middleware*, sistemas operativos, administración de bases de datos e infraestructura.

Infraestructura como servicio o *Infrastructure as a Service* (*IaaS*) es el modelo donde se venden servidores virtuales y servicios de cómputo relacionados. En *IaaS*, los proveedores alojan los componentes de infraestructura, que tradicionalmente se encuentran en un centro de datos local o *local data center*, como los servidores, el almacenamiento, el *hardware* de red, y la capa de virtualización o *hipervisor*. El proveedor de *IaaS* también suministra una serie de servicios adicionales para acompañar

[12] Los *media types* son los tipos de contenido que se aceptan tanto en las solicitudes como en las respuestas de los servicios. Alguno de los ejemplos más comunes son *application/json*, *application/xml*, *text/html*, *text/plain* y *multipart/form-data.* Para más datos respecto a todos los *media types* leer: <https://en.wikipedia.org/wiki/Media_type>.

esos componentes de infraestructura, como por ejemplo, monitoreo, gestión del registro de servicios, seguridad, balanceo de carga, respaldo, replicación y recuperación. *Amazon Web Services (AWS)* y *Google Cloud Platform (GCP)* son proveedores independientes de *IaaS*.

Tabla 3: Alcance de *SaaS*, *PaaS* e *IaaS*. Fuente propia.



Finalmente, otro tipo de categorización que se hace evidente es según el tipo de funcionalidad o lógica que encapsulan los servicios, el alcance del potencial de reúso y como se relacionan éstos con los dominios existentes dentro de la empresa. Como resultado, hay tres clasificaciones genéricas de servicio en base a la funcionalidad que representan los modelos principales utilizados en una arquitectura orientada a servicios:

- Servicios de utilidad: Son los servicios de *backend* que se dedican a proporcionar una funcionalidad de utilidad transversal y reutilizable, como el registro de eventos, la notificación y el manejo de excepciones. Lo ideal es que sean independientes de la aplicación, ya que puede consistir en una serie de capacidades que se basan en múltiples sistemas y recursos empresariales.
- Servicios de entidad: En casi todas las empresas, habrá documentos de modelos de negocio que definen las entidades comerciales relevantes de la organización. Ejemplos de entidades comerciales incluyen clientes, empleados y facturas. El modelo de servicio de entidades representa un servicio centrado en el negocio que basa sus límites y contexto funcionales en una o más entidades comerciales.
- Servicios de tareas: Un servicio de negocio con un límite funcional directamente asociado con una tarea o proceso de negocio principal específico se basa en el modelo de servicio de tareas. Este tipo de servicio tiende a tener menos potencial de reutilización y generalmente se posiciona como el controlador de una composición responsable de componer otros servicios más agnósticos del proceso. Un ejemplo es aquel servicio cuya única capacidad expuesta requerida sea iniciar su proceso de negocio principal encapsulado.

2.5. Arquitectura orientada a servicios y sus límites en el marco de esta tesis

No es objetivo de esta tesis proveer un análisis exhaustivo de los elementos que coforman la arquitectura *SOA,* sobre todo relacionado al detalle minucioso de conceptos como *Web Services Business Process Execution Language* (*WS-BPEL orchestration),* o de microservicios*;* por lo cual, a continuación, se provee un resumen en cuanto a los valores fundamentales necesarios para entenderlos ya que, tal como se menciona en [21], una arquitectura muestra los componentes principales de una solución de *software* y sirve como molde para el diseño pudiendo ser definida a través de diferentes modelos representativos los cuales ponen en manifiesto los diferentes componentes de una solución de *software* compleja.

SOA se trata de conectar las necesidades del cliente con las capacidades de la empresa, independientemente de su panorama tecnológico o límites arbitrarios existentes tal como lo define [22]. A un nivel muy bajo, es una arquitectura técnica con el apoyo de los formatos y protocolos estándar. A un nivel más general, representa un cambio dentro de la empresa para romper los silos organizacionales y sistemas monolíticos de información que permitan la flexibilidad en como las soluciones de los clientes se montan. No existen dos arquitecturas orientadas a servicios empresariales que posean el mismo aspecto. *SOA* es un estilo arquitectónico con un puñado de elementos comunes, temas y estrategias múltiples de implementación. Una modelo nominal y representativo se puede identificar con el fin de comprender mejor esta solución tecnológica compleja (ver figura 3).

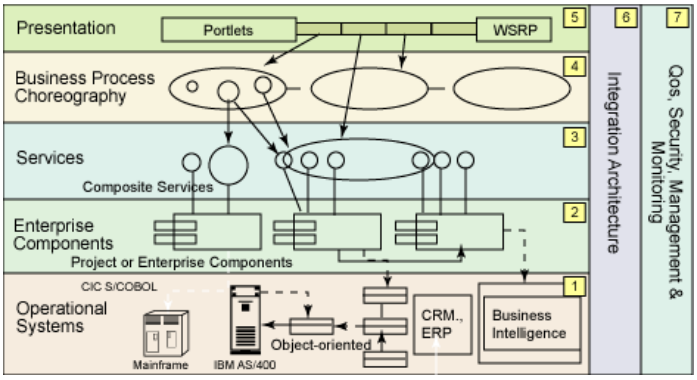


Figura 3: Las capas de una arquitectura *SOA*. Fuente [23].

Capa 1: *Operational System*: La capa de sistemas operacionales representa a los sistemas de legado existentes, aquellos sistemas ligados a la gestión de relaciones con los clientes (*CRM) y* la planificación de recursos empresariales (*ERP)*, aplicaciones antiguas

orientadas a objetos como así también aplicaciones referidas a *Business-inteligence*.

Capa 2 – *Enterprise Components*: La capa de componentes empresariales es responsable de realizar la funcionalidad negocio y mantener *QoS* de los servicios expuestos. Esta capa suele utilizar tecnologías basadas en *containers*, tales como servidores de aplicaciones para ejecutar componentes los cuales gestionan la carga de trabajo, proveen alta disponibilidad y otorgan balanceo de carga mediante técnicas de *clustering* (una de las tecnologías más reconocidas basadas en contenedores es *EJB*).

Capa 3 – *Services*: Esta capa posee los servicios expuestos los cuales el negocio opta financiar. Éstos pueden ser descubiertos o accedidos de forma estática y luego invocados, o posiblemente, mediante una coreografía de un servicio compuesto por varios. Descripciones de servicios, *QoS SLAs*[13], y también otros componentes claves relacionados con la definición de meta datos se incluyen en esta capa.

Capa 4 – *Business Process Choregraphy*: Los servicios individuales proporcionan un valor incremental para una organización, pero es probable que nunca transformen la manera en la cual se hacen los negocios. Los procesos de negocio, sin embargo, representan orquestaciones de uno o más servicios que resuelven un problema de negocio. Los servicios están agrupados en un flujo lógico (descrito como orquestación o coreografía) para resolver algún tipo de problema de negocio dese el inicio hasta el final. Algunas de las herramientas visuales reconocidas en el desarrollo de coreografías relacionadas al diseño de flujos de negocio son *IBM® WebSphere® Business Integration Modeler or Websphere Application Developer Integration Edition* en el caso de las pagas y *BPEL Designer* como *plugin* del proyecto *Eclipse* en el caso de las gratuitas. Los motores *BPEL* para coreografías de servicios *Web* más reconocidos son *IBM® WebSphere® Process Server* y *Oracle BPEL Process Manager* entre los pagos y *jBPM* de *jBOSS* entre los gratuitos, aunque vale destacar que su uso, en los años recientes, es reducido.

Capa 5 - *Access or Presentation*: La capa de presentación, por lo general, queda fuera del alcance de las discusiones en torno a una arquitectura *SOA* aunque cada vez toma más relevancia. Actualmente existen una mayor convergencia de las normas y tecnologías (*Remote Portlets Version 2.0*), lo cual hace que esta capa busque aprovechar y alivianar el uso de servicios *Web* en la interfaz de la aplicación (*UI*).

Nivel 6 – *Integration Enterprise Service Bus (ESB)*: Este nivel habilita la integración de

[13] *Service Level Agreement (SLA)* significa en castellano acuerdos a nivel de servicios y es citado en relación a los acuerdos respecto a la calidad de servicios.

los servicios a través de un conjunto de capacidades gestionadas por el *ESB* el cual es considerado como un elemento de infraestructura utilizado como un *broker*[14] de transacciones, responsable de vincular interfaces y grupos de datos (permitiendo que clientes y servicios con expectativas diferentes se comuniquen sin problemas), enrutar el tráfico a los servicios adecuados basados en lógica interna y ejecutar otras soluciones de *brokering* las cuales agregan valor a los servicios otorgando un desacople apropiado entre los servicios y sus clientes. Un componente arquitectónico del *ESB* a destacar es el administrador de las reglas de negocio o *Business Rules Management System (BRMS)* cuyo fin es desarrollar, almacenar, editar y ejecutar las reglas comerciales que definen la lógica del comportamiento empresarial de los servicios y suelen ser accedidos a través de agentes de servicio.

Nivel 7 – *QoS*: Esta capa posee las capacidades requeridas para monitorear, manejar y mantener la calidad de los servicios tales como seguridad, rendimiento y disponibilidad. Este es un proceso en segundo plano a través de mecanismos de evaluación y respuesta e instrumentos que monitorean la salud de las aplicaciones *SOA*, incluyendo las implementaciones de todos los estándares importantes de *WS-Management* [24] y otros protocolos y normas que implementan la calidad del servicio para una arquitectura orientada a servicios.

En una arquitectura orientada a servicios, los paquetes de software se subdividirán en pequeñas unidades de negocio interconectadas. Estas pequeñas unidades de negocio se interconectan entre sí a través de diferentes protocolos y así se logra otorgarle valor de negocio al cliente de manera ágil. Entonces, ¿En qué se diferencia la arquitectura de microservicios *(MSA)* de la *SOA*? En una palabra, *SOA* es un patrón de diseño arquitectónico y los microservicios son una metodología de implementación para llevar a cabo *SOA,* el cual se implementa con protocolos livianos, o podemos decir que el microservicio es un tipo de *SOA* tal como se señala en [4].

Tabla 4: En la siguiente tabla se enumeran brevemente las características de *SOA* y *MSA*. Fuente propia.

Componente	*SOA*	Microservicios
Patrón de diseño	*SOA* puede considerarse un paradigma de diseño donde sus componentes de software son	*MSA* es un estilo de *SOA*. Se considera una especialización de la arquitectura orientada a servicios.

[14] *Broker* en este contexto se refiere a que un *ESB* es una especie de gestor de transacciones.

	expuestos al mundo exterior para su uso en forma de servicios.	Algunos microservicios granulares no necesitan ser expuestos al mundo exterior (ejemplo, una tarea específica de una tecnología determinada que no necesita ser expuesta como unidad de negocio al cliente).
Dependencia	Las unidades de negocio son usualmente dependientes entre sí.	Las unidades de negocio deben ser independientes entre sí.
Tamaño	El tamaño del *software* es mayor a los enfoques tradicionales por unidad de negocio.	El tamaño del *software* es considerado relativamente pequeño si se analiza de manera granular, aunque en su totalidad, comparado al *SOA* base, puede llegar a ser mayor debido a las múltiples tecnologías involucradas en un entorno empresarial.
Tecnología	El *stack* de tecnologías en considerado pequeño en comparativa a los microservicios.	Los microservicios son heterogéneos en naturaleza ya que tecnologías exactas se utilizan para realizar una tarea específica. Los microservicios pueden ser considerados como un conglomerado de muchas tecnologías.
Autonomía y foco	Las aplicaciones *SOA* pueden ser desarrolladas para llevar a cabo múltiples tareas de negocio.	Cada microservicio es desarrollado para ejecutar una sola tarea de negocio específica.
Despliegue	El despliegue puede consumir tiempo sobre los diferentes entornos sino es planificado de forma debida.	El despliegue adecuado suele conllevar poco tiempo en cada entorno debido a la naturaleza granular de los microservicios.
Costo y efectividad	Aparentemente más costosos debido al cumplimiento de estándares e interdependencias, aunque si las tecnologías involucradas son homogéneas puede ser considerado relativamente barato en comparación a *MSA*.	Aparentemente menos costosos debido a que por naturaleza no necesitan seguir una especificación y son modulares; aunque en un entorno empresarial, si deben seguir una especificación y teniendo en cuenta la heterogeneidad de las tecnologías como un todo puede incurrir en costos muy elevados.
Escalabilidad	Menor nivel de escalabilidad debido a posible interdependencia entre servicios.	Completamente escalables por definición.

3. Trabajos relacionados

3.1. Preliminar

El relevamiento de trabajos relacionados se llevó a cabo, de manera sistemática, mediante el uso de los motores de búsqueda *Google Scholar*[15], *Springer*[16] e *IEEE Xplore*[17] filtrando los resultados por relevancia y el año de publicación. Primero, se indagó respecto a trabajos emergentes sobre servicios *Web* teniendo en cuenta palabras clave como "*servicios Web, SOA*, microservicios y composición de servicios" entre 2019 – 2010. Posteriormente, se realizaron búsquedas de los estudios previos sobre patrones de diseño y calidad de servicios *Web* teniendo en cuenta palabras clave como "patrones de diseño y *QoS* en servicios *Web*, *SOA* y microservicios" entre 2019 – 2003.

3.2. Conclusión analítica sobre servicios *Web,* evolución y tendencias investigativas

La orientación a servicios es la evolución natural de los modelos de desarrollos actuales tal como se indica en [25]. En los 80' se observaba el modelo orientado a objetos, en los 90 el modelo orientado a componentes y en la actualidad se posee la orientación a servicios. La orientación a servicios conserva los beneficios de desarrollo basado en componentes (como autodescripción, encapsulación y descubrimiento en *runtime*), pero con una diferencia fundamental en el paradigma ya que envés de la invocación de métodos remotos (por ejemplo, *EJB*) el modelo actual se basa en intercambio de mensajes. En el caso de los servicios *Web* los esquemas descriptores no sólo conforman la estructura de los mensajes, sino también los contratos de comportamiento aceptables que se utilizan para definir los patrones de intercambio de mensajes y políticas que indican la semántica de éstos. Esto promueve la interoperabilidad.

La orientación a servicios es un concepto poderoso y representa un modelo de negocio que ha tenido éxito en una variedad de industrias. Las empresas están en el proceso de

[15] *Google Scholar* es un motor de búsqueda *Web* de libre acceso que indexa textos completos y metadatos de literatura académica, de diferentes disciplinas, en múltiples formatos. Ver más en: <https://scholar.google.com.ar/>

[16] *Springer Publishing* es una empresa estadounidense de publicaciones académicas y libros que se centra en los campos de enfermería, gerontología, psicología, trabajo social, asesoramiento, salud pública y rehabilitación (neuropsicología) e informática. Para más información: <https://www.springer.com>

[17] *IEEE Xplore* es una base de datos de investigación para descubrir y acceder a artículos de revistas, actas de congresos, estándares técnicos y materiales relacionados en informática, ingeniería eléctrica y electrónica, y campos afines. Más datos en: <https://ieeexplore.ieee.org>

evaluación y adopción a la orientación a servicios, teniendo en cuenta el potencial que posee para transformar el modo en cual se hace el negocio y como brinda un alineamiento entre los objetivos de *IT* junto a los objetivos de negocio.

A continuación, se presenta un breve resumen de algunos trabajos ligados a los servicios *Web* y el dominio de esta tesis, los cuales son emergentes o marcan tendencia investigativa en la actualidad:

En [26] se indica que las arquitecturas originales cliente-servidor no fueron suficientes para hacer frente al número cada vez mayor de solicitudes y la necesidad de utilizar el ancho de banda de la red. La arquitectura orientada a servicios evolucionó hasta convertirse en una de las representaciones más exitosas con un valor empresarial agregado que proporciona servicios reutilizables y poco acoplados. *SOA* cumplió en parte con las expectativas de los clientes y los negocios, ya que aún dependía de los sistemas monolíticos. La resiliencia, la escalabilidad, la entrega rápida de software y el uso de menos recursos son características altamente deseables. La arquitectura de microservicios cumplió con las expectativas de desarrollo del sistema, pero presenta muchos desafíos. En dicho documento se ilustra como los sistemas distribuidos evolucionaron del modelo cliente-servidor tradicional a la arquitectura de microservicios recientemente propuesta. Se revisan todas las arquitecturas que contienen definiciones breves, algunos trabajos relacionados y el razonamiento de por qué tuvieron que evolucionar. También se proporciona una comparación de características de todas las arquitecturas.

En [27] se presenta un enfoque novedoso para modelar tanto los procesos de negocio como el conocimiento de dominio en una especificación unificada que podría producir un código repetitivo compatible con el tipo de arquitectura *SOA* de microservicios. También se presenta el *software MSstack* como marco de trabajo para crear tuberías de procesos de negocios utilizando la arquitectura de microservicios, con capacidades integradas de registro, monitoreo, balanceo de carga y escalamiento. Finalmente, en dicha investigación, se menciona que el uso de este nuevo enfoque para modelar junto a *MSstack* permiten a los desarrolladores crear un modelo de dominio, convertirlo en código repetitivo específico del dominio e implementar la lógica de negocios en él, al mismo tiempo que abstraen las complejidades de los microservicios y aceleran el desarrollo.

En [28] se menciona que, hoy en día, la composición de los servicios es un paradigma emergente en las redes de comunicación tales como entornos de en la nube, *Internet of Things (IoT)* y redes de sensores inalámbricos. Debido a que el objetivo de la composición de servicios es proporcionar las interacciones entre los requisitos del usuario y las entidades de los sistemas de comunicación inteligentes se han realizado muchos esfuerzos para utilizar métodos de verificación de estructuras y modelos de comportamiento para evaluar los mecanismos de composición del servicio. Por lo tanto, dicha investigación se focaliza en varios encuadres formales de verificación que se realizan para confirmar el rigor de la composición de los servicios en las redes de comunicación, teniendo como fin, clasificar y examinar exhaustivamente las técnicas de investigación actuales sobre la validación formal de la composición del servicio.

En [29] se menciona que el desarrollo eficiente de aplicaciones *Software as a Service (SaaS)* complejas se ve beneficiado por la arquitectura de microservicios y contenedores. Sin embargo, para los proveedores de *SaaS* promedio, existen muchos desafíos en la administración de microservicios a gran escala como el cumplimiento de *QoS*. En dicho artículo, se presenta *SmartVM*, un marco de implementación centrado en microservicios y basado en *SLA*, diseñado para agilizar el proceso de creación y despliegue de microservicios dinámicamente escalables que puedan manejar los picos de tráfico de una manera rentable.

En [30] se comenta que el rápido aumento de la heterogeneidad, la escala y la diversidad de los recursos y servicios de la nube lleva a una creciente complejidad en su gestión y control. El desarrollo de nuevos mecanismos para seleccionar, implementar y administrar los recursos para lograr la calidad requerida de los servicios es hoy en día un campo de investigación que generalmente se trata como *Resource Orchestration (RO)*. Se aplica principalmente a la capa *IaaS (Infraestructure as a Service)*, mientras que pocos estudios informan los resultados en *SaaS*. En dicho documento se menciona una metodología de desarrollo que tiene el objetivo de proporcionar servicios y recursos complejos mediante la interacción de otros más simples. Ésta muestra como se puede usar la descripción basada en guías de los servicios compuestos para definir la acción de orquestaciones adecuadas, a lo largo de todas las capas *"aaS"* de la arquitectura en la nube.

En [31] se indica que los algoritmos de planificación y selección existentes de servicios

están diseñados principalmente para el descubrimiento de éstos. Además, según dicho artículo, solo hay algunos trabajos que incorporan los requisitos del usuario final en la composición del servicio, por consiguiente, en el documento se propone un modelo gráfico de agrupación y selección de composición de servicios teniendo en cuentas múltiples factores de granularidad.

En [32] se señala a *GraphQL* como un lenguaje de consulta novedoso propuesto por *Facebook* en 2015 para implementar las *APIs* en la web y se presenta un estudio práctico sobre la migración de servicios *ReSTful* a esta nueva tecnología. Primero, se lleva a cabo una revisión de literatura para comprender en profundidad de los beneficios y características claves. Después de eso, se evalúan dichos beneficios en la práctica, al migrar siete sistemas a *GraphQL*, en lugar de las *APIs* basadas en el estilo *REST* estándar. Como resultado, en dicho artículo, se puede observar que *GraphQL* puede reducir el tamaño de los *JSON* retornados por las *APIs REST* en el 94% (en número de campos) y en 99% (en número de bytes), ambos resultados de la mediana de los datos.

En [33] se propone un marco de trabajo basado en *Model Driven Development (MDD)* para aspectos no funcionales en *SOA*. El *framework MDD* propuesto consiste en (1) un perfil *Unified Modeling Language (UML)* para poder modelar gráficamente aspectos no funcionales, y en (2) una herramienta *MDD* la cual acepta el modelo grafico *UML* definido en (1) y lo transforma en código de aplicaciones orientadas a servicios. Los resultados de las evaluaciones empíricas sobre el marco de trabajo *MDD* propuesto muestran como éste sirve para mejorar la reusabilidad y el mantenimiento de las aplicaciones orientadas a servicios mediante el ocultamiento de las tecnologías de implementación de bajo nivel en modelos *UML*.

En [34] se aborda la problemática vinculada a la interoperabilidad, primero presentando sus múltiples dimensiones y, a continuación, mediante la descripción de un modelo conceptual llamado *Generic Service Model (GeSMO)*, el cual es utilizado como una base para el desarrollo de lenguajes, herramientas y mecanismos que brindan apoyo a la interoperabilidad. Los autores ilustran como *GeSMO* es utilizada para la prestación de un lenguaje *peer-to-Peer (P2P)* de descripción de servicios y un mecanismo de invocación *P2P* que aprovecha la interoperabilidad tanto entre servicios heterogéneos *P2P* como entre servicios *P2P* y servicios *Web*.

En [35] se estudia un enfoque de testeo *online*. Respecto al espíritu del testeo metamórfico, la etapa *online* determina un conjunto de casos de prueba de éxito que sirven para la construcción de sus correspondientes casos de seguimiento. Estos casos de prueba serán ejecutados por los servicios metamórficos[18] los cuales encapsulan a los servicios y a las relaciones metamórficas aplicables. Así, cualquier fallo revelado por el enfoque será un fallo del servicio bajo testeo.

3.3. Estudios previos relacionados a patrones de diseño, servicios y *QoS*

Existen varios estudios relacionados al domino de los servicios *Web* y patrones de arquitectura; en menor medida también hay lecturas que vinculan a los servicios con patrones de diseño o calidad en la selección de recursos en la nube, finalmente, solo algunos trabajos asocian a los patrones diseño de servicio con *QoS* (incluyendo antipatrones de diseño)*;* por lo cual, a continuación, se destacan aquellos libros e investigaciones adecuadas dentro del marco de esta tesis:

[36] es una guía dirigida a desarrolladores y arquitectos que contiene una orientación fiable y precisa sobre como diseñar e implementar servicios *Web* seguros. La mayor parte del contenido de esta lectura se presenta en forma de patrones sobre la base de un conjunto directrices llamadas requisitos, estableciendo la siguiente división respecto a los diferentes niveles de abstracción de los problemas y soluciones recurrentes ligadas a los servicios *Web*:

- Patrones de arquitectura: Son aquellos que describen como estructurar la aplicación a nivel superior.
- Patrones de diseño: Estos describen como estructurar un subsistema o componente dentro de un sistema.
- Patrones de implementación: Estos describen a los patrones de nivel inferior los cuales son específicos a una plataforma o lenguaje en particular (en esta guía usando *Microsoft® .NET Framework* y *Web Services Enhancements (WSE) 3.0).*

[18] La palabra metamórfica, bajo el contexto de referencia aplicado, indica las transformaciones o cambios que sufren los servicios las cuales implican, a su vez, cambios en la evaluación de los mismos mediante pruebas de seguimiento.

[37] es una referencia de arquitecturas asociada a los tipos de aplicaciones *Web* comunes en la cual se proveen directrices para la selección de arquitectura y tecnología adecuadas, junto a los patrones más relevantes. En el capítulo vinculado a la capa de servicios se establece una sección clave para el dominio de esta tesis, en la cual se listan los patrones de diseño relevantes vinculados a los diferentes estilos de servicios categorizados en base a la comunicación, los canales de mensajería, la estructuración de los mensajes, el *endpoint*, el enrutamiento, la transformación de los mensajes, *ReST,* la interfaz del mensaje, y el formato *SOAP* (está sección servirá de referencia inicial respecto al estudio de patrones de diseño de servicios *Web* para mejorar *QoS*). También, en este trabajo se detalla el porqué de la necesidad de separar los servicios en una capa individual ayudando a entender como es la adaptación de este *layer* en la arquitectura de aplicación, y los pasos adecuados para diseñar la capa de servicios.

[38] es un libro que provee todo lo necesario respecto al modelado y diseño de las aplicaciones de *software*, desde casos de uso hasta arquitecturas en *UML*. El autor describe patrones ligados a varias arquitecturas, como *broker*, *discovery* y *transaction* respecto a arquitecturas orientadas a servicios. Esta lectura es perfecta para estudiantes de posgrado e ingenieros de *software* con experiencia que necesitan una referencia rápida en cada etapa del análisis, diseño y desarrollo de sistemas de *software* de gran estala. Existen tres capítulos a destacar: *Designing Object-Oriented Software Architectures*, *Designing Client/Server Software Architectures* y *Designing Service-Oriented Architectures*, siendo este último el más importante de acuerdo al ámbito de esta tesis.

[39] está orientado para los desarrolladores de *software* y arquitectos que actualmente están utilizando servicios *Web* o pensando en su uso. El objetivo de este trabajo es la familiarización con algunas de las soluciones *Web* más comunes para ayudar a determinar cuándo se deben utilizar. Éste junto a [40] forman parte del corazón de uno de los temas centrales de esta tesis (patrones de diseño vinculados a servicios *Web*). Sin embargo, en esta lectura no se hace foco en como mejorar *QoS* por lo cual es en este objetivo donde se encuentra la principal diferencia; aunque está claro que el detalle del material de estudio posee una profundidad mayor respecto a ejemplos y contribuciones de autores reconocidos. Finalmente, esta lectura se divide en seis grupos conceptuales diferentes respecto a la identificación, administración, organización y el uso de los servicios: *Service APIs, Client-Service Interactions, Request and Response Management, Web Service*

Implementation Styles, Web Service Infrastructures y *Web Service Evolution*.

En [40] se brinda una explicación minuciosa respecto al análisis y diseño de servicios *Web* y microservicios en una arquitectura orientada a servicios; motivo por el cual, dicho libro es la principal fuente de referencia de esta tesis de acuerdo al catálogo de patrones de diseño. Al igual que se mencionó previamente, se puede considerar que la precisión de este libro respecto a ejemplos y contribuciones de autores reconocidos es superior en comparación a esta tesis y no es objetivo de esta última profundizar aún más en la categorización, sino que, reusando la información existente se hará foco en *QoS*, algo que no ocurre en el libro citado.

[41] se centra en dos aspectos relacionados con la generalización de la computación en la nube: Primero, la definición de un formalismo común, basado en un modelo único y compartido que podría usarse para describir completamente los patrones en la nube; y segundo, la investigación de una metodología para automatizar el reconocimiento de elementos similares de diseño y patrones en la nube definiendo así un mapeo automático entre representaciones de aplicación que siguen estos patrones.

En [42] se presenta, por un lado, un nuevo marco de evaluación de *QoS* para servicios en la nube sobre entorno distribuidos, y por el otro, se proponen tres métodos de predicción de *QoS* y dos métodos para crear servicios *Web* tolerantes a fallas. No solo se proporcionan los últimos resultados de investigación, sino que también presenta una buena visión general de la gestión de *QoS* en las ciencias *Web*.

En [43] se plantea un enfoque de encolamiento y programación de mensajes en los servicios *Web* con el objetivo de reducir los tiempos de respuesta promedio. En dicho encuadre se utilizan un modelo de cola por prioridades junto a un algoritmo de fecha límite más temprana o en inglés *Earliest Deadline First (EDF*) para priorizar y planear la ejecución de los mensajes *Web* lo cual evidencia que se pueden identificar puntos de rendimiento de un servicio mediante la adaptación dinámica a los resultados de la perspectiva sugerida. Al final, se propone un algoritmo de programación basado en *QoS* para monitorear el rendimiento de los servicios *Web* compuestos durante el tiempo de ejecución y así encontrar el tiempo de respuesta promedio de éstos con el fin de proporcionar un servicio alternativo si el rendimiento de los servicios *Web* compuestos

no es el esperado.

En [44] se indica que la arquitectura orientada a servicios es un enfoque actual para crear aplicaciones distribuidas el cual reduce considerablemente el tiempo, el esfuerzo y el costo para desarrollar los proyectos de *software*. Además, se menciona que *SOA* proporciona un paradigma dinámico para componer aplicaciones distribuidas incluso en entornos heterogéneos por lo cual las características de composición dinámica de *SOA* hacen que la evaluación de *QoS* sea un desafío importante en cuanto a la detección de problemas relacionados.

En [45] se menciona que la composición de los servicios y su nexo con la calidad viene siendo un tema investigado en los años recientes; por lo cual, con el fin de optimizar la asignación de recursos de fabricación en la nube y mejorar la eficiencia de la composición del servicio, en dicho artículo se propone un nuevo enfoque de optimización de la composición llamado algoritmo genético mejorado de entropía[19] en la nube o por sus siglas en ingles *Cloud-entropy Enhanced Genetic Algorithm (CEGA)*.

[46] es un trabajo de revisión literaria cuya motivación es identificar patrones vinculados a cargas de trabajo en la nube según sus requisitos de *QoS* para un mejor aprovisionamiento de recursos. Además, en dicho artículo se discuten los problemas en los patrones y su terminología en la nube. Se debe destacar que dicho artículo hace foco en los patrones de las cargas de recursos y no en patrones de diseño de los servicios.

En [47] se explica que en base a la tendencia creciente de servicios *Web* en el mercado los cuales poseen diferentes *QoS*, en la actualidad, el desafío es poder seleccionar los componentes adecuados de éstos que maximicen sus utilidades y puedan cumplir con restricciones de calidad de servicio con bajo costo y en el menor tiempo posible. Debido a lo anterior, en dicho trabajo se presenta un enfoque de selección de servicios dinámico basado en la descomposición de restricciones globales y adaptativas de *QoS (AQCD*). Finalmente, el artículo en cuestión el cual no tiene en cuenta la influencia de patrones de diseño, menciona que el enfoque sugerido posee óptimos resultados respecto a la selección de servicios resaltando su escalabilidad y adaptación dejando para estudios

[19] En informática, y bajo este contexto, entropía se define como una medida de la incertidumbre existente ante un conjunto de mensajes, del cual va a recibirse uno solo

futuros la realización de investigaciones en entornos distribuidos.

Según [48] la clave radica en aprender inteligentemente las características de los servicios *Web* confiables con diferentes niveles de *QoS*, y luego identificar los no confiables de acuerdo con las características de las métricas de *QoS*. Como uno de los enfoques de identificación inteligente, la red neuronal profunda ha emergido como una técnica poderosa en los últimos años. En dicho documento, se propone un nuevo modelo de red neuronal de dos fases para identificar los servicios *Web* no confiables. En la primera fase, se recopila el conjunto de datos de *QoS* de los servicios *Web* publicados y luego se diseña un modelo de red neuronal de avance para construir el clasificador para servicios *Web* con diferentes niveles de *QoS*; y en la segunda fase, se emplea un modelo de red neuronal probabilística o como sus siglas en inglés *Probabilistic Neuronal Network (PNN)* para identificar los servicios *Web* no confiables de cada clasificación. Por último, los resultados obtenidos son prometedores respecto al enfoque planteado por dicha investigación y se sugiere hacer análisis a futuro teniendo en cuenta otros encuadres de redes neuronales como *Fuzzy Neural Network (FNN) y and Convolutional Neural Network (CNN)*.

En [49] se indica que el diseño de servicios *Web* en una arquitectura *SOA* debe tener en cuenta parámetros de calidad los cuales son afectados por la presencia de antipatrones. El alcance de dicha investigación se limita al reconocimiento automático en base a *thresholds* de descubrimiento, definidos por los autores, pertenecientes a los antipatrones "clase Dios y cortapluma suizo*"* sobre la base de métricas estáticas dejando para investigaciones futuras la elaboración de estudios similares para el resto de los antipatrones como así también el análisis sobre métricas dinámicas como lo es el tiempo de respuesta.

4. Consideraciones de diseño y principios

4.1. Consideraciones de diseño

En la capa de servicios se definen e implementan las interfaces de los servicios y los contratos asociados a los datos soportados (tipos de datos). Uno de los conceptos más importantes a tener en cuenta es que los servicios no deben exponer detalles de sus procesos internos o entidades de negocios usadas tal como lo señala [50]. Se debe asegurar, en una arquitectura orientada a servicios propiamente definida, que los componentes o entidades de la capa de negocios no posean una excesiva influencia sobre las *APIs* de los servicios.

Una aplicación, la cual brinda soluciones orientadas a servicios, puede exponer su capa de servicios para interactuar con los clientes u otros sistemas que la usan. Las *APIs* que se proporcionan a los clientes y aplicaciones conforman una manera de acceder a la lógica de negocios a través de la transmisión de mensajes. Los siguientes tipos de componentes se encuentran comúnmente en la capa de servicios:

- Interfaces de servicios: Los servicios exponen una interfaz para todos los mensajes entrantes a los cuales se les vincula una respuesta de salida. La definición del conjunto de mensajes que deben ser intercambiados con un servicio para que el servicio realice una tarea específica de negocio constituye un contrato.
- Tipos de mensajes: Mediante el intercambio de datos a través de la capa de servicios, las estructuras de datos se encuentran empaquetadas dentro de estructuras de mensajes que soportan diferentes tipos de operaciones (por ejemplo, se podría negociar el intercambio de un mensaje en formato texto plano, *XML*, *JSON* u otro tipo). En la capa de servicios se aíslan a los tipos de datos internos de aquellos contenidos en el tipo de mensaje evitando su exposición a los consumidores externos, lo que causaría problemas en términos de control de versiones de la interfaz y seguridad.

Una de las consideraciones relevantes a tener en cuenta respecto al diseño de servicios es que éstos usan interacciones basadas en mensajes, típicamente a través de una red de trabajo, la cuales son inherentemente más lentas que aquellas interacciones de procesamiento directas. Además, hay que tener en cuenta que los mensajes intercambiados entre los servicios y sus consumidores pueden ser re-enrutados, modificados y entregados en un orden diferente del cual fueron enviados, o hasta perdidos

si no existe ningún mecanismo que garantice la entrega. Debido a lo mencionado, se requiere un diseño que tenga en cuenta un comportamiento no determinístico; por los cual a continuación, se establece un breve resumen de los puntos a valorar:

- Diseñar los servicios de tipo *application-scoped* y no *component-scoped*[20]. Las operaciones del servicio deben ser menos granulares (genéricas) y centradas en las operaciones de aplicación ya que aquellas definiciones que son de más granulares (específicas) pueden dar lugar a problemas de rendimiento o escalabilidad. Sin embargo, se debe asegurar que el servicio no devuelva volúmenes ilimitados de datos muy grandes (por ejemplo, mediante paginado de los datos).
- Diseñar los servicios y sus contratos promulgando la extensibilidad y sin el supuesto de saber quién es el cliente. En otras palabras, los contratos de los datos deben diseñarse de modo que, si es posible, se pueden ampliar sin afectar a los consumidores del servicio (por ejemplo, mediante versionado de *APIs*). No se debe hacer suposiciones sobre el cliente, o sobre como se va a utilizar el servicio expuesto.
- Diseñar solo para el contrato del servicio. La capa de servicios debe implementar y proporcionar sólo la funcionalidad que se detalla en el contrato de servicio, y la implementación interna y los detalles de un servicio no deben ser expuestos a los consumidores externos.
- Separar los problemas de la capa de servicios (negocio) de aquellos de la capa de infraestructura. Si esto no se hace, se da lugar a implementaciones que son difíciles de extender y mantener. En general, se debe implementar código para gestionar las cuestiones de infraestructura en componentes aislados, y acceder a estos componentes desde aquellos ubicados en la capa de negocio.
- Diseñar asumiendo la posibilidad de *requests* inválidos. Nunca se debe asumir que los mensajes que le llegan a los servicios son válidos siempre. Se debe implementar la lógica de validación que verifique los valores, rangos y tipos de los datos ingresados en el *request* rechazando o corrigiendo[21] programáticamente los datos inválidos.
- Asegurarse de que los servicios puedan detectar y gestionar los mensajes repetidos descartando o evitando la redundancia (idempotencia).
- Asegurarse de que los servicios puedan gestionar los mensajes que llegan fuera de

[20] *Application-scoped* y no *component-scoped* se refiere en castellano a servicios más genéricos dentro del ámbito de la aplicación envés de aquellos más específicos a nivel de un componente.

[21] Corregir datos se relaciona a la depuración o adaptación de éstos siempre que sea posible. Por ejemplo, convertir un dato en formato no decimal a decimal agregando los dígitos necesarios, o al revés, redondeando o haciendo *trunk* desde un decimal a un número entero.

orden (conmutatividad), en el caso que se aplique esta característica. Si es posible que los mensajes lleguen fuera de orden, implementar un diseño que almacene los mensajes y luego los procese en el orden correcto.

4.1.1. Temas específicos de diseño

A continuación, se resume una serie de temas específicos a tener en cuenta respecto al diseño de los servicios:

Autenticación: La autenticación se utiliza para determinar la identidad del consumidor del servicio. Diseñar una estrategia de autenticación eficaz es importante para la seguridad y la fiabilidad de la aplicación ya que el fracaso de su empleo puede dejar en exposición vulnerabilidades a diferentes tipos de ataques informáticos como el robo de la sesión. En consecuencia, se sugiere:

- Identificar un mecanismo adecuado para la autenticación de los usuarios de forma segura (comúnmente mediante el uso de certificados).
- Tener en cuenta las implicaciones del uso de diferentes entornos de confianza para la ejecución de código de servicio.
- Asegurarse de que los protocolos de seguridad como *Secure Sockets Layer (SSL)* se utilicen cuando se usa una autenticación básica con credenciales que se pasen en texto plano sin cifrar. Considerar el uso de mecanismos de niveles de seguridad vinculados a mensaje apoyados por los estándares *WS-** (*Web Services Security, Web Services Trust* y *Web Services SecureConversation* con firma digital) en el caso de *SOAP* o *JSON Web Token (JWT)* para el enfoque *ReST*.

Autorización: La autorización se utiliza para determinar cuáles acciones o recursos pueden ser accedidos por un consumidor de servicios autenticado. Diseñar una estrategia de autorización adecuada es importante para la seguridad y la fiabilidad de la aplicación ya que si esto no ocurre la aplicación es vulnerable a la divulgación de información y manipulación indebida de datos. Se sugieren las siguientes directrices al diseñar una estrategia de autorización:

- Establecer permisos de acceso adecuados ligados a recursos para los usuarios, grupos y roles.
- Evitar la autorización genérica, si es posible ejecutando los servicios con la cuenta más restrictiva posible.
- Cuando sea posible, restringir el acceso a los métodos *Web* en base a roles.

Comunicación: En el diseño de la estrategia de comunicación del servicio, el protocolo a seleccionar debe estar basado en el escenario de despliegue de este servicio siendo compatible con éste. Se sugieren las siguientes directivas:

- Analizar las necesidades de comunicación y determinar si se necesita responder en cuanto a las solicitudes (comunicación de doble vía[22]) o si es posible realizar llamadas asincrónicas.
- Cuando la comunicación es asíncrona y, por consiguiente, posee menor confiabilidad se debe establecer un sistema de encolamiento de mensajes confiable.
- Si el servicio se despliega dentro de una red cerrada, el uso de *Transmission Control Protocol (TCP)* maximiza la eficiencia en la comunicación; por otro lado, si se lo hace en una red pública, considerar el uso de *HTTP*.
- Utilizar mecanismos de *Service Discover*y para poder deducir las *URLs* de los servicios de manera dinámica evitando así el *hardcodeo*[23] de las mismas.
- Validar las direcciones de los *endpoints* en los mensajes y asegurarse de proteger siempre a la información sensible.

Manejo de excepciones: Informar y gestionar las excepciones es una tarea costosa, aunque necesaria, por lo tanto, es importante tener en cuenta en el diseño que el rendimiento y la seguridad del servicio se ve afectados. En definitiva, se recomienda lo siguiente:

- Cachear solo las excepciones que se consideren posibles de ser manejadas, y considerar como gestionar la integridad del mensaje cuando una excepción ocurra
- Usar elementos *SOAP Fault* o extensiones customizadas con el objetivo de retornar los detalles de una excepción evitando la publicación de información interna.
- Asegurarse del logueo de las excepciones, y de que no se haga referencia a información sensitiva en mensajes de una excepción.

Canales de mensajería: La comunicación entre los colaboradores de los servicios se realiza mediante el envío de información a través de canales. En consiguiente se recomiendan las siguientes directivas de diseño:

- Determinar como se intercepta e inspecciona la información entre los *endpoints* si es necesario.

[22] En una comunicación relacionada a servicios telefónicos, de una empresa como *IVR,* primero se encola la llamada en una especie de comunicación inicial (1ra vía); tiempo después, en el momento de tomar la llamada, se solicita confirmación de la misma (2da vía para confirmar si sigue siendo necesario llamar) y finalmente se lleva a cabo la llamada en el caso de una confirmación positiva únicamente.

[23] Mediante el uso de *hardcodeo* el *endpoint URL* se carga estáticamente generando una alta dependencia del producto desplegado con el cliente (Por ejemplo, si el proveedor cambia el *endpoint* seria necesario en este caso desplegar nuevamente el cliente actualizando a mano nuevamente la dirección).

- Asegurarse de la gestión de excepciones condicionales en base de los canales.
- Considerar como proveer acceso a clientes que no soporten mensajería.

Construcción del mensaje: Cuando la información se intercambia entre los servicios y los consumidores, esta debe estar empaquetada[24] dentro del mensaje. El formato del mensaje está basado los tipos de operaciones a soportar, por lo cual, se recomienda:

- Dividir grandes cantidades de información en partes pequeñas y enviarlas en secuencia (loteo de mensajes).
- Cuando se usan canales lentos de entrega de mensajes, tener en cuenta la inclusión de información asociada en cuanto a cuando deben expirar aquellos mensajes sensibles al tiempo ocurrido.

Dirección o *endpoint* del mensaje: Cuando se diseña la implementación del servicio se debe tener en cuenta la posibilidad de que se envíen mensajes inválidos o duplicados hacia su dirección. Seguidamente se recomienda:

- Determinar si se aceptan todos los mensajes o si se aplica un filtro para manejar mensajes específicos.
- Diseñar la interfaz del servicio para cumplir con idempotencia, es decir, se pueden recibir mensajes duplicados desde un cliente pero procesar solo uno.
- Diseñar la interfaz del servicio para admitir conmutatividad lo cual puede incurrir en el establecimiento de fechas de solicitudes para poder procesar los mensajes en orden.
- Diseñar para escenarios desconectados. Por ejemplo, puede ser necesario diseñar para garantizar la entrega a través del guardado de mensajes que luego serán procesados.

Protección de los mensajes: Es necesario el establecimiento de alguna estrategia de protección de mensajes siempre y cuando se incluya información sensible durante la transmisión de datos entre el servicio y el consumidor. Se puede usar una protección a nivel de la capa de transporte (como *IPSec* o *SSL*) o una protección basada en el mensaje (como encriptación y firmas digitales). Se sugieren las siguientes directivas de diseño:

- Para la mayoría de los casos se recomiendan las técnicas de seguridad basadas en la protección del mensaje con el objetivo de proteger el contenido del mismo. Este tipo de mecanismos aseguran la protección de información sensible incluida en los mensajes mediante la encriptación de éstos, y la firma digital ayuda en la protección evitando el repudio de los mensajes.
- Si las interacciones entre los servicios y los consumidores no son enrutadas por

[24] Empaquetar en inglés de relaciona al concepto de *wrapper*.

intermediarios, como otros servidores o *routers*, se puede utilizar una seguridad a nivel de la capa de transporte como *IPSec* o *SSL*. En contraposición, siempre que el mensaje pase a través de intermediarios se debe utilizar una seguridad basada en la protección del mensaje.

- Para máxima seguridad se recomienda tener en cuenta a los dos mecanismos de seguridad mencionados. La seguridad a nivel de la capa de transporte ayuda a proteger información incluida en el encabezado la cual no puede ser protegida mediante una seguridad basada en la protección del mensaje.

Enrutamiento del mensaje: Un enrutador de mensaje es utilizado para desacoplar al consumidor del mensaje de la implementación del servicio. En consecuencia, se recomienda considerar:

- Determinar los patrones utilizados para el enrutamiento de mensajes (no es objetivo de esta tesis indagar al respecto de este tema, sólo conocer la necesidad). Existen dos tipos de enrutadores básicos los cuales se pueden usar: Los simples usan un solo enrutador para determinar el destinatario de un mensaje y los compuestos combinan un grupo de enrutadores simples en la gestión de flujos de mensajes más complejos.
- Si se envía una serie de mensajes en secuencia desde el mismo consumidor, el enrutador debe asegurar que todos los mensajes sean entregados al mismo *endpoint* siguiendo el orden requerido (conmutatividad).
- Un enrutador de mensajes puede inspeccionar el mensaje para poder determinar como enrutarlo. Como resultado, se debe asegurar que el enrutador sea capaz de visualizar la información de enrutamiento. Para esto, tal vez sea necesario agregar información de ruteo en el encabezado. En el caso de la encriptación del mensaje se debe asegurar que, una vez desencriptado, posea la información de enrutamiento en el encabezado.

Transformación del mensaje: Cuando se intercambia la información entre los colaboradores, existen múltiples casos en los cuales los mensajes deben ser transformados en un formato interpretable por el consumidor. Para este tema se recomienda lo siguiente:

- Determinar los requerimientos y los lugares donde se realizarán las transformaciones. Notar que el rendimiento se ve afectado mediante la ejecución de transformaciones por lo cual se recomienda minimizar su uso.
- Usar metadatos para definir el formato del mensaje.
- Considerar el uso de un repositorio externo para guardar metadatos.

Interfaz del servicio: Cuando se diseña el contrato del servicio se deben tener en cuenta los límites que serán atravesados y los tipos de consumidores del servicio. Se sugiere lo

siguiente:

- Considerar el uso de interfaces genéricas (gruesas en granularidad) que tiendan agrupar *requests* con valor de negocio similares y así minimizar el número de llamadas a través de la red de trabajo.
- Evitar el diseño de la interfaz de tal manera que un cambio en la lógica del negocio afecte su definición.
- No implementar reglas de negocio en la interfaz del servicio.
- No usar herencia de objetos para implementar el versionado de la interfaz del servicio.

Validación: Para proteger la capa de servicios, es necesario validar todas las *requests* recibidas, en consiguiente, se recomienda:

- Restringir, rechazar y corregir el contenido del mensaje, incluyendo los parámetros. Validar por tamaño, rango, formato y tipo.
- Considerar el uso de esquemas para validar los mensajes (Validadores *XML schema* o *JSON schema*).

4.2. Principios

Como se menciona en la sección 2.3, *ReST* y *SOAP* representan dos diferentes estilos para implementar un servicio. Técnicamente, *ReST* es un patrón arquitectónico construido con verbos simples que sobre *HTTP*. Si bien los principios de arquitectura *ReST* podrían ser aplicados a protocolos distintos, en la práctica las implementaciones generalmente se utilizan en conjunción con *HTTP*. *SOAP* es un protocolo de mensajería basado en *XML* que se puede utilizar con cualquier protocolo de comunicación, incluyendo *HTTP*.

ReST y *SOAP* se pueden utilizar en la mayoría de las implementaciones de los servicios; aunque el enfoque *ReST* es a menudo el más adecuado para los servicios de acceso público o aquellos casos en que un servicio puede ser accedido por los consumidores desconocidos. *SOAP* es mucho más adecuado para la aplicación de una serie de interacciones procedurales, tales como una interfaz entre las capas de una aplicación. *SOAP*, no se restringe al protocolo *HTTP*. Los estándares *WS-**, pueden ser utilizados en *SOAP*, proporcionando así un método estándar e interoperable para el tratamiento de cuestiones de mensajería comunes, tales como la seguridad, transacciones, direccionamiento y la confiabilidad. *ReST* también puede proporcionar el mismo tipo de funcionalidad, pero a menudo se debe crear un mecanismo personalizado, ya que sólo unas pocas normas existen actualmente para este estilo.

Por lo general, se puede utilizar los mismos principios en el diseño de las interacciones basadas en mensajes *SOAP* que en las interacciones sin estado *ReST*. Ambos enfoques de intercambio de datos (*payload*) usan verbos. En el caso del *SOAP*, el conjunto de verbos está definido por la interfaz del servicio. En el caso de *ReST*, el conjunto de verbos se restringe a los verbos preestablecidos que se reflejan en el protocolo *HTTP*. Se deben considerar los siguientes principios a la hora de elegir entre *ReST* y *SOAP*:

- *SOAP* es un protocolo que proporciona un marco de trabajo de mensajería básica sobre el cual se pueden construir capas abstractas, y es comúnmente usado como un marco de trabajo *RPC* que pasa las llamadas y respuestas a través de redes que utilizan mensajes *XML* con formato *SOAP*.
- *SOAP* se ocupa de temas como la seguridad a través de su implementación del protocolo interno, sin embargo, este estilo requiere que *SOAP stack* esté disponible.
- *ReST* es una técnica que puede utilizar protocolos livianos de mensajería, tales como *JavaScript Object Notation (JSON)*, el protocolo de publicación *Atom*, y formatos *Plain Old XML (POX)* personalizados.
- *ReST* expone una aplicación y los datos como una máquina de transición de estados, no sólo un *endpoint* de servicio. Permite efectuar llamadas *HTTP* estándares como *GET* y *PUT* para consultar y modificar el estado del sistema. *ReST* es *stateless* por naturaleza, lo que significa que cada solicitud enviada desde el cliente al servidor debe contener toda la información necesaria para comprender la petición ya que el servidor no almacena los datos de estado de sesión.

En [51] se establecen los siguientes principios claves respecto al estilo *ReST*:

- Cada recurso es identificado con un *ID* único, por ejemplo, *URI*.
- Vinculación de los recursos con el objetivo de establecer relaciones entre éstos.
- Utilización de métodos estándares (*HTTP, media types, XML*).
- Los recursos pueden tener múltiples representaciones las cuales reflejan diferentes estados de la aplicación.
- La comunicación no mantiene el estado y es sobre el protocolo *HTTP*.

Finalmente, en [52] se presentan los siguientes principios, los cuales conforman el corazón de una arquitectura orientada a servicios:

- Acoplamiento débil de acuerdo a que el convenio del servicio posee un bajo nivel de dependencia a los requisitos de los consumidores y por lo tanto se encuentra desacoplado del entorno circundante.

- Reusabilidad referida a cuando el servicio es considerado un artefacto de empresa que puede ser reutilizable en diferentes unidades de negocio ya que su lógica de implementación es agnósticas al contexto de negocio en que funciona.
- Interoperabilidad[25] en cuanto a la capacidad del servicio el cual, a través de su interfaz, puede funcionar en combinación con otros (actuales o futuros) sin poseer restricciones de implementación y acceso.
- Descubrimiento dinámico de los servicios sobre la base de que los éstos se deben complementar con información adicional (metadatos) mediante los cuales se pueden descubrir e interpretar de manera adecuada. Para que los servicios se posicionen como activos de *IT* con un retorno de la inversión *(ROI)* repetible, deben identificarse y entenderse fácilmente cuando se presentan las oportunidades de reutilización.
- Autonomía de acuerdo a que el servicio debe ejercer un alto nivel de control sobre sus recursos y entorno de ejecución subyacente, lo cual incrementa la confiabilidad y la previsibilidad del comportamiento de éste.
- Abstracción de servicio en el sentido de que el convenio de este solo posee información esencial, encontrándose limitado solamente a lo que se publica, lo cual hace que se fomente la abstracción ocultando los detalles de la implementación subyacente.
- Servicio sin estado en cuanto a que el éste minimiza el consumo de recursos y optimiza su rendimiento al diferir el manejo de la información del estado cuando sea posible.
- Contrato de servicio estándar respecto a que los servicios *Web* dentro de un mismo inventario, los cuales poseen la misma tecnología de arquitectura, cumplen con las mismas normas de diseño del contrato.
- Composición referida a la capacidad del servicio que le permite formar parte de una o más composiciones junto a otros, independientemente de la complejidad que esto atañe, lo cual permite resolver problemas mayores de los que resuelve aisladamente.

[25] El principio de interoperabilidad en servicios *Web* se considera el más evidente de todos ya que es apoyado por todos los principios restantes.

5. Patrones y su vínculo con los servicios *Web*

5.1. Introducción al concepto de patrones

En [53] se desarrolla el concepto de patrones por primera vez, relacionado a la arquitectura de estructuras edilicias. En el campo del *software,* el diseño de patrones se introduce en [54], en el cual se describen 23 patrones de diseño. Posteriormente, es en [55] donde se plantean patrones asociados a diferentes niveles de abstracción, desde patrones arquitectónicos de alto nivel, patrones de diseño y patrones vinculados a idiomas de bajo nivel.

Un patrón de diseño describe un problema de diseño recurrente a resolver, una solución al problema, y el contexto en el que dicha solución funciona, tal como se indica en [54], [55]. La descripción específica a los objetos y las clases que se adaptan para resolver un problema de diseño general en un contexto particular. Un patrón de diseño es una forma más genérica de reutilización que una clase ya que vincula a más de una clase y la interconexión entre los objetos de distintas clases. Un patrón de diseño se refiere a veces como microarquitectura, tal como se señala en [56]. A continuación, se listan los principales tipos de patrones de *software*:

- Los patrones de diseño son un grupo pequeño de objetos en colaboración.
- Los patrones de arquitectura son más genéricos que los de diseño y se encuentran ligados a la estructura de los subsistemas principales de un sistema.
- Los patrones de análisis modelos recurrentes que forman parte del análisis orientado a objetos los cuales se describen con guías estáticas y diagramas de clase.
- Los patrones de líneas de productos específicas son aquellos usados sobre áreas de aplicación específicas, tales como la automatización de fábrica o el comercio electrónico.
- Los idiomas son patrones de bajo nivel específicos de un lenguaje de programación determinado y en los cuales se describe la implementación de soluciones a un problema que utiliza las funciones del lenguaje (por ejemplo, *Java* o *C++*). Estos patrones están directamente ligados al código por lo cual solo pueden ser utilizados por las aplicaciones que están codificados en el mismo lenguaje de programación.

Como se comenta en la sección 1.2., el foco de esta tesis se centra en patrones de diseño de *software* por lo cual es necesario entender como éstos se describen siguiendo un formato, tal como se expresa en [54], que divide cada patrón en secciones de acuerdo a una plantilla la cual representa una estructura uniforme para entender, estudiar y comparar el uso de éstos (ver detalle de las plantillas de los patrones usadas en esta tesis en el Anexo).

En [54] se establece un esquema básico de clasificación basado en las categorías de creación, estructura y comportamiento. Posteriormente, en [57] se indica una clasificación más granular agregando dos categorías más, los patrones básicos, de colecciones, y concurrencia. Por último, en [40], [58] se definen categorizaciones específicas para el marco de esta tesis las cuales serán tenidas en cuenta en conjunción a las definiciones previas.

5.1.1. Necesidad de uso relacionada al diseño de servicios *Web*

Una arquitectura orientada a servicios es una arquitectura de *software* distribuida que consta de múltiples servicios autónomos. Los servicios se encuentran distribuidos de tal manera que pueden ser ejecutados en diferentes nodos por diferentes proveedores de servicios. Con *SOA*, el objetivo es desarrollar aplicaciones de *software* compuestas por servicios distribuidos, de tal manera que los servicios individuales pueden ser ejecutados en diferentes plataformas e implementados por diferentes lenguajes. Los protocolos estándar se proporcionan para permitir que los servicios puedan comunicarse entre sí e intercambiar información. Con el fin de permitir que las aplicaciones puedan descubrir y comunicarse con los servicios, cada servicio tiene una descripción del servicio; la descripción del servicio define el nombre del servicio, la ubicación del servicio, y sus requisitos de intercambio de datos.

Un proveedor debe exponer servicios y dar soporte para que sean consumidos por múltiples clientes. Por lo general, un cliente contrata un servicio proporcionado por un proveedor de servicios, tal como Internet o correo electrónico. *SOA* se basa en el concepto de servicios débilmente acoplados que pueden ser descubiertos y vinculado a los clientes (también conocidos como consumidores de servicios o solicitantes de servicios) con la

asistencia de *brokers*[26].

Un objetivo importante de *SOA* es diseñar servicios como componentes autónomos reusables. La intención de los servicios es que estos sean autocontenidos y débilmente acoplados, lo que significa que las dependencias entre servicios se mantienen al mínimo. En lugar de que un servicio dependa de otro, la coordinación de los servicios se presenta en situaciones en las que múltiples servicios tienen que ser accesibles y el acceso a los mismos debe ser secuenciado. Múltiples patrones de arquitectura y diseño se describen para una arquitectura orientada a servicios, por ejemplo, los que se detallan en la sección 7. Es necesario el uso de patrones para cumplir con los principios fundamentales de una arquitectura orientada a servicios como así también mejorar *QoS.*

5.2. Breve compendio de *anti-patterns* y sus características vinculadas con servicios *Web*

"El estudio de antipatrones es una actividad de investigación importante. La presencia de 'buenos' patrones no es suficiente en un sistema exitoso, también se debe demostrar la ausencia de patrones en los sistemas sin éxito. Del mismo modo, es útil mostrar la presencia de ciertos patrones (antipatrones) en los sistemas sin éxito, y su ausencia en los sistemas exitosos."
—Jim Coplien[27]

Al tratar de explicar la falta de éxito asociada al desarrollo de *software*, es adecuado llegar a la conclusión de que hay muchos más antipatrones en la práctica que los patrones de diseño, o por lo menos es más fácil llevarlos a cabo. En [59] se define que un antipatrón es una forma literaria que describe una solución que ocurre comúnmente respecto a un problema que genera consecuencias decididamente negativas y, además se menciona que, en los tiempos de Internet, la tecnología cambia tan rápido que los patrones de diseño de ayer pueden ser considerados los antipatrones de hoy.

Se considera que un patrón es una disposición de partes repetidas; un diseño o forma de dirigir una implementación; un modelo o especificación. Pero un patrón también puede

[26] *Brokers* de servicios se refiere al corredor de servicios el cual decide que servicios serán entregados al consumidor.

[27] Jim Coplien es un reconocido investigador en el campo de la ciencia de la computación, escritor y coautor de múltiples libros, en los cuales se destacan *Software Patterns* (Junio 1996*) y Lean Software Architecture for Agile Software Development* (Agosto 2010).

ligarse a un comportamiento negativo. Los antipatrones describen que es lo que se debe evitar; y la identificación de lo que debe evitar es un factor crítico en la comprensión de los modelos o patrones apropiados para la resolución de situaciones recurrentes asociados a un problema lo cual conlleva al éxito del desarrollo de software. En esta sección se detallarán brevemente los antipatrones más comunes siguiendo una platilla personalizada con características mínimas para cumplir con el propósito de resumen en [60] y haciendo foco sobre aquellos de diseño vinculados al ámbito de los servicios *Web*.

Un antipatrón bien documentado describe una situación genérica, las principales causas que le dieron origen, los síntomas que describen la forma de reconocerla, las consecuencias, y una solución de *refactoring*[28] que establece como cambiar el antipatrón en una situación saludable.

Los antipatrones son un método eficiente para la vinculación de una clase específica de soluciones con una situación general. Éstos proporcionan experiencia del mundo real en el reconocimiento de los problemas recurrentes en la industria del *software* y proporcionar una solución detallada para las situaciones difíciles más comunes. Los antipatrones ofrecen un vocabulario común para identificar los problemas y discutir soluciones.

Existen tres perspectivas principales en cuanto a antipatrones: desarrollo, arquitectura y gestión. Los antipatrones de desarrollo comprenden a los problemas y soluciones técnicas que se encuentran por los desarrolladores. Los antipatrones de arquitectura identifican y resuelven problemas comunes en como los sistemas se encuentran estructurados. Los antipatrones de gestión se refieren a los problemas comunes asociados a los procesos de desarrollo y el desarrollo de organizaciones. Estos tres puntos de vista son fundamentales en el desarrollo de *software*, y existen muchos problemas en cada uno. Seguidamente se establece una breve muestra de los antipatrones principales resumidos en cada una de las áreas mencionadas omitiendo indagar la lista completa de éstos ya que no es propósito de esta tesis (ver compendio completo en el apéndice):

Antipatrones de desarrollo:

5.2.01. Nombre del antipatrón: Gota.

Sinopsis: Una clase que monopoliza las responsabilidades de la aplicación; mientras que

[28] *Refactoring* es una solución de re-fabricación (o reimplementación) de componentes de *software* para la corrección de un problema de diseño o implementación; o para poder adaptarse al cambio de una necesidad de negocio de acuerdo a un sistema poco robusto.

los otros objetos solo encapsulan datos o ejecutan simples procesos. La solución incluye *refactoring* para la distribución de responsabilidades logrando así el aislamiento de los efectos de cambio.

Nombre alternativo: Clase Dios.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: *Refactoring* de responsabilidades.

Tipo de solución refactorizada: *Software*.

Causas raíz: Pereza, prisa y la influencia de un diseño procedural.

Desequilibrio de fuerzas: Manejo de funcionalidades, rendimiento y complejidad.

Evidencia anecdótica: "Esta es la clase que representa el corazón de la arquitectura".

Vínculo con los servicios *Web*: Interfaz gruesa en granularidad y un diseño poco robusto, permitiendo la ejecución de múltiples acciones con responsabilidades distintas sobre la misma *API*, lo cual conlleva a problemas de rendimiento y mantenimiento (en el enfoque *ReST* puede originarse este antipatrón sino se aplica el nivel 3 del *RMM* el cual fomenta la distribución de responsabilidades entre recursos de diferentes *APIs*). Si un mismo servicio realiza múltiples funciones con responsabilidades variadas y no es de orquestación, el mantenimiento poco aislado de los problemas monopolizados necesita *refactoring* distributivo de responsabilidades (ver figura 4).

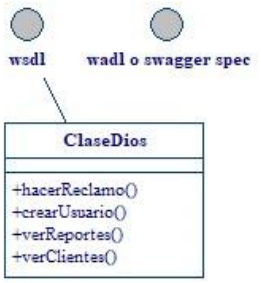


Figura 4: Gota **-** Todas las implementaciones de las operaciones definidas en la interfaz (*WSDL o Swagger Spec* dependiendo el estilo de servicio) son desarrolladas dentro de la clase Dios la cual implementa todas las funcionalidades de negocio. La poca adaptabilidad al cambio se enfatiza cuando sobre una misma interfaz se declaran múltiples *endpoints* con operaciones ligadas a diferentes unidades de negocio. Fuente propia elaborada con *StarUML*.

5.2.02. Nombre del antipatrón resumido: Caminando por un campo minado.

Síntesis: Los productos de *software* entregados poseen un gran número de defectos, de

hecho, los expertos estiman que el código fuente original contiene de dos a cinco errores por línea de código. Esto significa que el código necesitará de dos o más cambios por línea de código con el objetivo de remover todos los defectos posibles. Es indudable que numerosos productos son entregados mucho antes de estar listos para brindar soporte a los sistemas operativos de negocio.

Solución refactorizada: Necesidad de una inversión adecuada en las pruebas de *software* para hacer que los sistemas sean relativamente libres de errores. En algunas empresas progresistas, el tamaño del personal de pruebas suele superar al personal de programación. Un sistema típico puede requerir la ejecución de al menos cinco *test cases* más comparado con el código que se entrega en producción. Un *software* de prueba es a menudo más complejo de lo que es el *software* de producción debido a que se involucra de manera explícita la gestión del tiempo de ejecución con el propósito de detectar la mayor cantidad de errores posibles. El control de la configuración de los *test cases* es determinante ya que permite la gestión de los activos de *software* vinculados a las pruebas, por ejemplo, para brindar soporte a las pruebas de regresión. Otros enfoques efectivos relacionados al *testing* incluyen la ejecución de las pruebas de manera automatizada y el diseño de la prueba. La ejecución del *testing* manual es un trabajo intensivo, y no hay una base probada para la eficacia de las pruebas manuales. Por el contrario, la ejecución de pruebas automáticas permite la ejecución de las mismas en común acuerdo con el ciclo de generación. El *testing* de regresión se puede ejecutar sin intervención manual, lo que garantiza que las modificaciones en el *software* no causen defectos en los comportamientos previamente aprobados. El diseño del *testing* automatizado es compatible con la generación de *test suites* rigurosos, y existen múltiples herramientas disponibles para el cumplimiento de este propósito.

Vínculo con los servicios *Web*: La inversión de personal de testeo o el involucramiento de los desarrolladores en el uso de técnicas como *test driven development (TDD)* es fundamental en el dominio de los servicios al igual que en cualquier producto de *software*; por lo cual, se recomienda: a) el uso de herramientas como *soapUI* (ver figura 5) para *SOAP* o *Swagger Tester* en el caso de *ReST* para la generación de casos de prueba de sistema e integración y la gestión de *test suits* respecto al diseño de pruebas de regresión automatizadas; b) la utilización de instrumentos como *Selenium*[29] para la evaluación del

[29] *Selenium* es una herramienta desarrollada en *JAVA* que se utiliza para la automatización de acciones en el navegador motivo por el cual no se limita a las pruebas de sistema solamente, sino que puede ser usado para implementar acciones repetibles sobre un formulario de datos. Más información puede

rendimiento de los servicios *Web*; c) el empleo de mecanismos de pruebas de unidad como *Junit o TestNG*.

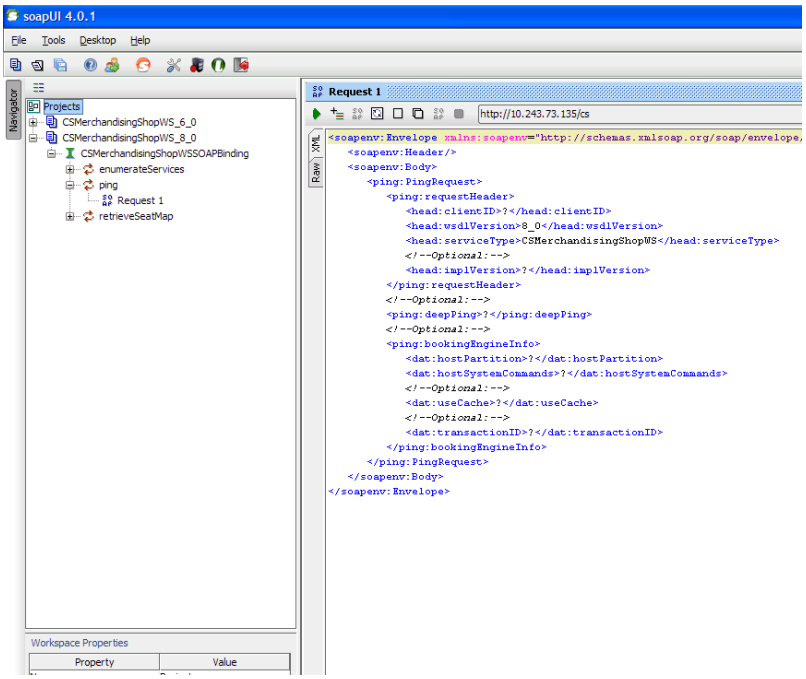


Figura 5: Herramienta *soapUI* para el testeo de servicios *Web*. Fuente propia.

Antipatrones de arquitectura:

5.2.03. Nombre del antipatrón resumido: Tubo de cocina autogenerado.

Síntesis: Ente antipatrón generalmente ocurre cuando se migra un sistema de *software* existente a una infraestructura distribuida; más precisamente, mediante la conversión de las interfaces de *software* actuales a las interfaces distribuidas. Si el mismo diseño se utiliza en la computación distribuida, emergen numerosos problemas. Por ejemplo, las interfaces de un sistema no distribuido suelen ser de granularidad fina respecto a la transferencia de datos, lo cual es ineficaz en un entorno distribuido debido a que el rendimiento general se puede ver afectado. Las interfaces existentes suelen ser específicas a la implementación causando interdependencias con los subsistemas cuando se utilizan en un sistema distribuido de mayor escala. Las operaciones locales a menudo hacen varias suposiciones acerca de la ubicación, incluyendo el espacio de direcciones y el acceso a sistema de archivos local. El exceso de complejidad puede surgir cuando múltiples interfaces existentes se exponen a través de un sistema de distribución a gran escala.

encontrarse en <https://www.seleniumhq.org>

Solución refactorizada: Cuando se migran las interfaces existentes a un sistema distribuido estas deben ser rediseñadas. Un modelo de objetos con granularidad gruesa, más general, debe ser tenido en cuenta para cada una de las interfaces intervinientes. La interoperabilidad funcional entre los sistemas distribuidos debe ser el objetivo primordial del diseño. La estabilidad de las nuevas interfaces es importante, ya que el código de cada implementación está fuertemente ligado al cumplimiento del contrato independientemente de la tecnología involucrada.

Vínculo con los servicios *Web*: Una mala migración de las interfaces de los sistemas antiguos o *legacy* puede afectar el rendimiento de manera sustancial respecto a la capacidad de transferencia de datos a través de la red en una arquitectura orientada a servicios como así también la visualización de los datos en la capa de presentación. Interfaces muy granulares respecto a clientes con restricciones respecto a los tiempos de respuestas hacen inviable la relación entre los colaboradores. Los principios de diseño establecidos en la sección 4.2. deben ser tenidos en cuenta respecto a una reingeniería distribuida vinculada a servicios *Web*.

5.2.04. Nombre del antipatrón: Sistema aislado.

Sinopsis: Este antipatrón se encuentra comúnmente vinculado a sistemas antiguos o *legacy* con calidad indeseable y su integración. Se atribuye que la causa de esta calidad inapropiada es la estructura interna del sistema. Los subsistemas se encuentran integrados a su propia manera utilizando múltiples estrategias de integración y mecanismos. El problema clave de este antipatrón es la falta de abstracciones comunes entre los subsistemas, mientras que, en el patrón Islas automatizadas, el problema es la falta de convenciones multisistémicas. El enfoque de integración entre los subsistemas es punto a punto y no se realiza de manera sencilla. Por otra parte, la implementación del sistema es frágil porque hay muchas dependencias implícitas sobre la configuración del sistema, detalles de instalación y el estado del sistema. Se considera que este tipo de sistemas es difícil de escalar ya que se agregan nuevas ligaduras punto a punto aumentando el nivel de complejidad y haciéndolo prácticamente inmantenible (ver figura 6).

Nombre alternativo: Sistema antiguo, integración ad hoc, tubo de cocina independiente o *stopevive.*

Escala más frecuente: Sistema.

Nombre de la solución refactorizada: Marco de trabajo o *Architecture Framework* es una arquitectura de componentes que proporciona una sustitución flexible de módulos de

software. Los subsistemas se encuentran modelados de forma abstracta de tal manera que existe una menor cantidad de interfaces expuestas de acuerdo con la cantidad de implementaciones de los subsistemas. La substitución puede ser estática (tiempo de compilación) o dinámica (tiempo de ejecución). La clave es descubrir las abstracciones apropiadas de las interfaces. Las abstracciones subsistémicas modelan la necesidad de interoperabilidad sin exponer diferencias innecesarias entre subsistemas y detalles de implementaciones específicas.

Tipo de solución refactorizada: *Software*.

Causas raíz: Prisa, la avaricia, ignorancia y pereza.

Desequilibrio de fuerzas: Manejo de complejidad y cambio.

Evidencia anecdótica: El proyecto de software supera el presupuesto estimado, los usuarios aún no observan las características esperadas, y el sistema es poco adaptable al cambio. Cada componente del sistema parece ser un tubo de estufa.

Vínculo con los servicios *Web*: El diseño apropiado de servicios *Web* con operaciones con granularidad adecuada (granularidad gruesa para la mayoría de las operaciones de negocio) y composiciones de servicios en *SOA* es la antítesis de este tipo de patrón.

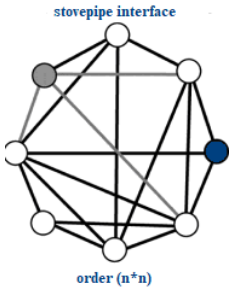


Figura 6: Ligaduras punto a punto entre las interfaces del sistema (Cada interfaz es única para cada subsistema). Fuente [61, Fig. 6.7].

5.2.05. Nombre del antipatrón resumido: Documentación frondosa.

Síntesis: Los procesos de desarrollo de *software* basados en documentación extensiva usualmente producen requerimientos y especificaciones poco útiles ya que los autores evaden la toma de decisiones importantes. A fin de evitar cometer un error, los autores toman un curso más seguro elaborando alternativas. Los documentos resultantes son voluminosos y se convierten en un enigma, no hay abstracción útil de los contenidos que transmiten la intención de los autores. Son entonces desafortunados los lectores, que tienen la obligación contractual vinculada al texto, ya que los detalles aturden la mente. Cuando no se toman decisiones y no se establecen las prioridades, los documentos tienen

un valor limitado. No es razonable contar con cientos de páginas de los requisitos que son igualmente importantes u obligatorios. Los desarrolladores se quedan sin orientación útil respecto a lo prioritario en la práctica.

Solución refactorizada: Los resúmenes o *blueprints* de arquitectura son abstracciones de los sistemas de información que facilitan la comunicación de requerimientos y planes técnicos entre los desarrolladores y los usuarios. Un *blueprint* de arquitectura es un pequeño conjunto de diagramas y tablas que comunican la operativa, técnica y arquitectura de los sistemas de información actuales y futuros [62]. Un modelo típico comprende no más de unas docenas de diagramas y tablas. Estos resúmenes de arquitectura son particularmente útiles en una empresa con múltiples sistemas de información. Cada sistema establece sus *blueprints* y la organización debe lograr la adecuación de éstos sobre los detalles específicos de cada uno, elaborando así un *blueprint* a nivel empresarial. Debido a que los resúmenes de arquitectura permiten que múltiples proyectos retraten sus tecnologías, las oportunidades de interoperabilidad y la reutilización son alcanzadas.

Vínculo con servicios *Web*: En [63] se menciona que sin un idioma común junto a *blueprints* de la industria, *SOA* puede fallar en el cumplimiento de los beneficios prometidos de reúso de los servicios, a nivel intra e interempresarial, e interoperabilidad (generando mayor complejidad a la infraestructura de *IT*). El *blueprint* de SOA se concibe como una colección de varios volúmenes de las publicaciones que actúan como una enciclopedia de referencia estándar para todas las partes interesadas respecto a *SOA*.

Antipatrones en la gestión de proyectos de *software*:

5.2.06. Nombre del antipatrón resumido: El mito del mes equivalente a cualquier hombre.

Síntesis: Los programadores talentosos son esenciales para el éxito de un proyecto de software. Estos producen *software* de mayor calidad y cantidad que el promedio de los desarrolladores. En los proyectos de gran escala con un *staff* numeroso existe el riesgo de perder desarrolladores o la necesidad de ampliar la cantidad de recursos por motivos de envergadura en un proyecto inicialmente subvalorado. La falacia de agregar más *staff* para el cumplimiento de objetivos de un proyecto de *software* se observa en [64] indicando que un proyecto que aumenta la cantidad de programadores no reduce la cantidad de tiempo que toma éste en completarse: “Los hombres y los meses son productos intercambiables sólo cuando una tarea se puede repartir entre muchos

trabajadores sin comunicación entre ellos. Este es el caso de cosechar el trigo o algodón, no es ni siquiera aproximadamente el caso de la programación de un sistema".

—Frederick Brooks[30]

Solución refactorizada: El tamaño ideal de un proyecto es 4 desarrolladores, la duración ideal de un proyecto son 4 meses [65]. Los equipos de proyecto que crecen más allá de cinco personas generalmente experimentan dificultades crecientes ligadas a la coordinación del grupo. Los miembros tienen problemas para tomar decisiones eficientes y mantener una visión común. Establecer objetivos a corto plazo es esencial para alentar al equipo a focalizarse y comenzar a producir una solución. Se considera entonces que los proyectos extremadamente grandes son caracterizados por esfuerzos inútiles, mientras que pequeños equipos de proyecto con responsabilidad individual son mucho más propensos a producir software con éxito.

Vínculo con los servicios *Web*: Este tipo de antipatrón aplica a nivel del proyecto de *software*, por lo cual, si la solución arquitectónica está vinculada al desarrollo de servicios estos proyectos deben cumplir con los mismos principios establecidos en la sección referida a la solución refactorizada de este patrón.

5.2.07. Nombre del antipatrón resumido: Ingeniería centrada en gráficos.

Síntesis: En algunos proyectos, los desarrolladores se atascan o pierden demasiado tiempo en la preparación de gráficos y documentos en lugar del desarrollo de software. La administración no obtiene las herramientas de desarrollo adecuadas, y los ingenieros no tienen más remedio que utilizar el *software* de oficina existente para producir diagramas seudotécnicos y documentos. Esta situación es frustrante para los ingenieros, que no utilizan su verdadero talento. Muchos de los desarrolladores atrapados en la preparación de gráficos y documentos deben ser redirigidos a la construcción de prototipos.

Solución refactorizada: Establecimiento de tecnología/metodología adecuada para la agilización en la elaboración de documentos, aunque también la tendencia es la reducción de evidencia minuciosa ya que los desarrollos se consideran incrementales, por ejemplo, usa solución simplificada es sacar fotos a las imágenes de los diseños discutidos en

[30] Brooks, Jr., F.P es un reconocido profesor de la universidad de Carolina del Norte el cual es autor de numerosos *papers* y publicaciones de libros entre las cuales se destacan: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition (1995), The Design of Design: Essays from a Computer Scientist (2000* y su reedición actualizada en *2010)* y *No Silver Bullet—Essence and Accidents of Software Engineering (1986)*. Para más información referirse a <http://www.cs.unc.edu/~brooks/>.

pizarrón durante cambios/incrementos sustanciales utilizándolas como evidencia en conjunción al uso de los *blueprint* de arquitectura.

6. *QoS*

6.1. Introducción

Una vez que los servicios individuales se implementan y despliegan en un entorno de producción es necesario tener en cuenta varias cuestiones cruciales para el éxito de una aplicación orientada a servicios. Una de las principales prioridades en la adopción de un proyecto en producción es la calidad del servicio o *Quality of Service (QoS)*. La gestión de los servicios es también esencial para asegurar *QoS* mediante el control, medición e intención de sostenibilidad en términos de disponibilidad, seguridad y rendimiento general.

6.2. Concepto de *QoS*

"En Internet y otras redes de trabajo, *QoS* es la idea de que las tasas de transmisión, las tasas de error, y otras características se pueden medir, mejorar, y, en cierta medida, garantizar con anticipación."
<http://whatis.techtarget.com/search/query?start=0&filter=1&q=QoS>
En [66] se menciona que existen muchas definiciones alternativas respecto a *QoS* como así también que el término evoca a connotaciones diferentes dependiendo del contexto de su uso. Sea cual sea el alcance y la dimensión de la calidad de servicio, ésta se asocia siempre con la fiabilidad y el rendimiento del servicio, implicando también comunicación y transmisiones seguras.

6.3. *QoS* para servicios *Web*

Con la proliferación de los servicios *Web* como una solución de negocio asociada a la integración de aplicaciones empresariales, la calidad de éstos es una propiedad a tener en cuenta por los proveedores y consumidores ya que traerá una ventaja competitiva. Sin embargo, debido a las características dinámicas e impredecibles de los servicios *Web*, proporcionar la deseada calidad de servicio no es una tarea fácil. Múltiples aplicaciones compuestas por servicios *Web* con requisitos de *QoS* distintos competirán por los recursos de red y sistema, tales como ancho de banda y tiempo de procesamiento. Para proporcionar una mejor calidad de servicio, primero es necesario identificar todos los

posibles requisitos de *QoS* relacionados a los servicios *Web* tal como se indica en [67].

Los requerimientos vinculados a la calidad de los servicios se refieren a características cualitativas en general. Estos pueden incluir el rendimiento, confiabilidad, escalabilidad, capacidad, robustez, manejo de excepciones, precisión, integridad, accesibilidad, disponibilidad, interoperabilidad, seguridad y requisitos de *QoS* para redes de trabajo.

Rendimiento: El desempeño de un servicio *Web* representa la velocidad en la cual una solicitud de servicio se completa. Éste se puede medir en cuanto a rendimiento, tiempos de respuesta, ejecución, transacción y latencia tal como se indica en [68], [69]. El rendimiento es el número de solicitudes de servicio *Web* procesadas en un intervalo de tiempo dado. El tiempo de respuesta es el tiempo requerido para completar una solicitud de servicio *Web*. El tiempo de ejecución es el tiempo que tarda un servicio *Web* para procesar una secuencia de actividades. El tiempo de transacción representa el tiempo que transcurre mientras que el servicio *Web* completa una transacción de negocio (este tiempo de la transacción puede depender de la definición del servicio *Web*). Por último, la latencia es el retardo de ida y vuelta o *Round Trip Delay (RTD)* entre el envío de una solicitud y la recepción de la respuesta.

Confiabilidad: En [70] se menciona que los servicios *Web* deben contar con una alta confiabilidad la cual representa la capacidad para realizar sus funciones requeridas bajo las condiciones establecidas en un determinado intervalo de tiempo. La confiabilidad es la medida global de un servicio *Web* para mantener la calidad del servicio. La medida global de un servicio *Web* está relacionada con el número de fallas por día, semana, mes o año. La confiabilidad se relaciona también con la entrega asegurada y el orden de los mensajes que se transmiten mediante la emisión de solicitudes y recepción de respuestas.

Escalabilidad: La escalabilidad representa la capacidad que posee el sistema del proveedor del servicio *Web* respecto al procesamiento vinculado al aumento de solicitudes de los clientes durante un intervalo de tiempo dado. Es deseable que un proveedor de servicios pueda responder de manera adecuada ante el incremento de solicitudes durante un período de tiempo sin colapso de servicio, aunque existen umbrales de respuesta determinados de acuerdo al entorno. Los servicios *Web* deben ser escalables en función del número de operaciones o transacciones soportadas.

Capacidad: La capacidad es el límite del número de solicitudes simultáneas que deben ser provistas por un servicio *Web* con un rendimiento garantizado tal como se señala en [68]. Es deseable que servicios *Web* soporten el procesamiento de solicitudes y respuestas

simultáneas.

Robustez: Robustez indica el grado en que un servicio *Web* puede funcionar correctamente incluso en presencia de ingresos no válidos, incompletos o contradictorios [68]. Los servicios *Web* deberían funcionar incluso si se proporcionan parámetros incompletos en a la invocación de peticiones de servicios.

Manejo de excepciones: En [68] se observa que es imposible pretender que el diseñador del servicio pueda especificar todas las alternativas y resultados posibles (sobre todo respecto a casos especiales y posibilidades no previstas), por lo cual, las excepciones deben ser manejadas adecuadamente. El manejo de excepciones se refiere a como el servicio se encarga de manipular estas excepciones.

Precisión: [68] define precisión como la tasa de error generada por el servicio *Web*. El número de errores que el servicio genera durante un intervalo de tiempo debe ser minimizado.

Integridad: La integridad de los servicios *Web* se refiere a impedir el acceso no autorizado sobre componentes del servicio o la modificación de los datos vinculados a la interrelación entre los colaboradores. Existen dos tipos de integridad: integridad de los datos y la integridad transaccional. La integridad de los datos define si los datos transferidos fueron modificados en tránsito mientras que la integridad transaccional se refiere a un procedimiento o conjunto de procedimientos asociados para garantizar y preservar la integridad de una transacción en base de datos tal como se da a detalla en [69].

Accesibilidad: En [69] se define a accesibilidad de un servicio *Web* como la capacidad del mismo para poder atender las solicitudes de los clientes. Un alto nivel de accesibilidad se puede lograr, por ejemplo, mediante la construcción de sistemas altamente escalables.

Disponibilidad: Esta característica se refiere a que un servicio *Web* debe estar listo o disponible para ser consumido en cualquier momento. Ésta se define como la probabilidad que un sistema este habilitado para responder y se vincula de manera directa a la confiabilidad tal como se señala en [71]. *Time To Repair (TTR)* es el tiempo de reparación asociado para resolver la disponibilidad de un servicio. Se desea que un servicio se encuentre disponible inmediatamente al ser invocado.

Interoperabilidad: En [69] se menciona que los servicios *Web* deben ser interoperables entre los distintos entornos de desarrollo utilizados para implementarlos de tal manera que los desarrolladores que usan éstos no tienen que pensar en qué lenguaje de programación o sistema operativo se encuentran alojados.

Seguridad: El proveedor de servicios *Web* puede aplicar diferentes tipos de enfoques y niveles respecto a la prestación de las políticas de seguridad en función del solicitante del servicio. La seguridad ligada a los servicios *Web* significa que éstos proporcionan autenticación respecto a que los clientes que acceden a los datos deben estar autenticados, autorizados de acuerdo al acceso a recursos protegidos por rol, confidencialidad respecto a que los datos solo pueden ser accedidos o modificados por clientes autorizados, trazabilidad y auditabilidad de acuerdo a que debe ser posible realizar un seguimiento al historial de un servicio una vez que una solicitud haya sido servida por este mismo, encriptación de datos respecto a que los datos pueden ser encriptadas cuando el proveedor así lo desee, no repudio de acuerdo con que el cliente no puede negar la solicitud de datos una vez efectuada la misma y, por último, rendición de cuentas respecto a que el proveedor puede cobrar por los servicios brindados.

Para lograr la deseada *QoS*, los mecanismos de calidad que operan al nivel de aplicaciones *Web* deben funcionar junto con aquellos que operan en la red de transporte los cuales son bastante independientes de la aplicación. Los parámetros básicos de *QoS* a nivel de una red de trabajo incluyen el retraso de la red, la variación del retardo y la pérdida de paquetes. El retraso de la red es el tiempo promedio que un paquete atraviesa en una red y puede ser manejado a través de un buen diseño de la misma el cual minimice el número de saltos y promueva una conmutación veloz mediante la selección del camino a través de dispositivos rápidos. La variación del retardo es la variación existente en el tiempo de llegada entre paquetes (que conduce a lagunas o fluctuaciones entre paquetes) introducida por el retraso de transmisión variable sobre la red la cual se soluciona mediante el establecimiento de paquetes en *buffers* manteniéndolos el tiempo suficiente con el objetivo de lograr que los paquetes más lentos lleguen a tiempo respetando así el orden correcto. Internet no garantiza la entrega de los paquetes de información los cuales se pierden cuando existen períodos de congestión en la red, aunque para compensar este problema existen enfoques que incluyen la repetición del último paquete y la transmisión de información redundante. En conclusión, estos mecanismos afectan el manejo de *QoS* para los servicios *Web* ya que conforman la base que sustenta a las aplicaciones *Web*.

6.4. *QoS* en *SOA*

El tema principal en la orientación a servicios es la integración lograda a través de la

utilización de servicios locales y remotos, orquestación de éstos, y la publicación de servicios compuestos. En todas estas interacciones *QoS* es esencial tal como se indica en [66] y aunque no existe una definición unívoca de calidad de servicio respecto a *SOA* es verdad su existencia para las infraestructuras de redes de trabajo como el rendimiento, la disponibilidad, confiabilidad y seguridad para los servicios. *QoS* para los servicios se focaliza en seguridad, la cual es crítica en el logro de interoperabilidad extendida a nivel empresarial.

Como se detalla en la sección 6.3, los niveles de *QoS* para servicios *Web* es extensiva e incluye diferentes dimensiones de calidad. *QoS* en el contexto de orientación a servicios se relaciona a las transacciones comerciales globales llevadas a cabo entre los colaboradores a través de servicios accedidos (ver figura 7).

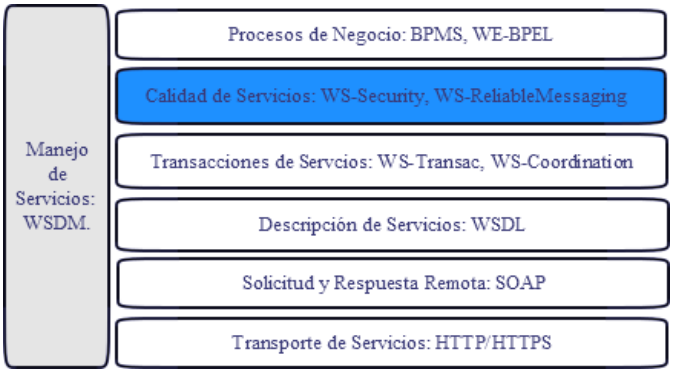


Figura 7: Pila de protocolos que constituyen *QoS* y su relación con los servicios *Web*. Fuente propia elaborada con Dia.

La calidad de servicios es definida usualmente por tres pilares: rendimiento, confiabilidad y seguridad. En el marco de esta tesis, se le añade la interoperabilidad como agregado de relevancia, ya que es objetivo fundamental de una solución orientada a servicios la capacidad de éstos en cuanto a poder ser reutilizados en múltiples unidades de negocio lo cual incrementa el valor comercial. Por lo tanto, esta es la fórmula cualitativa = [Rendimiento + confiablidad + seguridad (+ interoperabilidad)].

6.4.1. Especificaciones para los servicios *Web* vinculadas a *QoS*

La pila de especificaciones relevantes de los servicios *Web* vinculada a *QoS* se puede dividir entre especificaciones de seguridad y de gestión del mensaje: En el primer grupo se encuentran principalmente *WS-Policy* y *WS-Security* mientras que en el segundo grupo se encuentran *WS-Reliablity* y *WS-ReliableMessaging*. A continuación, un breve resumen de los protocolos mencionados:

WS-Policy: En [72] se define el marco de referencias de políticas (*WS-Policy 1.5*) el cual provee un modelo general junto a la sintaxis correspondiente para describir políticas de entidades en un sistema basado en servicios *Web*. Ésta define un conjunto básico de construcciones que pueden ser utilizadas y extendidas por otras especificaciones de servicios *Web*, como *WS-Security,* para describir una amplia gama de requisitos y capacidades de servicio. Cabe destacar que, en un entorno empresarial, puede ser necesario establecer un paralelismo con este modelo en los servicios *ReSTful* mediante el uso de *Open Policy Agent (OPA)* como gestor de políticas de propósito general liviano (para más información ver <https://www.openpolicyagent.org/docs/latest/rest-api>).

WS-Security: Es una especificación que define como se implementan las medidas de seguridad en los servicios *Web* para protegerlos de ataques externos siguiendo principios de confidencialidad, integridad y autenticación. Debido a que los servicios *Web* son independientes de cualquier implementación de hardware y software, los protocolos *WS-Security* deben ser lo suficientemente flexibles para adaptarse a los nuevos mecanismos de seguridad y proporcionar mecanismos alternativos si un enfoque no es adecuado (los mensajes basados en *SOAP* atraviesan múltiples intermediarios a través de la red, por lo cual, los protocolos de seguridad deben ser capaces de identificar la manipulación de datos por terceros e impedir la interpretación de la información por entidades no autorizadas). *WS-Security* combina los mejores enfoques para abordar diferentes problemas de seguridad al permitir que el desarrollador personalice una solución de seguridad particular para una parte del problema (por ejemplo, el desarrollador puede seleccionar firmas digitales para el no repudio y *Kerberos* para la autenticación). Todos los datos relacionados con la seguridad se agregan como parte del encabezado *SOAP*, por lo tanto, la conformación de mensajes es más lenta cuando se activan los mecanismos de seguridad. La seguridad se implementa mediante un encabezado que consiste en un conjunto de pares clave-valor donde el valor se modifica de acuerdo al mecanismo de seguridad subyacente utilizado. Este mecanismo ayuda a identificar la identidad de la persona que llama (si se utiliza una firma digital, el encabezado contiene información sobre como se ha firmado el contenido y la ubicación de la clave utilizada para firmar el mensaje). La información relacionada con el cifrado también se almacena en el encabezado *SOAP*. El atributo *ID* se establece como parte del encabezado *SOAP*, lo que simplifica el procesamiento. Adicionalmente, la marca de tiempo o *timestamp* se utiliza como un nivel de protección contra ataques a la integridad del mensaje. Cuando se crea un mensaje, se asocia una marca de tiempo con el mensaje que indica cuándo se creó (se

usan marcas de tiempo adicionales para la expiración del mensaje y para indicar cuándo se recibió el mensaje en el destino).

Síntesis de los mecanismos de seguridad disponibles en *WS-Security:*

- Enfoque de nombre de usuario y contraseña: La combinación de nombre de usuario y contraseña es un mecanismo básico de autenticación utilizado, y es análogo a los métodos de autenticación basados en *HTTP Digest* y *Basic*. El elemento de *token* de nombre de usuario se utiliza para pasar las credenciales de usuario para la autenticación. La contraseña se puede transportar como texto plano o en formato *digest* (cifrada). Cuando se utiliza el enfoque de *digest*, la contraseña se cifra mediante la técnica de *hashing SHA1*.
- Enfoque X.509: Este enfoque identifica al usuario por una infraestructura de clave pública que mapea el certificado X.509 a un usuario particular. Se puede agregar más seguridad utilizando una clave pública y una clave privada para cifrar y descifrar el certificado X.509. Para asegurarse de que los mensajes no se reproduzcan, se puede establecer un límite de tiempo para rechazar los mensajes que llegan después de un cierto tiempo transcurrido.
- *Kerberos*: El concepto de un boleto o *tikcet* conforma el mecanismo subyacente de *Kerberos*. El cliente debe autenticarse con un centro de distribución de claves o *Key Distribution Center* (*KDC*) utilizando una combinación de nombre de usuario y contraseña o un certificado X.509. En una autenticación exitosa, al usuario se le otorga un *ticket* de concesión de *ticket* o *Ticket Grant Ticket* (*TGT*). Usando el *TGT*, el cliente intenta acceder a un servicio de otorgamiento de *tickets* (*TGS*). En este paso, los dos primeros roles de identificación y autorización han terminado. Luego, el cliente solicita un ticket de servicio o *Service Ticket (ST)* para adquirir un recurso particular del *TGS* y se le otorga el *ST*. Finalmente, el cliente utiliza el *ST* para acceder al servicio.
- Firma digital: Las firmas de los mensajes *XML* se utilizan para proteger el mensaje de modificaciones e interpretaciones. La firma debe ser realizada por una parte confiable o el remitente real.
- Cifrado: La encriptación de los mensajes *XML* se utiliza para proteger los datos de la interpretación al hacerlos ilegibles para un tercero no autorizado. Se pueden utilizar enfoques tanto simétricos (mediante contraseña) como asimétricos (clave pública y privada).

WS-Reliablity y *WS-ReliableMessaging*: Primero *OASIS* creó la especificación *WS-Reliability*, basada en las primeras versiones de *SOAP,* la cual define los requerimientos de intercambio de mensajes confiables entre servicios *W*eb, por ejemplo, incluyendo la política de acuse de recibo cuando es necesario garantizar la recepción del mensaje, pero sin tener en cuenta las condiciones de la red. Posteriormente, varios años después, nació el protocolo *WS-ReliableMessaging* donde se indican los requisitos para que los mensajes *SOAP* se envíen de manera confiable entre aplicaciones distribuidas en presencia de fallas de componentes de *software*, sistemas o redes, incluyendo las políticas de reenvío de mensajes (para más detalle en cuanto a las especificaciones de seguridad y confiabilidad de mensajes en *SOAP* y su contraparte en el estilo de arquitectura *ReST* ver <https://www.apicasystems.com/blog/understanding-security-dependability-soap-rest>).

6.4.2. Identificación positiva de clientes

Hoy en día, los *API Gateways* (ver patrones "único punto de entrada y subsistema de confianza") proveen los siguientes métodos de autenticación (algunos se repiten en las especificaciones de la sección previa):

- Autenticación básica: Manejo de credenciales en texto plano o encriptadas las cuales son verificadas contra un servicio de *backend*.
- *2-way Transport Layer Security (TLS)*: Enfoque de gestión de certificados de seguridad bidireccional (públicos y privados tal como se detalla en la sección anterior).
- *IPSec* es un conjunto de protocolos cuya función es asegurar las comunicaciones sobre el *Protocolo de Internet (IP)* autenticando y/o cifrando cada paquete IP en un flujo de datos. Estos protocolos actúan sobre la capa de red (nivel 3 del modelo *OSI*) mientras que otros protocolos de seguridad como *SSL, TLS y SSH* actúan en la capa de presentación (nivel 7 del modelo *OSI*). Esto hace que *IPsec* sea más flexible, ya que puede ser utilizado para proteger protocolos del nivel 4 del modelo *OSI*, incluyendo *TCP* y *UDP*. Para información detallada respecto a los protocolos de *IPSec* como *Authentication Header (AH), Encapsulating Security Payload (ESP)* e *Internet key exchange (IKE)* ver: <https://es.wikipedia.org/wiki/IPsec>
- *IP whitelisting*: El blanqueamiento explícito de direcciones de *IP* es una práctica que puede llegar a ser compleja sino se puede determinar las direcciones estáticas de las

entidades permitidas. También, el uso de este método reduce el escalamiento horizontal.

- *2-legged OAuth2*: Usar el tipo de concesión "*client_credentials"* entre el proveedor de *API* y el *API Gateway* conlleva a que este último gestione el *ID* del cliente y el secreto de manera segura para cada proveedor de servicios; incluyendo también manejar la caducidad de los tokens de acceso. Al inicio, el *ID* del cliente y el secreto son usados para obtener el *token* de registro del consumidor (normalmente con una frecuencia anual de expiración) el cual se utiliza posteriormente para obtener el token de acceso operativo el cual suele expirar en una frecuencia establecida en horas (usualmente menos de 1 día).
- *Signed Tokens*: La generación de un *token* firmado para cada llamada a los servicios disminuye notablemente la performance, por lo cual, es imperativo el almacenamiento en caché de *tokens* con fecha de expiración. Los tokens pueden ser auto emitidos o emitidos por un tercero. Por ejemplo, *JSON Web Token* (*JWT*).
- Sin chequeo de autorización: Dado que la propagación de la identidad a menudo ocurre dentro de la red de la organización, muchos administradores asumen ingenuamente que es seguro propagar los datos sin protección. Pero en realidad, la mayoría de las brechas de seguridad ocurren dentro de la red interna de la organización.

6.4.3. Rendimiento y *benchmarketing*[31]

El rendimiento se vincula con la entrega de los mensajes y la invocación de los servicios tal como se detalla en la sección 6.3. Algunas aplicaciones deben cumplir con objetivos de desempeño y tiempo de respuesta lo cual puede resultar un reto para las implementaciones de los servicios. Simplemente presionando sobre un botón se puede producir un llamado a múltiples componentes, plataformas y sistemas involucrados en la interacción de ida y vuelta de la solicitud del servicio y la posterior respuesta. La invocación de los servicios se debe analizar holísticamente para poder asegurar que el rendimiento de cada componente participante en la interacción cumple con los

[31] *Benchmarketing* es una técnica con la que se trata de comparar una compañía con otra líder en el mercado. Esta comparación tiene la finalidad de obtener información que ayude a mejorar las prácticas de la empresa (en este contexto se refiere a la comparación del rendimiento de los servicios de las empresas).

requerimientos de desempeño y tiempos de respuesta establecidos.

Networking: Las empresas orientadas a servicios dependen en gran medida de las redes de trabajo, tanto para la conexión intranet dentro de la empresa, así como la integración en Internet con otras empresas. Las diferentes capas de las redes de trabajo permiten la comunicación entre los servidores contendores de servicios y es *Transmission Control Protocol/Internet Protocol (TCP/IP)* el protocolo más popular de las arquitecturas orientadas a servicios. La capa de nivel inferior es denominada link y en esta se encuentra el *hardware* base como dispositivos físicos y los controladores de los mismos; la siguiente capa es *IP* la cual proporciona la comunicación de las redes de trabajo organizada en paquetes de información (incluyendo encabezado y contenido) através de Internet e Intranets; a continuación se presenta la capa *TCP* la cual es responsable del orden y la confiabilidad de los paquetes transmitidos; finalmente los mensajes *SOAP* son transportados a través de mensajes *HTTP* y aunque otros protocoles como el protocolo *JMS* puede proporcionar una tecnología similar mediante encolamiento de trabajo es el primero el más popular en Internet.

XML y JSON: El rendimiento en los servicios es crítico, por lo cual, es una característica clave de *QoS* en la selección de servicios *Web*. Comúnmente los servicios se basan en *XML* para el estilo basado en mensajes *SOAP* y en *JSON* para *ReST,* por lo cual, éstos heredan todas las ventajas y desventajas del uso de estas estructuras. *XML* es elegante y legible por humanos, aunque, comparado con el texto críptico y representaciones binarias, es muy detallado teniendo en consecuencia un rendimiento inferior motivo por el cual el uso de notaciones *Java Script Object* es considerado más liviano. No sólo es el espacio asignado a los mensajes o documentos *XML* mucho mayor, sino que también existe una sobrecarga asociada al parseo del documento *XML* respecto a la verificación del mismo de acuerdo a si se encuentra bien formateado; y además, dependiendo el estilo, a veces el *XML* debe ser validado contra un *XML Schema* de acuerdo al cumplimiento de los valores aceptados por el documento (esto segundo también es también válido en el caso de *JSON* con validadores de esquema). Toda esta sobrecarga es y sigue siendo una de las razones de acuerdo al porque muchas empresas que deben procesar grandes volúmenes de transacciones poseen formatos propietarios diferentes al uso de servicios *Web;* aunque en los últimos años las tecnologías de parseo *XML/JSON* han mejorado considerablemente. Algunas de las tecnologías que intentan mejorar el desempeño de *XML y JSON* son, por

un lado en el caso *SOAP, XML Binary Information Set (XBIS)* <http://xbis.sourceforge.net/> siguiendo un encuadre de representación binaria del documento *XML* y *XMill* <http://www.liefke.com/hartmut/xmill/xmill.html> mediante la compresión (codificación) convencional; y por el otro en el caso de servicios *ReSTful, Apache Avro* <https://avro.apache.org/docs/current/> el cual es un sistema de serialización que compacta los datos en formato binario de manera eficiente y rápida (similar al enfoque anterior). Finalmente, desde que han emergido *XML y JSON,* especialmente con *SOAP* y otras soluciones como *ReST* y microservicios*,* se puede concluir que estas tecnologías son ubicuas y parece ser poco probable que surja otro formato que gane popularidad y acepción masiva en la industria, aunque recientemente haya surgido *GraphQL* tal como se indica en [32] el cual podría ser considerado como una tercera opción en un futuro cercano (ver sección 3.2 para más detalles).

SOAP en servicios *Web* clásicos: Como se detalla en la sección 2.2, gran parte de los estándares de servicios están basados en *XML*. *SOAP* es tal vez el protocolo primordial para el intercambio de mensajes mientras que *WSDL* define el estilo del mensaje basado en documento o *RPC*:

- Documento: En esta opción no existe la definición del formato de la operación o método dentro del cuerpo <*soap:body*> de un mensaje *SOAP* y tanto el emisor como el receptor deben acordar el contenido de las partes del mensaje *SOAP* dentro del <*soap:body*> ya que los datos se definen en la sección <types> normalmente mediante el uso de *XSD* (esquema de validación de datos).
- *RPC*: El <*soap:body*> contiene la estructura del método invocado (o su retorno), incluyendo el nombre del método y sus parámetros, por lo tanto, la sección </types> se encuentra vacía y tampoco se usan esquemas de validación.

En base a lo anterior, la diferencia fundamental es que el estilo basado en documento responsabiliza al receptor en el parseo y procesamiento de la solicitud / respuesta mientras que en el estilo *RPC* existe un motor *SOAP* responsable del parseo y la llamada de la implementación de servicio adecuada respecto a una función determinada dando como resultado que el estilo literal de documento tiene un mayor rendimiento respecto a la delegación a los receptores aunque *RPC* es más fácil de entender para los desarrolladores. Varios objetos interactúan desde que un cliente hace un requerimiento de servicio hasta la respuesta del servidor. Una secuencia típica de aplicaciones *SOAP* comienza cuando

un cliente invoca a una *application program interface (API*[32]*)*; esta implementación *API* genera el requerimiento *SOAP* el cual es enviado al servidor *Web* a través del protocolo *HTTP*; del lado del servidor la información se recibe a través del protocolo *HTTP* y se invoca al motor *SOAP*, típicamente como un *servlet*; el motor *SOAP* representa cualquier servicio o servidor que toma el mensaje *SOAP*, lo valida, lo parsea y posteriormente llama al servicio apropiado a través de una *API* específica de un lenguaje en particular (Por ejemplo, llamada de una *Java API*) –Ésta es la llamada a un servicio de un lenguaje determinado, el cual se encuentra desplegado dentro de un servidor *SOAP* el cual sabe como localizarlo; finalmente los resultados del servicio son retornados al motor *SOAP* el cual popula la respuesta y la trasfiere al servidor *Web* el cual es responsable de otorgarle la respuesta al cliente a través del protocolo *HTTP*. Por lo tanto, de manera similar a otros entornos distribuidos, existe un *marshaling* y *unmarshaling* de mensajes *SOAP* desde el cliente al servidor del servicio *Web*, ya que con *SOAP* como intermediario se usa el modelo requerimiento/respuesta para lograr interoperabilidad.

La accesibilidad, confiabilidad y disponibilidad respecto a la calidad de los servicios se traduce en el grado de acuerdos o *service level agreements (SLAs)* existentes entre las entidades participantes (cliente y servidor) ya que en un *SLA* se definen los tiempos de respuesta. *Service requests* sincronizados pueden ser medidos en términos de microsegundos de acuerdo a los tiempos de respuestas, y la medición de los *requests* no sincronizados está relacionada con garantizar el tiempo en el cual el servidor se compromete a proveer una respuesta simple, el cual puede ser establecido en términos de segundos, minutos, horas y hasta días. Típicamente, los *requests* involucran acuerdos asociadas a transacciones de negocio a cumplir: Por ejemplo, si un cliente emite una orden de compra, el proveedor debe comprometerse en responder en un tiempo de respuesta previamente establecido -usualmente mediada en días- indicando que la orden fue aceptada y entregada. Tanto los *SLA* de los tiempos de respuesta vinculados con *requests* de servicios *Web* sincronizados como aquellos acuerdos respecto a las transacciones de negocio son compromisos críticos que garantizan la calidad general de las aplicaciones orientadas a servicios.

El concepto asociado a los *SLA* es la disponibilidad de los servicios ya que en éstos se especifican los límites respecto al tiempo fuera de servicio y el máximo tiempo de

32 Las *APIs* se suelen denominar implementaciones *client proxy* o intermediarias para la invocación de un servicio en lenguajes orientados a objetos.

respuesta aceptable en los cuales un servicio *Web* se considera operacional. Esto requiere que el proveedor del servicio cuente con un entorno (servidor de aplicaciones) tolerante a fallas y capaz de cumplir con la disponibilidad requerida por los clientes. Hoy en día, la mayoría de los servidores de aplicaciones (por ejemplo, *IBM WAS*) permiten la creación de granjas de servidores en *cluster*, de tal manera que si el estado de un servidor se encuentra fuera de servicio existe una redistribución de carga hacia otro servidor disponible como así también un balanceo optimizado del procesamiento entre la granja de servidores que garantizan la disponibilidad de los servicios. Esta tolerancia a las fallas en el entorno de un proveedor de servicios es esencial para lograr el objetivo de disponibilidad de servidos y tiempo de respuesta adecuados debido a que el uso de *clustering* también sirve para mejorar el rendimiento mediante mecanismos de paralelismo de trabajo balanceado.

Arquitectura de múltiples capas: La disponibilidad y los tiempos de respuestas de los servicios dependen de ciertos factores críticos como el rendimiento de la implementación del servicio, el *middleware* que se comunica con el servicio, el servidor de aplicaciones, y la disponibilidad de las comunicaciones en la red. En una arquitectura punto a punto de tipo cliente servidor la solicitud puede atravesar varios componentes; el acceso es a través del *firewall* externo del proveedor de servicios, y entre éste y el *firewall* interno se encuentra la zona desmilitarizada *(DMZ)* (ver figura 8).

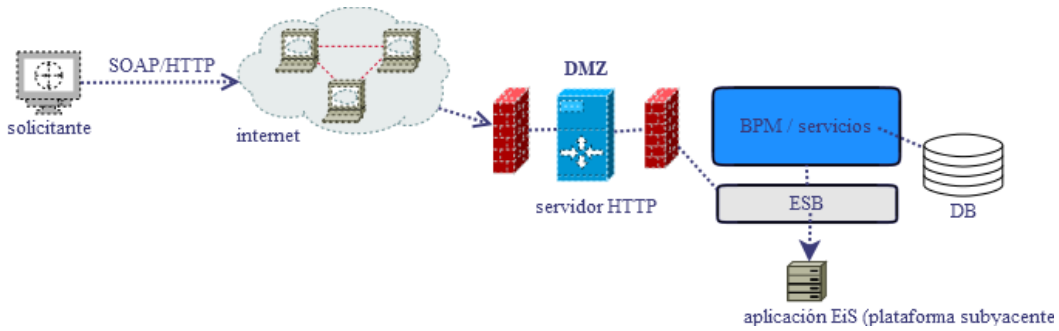


Figura 8: Ilustración de una simple arquitectura punto a punto en la cual el cliente solicita a *BPM*, donde se encuentran los servicios, a través del *ESB*; la información se persiste en la base de datos y *BPM* también puede acceder aplicaciones *back-end Executive Information Systems (EIS)* obteniendo información de negocio a través del *ESB*. Fuente elaborada con Día.

7. Inventariado de patrones de diseño de servicios para mejorar *QoS*

7.1. Prólogo para comprender el catálogo de patrones

En la sección 2.5 se establece que la orientación al servicio es un paradigma de diseño que consta de un conjunto de principios de diseño, cada uno de los cuales proporciona una norma general o una guía para realizar ciertas características de diseño. El paradigma en sí parece bastante completo, y en realidad lo es. Sin embargo, para aplicarla con éxito en el mundo real se requiere algo más que una comprensión teórica de sus principios y esto se debe a que la realización de las características de diseño deseadas con frecuencia se dificulta por varios factores, que incluyen: Restricciones impuestas por los requisitos y prioridades del proyecto vinculada a la entrega de valor de negocio (por ejemplo, la entrega de servicios de utilidad que generan menos valor de negocio aparente debe ser coordinada con una estrategia de negocio global la cual involucre servicios de las capas intermedia y superior) y restricciones impuestas por la tecnología o los sistemas que residen o se usan durante la construcción del proyecto (un ejemplo es el uso de una base de datos no relacional, como dependencia de un servicio, la cual genera redundancia de datos).

A modo de resumen, en la sección 5.1 se indica que un patrón de diseño documenta la solución en un formato de plantilla común para que pueda aplicarse repetidamente. El conocimiento de los patrones de diseño no solo le brinda una comprensión de los problemas potenciales a los que se pueden someter los diseños, sino que también proporciona respuestas sobre la mejor manera de tratar estos problemas. Los patrones de diseño nacen de la experiencia. Se hubo que someter a ciclos de prueba y error y aprender de lo que no funcionó, por lo tanto, cuando un problema y su solución correspondiente se identificaron como suficientemente comunes, se forma la base de un patrón de diseño. Es importante remarcar que una serie de patrones que utilizan una tecnología de arquitectura común puede conformar un catálogo que resuelve un problema más grande.

7.2. Catálogo de patrones de diseño de servicios para mejorar *QoS*

Lo patrones de diseño de servicios en *SOA* se categorizan en dos grandes grupos, aquellos

que afectan a los servicios de manera independiente y los que afectan al inventario de servicios. Por un lado, en el primer grupo se pueden clasificar en patrones de seguridad, mensajería, gestión, transformadores, contratos, herencia encapsulada, capacidades, composición, originados en *ReST*, e implementación y despliegue. Por otro lado, en el segundo grupo es posible clasificarlos en patrones de centralización, lógicos, institucionales, gestión, e implementación.

7.2.1. Patrones de los servicios independientes:

7.2.1.1. Patrones de seguridad:

7.2.1.1.1. Nombre: Excepciones blindadas (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño advierte que los servicios *Web* protejan la exhibición de errores con el objetivo de evitar problemas de seguridad asociados a la exposición de datos sensibles.

Nombre alternativo: Excepciones protegidas.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Exposición de información sensible que compromete la seguridad de los servicios y el entorno subyacente a través de excepciones.

Solución: Los servicios *Web* hacen un saneamiento de las excepciones antes de ser presentadas a los consumidores.

Aplicación: Diseño e implementación que incluye lógica para el saneamiento de datos sensibles en excepciones de servicios. Por ejemplo, mediante una funcionalidad de depuración de datos, usando aspectos[33], que se añade a la lógica principal del servicio la cual se habilita cuando ocurren excepciones y las formatea de manera adecuada para que éstas sean exhibidas ante los clientes sin detalles sensitivos.

Efectos: Mayor complejidad para investigar las causas de las excepciones ya que estas últimas se entregan con pocos detalles a los consumidores.

[33] Aspectos en este contexto indica el uso del paradigma de programación orientada a aspectos o *Aspect Oriented Programming (AOP)* el cual fomenta una apropiada modularización de las aplicaciones y posibilita una mejor separación de responsabilidades; por ejemplo, con anotaciones que se ejecutan en tiempo de ejecución las cuales son escritas en un módulo separado al código principal.

Principios soportados: Abstracción de servicios.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+-0))].

Menor rendimiento:

⊖ Ya que añadir una funcionalidad para el saneamiento de información aumenta los tiempos de respuestas.

Incremento de seguridad:

⊕ En cuanto a la presentación de excepciones de servicios ante los consumidores ya que éstas fueron diseñadas para no proveer datos sensibles.

7.2.1.1.2. Nombre: Verificación de mensajes (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño señala que los servicios deben validar los mensajes de ingreso con el propósito de detectar información peligrosa y así evitar problemas de seguridad.

Nombre alternativo: Detección de mensajes.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Atacantes que transmiten mensajes con contenido malicioso o formato invalido para vulnerar la seguridad de los servicios y el ambiente subyacente.

Solución: Los servicios *Web* verifican los mensajes de entrada, suponiendo que contienen información peligrosa, antes de procesarlos.

Aplicación: Desarrollo de funcionalidad que analiza el contenido de los mensajes de ingreso en búsqueda de información dañina. Por ejemplo, al igual que el patrón anterior, se puede escribir un aspecto, en un módulo separado, con código que se especialice en comprobaciones de mensajes de tal manera de no afectar la lógica principal.

Efectos: Cada intercambio de mensaje implica procesamiento adicional en cuanto a la validación de datos en tiempo de ejecución. También, las comprobaciones pueden requerir funcionalidad especializada para chequear contenido binario, como datos adjuntos, aunque en algunos casos no es factible validar todas las formas posibles de contenido dañino, por ejemplo, en el caso de un ataque *Zip Bomb*[34].

[34] *Zip Bomb* es un archivo malicioso (comprimido) que se encuentra diseñado para bloquear o inutilizar el programa o sistema que lo lee. El desempaquetado de estos archivos, por ejemplo, mediante un

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

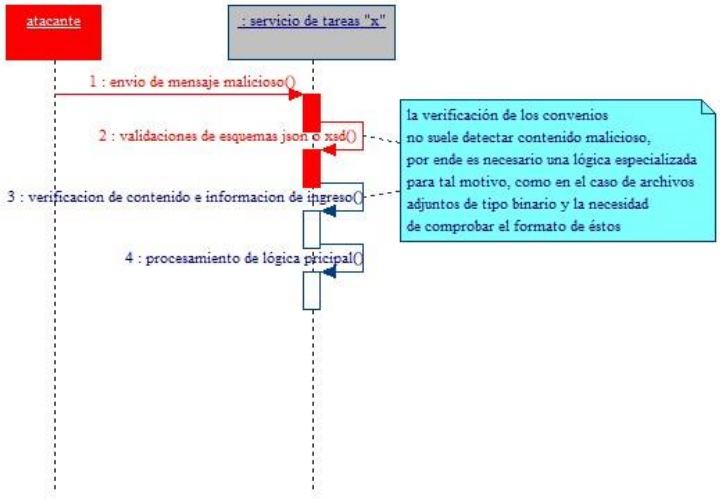


Figura 38: La presencia de lógica de detección de mensajes maliciosos evita que estos dañen a los servicios o el entorno subyacente. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+-0))].

Disminución de rendimiento:

⊖ Debido a que agregar una lógica para comprobar los datos y la estructura de los mensajes añade latencia al procesamiento.

Incremento de confiabilidad:

⊕ Ya que la conducta de los servicios se hace más previsible mediante el descarte de mensajes inválidos que pueden generar comportamiento inesperado.

Aumento de seguridad:

⊕ Respecto a la detección de mensajes maliciosos antes de ser procesados por la lógica principal del servicio *Web*.

7.2.1.1.3. Nombre: Servicio de protección perimetral (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño indica el establecimiento de un servicio de guardia perimetral como punto de enlace seguro entre los consumidores externos y los servicios de la empresa.

escáner de virus, requiere una cantidad excesiva de tiempo, espacio en disco o memoria. Más información en: <https://stackoverflow.com/questions/1459673/how-does-one-make-a-zip-bomb>

Nombre alternativo: Cortafuego.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Clientes externos que obtienen acceso a recursos internos de la empresa mediante el ataque a servicios de una red privada.

Solución: Establecimiento de un servicio intermediario, como punto de contacto seguro entre los consumidores externos y los servicios internos, ubicado en el perímetro de la red privada.

Aplicación: Instauración de un servicio de cortafuegos o *firewall*, como *WAF*[35] para *AWS*, el cual está diseñado para funcionar como un mecanismo de puente seguro entre las redes externas e internas.

Efectos: La existencia de un servicio perimetral agrega complejidad e incremento de procesamiento debido a que esta lógica intermedia participa en todas las comunicaciones entre consumidores externos y los servicios de la empresa.

Principios soportados: Abstracción de servicios y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

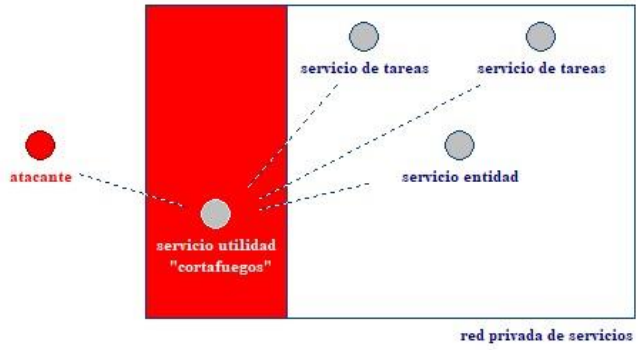


Figura 39: El servicio perimetral procesa el mensaje del atacante y, al determinar su intención maliciosa, lo rechaza (esto evita la exposición del servicio interno subyacente y el procesamiento innecesario relacionado con la seguridad). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) +

[35] *Web Application Firewall (WAF)* es un cortafuegos para aplicaciones *Web* que ayuda a protegerlas de ataques externos que ponen en riesgo la seguridad de la empresa. El *WAF* puede ser instalado en frente de *API Gateways* como *Kong* el cual se usa como escudo de los servicios del inventario (Ver sección 7.2.2.5.4). Para más información referirse a: <https://aws.amazon.com/es/waf/>

seguridad (+2) (+ interoperabilidad (+-0))].

Disminución rendimiento:

⊖ Ya que la existencia de lógica intermediaria para detención de ataques añade un nivel de procesamiento lo cual aumenta los tiempos de respuesta.

Mayor seguridad:

⊕ En cuanto a que la presencia de un servicio de cortafuegos sirve de primer filtro respecto a mensajes maliciosos y, también, mediante políticas de seguridad evita ataques *DoS*[36] los cuales afectan la disponibilidad de los servicios de la empresa.

⊕ Debido al incremento de integridad referido a impedir el acceso no autorizado sobre componentes del servicio o la modificación de los datos vinculados a la interrelación entre los colaboradores.

7.2.1.1.4. Nombre: Subsistema de confianza (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño se refiere a la instauración del servicio que se comporta como un subsistema de confianza para verificar las credenciales de los consumidores externos que intentan acceder a recursos de la empresa (como los servicios del inventario).

Nombre alternativo: Servicio de identidad.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Clientes externos acceden a recursos internos de los servicios de manera directa comprometiendo la integridad de los datos.

Solución: Los servicios subyacentes confían en servicios de identidad para realizar funciones de seguridad incluyendo la verificación de credenciales (autenticación) y la autorización al acceso de recursos en base a roles.

Aplicación: Establecimiento de un *API Gateway* o servicio de identidad, como por ejemplo *Kong* (ver sección 7.2.2.5.4), para la identificación positiva de clientes y posterior autorización al acceso de recursos (ver sección 6.4.2. para información respecto a los diferentes métodos de autenticación disponibles).

[36] *Denial of Service Attacks (DoS)* es un ciberataque en el cual el perpetrador busca hacer que una máquina o recurso de red no esté disponible para sus usuarios previstos al interrumpir temporalmente o indefinidamente los servicios de un host conectado a *Internet*. Ver el siguiente enlace respecto a tipos de ataque *DoS*: <https://en.wikipedia.org/wiki/Denial-of-service_attack>

Efectos: Si este tipo de servicio es comprometido por atacantes o consumidores no autorizados, puede ser explotado para obtener acceso a una amplia gama de recursos subyacentes.

Principios soportados: Acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+2) (+ interoperabilidad (+-0))].

Menor rendimiento:

⊖ En cuanto a que este servicio añade una lógica de autenticación por cada llamada a los servicios, la cual, si no es gestionada de manera performante (por ejemplo, usando mecanismos de cache) incrementa los tiempos de acceso.

Mayor seguridad:

⊕ Respecto al establecimiento de funcionalidad de autenticación y autorización para el acceso a recursos como servicios subyacentes (por ejemplo, un servicio de base de datos) los cuales se encuentran protegidos de entidades externas no autorizadas.

⊕ Ya que existe mayor integridad referida a impedir el acceso no autorizado sobre componentes del servicio evitando la incorrecta manipulación de los recursos.

7.2.1.1.5. Nombre: Autenticación vía intermediario (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño indica el establecimiento del agente intermediario para autenticar a los consumidores y permitirles que se comuniquen con los servicios.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La autenticación directa (ver sección 7.2.1.1.8) es ineficiente o incluso imposible cuando los consumidores y los servicios no confían entre sí y los clientes tienen que acceder a múltiples componentes en la nube como parte de la misma actividad en tiempo de ejecución.

Solución: Instauración del agente intermediario, conformado por un almacén central de identidad, que asume la responsabilidad de autenticar al consumidor y emitir la ficha o *token* (ver secciones 6.4.1 y 6.4.2) que el cliente usa para acceder múltiples servicios.

Aplicación: El producto de autenticación se añade a la arquitectura de inventario llevando a cabo la autenticación intermedia y emisión de credenciales temporales utilizando

tecnologías como los certificados *X.509* o los *tokens Kerberos* (ver secciones 6.4.1 y 6.4.2).

Efectos: La violación de acceso al agente intermediario de autenticación pone en peligro todo el inventario de servicios, convirtiéndolo en el único punto de falla.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

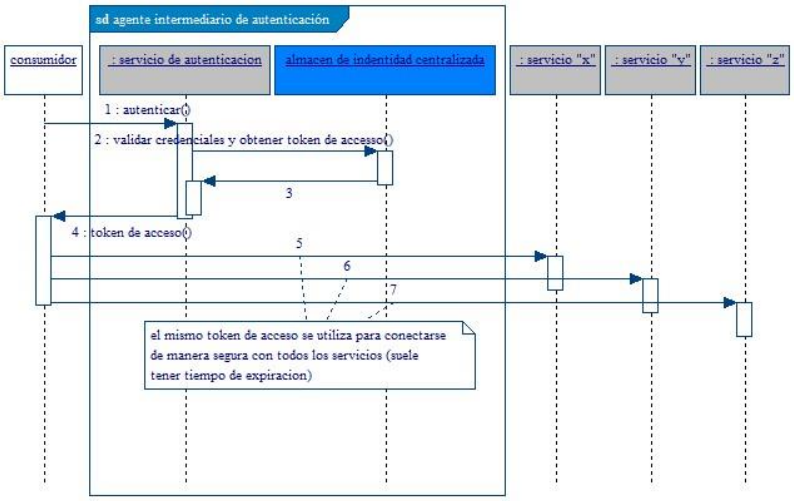


Figura 61: Primero**,** el consumidor envía una solicitud con credenciales al agente de autenticación intermediario el cual verifica el acceso en base al almacén de identidad central. Posteriormente, el agente de autenticación responde con un *token* (ver sección 6.4.2) que el cliente utiliza para acceder a los servicios (x, y, z). Vale aclarar que ninguno de los servicios (x, y, z) posee un almacén de identidad independiente, sino que la confianza establecida es a través del almacén central del agente intermediario. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+-0))].

Aumento de rendimiento:

⊕ En cuanto a que con la implementación de este patrón existe comunicación eficiente y segura entre el cliente y múltiples servicios.

Mayor confiabilidad:

⊕ Referida a que todos los componentes en la nube son accedidos de la misma forma a través de un agente intermediario con un almacén de identidad centralizado.

Incremento de seguridad:

⊕ Ya que el agente intermediario de autenticación permite que los clientes se

comuniquen con los servicios de manera protegida.

7.2.1.1.6. Nombre: Confidencialidad de los datos (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño se refiere a la encriptación del contenido de los mensajes de servicio, independientemente del transporte, con el fin de asegurar que solo los destinatarios adecuados puedan comprender la información protegida.

Problema: En las composiciones de servicio, a menudo se requiere que los datos pasen a través de uno o más intermediarios. Los protocolos de seguridad punto a punto, como los que se utilizan con frecuencia en la capa de transporte, pueden admitir que los intermediarios intercepten y visualicen los mensajes que contienen información confidencial.

Solución: El contenido del mensaje se cifra aparte del transporte, lo que garantiza que sólo los destinatarios deseados puedan acceder a los datos protegidos.

Aplicación: Se aplica un algoritmo de cifrado y descifrado simétrico (mediante contraseña) o asimétrico (clave pública y privada) a nivel de mensaje, tal como se explican en la sección 6.4.1.

Efectos: Mayor complejidad de gestión en cuanto a la administración de claves.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

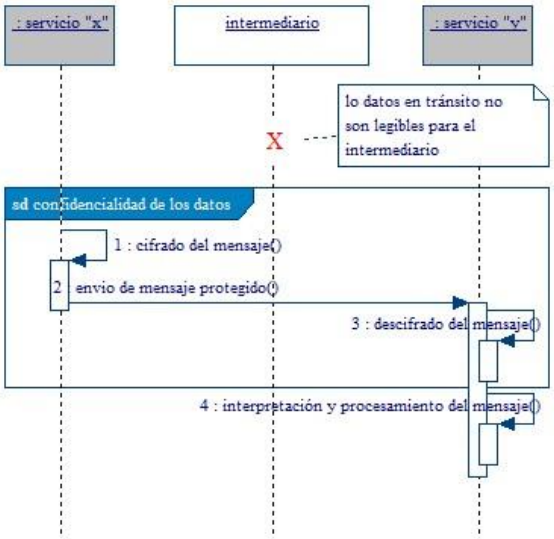


Figura 62: El cifrado de la información protege los mensajes mientras se encuentran en tránsito, previniendo la interpretación de datos confidenciales por parte de terceras partes no autorizadas. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+-0))].

Menor rendimiento:

⊖ Debido a que la encriptación y desencriptación de todo el contenido del mensaje requiere mayor tiempo de procesamiento operativo.

Aumento de seguridad:

⊕ Ya que el cifrado del contenido del mensaje, independientemente de la capa de transporte, evita que intermediarios no autorizados puedan observar información confidencial.

7.2.1.1.7. Nombre: Origen auténtico de los datos (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño señala la firma digital de los mensajes, separada del transporte, para garantizar la autenticidad del origen y evitar la alteración de datos en tránsito.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: En las composiciones de servicio, usualmente se requiere que los datos pasen a través de uno o más intermediarios. Los protocolos de seguridad punto a punto, como los que se utilizan a menudo en la capa de transporte, pueden permitir que los intermediarios intercepten y alteren los mensajes que contienen información confidencial.

Solución: El mensaje se firma digitalmente para que los destinatarios puedan verificar que se originó en la fuente esperada y sin ser manipulado durante el tránsito.

Aplicación: Se aplica el algoritmo de firma digital al mensaje para proporcionar una "prueba de origen", lo que permite que la información confidencial del mensaje esté protegida contra manipulación de terceros. Esta tecnología debe ser soportada tanto por la fuente como por el servicio destino.

Efectos: Los intermediarios pueden observar los datos en tránsito, aunque sin posibilidad de manipulación.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

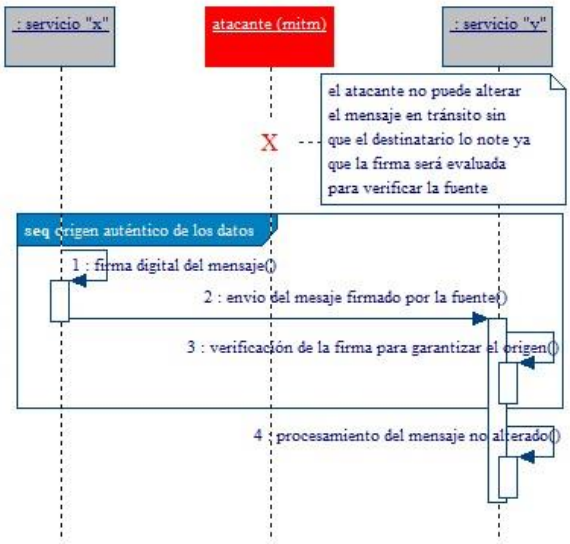


Figura 63: La firma digital de los mensajes en tránsito evita que éstos sean manipulados por terceras partes no autorizadas. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+1 -0,5) (+ interoperabilidad (+-0))].

Mayor rendimiento:

⊕ Comparado con el encriptado de todo el contenido del mensaje (ver sección 7.2.1.1.6) ya que la firma digital sólo se añade como parte del encabezado del mensaje o como una sección del cuerpo, lo que resulta en menor tiempo de procesamiento para la verificación de la firma por parte de los participantes comparado con el cifrado y descifrado del mensaje completo.

Incremento de seguridad:

⊕ En cuanto a que la firma digital del mensaje, independientemente de la capa de transporte, evita los ataques de intermediarios o de tipo *man in the middle*[37] *(MITM)* los cuales alteran el contenido de los datos entre la fuente y el destino.

Menor seguridad:

⊖ Comparado con el cifrado de todo el mensaje ya que los intermediarios pueden visualizar la información confidencial en tránsito (sin poder alterarla).

7.2.1.1.8. Nombre: Autenticación directa (Delgado, Smith, Taylor).

Sinopsis: Este patrón de diseño se refiere a que los servicios posean almacenes de

[37] *Man in the middle (MITM)* es un ataque en el que el atacante retransmite y posiblemente altera las comunicaciones entre dos partes las cuales confían que se están comunicando directamente entre sí. Ver más información en: <https://en.wikipedia.org/wiki/Man-in-the-middle_attack>

identidad para autenticar de forma directa a los consumidores.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Atacantes que acceden a funcionalidad privada de servicio o datos confidenciales destinados a un grupo reducido de consumidores con la intención de poner en peligro el negocio de la empresa.

Solución: Establecimiento de servicios con funcionalidad para poder autenticar a los consumidores de manera directa.

Aplicación: Implementación de servicios compuestos con almacenes de identidad, lo que les permite autenticar a los consumidores directamente.

Efectos: Este patrón fomenta el uso de múltiples almacenes de identidad, lo que añade dificultad en la gestión de los componentes en la nube. Además, los consumidores deben proporcionar credenciales compatibles con la lógica de autenticación de los servicios.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1,5) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+-0))].

Disminución de rendimiento:

- ⊖ En cuanto a que la aplicación de este patrón es ineficiente o incluso irrealizable cuando los clientes y los proveedores no confían entre sí y los consumidores deben acceder a múltiples servicios como parte de la misma actividad en tiempo de ejecución (ver sección 7.2.1.1.6) debido al procesamiento vinculado a la existencia de diversos almacenes de identidad.

Aumento de seguridad:

- ⊕ Ya que no es factible el acceso a funcionalidad confidencial de servicio por parte de consumidores no autorizados.

7.2.1.2. Patrones de mensajería:

7.2.1.2.1. Nombre: Encolamiento asíncrono de mensajes (Little, Simon).

Sinopsis: Este patrón de diseño indica el estableciendo de un mecanismo intermedio de encolamiento punto a punto o *Point to Point* (*P2P*) para el envío de mensajes asíncronos

entre servicios, lo cual mejora la performance percibida.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: El sistema de encolamiento *P2P* (sólo un servicio recibe el menaje) puede ser considerado un servicio de utilidad que afecta a todos los otros tipos (de tareas, entidad y utilidad).
- Por estilo: *SOAP* mediante el protocolo *JMS* + una tecnología que permite el intercambio de mensajes *P2P* como *IBM MQ*[38] (ver sección 2.2) y *ReSTful* a través de un sistema de mensajería como *RabbitMQ*[39] con enrutamiento *P2P*.

Problema: Reducción de performance y confiabilidad causada por bloqueos de procesamiento de una solución con servicios síncronos que poseen alta frecuencia de uso.

Solución: Los servicios intercambian mensajes con los consumidores mediante un sistema de mensajería intermedio *P2P* que los encola, lo que permite que el servicio y el consumidor procesen el mensaje de manera independiente permaneciendo desconectados temporalmente (disminuyendo el bloqueo en espera de procesamiento).

Aplicación: Instauración de tecnologías que admiten encolamiento de mensajes *P2P* como *JMS +IBM MQ* (ver sección 2.2) en los servicios *SOAP* o *RabbitMQ* para los *ReSTful.* También, los sistemas de encolamiento proveen mecanismos para el respaldo de datos en caso de pérdidas.

Efectos: Dependiendo de la estrategia de implementación seleccionada, es factible que los mensajes entregados de manera exitosa no se confirmen. Adicionalmente, no se suelen admitir transacciones atómicas en este tipo de solución.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y servicio sin estado.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

[38] *IBM MQ* es un sistema intermediario creado por *IBM* para encolamiento y procesamiento de mensajes *JMS* que admite, entre varias estrategias, los envíos de datos *P2P* y también mensajería dirigida por eventos siguiendo el modelo publicar/subscribir. Ver más información en: <https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q001020_.htm>.

[39] *RabbitMQ* es uno de los sistemas de mensajería de código abierto más utilizado del mercado. Esta tecnología de intercambio de mensajes soporta, entre múltiples mecanismos, el enrutamiento *P2P* de los datos. Para más detalle referirse a: <https://www.rabbitmq.com/>.

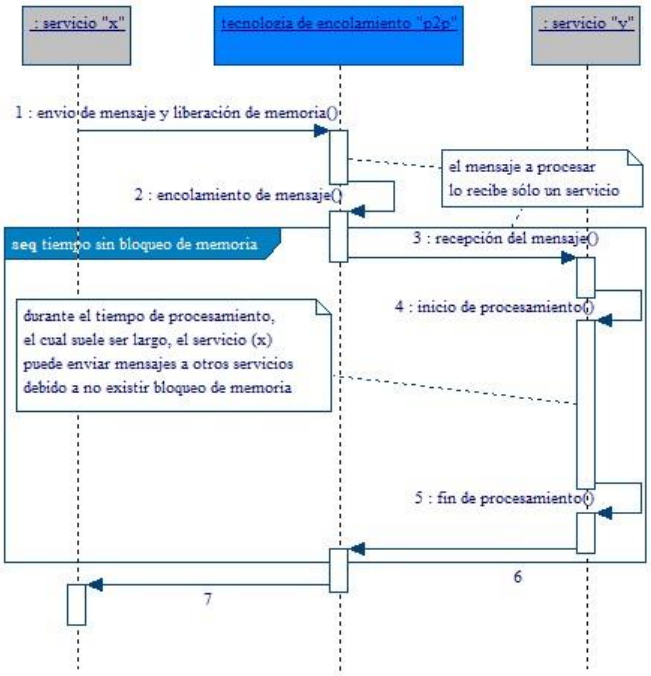


Figura 52: El servicio (x) envía un mensaje otro servicio (y), mediante una tecnología de mensajería de encolamiento *P2P*, liberando su memoria (lo que le permite poder iniciar conversaciones con otros servicios si fuere necesario). Posteriormente, el servicio receptor (y) recibe el mensaje a través de la tecnología intermediaria, lo procesa y finalmente envía la respuesta de vuelta. Finalmente, el servicio inicial (x) obtiene la respuesta de la tecnología intermedia sin necesidad de bloquear su memoria ante la espera del procesamiento de la respuesta. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de rendimiento:

- Referido a los tiempos de respuesta debido a que el proceso asíncrono de mensajes, mediante el uso de tecnología de encolamiento *P2P*, permite que los consumidores/servicios hagan un mejor uso de memoria evitando bloqueos en espera de procesamientos (como cuando se usan mecanismos síncronos).

Mayor confiabilidad:

- En cuanto al comportamiento esperado de los servicios bajo situaciones de alta demanda de solicitudes concurrentes.

7.2.1.2.2. Nombre: Mensajería dirigida por eventos (Little, Simon).

Sinopsis: Este patrón de diseño señala la instauración de un sistema de mensajería basado

en el modelo publicar/subscribir para el envío de eventos asíncronos entre servicios, lo cual es adecuado para el caso donde múltiples consumidores reciben el mismo evento.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: El sistema de mensajería dirigido por eventos (varios servicios reciben el mismo mensaje) puede ser considerado un servicio de utilidad que afecta a todos los otros tipos (de tareas, entidad y utilidad).
- Por estilo: *SOAP* mediante el protocolo *JMS* + una tecnología que permite el intercambio de eventos como *IBM MQ* (ver secciones 2.2 y 7.2.1.2.1) y *ReSTful* a través de un sistema de mensajería conducido por eventos como *Apache Kafka*® (ver sección sección 7.2.1.10.8).

Problema: Clientes que no pueden aprender de los eventos relevantes que ocurren dentro del contexto funcional de los servicios ya que no existe una forma eficiente de recuperar los datos existentes.

Solución: El servicio publica eventos en canales a los cuales se encuentran subscriptos uno o más consumidores. A medida que llegan los mensajes al canal, los clientes subscriptos los toman y marcan como leídos (dependiendo la estrategia, cada consumidor puede marcar su lectura por separado o competir en la lectura con los otros participantes). Si el consumidor no llega a poder visualizar el mensaje, al no marcarlo como leído, lo puede volver a observar (en el caso de una estrategia de lectura individual).

Aplicación: Establecimiento de un sistema con tecnología de mensajería conducida por eventos como *JMS +IBM MQ* (ver secciones 2.2 y 7.2.1.2.1) en los servicios *SOAP* o *Apache Kafka*® para los *ReSTful*. Además, los sistemas de mensajería proveen mecanismos para el seguimiento de datos.

Efectos: Mediante el uso de este patrón se hace muy compleja la admisión de transacciones atómicas.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y servicio sin estado.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

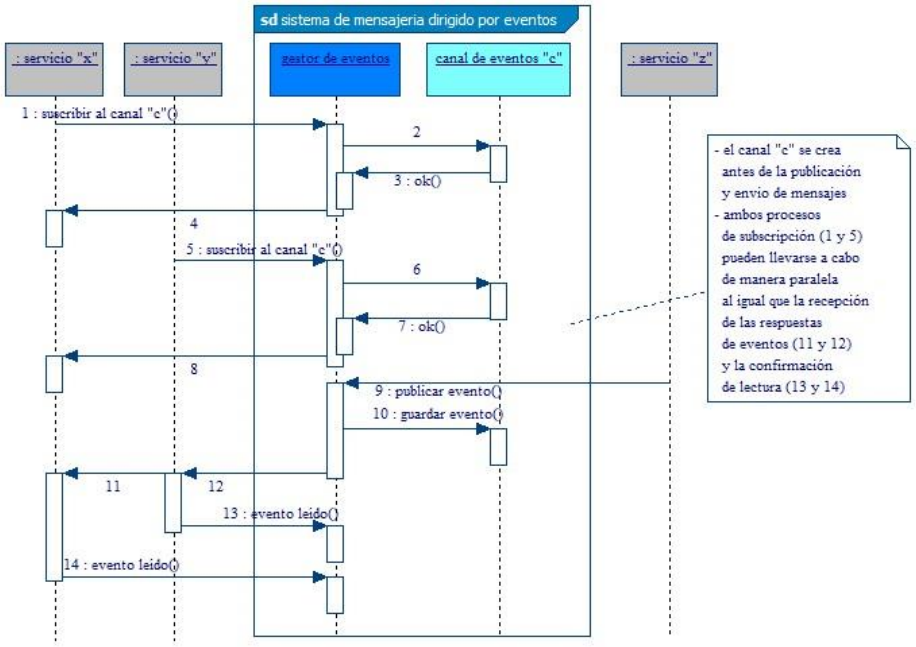


Figura 53: Los servicios (x e y) se subscriben a un canal (c) mediante un sistema gestor de eventos. Después, el servicio (z) publica un evento al canal (c) donde existen uno o más suscriptores (incluidos los servicios x e y). Por último, los consumidores leen los mensajes que se encuentran en el canal (c) y los marcan como leídos (estrategia de lectura individual en la cual los suscriptores no compiten por el mismo mensaje). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de rendimiento:

- Ya que la publicación concurrente de eventos relevantes de servicios sobre un sistema de mensajería dirigido por sucesos permite que los consumidores recuperen los datos de forma eficiente.

Aumento de confiabilidad:

- Referida al comportamiento esperado de los servicios en situaciones de escritura y lectura intensiva de eventos simultáneos.

7.2.1.2.3. Nombre: Enrutamiento intermedio de mensajes (Little, Simon).

Sinopsis: Este patrón de diseño se refiere a la instauración de lógica intermedia para la determinación dinámica de rutas de mensajes en soluciones asíncronas.

Nombre alternativo: Enrutamiento inteligente de mensajes.

Servicios afectados:

- Por funcionalidad: La lógica de enrutamiento puede considerarse parte de servicios de utilidad o de tareas dependiendo si los factores que indican los caminos se refieren al rendimiento o al contenido del mensaje, aunque afecten a todos los tipos.
- Por estilo: *SOAP* y *ReSTful.*

Problema: Cuanto más grande y compleja son las composiciones de servicios, más difícil es anticipar y diseñar de antemano todos los posibles escenarios en tiempo de ejecución, especialmente en soluciones donde existe intercambio asíncrono de mensajes.

Solución: Las rutas de los mensajes se determinan dinámicamente mediante el uso de lógica de intermedia.

Aplicación: Establecimiento de una gama variada de funcionalidad intermedia para crear rutas de mensajes basadas en el contenido o factores en tiempo de ejecución.

Efectos: La determinación dinámica de rutas de mensajes añade capas de lógica de procesamiento y, en consecuencia, existen aumento de complejidad funcional y mayor latencia operativa.

Principios soportados: Reusabilidad de servicio, acoplamiento débil y capacidad de composición.

Componentes de arquitectura involucrados: Composición de servicios.

Diagrama de arquitectura:

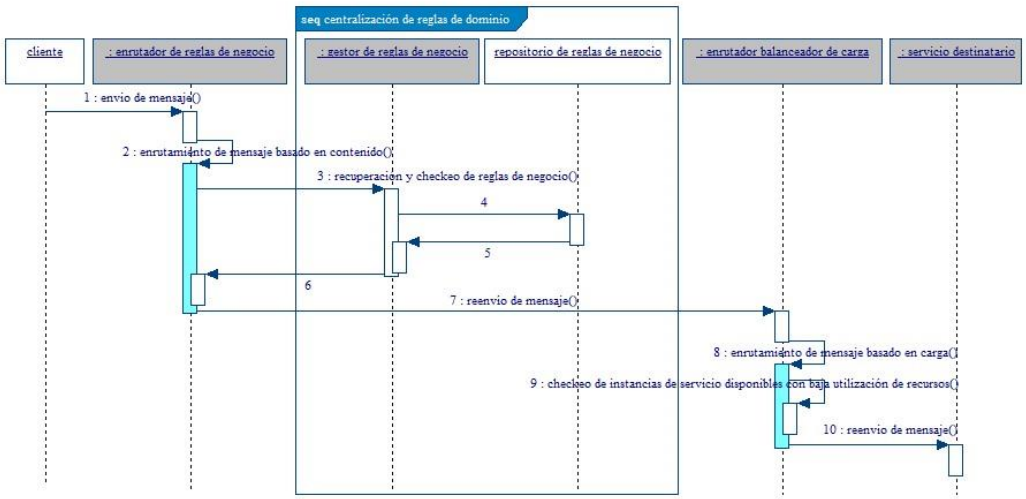


Figura 54: El mensaje pasa a través de dos módulos de enrutamiento antes de llegar a su destino. Primero, un enrutador identifica el servicio de destino según una regla de negocio en base al contenido del mensaje, como consecuencia del uso reglas de dominio centralizadas (ver sección 7.2.2.4.4). Después, otro enrutador verifica las estadísticas de uso actuales del servicio destinatario antes de decidir a qué instancia del servicio enviar el mensaje. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor rendimiento:

⊕ Referido al establecimiento de lógica de enrutamiento en base a factores de ejecución lo cual permite el balanceo adecuado de carga entre instancias de servicios.

Disminución de rendimiento:

⊖ En cuanto tiempos de respuesta ya que la existencia de enrutadores implica la presencia de lógica adicional de procesamiento que aumenta la latencia operativa.

Incremento de interoperabilidad:

⊕ Debido a enrutadores en base reglas de negocio centralizadas lo cual disminuye la redundancia de lógica e impulsa la reutilización de los servicios.

7.2.1.2.4. Nombre: Mensajes con metadatos (Erl).

Sinopsis: Este patrón de diseño se refiere al envío de mensajes de servicios con metadatos los cuales son interpretados de manera aislada a la lógica principal.

Nombre alternativo: Mensajería de metadatos.

Servicios afectados:

- Por funcionalidad: De tareas, utilidad y entidad.
- Por estilo: *SOAP* y *ReSTful.*

Problema: Dificultad de acceso a información de estado en tiempo de ejecución debido a que las conexiones entre los clientes y servicios no son persistentes.

Solución: El contenido de los mensajes se complementa con metadatos específicos de la actividad en ejecución que se comprenden y procesan por separado.

Aplicación: Este patrón requiere de tecnología que admita el análisis de encabezados o propiedades añadidas al contenido del mensaje. Por ejemplo, el uso de encabezados para transmitir información de origen como las credenciales de autenticación encriptadas (*Basic* o *Bearer*), el tipo de carácter con el cual el mensaje fue codificado (*ASCII* o *UTF-8*), el tipo de contenido del mensaje (*application/json* o *application/xml*), el identificador de una transacción de negocio (*UUID*[40]), etcétera.

Efectos: Le interpretación y procesamiento de metadatos implica añadir un procesamiento

[40] *Unique Universal ID (UUID)* es número de 128 *bits* que representa un identificador único universal dentro de un sistema de negocio. La probabilidad de repetición de este número es considerada muy baja por lo cual se suele usar como identificador transaccional del sistema. Para más información ver: <https://es.wikipedia.org/wiki/Identificador_%C3%BAnico_universal>

adicional a la lógica existente.

Principios soportados: Servicio sin estado y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor interoperabilidad:

⊕ Debido a que la información de metadatos se refiere a datos particulares de actividades en ejecución que pueden ser comprendidos de manera aislada y eficiente sin necesidad de persistir la información de estado entre los servicios por lo cual se impulsa el reúso.

7.2.1.2.5. Nombre: Mensajería confiable (Little, Simon).

Sinopsis: Este patrón de diseño señala el establecimiento de un mecanismo intermediario de confianza para garantizar la entrega de los mensajes.

Nombre alternativo: Mensajes confiables.

Servicios afectados:

- Por funcionalidad: De utilidad y afecta a todos los otros tipos de servicios.
- Por estilo: En los servicios *SOAP* mediante la implementación del estándar *WS-ReliableMessaging* (ver sección 6.4.1) y extensiones que garanticen la entrega como un repositorio que persiste los mensajes intercambiados. Los *ReSTful* por definición no, aunque si es factible en una arquitectura *SOA* de microservicios a través de una tecnología de mensajería intermedia como *Apache Kafka*® (ver secciones 7.2.1.10.8 y 7.2.1.2.1) la cual puede avalar la entrega de mensajes si se configura para tal motivo.

Problema: Imposibilidad de garantizar la comunicación entre servicios debido al uso de protocolos de intercambio de mensajes poco confiables o la existencia de un entorno subyacente que no es de fiar.

Solución: Instauración de un mecanismo intermedio de confianza para garantizar el intercambio de mensajes entre servicios del inventario empresarial.

Aplicación: Implementación de tecnología intermediaria o *middleware*, como *ESB* para los servicios clásicos (ver sección 2.5) o *Apache Kafka*® para los *ReSTful* (ver secciones 7.2.1.10.8 y 7.2.1.2.1), la cual cuenta con un repositorio de datos y un gestor de mensajería para rastrear las entregas, administrar la emisión de acuses de recibo y recuperar mensajes almacenados en condiciones de falla.

Efectos: La presencia de infraestructura intermedia de confianza para garantizar la entrega

añade una lógica de procesamiento que afecta la performance durante el intercambio de mensajes y, además, no es compatible con soluciones que admiten transacciones atómicas de servicios.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

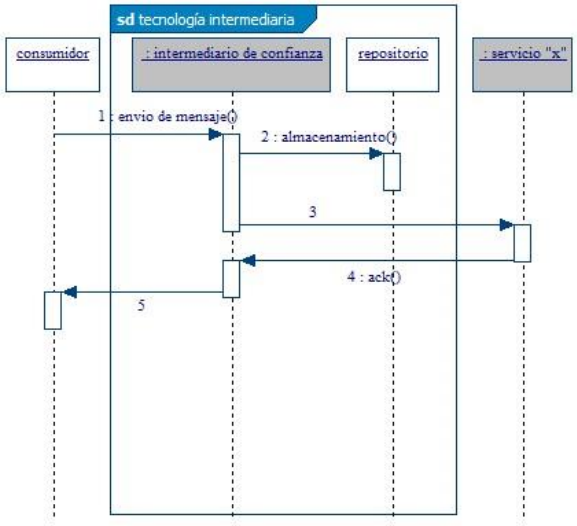


Figura 55: El consumidor envía un mensaje, mediante un mecanismo intermediario que garantiza la entrega, y posteriormente el servicio (x) transmite el acuse de recibo o *acknowledge* (*ACK*) antes de procesar la información. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Menor rendimiento:

⊖ Ya que la presencia de lógica intermedia para asegurar la transmisión de mensajes agrega latencia de procesamiento y, asimismo, puede existir redundancia de datos ante condiciones de falla.

Incremento de confiabilidad:

⊕ Respecto a la entrega de mensajes entre servicios ya que los mecanismos intermediarios de confianza garantizan el envío y la recepción de datos a través de estrategias de acuse de recibo, persistencia y recuperación de mensajes ante condiciones de falla.

7.2.1.2.6. Nombre: Agente de servicio (Erl).

Sinopsis: Este patrón de diseño indica la instauración de programas dirigidos por eventos para reducir la performance y el tamaño de las composiciones en soluciones conducidas por sucesos.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: La aplicación de este patrón reduce la frecuencia de llamadas a los servicios de utilidad y dominio con características específicas.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Composiciones genéricas que son ineficientes para resolver problemas específicos a través de llamadas a operaciones de servicio granulares.

Solución: Establecimiento de programas dirigidos por eventos que no requieren invocación explícita, lo que reduce la performance y el tamaño de las composiciones de servicios en soluciones conducidas por sucesos.

Aplicación: Implementación de agentes de servicio que son diseñados para responder automáticamente a condiciones predefinidas sin invocación explícita a través de un contrato publicado. Estos programas reaccionan ante eventos particulares, por ejemplo, para validar reglas de negocio específicas (ver sección 7.2.1.5.5) o para mediar entre la interfaz de usuario y los servicios ante condiciones peculiares (ver sección 7.2.1.10.6).

Efectos: La presencia de agentes de servicio controlados por eventos forma parte central de la arquitectura del inventario lo que implica que, una vez implementados, exista alta dependencia hacia estos programas.

Principios soportados: Reusabilidad de servicio y acoplamiento débil.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+-0))].

Aumento de rendimiento:

- ⊕ Debido a que la presencia de agentes de servicio disminuye la cantidad de llamadas a los servicios de utilidad y negocio con capacidades granulares de manera explícita, optimizando así el uso de recursos del inventario.

7.2.1.2.7. Nombre: Devolución de llamada[41] de servicio (Sin autor).

[41] La devolución de llamada o *callback* es una técnica por la cual se le ofrece al cliente que esta en cola de espera la posibilidad de retornarle la llamada en un tiempo determinado. Para más información ver: <https://es.wikipedia.org/wiki/Callback>

Sinopsis: Este patrón de diseño se refiere al envío de mensajes con dirección e información correlativa para que los servicios que los procesen puedan hacerlo de manera asincrónica devolviendo las llamadas con las respuestas en un tiempo posterior.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: No es factible comunicarse de forma sincrónica si el servicio necesita responder a una solicitud a través de la emisión de múltiples mensajes o cuando el procesamiento de mensajes requiere una gran cantidad de tiempo.

Solución: El servicio requiere que los consumidores se comuniquen con él de forma asíncrona, proporcionando una dirección a donde devolver el mensaje de respuesta.

Aplicación: Instauración de mecanismos para la generación de mensajes con información correlativa[42] junto a direcciones para devolver las llamadas cuando el procesamiento asíncrono haya terminado. Adicionalmente, si existe una infraestructura adecuada, es factible que la solución se diseñe para que se retornen funciones a ejecutarse o *callback functions* en las direcciones especificadas en vez de respuestas a mensajes en formato *XML* o *JSON* dependiendo el estilo de servicio.

Efectos: La comunicación asíncrona puede presentar problemas de confiabilidad y además requerir que la infraestructura circundante se actualice para permitir la correlación del retorno de llamadas de servicio.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

[42] La información correlativa se refiere a números *UUID* (ver sección 7.2.1.2.4) vinculados a transacciones de negocio.

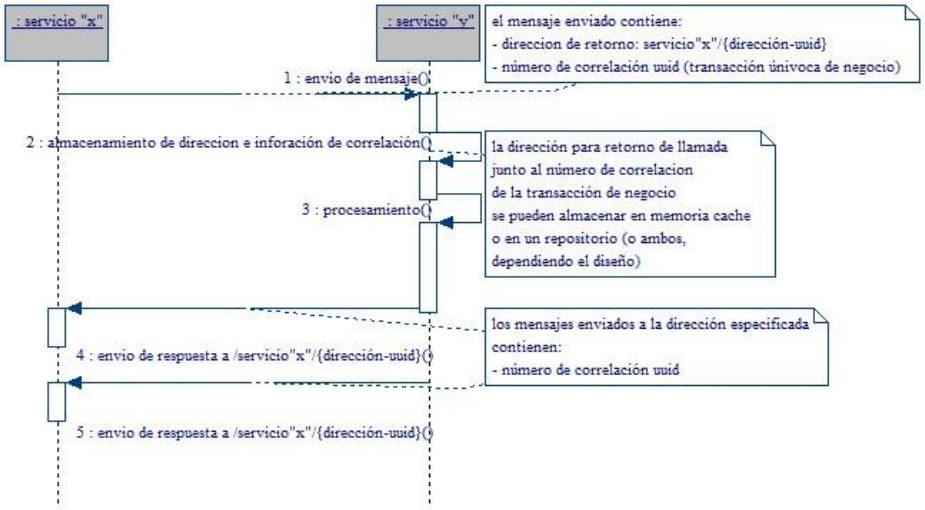


Figura 56: El servicio (x) envía un mensaje con la dirección para la devolución de llamada junto a información de correlación a otro servicio (y). Mientras el servicio final (y) está procesando el mensaje, el servicio inicial (x) está desbloqueado. Después, el último servicio (y), en algún momento posterior, envía la respuesta a la dirección indicada para la devolución de llamada. Finalmente, debido a que el servicio final (y) conserva esta dirección de devolución de llamada, puede continuar emitiendo mensajes de respuesta subsiguientes al servicio (x). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de rendimiento operativo:

- ⊕ Respecto a que los servicios que responden de manera asíncrona a las direcciones establecidas les permiten a los consumidores realizar otras tareas mientras que el procesamiento del mensaje ocurre, lo que implica un mejor uso de los recursos. Menor confiabilidad:
- ⊖ En cuanto al comportamiento esperado de servicios ya que los clientes que se encuentran en cola de espera pueden no recibir respuesta alguna cuando la solución es asíncrona.

7.2.1.2.8. Nombre: Enrutamiento de instancia de servicio (Karmarkar).

Sinopsis: Este patrón de diseño señala que el servicio brinde su identificador de instancia junto a la dirección acceso para los casos donde los consumidores necesitan acceder a información de estado de manera reiterada.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful.*

Problema: Alto acoplamiento entre consumidores y servicios en los casos donde los clientes necesitan acceder a información de estado de manera repetida debido a la implementación de lógica personalizada para tal propósito que afecta negativamente el mantenimiento y la evolución del código.

Solución: El servicio proporciona un identificador de instancia junto con su información de destino en formato estándar para evitar que los consumidores tengan que recurrir a la lógica personalizada.

Aplicación: Implementación de lógica de generación y gestión de instancias de servicios. Además, tanto los consumidores como los servicios requieren una infraestructura de mensajería común para establecer los vínculos entre instancias y los clientes, por ejemplo, mediante tecnología que admite la adherencia a una sesión o *stickiness cookie* añadida como metadato en los encabezados de los mensajes (con *AWS ELB y las stickiness cookies* se ligan los consumidores a las instancias de servicios por un período de tiempo determinado).

Efectos: El uso de este patrón de manera inadecuada puede causar una violación al principio que establece que los servicios no deben mantener información de estado para lograr una mayor interoperabilidad.

Principios soportados: Servicio sin estado, acoplamiento débil y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

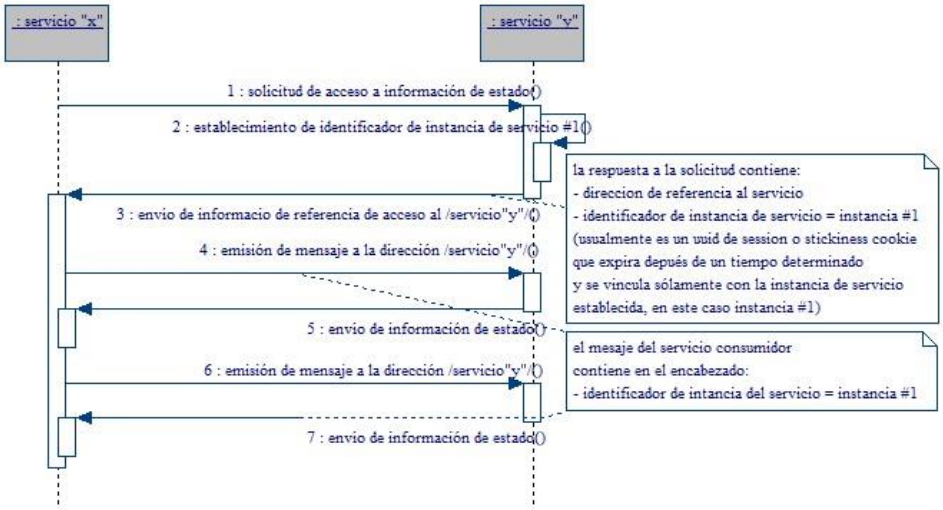


Figura 57: El servicio (x) que actúa como consumidor, emite un mensaje de solicitud otro servicio (y). Se crea la instancia #1 del servicio proveedor (y), y se genera un nuevo mensaje que contiene una referencia a la dirección de destino (que incluye el identificador de la instancia #1) el cual es enviado al servicio inicial (x). El servicio cliente (x) emite un segundo mensaje que se enruta a la instancia #1 del servicio proveedor (y) sin la necesidad de una lógica propietaria. El identificador de instancia se encuentra en el encabezado de este mensaje y, por lo tanto, se mantiene separado del cuerpo del mensaje. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que evitar que los clientes desarrollen código personalizado para acceder a información de estado de manera reiterada, mediante el uso de identificadores de instancias de servicio, disminuye el acoplamiento entre los participantes fomentando la reutilización.

7.2.1.2.9. Nombre: Mensajería de servicios (Erl).

Sinopsis: Este patrón de diseño indica el uso de tecnología de mensajería para el intercambio de mensajes entre servicios con el fin de evitar la comunicación a través de protocolos que imponen conexiones persistentes.

Nombre alternativo: -.

Servicios afectados:

➢ Por funcionalidad: La tecnología de mensajería puede ser considerado un servicio de

utilidad que afecta a todos los otros tipos (de tareas, entidad y utilidad).

- Por estilo: *SOAP* mediante el protocolo *JMS* + una tecnología de mensajería como *IBM MQ* (ver secciones 2.2 y 7.2.1.2.1) y *ReSTful* a través de sistemas de mensajería como *Apache Kafka*® (ver sección sección 7.2.1.10.8) o *RabbitMQ* (ver sección 7.2.1.2.1).

Problema: Potencial de reutilización limitado debido a servicios que dependen de protocolos tradicionales de comunicación remota que imponen la necesidad de conexiones persistentes e intercambios de datos estrechamente acoplados.

Solución: Diseño de servicios que interactúan con los consumidores mediante una tecnología de mensajería que elimina la necesidad de conexiones persistentes y reduce el acoplamiento.

Aplicación: Establecimiento de una tecnología de mensajería, como *IBM MQ* para los *SOAP* y *Apache Kafka*® o *RabbitMQ* para los *ReSTful*, la cual permite el intercambio de mensajes de servicios sin la necesidad de conexiones persistentes. Además, los servicios deben ser diseñados de manera asíncrona (posiblemente dirigidos por eventos).

Efectos: Mediante el uso de tecnología de mensajería se dificulta la admisión y el seguimiento de transacciones atómicas.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1,5 -0,5) + confiabilidad (-1) + seguridad (-1) (+ interoperabilidad (+1,5))].

Sustancial mejora de rendimiento e interoperabilidad:

⊕ Debido a que este tipo de soluciones permite liberar a los clientes de esperas ante respuestas de servicio que pueden tardar, lo cual implica un mejor uso de recursos y fomenta la reutilización ya que existe un menor acoplamiento. Pero, si no se implementa de manera adecuada, la tecnología de mensajería trae consigo problemas de calidad de servicio como:

Disminución de rendimiento:

⊖ Si se hace uso intensivo de la red envés de utilizar la estrategia de intercambio de mensajes en lotes.

Disminución de confiabilidad:

⊖ Sino se garantizan las respuestas.

Menor seguridad:

⊖ Sino se establece el uso de protocolos seguros de conexión como *TLS* (ver sección

6.4.2).

7.2.1.2.10. Nombre: Mensajes con información de estado (Karmarkar).

Sinopsis: Este patrón de diseño se refiere al envió de información de estado como parte del contenido de los mensajes envés de almacenar los todos datos en la memoria de los servicios.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas y entidad principalmente, aunque los de utilidad en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Disminución de escalabilidad y rendimiento en cuanto a servicios diseñados para mantener toda la información de estado en memoria durante el intercambio de mensajes.

Solución: El almacenamiento de información de estado se delega temporalmente a los mensajes en lugar de conservar todos los datos en la memoria de los servicios.

Aplicación: Tanto los servicios como los consumidores deban diseñarse para procesar información de estado contenida en los mensajes de intercambio.

Efectos: Este patrón no es adecuado para todos los escenarios, por ejemplo, en el caso de intercambio de información de estado incluyendo grandes volúmenes de datos.

Así mismo, existe la posibilidad de perder información de estado vital si se extravían los mensajes y no existen mecanismos que garanticen la entrega.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y capacidad de composición.

Componentes de arquitectura involucrados: Composición de servicios.

Mejora de *QoS:* Formula cualitativa = [Rendimiento (+-0) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de confiabilidad:

- ⊖ Debido a que la información de estado se puede extraviar sino existen mecanismos que garanticen la entrega de mensajes.

Aumento de interoperabilidad:

- ⊕ Ya que él envió de información de estado incluida en los mensajes disminuye la complejidad de las implementaciones de los servicios en cuanto a mantener todos los datos en memoria, lo cual mejora la escalabilidad de los servicios e impulsa la

reutilización.

7.2.1.3. Patrones de gestión de servicios:

7.2.1.3.1. Nombre: Compatible al cambio (Orchard, Riley).

Sinopsis: Este patrón de diseño indica el establecimiento de convenios de servicio flexibles que se adaptan a modificaciones requeridas evitando la invalidación de clientes existentes.

Nombre alternativo: Modificaciones compatibles.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Modificaciones en las interfaces de los servicios existentes afectan a los consumidores negativamente, inutilizando sus operaciones.

Solución: Algunas modificaciones en los convenios se adaptan al cambio sin afectar las definiciones establecidas previamente.

Aplicación: Las modificaciones en los convenios de servicio se acomodan a través de extensiones, flexibilización de restricciones o mediante la aplicación del patrón "contratos concurrentes".

Efectos: La introducción de cambios compatibles requiere esfuerzo de gestión en cuanto al versionado de las modificaciones. Adicionalmente, la flexibilización de restricciones en los convenios resulta en contratos generales con baja especialización.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor confiabilidad:

⊖ En cuanto al comportamiento funcional esperado, ya que, una reducción de especialización en las interfaces, permitiendo flexibilidad operativa, añade complejidad a la lógica del servicio individual la cual debe ajustarse a una mayor cantidad de alternativas.

Incremento de interoperabilidad:

⊕ Ya que la adaptabilidad de los contratos existentes a los cambios de negocio aumenta la cantidad de clientes potenciales fomentando el reúso de servicios.

7.2.1.3.2. Nombre: Descomposición de servicios (Erl).

Sinopsis: Este patrón diseño indica la descomposición de servicios genéricos en más granulares para mejorar la interoperabilidad.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Servicios con capacidades muy genéricas inhiben el diseño de composiciones.

Solución: Descomposición de servicios genéricos en dos o más servicios con granularidad más fina.

Aplicación: La lógica de servicio subyacente se reestructura y se establecen nuevos contratos de servicio. En el caso de clientes existentes, este patrón requiere de capacidad de mediación para conservar la integridad del convenio de servicio original.

Efectos: El aumento de servicios más granulares naturalmente conduce a diseños de composiciones más complejas y de mayor tamaño.

Principios soportados: Capacidad de composición y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

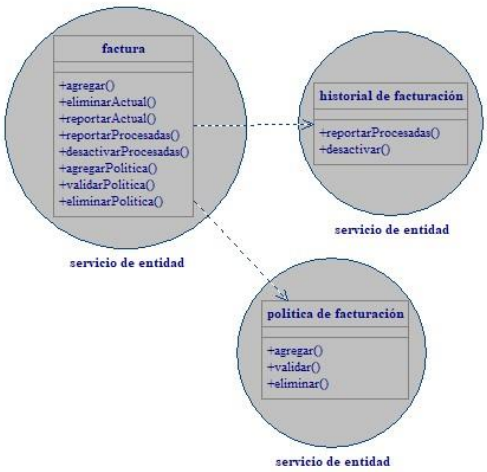


Figura 47: El servicio de entidad genérico (factura) se descompone en dos servicios de entidad más granulares (historial de facturación y política de facturación), lo que deriva en la existencia total de 3 entidades. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que la descomposición de servicios generales en servicios granularidad más fina se fomenta el diseño de composiciones, lo que resulta en mayor reutilización de componentes en la nube como recursos empresariales que pueden ser usados en múltiples proyectos.

7.2.1.3.3. Nombre: Refactorización de servicios (Sin autor).

Sinopsis: Este patrón de diseño establece la necesidad de refactorización de lógica de servicio, manteniendo la interfaz, cuando la implementación o tecnología existentes se vuelven obsoletas o inadecuadas.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La lógica o tecnología implementadas en el servicio han caducado o son impropias, pero el componente en la nube se ha afianzado demasiado para ser reemplazado.

Solución: Refactorización de lógica y/o tecnología de servicio subyacentes preservando el contrato de servicio para mantener las dependencias existentes con los consumidores.

Aplicación: Mejora o actualización gradual de lógica y tecnología de servicio que deben someterse a pruebas adicionales para garantizar que los cambios efectuados no interrumpan la operativa existente.

Efectos: El establecimiento de funcionalidad y/o tecnología nueva introduce riesgo de aplicación y gestión que suele ser mitigado con capacitación de personal.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y abstracción.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+-0))].

Incremento de rendimiento, confiabilidad y seguridad:

⊕ Respecto a la implementación del servicio subyacente mediante la mejora y/o actualización de la lógica principal y/o tecnología del componente en la nube.

7.2.1.3.4. Nombre: Notificación de finalización (Orchard, Riley).

Sinopsis: Este patrón de diseño se refiere a la notificación de fin de servicio mediante

políticas explicitadas en los convenios de servicio para que los consumidores estén al tanto cuando esto ocurra.

Nombre alternativo: Notificación de expiración.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* mediante el uso de *WS-Policy* y en los *ReSTful* se puede utilizar *OPA* (ver sección 6.4.1).

Problema: Fallas en aplicaciones de clientes que no se dieron cuenta que la versión del contrato/servicio que utilizan han caducado o fue reemplazada.

Solución: Las interfaces de los servicios se diseñan para expresar información de expiración y reemplazo que es comprendida por humanos y programas.

Aplicación: Se añaden políticas y anotaciones en los convenios las cuales son legibles para los humanos y programas clientes. Las tecnologías involucradas en la notificación de información de fin varían según el estilo de servicio, es decir, en el caso de servicios clásicos a través en archivos *WS-Policy* (en formato *XML*) mientras que para aquellos que siguen el estilo *ReST* mediante registros *OPA* (en *JSON*).

Efectos: La sintaxis y las convenciones utilizadas para expresar la información de fin deben ser entendidas por los consumidores del servicio para que esta información pueda ser utilizada de manera efectiva.

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

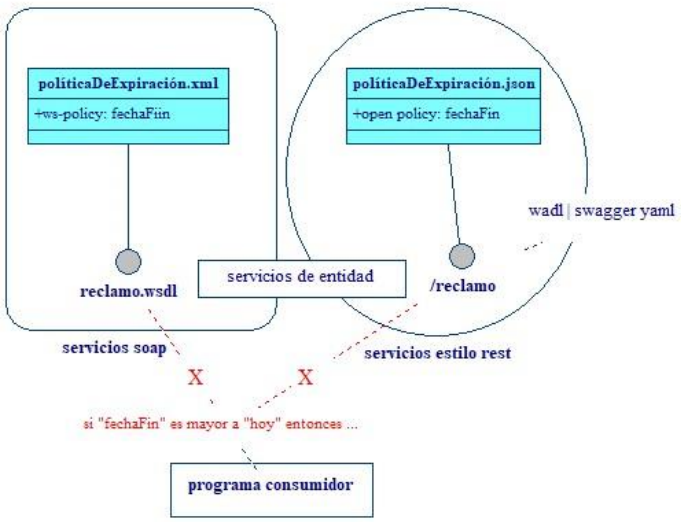


Figura 48: El convenio de servicio informa la fecha de expiración, por lo cual, el

consumidor no intenta invocarlo después que haya caducado. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Aumento de confiabilidad:

- ⊕ En cuanto al comportamiento esperado del servicio debido a que la información de expiración es de carácter relevante para que los clientes puedan anticipar las acciones a tomar cuando la operativa finalice, evitando fallas en tiempo de ejecución en sus programas.

7.2.1.3.5. Nombre: Identificación de versión (Orchard, Riley).

Sinopsis: Este patrón de diseño señala el establecimiento de versiones de servicios para que los clientes estén al tanto de cambios efectuados que puedan afectar el comportamiento esperado.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* mediante contratos versionados y los *ReSTful* en las direcciones de recursos especificadas en *Swagger* (*Open API*).

Problema: Los cambios en las interfaces de servicios existentes afectan negativamente a los clientes que no están al tanto de las modificaciones realizadas.

Solución: La información sobre las versiones compatibles puede explicitarse en los convenios y/o en las direcciones de los servicios (*URLs*).

Aplicación: En el caso de contratos de servicios clásicos, los números de versión se incorporan en los espacios de nombres definidos en el *WSDL*. En el caso de interfaces *ReSTful,* las versiones de las interfaces se especifican en las *URLs* de los recursos *Swagger*.

Efectos: La información de versionado debe explicitarse de manera adecuada para que los consumidores puedan comprenderla, por ejemplo, mediante el uso de semántica de versionado (ver sección 7.2.2.4.1).

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

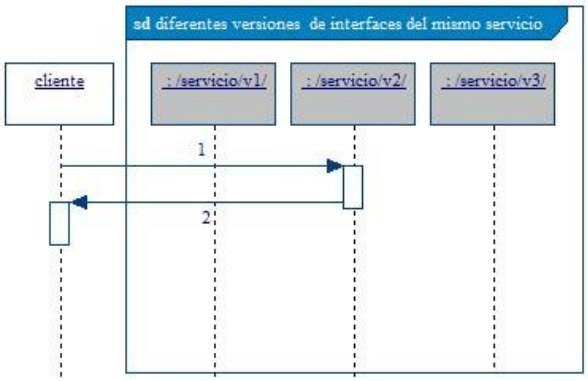


Figura 49: El cliente selecciona la versión de servicio (v2) que es compatible con su implementación (otros clientes pueden seleccionar otras versiones). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de confiabilidad:

⊕ En cuanto al comportamiento esperado del servicio ya que la información de versionado sirve para que los consumidores adapten el código de sus implementaciones evitando problemas de compatibilidad.

7.2.1.4. Patrones de transformadores:

7.2.1.4.1. Nombre: Transformación del formato de los datos (Little, Simon).

Sinopsis: Este patrón de diseño se refiere a la necesidad de transformar el formato de la información durante el intercambio de mensajes entre servicios y consumidores que se comunican a través de tipos de datos dispares.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: Primordialmente son servicios de utilidad que centralizan las trasformaciones de formato de los demás componentes en la nube. También es factible que un servicio particular, de otra capa, se relacione con un programa de aplicación específico en formato dispar (esta segunda opción es minimalista y poco escalable).
- Por estilo: *SOAP* y *ReSTful*.

Problema: Comunicación incompatible entre consumidores y servicios debido a que los primeros transmiten datos en un formato diferente al esperado por los destinatarios.

Solución: Lógica intermediaria para transformar dinámicamente el formato de la información entre consumidores y servicios con tipos de datos desiguales.

Aplicación: Se aplica esta lógica de transformación como parte del servicio interno que necesita comunicarse con un consumidor particular o mediante servicios de transformación dedicados.

Efectos: Se añade una lógica de procesamiento adicional vinculada con transformadores de formato la cual implica mayor esfuerzo de desarrollo y complejidad de gestión.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

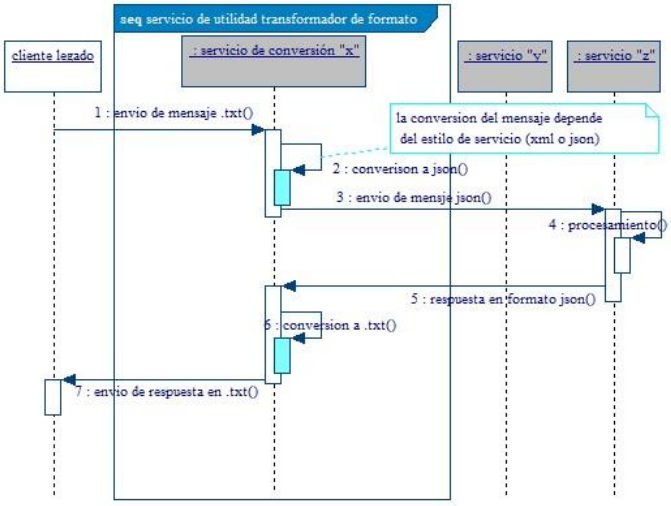


Figura 64: El servicio de conversión de formato (x) recibe los datos en texto plano (.txt) que provienen del consumidor y los transforma en *XML* o *JSON* dependiendo el estilo arquitectónico de servicios del inventario. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor rendimiento:

⊖ Ya que el uso de procesamiento intermediario para transformar los diferentes tipos de datos en tiempo de ejecución incrementa la latencia de respuesta de los servicios.

Incremento de interoperabilidad:

⊕ Debido a que la presencia de transformadores hace factible que consumidores y servicios con datos dispares se comuniquen con un bajo nivel de acoplamiento entre las diferentes tecnologías de los participantes.

7.2.1.4.2. Nombre: Transformación del modelo de datos (Erl).

Sinopsis: Este patrón de diseño señala el requisito de transformar el modelo de los datos durante el intercambio de mensajes entre servicios y consumidores que se comunican mediante convenios desiguales.

Nombre alternativo: Adaptador de mensaje.

Servicios afectados:

- Por funcionalidad: De utilidad principalmente.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Complejidad de interacción entre consumidores y servicios que intercambian mensajes mediante convenios dispares que representan los mismos datos.

Solución: Funcionalidad intermediaria para adaptar los modelos entre consumidores y servicios con convenios incompatibles.

Aplicación: Instauración de lógica de mapeo, como parte de los servicios participantes o mediante una plataforma intermediaria, que interpreta los datos compatibles con el modelo del origen y los convierte dinámicamente en otro modelo de datos acorde con el destinatario.

Efectos: El uso excesivo de este patrón reduce el potencial de recomposición de los servicios del inventario. Además, implica mayor esfuerzo de desarrollo y complejidad de gestión la cual puede ser evitada con el diseño apropiado de los convenios.

Principios soportados: Contrato de servicio estándar, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

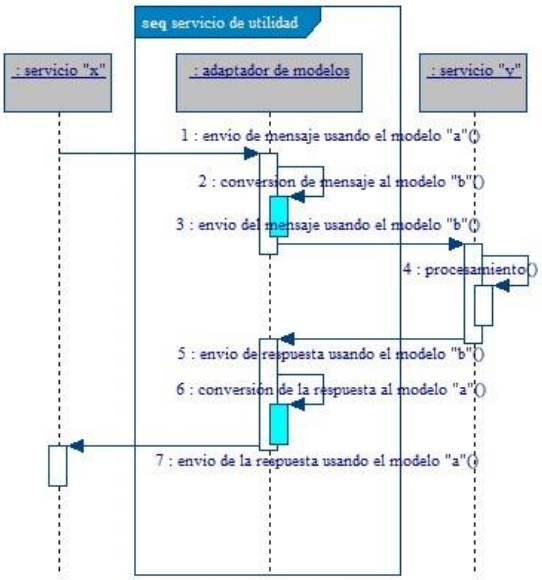


Figura 65: Los servicios (x e y) se comunican entre sí a través del servicio transformador de mensajes el cual adapta los convenios dispares (a y b) entre los participantes permitiendo la comunicación. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

Disminución de rendimiento:

⊖ Ya que el uso de procesamiento intermediario para transformar los modelos desiguales en tiempo de ejecución incrementa los tiempos de respuesta de los servicios.

Aumento de interoperabilidad:

⊕ Debido a que la presencia del transformador intermediario hace factible que consumidores y servicios con convenios dispares se comuniquen lo cual fomenta la reutilización.

Menor interoperabilidad:

⊖ Si este patrón se usa de manera excesiva ya que se reduce el potencial de recomposición de los servicios y, en consecuencia, se complejiza la lógica de los transformadores.

7.2.1.4.3. Nombre: Puente entre protocolos (Little, Simon).

Sinopsis: Este patrón de diseño indica la presencia de un servicio intermediario que hace de puente entre consumidores que utilizan diferentes protocolos para comunicarse con los servicios del inventario.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Los consumidores que utilizan diferentes protocolos de comunicación o versiones desiguales de la misma tecnología de intercambio de datos no pueden conectarse con los servicios.

Solución: Instauración de lógica intermediaria que admite la comunicación entre diferentes protocolos de intercambio de datos mediante la conversión dinámica de un protocolo a otro en tiempo de ejecución.

Aplicación: En lugar de conectarse directamente entre sí, los consumidores se conectan al intermediario que proporciona una lógica de puente para convertir los protocolos dinámicamente.

Efectos: Las tecnologías de puente dificultan el establecimiento de funcionalidad transaccional para garantizar el comportamiento de los servicios ya que los identificadores pueden ser extraviados por los intermediarios. Además, este patrón añade lógica de procesamiento para la conversión que afecta negativamente la performance de los servicios que participan de comunicaciones multi protocolo.

Principios soportados: Contrato de servicio estándar, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

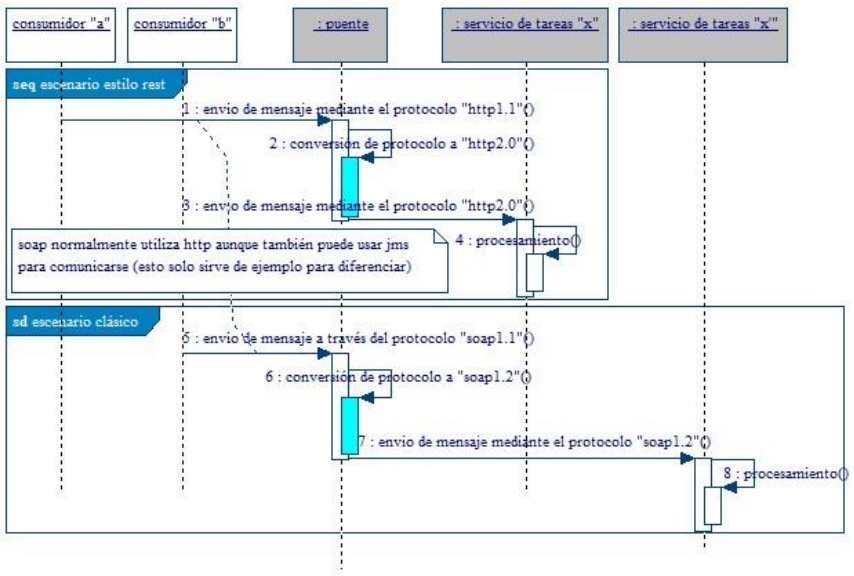


Figura 66: Los consumidores (a y b) se comunican con el servicio de tareas (x/x') mediante un servicio intermediario que convierte los protocolos de comunicación en tiempo de ejecución para que los mensajes sean procesados en la nube. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1,5) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Importante disminución de rendimiento:

⊖ Debido a que las tecnologías que hacen de puente entre protocolos de comunicación dispares poseen un gran impacto negativo en la performance general del sistema.

Incremento de interoperabilidad:

⊕ Referido a que la instauración de un servicio intermediario que permite la comunicación con consumidores a través de protocolos desiguales aumenta la reutilización.

7.2.1.5. Patrones de diseño de contratos:

7.2.1.5.1. Nombre: Contratos concurrentes (Erl).

Sinopsis: Este patrón de diseño establece la creación de múltiples convenios para el mismo servicio con el fin de atender las necesidades de distintos clientes.

Nombre alternativo: Interfaces concurrentes.

Servicios afectados:

- Por funcionalidad: De tareas.
- Por estilo: Primordialmente en los *SOAP* y en los *ReSTful* en menor medida aplicado a múltiples interfaces.

Problema: El contrato del servicio puede no ser adecuado o aplicable a todos los posibles clientes.

Solución: Creación de múltiples contratos para el mismo servicio, cada uno dirigido a un tipo específico de cliente.

Aplicación: Este patrón se debe implementar en conjunción con el patrón "fachada de servicio" (ver sección 7.2.1.10.5) para lograr la adecuación a los diferentes convenios con un acoplamiento débil a recursos particulares de los distintos consumidores.

Efectos: Mayor complejidad de gestión ya que cada nuevo convenio añade efectivamente un nuevo *endpoint* de servicio.

Principios soportados: Contratos de servicio estándar, reusabilidad y servicios sin estado.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

Incremento de interoperabilidad:

⊕ En cuanto a que la aplicación de múltiples convenios o interfaces, en un mismo servicio, que se adaptan a las diferentes necesidades de los clientes fomenta la reusabilidad.

Disminución de interoperabilidad:

⊖ Respecto a menor robustez en cuanto al grado en que el servicio *Web* puede funcionar correctamente, en presencia de ingresos no válidos, ya que la lógica para soportar los diferentes requisitos de los consumidores, en el mismo componente, es más compleja incluso en combinación con el patrón "fachada de servicio".

7.2.1.5.2. Nombre: Centralización en contratos (Erl).

Sinopsis: Este patrón de diseño indica el uso de contratos de servicios para acceder a recursos de la empresa con el fin de evitar el acoplamiento a implementaciones dispares por parte de los consumidores.

Nombre alternativo: Centralización en convenios.

Servicios afectados:

- Por funcionalidad: De tareas, utilidad y entidad.
- Por estilo: Principalmente en los *SOAP* mediante *WSDL* y en los *ReSTful* a través de *WADL* o especificaciones *Swagger*.

Problema: Redundancia de lógica en cuanto a la existencia de diferentes implementaciones que acceden a recursos similares de un mismo componente en la nube. El inconveniente se evidencia con la presencia de clientes que utilizan distintos *endpoints* del mismo servicio para acceder a recursos análogos, lo que resulta en mayor dificultad de adaptación a los cambios de requerimientos.

Solución: El acceso a la lógica de servicio está limitado al contrato de servicio, lo que obliga a los consumidores a evitar el acoplamiento directo a las implementaciones subyacentes.

Aplicación: Instauración de contratos de servicio, *WSDL o Swagger Spec* dependiendo del estilo (ver sección 2.2), que poseen información esencial, encontrándose limitados solamente a lo que se publica, lo cual hace que se fomente la abstracción ocultando los detalles de la implementación subyacente.

Efectos: El acceso a funcionalidad y recursos de los componentes en la nube mediante convenios requiere un esfuerzo continuo de gestión y normalización de los servicios empresariales.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y abstracción.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

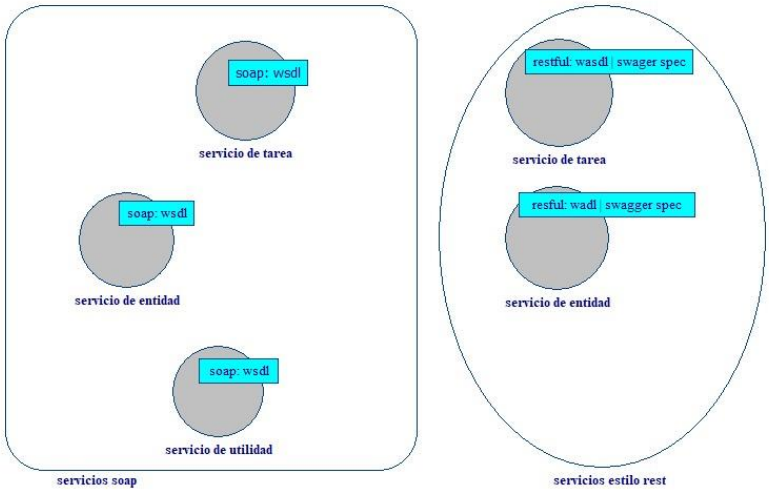


Figura 40: Es de suma relevancia el diseño de contratos ya que éstos conforman el centro de una arquitectura orientada a servicios. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+2))].

Disminución de rendimiento:

⊖ En comparación al uso de tecnologías de llamadas remotas a procesos o *remote procedure calls* (*RPC*) como *Apache Thrift*[43] debido a que la instauración de contratos requiere el uso de adaptadores entre los datos del convenio y la funcionalidad principal del servicio para evitar acoplamiento (ver patrón "fachada de servicio" en sección 7.2.1.10.5).

Aumento de interoperabilidad:

⊕ Ya que el establecimiento de convenios entre clientes y proveedores impulsa el reúso de servicios *Web* evitando el acoplamiento a implementaciones internas.

⊕ Referido a que existe una mayor escalabilidad horizontal (más instancias) en cuanto

[43] *Apache Thrift* es una tecnología que permite definir tipos de datos e interfaces de servicio en un archivo de definición simple, con el cual el compilador genera el código que se usará para crear fácilmente clientes y servidores *RPC* que se comuniquen sin problemas entre diferentes lenguajes de programación (en lugar de escribir código repetitivo para serializar y transportar objetos e invocar métodos remotos). Ver más información en: <https://thrift.apache.org/>

a la capacidad que poseen los proveedores de evolucionar la lógica de sus servicios mediante la aplicación del principio de abstracción ocultando detalles de las implementaciones subyacentes.

7.2.1.5.3. Nombre: Desnormalización de contratos (Erl).

Sinopsis: Este patrón de diseño indica la desnormalización medida del contrato de servicio para proveer operaciones con cierta redundancia funcional, lo cual mejora el rendimiento operativo.

Nombre alternativo: Desnormalización de convenios.

Servicios afectados:

- Por funcionalidad: De tareas, utilidad y entidad.
- Por estilo: Los *SOAP* mediante la aplicación de este patrón y los *ReSTful* cumpliendo con *AMM3* (ver sección 2.4.1).

Problema: Contratos de servicios estrictamente normalizados que imponen demandas innecesarias de rendimiento y lógica en los consumidores.

Solución: Contratos de servicio que poseen un grado medido de desnormalización, permitiendo expresar funciones con algo de redundancia para diferentes tipos de clientes.

Aplicación: Convenios de servicio que se extienden cuidadosamente con características adicionales que proporcionan variaciones operativas de funcionalidad principal.

Efectos: Los contratos que usan de manera desmedida este patrón aumentan dramáticamente su tamaño, haciendo que sean difícil de interpretar y gestionar. También, la redundancia excesiva rompe con la cohesión funcional de las operaciones y reduce la reutilización.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (-1))].

Incremento de rendimiento:

⊕ Referido a la existencia de operaciones extendidas que cumplen con necesidades específicas de diferentes tipos de clientes.

Menor interoperabilidad:

⊖ En cuanto al uso exacerbado de este patrón lo cual fomenta la redundancia de lógica y disminuye el reúso.

7.2.1.5.4. Nombre: Desacoplamiento de contrato (Erl).

Sinopsis: Este patrón de diseño establece que necesidad de desacoplamiento físico entre los contratos de servicios y sus implementaciones para evitar dependencias indeseadas entre el entorno subyacente y los requisitos de negocio.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, utilidad y entidad.
- Por estilo: *SOAP* y los *ReSTful* en sus interfaces.

Problema: Contratos de servicios altamente acoplados a sus implementaciones, lo cual reduce la eficacia de los servicios como recursos empresariales ya que existe menor reúso.

Solución: Desacoplamiento físico entre los convenios y sus implementaciones.

Aplicación: Las interfaces técnicas de los servicios se encuentran físicamente aisladas y sujetas a los principios de diseño relevantes de una arquitectura orientada a servicios.

Efectos: Las funcionalidades de los servicios están limitadas al conjunto de operaciones expuestas en el contrato desacoplado.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Ya que, al desacoplar los contratos de servicio, las implementaciones evolucionan sin afectar directamente a los consumidores del servicio, lo cual aumenta la cantidad de clientes potenciales (y la reutilización correspondiente).

7.2.1.5.5. Nombre: Abstracción de validaciones (Erl).

Sinopsis: Este patrón de diseño se refiere al requisito de disminuir las validaciones granulares en los convenios de servicio para hacerlos más reusables y perdurables.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, utilidad y entidad.
- Por estilo: En el caso de los servicios *SOAP* mediante la reducción de las restricciones escritas en los documentos *WSDL* o en los esquemas *XSD*. Por otro lado, en el caso de los servicios *ReSTful* en la disminución de las verificaciones especificadas en *Swagger* o *WADL* a través del uso de *YAML* o *JSON Schema* (ver sección 2.2).

Problema: Contratos de servicio con validaciones específicas que se invalidan cuando cambian las reglas de negocio.

Solución: Las reglas de validación granulares se abstraen de los contratos de servicio, lo que reduce los detalles especificados y aumenta la longevidad potencial del convenio.

Aplicación: Las reglas de validación granulares se abstraen en la lógica del servicio subyacente, a un servicio diferente o a un agente de servicio (ver patrón "verificación de mensajes" en sección 7.2.1.1.2).

Efectos: La reducción excesiva de verificaciones en los convenios complica la normalización de los esquemas del inventario de servicios.

Principios soportados: Contrato de servicio estándar, abstracción y acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

- En cuanto a que la disminución de verificaciones particulares en los contratos impulsa el reúso de los servicios y también el incremento de consumidores potenciales.

7.2.1.6. Patrones de herencia encapsulada:

7.2.1.6.1. Nombre: Mediador de archivos (Roy).

Sinopsis: Este patrón de diseño señala la instauración de una lógica intermediaria entre los servicios y los sistemas antiguos basados en archivos reduciendo el acoplamiento.

Nombre alternativo: Puerta de enlace de archivos.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Sistemas antiguos que procesan y almacenan sus datos en archivos de texto plano incapaces de hacer y recibir llamadas de servicios.

Solución: Establecimiento de una lógica de procesamiento de archivos de dos vías que hace de intermediaria entre los sistemas heredados y los servicios.

Aplicación: En cuanto a los datos entrantes, la lógica de procesamiento intermedia recibe los archivos en texto plano y posteriormente realiza las transformaciones de modelo y formato de datos. Para los datos salientes, esta lógica intercepta la información producida por los servicios y los empaqueta (con posible transformación) en archivos nuevos o

existentes para el consumo del sistema heredado. En ambos casos se deben utilizar mecanismos performantes para la lectura y escritura de archivos de texto plano, como el procesamiento en lotes, debido a que las operaciones intensivas de *I/O*[44] en disco son costosas en términos de uso de recursos y tiempo operativo.

Efectos: Complejidad de implementación y gestión en cuanto a la lógica de procesamiento bidireccional de archivos debido a que aumenta la dificultad operativa y requiere una administración especializada. También, dependiendo de la estrategia de lectura y escritura seleccionada, este patrón puede no ser adecuado para los casos donde los servicios y el sistema heredados requieran respuestas inmediatas.

Principios soportados: Acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

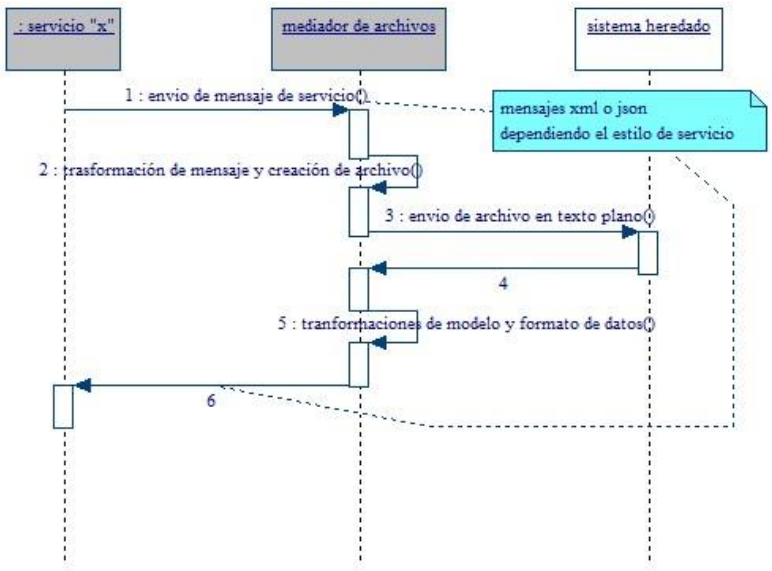


Figura 41: La lógica intermediaria actúa como mediador entre un servicio y un sistema heredado basado en archivos de texto plano. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

- ⊖ Si la lectura y escritura de archivos en disco no se realiza con mecanismos adecuados de procesamiento intensivo, como *Java NIO*[45] mediante el tratamiento de información

[44] *Input/Output (I/O)* se refiere a la lectura y escritura de datos que se encuentran en el disco físico de un sistema. Ver más información en: <https://scoutapm.com/blog/understanding-disk-i-o-when-should-you-be-worried>

[45] *Java Non-blocking Input Output (NIO)* es una colección de *APIS* en *Java,* con estrategias no

en lotes para disminuir *I/O*.

Aumento de interoperabilidad:

⊕ En cuanto a que la lógica intermediaria bidireccional permite interacción con bajo acoplamiento entre los servicios de la empresa y los sistemas heredados, lo cual aumenta el reúso.

7.2.1.6.2. Nombre: Paquete heredado (Roy).

Sinopsis: Este patrón de diseño indica el reemplazo o refactorización de servicios heredados que presentan convenios acoplados a implementaciones subyacentes mediante servicios con convenios estándar que hacen de empaquetadores de las *APIs* legadas.

Nombre alternativo: Envoltorio antiguo.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Acoplamiento elevando entre los clientes y las implementaciones de servicios que encapsulan lógica heredada debido a que estos últimos poseen interfaces poco normalizadas dependientes de requisitos tecnológicos del entorno subyacente.

Solución: Reemplazo o refactorización de *APIs* que proveen lógica legada mediante servicios con contratos estándar que encapsulan la lógica necesaria eliminando detalles técnicos heredados del contexto subyacente.

Aplicación: Desarrollo de un servicio con contrato especializado y una lógica apropiada para representar la interfaz heredada.

Efectos: La introducción de un servicio adicional añade una capa de procesamiento.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y abstracción.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

bloqueantes, para las operaciones de lectura y escritura intensivas. Para más información ver: <https://dzone.com/articles/java-nio-vs-io>

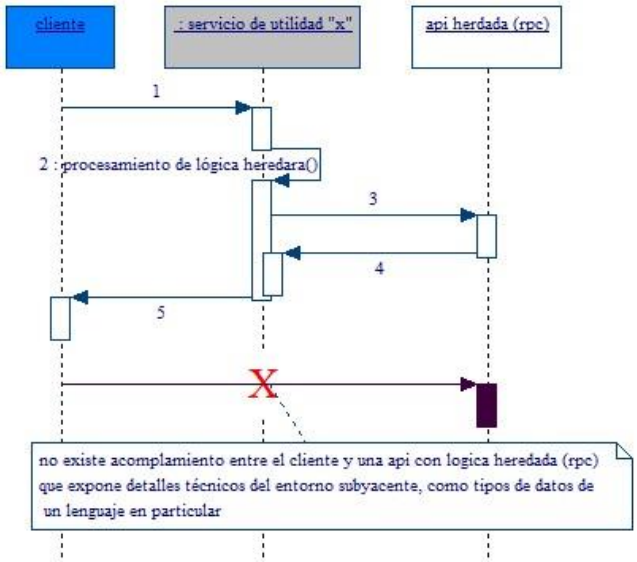


Figura 42: El servicio de utilidad (x) con un contrato estándar que procesa lógica heredada hacia una *API* de un sistema legado (*RPC*). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor performance:

⊖ En comparación a la interrelación directa entre las *API* de un sistema legado y los clientes ya que se añade una lógica de procesamiento adicional.

Incremento de interoperabilidad:

⊕ En cuanto a que la presencia de un servicio añadido que empaqueta lógica de una *API* de sistema heredado reduce el acoplamiento entre las implementaciones legadas y los clientes, lo que resulta en mayor autonomía y promueve la reutilización.

7.2.1.6.3. Nombre: Servicio multicanal (Roy).

Sinopsis: Este patrón de diseño se refiere a la instauración de un servicio multicanal que hace de intermediario entre clientes y diferentes canales de comunicación vinculados a sistemas legados.

Nombre alternativo: Interfaz multicanal.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Silos de aplicaciones complejas de gestionar en cuanto múltiples sistemas

heredados, hechos a medida sobre la base de canales de comunicación específicos (teléfonos, escritorio, kioscos, etcétera).

Solución: Diseño de un servicio intermediario que provee un convenio estándar a los clientes y encapsula la lógica específica de múltiples canales de comunicación de sistemas heredados.

Aplicación: Establecimiento de un servicio multicanal con infraestructura adecuada debido a que brindar soporte a múltiples tecnologías de comunicación requiere de una lógica compleja de procesamiento y coordinación para el intercambio de datos con los diferentes sistemas heredados.

Efectos: La lógica de procesamiento multicanal establecida por este patrón introduce la necesidad de actualizaciones de infraestructura con capacidad de orquestación y, además, puede convertirse en un cuello de botella.

Principios soportados: Acoplamiento débil y reusabilidad.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

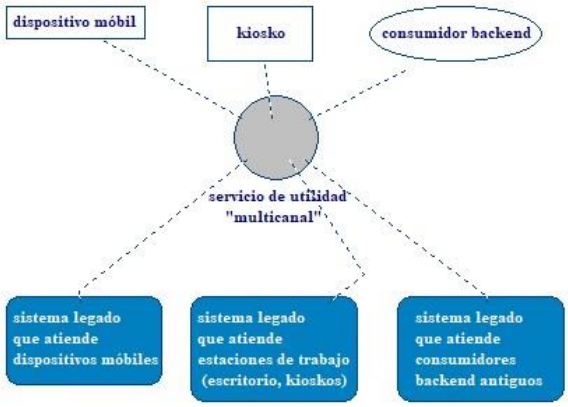


Figura 43: El servicio multicanal es el punto de contacto entre los sistemas legados y diferentes aplicaciones clientes. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor rendimiento:

⊖ En cuanto a que el servicio multicanal necesita de infraestructura adecuada para no convertirse en un cuello de botella debido a la existencia de lógica de orquestación y procesamiento de mensajes.

Aumento de interoperabilidad:

⊕ Respecto a que la presencia de un servicio intermediario que encapsula la lógica de

múltiples canales de comunicación impulsa el reúso, aumenta la cantidad de clientes potenciales y disminuye el acoplamiento entre los consumidores y las tecnologías de transmisión de datos.

7.2.1.7. Patrones de capacidades de servicios:

7.2.1.7.1. Nombre: Capacidad de descomposición (Erl).

Sinopsis: Este patrón de diseño se refiere al modelado de servicios con funciones granulares lo cual simplifica la descomposición en el caso de nuevas abstracciones de servicios.

Nombre alternativo: Descomposición de funciones.

Servicios afectados:

- Por funcionalidad: De entidad principalmente, aunque también los de tareas y utilidad en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Evolución de servicios limitada en cuanto a la complejidad de descomposición de capacidades, posterior a las implementaciones y que requieren deconstrucción de lógica, debido a la presencia de convenios con funcionalidad genérica.

Solución: Los servicios propensos a disgregaciones futuras se definen con capacidades granulares que facilitan su descomposición.

Aplicación: Llevar a cabo un modelado de servicios adicional para definir operaciones granulares y más fácilmente distribuidas. También, cuando la disgregación ocurra, el servicio base puede preservar todas sus operaciones haciendo de intermediario entre las abstracciones nuevas y los consumidores legados para evitar la disrupción inmediata.

Efectos: Hasta que el servicio finalmente se descomponga, puede ser representado por un contrato inflado de funciones especializadas que se mantienen en él (sin uso). Además, el uso excesivo de este patrón contradice lo expresado en *AMM3* (ver sección 2.4.1).

Principios soportados: Contrato de servicio estándar y abstracción.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

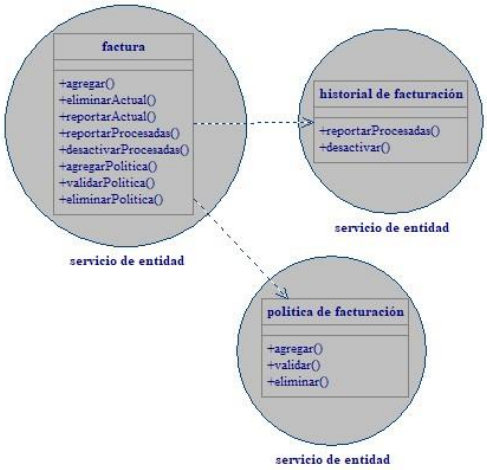


Figura 44: El servicio de entidad (factura) se modela con funcionalidad granular lo cual facilita la descomposición en servicios nuevos (historial de facturación y política de facturación) cuando sea necesario. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

⊕ Respecto a que el modelado de servicios con funciones granulares facilita la descomposición de lógica en abstracciones futuras.

7.2.1.7.2. Nombre: Capacidad distributiva (Erl).

Sinopsis: Este patrón de diseño indica la distribución de lógica del servicio particular, en casos de requerimientos de procesamiento específicos, para no afectar negativamente la integridad del contexto de servicio reduciendo la interoperabilidad.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De entidad primordialmente.
- Por estilo: *SOAP* principalmente ya que los *ReSTful* se abstraen de forma simplificada, en lo posible, sin tantos niveles de implementación interna.

Problema: Requisitos de procesamiento únicos que no pueden ser atenidos por la implementación del servicio por defecto ni tampoco mediante el desacople de capacidades que comprometen la integridad del contexto del servicio.

Solución: Distribución de la lógica del servicio subyacente, lo que permite que la lógica de implementación para una capacidad con requisitos de procesamiento únicos se separe físicamente de la funcionalidad principal, mientras continúa representada por el mismo

contrato de servicio.

Aplicación: Separación de la lógica procesamiento específico a una ubicación física diferente de la implementación principal. Adiciónamele, se utiliza el patrón "fachada de servicio" para actuar de intermediario entre la funcionalidad diferenciada (requiere comunicación remota) y la lógica principal.

Efectos: La distribución de funcionalidad específica requiere la presencia de procesamiento intermedio que agrega latencia y complejidad operativa.

Principios soportados: Contrato de servicio estándar y autonomía.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

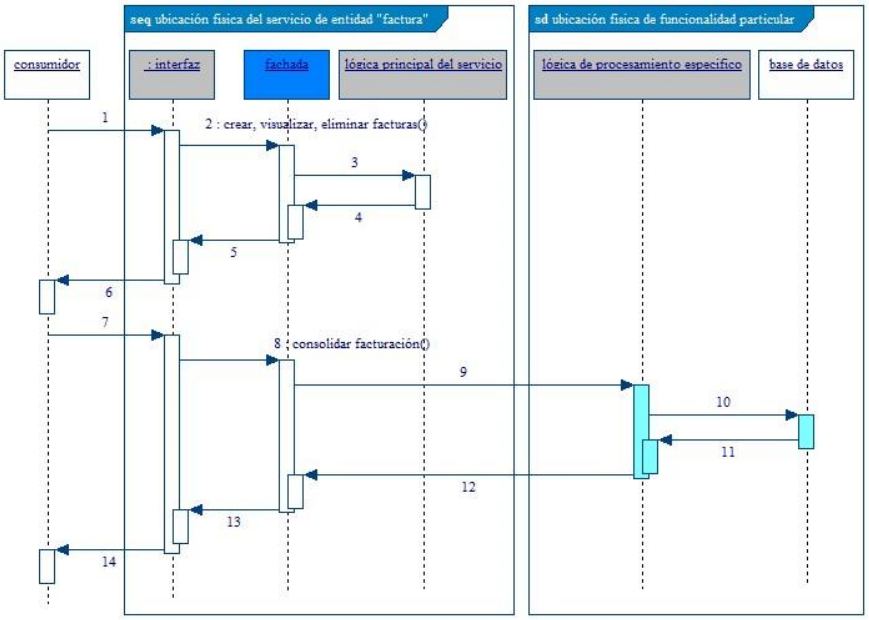


Figura 45: La lógica con capacidad de procesamiento particular (consolidar facturación) se reubica en un entorno físico separado. El componente "fachada de servicio" interactúa con la funcionalidad específica en nombre del convenio de la entidad (factura). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

⊖ Referido a que separar la lógica de procesamiento específica en una ubicación física diferente agrega latencia operativa.

Mayor interoperabilidad:

⊕ Respecto a que ejecutar funcionalidad aislada de las implementaciones principales

provee una mayor autonomía del contexto subyacente en casos independientes.

7.2.1.7.3. Nombre: Capacidad de mediar (Erl).

Sinopsis: Este patrón de diseño señala la conservación de la interfaz base y el establecimiento de lógica de mediación, en casos de descomposición de capacidades en nuevos servicios, para evitar afectar a los consumidores actuales.

Nombre alternativo: Capacidad de conciliar.

Servicios afectados:

- Por funcionalidad: De tareas y entidad con mayor frecuencia que los de utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La descomposición de las funcionalidades en los convenios causan disrupción de servicio en los clientes existentes.

Solución: Preservación de todas las operaciones del convenio original e instauración de una lógica de servicio que hace de mediador entre las abstracciones nuevas y los consumidores existentes para evitar la interrupción.

Aplicación: La fachada de servicio debe introducirse para retransmitir solicitudes y respuestas entre el mediador y las capacidades recientemente ubicadas.

Efectos: La solución práctica proporcionada por este patrón da como resultado una desnormalización del servicio medida.

Principios soportados: Acoplamiento débil.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

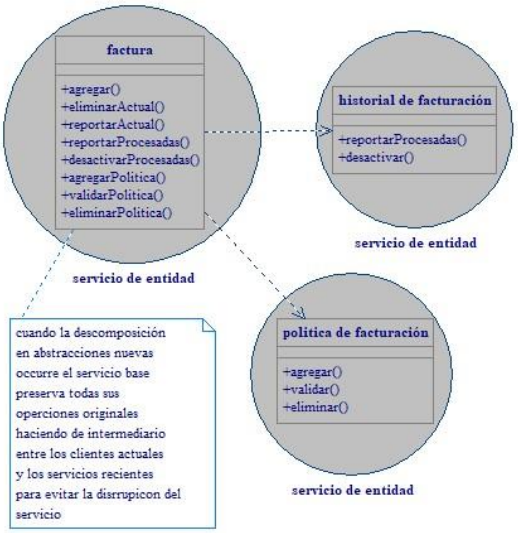


Figura 46: El servicio base preserva su convenio original (factura) y hace de mediador

entre los clientes existentes y las abstracciones nuevas (historial de facturación y política de facturación). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

⊕ Debido a que la existencia de un mediador entre los clientes existentes y las nuevas atracciones hace que el recurso empresarial base continúe siendo utilizado e impulsa el reúso especializado de los nuevos componentes para casos específicos.

7.2.1.7.4. Nombre: Capacidad agnóstica al contexto (Erl).

Sinopsis: Este patrón de diseño se refiere a la identificación de las capacidades de los servicios para abordar problemas usuales independientes del contexto subyacente, lo cual promueve la reusabilidad.

Nombre alternativo: Capacidad agnóstica + contexto agnóstico.

Servicios afectados:

- Por funcionalidad: De tareas y entidad principalmente, aunque también los de utilidad, pero en menor medida, ya que en ciertos casos es recomendable que estos últimos resuelvan temas particulares del entorno.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Las capacidades de los servicios derivadas de situaciones específicas dependientes del contexto, como la tecnología y plataforma subyacentes, pueden no ser útiles para múltiples consumidores de servicios, lo que reduce el potencial de reutilización de servicios independientes.

Solución. Identificación de lógica de servicio habitual agnóstica al contexto, que sirve para abordar problemas comunes independientes del ambiente subyacente.

Aplicación: Las capacidades agnósticas de los servicios se definen y se refinan iterativamente a través de procesos de análisis y modelado.

Efectos: La declaración de capacidades agnósticas de los servicios requiere esfuerzo de análisis y diseño.

Principios soportados: Contrato de servicio estándar, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de rendimiento:

⊕ Referido a composiciones bien definidas, conformadas por servicios de utilidad que resuelven temas del ambiente evitando la disminución de performance asociada a lógica repetida.

Mayor de interoperabilidad:

⊕ Debido a que el reconocimiento de servicios con funcionalidad agnóstica al entorno subyacente fomenta la reusabilidad de éstos y las composiciones.

7.2.1.7.5. Nombre: Descomposición funcional (Erl).

Sinopsis: Este parón de diseño indica la descomposición de un problema de dominio empresarial considerable en unidades lógicas más pequeñas vinculadas entre sí, tal vez en forma de servicios.

Nombre alternativo: Descomposición de lógica.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Una aplicación de *software* autocontenida, con restricciones de gestión y reutilización, como resultado del desarrollo de una cantidad de lógica considerable para resolver un problema empresarial complejo y de gran envergadura.

Solución. Descomposición de un problema de negocio amplio en grupo de problemas más pequeños y relacionados, lo que permite que la solución funcional requerida también se divida en un conjunto correspondiente de unidades lógicas.

Aplicación: Establecimiento de un proceso de análisis orientado a servicios para deconstruir un problema considerable en unidades pequeñas.

Efectos: Complejidad de diseño y gestión asociada a la responsabilidad de mantener múltiples soluciones reducidas conectadas entre sí.

Principios soportados: Capacidad de composición.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

⊕ Debido a que la descomposición funcional, si se hace de manera consistente, promueve las composiciones y el reúso de servicios.

7.2.1.7.6. Nombre: Capacidad dependiente del contexto (Erl).

Sinopsis: Este patrón de diseño señala la identificación de lógica dependiente del contexto que es asignada a servicios específicos para así evitar disminución de reusabilidad de componentes agnósticos.

Nombre alternativo: Contexto no agnóstico.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Capacidad de composiciones de servicios inhibida debido a la existencia de lógica dependiente del contexto que no es orientada a servicios, como la tecnología y plataforma subyacentes.

Solución: Identificación de servicios particulares del inventario que encapsulan funcionalidad dependiente del contexto, como un servicio para gestionar base de datos relacionales o *Relational Database Service* (*RDS*).

Aplicación: Definición de contextos de servicios individuales para propósitos específicos, como la administración de una base de datos particular o una tecnología de balanceo de carga de trabajo para cumplir con características propias de un dominio.

Efectos: Se esperar que los servicios dependientes del contexto cumplan con los principios de una arquitectura orientada a servicios, aunque su potencial de reutilización sea reducido.

Principios soportados: Contrato de servicio estándar y capacidad de composición.

Componentes de arquitectura involucrados: Servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de rendimiento:

⊕ En cuanto a que los servicios dependientes del contexto pueden resolver problemas específicos de manera especializada, sin afectar las composiciones de servicios agnósticos.

7.2.1.7.7. Nombre: Encapsulación de servicio (Erl).

Sinopsis: Este patrón de diseño se refiere al establecimiento de servicios que encapsulan lógica que puede ser reutilizada en diferencies soluciones empresariales.

Nombre alternativo: Servicio encapsulado.

Servicios afectados:

- Por funcionalidad: De tareas y entidad principalmente, aunque también los de utilidad en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El diseño de una lógica específica, ligada a un ambiente particular, posee un potencial restringido de reutilización en otras partes de la compañía.

Solución: Encapsulación de lógica de solución en servicios que se posicionan como recursos empresariales que pueden utilizarse en diferentes soluciones del dominio.

Aplicación: Identificación de lógica de solución adecuada que es encapsulada en servicios del inventario empresarial.

Efectos: La funcionalidad encapsulada en los servicios está sujeta a consideraciones de diseño y gestión adicionales.

Principios soportados: Reusabilidad de servicios.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

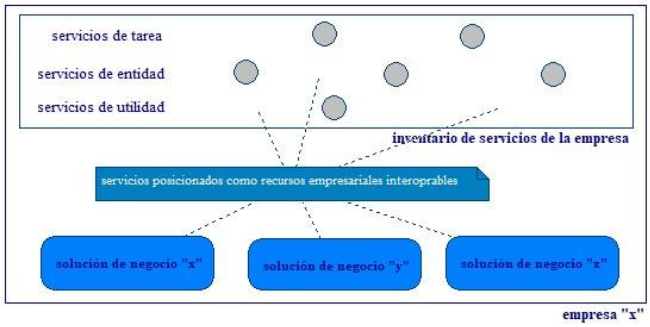


Figura 32: Una empresa donde los servicios encapsulan funcionalidad que es reutilizada en diferentes soluciones de negocio. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor confiabilidad:

⊕ En cuando al comportamiento esperado de los servicios los cuales actúan de manera similar en diferentes soluciones de negocio.

Aumento de interoperabilidad:

⊕ Ya que la encapsulación de funcionalidad en los servicios fomenta la reutilización de éstos en diferentes soluciones empresariales.

7.2.1.7.8. Nombre: Capacidad de composición (Erl).

Sinopsis: Este patrón diseño indica a la capacidad del servicio que le permite formar parte

de una o más composiciones, independientemente de la complejidad que esto atañe, lo cual permite resolver problemas mayores de los que resuelve un servicio individual.

Nombre alternativo: Capacidad de diseño caja negra.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful.*

Problema: Servicio desnormalizado de forma inadecuada en base al cumplimiento de requisitos de procesamiento que se encuentran fuera del contexto funcional del componente en la nube.

Solución: Diseño de lógica que permite el establecimiento de composiciones de dos o más servicios para resolver problemas que están fuera del ámbito funcional de un servicio específico.

Aplicación: La funcionalidad encapsulada por una capacidad incluye lógica que invoca capacidades de otros servicios. Es decir, las operaciones de los servicios hacen llamado a otros servicios para la resolución de problemas de manera conjunta. En el caso de los servicios *ReSTful*, el cumplimiento de *RMM* nivel 3 mediante estructuras explicitadas en controles o *links* hace evidente la conjunción de composiciones proveyendo mayor interoperabilidad que los servicios *SOAP*.

Efectos: Disminución de autonomía de servicio en cuanto a la generación de dependencias entre los componentes de las composiciones.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción, reusabilidad, autonomía, servicio sin estado, capacidad de composición y descubrimiento dinámico (éste último equivale a *RMM* nivel 3 para servicios de estilo *ReST,* mediante el uso de especificaciones

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

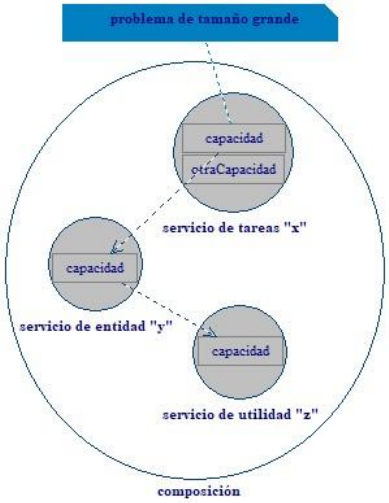


Figura 50: Conglomeración de capacidades individuales de los servicios para resolver un problema de tamaño considerable que no podría haber sido resuelto de manera aislada. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor rendimiento:

⊖ En cuanto a que la comunicación entre los servicios de la composición se realiza en forma remota, por lo cual en las arquitecturas *SOA* de microservicios, se recomienda establecer una tecnología de mensajería de alto rendimiento para tal propósito como *Apache Kafka*® (ver sección 7.2.1.10.8).

Aumento de interoperabilidad:

⊕ Debido a que el establecimiento de composiciones impulsa la reutilización de servicios para resolver problemas de negocio de gran envergadura.

7.2.1.7.9. Nombre: Capacidad de recomposición (Erl).

Sinopsis: Este patrón de diseño señala la necesidad de reúso de funcionalidades de servicio como parte de distintas composiciones que resuelven múltiples problemas de negocio.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful.*

Problema: El uso de funcionalidad de servicio independiente para resolver un solo

problema es inútil ya que no aprovecha el potencial de reutilización existente.

Solución: Diseño de funcionalidad de servicio agnóstica que es invocada repetidamente en apoyo de diferentes composiciones que resuelven múltiples problemas.

Aplicación: La recomposición efectiva requiere la aplicación coordinada, exitosa y repetida de varios patrones adicionales como "capacidad agnóstica al contexto, capacidad de composición, centralización de procesos y transformación del modelo de datos".

Efectos: Mayor esfuerzo de gestión y normalización de servicios para lograr reutilizar sus capacidades de manera repetida en múltiples composiciones.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción, reusabilidad, autonomía, servicio sin estado, capacidad de composición y descubrimiento dinámico.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

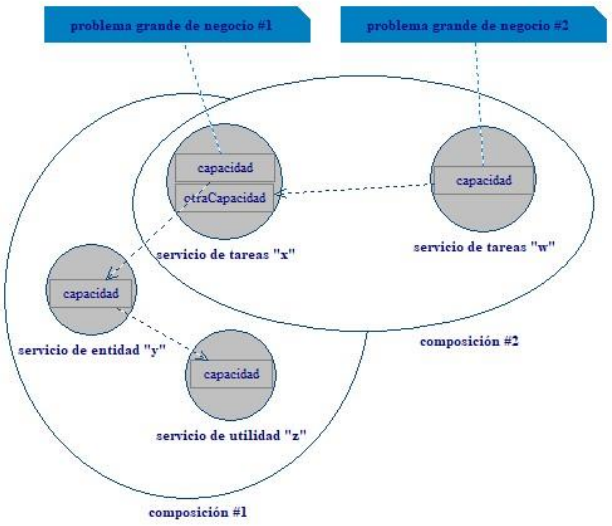


Figura 51: Reutilización de capacidades individuales de servicios en distintas composiciones para resolver varios problemas de tamaño considerable. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que el establecimiento de recomposiciones impulsa el reúso de capacidades de servicios como recursos empresariales intercambiables los cuales pueden ser usados en diferentes soluciones de negocio.

7.2.1.8. Patrones de composición:

7.2.1.8.1. Nombre: Controlador agnóstico de subtareas (Erl).

Sinopsis: Este patrón de diseño señala el establecimiento del servicio controlador de subtareas con capacidades agnósticas para facilitar la recomposición de subconjuntos de operaciones específicas.

Nombre alternativo: Sub controlador agnóstico.

Servicios afectados:

- Por funcionalidad: De tareas principalmente y de entidad en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Las composiciones de servicio generalmente están configuradas para cumplir con una tarea principal, inhibiendo el potencial de reutilización que puede existir dentro de subconjuntos de operaciones pertenecientes al conglomerado de servicios.

Solución: Abstracción de subconjuntos de capacidades, correspondientes a diferentes servicios de la composición vinculados entre sí, mediante un controlador independiente de subtareas, lo cual aumenta el potencial de recomposición.

Aplicación: Creación de un nuevo servicio de tareas independiente que hace de controlador de subtareas de la composición. Así mismo, se le puede añadir este subconjunto de funcionalidad a un servicio de la capa de dominio ya existente.

Efectos: La creación desmesurada de controladores agnósticos de subtareas incrementa la complejidad de las composiciones y puede ocasionar violaciones al modelado de servicio en capas (ver secciones 7.2.2.1.1, 7.2.2.1.2 y 7.2.2.1.3).

Principios soportados: Reusabilidad de servicio y capacidad de composición.

Componentes de arquitectura involucrados: Composición de servicios.

Diagrama de arquitectura:

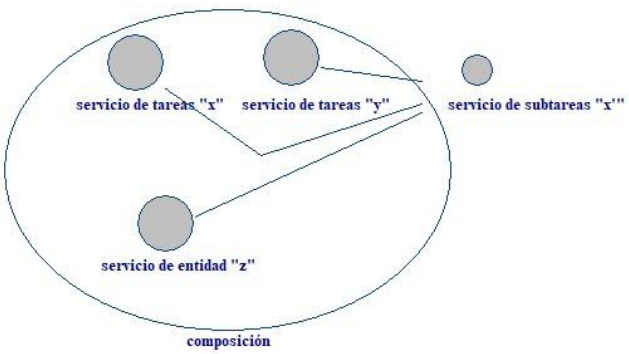


Figura 58: El servicio de subtareas (x') abstrae un subconjunto de funcionalidad de la

composición de servicios (x, y, z). Fuente propia elaborada con *StarUML*.

Mejora de *QoS:* Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Ya que la abstracción de subtareas de la composición, mediante un servicio controlador, aumenta el potencial de recomposición e impulsa la reutilización de servicios.

7.2.1.8.2. Nombre: Transacción atómica de servicio (Erl).

Sinopsis: Este patrón de diseño indica el uso de transacciones atómicas que se distribuyen entre las actividades del servicio en tiempo de ejecución para simplificar el seguimiento de las tareas de negocios y, también, reiniciar las acciones incompletas en caso de fallas.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Cuando las actividades de los servicios en tiempo de ejecución fallan las tareas de dominio se consideran incompletas, lo que da como resultado, acciones en estado intermedio que pueden comprometer la integridad de la solución de negocio y arquitectura subyacentes.

Solución: La tarea de dominio se vincula a una transacción atómica la cual es distribuida entre los servicios que participan de las actividades en tiempo de ejecución con el fin de simplificar el seguimiento de las acciones implicadas y, además, reiniciar aquellas que quedan en estado inconsistente cuando existen fallas.

Aplicación: Instauración de un sistema gestor de transacciones que forma parte de la arquitectura del inventario y se utiliza en aquellas composiciones de servicio que requieren funcionalidad de seguimiento y reinicio de actividades ante fallas. Dependiendo del diseño definido, la estrategia de recuperación ante fallas puede reiniciar el estado de todas las actividades de servicios relacionadas a la transacción atómica o sólo reestablecer el estado de aquellas que quedaron incompletas. El primer encuadre de recuperación se utiliza en dominios de negocio asociados a la seguridad como los bancos y el segundo se denomina enfoque de “mejor esfuerzo” y se usa en casos donde la seguridad transaccional puede ser más laxa, por ejemplo, la edición de interfaz de usuario dividida en secciones que no poseen información sensitiva (si falla una sección no es necesario volver a editar

todo de nuevo).

Efectos: El uso de transacciones atómicas distribuidas entre servicios añade complejidad de diseño ya que se debe tener en cuenta el mantenimiento de información transaccional durante las actividades en tiempo de ejecución, lo cual es bastante difícil cuando se implementan soluciones dirigidas por eventos (ver secciones 7.2.1.2.1, 7.2.1.2.7 y 7.2.1.2.9).

Principios soportados: Servicio sin estado.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

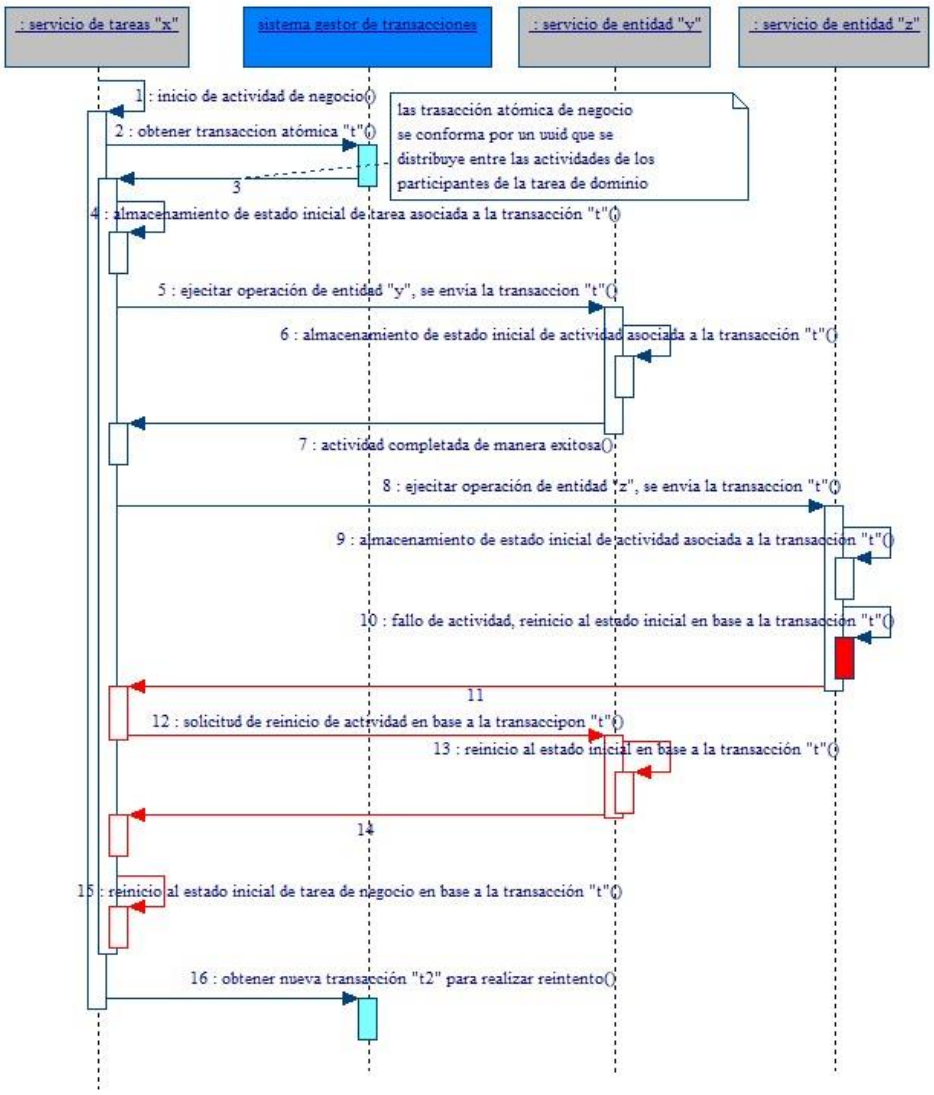


Figura 59: La transacción atómica (t) se distribuye ente las actividades en tiempo de ejecución de los servicios (y, z) que son orquestadas por la tarea de negocio (x) mediante el sistema gestor de transacciones. Posteriormente, el servicio de entidad (y) alacena su estado inicial y realiza su acción de manera exitosa. Por último, otro servicio de entidad

(z) guarda su estado base y realiza su función, pero como falla se solicita el reinicio del estado de la transacción atómica (t) entre todos los participantes, lo cual garantiza la integridad de la solución de negocio evitando estados incompletos de operaciones. Posteriormente Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Disminución de rendimiento:

⊖ Ya que el uso transacciones distribuidas implica un mayor uso de memoria/recursos de los servicios que deben conservar su estado inicial hasta que se solicite el reinicio por falla o se confirme que la transacción atómica concluyó exitosamente.

Aumento de confiabilidad:

⊕ Respecto al comportamiento esperado de los servicios relacionados a la tarea de negocio ante fallas debido a que la integridad de la solución no se compromete en cuanto a que las actividades participantes que quedaron inconsistentes pueden seguirse de manera común y reestablecerse adecuadamente.

7.2.1.8.3. Nombre: Compensación de transacciones de servicio (Maier, Normann).

Sinopsis: Este patrón de diseño se refiere a la escritura de lógica de compensación para reducir el uso de recursos que se emplean mediante la implementación de una solución de servicio basada en transacciones atómicas (ver sección 7.2.1.8.2).

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La utilización de transacciones atómicas para resolver problemas de integridad de servicios ante posibles situaciones de fallas en tiempo de ejecución requiere de un elevado uso de memoria/recursos para conservar el estado de origen de los participantes en el caso de que sea necesario reiniciar las actividades que quedaron inconsistentes.

Solución: Instauración de rutinas de compensación que permiten que las excepciones en tiempo de ejecución se resuelvan moderando el consumo de memoria/recursos.

Aplicación: Implementación de la lógica de compensación como parte del servicio de tareas principal el cual orquesta las actividades de los servicios que conforman la composición o bien mediante los servicios individuales que presentan características especiales para rehacer las actividades en curso de manera eficiente. Por ejemplo, a través

de tecnología de cache como *Guava* (ver sección 7.2.1.10.7) para guardar información de estado durante períodos razonables evitando emplear de manera intensiva el acceso a base de datos, lo que implica que tampoco se guarda toda la información en memoria ya que el uso de cache se encuentra limitado por tiempos, lo que resulta en la utilización moderada de ambas tecnologías de almacenamiento.

Efectos: A diferencia de las transacciones atómicas que se rigen por reglas específicas, el uso de la lógica de compensación es abierto, lo que implica que su efectividad real varíe de acuerdo a la funcionalidad personalizada que se aplicó.

Principios soportados: Acoplamiento débil de servicio.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Incremento de rendimiento:

⊕ Ya que con el empleo de lógica de compensación de transacciones se mitiga el uso elevado de memoria/recursos.

Menor confiabilidad:

⊖ Debido a que la funcionalidad de compensación es de carácter personalizado y, por lo tanto, puede variar en su efectividad.

7.2.1.8.4. Nombre: Composición autónoma (Erl).

Sinopsis: Este patrón de diseño señala la instauración aislada de las composiciones de servicios para elevar la autonomía da las mismas.

Nombre alternativo: Autonomía de composiciones.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Los servicios de tareas que orquestan las acciones de la composición pierden autonomía al delegar las actividades de procesamiento a participantes que son utilizados en múltiples composiciones.

Solución: Todos los participantes de la composición se aíslan para maximizar la autonomía de esta en su conjunto.

Aplicación: Los miembros de la composición, incluido el servicio de tareas orquestador, se despliegan de manera redundante en un entorno aislado. Es decir, se crean instancias de servicios de la composición que son replica de los originales y se independizan en un

ambiente separado.

Efectos: Elevar la autonomía a nivel de composiciones resulta en incremento de costos de infraestructura y mayor complejidad de gestión.

Principios soportados: Autonomía de servicio, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Composición de servicios.

Diagrama de arquitectura:

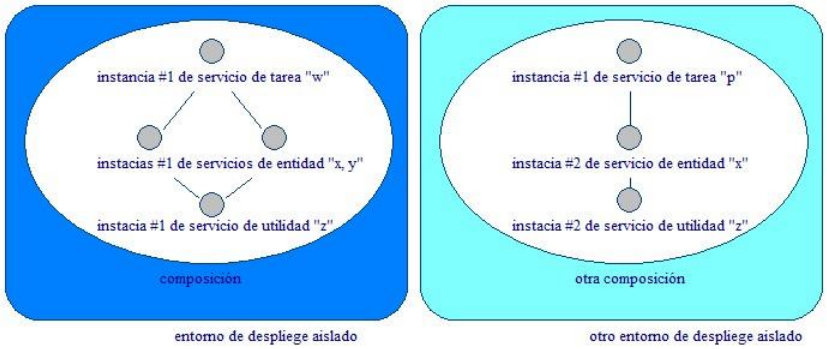


Figura 60: Al agrupar instancias diferentes de los servicios en las composiciones sobre entornos de despliegue aislados, la autonomía colectiva de estas últimas se maximiza ya que los miembros comunes (x, z) no se comparten. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor confiabilidad:

⊖ Respecto al comportamiento esperado de los servicios redundantes o instancias ya que la conducta de estos puede variar dependiendo del entorno de despliegue, por ejemplo, si un ambiente se configuró de manera inadecuada, sin suficientes recursos de *hardware*, puede existir la percepción de que una instancia de servicio independiente está fallando cuando en realidad fue diseñada para funcionar de manera agnóstica.

Aumento de interoperabilidad:

⊕ Debido a que el despliegue aislado de las composiciones en ambientes separados disminuye el acoplamiento entre los servicios que participan en múltiples composiciones, lo que resulta en más autonomía de las composiciones respecto a los entornos subyacentes y mayor reúso de los miembros.

7.2.1.9. Patrones originados en *ReST*:

7.2.1.9.1. Nombre: Entidades vinculadas (Balasubramanian).

Sinopsis: Este patrón de diseño señala que los servicios expongan en sus interfaces los vínculos con otras entidades de la composición para facilitar el descubrimiento de las interrelaciones.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas y entidad primordialmente, aunque los de utilidad en menor medida también.
- Por estilo: *ReSTful* principalmente mediante el cumplimiento de *RMM* de nivel 3 (ver sección 2.3.2.1). Los *SOAP* siguen este patrón si se cumple con *HATEOAS* en los mensajes *XML* de las respuestas o, de forma minimalista, si en la sección documentación del *WSDL* se explicitan los vínculos a entidades de la composición.

Problema: A los clientes se les dificulta entender las interacciones entre los servicios que consumen ya que estos últimos son diseñados de manera autónoma complicando el descubrimiento de interrelaciones necesarias para cumplir con el objetivo de negocio de la composición. Además, complejidad de mantenimiento si el cliente desarrolla las interrelaciones entre los servicios consumidos y existen cambios de vínculos posteriores a los establecidos originalmente.

Solución: Los servicios informan a sus clientes sobre la existencia de entidades relacionadas como parte de las interacciones del consumidor con los componentes en la nube.

Aplicación: Inclusión de enlaces en los mensajes de respuesta de los servicios que permiten a los consumidores poder navegar de una entidad a otra acumulando conocimiento de negocio. De esta manera los clientes pueden comenzar a desarrollar su lógica de negocio con un esfuerzo inicial reducido sin necesidad de conocer todas las interrelaciones posibles desde el comienzo. En los servicios *ReSTful* este patrón se lleva a cabo acatando especificaciones establecidas en 2.3.2.1.

Efectos: Los identificadores de recursos que representan a las entidades comerciales deben permanecer relativamente estables a lo largo de la vida útil de éstas, ya que, los consumidores de los servicios los consultan de manera frecuente. La definición de los enlaces se dificulta si los identificadores de las entidades son específicos a los servicios que pertenecen, por lo cual, se sugiere el establecimiento de vínculos con una sintaxis uniforme entre las composiciones del inventario. Los *links* no son valiosos si el cliente del servicio no puede acceder a la información sobre la entidad vinculada, por lo tanto, se

recomienda la inclusión de las entidades relevantes de negocio con poca frecuencia de cambio.

Principios soportados: Abstracción de servicio, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

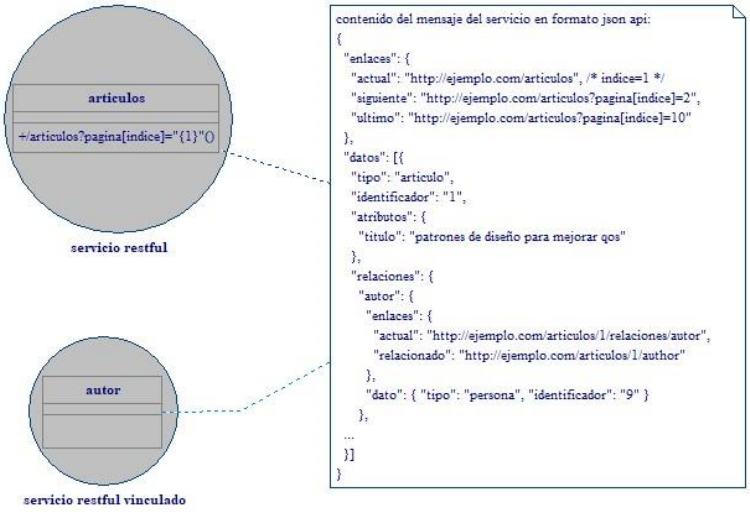


Figura 67: Enlaces como parte de la respuesta del servicio *ReSTful* (operación "obtener artículos de la página número 1") cumpliendo con *JSON API*. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor confiabilidad:

- En cuanto al comportamiento de las interrelaciones de los servicios de la composición ya que los vínculos en los mensajes de servicio presentan una visión actualizada de la conducta de negocio esperada, evitando así que los clientes implementen relaciones inadecuadas debido a la falta de información.

Aumento de interoperabilidad:

- Debido a que la presencia de enlaces en las respuestas de los mensajes de servicio permite que los consumidores comprendan la lógica de negocio con un esfuerzo inicial mínimo que impulsa la reutilización.

7.2.1.9.2. Nombre: Enlaces granulares (Balasubramanian).

Sinopsis: Este patrón de diseño indica que las interfaces de los servicios manifiesten vínculos livianos a capacidades granulares o entidades de negocio específicas impulsando

el potencial de recomposición.

Nombre alternativo: Enlaces livianos.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *ReSTful* principalmente mediante el cumplimiento de *RMM* de nivel 3 (ver sección 2.3.2.1) siguiendo especificaciones establecidas en 2.2.

Problema: Intercambio inútil de datos e ineficiencia de procesamiento por parte de los consumidores debido a que las interfaces de los servicios que proveen las funcionalidades de negocio poseen vínculos a capacidades innecesariamente integrales o muy genéricas.

Solución: Los enlaces definidos en las interfaces de los servicios hacen referencia a capacidades de negocio granulares o entidades de negocio específicas, por lo tanto, se optimiza el intercambio de datos entre las partes y el procesamiento de los consumidores.

Aplicación: Definición de convenios de servicio que exponen funcionalidad granular mediante vínculos livianos a entidades comerciales específicas.

Efectos: Interfaces de servicio verborragias con múltiples enlaces a operaciones granulares relacionadas que requieren mayor esfuerzo de gestión.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

<u>Aumento de rendimiento:</u>

⊕ Debido a que la definición de convenios con vínculos livianos a funcionalidad granular permite que los consumidores optimicen el procesamiento con actividades de negocio específicas a las necesidades.

<u>Mayor interoperabilidad:</u>

⊕ Ya que las interfaces que poseen enlaces a entidades poco genéricas fomentan la reusabilidad de los servicios y la potencialidad de recomposición.

<u>Reducción de interoperabilidad:</u>

⊖ Referido a que el uso excesivo de interfaces que poseen enlaces muy granulares a entidades de negocio independientes sin una combinación adecuada junto a funcionalidad de negocio esencial con baja frecuencia a cambio dificulta el mantenimiento del código y la reutilización.

7.2.1.9.3. Nombre: Contrato reusable (Balasubramanian).

Sinopsis: Este patrón de diseño se refiere a la definición de contratos reusables con funcionalidad relevante de negocio y baja frecuencia de cambio para evitar que los clientes tengan que actualizar su funcionalidad de manera recurrente.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas principalmente y los de entidad en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Contratos con alta frecuencia de cambio que requieren actualizaciones reiteradas por parte de los consumidores los cuales, desafortunadamente, están altamente acoplados a los ciclos de desarrollo de los servicios.

Solución: Los servicios de dominio exponen contratos con funcionalidad relevante de negocio, de alto nivel, que pose menos probabilidad de verse afectada mediante cambios de requerimientos.

Aplicación: Los servicios de tareas establecen contratos reutilizables que proporcionan métodos que abstraen funcionalidad de negocio principal y son agnósticos al contexto subyacente. Las funcionalidades de los contratos reusables generalmente se enfocan en tipos de información en lugar de acoplarse a la operativa de negocio específica. En los servicios *ReSTful,* la sintaxis de este parón se aplica mediante el cumplimiento de *RMM* nivel 3 (ver sección 2.3.2.1) debido a la combinación entre verbos comunes e identificadores de recursos los cuales se expresan bajo los lineamientos de *HTTP* y, por lo tanto, existe baja frecuencia de cambio de la interfaz del protocolo de comunicación. Adicionalmente, se considera que *AMM* nivel 3 (ver sección 2.4.1) implementa la semántica expresada por este patrón de diseño, ya que en este nivel se plantea el establecimiento operaciones de negocio enfocadas a las necesidades generales de los clientes. Finalmente, en los servicios *SOAP*, se pueden definir contratos reutilizables mediante la declaración de documentos *WSDL* centralizados con operaciones comunes de negocio.

Efectos: Compartir el mismo contrato entre múltiples clientes y servicios aumenta la importancia de que éste se defina correctamente desde el inicio para que posea mayor vigencia, por lo tanto, éste requiere mayor esfuerzo de gestión en comparación a convenios específicos de servicio.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que la publicación de contratos de servicio de dominio con operaciones de negocio relevantes y baja frecuencia de cambio que promueven la reutilización.

7.2.1.9.4. Nombre: Negociación de contenido (Balasubramanian).

Sinopsis: Este patrón de diseño indica que los servicios admiten la negociación de múltiples formatos de contenido durante el intercambio de los mensajes con los consumidores.

Nombre alternativo: Tipo de medio.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Consumidores con diferentes requisitos en cuanto a como se debe representar la información proporcionada por los servicios.

Solución: Los servicios admiten formatos alternativos de contenido, para representar los datos de los mensajes, que son negociados en tiempo de ejecución.

Aplicación: Este patrón se cumple mediante de la admisión de diferentes tipos de contenido o *accept+content-type* (ver sección 2.4 *media types*) como metadatos del encabezado de los mensajes a través del protocolo *HTTP* donde se definen el formato de envío y la representación de la respuesta. De esta manera, los clientes establecen los metadatos en cada mensaje solicitando formatos de envío y respuesta preferidos y, posteriormente, los servicios intentan adaptarse a las preferencias, aunque también se puedan aceptar y emitir los mensajes por medio de otros tipos de contenido.

Efectos: Proveer funcionalidad para aceptar diferentes formatos de envío y respuesta implica la existencia de lógica de procesamiento adicional que se puede complejizar mediante la admisión exagerada de múltiples medios de representación de datos en base a diversas versiones.

Principios soportados: Contrato de servicio estándar, acoplamiento débil y reusabilidad.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

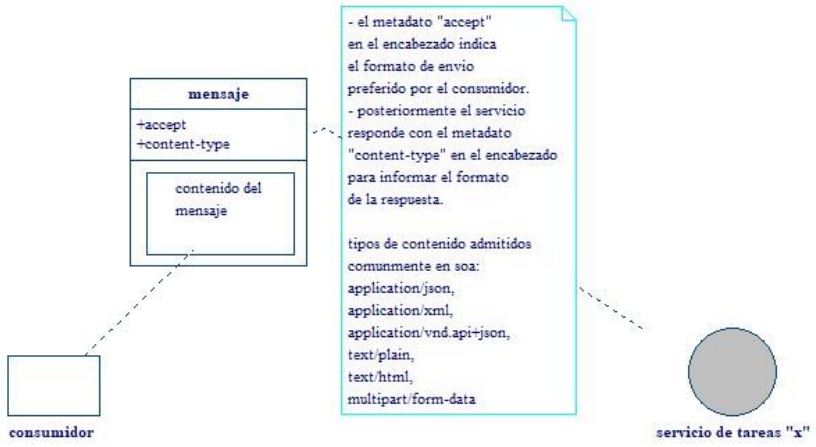


Figura 68: El servicio (x) admite el intercambio de mensajes a través de diferentes tipos de contenido explicitados como metadatos del encabezado. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de rendimiento:

⊕ Respecto a que los clientes negocian con los servicios sus preferencias de tipo de contenido y, por lo tanto, envían y reciben los modelos de datos sin necesidad de añadir lógica personal para adaptar el formato de la información intercambiada.

Mayor interoperabilidad:

⊕ Ya que la admisión de múltiples formatos de interacción de mensajes fomenta el reúso de servicios por parte de consumidores con diferentes necesidades.

7.2.1.9.5. Nombre: Idempotencia (Wilhelmsen).

Sinopsis: Este patrón señala el diseño de servicios idempotentes que admiten la recepción de mensajes repetidos sin consecuencias negativas sobre la operativa del negocio.

Nombre alternativo: Capacidad idempotente.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Servicios incapaces de procesar copias del mismo mensaje de manera reiterada debido al reenvió de información por fallas en la red o problemas en el *hardware* de los participantes.

Solución: Diseño de servicios con lógica idempotente (ver sección 4.1.1) que les permite

aceptar, de manera confiable, intercambios repetidos de mensajes.

Aplicación: La idempotencia garantiza que las llamadas repetidas a las operaciones del servicio no posean ningún efecto negativo. Las acciones de consulta o lectura se consideran idempotente por naturaleza ya que no modifican el estado de la información. El diseño de la funcionalidad idempotente incluye el uso de identificador unívoco o *UUID* (ver sección 7.2.1.2.4) en cada envío para que las solicitudes de mensajes repetidos sean ignoradas o descartadas, en lugar de ser procesadas nuevamente.

Efectos: La utilización del identificador único para asegurar el cumplimiento de funcionalidad idempotente requiere que el estado de la sesión sea almacenado de manera confiable por el servicio, conservándolo si ocurren fallas, lo cual puede disminuir la escalabilidad del servicio.

Principios soportados: Contrato de servicio estándar, servicio sin estado y capacidad de composición.

Componentes de arquitectura involucrados: Inventario, composición y servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de confiabilidad:

⊕ Respecto al comportamiento esperado de los servicios idempotentes el cual no se ve afectado de manera negativa mediante envíos repetidos de mensajes.

Mayor de interoperabilidad:

⊕ En cuanto al aumento de robustez ya que los servicios pueden funcionar correctamente incluso en presencia de ingresos no válidos, en este caso debido a ingreso de copias de mensajes reiterados, que son descartados mediante la lógica idempotente.

7.2.1.10. Patrones de implementación y despliegue:

7.2.1.10.1. Nombre: Aplazamiento parcial de estado (Erl).

Sinopsis: Este patrón de diseño establece la necesidad de aplazar parcialmente la preservación del estado en servicios donde es necesario mantenerlo, logrando una optimización del uso de recursos.

Nombre alternativo: Estado parcial diferido.

Servicios afectados:

➢ Por funcionalidad: De utilidad principalmente, ya que los servicios de capas

superiores suelen eludir el mantenimiento de información de estado.

- Por estilo: *SOAP* y *ReSTful*.

Problema: Incremento en el uso de memoria y reducción de escalabilidad como resultado de la gestión y almacenamiento de gran cantidad de información de estado en los servicios *Web* que así lo requieren.

Solución: Aún en casos donde es necesario mantener la información de estado en los servicios, se puede diferir temporalmente un subconjunto de datos de este tipo.

Aplicación: Existen varias alternativas para la gestión de información de estado de manera diferida, como servicios de utilidad que mantienen partes de información replicada en memoria o un repositorio de almacenamiento y recuperación de datos de forma parcial o *lazy load*.

Efectos: Mayor complejidad de diseño ligada a la administración de información de estado parcial en los servicios del inventario.

Principios soportados: Servicios sin estado.

Componentes de arquitectura involucrados: Inventario y servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+2) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+0))].

Incremento de rendimiento:

- ⊕ Debido a que la aplicación de este patrón da como resultado la misma cantidad de instancias de servicio concurrentes pero un menor consumo general de memoria relacionada con información de estado.
- ⊕ En cuanto a que la carga de información parcial o *lazy load* de datos es más rápida que traer toda la información o *eager load,* lo cual es incluso eficiente respecto a una reducción de redundancia de datos y menor de tráfico de red.

7.2.1.10.2. Nombre: Validación parcial de datos (Orchard, Riley).

Sinopsis: Este patrón de diseño señala que los clientes sólo validen información relevante de los servicios para lograr desacoplamiento y evitar redundancia en la verificación de datos poco significativos.

Nombre alternativo: Verificación parcial.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y los de utilidad en menor medida, ya que, en las composiciones los últimos casi nunca son consumidores.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Verificación de datos innecesarios por parte de clientes de servicios en base a convenios que imponen datos redundantes, como consecuencia de capacidades genéricas de servicios agnósticos al contexto.

Solución: Diseñar a los consumidores de servicios para que verifiquen sólo los datos significativos, ignorando el resto.

Aplicación: La aplicación de este patrón es específica a la tecnología utilizada en la implementación del consumidor del servicio. Por ejemplo, con servicios *Web* clásicos puede usarse *XPath*[46] y en los de estilo *ReST* se utiliza *JSONPath*[47] para filtrar datos innecesarios antes de la validación.

Efectos: Mayor esfuerzo de diseño e implementación de lógica adicional para el filtrado de datos innecesarios en tiempo de ejecución, lo cual reduce el procesamiento al evitar la validación de datos poco relevantes.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Composición de servicios.

Diagrama de arquitectura:

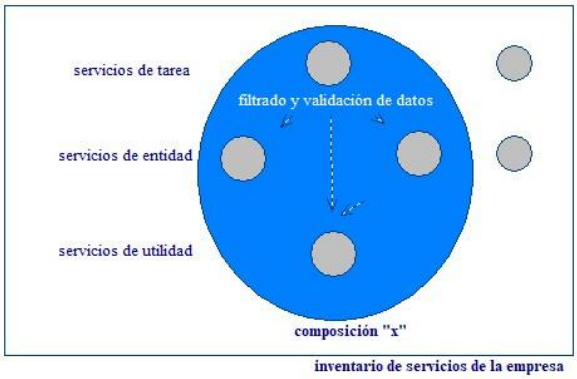


Figura 33: Servicios de tarea y entidad que son proveedores de lógica y también consumidores que verifican datos significativos al relacionarse con otros componentes de la composición. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de rendimiento:

⊕ Respecto a la operativa de los servicios que conforman las composiciones ya que los

[46] *XPath* es un lenguaje de consulta para seleccionar nodos de un documento *XML* de manera ágil. Para más información ver: <https://en.wikipedia.org/wiki/XPath>.

[47] *JSONPath* es una herramienta para seleccionar partes de una estructura *JSON* de manera análoga a lo que es *XPath* para *XML*. Para mas datos referirse a: <https://goessner.net/articles/JsonPath/>

datos innecesarios son filtrados, lo cual reduce el tiempo de verificación de información por parte de consumidores en las composiciones.

Mejora de interoperabilidad:

⊕ Debido a que la disminución de redundancia de datos mediante tecnologías de filtrado, como *XPath* o *JSONPath,* fomenta el reúso de servicios.

7.2.1.10.3. Nombre: Implementación redundante (Erl).

Sinopsis: Este patrón indica el requisito de instaurar servicios reutilizables a través de implementaciones redundantes y con recuperación ante errores o *failover*, para obtener alta disponibilidad.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El servicio que es reutilizado activamente puede inyectar puntos de falla únicos, cuando se producen condiciones de error inesperado, que ponen en peligro la confiabilidad de todas las composiciones en las que participa.

Solución: Despliegue de servicios reutilizables mediante mecanismos de implementación redundantes y recuperación ante errores.

Aplicación: Establecimiento de una infraestructura de despliegue de servicios con funciones de redundancia y recuperación ante errores, como *Amazon Global Infraestructure*[48].

Efectos: Mayor esfuerzo de gestión para mantener las implementaciones redundantes sincronizadas.

Principios soportados: Autonomía de servicios.

Componentes de arquitectura involucrados: Servicios.

Diagrama de arquitectura:

[48] Los componentes de la infraestructura de *AWS* se diseñan y crean para ofrecer redundancia y fiabilidad, desde las regiones y los enlaces de red a los balanceadores de carga, los *routers* y el *firmware*.

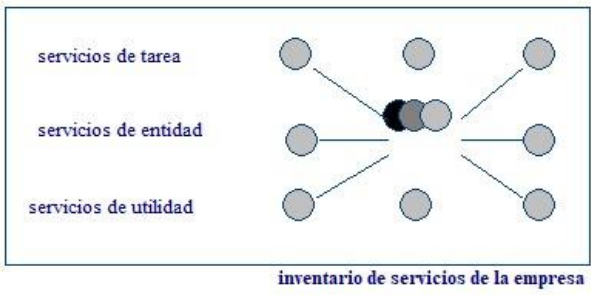


Figura 34: Un servicio agnóstico al contexto con múltiples implementaciones redundantes proporciona mayor tolerancia a fallas que otros, ya que, los mecanismos de recuperación ante errores garantizan que si una de estas implementaciones se cae otra puede tomar su trabajo. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Aumento de confiabilidad:

- ⊕ En cuanto al comportamiento esperado de servicios reutilizables debido a que el despliegue de implementaciones redundantes junto a mecanismos de recuperación ante errores incrementa la disponibilidad de los servicios del inventario.

7.2.1.10.4. Nombre: Replicación de datos de servicios (Erl).

Sinopsis: Este patrón de diseño se refiere a la necesidad de establecer servicios con bases de datos dedicadas y replicación de información compartida para mejorar la autonomía.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De entidad y utilidad principalmente ya que los de tarea suelen ser consumidores de servicios de nivel inferior que acceden a los datos de manera directa.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Los servicios pierden autonomía cuando se requiere acceso a información compartida.

Solución: Servicios con bases de datos dedicadas que replican información de una fuente de datos compartida.

Aplicación: Instauración de bases de datos adicionales, dedicadas a los servicios, y canales de replicación entre éstas y una fuente de datos compartida.

Efectos: Mayor complejidad de gestión de infraestructura vinculada a los canales de replicación y la sincronización de información compartida.

Principios soportados: Autonomía de servicios.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

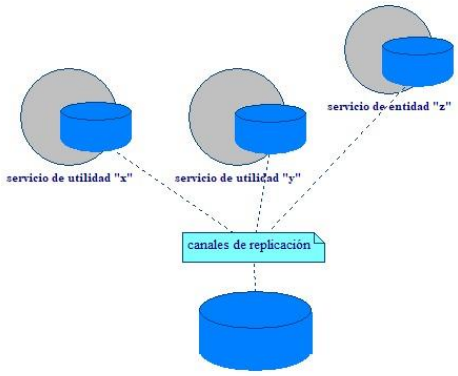


Figura 35: Al proporcionar a cada servicio su propia base de datos replicada, se incrementa la autonomía y también se reduce la presión sobre la fuente de datos compartida que centraliza toda la información. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))].

Aumento de performance:

⊕ Debido a la recuperación de datos distribuidos se hace de manera dedicada.

Degradación de rendimiento:

⊖ En cuanto al almacenamiento de información debido a que los canales de replicación y sincronización de datos compartidos agregan latencia bajo esta circunstancia.

Incremento de confiabilidad:

⊕ Referido al comportamiento esperado, ya que, al incrementar la autonomía de los servicios con bases de datos dedicadas la conducta de éstos se hace más previsible al haber mayor control sobre el contexto subyacente.

7.2.1.10.5. Nombre: Fachada de servicio (Sin autor).

Sinopsis: Este patrón de diseño señala el desarrollo de una fachada, como parte de la implementación del servicio, para que la lógica principal del éste se encuentre débilmente acoplada a recursos que afecten la interoperabilidad.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Primordialmente en los *SOAP* ya que la tendencia de los *ReSTful* es la

simplificación mediante el uso directo de recursos desde la funcionalidad principal de los servicios, lo cual trata de evitarse mediante el uso de este patrón.

Problema: El acoplamiento de la lógica principal de servicio a los convenios y recursos de implementación suele inhibir la evolución e impactar negativamente a los consumidores de servicios.

Solución: Uso de fachadas para la abstracción de partes de la arquitectura del servicio que poseen un potencial negativo de acoplamiento.

Aplicación: Incorporación de un componente de fachada durante el diseño del servicio *Web*. Por ejemplo, haciendo de intermediario entre el convenio y la lógica principal del servicio.

Efectos: Mayor esfuerzo de diseño y sobrecarga de desarrollo debido a la integración de un componente de fachada como parte del servicio.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Servicio.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

⊖ En cuanto a que añadir un componente de fachada, como parte del servicio, implica la implementación de una lógica de transformación de datos lo cual agrega latencia en comparación al acceso de estos de manera directa.

Aumento de interoperabilidad:

⊕ En consecuencia de permitir que la lógica principal de los servicios se encuentre débilmente acoplada a recursos vinculados al contexto mediante el uso de fachadas, lo cual fomenta el reúso y reduce el impacto negativo en los consumidores, por ejemplo, en situaciones de cambios de convenios o interfaces.

7.2.1.10.6. Nombre: Mediador de interfaz de usuario (Maier, Normann).

Sinopsis: Este patrón de diseño indica el establecimiento de un mediador entre los servicios *Web* y la capa de presentación para evitar que los clientes perciban comportamiento heterogéneo en la interfaz de usuario o *UI*.

Nombre alternativo: Mediador de *UI*.

Servicios afectados:

➢ Por funcionalidad: Aunque usualmente se apunte a los de utilidad como responsables de aplicar este patrón debido a que no orquestan procesos de negocio y pueden

cumplir tareas de mediación, existe controversia, ya que, en algunos casos, se señala a los de tarea por su ubicación (nivel superior). En definitiva, lo más apropiado es categorizar a estos servicios en un nivel nuevo de procesamiento denominado "servicios mediadores de *UI*".

- Por estilo: *SOAP* y *ReSTful*.

Problema: Mala experiencia de usuario debido al comportamiento fluctuante entre diferentes servicios *Web* de una solución *SOA* (usualmente, los servicios varían de acuerdo a su diseño, el tiempo de ejecución y la carga de trabajo asociada a realizar una tarea determinada).

Solución: Instauración de una lógica que haga de mediador entre los servicios y la capa de presentación, cuyas responsabilidades sean garantizar la interacciones con la interfaz de usuario y obtener retroalimentación oportuna ante mejoras.

Aplicación: Creación de un servicio mediador de utilidad o agente de servicio que se posiciona como el destinatario inicial de los mensajes que se originan en la interfaz de usuario. Esta lógica de mediación responde de manera oportuna y coherente, independientemente del comportamiento de la solución subyacente.

Efectos: La presencia de lógica de mediación añade un nivel más de procesamiento que impacta en la gestión y el rendimiento de los servicios del inventario.

Principios soportados: Acoplamiento débil.

Componentes de arquitectura involucrados: Composición de servicios.

Diagrama de arquitectura:

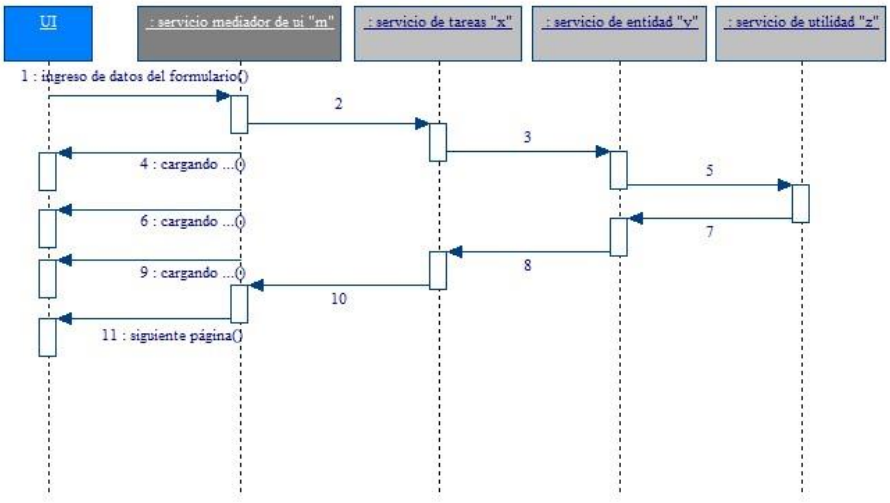


Figura 36: El servicio mediador de *UI* (m) actualiza regularmente la interfaz de usuario mientras que otros servicios (x, y, z) trabajan detrás de escena para completar la tarea. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de rendimiento:

⊕ Ya que el usuario advierte una respuesta rápida ante las acciones de la *UI*, lo que resulta en mejora de performance percibida.

Menor rendimiento:

⊖ En cuanto al tiempo de ejecución aislado debido a que una capa de procesamiento accesoria incrementa la latencia de servicios individuales.

Incremento de confiabilidad:

⊕ Respecto al comportamiento esperado de todos los servicios del inventario empresarial ante los usuarios, ya que, el agregado de un nivel adicional de servicios responsable de consolidar las interacciones con la *UI* provee una percepción homogénea de conducta.

Incremento de interoperabilidad:

⊕ En cuanto a que la lógica de mediación se encuentra encapsulada en un nuevo nivel de servicios independiente a la solución inferior, fomentando el reúso.

7.2.1.10.7. Nombre: Centralización de datos referenciales (O'Brien).

Sinopsis: Este patrón de diseño requiere el establecimiento de servicios comunes para la centralización de datos referenciales, lo cual evita la redundancia.

Nombre alternativo: Datos de referencia centralizados.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Redundancia de datos referenciales en múltiples servicios de dominio como, por ejemplo, códigos de negocio, países y ciudades. Adicionalmente, existe dificultad respecto a la administración de datos duplicados.

Solución: Gestión y almacenamiento de datos referenciales desde una ubicación física centralizada, la cual es accedida mediante el uso de servicios de utilidad para tal propósito únicamente. La capa de presentación y los servicios de negocio acceden a estos servicios de utilidad que presentan una vista común de información referencial.

Aplicación: Instauración de un repositorio centralizado de metadatos el cual es accedido mediante el uso de servicios de utilidad que se comunican con la interfaz gráfica u otros servicios de dominio proveyendo una vista compartida. Ejemplo, servicios de

internacionalización que se llaman desde la capa de presentación para mostrar información de idiomas y regiones de forma centralizada.

Efectos: La presencia de un nivel adicional de procesamiento compartido entre servicios puede convertirse en un cuello de botella sino se implementa de manera adecuada, por ejemplo, a través del uso de un mecanismo de cache para recuperar datos frecuentes.

Principios soportados: Reusabilidad de servicio.

Componentes de arquitectura involucrados: Inventario de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Menor rendimiento:

⊖ En cuanto a que la implementación de una lógica adicional compartida para acceder a datos referenciales requiere más procesamiento y, si es llevada a cabo de manera inadecuada, puede convertirse en un punto común de falla, motivo por el cual se recomienda el uso de tecnología como *Guava's Cache*[49] para datos frecuentes e instancias de servicios replicadas respecto al balanceo de carga y la recuperación ante fallas cuando se use este patrón.

Aumento de interoperabilidad:

⊕ Respecto a que la información referencial es gestionada desde una ubicación central, mediante servicios especializados para tal tarea, lo cual disminuye la redundancia y fomenta el reúso de servicios.

7.2.1.10.8. Nombre: Despliegue de microservicios (Merson).

Sinopsis: Este patrón de diseño señala la entrega independiente de cada uno de los servicios de la empresa, en paquetes separados, para mejorar la autonomía.

Nombre alternativo: Entrega de microservicio.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Los *ReSTful* primordialmente, ya que éstos se encuentran ligados fuertemente a una arquitectura *SOA* de microservicios, y los *SOAP* en menor medida.

Problema: Cuando los servicios *Web* y otros componentes, pertenecientes a una misma solución de *software,* se entregan en un mismo paquete se denomina despliegue

[49] *Guava's Cache* es una tecnología de almacenamiento de datos en memoria desarrollada por *Google* para el acceso a datos frecuentes sobre todo en casos donde es costosa la recuperación de éstos, por ejemplo, en entornos de servicios con una pobre latencia de respuesta. Ver más información en: <https://github.com/google/guava/wiki/CachesExplained>

monolítico. Usualmente, este tipo de implementación requiere un nuevo empaquetado de toda la solución de *software* cada vez que existe una nueva versión de un servicio en particular. Además, existe menos flexibilidad en cuanto al establecimiento de configuraciones específicas asociadas a los servicios, como por ejemplo, la escalabilidad, disponibilidad, persistencia, lógica de seguridad y el monitoreo, que se definen de manera conjunta (compartiendo recursos).

Solución: Cada servicio se trata como un producto independiente el cual se entrega en un paquete aislado.

Aplicación: Los servicios se empaquetan e implementan en un entorno altamente autónomo mediante el uso de tecnologías de contenedores (ver siguiente patrón). El empaquetado y el despliegue de servicios son altamente automatizados sobre la base de conceptos como *IaaS, PaaS* o *SaaS* (ver sección 2.4). Este patrón se lleva a cabo en arquitecturas donde la comunicación interna entre servicios se implementa de manera asíncrona, a través de canales de mensajería como *Apache Kafka®*[50].

Efectos: Los servicios son desarrollados y evolucionan de forma aislada. Las entregas son hechas a medida, de acuerdo a necesidades particulares. El despliegue de nuevas versiones de servicios conlleva tiempos de inactividad mínimos ya que no afecta a todo el conjunto. La automatización de la entrega y el empaquetado independiente en *IaaS* requiere un uso de red elevado lo cual afecta la performance sino se gestiona de manera adecuada.

Principios soportados: Autonomía de servicios y acoplamiento débil.

Componentes de arquitectura involucrados: Composiciones de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de rendimiento:

⊕ En cuanto a los tiempos de despliegue debido a que los servicios son entregados de manera aislada lo cual resulta en menor tiempo de inactividad.

Menor performance:

⊖ En cuanto al uso de red durante la automatización del despliegue.

Incremento de interoperabilidad:

⊕ En cuanto a que independizar la entrega de los servicios *Web* en paquetes específicos

[50] *Apache Kafka®* es una plataforma distribuida para el procesamiento rápido de mensajes usada por empresas como *Facebook*, *LinkedIn* y *McAfee.* Para mas detalle referirse a: <https://kafka.apache.org/intro>

permite reducir el acoplamiento entre éstos.

7.2.1.10.9. Nombre: Contenedores (Stoffers).

Sinopsis: Este patrón de diseño se refiere a la necesidad de entregar composiciones o servicios individuales empaquetados dentro de contenedores de imágenes, lo cual acrecienta la autonomía.

Nombre alternativo: Contenedores de servicios.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Los *ReSTful* en una arquitectura *SOA* de microservicios, y los *SOAP* en menor medida.

Problema: El despliegue monolítico de una solución de servicios conduce a una reducción de rendimiento y disponibilidad generalizada cuando cualquiera de los participantes sufre alguna interrupción o errores inesperados. También, si todos los servicios son desplegados, de manera conjunta, en los mismos servidores físicos o virtualizados para mejorar la portabilidad, se impone una huella significativa respecto al tamaño de la entrega conglomerada.

Solución: Las composiciones o servicios independientes se entregan como unidades autónomas que se empaquetan en imágenes de contenedores, como *Docker* (ver sección 2.2), las cuales son gestionadas de manera particular (incluyendo las dependencias con los sistemas subyacentes). También, se proporcionan las herramientas para administrar la construcción, el despliegue y la operativa de los contenedores.

Aplicación: Instauración de un motor o sistema de gestión de contenedores, como *Kubernates* (ver sección 2.2), para el despliegue y la operativa de las imágenes que empaquetan los servicios.

Efectos: El uso de contenedores impone requisitos de infraestructura adicionales y mayor complejidad en la gestión de esta tecnología.

Principios soportados: Autonomía de servicios y acoplamiento débil.

Componentes de arquitectura involucrados: Composiciones de servicios.

Diagrama de arquitectura:

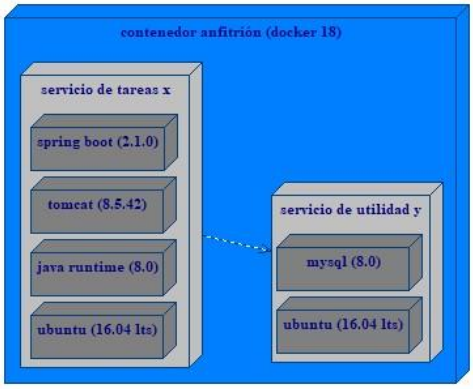


Figura 37: Imágenes de servicio de tareas (x) y de utilidad (y) como una composición empaquetada a través de un contenedor. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

Aumento de confiabilidad:

⊕ En cuanto al comportamiento esperado de que los servicios independientes ya poseen una mayor autonomía respecto a la gestión de sus recursos y el entorno de ejecución subyacente.

Incremento de interoperabilidad:

⊕ Referido a que los servicios, al ser empaquetados como unidades separadas, poseen un acoplamiento débil entre sí, lo cual fomenta la reutilización.

Menor interoperabilidad:

⊖ Referido a que el código de despliegue en *IaaS* o *PaaS* puede ser redundante si no es administrado de manera adecuada, por ejemplo, en contenedores de imágenes similares codificados de manera diferente respecto a como resolver dependencias análogas.

7.2.2. Patrones del inventario de servicios:

7.2.2.1. Patrones de clasificación lógica del inventario:

7.2.2.1.1. Nombre: Abstracción de entidades de negocio (Erl).

Sinopsis: Este patrón indica la necesidad de identificar servicios independientes que abstraen la lógica de negocio, promueven la reusabilidad y pueden ser gestionados aisladamente.

Nombre alternativo: Abstracción de entidades.

Servicios afectados:

- Por funcionalidad: De entidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El desarrollo de servicios con lógica de procesos centralizada mezclada a procesos de negocio independientes suele dar como resultado la presencia de redundancia de lógica de negocio a través de múltiples servicios.

Solución: Establecimiento de una capa de servicios de dominio, la cual agrupa los servicios que basan su contexto funcional en las entidades comerciales existentes.

Aplicación: Los modelos de las entidades de negocios (como clientes, empleados y facturas) se usan como base para establecer el agrupamiento lógico de servicios de entidades, la cual se define durante la fase de análisis.

Efectos: La naturaleza de este patrón conforma el corazón del negocio, por lo cual, los servicios introducidos requieren de modelado adecuado, prestando atención al diseño de los servicios de entidades de dominio y sus requisitos de gestión que pueden imponer cambios organizativos drásticos.

Principios soportados: Servicios débilmente acoplados, abstracción, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

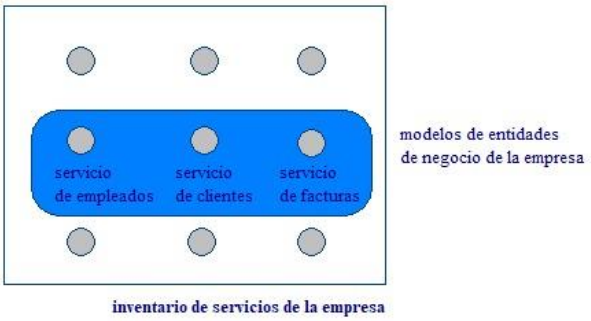


Figura 16: Servicios de entidades de dominio los cuales encapsulan el procesamiento asociado con una o más entidades comerciales específicas (o un grupo de entidades relacionadas). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de confiabilidad:

- ⊕ Respecto a la previsibilidad de comportamiento para los servicios de dominio ya que ejercen un alto nivel de control sobre sus recursos de negocio.

Incremento de interoperabilidad:

⊕ Debido a que se reduce la cantidad de servicios del inventario promoviendo la reusabilidad.

7.2.2.1.2. Nombre: Abstracción de tareas empresariales (Erl).

Sinopsis: Este patrón de diseño establece la necesidad de agrupar servicios de tareas los cuales pueden ser gestionados de manera independiente fomentando la reusabilidad.

Nombre alternativo: Abstracción de procesos.

Servicios afectados:

- Por funcionalidad: De tareas.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El agrupamiento de procesos centralizados junto con servicios de tareas independientes dificulta la gestión de lógica específica de las tareas disminuyendo la reusabilidad.

Solución: Instauración de una capa de servicios de tareas empresariales principales que respaldan la independencia de gestión y el posicionamiento de estos servicios como recursos empresariales potenciales.

Aplicación: Los servicios de las tareas primordiales de la empresa se definen después de que se hayan identificado los servicios de utilidad y de entidad, ya que éstos, los cuales se encuentran en la capa superior, son dependientes de aquellos ubicados en las capas intermedia e inferior.

Efectos: Además de las consideraciones de modelado y diseño asociadas con la creación de servicios de tareas (inversión inicial de análisis y gestión), estos procesos comerciales poseen una dependencia inherente en la ejecución causada por la composición de otros servicios.

Principios soportados: Servicios débilmente acoplados, abstracción, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

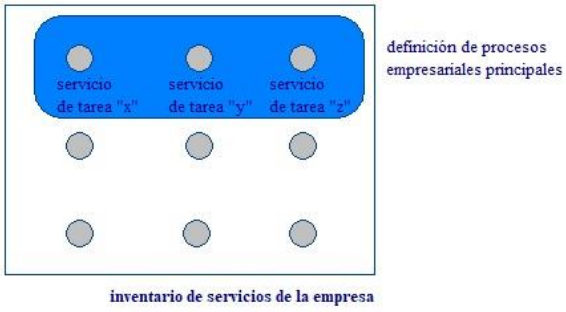


Figura 17: La solución empresarial se encuentra limitada al cumplimiento de los procesos empresariales primarios los cuales se abstraen en servicios de tareas independientes. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Respecto a la composición de servicios de tareas empresariales primordiales los cuales se pueden reutilizar en soluciones en procesos de negocio más complejos.

7.2.2.1.3. Nombre: Abstracción de utilidades (Erl).

Sinopsis: Este patrón de diseño se refiere al requisito de identificación y agrupamiento de servicios cuya funcionalidad no se relaciona directamente al negocio, lo cual impulsa la gestión independiente de los servicios de utilidades.

Nombre alternativo: Abstracción de utilidad.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El desarrollo de servicios con lógica de procesos de dominio mezclada a procesos de utilidades agnósticas al negocio suele dar como resultado la existencia de lógica redundante de utilidades a través de múltiples servicios.

Solución: Implementación de una capa de servicios dedicada a la ejecución de utilidades lo cual proporciona servicios reutilizables en el inventario.

Aplicación: El modelado de servicio de utilidad se incorpora en los procesos de análisis y diseño en apoyo a lógicas que abstraen utilidades las cuales son agnósticas al negocio principal de la empresa.

Efectos: Cuando la lógica de la capa de utilidad se distribuye a través de múltiples servicios, puede aumentar el tamaño, la complejidad y las demandas de rendimiento de

las composiciones.

Principios soportados: Servicios débilmente acoplados, abstracción, reusabilidad y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

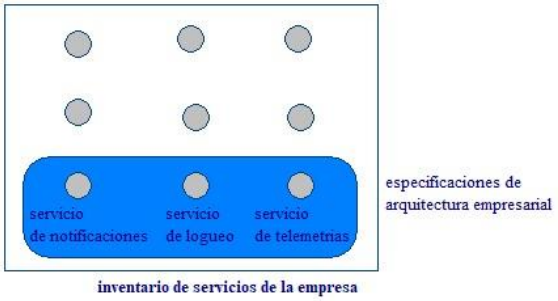


Figura 18: Las funcionalidades transversales agnósticas al negocio primario, en forma de utilidades, se identifican con la ayuda de las especificaciones de arquitectura de tecnología empresarial y luego se abstraen en una capa de servicios dedicados basados en el modelo de servicios de utilidad. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

- Referido a las operaciones de los servicios que añaden utilidades de monitoreo y *throublehsoot* como telemetrías y logueo que agregan procesamiento adicional, aunque existen técnicas de mitigación (por ejemplo, logueo de información por niveles de manera flexible según el entorno de ejecución evitando el uso, por defecto, de logueo excesivo en modo *debug o trace* en entornos superiores; salvo que sea necesaria la investigación o *troubleshoot* de un caso específico lo cual puede incurrir en el requisito momentáneo de habilitar en un período determinado de tiempo un modo más detallado).

Incremento de confiabilidad:

- Respecto al comportamiento esperado de todos los servicios del inventario ya que los servicios de utilidad sirven, primordialmente, para monitorear el comportamiento global del sistema, mediante diferentes tipos de herramientas como notificaciones, logueo y telemetría.

Aumento de interoperabilidad:

- En cuanto a la composición de servicios de utilidades comunes los cuales se pueden

reusar en el inventario.

7.2.2.1.4. Nombre: Abstracción de micro tareas (Erl).

Sinopsis: Este patrón de diseño explica la necesidad de identificación y agrupamiento de micro tareas ligadas al entorno subyacente bajo requerimientos específicos.

Nombre alternativo: Abstracción de tareas pequeñas.

Servicios afectados:

- Por funcionalidad: De tareas (Micro tareas en este caso ya que son una especialización de las tareas empresariales primarias).
- Por estilo: Es más común en servicios *ReSTful* vinculados a una arquitectura de microservicios.

Problema: El agrupamiento de la lógica del dominio vinculada con los requisitos de implementación y procesamiento especializados junto con la lógica empresarial que no tiene tales requisitos puede comprometer la capacidad de los primeros para cumplir de manera consistente con sus requerimientos.

Solución: Separación en unidades individuales de lógica dominio (micro tareas) con requisitos de implementación y procesamiento especializados mediante el modelo de microservicios; en el cual se establece una capa de microservicios en la que existe la libertad arquitectónica de adaptar los entornos para respaldar los requisitos mencionados.

Aplicación: Una vez que la lógica vinculada a procesos empresariales primarios se ha separado de la lógica agnóstica, se revisa para identificar unidades de lógica con requisitos de implementación y procesamiento especializados adecuados para la capa de microservicios.

Efectos: El establecimiento de micro tareas de dominio en una capa de servicio separada puede introducir sobrecarga en el análisis, el diseño y la gestión de los servicios. El modelo de microservicios se aplica comúnmente a la lógica de las micro tareas para implantar el entorno de implementación de servicios especializado, lo cual puede fomentar el uso de protocolos de comunicación dispares y exigir aún más la introducción de tecnología de implementación especializada que puede imponer nuevos requisitos de infraestructura, administración y gestión.

Principios soportados: Autonomía de servicios, acoplamiento débil, abstracción y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) +

seguridad (+-0) (+ interoperabilidad (-1))].

Aumento de rendimiento:

⊕ De acuerdo con la velocidad en la cual una solicitud de servicio se completa, debido a la mayor autonomía de los servicios de micro tareas que se encuentran ligados a requerimientos de implementación y procesamiento especializado.

Disminución de interoperabilidad:

⊖ Causada por la instauración de tecnología específica la cual puede incurrir en el uso de protocolos dispares de comunicación que afectan negativamente la reusabilidad y el acceso de los servicios de micro tareas de dominio pertenecientes a la capa de microservicios.

7.2.2.2. Patrones institucionales del inventario:

7.2.2.2.1. Nombre: Protocolo único (Erl).

Sinopsis: Este patrón de diseño se refiere a la necesidad de que los servicios del inventario usen un protocolo de comunicación único para así evitar dificultades vinculadas a dar soporte a múltiples tecnologías de comunicación.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Principalmente *SOAP* debido a que *ReSTful* usa *HTTP* por defecto; aunque en este segundo caso, tal como se indica en la sección Aplicación, se debe acordar el uso de una versión común.

Problema: Los servicios que admiten diferentes tecnologías de comunicación comprometen la interoperabilidad, limitan la cantidad de consumidores potenciales e introducen la necesidad de medidas indeseables como el establecimiento de protocolos que hacen de puente.

Solución: La arquitectura establece una sola tecnología de comunicaciones como el medio único o principal mediante el cual los servicios pueden interactuar.

Aplicación: Los protocolos de comunicación (incluidas las versiones de éstos) los cuales son utilizados dentro de los límites del inventario de servicios están estandarizados para todos los servicios.

Efectos: La arquitectura de un inventario de servicios en la cual los protocolos de comunicación están estandarizados está sujeta a cualquier limitación impuesta por la

tecnología de comunicación selecta.

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

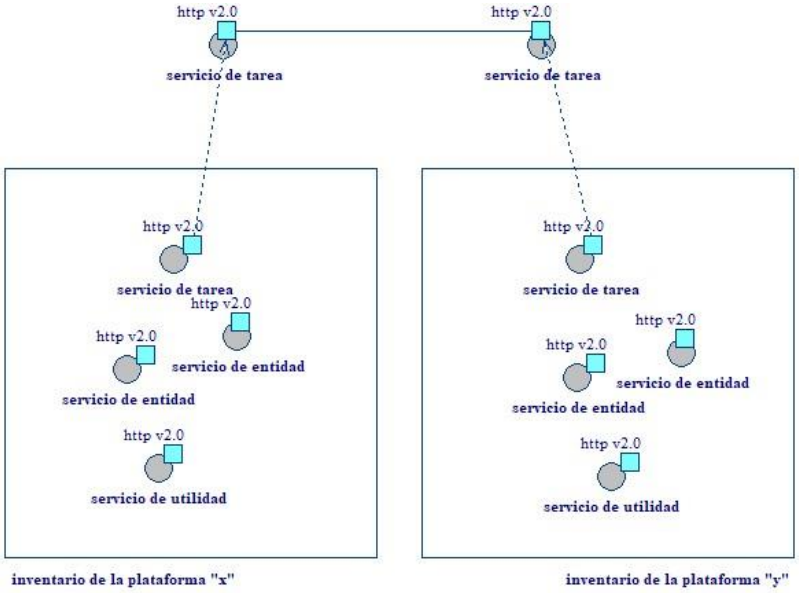


Figura 9: Servicios implementados sobre diferentes plataformas subyacentes mediante el uso de un mismo protocolo de comunicación. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+2))].

Aumento de confiabilidad:

- Respecto al comportamiento de las comunicaciones entre los servicios debido al uso de un mismo protocolo de conexión. Existen más variables, tal como se señala en la sección 6.3, pero tener un mecanismo de acceso común permite mayor predictibilidad en cuanto al tiempo de acceso.

Aumento de seguridad:

- Si el protocolo de comunicación seleccionado es seguro, como *HTTPS,* el rendimiento disminuye en comparación a *HTTP* pero la seguridad incrementa, lo cual se considera un balance necesario sobre todo para los servicios de tareas, los cuales pertenecen a la capa superior, que poseen un mayor vínculo con las aplicaciones de negocio (*UIs*).

Mayor interoperabilidad:

- Si el protocolo elegido soporta la programación asincrónica, como *JMS,* la capacidad de procesar múltiples solicitudes simultaneas aumenta ya que este paradigma evita los bloqueos de código en base a esperas de respuesta mediante el encolamiento de

solicitudes, aunque puede existir redundancia de datos (en el caso de una estrategia que garantice no perder datos), mientras que los protocolos síncronos poseen una precisión superior en cuanto a la tasa de error generada ya que estos últimos proveen mayores garantías en la entrega de respuestas.

⊕ El tener un protocolo común aumenta la disponibilidad de los servicios del inventario ya que no existe disparidad de acceso entre servicios lo cual, en promedio, determina un valor común de tiempo de reparación (*TTR*) para todo el inventario.

7.2.2.2.2. Nombre: Esquema de datos unívoco (Erl).

Sinopsis: Este patrón de diseño alude a que los servicios sean diseñados para evitar transformaciones de los modelos de datos mediante el uso de esquemas de datos unívocos.

Nombre alternativo: Esquema de datos único.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Aunque a simple vista pareciese que este patrón es exclusivo a los servicios clásicos *SOAP,* en los cuales los convenios *WSDL* pueden ser definidos a través de esquemas de datos *XSD* que se recomiendan sean compartidos entre servicios, en el caso de servicios *ReSTful* es de igual importancia ya que, a nivel empresarial, se suele usar *JSON Schema* o *YAML* en *Swagger* los cuales se pueden utilizar para definir el contenido univoco de un inventario de servicios.

Problema: Los servicios que presentan modelos dispares para datos similares imponen requisitos de transformación o adaptación los cuales aumentan el esfuerzo de desarrollo, la complejidad del diseño y el tiempo de ejecución.

Solución: Los modelos para conjuntos de datos equivalentes, dentro de los límites del inventario, están normalizados en todos los convenios de servicios.

Aplicación: Diseño de modelos normalizados de datos para los esquemas utilizados en los contratos de los servicios como parte de un proceso de diseño formal.

Efectos: El mantenimiento de la normalización de los esquemas de contrato puede introducir un importante esfuerzo de gestión, lo cual significa un desafío cultural.

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

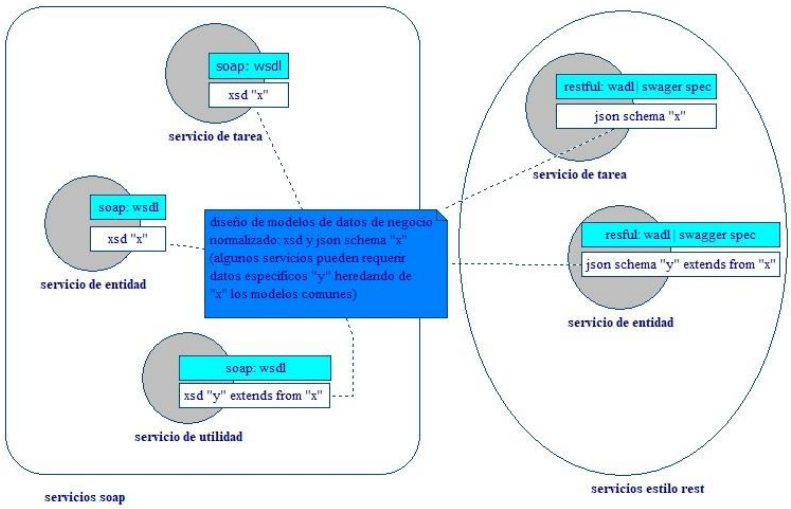


Figura 10: Múltiples servicios *Web* utilizan esquemas normalizados (*XSD o JSON Schema*) como resultado de la aplicación de este patrón. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de rendimiento:

⊕ En cuanto a reducción de tiempos de ejecución al evitar el uso de transformadores de datos o adaptadores entre servicios.

Incremento de interoperabilidad:

⊕ Debido a una mayor robustez respecto al ingreso de datos inválidos ya que los esquemas normalizados, compartidos entre servicios y propiamente diseñados, también sirven para validar los tipos de datos modelados (por ejemplo, estableciendo tamaños de los tipos de datos de ingreso o mediante expresiones regulares como en la validación de un atributo de tipo dirección *IP*).

7.2.2.2.3. Nombre: Inventarios de servicios de dominio (Erl).

Sinopsis: Este patrón de diseño apunta a la exigencia de proveer servicios reutilizables, cuando no es posible la normalización de éstos en toda la empresa, a través de inventarios de servicios de dominio del negocio.

Nombre alternativo: Inventarios de la empresa.

Servicios afectados:

➢ Por funcionalidad: De tareas y entidad primordialmente, aunque un inventario completo suele estar compuesto por todas las categorías (incluyendo los servicios de

utilidad).

- Por estilo: *SOAP* y *ReSTful* de igual manera.

Problema: El establecimiento de un solo inventario de servicios a nivel de toda la empresa puede ser inmanejable para algunas compañías, y los intentos de hacerlo pueden poner en peligro el éxito de la adopción de *SOA* en su conjunto.

Solución: Los servicios se pueden agrupar en inventarios de servicios manejables, específicos del dominio, los cuales pueden ser normalizados y gestionados de manera independiente por unidad de negocio.

Aplicación: Disposición cuidadosa del alcance de los inventarios de servicios por unidad de negocio.

Efectos: La disparidad en la normalización de inventarios de servicios de dominio por unidad de negocio impone requisitos de transformación y reduce el potencial de beneficios generales que se obtienen con la adopción de *SOA* a nivel global.

Principios soportados: Contrato de servicio estándar, abstracción de servicios y capacidad de composición.

Componentes de arquitectura involucrados: Unidades de negocio de la empresa e inventario de servicios.

Diagrama de arquitectura:

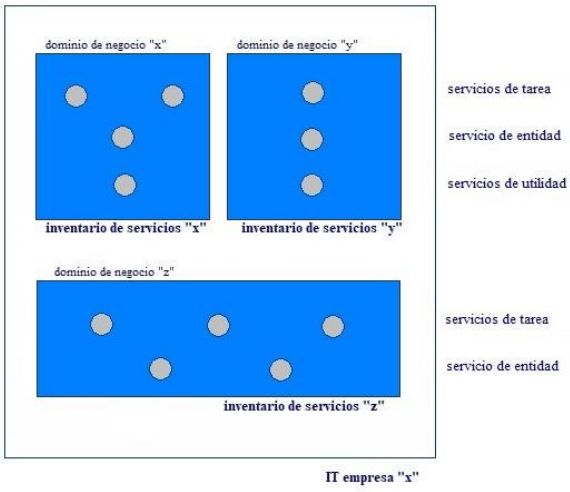


Figura 11: El comercio de una empresa divido entre inventarios de servicios de dominio por unidad de negocio, cada uno representando un dominio predefinido (normalizado). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (-1))].

Disminución de rendimiento:

⊖ En este patrón, el cual se vincula más con una necesidad de gestión individual por dominio debido a restricciones organizacionales (y no a nivel de *SOA* como un todo en la empresa, como se espera), se disminuye la performance en cuanto al proceso de las operaciones cuando los servicios de diferentes dominios interactúan entre sí ya que se requiere el uso de transformadores de datos o adaptadores para los datos que fueron normalizados de manera desigual en cada inventario.

Aumento de confiabilidad:

⊕ Referida a la predictibilidad del comportamiento de los servicios por unidad de negocio (aunque en cuanto a inventarios dispares por dominio se dificulta determinar la confiabilidad operativa en la empresa).

Menor interoperabilidad:

⊖ Cabe destacar que este patrón se considera una alternativa para mitigar el problema de la imposibilidad de normalizar los esquemas de datos dentro de la empresa como un todo a través de un inventario único, por lo tanto, la interoperabilidad de los servicios se reduce ya que es posible, por ejemplo, que un servicio de dominio pueda no comportarse de manera esperada si se lo traslada a un inventario de unidad de negocio disímil a la del origen de éste.

7.2.2.2.4. Nombre: Inventario global de servicios en la empresa (Erl).

Sinopsis: Este patrón de diseño se refiere a la necesidad de proveer servicios que reutilizables dentro de la empresa como un todo, a través de un inventario de servicios común en toda la empresa.

Nombre alternativo: Inventario global.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La entrega de servicios de forma independiente a través de diferentes equipos de proyecto en una empresa establece un riesgo constante de producir implementaciones de servicio y arquitectura inconsistentes, comprometiendo las oportunidades de reutilización y recomposición de éstos.

Solución: Diseño de servicios con una arquitectura de inventario estandarizada para toda la empresa, la cual permita la reutilización y recomposición de los servicios de manera libre y repetida.

Aplicación: Modelado, por adelantado, de un inventario común de todos los servicios de

la empresa, de tal manera que los estándares se aplican a nivel global afectando a todos los proyectos de la empresa, tal como se espera en *SOA*.

Efectos: Se requiere un análisis inicial significativo en la definición de un plan para establecer un inventario global de servicios en la empresa. También, existe impacto organizacional resultante de los requisitos de gestión de una solución a nivel de la empresa como un todo (la gestión individual por dominio suele ser más simple, aunque menos interoperable).

Principios soportados: Contrato de servicio estándar, abstracción de servicios y capacidad de composición.

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

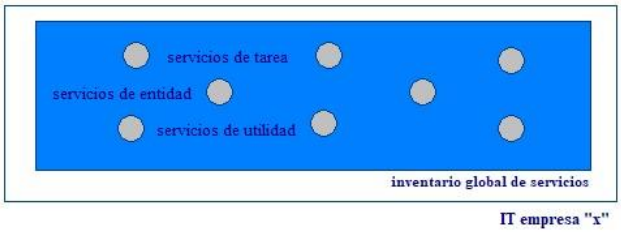


Figura 12: Un inventario global de servicios empresariales establece un alcance arquitectónico a nivel compañía como un todo, lo cual promueve la interoperabilidad, reutilización y recomposición entre todos los servicios. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1,5))].

Incremento de rendimiento:

- Referido al tiempo de ejecución de todos los servicios debido a que no se necesitan transformadores de datos o adaptadores entre servicios desarrollados por diferentes proyectos.

Aumento de confiabilidad:

- En cuanto a la predictibilidad del rendimiento en base al tiempo de ejecución de todos los servicios en la empresa.

Maximización de interoperabilidad:

- Es importante resaltar que el uso de este patrón de diseño se considera el esperado por una arquitectura orientada a servicios ya que se optimiza la interoperabilidad, incluyendo la reutilización y recomposición de servicios principalmente, aunque

requiere esfuerzo de gestión para normalizar los servicios en los diferentes proyectos de dominio.

7.2.2.2.5. Nombre: Lógica centralizada (Erl).

Sinopsis: En este patrón de diseño se sugiere evitar la presencia de funcionalidad redundante de los servicios mediante el diseño y la gestión apropiada de éstos.

Nombre alternativo: Centralización de lógica.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Cuando los servicios, diseñados de manera agnóstica (independiente), no se reutilizan constantemente, puede existir lógica redundante en otros servicios los cuales necesitan de funcionalidad similar, lo que resulta en problemas asociados con la gestión de los servicios, responsabilidad de los servicios compartida y desnormalización del inventario.

Solución: El acceso a la funcionalidad reutilizable está limitado a los servicios oficiales, los cuales son diseñados para ejecutarse de manera agnóstica, mediante el principio de reusabilidad, teniendo en cuenta el alcance de las responsabilidades de los servicios del inventario.

Aplicación: Diseño y gestión de servicios independientes de forma adecuada, teniendo en cuenta el uso de estándares de la empresa en cuanto a la utilización de éstos, los cuales promueven la centralización de las funcionalidades evitando la redundancia de lógica.

Efectos: Existencia de problemas organizacionales vinculados a proyectos anteriores, los cuales estaban funcionando de manera redundante, aunque cumpliendo el objetivo de negocio, por lo tanto, pueden crear obstáculos para la aplicación de este patrón ya que es más difícil de justificar el valor de negocio en este caso.

Principios soportados: Reusabilidad de servicios y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

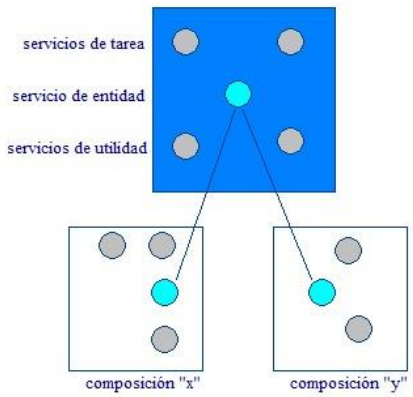


Figura 13: Los consumidores de servicios deben reutilizar la funcionalidad proporcionada por un único servicio independiente. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

⊕ Debido a la reutilización y recomposición de servicios primordialmente, lo cual elude la redundancia de lógica.

7.2.2.2.6. Nombre: Capas de servicios (Erl).

Sinopsis: En este patrón de diseño se recomienda agrupar los servicios del inventario por funcionalidad común, lo cual favorece la reusabilidad.

Nombre alternativo: Niveles de Servicios.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La definición arbitraria de servicios, gestionados por diferentes equipos del proyecto, suele producir inconsistencias de diseño y redundancia funcional inadvertida en todo el inventario de servicios.

Solución: Estructuración del inventario en dos o más capas de servicios lógicos responsables de abstraer la lógica en función a tipo de comportamiento similar (ver sección 2.4 – clasificación funcional).

Aplicación: Selección de modelos de servicios que luego conformen la base para las capas de servicio que se establecen como estándares de diseño y modelado.

Efectos: Impacto en la gestión de tiempo y recursos respecto al esfuerzo de análisis inicial

y la definición de estándares los cuales deben ser aceptados.

Principios soportados: Reusabilidad de servicios y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

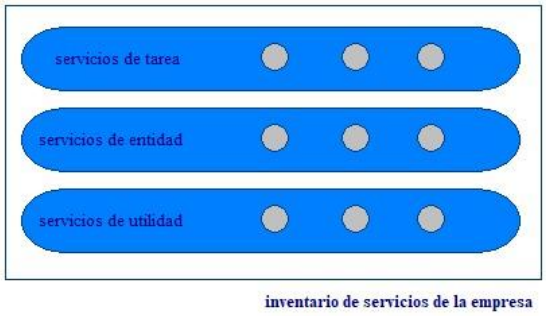


Figura 14: Servicios agrupados por comportamiento similar en capas lógicas, en este caso, el inventario se encuentra organizado como se indica en la sección 2.4 – clasificación funcional. A largo plazo, la gestión de los niveles de servicios suele estar asignada a equipos especializados transversales a los equipos del proyecto (especialización). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor confiabilidad:

⊕ En cuanto al comportamiento esperado de los servicios en ejecución, por niveles, los cuales pueden estar definidos con diferentes *SLA*.

Incremento de interoperabilidad:

⊕ En todo el inventario de servicios en cuanto a la reutilización y recomposición de éstos los cuales, al ser agrupados en niveles, son más fáciles de gestionar, ya que no dependen de definiciones de proyectos específicas, lo cual evita la redundancia de lógica.

7.2.2.2.7. Nombre: Normalización de servicios (Sin autor).

Sinopsis: Este patrón indica eludir la existencia de lógica redundante entre servicios del inventario a través del establecimiento adecuado del alcance de las responsabilidades.

Nombre alternativo: Estandarización de servicios.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: En la definición de los servicios, como parte de un inventario, existe un riesgo constante de que éstos posean responsabilidades solapadas, lo que dificulta la reusabilidad en general.

Solución: Diseño del inventario de servicios con un énfasis en el alcance de las responsabilidades individuales de éstos (evitando solapamiento).

Aplicación: El alcance de la responsabilidad funcional de los servicios se modela como parte de un proceso de análisis formal, el cual persiste durante el diseño y la gestión del inventario.

Efectos: Asegurarse de que existan límites entre las responsabilidades de los servicios y que ésos permanezcan alineados requiere análisis inicial adicional y un esfuerzo continuo de gestión.

Principios soportados: Autonomía de servicios.

Componentes de arquitectura involucrados: Inventario y límite entre servicios.

Diagrama de arquitectura:

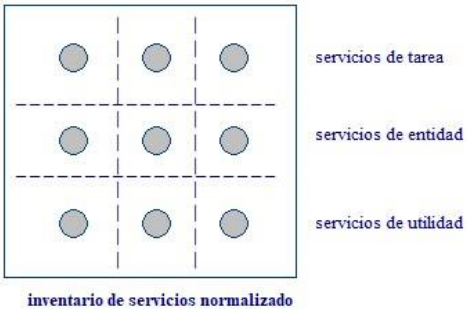


Figura 15: La entrega de servicios con límites claros y bien alineados logra la normalización en todo el inventario. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Mayor interoperabilidad:

⊕ Debido a la reducción de la cantidad de servicios del inventario en base a la definición adecuada del alcance de responsabilidades, lo cual promueve la reusabilidad.

7.2.2.3. Patrones de gestión del inventario:

7.2.2.3.1. Nombre: Convención de nombres (Erl).

Sinopsis: Este patrón de diseño indica la necesidad de usar expresiones estándar, en los convenios de los servicios del inventario, para evitar inconsistencias de interpretación

entre servicios.

Nombre alternativo: Expresiones de nombres estándar.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* primordialmente ya que en los *ReSTful* si se cumple con *RMM* nivel 2 (de mínima) sería lo mismo (ver sección 2.3.2. *ReST y ReSTful*). También, vale mencionar que en los servicios de estilo *ReST*, si la aplicación de este patrón se combina con *AMM* nivel 3 (ver sección 2.4) se pueden obtener resultados óptimos como *APIs* con operaciones normalizadas que esperan los consumidores.

Problema: Contratos de servicios *Web* que expresan características similares de diferentes maneras causando incoherencia de desarrollo y errores de interpretación.

Solución: Normalización de los contratos de servicios mediante el uso de convenciones de nombre.

Aplicación: Las convenciones de nombre se aplican a los convenios de servicio como parte de los procesos formales de análisis y diseño.

Efectos: El uso de convenciones de nombre globales implica la instauración de un nuevo estándar en toda la empresa, el cual debe aplicarse de manera consistente.

Principios soportados: Contrato de servicio estándar, abstracción y descubrimiento dinámico (éste último equivale a *RMM* nivel 3 para servicios de estilo *ReST,* mediante el uso de especificaciones

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

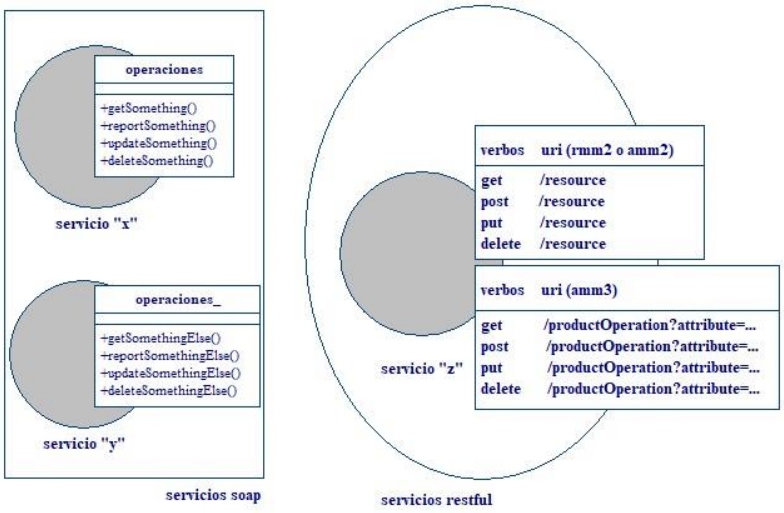


Figura 30: Por un lado, en *SOAP*, las expresiones en los contratos de los servicios se

encuentran alineadas en todos los servicios del inventario; por otro lado, en *ReST* se cumple de igual manera con la aplicación del *RMM* niveles 2 o 3 y con *AMM* nivel *3* para el establecimiento de *APIs* con operaciones estándar que esperan los clientes (resultado óptimo). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de confiabilidad:

- Referido a que los servicios del inventario se comportan de manera esperada ante operaciones declaradas de forma similar, lo que hace que éstos sean interpretados consistentemente por los clientes.

Mejora de interoperabilidad:

- En cuando a que la definición de funcionalidad estandarizada en los contratos de los servicios del inventario, mediante convenciones de nombres, evita la redundancia de lógica y promueve el reúso de servicios.

7.2.2.3.2. Nombre: Versionado estándar (Preston-Werner).

Sinopsis: Este patrón de diseño indica la instauración de reglas de versionado normalizadas aplicadas a todos los servicios del inventario para minimizar problemas de interoperabilidad.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Los servicios *Web* versionados de maneras diferentes causan dificultad en la gestión del inventario y problemas de interoperabilidad.

Solución: Establecimiento de reglas de versionado estándar para todos los servicios del inventario, incluyendo como expresar y como controlar las versiones.

Aplicación: Empleo de estándares de diseño, como *Semantic Versioning*[51] para expresar los números de versiones, con el fin de asegurar que todas las versiones de los servicios del inventario son establecidas de manera consistente. Además, los convenios de los servicios, *WSDL* para el caso de *SOAP* y *Swagger Spec o WADL* para *ReST*, deben ser

[51] En *Semantic Versioning* se declaran un conjunto de reglas y requisitos que dictan como los números de versiones son asignados e incrementados en la industria del desarrollo de *software*. Para más información ver: <https://semver.org/>.

administrados mediante una herramienta de control de versionado normalizada, como *Git*[52], en todo el inventario.

Efectos: El cumplimiento de estándares de versionado en todos los servicios del inventario implica mayor esfuerzo de gestión.

Principios soportados: Contrato de servicio estándar.

Componentes de arquitectura involucrados: Inventario y servicios.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Aumento de interoperabilidad:

⊕ Ya que los servicios que se versionan de acuerdo con la misma estrategia general son más fáciles de entender por los clientes, lo que fomenta la reutilización.

7.2.2.4. Patrones de centralización del inventario:

7.2.2.4.1. Nombre: Centralización de metadatos (Erl).

Sinopsis: Este patrón de diseño señala la centralización de metadatos de los componentes del inventario mediante un registro compartido el cual es usado para el descubrimiento de los servicios, lo cual evita la redundancia de funcionalidad.

Nombre alternativo: Metadatos centralizados.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Presencia de lógica de servicio redundante y desnormalización en el inventario como resultado de la implementación de tareas ya existentes en empresas de envergadura grande, donde existen muchos proyectos.

Solución: Publicación de los metadatos de manera centralizada en un registro de servicios que es usado para el descubrimiento de los servicios *Web* del inventario.

Aplicación: Instauración de un registro de servicios privado, como *UDDI* o *Eureka* (ver sección 2.2. Componentes), como parte central de la arquitectura del inventario el cual es respaldado por procesos formales de registro y descubrimiento.

Efectos: La tecnología para el registro y descubrimiento de los servicios *Web* debe ser madura y confiable, y su utilización y mantenimiento deben incorporarse en todos los

[52] *Git* es un sistema abierto de control de versionado distribuido diseñado para pequeños y grandes proyectos. Para más datos ver: <https://git-scm.com/>.

procesos de gestión y entrega de servicios. Vale la pena aclarar que, el cumplimiento de *RMM* nivel 3 en los servicios de estilo *ReST,* fomenta el auto descubrimiento de los servicios de manera distribuida mediante controles o *links*, lo cual, en conjunción al registro de servicios, provee mayor interoperabilidad que los servicios *SOAP*.

Principios soportados: Descubrimiento dinámico de los servicios.

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

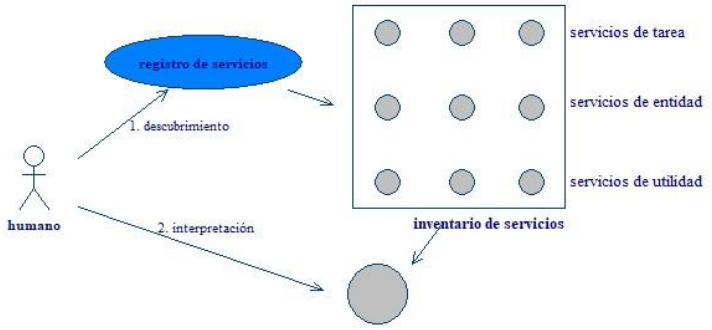


Figura 31: El proceso de descubrimiento e interpretación de los servicios durante el cual un ser humano ubica un servicio potencial a través de un registro del inventario y luego comprende como funciona. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que la centralización de los metadatos de los servicios mediante un registro del inventario impulsa la reutilización de los servicios disminuyendo la redundancia de funcionalidad existente.

7.2.2.4.2. Nombre: Centralización de políticas (Erl).

Sinopsis: Este patrón de diseño alude al requisito de aplicar políticas normalizadas de manera consistente en múltiples servicios evitando lógica redundante.

Nombre alternativo: Políticas centralizadas.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: Es más común en servicios *SOAP* mediante el uso de *WS-Policy* aunque en los *ReSTful* se puede utilizar *OPA* (ver sección 6.4.1).

  Problema: Las políticas que se aplican a múltiples servicios pueden introducir

redundancia e inconsistencia dentro de la lógica de éstos y sus contratos.

Solución: Identificación de políticas generales y específicas que luego son aplicadas a múltiples servicios de manera consistente, evitando redundancia y favoreciendo la reutilización.

Aplicación: Se recomienda un esfuerzo inicial de análisis para definir y establecer políticas reutilizables junto a un marco de referencias o *framework* de políticas adecuado.

Efectos: Los *frameworks* de políticas, como *WS-Policy* para *SOAP* y *OPA* para *ReST,* suelen introducir una sobrecarga de rendimiento y e imponer dependencias en tecnologías propietarias. Además, pueden existir conflictos entre las políticas generales y las específicas del servicio las cuales normalmente no son resueltas por los *frameworks*.

Principios soportados: Contrato de servicio estándar, acoplamiento débil de servicio y abstracción.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

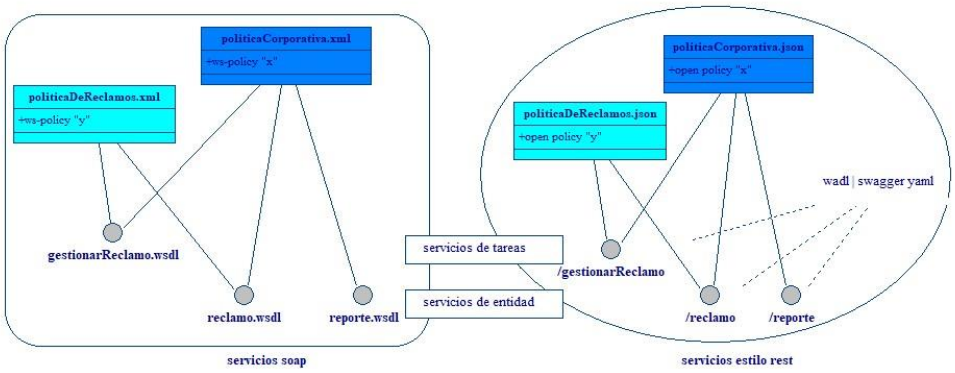


Figura 19: Se define una política global (politicaCorporativa) que se aplica a todos los servicios, y posteriormente, se crea una política de dominio adicional (politicaDeReclamos) la cual se aplica a algunos servicios. Esta nueva estructura de políticas elimina la redundancia del contenido de las políticas y garantiza un cumplimiento coherente de las mismas. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+-0))].

Disminución del rendimiento:

⊖ En cuanto al tiempo de ejecución de los servicios provocada por los *frameworks* de gestión de políticas los cuales sobrecargan el comportamiento operacional.

Aumento de seguridad:

⊕ Mediante el establecimiento de políticas de manera consistente lo cual permite

implementar mecanismos específicos a servicios independientes que así lo requieran (por ejemplo, mediante la utilización de un algoritmo de encriptación vinculado a una operación sensitiva aislada).

7.2.2.4.3. Nombre: Centralización de procesos de dominio (Erl).

Sinopsis: Este patrón de diseño indica la gestión centralizada de servicios de dominio, en entornos distribuidos, para fomentar la reutilización y la escalabilidad.

Nombre alternativo: Procesos centralizados.

Servicios afectados:

- Por funcionalidad: De tareas.
- Por estilo: Los motores *BPEL* (ver sección 2.5 – *Business Process Choregraphy*) suelen estar más vinculados a servicios *SOAP,* pero también sirven para orquestar los *ReSTful*.

Problema: Es difícil que los servicios extiendan funcionalidad y evolucionen si la lógica de los procesos de negocio se distribuye entre éstos de manera aislada.

Solución: Gestionar la lógica de negocio y la entrega de los servicios de dominio distribuidos desde una ubicación central.

Aplicación: Utilización de tecnologías de orquestación de servicios *Web* mediante plataformas *middleware,* como motores *BPEL,* las cuales centralizan la administración de los servicios de dominio.

Efectos: El establecimiento de una plataforma *middleware* introduce cambios importantes en la arquitectura y la infraestructura *IT* de la empresa.

Principios soportados: Servicios sin estado, autonomía y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y composición de servicios.

Diagrama de arquitectura:

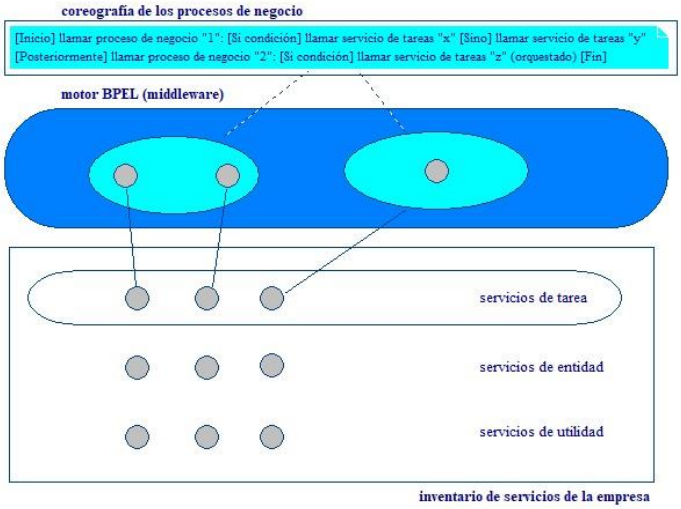


Figura 20: Los servicios de tareas siguen implementándose como servicios *Web* separados, pero como parte de una plataforma de orquestación o *middleware*, la cual administra la lógica de los procesos de negocios de forma centralizada (lo que da como resultado servicios de tareas "orquestados"). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

- ⊖ Debido a que los motores *BPEL* instauran una capa o *layer* adicional de procesamiento, lo cual suele ser excesivo en entornos de empresas con infraestructura *IT* mediana o pequeña.

Incremento de interoperabilidad:

- ⊕ Respecto a la orquestación de servicios de tareas mediante composiciones, gestionadas por la plataforma *middleware,* que promueven la reutilización y evolución de los servicios de manera centralizada.

7.2.2.4.4. Nombre: Centralización de reglas de dominio (Erl).

Sinopsis: Este patrón de diseño se refiere a la gestión centralizada de reglas de dominio, en entornos distribuidos de servicios, para evitar la redundancia de lógica de negocio y favorecer la aplicación de éstas consistentemente.

Nombre alternativo: Reglas de negocio centralizadas.

Servicios afectados:

- ➢ Por funcionalidad: De tareas principalmente, aunque la aplicación de reglas también puede afectar, en menor medida, a los de entidad y utilidad.

- Por estilo: *SOAP* y *ReSTful*.

Problema: Pueden existir problemas de redundancia y gestión cuando reglas similares de negocio se aplican a diferentes servicios de dominio.

Solución: Almacenamiento y la administración de las reglas comerciales mediante una extensión arquitectónica dedicada desde donde se pueden acceder y mantener de modo centralizado.

Aplicación: Empleo de un sistema o *engine* para la administración de reglas de negocio, como *BRMS* del *ESB* (ver sección 2.5 – *Integration Enterprise Service Bus*), el cual puede ser accedido a través de un servicio dedicado.

Efectos: Erigir una arquitectura adicional para gestión de reglas de dominio genera dependencia tecnológica la cual conlleva riesgo y, también, sobrecarga en el rendimiento de los servicios *Web*.

Principios soportados: Reusabilidad de servicios.

Componentes de arquitectura involucrados: Inventario de servicios.

Diagrama de arquitectura:

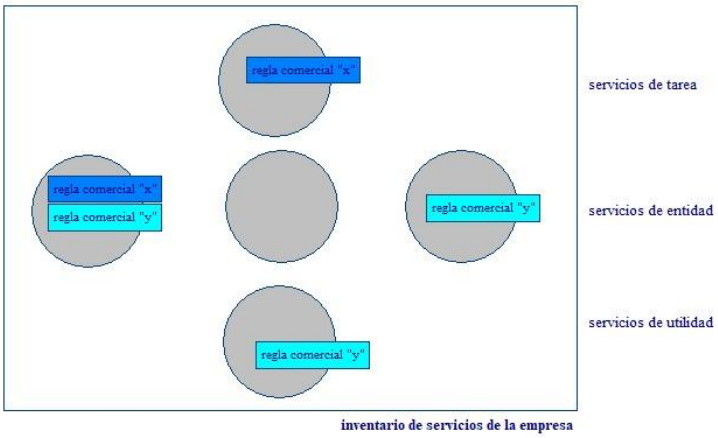


Figura 21: Solo dos reglas comerciales (x e y) se aplican en servicios empresariales diferentes y, en este caso, incluso en un servicio público (de tareas). Por lo tanto, un cambio global a cualquiera de las reglas afectará a múltiples servicios. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

- ⊖ Ya que las tecnologías de administración de reglas de negocio, como *BRMS,* sobrecargan el comportamiento original.

Aumento de interoperabilidad:

- En cuanto a la reusó de reglas de negocio de manera consistente lo cual disminuye la redundancia de lógica e impulsa la reutilización de los servicios.

7.2.2.4.5. Nombre: Centralización de esquemas (Erl).

Sinopsis: Este patrón de diseño establece el requisito de compartir esquemas de definición de servicios con responsabilidades similares para evitar la redundancia de éstos.

Nombre alternativo: Esquemas centralizados.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: En los servicios *SOAP,* los convenios *WSDL* pueden ser definidos a través de esquemas de datos *XSD* compartidos, y em los *ReSTful*, a nivel empresarial, se pueden usar *JSON Schema* o *YAML* en *Swagger* los cuales se pueden utilizar para definir el contenido común.

Problema: Presencia de esquemas de definición redundantes causada por convenios de servicios independientes que expresan características similares en cuanto a capacidades de procesamiento de negocio o datos parecidos. Además, la gestión de los esquemas se dificulta si existe reiteración de lógica en diferentes lugares (servicios).

Solución: Selección de esquemas de definición, ubicados físicamente separados a los convenios de servicios, los cuales son compartidos entre aquellos que poseen comportamiento o datos similares de manera consistente.

Aplicación: Se requiere un esfuerzo análisis inicial para establecer una capa de esquemas independientes a la capa de servicios con el fin de brindarle soporte a esta última.

Efectos: Es necesaria la gestión adecuada de esquemas compartidos debido a que múltiples servicios conforman sus convenios sobre la base de estas definiciones comunes.

Principios soportados: Contrato de servicio estándar y acoplamiento débil.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

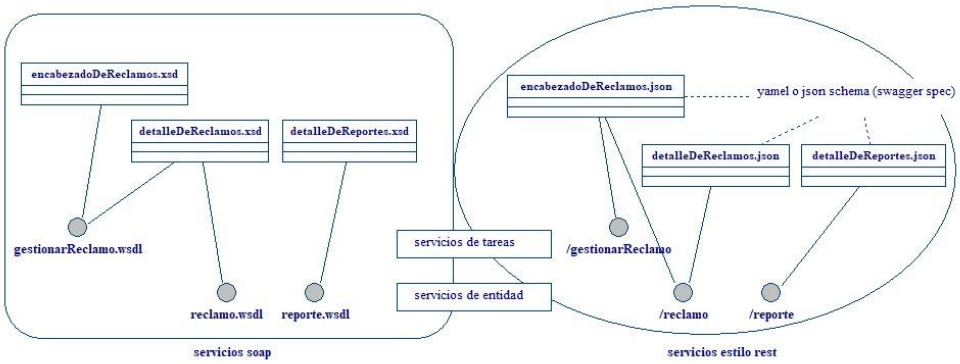


Figura 22: Contratos de servicios que comparten esquemas de definición comunes, los cuales poseen los mismos modelos de datos. Se fomenta la merma del contenido redundante, como así también, la creación de esquemas de menor tamaño. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+2))].

Incremento de interoperabilidad:

- ⊕ Debido a una mayor robustez respecto al ingreso de datos inválidos ya que los esquemas compartidos entre servicios, diseñados de forma adecuada, sirven para validar los tipos de datos modelados de manera centralizada (por ejemplo, estableciendo tamaños de los datos de ingreso o mediante verificaciones de estructura de los tipos de datos como el formato de las fechas en un rango determinado de tiempo).
- ⊕ Ya que el reúso de esquemas comunes disminuye la redundancia de funcionalidad fomentando la reutilización de los servicios *Web*.

7.2.2.5. Patrones de implementación del inventario:

7.2.2.5.1. Nombre: Recursos de infraestructura homogéneos (Erl).

Sinopsis: Este patrón de diseño señala el uso de recursos de infraestructura homogéneos para sortear la sobre arquitectura y la repetición tecnológica.

Nombre alternativo: Recursos de infraestructura únicos.

Servicios afectados:

- Por funcionalidad: De utilidad y entidad primordialmente, y los de tareas en menor medida.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Sobre arquitectura y redundancia tecnológica causada por implementaciones de servicios *Web* que introducen, innecesariamente, recursos de infraestructura dispares. Adicionalmente, se dificulta la gestión de las tecnologías diferentes.

Solución: Equipar la arquitectura subyacente con recursos de infraestructura y extensiones homogéneas, los cuales pueden ser usadas de manera recurrente por diferentes servicios.

Aplicación: Establecimiento de estándares de diseño, a nivel empresarial, para formalizar el uso requerido de recursos de arquitectura normalizados.

Efectos: Se puede disminuir la autonomía y escalabilidad de los servicios si existe una excesiva dependencia sobre los recursos de infraestructura compartidos.

Principios soportados: Autonomía de servicios.

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

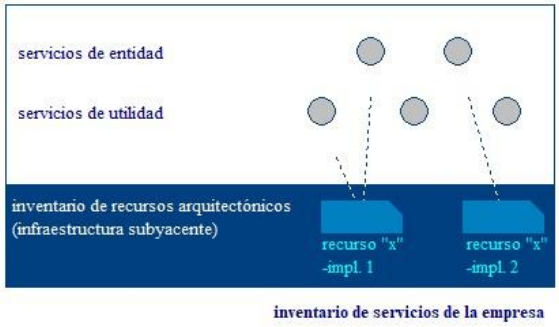


Figura 23: Distintos servicios *Web* utilizan, para propósitos similares, el mismo recurso de infraestructura normalizado (mediante implementaciones diferentes en algunos casos). Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (-1))].

Aumento de rendimiento:

- En cuanto al tiempo de ejecución de las funciones de los servicios a través del uso de recursos de infraestructura homogéneos.

Mayor confiabilidad:

- Ya que la autonomía de los servicios fomenta un mayor control sobre la tecnología subyacente, la cual, al estar normalizada, permite incrementar la predictibilidad del comportamiento de todos los servicios *Web* del inventario.

Disminución de interoperabilidad:

- Respecto a que, el mal reúso de tecnología compartida puede generar un excesivo nivel de dependencia hacia recursos de infraestructura propietarios; por ejemplo, en el caso de una dependencia de los servicios de utilidad en base a una tecnología de base de datos específica, propietaria, sin tener en cuenta el uso de motores *Object Related Mapping (ORM*) como *Hibernate* que sirven para independizarse de la tecnología subyacente.

7.2.2.5.2. Nombre: Capa de servicios de utilidad multi dominio (Erl).

Sinopsis: Este patrón de diseño indica la necesidad de usar la misma capa de servicios de utilidad en diferentes dominios de negocio, en el caso de una arquitectura con múltiples inventarios de servicios por unidad comercial.

Nombre alternativo: Utilidades multi dominio.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Redundancia innecesaria dentro de las capas de servicios de utilidad en arquitecturas donde existen múltiples inventarios de servicios de dominio para la gestión de unidades de negocio independientes.

Solución: Instauración de una capa de servicios de utilidad común, que abarque dos o más inventarios de servicios de dominio.

Aplicación: Definición de un conjunto de servicios de utilidades generales que debe ser normalizado en coordinación con los diferentes propietarios de los inventarios de servicios de dominio.

Efectos: La coordinación de una capa de servicios multi dominio puede requerir de un esfuerzo de gestión acentuado.

Principios soportados: Reusabilidad de servicios y capacidad de composición.

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

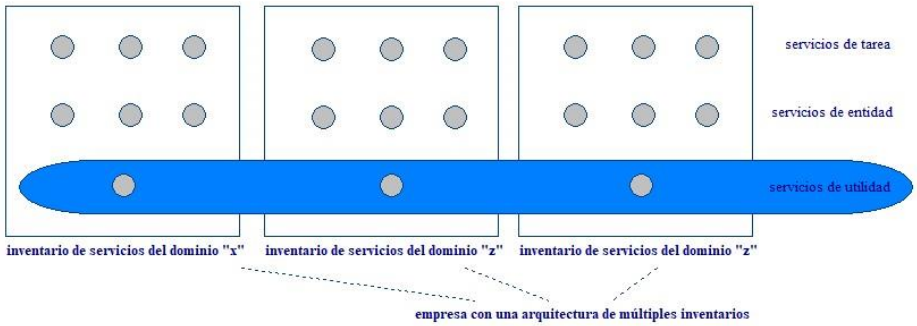


Figura 24: Capa de servicios de utilidad multi dominio, compuesta por un grupo de servicios *backend* normalizados, que abordan consideraciones transversales. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de interoperabilidad:

⊕ Debido a que la reducción de lógica redundante, mediante servicios de utilidad normalizados multi dominio, favorece la composición de los servicios impulsando la reutilización de éstos.

7.2.2.5.3. Nombre: Protocolo dual (Erl).

Sinopsis: Este patrón de diseño se utiliza en las empresas donde no es posible seleccionar una sola tecnología de comunicaciones y, a pesar de ser una solución semi federada, se evita el tener que mantener múltiples tecnologías heterogéneas.

Nombre alternativo: -.

Servicios afectados:

- Por funcionalidad: Principalmente los de tareas, aunque también de entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El patrón de diseño "protocolo único" advierte que todos los servicios se ajusten al uso de la misma tecnología de comunicaciones; sin embargo, es posible que un solo protocolo no pueda adaptarse a todos los requisitos de los servicios, lo que introduce limitaciones.

Solución: Establecimiento de una arquitectura de inventario de servicios diseñada para sustentar servicios basados en protocolos primarios y secundarios.

Aplicación: Creación de niveles de servicio primario y secundario que representan, en conjunto, las *URLs* o puntos de salida. Adicionalmente, todos los servicios deben estar sujetos a consideraciones de diseño de una arquitectura orientada a servicios siguiendo

pautas de normalización de convenios para minimizar el impacto de no utilizar una única tecnología de comunicaciones.

Efectos: El uso de este patrón suele impulsar la instauración de una arquitectura de inventario de servicios complicada debido a que requiere mayor esfuerzo de gestión (incluyendo el costo de mantenimiento de tecnologías de comunicación diferentes). Adicionalmente, si es administrado de manera deficiente, puede acentuar una alta dependencia a los servicios que conforman los puntos de salida, causando una disminución en la cantidad de consumidores potenciales y las oportunidades de reutilización.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción, autonomía y capacidad de composición.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

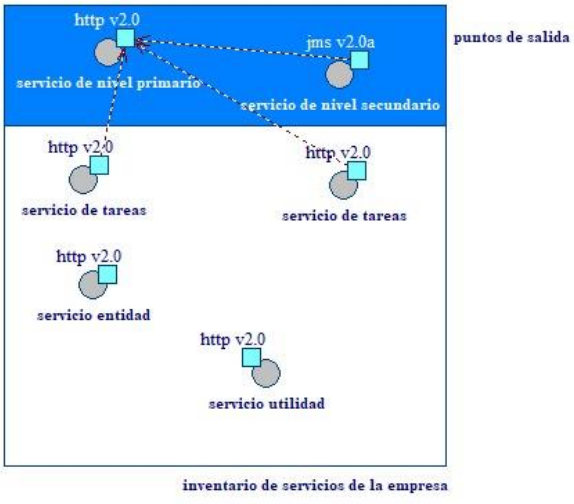


Figura 25: Servicios de nivel primario y secundario como puntos de salida. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

Aumento de rendimiento:

⊕ En cuanto a que se pueden establecer protocolos livianos de salida en el caso de ser posible, como *HTTP*, combinado con *HTTPS*.

Incremento de interoperabilidad:

⊕ Debido a una mayor capacidad respecto al número de solicitudes simultáneas que pueden ser recibidas, ya que, si la selección de uno de los protocolos duales de salida

así lo permite, se puede soportar la programación asincrónica mediante el uso de *JMS*.

Disminución de interoperabilidad:

- ⊖ Comparado con el uso del patrón "protocolo único" ya que el establecimiento de niveles duales como puntos de salida puede generar excesiva dependencia menguando la reutilización de servicios.

7.2.2.5.4. Nombre: Único punto de entrada del inventario (Erl, Richardson).

Sinopsis: Este patrón de diseño habla de la necesidad de implantar un único punto de entrada seguro para proteger todos los servicios de inventario.

Nombre alternativo: *API Gateway*.

Servicios afectados:

- Por funcionalidad: Primordialmente de tareas, aunque también puede considerarse de utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Por razones de seguridad y gestión no es conveniente exponer todos los servicios del inventario de una empresa a consumidores externos.

Solución: Erigir un único servicio como punto de entrada seguro el cual hace de escudo protector de todos los servicios del inventario. Éste único servicio debe abstraer las características principales del inventario de servicios, comúnmente haciendo de nexo entre los servicios de dominio relevantes y los consumidores externos.

Aplicación: El servicio que hace de enlace externo, como *Kong Gateway*[53], expone una interfaz similar a la de los servicios subyacentes, pero complementado con políticas u otras características que le permiten adaptarse a los requisitos de interacción del exterior.

Efectos: Los servicios que hacen de puntos de entrada incrementan la libertad de gestión de los servicios subyacentes, aunque también pueden aumentar el esfuerzo de gestión al introducir, en el inventario, una lógica de servicio y convenios redundantes.

Principios soportados: Contrato de servicio estándar, acoplamiento débil, abstracción.

Componentes de arquitectura involucrados: Inventario de servicios.

Diagrama de arquitectura:

[53] *Kong Gateway* es una *API* de servicios *open source* muy reconocida en el mercado que se utiliza de escudo para los servicios *ReSTful* y, también, los *SOAP* (versión paga). Para más información ver <https://konghq.com/kong/>.

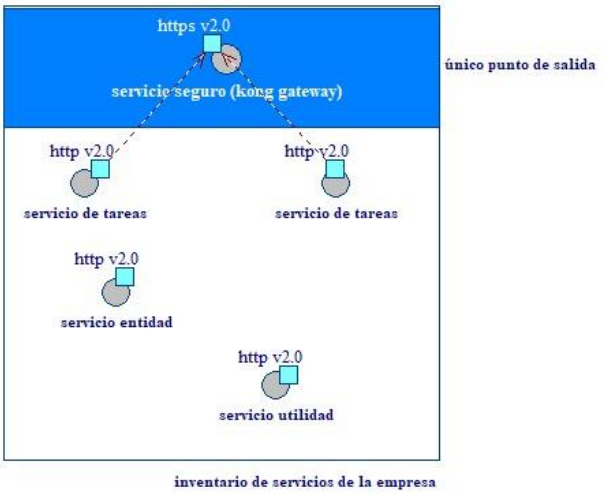


Figura 26: Introducción de un nuevo servicio que garantiza que los servicios nativos de inventario no se vean afectados por requisitos de los consumidores externos. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+1))].

Aumento de seguridad:

⊕ En cuanto a que el servicio que hace de *gateway* puede complementarse con políticas y características propias que le otorgan protección al inventario del exterior, sin afectar el rendimiento de los servicios subyacentes.

Incremento de interoperabilidad intrínseca:

⊕ Respecto a que los servicios, al ser protegidos, tienen la capacidad de interactuar con una gran variedad de consumidores externos, los cuales pueden no existen (consumidores potenciales).

7.2.2.5.5. Nombre: Repositorio para gestionar información de estado (Erl).

Sinopsis: Este patrón de diseño señala el requisito de establecer un lugar donde se puede persistir información de estado de negocio por un período de tiempo extendido, fomentando que los servicios eviten mantener este tipo de información.

Nombre alternativo: Repositorio de estado.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: El almacenamiento en memoria cache de grandes cantidades de datos de estado

para respaldar las actividades de las composiciones de servicios en ejecución puede ser excesivo, especialmente para tareas que se efectúan durante largos períodos de tiempo, lo que disminuye la escalabilidad.

Solución: La información de estado, de los servicios que así lo requieran, se pueden escribir temporalmente y luego recuperarse mediante un repositorio de estado dedicado, responsable de gestionar este tipo de información.

Aplicación: Instauración de un repositorio compartido o dedicado como parte de la arquitectura del inventario de servicios.

Efectos: El agregado de una funcionalidad de escritura y lectura aumenta la complejidad de diseño de los servicios y disminuye la performance.

Principios soportados: Servicios sin estado.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

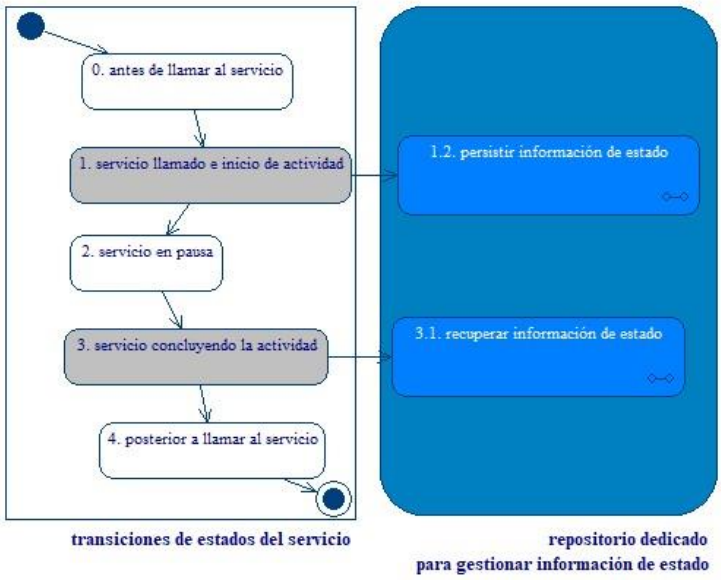


Figura 28: Al diferir información de estado a un repositorio compartido o dedicado, el servicio puede transicionar a una condición sin estado durante la pausa en la actividad, liberando temporalmente los recursos del sistema. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))].

Disminución de rendimiento:

- Ya que la presencia de un repositorio para escribir y recuperar datos almacenados requiere mayor tiempo en comparación al acceso de información en memoria.

Incremento de interoperabilidad:

⊕ Debido a que sortear el hecho de mantener información de estado en los servicios, durante períodos de tiempos prolongados, permite que éstos sean más escalables.

7.2.2.5.6. Nombre: Servicios que mantienen información de estado (Erl).

Sinopsis: Este patrón de diseño indica la creación de servicios de utilidad particulares que mantienen información de estado para evitar que las composiciones de servicios de dominio guarden este tipo de información, logrando mayor escalabilidad.

Nombre alternativo: Servicios con estado.

Servicios afectados:

- Por funcionalidad: De utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: La información de estado, asociada a la actividad de un servicio particular, impone complejidad en la gestión del tiempo de ejecución de las composiciones de servicios vinculadas, reduciendo así la escalabilidad.

Solución: Creación de servicios de utilidad especiales que mantienen y gestionan la información de estado de los servicios del inventario.

Aplicación: Instauración de servicios de utilidad que almacenan la información de estado de los servicios en memoria. Adicionalmente, estos servicios particulares, administran los datos del contexto de las actividades de los servicios.

Efectos: La implementación incorrecta de servicios de utilidad que mantienen información de estado puede generar cuellos de botellas.

Principios soportados: Servicios sin estado.

Componentes de arquitectura involucrados: Inventario y servicios.

Diagrama de arquitectura:

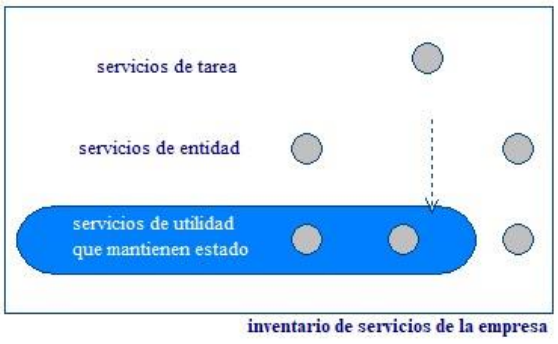


Figura 29: La gestión de información de estado se difiere a los servicios de utilidad, mejorando el escalamiento de los servicios de las capas superiores. Fuente propia

elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))].

Mayor rendimiento:

⊕ Comparado con el uso de un repositorio para la persistencia de información de estado, ya que los servicios de utilidad guardan la información en memoria.

Incremento de interoperabilidad:

⊕ Debido a que los servicios de dominio pueden escalar mejor ya que la responsabilidad de gestión de información de estado forma parte de los servicios de *backend*.

Disminución de interoperabilidad:

⊖ Si este patrón se utiliza de manera incorrecta, para datos perdurables (mantienen estado), la capacidad de los servicios disminuye en cuanto a recibir solicitudes simultáneas generando cuellos de botella.

7.2.2.5.7. Nombre: Grilla de servicios (Chappell).

Sinopsis: Este patrón de diseño indica el establecimiento de una grilla con servicios replicados, en los casos donde es necesario mantener el estado, para lograr escalabilidad y tolerancia a fallas.

Nombre alternativo: Cuadrícula de servicios.

Servicios afectados:

- Por funcionalidad: De tareas, entidad y utilidad.
- Por estilo: *SOAP* y *ReSTful*.

Problema: Los servicios *Web* que mantienen estado de negocio son propensos a ser cuellos de botella y poseer baja tolerancia a fallas, sobre todo cuando son expuestos a cargas de uso elevadas.

Solución: Instauración de una grilla de servicios a la cual se le puede diferir la información de estado, mediante el uso de mecanismos de replicación de memoria y redundancia junto una infraestructura de soporte adecuada, proveyendo alta escalabilidad y tolerancia a fallas.

Aplicación: Instalación de una tecnología de replicación o grilla, como *Amazon Grid Computing*[54], en la arquitectura del inventario de servicios de la empresa.

[54] *Amazon Grid Computing* es una tecnología de grilla propietaria que le brinda a los servicios de las empresas capacidades elásticas (dinámicas) para escalar ante picos de procesamiento. Ver más información en <https://aws.amazon.com/es/financial-services/grid-computing/>.

Efectos: Se requiere de una actualización significativa de la infraestructura y, en consecuencia, mayor esfuerzo en la gestión del inventario de servicios.

Principios soportados: Servicios sin estado.

Componentes de arquitectura involucrados: Empresa e inventario de servicios.

Diagrama de arquitectura:

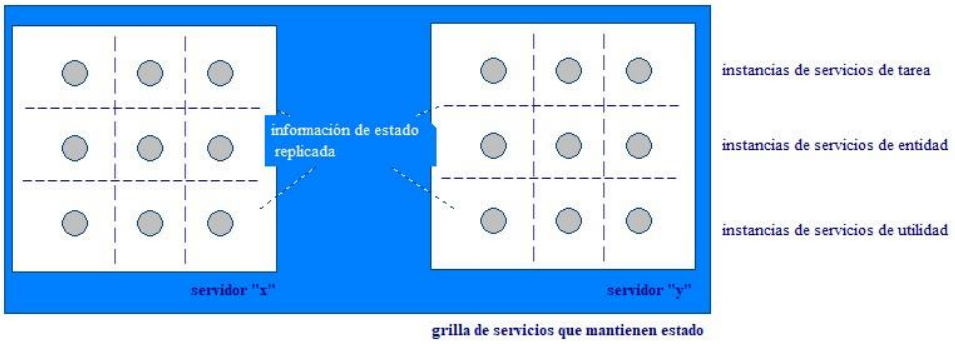


Figura 27: Una grilla de servicios que posee instancias replicadas con estado en diferentes servidores, lo que resulta en mayor escalabilidad y tolerancia a fallas. Fuente propia elaborada con *StarUML*.

Mejora de *QoS*: Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))].

Incremento de rendimiento:

⊕ Respecto a la velocidad en la cual se pueden completar las solicitudes de los servicios que mantienen el estado ante picos de procesamiento.

Disminución de performance:

⊖ Referido a un mayor uso de red necesario para la actualización de la información replicada (comparado con una solución que mantiene datos en memoria).

Mayor confiabilidad:

⊕ De acuerdo con la capacidad que tienen los servicios del inventario para realizar sus funciones debido a que una tecnología de grilla permite que la tolerancia a fallos sea alta, repartiendo la carga de procesamiento.

Incremento de interoperabilidad:

⊕ Ya que existe mayor capacidad en cuanto al número de solicitudes que se pueden procesar en simultáneo, ya que se disminuyen los cuellos de botella, causados por servicios que mantienen el estado, mediante la replicación de instancias que pueden tomar lo pedidos de manera elástica (auto escalamiento).

8. Utilización de patrones de servicios en un sistema referente de la actualidad

8.1. Relevamiento de dominio

El objetivo de este capítulo es hacer una evaluación *QoS* de los patrones de diseño en arquitecturas *SOA* de un sistema referente en la actualidad teniendo en cuenta los estilos *SOAP* y *ReSTful*. Para tal motivo se seleccionó el sistema *SABRE* ya que es un líder en el desarrollo *de software* a través de tecnología de servicios variados.

Semi-automated Business Research Environment (SABRE[55]) es el sistema de reserva o *computer reservation system (CRS)* más popular de la industria del transporte aeronáutico el cual se utiliza para almacenar, recuperar información y realizar transacciones relacionadas con viajes aéreos, hoteles, alquiler de automóviles u otras actividades. Éste conforma la base de los sistemas de las aerolíneas más importantes de la actualidad. Además, este sistema de reservas es conocido por proveer servicios *SOAP* siguiendo estándares de desarrollo (*WS-**), aunque también, en los años recientes se incluyó soporte a los de estilo más nuevo.

Es importante resaltar que el detalle de la arquitectura en este sistema puede considerarse extenso, con más de 300 *APIs* publicadas, por consiguiente, en esta tesis se hace foco en los *blueprints* de arquitectura del producto (ver sección 5.2.19) seleccionando las áreas y servicios esenciales para el análisis de *QoS*.

8.2. Relevamiento de diseño teniendo en cuenta los distintos estilos de servicios *Web* en *SABRE*

Las *APIs* de *SABRE*, también conocidas como servicios *Web* de *SABRE*, brindan acceso fácil, rápido y flexible a la funcionalidad y los productos del sistema de reserva de pasajes a través de Internet. Como resultado, se pueden integrar los productos y servicios del *CRS* con las aplicaciones de terceros añadiendo fácilmente las funciones necesarias para

[55] *SABRE* es propiedad de la corporación *SABRE* con sede en *Southlake, Texas*. Es el mayor proveedor de sistemas de distribución global para reservas aéreas en América del Norte. *American Airlines* fundó la compañía en 1960, y se escindió en 2000. Para más datos referirse a: <https://www.sabre.com>

vender viajes. Las *APIs* son ideales para los desarrolladores que desean crear o actualizar aplicaciones de reserva personalizadas.

Beneficios principales del uso de servicios *Web* de *SABRE*:

- Amplia gama de servicios de reserva disponible, incluyendo pasajes de avión, automóviles, hoteles e información relevante de pasajeros.
- Único punto de acceso oficial al sistema *SABRE*.
- Disminución de tiempos y costos de desarrollo, vinculados a integraciones con terceros, debido a la utilización de estándares de la industria de viajes para el intercambio de información, como el uso de tecnologías *XML* y *SOAP*.
- Acoplamiento débil entre el *CRS* y las aplicaciones de terceros y sus tecnologías subyacentes debido al uso de *APIs* para el acceso a información de reserva.
- Las funcionalidades publicadas por *SABRE,* mediante sus servicios *Web,* se pueden integrar de manera sencilla en aplicaciones personalizadas de terceros, como las páginas *Web* de las aerolíneas.
- Servicios de notificación de eventos o *Event Notification Services (ENS)* para los consumidores subscriptos a eventos específicos, como pérdida de valijas, lo cual brinda una mejor calidad de servicios.
- Servicios orquestados que realizan operaciones de negocios esenciales/más frecuentes en una sola llamada, lo que minimiza el tiempo de desarrollo y los gastos tecnológicos asociados.

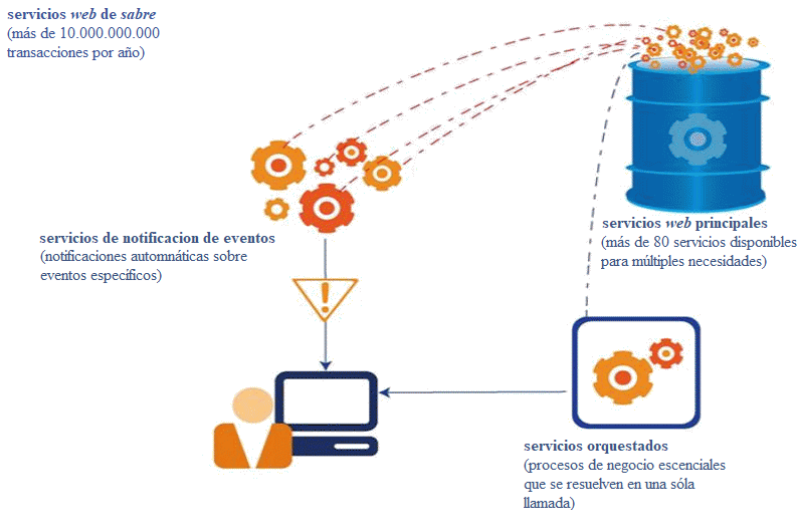


Figura 69: Organización de servicios *Web* en *SABRE*. Fuente adaptada en base a [73].

Servicios *Web* principales: Con más de 300 *APIs* disponibles, los clientes tienen acceso a

una amplia gama de servicios asociados a la reserva de viajes. Estas *APIs* ofrecen formas sencillas y económicas para crear aplicaciones de reserva personalizadas en la *Web*. Además, brindan acceso fácil y la flexibilidad para integrar información de viaje con sistemas de terceros dependientes de bases de datos propietarias. En definitiva, con las *APIs* de *SABRE* se simplifica la creación (o actualización) de un sitio *Web* de reservas de viajes satisfaciendo necesidades comerciales únicas.

Las *APIs* son compatibles con las tecnologías desarrolladas por *OpenTravel Alliance* utilizando estándares genéricos de tecnología de *API* respaldados por *Sun*, *IBM* y *Microsoft*. Estos estándares permiten que los diferentes sistemas se comuniquen entre sí mediante el uso de interfaces.

Servicios orquestados: Arriba de los servicios *Web* centrales se encuentran los servicios orquestados, los cuales son diseñados para ejecutar funciones de negocio esenciales/frecuentes en una sola llamada. Por lo tanto, estos servicios orquestados agilizan el desarrollo de aplicaciones y reducen la cantidad de procesamiento en los clientes. En definitiva, se puede realizar un desarrollo de aplicaciones más rápido cuando se usan servicios orquestado, ya que una llamada de servicio ejecuta múltiples servicios que un desarrollador/consumidor normalmente tendría que implementar individualmente.

Servicios de notificación de eventos o *Event Notification Services (ENS)*: Son un conjunto opcional de servicios que se ofrecen como parte las *APIs* de *SABRE*. Éstos permiten a los clientes suscribirse a ciertos eventos sobre información específica de viajes publicada a través de una infraestructura con tecnología de publicación y suscripción a través de Internet, por ejemplo, para informar sobre el extravío de valijas o condiciones climáticas desfavorables. *ENS* se basa en el estándar de la industria *WS-Eventing*. El modelo para *ENS* es: "No se preocupe por llamarnos; lo llamaremos" a medida que se detecten cambios en la información de viaje para la cual tiene una suscripción. *ENS* enviará automáticamente al cliente suscrito una notificación sobre el cambio al tema de la suscripción. Este tipo de tecnologías mejora sustancialmente el rendimiento general del sistema de reserva ante escenarios de información dirigida a múltiples consumidores al mismo tiempo.

Centro de recursos dedicados para desarrolladores o *Dedicated Developer Resource Center (DRC)*: Es un sitio *Web* dedicado a los desarrolladores/consumidores que proporciona información para el uso de las *APIs*, incluyendo detalles de como obtener acceso a los últimos servicios a través del repositorio conformado por un directorio completo de servicios y documentación relacionada para utilizarlos.

Inventarios de servicios *Web* agrupados en áreas de negocio:

- Servicios de viajes corporativos: Con el respaldo de la plataforma *SABRE*, estas *APIs* brindan una experiencia de compras y reservas basada en políticas de viajes.
- Servicios de datos globales: Acceso a información del mercado mundial del transporte aéreo mediante la recuperación de datos en lotes o a través de flujos de información específica respecto a vuelos y tarifas disponibles. Este catálogo general incluye los servicios de disponibilidad aérea, información de hoteles y los de tarifas.
- Servicios de información de disponibilidad aérea: Éstas *APIs* ofrecen la información de los aviones disponibles en cuanto a fechas y condiciones de viaje establecidas.
- Servicios de información de hoteles: Estos servicios *Web* proporcionan información descriptiva y de disponibilidad respecto los más de 176.000 hoteles registrados en *SABRE*.
- Servicios de notificación de eventos: Con estos servicios los clientes se subscriben para recibir mensajes vinculados a modificaciones en la reserva o en el viaje que ocurren en tiempo real, por ejemplo, cambio en las fechas de reserva debido a clima desfavorable o notificación de extravío de valijas.
- Servicios de información de las tarifas: Estas *APIs* proveen información de las tarifas aéreas en cuanto a la selección de itinerarios.

Tabla 5: El catálogo completo de servicios está conformado por 262 de estilo *SOAP* y 73 *ReSTful*, dando un total de 335 *APIs*. Fuente propia en base a [73].

Audiencia	Categoría	Función	Estilo
aerolíneas (193) hotelería (44) agencias de viaje (236)	transporte aéreo (188) autos (9) cruceros (1) área geográfica (1) hoteles (55) perfiles (31) trenes (29) manejo de sesión (8) utilidad (7)	restricciones de negocio (13) *check-in* (10) inteligencia (5) inventario (14) clientes leales (7) pagos (7) establecimiento de precios (23) perfiles (8) colas de mensajes (5) reportes (6)	*SOAP* (262) *ReSTful* (73)

		reserva (84) consulta (56) asientos (7) emisión de boletos (19) utilidad (24)	

Sobre la base del catálogo completo del producto junto al *blueprint* de arquitectura de las *APIs* de *SABRE,* se seleccionan los siguientes servicios esenciales en el dominio que sirven de muestra para realizar la valoración de *QoS* a nivel de componente:

S1. Nombre de servicio: Obtener detalles de vuelo (*Get Flight Details*).
Audiencia: Aerolíneas y agencias de viaje.
Categoría: Transporte aéreo.
Funcionalidad: Consulta.
Estilo: *SOAP*.
Sinopsis: Se utiliza para recuperar información detallada de un vuelo específico o *Flight Information (FLIFO)*.

Tabla 6: Interfaz:

WSDL (versión *OTA_AirFlifoLLS2.1.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:sws="https://webservices.sabre.com/websvc" xmlns:eb="http://www.ebxml.org/namespaces/messageHeader"
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext" targetNamespace="https://webservices.sabre.com/websvc">
        <wsdl:types>
                <xsd:schema>
                        <xsd:import namespace="http://webservices.sabre.com/sabreXML/2011/10"
schemaLocation="OTA_AirFlifoLLS2.1.0RQRS.xsd"/>
                        <xsd:import namespace="http://www.ebxml.org/namespaces/messageHeader"
schemaLocation="msg-header-2_0.xsd"/>
                        <xsd:import namespace="http://schemas.xmlsoap.org/ws/2002/12/secext"
schemaLocation="wsse.xsd"/>
                        <xsd:import namespace="http://services.sabre.com/STL/v01"
schemaLocation="STL_For_SabreProtocol_v.1.2.0.xsd"/>
                </xsd:schema>
        </wsdl:types>
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/OTA_AirFlifoLLS2.1.0RQ.wsdl -->
        <wsdl:binding name="OTA_AirFlifoSoapBinding" type="sws:OTA_AirFlifoPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="OTA_AirFlifoRQ">
```

```
                    <soap:operation soapAction="OTA_AirFlifoLLSRQ"/>
                    <wsdl:input>
                        <soap:header message="sws:OTA_AirFlifoInput" part="header" use="literal"/>
                        <soap:header message="sws:OTA_AirFlifoInput" part="header2" use="literal"/>
                        <soap:body parts="body" use="literal"/>
                    </wsdl:input>
                    <wsdl:output>
                        <soap:header message="sws:OTA_AirFlifoOutput" part="header" use="literal"/>
                        <soap:header message="sws:OTA_AirFlifoOutput" part="header2" use="literal"/>
                        <soap:body parts="body" use="literal"/>
                    </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="OTA_AirFlifoService">
                <wsdl:port name="OTA_AirFlifoPortType" binding="sws:OTA_AirFlifoSoapBinding">
                    <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 7: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Display FLIFO based upon a flight number.-->
<!--Equivalent Sabre host command: 2AA1771-->
<OTA_AirFlifoRQ Version="2.1.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <OriginDestinationInformation>
                <FlightSegment FlightNumber="1771">
                        <MarketingAirline Code="AA" FlightNumber="1771"/>
                </FlightSegment>
        </OriginDestinationInformation>
</OTA_AirFlifoRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<OTA_AirFlifoRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
        <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2019-11-04T17:04:52-06:00"/>
        </stl:ApplicationResults>
        <FlightInfo AirlineCode="AA" DepartureDateTime="01-15" FlightNumber="1505">
                <ScheduledInfo>
                        <FlightLeg LocationCode="DFW">
                                <DepartureDateTime Gate="D27" Scheduled="01-15T17:10" Terminal="D"/>
                        </FlightLeg>
                        <FlightLeg LocationCode="DEN">
                                <ArrivalDateTime BaggageClaim="17" Gate="A47" Scheduled="01-15T18:20"/>
                        </FlightLeg>
```

```
                </ScheduledInfo>
        </FlightInfo>
</OTA_AirFlifoRS>
```

Características de diseño principales:

- Servicio sincrónico de funcionalidad única (una sola operación) cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen diferentes formatos de fecha expresados en distintas secciones (*timeStamp* vs *dateTime*).
- Existe documentación.
- Autenticación mediante *token* de sesión.
- Posee un comportamiento por defecto en el caso de no incluirse la fecha de partida en la solicitud, es decir, cuando se ingresa solamente el número de vuelo se asumen el día y el mes de entrada teniendo en cuenta la fecha actual.

S2. Nombre de servicio: Disponibilidad aérea (*Air Availability*).

Audiencia: Aerolíneas y agencias de viaje.

Categoría: Transporte aéreo.

Funcionalidad: Inventario.

Estilo: *SOAP*.

Sinopsis: Buscar vuelos disponibles e información pertinente para una fecha determinada.

Tabla 8: Interfaz:

WSDL (versión *OTA_AirAvailLLS2.4.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/OTA_AirAvailLLS2.4.0RQ.wsdl -->
        <wsdl:binding name="OTA_AirAvailSoapBinding" type="sws:OTA_AirAvailPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="OTA_AirAvailRQ">
                        <soap:operation soapAction="OTA_AirAvailLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:OTA_AirAvailInput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirAvailInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:OTA_AirAvailOutput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirAvailOutput" part="header2"
```

```
use="literal"/>
                                        <soap:body parts="body" use="literal"/>
                                </wsdl:output>
                        </wsdl:operation>
                </wsdl:binding>
                <wsdl:service name="OTA_AirAvailService">
                        <wsdl:port name="OTA_AirAvailPortType" binding="sws:OTA_AirAvailSoapBinding">
                                <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                        </wsdl:port>
                </wsdl:service>
</wsdl:definitions>
```

Tabla 9: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <eb:MessageHeader xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" soapenv:mustUnderstand="0">
      <eb:From>
        <eb:PartyId />
      </eb:From>
      <eb:To>
        <eb:PartyId />
      </eb:To>
      <eb:CPAId>7TZA</eb:CPAId>
      <eb:ConversationId>V1@280b16ec-5eac-46c0-893f-c88f8e8cb632@310b16ec-5dad-46c0-893f-c88f8e8cb643@780b16ec-
5eac-46c0-893f-c88f8e8cb699</eb:ConversationId>
      <eb:Service>OTA_AirAvailLLSRQ</eb:Service>
      <eb:Action>OTA_AirAvailLLSRQ</eb:Action>
      <eb:MessageData>
        <eb:MessageId>mid:20001209-133003-2333@clientofsabre.com</eb:MessageId>
        <eb:Timestamp>2017-11-27T10:28:30Z</eb:Timestamp>
      </eb:MessageData>
    </eb:MessageHeader>
    <eb:Security xmlns:eb="http://schemas.xmlsoap.org/ws/2002/12/secext" soapenv:mustUnderstand="0">
      <eb:BinarySecurityToken>Shared/IDL:IceSess\/SessMgr:1\.0.IDL/Common/!ICESMS\/CERTG!ICESMSLB\/CRT.LB!-
3206393158292768888!455926!0</eb:BinarySecurityToken>
    </eb:Security>
  </soapenv:Header>
  <soapenv:Body>
    <OTA_AirAvailRQ xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Version="2.4.0">
      <OriginDestinationInformation>
        <FlightSegment DepartureDateTime="01-06">
          <DestinationLocation LocationCode="CMH" />
          <OriginLocation LocationCode="DFW" />
        </FlightSegment>
      </OriginDestinationInformation>
    </OTA_AirAvailRQ>
  </soapenv:Body>
```

```
</soapenv:Envelope>
```

Respuesta

```
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
        <soap-env:Header>
                <eb:MessageHeader eb:version="1.0" soap-env:mustUnderstand="1"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader">
                        <eb:From>
                                <eb:PartyId eb:type="URI"/>
                        </eb:From>
                        <eb:To>
                                <eb:PartyId eb:type="URI"/>
                        </eb:To>
                        <eb:CPAId>7TZA</eb:CPAId>
                        <eb:ConversationId>V1@280b16ec-5eac-46c0-893f-c88f8e8cb632@310b16ec-5dad-46c0-893f-
c88f8e8cb643@780b16ec-5eac-46c0-893f-c88f8e8cb699</eb:ConversationId>
                        <eb:Service>OTA_AirAvailLLSRQ</eb:Service>
                        <eb:Action>OTA_AirAvailLLSRS</eb:Action>
                        <eb:MessageData>
                                <eb:MessageId>2364315593094580151</eb:MessageId>
                                <eb:Timestamp>2017-11-27T16:28:30</eb:Timestamp>
                                <eb:RefToMessageId>mid:20001209-133003-
2333@clientofsabre.com</eb:RefToMessageId>
                        </eb:MessageData>
                </eb:MessageHeader>
                <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
                        <wsse:BinarySecurityToken valueType="String"
EncodingType="wsse:Base64Binary">Shared/IDL:IceSess\/SessMgr:1\.0.IDL/Common/!ICESMS\/CERTG!ICESMSLB\/CRT.LB!-
3206393158292768888!455926!0</wsse:BinarySecurityToken>
                </wsse:Security>
        </soap-env:Header>
        <soap-env:Body>
                <OTA_AirAvailRS Version="2.4.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
                        <stl:ApplicationResults status="Complete">
                                <stl:Success timeStamp="2017-11-27T10:28:29-06:00"/>
                        </stl:ApplicationResults>
                        <OriginDestinationOptions DestinationTimeZone="EST" OriginTimeZone="CST"
TimeZoneDifference="‡1">
                                <OriginDestinationOption RPH="1">
                                        <FlightSegment ArrivalDateTime="01-06T10:36" DOT_Ind="N"
DepartureDateTime="01-06T07:15" FlightNumber="2180" OnTimeInd="9" RPH="1" SmokingAllowed="false"
StopQuantity="0" eTicket="true">
                                                <BookingClassAvail AggregatedContent="false"
Availability="7" RPH="1" ResBookDesigCode="J"/>
<!-- sección abreviada, para ver la información completa referirse a
https://beta.developer.sabre.com/docs/soap_apis/air/search/Air_Availability -->
                                                <BookingClassAvail AggregatedContent="false"
Availability="4" RPH="21" ResBookDesigCode="N"/>
                                                <DaysOfOperation>
```

```
                                                    <OperationSchedule>
                                                        <OperationTimes>
                                                            <OperationTime Fri="true"
Mon="true" Sat="true" Sun="true" Thur="true" Tue="true" Weds="true"/>
                                                        </OperationTimes>
                                                    </OperationSchedule>
                                                </DaysOfOperation>
                                                <DestinationLocation LocationCode="CMH"/>
                                                <DisclosureAirline Code="**">
                                                    <Text>OPERATED BY /REPUBLIC AIRLINES DBA
UNITED EXPRESS</Text>
                                                </DisclosureAirline>
                                                <Equipment AirEquipType="E70"/>
                                                <FlightDetails Canceled="false" Charter="false"
CodeshareBlockDisplay="*" GroundTime="45" TotalTravelTime="272"/>
                                                <MarketingAirline Code="UA" FlightNumber="3613"
ParticipationLevel="DCA"/>
                                                <Meal MealCode="R"/>
                                                <OperatingAirline Code="**"/>
                                                <OriginLocation LocationCode="ORD"/>
                                            </FlightSegment>
                                        </OriginDestinationOption>
                                    </OriginDestinationOptions>
                                </OTA_AirAvailRS>
                            </soap-env:Body>
</soap-env:Envelope>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Firma digital en el encabezado del mensaje.
- El detalle de la respuesta es extenso, puede demorar por tal motivo, y existen valores de campos en formato texto plano.
- Se presentan diferentes formatos de fecha expresados en distintas secciones (*timeStamp* vs *dateTime*).
- Documentación existente.
- Autenticación mediante *token* de sesión.
- Las búsquedas de disponibilidad de vuelo se pueden refinar en cuanto a información de origen, destino, número de vuelo, aerolínea, tipo de vuelo (con conexión o directo), clase de reserva y número de asientos.

S3. Nombre de servicio: Reservar desde la disponibilidad aérea (*Book From Air Availability*).

Audiencia: Aerolíneas y agencias de viaje.

Categoría: Transporte aéreo.

Funcionalidad: Reserva.

Estilo: *SOAP*.

Sinopsis: Se utiliza para la venta rápida de vuelos con segmentos directos a partir de una respuesta de disponibilidad aérea.

Tabla 10: Interfaz:

WSDL (versión *ShortSellLLS2.1.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/ShortSellLLS2.1.0RQ.wsdl -->
        <wsdl:binding name="ShortSellSoapBinding" type="sws:ShortSellPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="ShortSellRQ">
                        <soap:operation soapAction="ShortSellLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:ShortSellInput" part="header" use="literal"/>
                                <soap:header message="sws:ShortSellInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:ShortSellOutput" part="header" use="literal"/>
                                <soap:header message="sws:ShortSellOutput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="ShortSellService">
                <wsdl:port name="ShortSellPortType" binding="sws:ShortSellSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 11: Ejemplo de solicitud y respuesta:

Solicitud

```
<ShortSellRQ Version="2.1.0">
        <OriginDestinationInformation>
                <FlightSegment DepartureDateTime="04-22" FlightNumber="2401" NumberInParty="1"
ResBookDesigCode="Y" Status="NN">
                        <DestinationLocation LocationCode="LAX"/>
                        <MarketingAirline Code="AA" FlightNumber="2401"/>
```

```
                    <OriginLocation LocationCode="DFW"/>
                </FlightSegment>
            </OriginDestinationInformation>
</ShortSellRQ
```

Respuesta

```
<ShortSellRS Version="2.1.0">
            <stl:ApplicationResults status="Complete">
                    <stl:Success timeStamp="2016-03-01T10:30:00-06:00"/>
            </stl:ApplicationResults>
            <OriginDestinationOption>
                    <FlightSegment ArrivalDateTime="11-22T08:50" DepartureDateTime="11-22T07:45"
FlightNumber="887" NumberInParty="2" ResBookDesigCode="Y" RPH="1" Status="HK">
                            <DestinationLocation LocationCode="LAS"/>
                            <MarketingAirline Code="AA" FlightNumber="887"/>
                            <OriginLocation LocationCode="DFW"/>
                    </FlightSegment>
            </OriginDestinationOption>
</ShortSellRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen secciones con formato de fecha variado (*timeStamp* vs *dateTime*).
- Se presenta documentación.
- Procesamiento rápido e información resumida.
- Autenticación mediante *token* de sesión.
- Sólo admite reservas de aerolíneas que proveen los vuelos de manera directa (no permite realizar ventas a través de aerolíneas asociadas).

S4. Nombre de servicio: Reservar segmento aéreo (*Book Air Segment*).

Audiencia: Aerolíneas y agencias de viaje.

Categoría: Transporte aéreo.

Funcionalidad: Reserva.

Estilo: *SOAP*.

Sinopsis: Se utiliza para reservar uno o más segmentos de vuelo.

Tabla 12: Interfaz:

WSDL (versión *OTA_AirBookLLS2.2.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
```

```
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/OTA_AirBookLLS2.2.0RQ.wsdl -->
        <wsdl:binding name="OTA_AirBookSoapBinding" type="sws:OTA_AirBookPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="OTA_AirBookRQ">
                        <soap:operation soapAction="OTA_AirBookLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:OTA_AirBookInput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirBookInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:OTA_AirBookOutput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirBookOutput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="OTA_AirBookService">
                <wsdl:port name="OTA_AirBookPortType" binding="sws:OTA_AirBookSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 13: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Book a one way itinerary.-->
<OTA_AirBookRQ Version="2.2.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <OriginDestinationInformation>
                <FlightSegment DepartureDateTime="2019-12-21T12:25" ArrivalDateTime="2019-12-21T13:25"
FlightNumber="1717" NumberInParty="2" ResBookDesigCode="Y" Status="NN">
                        <DestinationLocation LocationCode="LAS"/>
                        <Equipment AirEquipType="757"/>
                        <MarketingAirline Code="AA" FlightNumber="1717"/>
                        <OperatingAirline Code="AA"/>
                        <OriginLocation LocationCode="DFW"/>
                </FlightSegment>
        </OriginDestinationInformation>
</OTA_AirBookRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<OTA_AirBookRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.2.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
```

```
        <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2019-05-13T16:10:32-06:00"/>
        </stl:ApplicationResults>
        <OriginDestinationOption>
                <FlightSegment ArrivalDateTime="04-30T16:00" DepartureDateTime="04-30T14:40"
FlightNumber="0891" NumberInParty="002" ResBookDesigCode="Y" Status="NN" eTicket="true">
                        <DestinationLocation LocationCode="LHR"/>
                        <MarketingAirline Code="BA" FlightNumber="0891"/>
                        <OriginLocation LocationCode="SOF"/>
                </FlightSegment>
        </OriginDestinationOption>
</OTA_AirBookRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen secciones con formato de fecha variado (*timeStamp* vs *dateTime*).
- Se presenta documentación.
- Autenticación mediante *token* de sesión.
- Es una versión que permite más variantes de ingreso que en la venta rápida (ver servicio S3).
- Sirve para reservar vuelos de ida y vuelta.
- Se pueden hacer modificaciones sobre itinerarios abiertos.

S5. Nombre de servicio: Hacer reserva de hotel (*Book Hotel Reservation*).

Audiencia: Hotelería, agencias de viaje y aerolíneas.

Categoría: Hoteles.

Funcionalidad: Reserva.

Estilo: *SOAP*.

Sinopsis: Se usa para reservar una o más habitaciones de hotel asociado a un segmento del registro de viaje del pasajero o *Passenger Name Record* (*PNR*).

Tabla 14: Interfaz:

WSDL (versión *OTA_HotelResLLS2.2.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/OTA_HotelResLLS2.2.0RQ.wsdl -->
        <wsdl:binding name="OTA_HotelResSoapBinding" type="sws:OTA_HotelResPortType">
```

```
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="OTA_HotelResRQ">
                        <soap:operation soapAction="OTA_HotelResLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:OTA_HotelResInput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_HotelResInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:OTA_HotelResOutput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_HotelResOutput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="OTA_HotelResService">
                <wsdl:port name="OTA_HotelResPortType" binding="sws:OTA_HotelResSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 15: Ejemplo de solicitud y respuesta:

Solicitud

```
<OTA_HotelResRQ xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Version="2.2.0">
        <Hotel>
                <BasicPropertyInfo RPH="2"/>
                <Guarantee Type="GDPST">
                        <CC_Info>
                                <PaymentCard Code="AX" ExpireDate="2012-12" Number="1234567890"/>
                                <PersonName>
                                        <Surname>TEST</Surname>
                                </PersonName>
                        </CC_Info>
                </Guarantee>
                <RoomType NumberOfUnits="1"/>
                <SpecialPrefs>
                        <WrittenConfirmation Ind="true"/>
                </SpecialPrefs>
        </Hotel>
</OTA_HotelResRQ>
```

Respuesta

```
<OTA_HotelResRS xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01" Version="2.2.0">
        <stl:ApplicationResults status="Complete">
```

```
                <stl:Success timeStamp="2013-04-16T10:45:00-06:00"/>
        </stl:ApplicationResults>
        <Hotel NumberInParty="2" NumberOfUnits="1" SegmentNumber="4" Status="HK">
                <BasicPropertyInfo ChainCode="HG" HotelCityCode="DFW" HotelCode="24551"
HotelName="HOMEWOOD SUITES DAL">
                        <CancelPenalty PolicyCode="C24H"/>
                </BasicPropertyInfo>
                <Guarantee>XX4000000000006EXP 12 12-TEST</Guarantee>
                <RoomRates>
                        <Rate Amount="149.00" CurrencyCode="USD" DCS_AuxRateCode="A03LV2">
                                <AdditionalGuestAmounts>
                                        <AdditionalGuestAmount>
                                                <Charges CurrencyCode="USD"/>
                                        </AdditionalGuestAmount>
                                </AdditionalGuestAmounts>
                        </Rate>
                </RoomRates>
                <SpecialPrefs>
                        <WrittenConfirmation Ind="false"/>
                </SpecialPrefs>
                <Text>24551 ARR21NOV 24 HR CANCELLATION REQUIRED</Text>
                <Text>24551 ARR21NOV GUARANTEED BY CREDIT CARD</Text>
                <Text>HOMEWOOD SUITES BROWNSVILLE TEXAS</Text>
                <TimeSpan Duration="3" End="11-24" Start="11-21"/>
        </Hotel>
</OTA_HotelResRS>
```

Características de diseño principales:

- Servicio sincrónico con de funcionalidad única cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen secciones con formato de fecha variado (*timeStamp* vs *duration*).
- Se presenta documentación.
- Autenticación mediante *token* de sesión.
- Se hallan múltiples valores de campos con resultados en texto plano.
- Se admiten reservas en base a campos de ingreso variados como números de confirmación, viajero frecuente, identificación corporativa, cantidad de camas, invitados y muchas más alternativas.

S6. Nombre de servicio: Tarifa aérea de una tupla de ciudades (*Air Fare By City Pairs*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Establecimiento de precios.

Estilo: *SOAP*.

Sinopsis: Se utiliza para buscar tarifas asociadas con mercados particulares.

Tabla 16: Interfaz:

WSDL (versión *FareLLS2.9.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/FareLLS2.9.0RQ.wsdl -->
        <wsdl:binding name="FareSoapBinding" type="sws:FarePortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="FareRQ">
                        <soap:operation soapAction="FareLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:FareInput" part="header" use="literal"/>
                                <soap:header message="sws:FareInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:FareOutput" part="header" use="literal"/>
                                <soap:header message="sws:FareOutput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="FareService">
                <wsdl:port name="FarePortType" binding="sws:FareSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 17: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Perform a basic fare quote.-->
<!--Equivalent Sabre host command: FQDFWLAS21DEC-->
<FareRQ Version="2.9.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <OptionalQualifiers>
                <TimeQualifiers>
                        <TravelDateOptions Start="12-21"/>
                </TimeQualifiers>
        </OptionalQualifiers>
        <OriginDestinationInformation>
                <FlightSegment>
                        <DestinationLocation LocationCode="LAS"/>
                        <OriginLocation LocationCode="DFW"/>
```

```
                </FlightSegment>
        </OriginDestinationInformation>
</FareRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<FareRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.9.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
        <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2019-12-12T16:59:49-06:00"/>
        </stl:ApplicationResults>
        <FareBasis Code="Y0Z5NR" CurrencyCode="USD" RPH="1">
                <AdditionalInformation Acknowledgement="false" Constructed="false" FareType="P" FareVendor="A"
Net="false" Private="false" ResBookDesigCode="Y" RoutingNumber="21" SameDay="false" YY="false">
                        <AdvancePurchase>-/.</AdvancePurchase>
                        <Airline Code="NK"/>
                        <Cabin>E</Cabin>
                        <ExpirationDate>2013-12-31</ExpirationDate>
                        <Fare Amount="159.00"/>
                        <Fare Amount="318.00"/>
                        <OneWayRoundTrip Ind="X"/>
                        <Rule>
                                <Category>50</Category>
                                <Category>22</Category>
                        </Rule>
                </AdditionalInformation>
                <BaseFare Amount="147.91" CurrencyCode="USD"/>
                <Category22 Ind="true"/>
                <PassengerType Code="ADT"/>
        </FareBasis>
<!-- sección abreviada, para ver la información completa referirse a
https://beta.developer.sabre.com/docs/soap_apis/air/book/air_fare_by_city_pairs -->
        <HeaderInformation>
                <AdditionalVendorInfo>
                        <MarketingAirline Code="AA" NoReturnFlights="false" NumConnecting="25"
NumIntermediates="0" NumNonstops="12"/>
                        <MarketingAirline Code="DL" NoReturnFlights="false" NumConnecting="27"
NumIntermediates="0" NumNonstops="0"/>
                </AdditionalVendorInfo>
                <CurrencyCode>USD</CurrencyCode>
                <OriginDestinationOption>
                        <FlightSegment DepartureDateTime="2013-12-21">
                                <DestinationLocation LocationCode="LAS"/>
                                <OriginLocation LocationCode="DFW"/>
                        </FlightSegment>
                </OriginDestinationOption>
                <Text>//SEE FQHELP FOR INFORMATION ABOUT THE NEW FARE DISPLAYS//</Text>
                <Text>SURCHARGE FOR PAPER TICKET MAY BE ADDED WHEN ITIN PRICED</Text>
        </HeaderInformation>
</FareRS>
```

Características de diseño principales:

- Servicio sincrónico conformado por una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación.
- Autenticación mediante *token* de sesión.
- Se hallan algunos valores de campos con resultados en texto plano.
- El detalle de la respuesta es extenso y puede demorar por tal motivo. No está paginado.
- Se admiten cotizaciones de tarifas variadas especificando líneas aéreas preferidas, códigos de moneda, número de identificación corporativo, tarifa base y varios otros parámetros.

S7. Nombre de servicio: Determinar precio del itinerario de vuelo (*Price Air Itinerary*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Establecimiento de precios.

Estilo: *SOAP*.

Sinopsis: Se utiliza para establecer la cotización del itinerario de vuelo.

Tabla 18: Interfaz:

WSDL (versión *OTA_AirPriceLLS2.17.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/OTA_AirPriceLLS2.17.0RQ.wsdl -->
        <wsdl:binding name="OTA_AirPriceSoapBinding" type="sws:OTA_AirPricePortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="OTA_AirPriceRQ">
                        <soap:operation soapAction="OTA_AirPriceLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:OTA_AirPriceInput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirPriceInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:OTA_AirPriceOutput" part="header" use="literal"/>
                                <soap:header message="sws:OTA_AirPriceOutput" part="header2"
use="literal"/>
```

```
                    <soap:body parts="body" use="literal"/>
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="OTA_AirPriceService">
            <wsdl:port name="OTA_AirPricePortType" binding="sws:OTA_AirPriceSoapBinding">
                <soap:address location="https://webservices.havail.sabre.com/websvc"/>
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 19: Ejemplo de solicitud y respuesta:

Solicitud

```
<OTA_AirPriceRQ Version="2.17.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <PriceRequestInformation Retain="true">
            <OptionalQualifiers>
                <PricingQualifiers>
                    <PassengerType Code="ADT" Quantity="2"/>
                </PricingQualifiers>
            </OptionalQualifiers>
        </PriceRequestInformation>
</OTA_AirPriceRQ>
```

Respuesta

```
< OTA_AirPriceRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.17.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
        <stl:ApplicationResults status="Complete">
            <stl:Success timeStamp="2017-09-04T03:58:53-06:00"/>
        </stl:ApplicationResults>
        <PriceQuote>
            <MiscInformation>
                <BaggageInfo>
                    <SubCodeProperties RPH="1" SolutionSequenceNmbr="1">
                        <AncillaryFeeGroupCode>BG</AncillaryFeeGroupCode>
                        <BookingMethod>01</BookingMethod>
                        <CommercialNameofBaggageItemType>BAG MAX 23KG 51LB
208LCM 81LI</CommercialNameofBaggageItemType>
                        <DescriptionOne Code="23">
                            <Text>UP TO 50 POUNDS/23 KILOGRAMS</Text>
                        </DescriptionOne>
                        <DescriptionTwo Code="6C">
                            <Text>UP TO 81 LINEAR INCHES/208 LINEAR
CENTIMETERS</Text>
                        </DescriptionTwo>
                        <EMD_Type>4</EMD_Type>
                        <ExtendedSubCodeKey>0IZACBA</ExtendedSubCodeKey>
                        <RFIC>C</RFIC>
                        <SizeWeightInfo>
```

```
                                        <MaximumSizeInAlternate
Units="C">208</MaximumSizeInAlternate>
                                        <MaximumSize Units="I">81</MaximumSize>
                                        <MaximumWeightInAlternate
Units="K">23</MaximumWeightInAlternate>
                                        <MaximumWeight Units="L">50</MaximumWeight>
                                </SizeWeightInfo>
                                <SSR_Code>XBAG</SSR_Code>
                        </SubCodeProperties>
                        <SubCodeProperties RPH="2" SolutionSequenceNmbr="1">
                                <AncillaryFeeGroupCode>BG</AncillaryFeeGroupCode>
                                <AncillaryService SubGroupCode="CY">
                                        <Text>CARRY ON HAND BAGGAGE</Text>
                                </AncillaryService>
                                <CommercialNameofBaggageItemType>CARRYON HAND BAGGAGE
ALLOWANCE</CommercialNameofBaggageItemType>
                                <EMD_Type>4</EMD_Type>
                                <ExtendedSubCodeKey>0LNABBA</ExtendedSubCodeKey>
                                <RFIC>C</RFIC>
                        </SubCodeProperties>
                        <SubCodeProperties RPH="3" SolutionSequenceNmbr="1">
                                <AncillaryFeeGroupCode>BG</AncillaryFeeGroupCode>
                                <AncillaryService SubGroupCode="CY">
                                        <Text>CARRY ON HAND BAGGAGE</Text>
                                </AncillaryService>
                                <CommercialNameofBaggageItemType>CABIN BAG MAX 23KG 51LB
126LCM</CommercialNameofBaggageItemType>
                                <DescriptionOne Code="23">
                                        <Text>UP TO 50 POUNDS/23 KILOGRAMS</Text>
                                </DescriptionOne>
                                <DescriptionTwo Code="5C">
                                        <Text>UP TO 50 POUNDS/127 LINEAR
CENTIMETERS</Text>
                                </DescriptionTwo>
                                <EMD_Type>4</EMD_Type>
                                <ExtendedSubCodeKey>0MOACBA</ExtendedSubCodeKey>
                                <RFIC>C</RFIC>
                                <SizeWeightInfo>
                                        <MaximumSize Units="C">127</MaximumSize>
                                        <MaximumWeightInAlternate
Units="K">23</MaximumWeightInAlternate>
                                        <MaximumWeight Units="L">50</MaximumWeight>
                                </SizeWeightInfo>
                        </SubCodeProperties>
<!-- sección abreviada, para ver la información completa referirse a
https://beta.developer.sabre.com/docs/soap_apis/air/book/Price_Air_Itinerary -->
                        <FareCalculation>
                                <Text>DFW BA LON2072.00NUC2072.00END ROE1.00
XFDFW4.5</Text>
                        </FareCalculation>
```

```
<!-- sección abreviada, para ver la información completa referirse a
https://beta.developer.sabre.com/docs/soap_apis/air/book/Price_Air_Itinerary -->
                                        <BaseFare Amount="2072.00" CurrencyCode="USD"/>
                                        <Taxes TotalAmount="286.90">
                                                <Tax Amount="259.00" TaxCode="YQI"
TaxName="SERVICE FEE - INSURANCE" TicketingTaxCode="YQ"/>
                                        <Tax Amount="17.80" TaxCode="US2" TaxName="US
INTERNATIONAL TRANSPORTATIO" TicketingTaxCode="US"/>
                                        <Tax Amount="5.60" TaxCode="AY" TaxName="US
SECURITY FEE" TicketingTaxCode="AY"/>
                                        <Tax Amount="4.50" TaxCode="XF"
TaxName="PASSENGER FACILITY CHARGES" TicketingTaxCode="XF"/>
                                        </Taxes>
                                        <TotalFare Amount="2358.90" CurrencyCode="USD"/>
<!-- sección abreviada, para ver la información completa referirse a
https://beta.developer.sabre.com/docs/soap_apis/air/book/Price_Air_Itinerary -->
                                </ItinTotalFare>
                                <PassengerTypeQuantity Code="ADT" Quantity="1"/>
                                <PTC_FareBreakdown>
                                        <Cabin>Y</Cabin>
                                        <FareBasis Code="Y1N0O1C5" FareAmount="2072.00"
FarePassengerType="ADT" FareType="P" FilingCarrier="BA" GlobalInd="AT" Market="DFWLON"/>
                                        <FreeBaggageAllowance>PC001</FreeBaggageAllowance>
                                </PTC_FareBreakdown>
                        </AirItineraryPricingInfo>
                </PricedItinerary>
        </PriceQuote>
</OTA_AirPriceRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existe documentación.
- Se hallan algunos valores de campos con resultados en texto plano.
- El detalle de la respuesta es extenso y puede demorar por tal motivo. No está paginado.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.
- Para ejercitar esta *API* debe existir al menos un segmento de vuelo en el área de trabajo/sesión actual (por este motivo la solicitud puede omitir los datos del vuelo).
- Además de establecer el precio o *Price Quote (PQ)* y guardarlo en el registro de viaje del pasajero o *Passenger Name Record* (*PNR*) también se pueden especificar cotizaciones en base a viajeros frecuentes, comisiones y otras variantes.

S8. Nombre de servicio: Exhibir cotización de precio (*Display Price Quote*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Establecimiento de precios.

Estilo: *SOAP*.

Sinopsis: Se utiliza para mostrar la información referida a las tarifas que se encuentra almacenada en el registro de viaje del pasajero o *Passenger Name Record* (*PNR*).

Tabla 20: Interfaz:

WSDL (versión *DisplayPriceQuoteLLS2.5.2RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/DisplayPriceQuoteLLS2.5.2RQ.wsdl -->
        <wsdl:binding name="DisplayPriceQuoteSoapBinding" type="sws:DisplayPriceQuotePortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="DisplayPriceQuoteRQ">
                        <soap:operation soapAction="DisplayPriceQuoteLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:DisplayPriceQuoteInput" part="header"
use="literal"/>
                                <soap:header message="sws:DisplayPriceQuoteInput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:DisplayPriceQuoteOutput" part="header"
use="literal"/>
                                <soap:header message="sws:DisplayPriceQuoteOutput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="DisplayPriceQuoteService">
                <wsdl:port name="DisplayPriceQuotePortType" binding="sws:DisplayPriceQuoteSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 21: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!--Pre-Conditions: a PNR with at least one PQ record must be in the current SWS work area/session.-->
<!--Display an extended price quote record.-->
<!--Equivalent Sabre host command: *PQ1‡E-->
<DisplayPriceQuoteRQ Version="2.5.2" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <AirItineraryPricingInfo Extended="true">
                <Record Number="1"/>
        </AirItineraryPricingInfo>
</DisplayPriceQuoteRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Pre-Conditions: a PNR with at least one PQ record must be in the current SWS work area/session.-->
<DisplayPriceQuoteRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.5.2"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
        <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2019-05-03T13:14:27-05:00"/>
        </stl:ApplicationResults>
        <PriceQuote RPH="1">
                <MiscInformation>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/soap_apis/air/book/display_price_quote -->
                        <BaggageFees>
                                <Text>EMBARGOES-APPLY TO EACH PASSENGER</Text>
                        </BaggageFees>
                        <BaggageFees>
                                <Text>DFWLAS-AA</Text>
                        </BaggageFees>
                        <BaggageFees>
                                <Text>OVER 100 POUNDS/45 KILOGRAMS NOT PERMITTED</Text>
                        </BaggageFees>
                        <SignatureLine CreateDateTime="2013-08-26T12:11" CreationAgent="AW3"
HomePseudoCityCode="7TZA" PseudoCityCode="7TZA" Source="SYS"/>
                </MiscInformation>
                <PricedItinerary DisplayOnly="false" InputMessage="RQ">
                        <AirItineraryPricingInfo>
                                <FareCalculation>
                                        <Text>DFW AA LAS Q27.91Q27.91 897.67USD953.49END ZPDFW
XFDFW4.5</Text>
                                </FareCalculation>
                                <ItinTotalFare>
                                        <BaseFare Amount="953.49" CurrencyCode="USD"/>
                                        <Taxes TaxCode="XT" TotalAmount="82.41">
                                                <Tax Amount="71.51" TaxCode="US"/>
                                                <Tax Amount="3.90" TaxCode="ZP"/>
                                                <Tax Amount="2.50" TaxCode="AY"/>
                                                <Tax Amount="4.50" TaxCode="XF"/>
                                        </Taxes>
                                        <TotalFare Amount="1035.90" CurrencyCode="USD"/>
                                        <Warnings>
```

```
                                        <Warning ShortText="FARE NOT GUARANTEED UNTIL
TICKETED"/>
                                    </Warnings>
                                </ItinTotalFare>
                                <PassengerTypeQuantity Code="ADT" Quantity="01"/>
                                <PTC_FareBreakdown>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/soap_apis/air/book/display_price_quote -->
                                </PTC_FareBreakdown>
                                <ResTicketingRestrictions>11-22T07:55</ResTicketingRestrictions>
                            </AirItineraryPricingInfo>
                        </PricedItinerary>
                        <ResponseHeader>
                            <Text>VALIDATING CARRIER - AA</Text>
                            <Text>FARE SOURCE - ATPC</Text>
                            <Text>VALIDATING CARRIER-AA</Text>
                        </ResponseHeader>
                    </PriceQuote>
</DisplayPriceQuoteRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola operación cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación.
- Se hallan múltiples valores de campos con resultados en texto plano.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.
- Con esta *API* se pueden visualizar todos los registros de cotizaciones de precios (incluyendo históricos), uno específico o un rango. Además, se puede obtener un resumen de cotizaciones de precios desglosadas.

S9. Nombre de servicio: Reserva de vuelo orquestada (*Orchestrated Air Booking*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Reserva y establecimiento de precios.

Estilo: *SOAP*.

Sinopsis: Se usa para hacer reservas de vuelos, fijar precios y comparar tarifas de segmentos aéreos.

Tabla 22: Interfaz:

WSDL (versión *EnhancedAirBook3.10.0RQ*)
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:eab="http://services.sabre.com/sp/eab/v3_10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
        <wsdl:types>
                <xsd:schema>
                        <xsd:import namespace="http://services.sabre.com/sp/eab/v3_10"
schemaLocation="EnhancedAirBook3.10.0RQRS.xsd"/>
                        <xsd:import namespace="http://www.ebxml.org/namespaces/messageHeader"
schemaLocation="built-ins/msg-header-2_0.xsd"/>
                        <xsd:import namespace="http://schemas.xmlsoap.org/ws/2002/12/secext"
schemaLocation="built-ins/wsse.xsd"/>
                </xsd:schema>
        </wsdl:types>
<!-- sección abreviada, para ver la información completa referirse a:
http://files.developer.sabre.com/wsdl/sabreXML1.0.00/ServicesPlatform/EnhancedAirBook3.10.0RQ.wsdl -->
        <wsdl:binding name="EnhancedAirBookSoapBinding" type="sws:EnhancedAirBookPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="EnhancedAirBookRQ">
                        <soap:operation soapAction="EnhancedAirBookRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:EnhancedAirBookInput" part="header"
use="literal"/>
                                <soap:header message="sws:EnhancedAirBookInput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:EnhancedAirBookOutput" part="header"
use="literal"/>
                                <soap:header message="sws:EnhancedAirBookOutput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="EnhancedAirBookService">
                <wsdl:port name="EnhancedAirBookPortType" binding="sws:EnhancedAirBookSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 23: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<EnhancedAirBookRQ version="3.10.0" xmlns="http://services.sabre.com/sp/eab/v3_10" HaltOnError="true">
        <OTA_AirBookRQ>
                <OriginDestinationInformation>
                        <FlightSegment       DepartureDateTime="2019-06-03T12:30:00"       FlightNumber="1022"
```

```
NumberInParty="1" ResBookDesigCode="F" Status="NN" InstantPurchase="false">
                                        <DestinationLocation LocationCode="LAS"/>
                                        <MarketingAirline Code="US" FlightNumber="1022"/>
                                        <OriginLocation LocationCode="DFW"/>
                                </FlightSegment>
                        </OriginDestinationInformation>
                </OTA_AirBookRQ>
                <PostProcessing IgnoreAfter="true">
                        <RedisplayReservation/>
                </PostProcessing>
                <PreProcessing IgnoreBefore="false">
                        <UniqueID ID="JEGYLT"/>
                </PreProcessing>
</EnhancedAirBookRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<EnhancedAirBookRS xmlns="http://services.sabre.com/sp/eab/v3_10">
        <stl:ApplicationResults xmlns:stl="http://services.sabre.com/STL_Payload/v02_01" status="Complete">
                <stl:Success timeStamp="2019-02-08T07:34:51.136-05:00"/>
        </stl:ApplicationResults>
        <OTA_AirBookRS>
                <OriginDestinationOption>
                        <FlightSegment ArrivalDateTime="06-03T13:45" DepartureDateTime="06-03T12:55"
FlightNumber="1022" NumberInParty="001" ResBookDesigCode="F" Status="QF" eTicket="true">
                                        <DestinationLocation LocationCode="LAS"/>
                                        <MarketingAirline Code="US" FlightNumber="1022"/>
                                        <OriginLocation LocationCode="DFW"/>
                                </FlightSegment>
                        </OriginDestinationOption>
        </OTA_AirBookRS>
        <TravelItineraryReadRS>
                <TravelItinerary>
                        <CustomerInfo>
                                <Address>
                                        <AddressLine>SABRE TRAVEL</AddressLine>
                                        <AddressLine>3150 SABRE DRIVE</AddressLine>
                                        <AddressLine>SOUTHLAKE, TX US</AddressLine>
                                        <AddressLine>76092</AddressLine>
                                </Address>
                                <ContactNumbers>
                                        <ContactNumber LocationCode="DFW" Phone="817-555-1212-H-1.1"
RPH="001"/>
                                </ContactNumbers>
                                <PersonName WithInfant="false" NameNumber="01.01"
NameReference="ABC123" PassengerType="ADT" RPH="1">
                                        <GivenName>SP</GivenName>
                                        <Surname>TEST</Surname>
                                </PersonName>
                        </CustomerInfo>
                        <ItineraryInfo>
```

```
                    <ReservationItems>
                        <Item RPH="1">
                            <MiscSegment DayOfWeekInd="6" DepartureDateTime="12-
12" NumberInParty="01" SegmentNumber="0001" Status="GK" Type="OTH">
                                <OriginLocation LocationCode="FSG"/>
                                <Text>TEST</Text>
                                <Vendor Code="UA"/>
                            </MiscSegment>
                        </Item>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/soap_apis/air/book/orchestrated_air_booking -->
                </ReservationItems>
            </ItineraryInfo>
            <ItineraryRef AirExtras="false" ID="DJEJWQ" InhibitCode="U" PartitionID="AA"
PrimeHostID="1S">
                    <Source AAA_PseudoCityCode="7TZA" CreateDateTime="2015-01-22T08:27"
CreationAgent="AW5" HomePseudoCityCode="7TZA" PseudoCityCode="7TZA" ReceivedFrom="DEMO"
LastUpdateDateTime="2019-05-22T08:27" SequenceNumber="2"/>
            </ItineraryRef>
            <RemarkInfo>
                    <Remark RPH="001" Type="Historical">
                        <Text>TEST HISTORICAL REMARK</Text>
                    </Remark>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/soap_apis/air/book/orchestrated_air_booking -->
            </RemarkInfo>
            <SpecialServiceInfo RPH="001" Type="GFX">
                    <Service SSR_Code="OSI">
                        <Airline Code="UA"/>
                        <PersonName NameNumber="01.01">TEST/SP</PersonName>
                        <Text>TEST1</Text>
                    </Service>
            </SpecialServiceInfo>
            <SpecialServiceInfo RPH="002" Type="GFX">
                    <Service SSR_Code="OSI">
                        <Airline Code="UA"/>
                        <PersonName NameNumber="01.01">TEST/SP</PersonName>
                        <Text>TEST1</Text>
                    </Service>
            </SpecialServiceInfo>
            <OpenReservationElements/>
        </TravelItinerary>
    </TravelItineraryReadRS>
</EnhancedAirBookRS>
```

Características de diseño principales:

- Servicio de orquestación de tareas sincrónico con una sola operación cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen campos de fecha en diferentes formatos (*timeStamp* vs *dateTime*).

- Se presenta documentación.
- Se hallan valores de campos con resultados en texto plano (*remarks*).
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.
- Las principales tareas orquestadas por este servicio son la reserva del itinerario de vuelo, establecimiento del precio y cálculo de impuestos.

S10. Nombre de servicio: Cambio de reserva orquestado (*Exchange Booking*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Reserva y establecimiento de precio.

Estilo: *ReST*.

Sinopsis: Se utiliza para la actualización del itinerario y su cotización, así como la emisión del boleto de cambio.

Tabla 24: Interfaz:

YAML de *Swagger* (versión *Exchange Booking 1.0.0*)

```
< # https://developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking
---
swagger: '2.0'
info:
  title: Exchange Booking
  version: 1.0.0
  description: 'Orchestrated APIs bundle several functions and operations into a single API call thus maximizing development efforts, and providing operational efficiencies by calling multiple services to fulfill a desired workflow.'
  contact:
    name: Contact Us
    url: 'https://beta.developer.sabre.com/contact'
x-project-id: exchangebookingv100
host: api-crt.cert.havail.sabre.com
basePath: /
schemes:
  - https
consumes:
  - application/json
produces:
  - application/json
paths:
  /v1.0.0/exchange/booking:
    post:
      operationId: exchangeBooking
      parameters:
        - name: exchangeBookingRequest
```

```
      in: body
      required: true
      schema:
        $ref: '#/definitions/ExchangeBookingRequest'
    responses:
      '200':
        schema:
          $ref: '#/definitions/ExchangeBookingResponse'
        description: Success
      '400':
        description: |
          Bad Request

          - The request was not valid.
      '404':
        description: |
          Not Found

          - No results were found.
definitions:
  ExchangeBookingRequest:
    type: object
    properties:
      ExchangeBookingRQ:
        type: object
        description: Exchange Booking Request.
        required:
          - Itinerary
          - AutomatedExchanges
        properties:
          version:
            type: string
            example: 1.0.0
            description: Version of the payload message.
          Itinerary:
            type: object
            description: Itinerary.
            required:
              - id
            properties:
              id:
                type: string
                example: TNJBGO
                description: |
                  "id" is used to specify a record locator.
          Cancel:
            type: object
            description: Cancel.
            required:
              - Segment
```

```
        properties:
          Segment:
            type: array
            description: Segment.
            items:
              type: object
              required:
                - Number
              properties:
                Number:
                  type: integer
                  example: 1
                  description: |
                    "Number" is used to specify a particular segment number to cancel.
      PostProcessing:
        type: object
        required:
          - acceptIncompleteTransactions
          - EndTransaction
        properties:
          acceptIncompleteTransactions:
            type: boolean
            example: false
            description: |
              "acceptIncompleteTransactions" is used to accept a transaction where at least one PQR creation on a ticket number
failed.
          returnPQRInfo:
            type: boolean
            example: true
            description: |
              "returnPQRInfo" is used to instruct the system to return the details of generated Price Quote Reissue information in the
service response.
# sección abreviada, ver la información completa en <https://beta.developer.sabre.com/sites/default/files/sHhv0Kjq.yaml>
    Links:
      - rel: self
        href: 'https://api.sabre.com/v1.0.0/exchange/booking'
      - rel: linkTemplate
        href: 'https://api.sabre.com/<version>/exchange/booking'
securityDefinitions:
  oauth2_authentication:
    type: oauth2
    tokenUrl: 'https://api-crt.cert.havail.sabre.com/v2/auth/token'
    flow: application
    x-base64-encode-client-credentials: true
security:
  - oauth2_authentication: []
```

Tabla 25: Ejemplo de solicitud y respuesta:

Solicitud (*URI=/exchange/booking* – verbo *POST*)
{

```
  "ExchangeBookingRQ": {
   "version": "1.0.0",
   "targetCity": "G7HE",
   "Itinerary": {
    "id": "IJNPIE"
   },
   "Cancel": {
    "Segment": [
     {
      "Number": 1
     },
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
    ]
   },
   "AirBook": {
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
    "OriginDestinationInformation": {
     "FlightSegment": [
      {
       "ArrivalDateTime": "2019-05-03T08:19:00",
       "DepartureDateTime": "2019-05-03T06:00:00",
       "FlightNumber": "781",
       "NumberInParty": "1",
       "ResBookDesigCode": "G",
       "Status": "NN",
       "DestinationLocation": {
        "LocationCode": "PDX"
       },
       "MarketingAirline": {
        "Code": "AS",
        "FlightNumber": "781"
       },
       "OriginLocation": {
        "LocationCode": "LAS"
       }
      },
      {
       "ArrivalDateTime": "2019-05-03T10:20:00",
       "DepartureDateTime": "2019-05-03T09:30:00",
       "FlightNumber": "2172",
       "NumberInParty": "1",
       "ResBookDesigCode": "G",
       "Status": "NN",
       "DestinationLocation": {
        "LocationCode": "SEA"
       },
```

```
          "MarketingAirline": {
            "Code": "AS",
            "FlightNumber": "2172"
          },
          "OriginLocation": {
            "LocationCode": "PDX"
          }
        },
        {
          "ArrivalDateTime": "2019-05-03T17:30:00",
          "DepartureDateTime": "2019-05-03T11:45:00",
          "FlightNumber": "642",
          "NumberInParty": "1",
          "ResBookDesigCode": "G",
          "Status": "NN",
          "DestinationLocation": {
            "LocationCode": "DFW"
          },
          "MarketingAirline": {
            "Code": "AS",
            "FlightNumber": "642"
          },
          "OriginLocation": {
            "LocationCode": "SEA"
          }
        },
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
      ]
    }
  },
  "AutomatedExchanges": [
    {
      "ExchangeComparison": {
        "OriginalTicketNumber": "0277173836173",
        "PriceRequestInformation": {
          "OptionalQualifiers": {
            "PricingQualifiers": {
              "NameSelect": {
                "NameNumber": "1.1"
              }
            }
          }
        }
      },
      "PriceComparison": {
        "amountSpecified": 0,
        "AcceptablePriceIncrease": {
          "haltOnNonAcceptablePrice": true,
```

```
          "Amount": 10
        },
        "AcceptablePriceDecrease": {
          "haltOnNonAcceptablePrice": false,
          "Amount": 10
        }
       }
      }
    ],
    "PostProcessing": {
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
}
```


Respuesta


```
{
  "ExchangeBookingRS": {
    "ApplicationResults": {
      "status": "Complete",
      "Success": [
        {
          "timeStamp": "2019-02-07T05:24:07.797-06:00"
        }
      ]
    },
    "ExchangeConfirmation": [
      {
        "PQR_Number": "02",
        "PriceComparison": {
          "amountReturned": "0.00",
          "amountSpecified": "0.00"
        }
      }
    ],
    "PriceQuoteReissue": [
      {
        "PQR_Number": "2",
        "MiscInformation": {
          "BaggageFees": [
            {
              "Text": "BAG ALLOWANCE     -LASDFW-NIL/AS"
            },
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
          ],
          "SignatureLine": [
            {
              "CreationAgent": "ASP",
              "CreateDateTime": "2019-02-07T05:24",
```

```
      "HomePseudoCityCode": "G7HE",
      "PseudoCityCode": "G7HE",
      "Source": "SYS"
     }
    ]
   },
   "PricedItinerary": {
    "InputMessage": "WFRF",
    "AirItineraryPricingInfo": {
     "ExchangeDetails": {
      "DocNumber": "0277173836173",
      "CurrencyCode": "USD",
      "PQR_Status": "E",
      "TicketValue": "1222.01",
      "ChangeFeeInformation": [
       {
        "Amount": "N/A",
        "content": ""
       }
      ],
      "PersonName": {
       "Surname": "DZIK/MARCIN"
      },
      "Text": [
       "RESIDUAL AMT REFUNDABLE PER RULE 739.81"
      ],
      "TransactionInformation": [
       {
        "Amount": "0.00",
        "CurrencyCode": "USD",
        "Text": "EVEN"
       }
      ]
     },
     "FareCalculation": {
      "Text": [
       "LAS AS X/PDX AS X/SEA AS DFW200.93AS X/PDX AS LAS200.93USD40",
       "1.86END ZPLASPDXSEADFWPDX XFLAS4.5PDX4.5DFW4.5PDX4.5"
      ]
     },
     "ItinTotalFare": {
      "BaseFare": {
       "Amount": "401.86",
       "CurrencyCode": "USD"
      },
      "Taxes": {
       "TaxCode": "XT",
       "TotalAmount": "80.34",
       "Tax": [
        {
```

```
            "Amount": "30.14",
            "TaxCode": "US"
           },
           {
            "Amount": "21.00",
            "TaxCode": "ZP"
           },
           {
            "Amount": "11.20",
            "TaxCode": "AY"
           },
           {
            "Amount": "18.00",
            "TaxCode": "XF"
           }
          ]
         },
         "TotalFare": {
          "Amount": "482.20",
          "CurrencyCode": "USD"
         }
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/fulfill/exchange_booking/reference-
documentation#/default/exchangeBooking>
 },
 "Links": [
  {
   "rel": "self",
   "href": "https://api.sabre.com/v1.0.0/exchange/booking"
  },
  {
   "rel": "linkTemplate",
   "href": "https://api.sabre.com/<version>/exchange/booking"
  }
 ]
}
```


Características de diseño principales:

- Servicio de orquestación de tareas sincrónico que admite una sola *URI* y un único verbo (un solo método).
- La interfaz es realmente grande (2320 líneas de código) debido a que casi no se usan referencias comunes a esquemas (ejemplo, *$ref": "#/definitions/another_schema"*).
- Los campos de fecha se tratan como texto (ejemplo, *CreateDateTime:type:string*).
- Se hallan valores de campos con resultados en texto plano.
- Se presenta documentación concisa.
- Método de autenticación *OAuth2*.
- Se admite y se procesa un solo contenido de mensaje (*application/json*).

- Las principales tareas orquestadas por este servicio son la actualización del itinerario de vuelo, restablecimiento de precio y emisión de boleto de cambio.

S11. Nombre de servicio: Obtener aprobación de tarjeta de crédito (*Get Credit Card Approval*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Pagos.

Estilo: *SOAP*.

Sinopsis: Se utiliza para obtener la aprobación de la tarjeta de crédito a través del sistema *SABRE*.

Tabla 26: Interfaz:

WSDL (versión *CreditVerificationLLS2.2.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/CreditVerificationLLS2.2.0RQ.wsdl -->
        <wsdl:binding name="CreditVerificationSoapBinding" type="sws:CreditVerificationPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="CreditVerificationRQ">
                        <soap:operation soapAction="CreditVerificationLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:CreditVerificationInput" part="header"
use="literal"/>
                                <soap:header message="sws:CreditVerificationInput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:CreditVerificationOutput" part="header"
use="literal"/>
                                <soap:header message="sws:CreditVerificationOutput" part="header2"
use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="CreditVerificationService">
                <wsdl:port name="CreditVerificationPortType" binding="sws:CreditVerificationSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
```

```
        </wsdl:service>
</wsdl:definitions>
```

Tabla 27: Ejemplo de solicitud y respuesta:

Solicitud

```
<CreditVerificationRQ Version="2.2.0">
        <Credit>
                <CC_Info>
                        <PaymentCard AirlineCode="LH" Code="VI" ExpireDate="2017-04"
Number="4024007177454856"/>
                </CC_Info>
                <ItinTotalFare>
                        <TotalFare Amount="300" CurrencyCode="USD"/>
                </ItinTotalFare>
        </Credit>
</CreditVerificationRQ>
```

Respuesta

```
<CreditVerificationRS Version="2.2.0">
        <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2016-07-19T10:45:00-06:00"/>
        </stl:ApplicationResults>
        <Text>OK 123456 VERIFY CARDHOLDER SIGNATURE AND EXPIRATION DATE</Text>
</CreditVerificationRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Existen campos de fecha en diferentes formatos (*timeStamp* vs *dateTime*).
- Se presenta documentación.
- Se hallan valores de campos con resultados en texto plano (*text*).
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.

S12. Nombre de servicio: Consultar información de pago (*Query Payment Information*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Pagos.

Estilo: *SOAP*.

Sinopsis: Se utiliza para recuperar los detalles de la/s forma/s de pago.

Tabla 28: Interfaz:

WSDL (versión *PaymentQueryRQ3.0*)

```
<!-- La documentación del contrato no se encuentra disponible -->
```

Tabla 29: Ejemplo de solicitud y respuesta:

Solicitud

```
<?xml version="1.0" encoding="UTF-8"?>
<PaymentQueryRQ xmlns="http://services.sabre.com/PYMT/query/v01" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" SystemDateTime="2001-12-17T09:30:47Z" Version="3.9.0" xsi:schemaLocation="http://services.sabre.com/PYMT/query/v01 PaymentQueryRQ_v3.9.0.xsd">
  <POS StationNumber="12345678" ISOCountry="CO" CityCode="String" PseudoCityCode="String" ChannelID="AGY" SourceID="String" LocalDateTime="2001-12-17T09:30:47Z" OfficeCode="OF"> </POS>
  <MerchantDetail MerchantID="1S"/>
  <QueryRQ005 PaymentRef="01011362720893748009" RecordLocator="SSITAD" DocNumber="7382130350467"> </QueryRQ005>
</PaymentQueryRQ>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<PaymentQueryRS xmlns="http://services.sabre.com/PYMT/query/v01" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" SystemDateTime="2015-12-01T03:49:04" SabreTransactionID="12345447667970177080" Version="3.9.0" xsi:schemaLocation="http://services.sabre.com/PYMT/query/v01 PaymentQueryRQ_v3.9.0.xsd">
  <Result ResultCode="SUCCESS" Description="Successful Transaction"/>
  <QueryRS005>
    <PaymentDetail PaymentRef="01011362720893748009" FOP_Type="CC" FOP_Code="VI" FOP_Description="Visa" PNR_Locator="SSITAD" DocNumber="7382130350467">
      <AccountDetail MaskedAccountNumber="400555XXXXXX4403" ExpireDate="122013"/>
      <Vouchers VoucherType="AIR">
        <VoucherDetail Name="Date" Value="2016-02-05 18:22"/>
        <VoucherDetail Name="MID" Value="2794"/>
      </Vouchers>
      <Vouchers VoucherType="FEE">
        <VoucherDetail Name="COMPANY_NAME" Value="AGENCIA TURISMO"/>
        <VoucherDetail Name="TOTAL_AMOUNT" Value="30740.00"/>
      </Vouchers>
    </PaymentDetail>
  </QueryRS005>
</PaymentQueryRS>
```

Características de diseño principales:

- Servicio aparentemente sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación incompleta (falta el convenio del servicio).
- Autenticación mediante *token* de sesión.
- Se obtiene el detalle del pago mediante el ingreso de múltiples opciones de consulta como el número de identificación de la estación de venta, los atributos de referencia de pago (como *PNR* y número de documento), las opciones de pago y el comprobante de recibo electrónico.

S13. Nombre de servicio: Emitir boleto aéreo (*Issue Air Ticket*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Emisión de boletos.

Estilo: *SOAP*.

Sinopsis: Se utiliza para emitir boletos aéreos.

Tabla 30: Interfaz:

WSDL (versión *S13_AirTicketLLS2.12.0RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/AirTicketLLS2.12.0RQ.wsdl -->
        <wsdl:binding name="AirTicketSoapBinding" type="sws:AirTicketPortType">
                <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="AirTicketRQ">
                        <soap:operation soapAction="AirTicketLLSRQ"/>
                        <wsdl:input>
                                <soap:header message="sws:AirTicketInput" part="header" use="literal"/>
                                <soap:header message="sws:AirTicketInput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:header message="sws:AirTicketOutput" part="header" use="literal"/>
                                <soap:header message="sws:AirTicketOutput" part="header2" use="literal"/>
                                <soap:body parts="body" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="AirTicketService">
                <wsdl:port name="AirTicketPortType" binding="sws:AirTicketSoapBinding">
                        <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Tabla 31: Ejemplo de solicitud y respuesta:

Solicitud

```
<AirTicketRQ NumResponses="1" Version="2.12.0" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <OptionalQualifiers>
                <FlightQualifiers>
                        <VendorPrefs>
                                <Airline Code="XX"/>
                        </VendorPrefs>
                </FlightQualifiers>
```

```
                    <MiscQualifiers>
                        <Ticket Type="ETR"/>
                    </MiscQualifiers>
                    <PricingQualifiers>
                        <PriceQuote>
                            <Record Number="1"/>
                        </PriceQuote>
                    </PricingQualifiers>
            </OptionalQualifiers>
</AirTicketRQ>
```

Respuesta

```
<AirTicketRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.12.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
            <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2018-01-03T15:45:22-06:00"/>
            </stl:ApplicationResults>
            <Text>OK     636.50</Text>
            <Text>ETR MESSAGE PROCESSED</Text>
            <Text>NO COMMISSION WAS ENTERED</Text>
</AirTicketRS>
```

Características de diseño principales:

- Servicio sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación.
- Se hallan valores de campos con resultados en texto plano.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.
- Los boletos que se emiten pueden ser en papel, electrónicos o para uso exclusivo de la aerolínea.

S14. Nombre de servicio: Realizar *check-in* del pasajero (*Check In Passenger*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: *Check-in*.

Estilo: *SOAP*.

Sinopsis: Se emplea para registrar el ingreso del pasajero y emitir la tarjeta de embarque.

Tabla 32: Interfaz:

WSDL (versión *ACS_CheckInPassengerRQ3.0.0*)

```
<!-- La documentación del contrato no se encuentra disponible -->
```

Tabla 33: Ejemplo de solicitud y respuesta:

Solicitud

```
<ns9:ACS_CheckInPassengerRQ>
        <Itinerary>
                <Airline>EY</Airline>
                <Flight>371</Flight>
                <BookingClass>Y</BookingClass>
                <DepartureDate>2014-03-28</DepartureDate>
                <Origin>AUH</Origin>
        </Itinerary>
        <PrintingOptions>
                <PrintFormat>PNG</PrintFormat>
        </PrintingOptions>
        <PassengerInfoList>
                <PassengerInfo>
                        <LastName>LAM</LastName>
                        <PassengerID>EC998BE90001</PassengerID>
                </PassengerInfo>
        </PassengerInfoList>
</ns9:ACS_CheckInPassengerRQ>
```

Respuesta

```
<ns3:ACS_CheckInPassengerRS>
        <ItineraryPassengerList>
                <ns2:ItineraryPassenger>
                        <ns2:ItineraryDetail>
                                <ns2:Airline>EY</ns2:Airline>
                                <ns2:Flight>371</ns2:Flight>
                                <ns2:Origin>AUH</ns2:Origin>
                                <ns2:DepartureDate>2014-03-28</ns2:DepartureDate>
                                <ns2:DepartureTime>08:40AM</ns2:DepartureTime>
                                <ns2:DepartureGate>2</ns2:DepartureGate>
                                <ns2:AircraftType>340</ns2:AircraftType>
                        </ns2:ItineraryDetail>
                        <ns2:PassengerDetailList>
                                <ns2:PassengerDetail>
                                        <ns2:LineNumber>1</ns2:LineNumber>
                                        <ns2:LastName>LAM</ns2:LastName>
                                        <ns2:FirstName>TESTB</ns2:FirstName>
                                        <ns2:PassengerID>EC998BE90001</ns2:PassengerID>
                                        <ns2:BookingClass>Y</ns2:BookingClass>
                                        <ns2:Cabin>Y</ns2:Cabin>
                                        <ns2:Destination>BAH</ns2:Destination>
                                        <ns2:DepartureGate>2</ns2:DepartureGate>
                                        <ns2:Seat>15D</ns2:Seat>
                                        <ns2:SmokingRowFlag>N</ns2:SmokingRowFlag>
                                        <ns2:BoardingPassFlag>*</ns2:BoardingPassFlag>
                                        <ns2:PassengerType>F</ns2:PassengerType>
                                        <ns2:CheckInNumber>1</ns2:CheckInNumber>
                                        <ns2:BagCount>NB</ns2:BagCount>
```

```
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/sabre_sonic_apis/soap/check_in/check_in_passenger -->
                                        </ns2:PassengerDetail>
                                    </ns2:PassengerDetailList>
                                </ns2:ItineraryPassenger>
                            </ItineraryPassengerList>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/sabre_sonic_apis/soap/check_in/check_in_passenger -->
                            <Result messageId="ID-pichli009-58762-1395433336212-6-5439" timeStamp="2014-03-27T08:49:54.310-05:00">
                                <ns2:Status>Success</ns2:Status>
                                <ns2:CompletionStatus>Complete</ns2:CompletionStatus>
                                <ns2:System>ACS-BSO</ns2:System>
                            </Result>
</ns3:ACS_CheckInPassengerRS>
```

Características de diseño principales:

- Servicio aparentemente sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación incompleta (falta el contrato).
- Se hallan valores de campos con resultados en texto plano (alto porcentaje de información en texto plano).
- Imagen en formato *PNG* cifrada en *base64* como parte del valor de un campo.
- Autenticación mediante *token* de sesión.

S15. Nombre de servicio: Obtener reporte de ventas (*Retrieve Agent Sales Report*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Reportes.

Estilo: *SOAP*.

Sinopsis: Se utiliza para obtener los informes de ventas de los agentes o *Agent Sales Report* (*ASR*) y también para actualizarlos.

Tabla 34: Interfaz:

WSDL (versión *TKT_AsrServicesRQ1.2.1*)
<!-- La documentación del contrato no se encuentra disponible -->

Tabla 35: Ejemplo de solicitud y respuesta:

Solicitud
<?xml version="1.0" encoding="UTF-8"?> *<SOAP-ENV:Envelope* *xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"* *xmlns:xsd="http://www.w3.org/2001/XMLSchema"* *xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">*

```
<SOAP-ENV:Header>
 <ns1:MessageHeader
  ns1:version=""
  xmlns:ns1="http://www.ebxml.org/namespaces/messageHeader">
   <ns1:From>
    <ns1:PartyId>999999</ns1:PartyId>
   </ns1:From>
   <ns1:To>
    <ns1:PartyId>123123</ns1:PartyId>
   </ns1:To>
   <ns1:CPAId>IPCC</ns1:CPAId>
   <ns1:ConversationId>ABC123</ns1:ConversationId>
   <ns1:Service>TKT_AsrServicesRQ</ns1:Service>
   <ns1:Action>TKT_AsrServicesRQ</ns1:Action>
   <ns1:MessageData>
    <ns1:MessageId>1000</ns1:MessageId>
    <ns1:Timestamp>2015-10-28T11:11:11Z</ns1:Timestamp>
   </ns1:MessageData>
 </ns1:MessageHeader>
 <ns1:Security
  xmlns:ns1="http://schemas.xmlsoap.org/ws/2002/12/secext">
   <ns1:BinarySecurityToken>Shared/IDL:IceSess\/SessMgr:1\.0.IDL/Common/!ICESMS\/STSB!ICESMSLB\/STS.LB!-
3000398624414709619!1615713!0</ns1:BinarySecurityToken>
 </ns1:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
 <AccountingRQ
  version="1.2.1" xmlns="http://www.sabre.com/ns/Ticketing/AsrServices/1.0">
   <Header/>
   <SelectionCriteria>
    <AccountingReportOperation>display</AccountingReportOperation>
    <TicketingProvider>EY</TicketingProvider>
    <StationNumber>86493503</StationNumber>
    <EmployeeNumber>902606</EmployeeNumber>
    <ReportDate>2019-07-02</ReportDate>
   </SelectionCriteria>
 </AccountingRQ>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Respuesta

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
 <soap-env:Header>
  <eb:MessageHeader
   eb:version="1.0" soap-env:mustUnderstand="1"
    xmlns:eb="http://www.ebxml.org/namespaces/messageHeader">
   <eb:From>
    <eb:PartyId
     eb:type="URI">123123</eb:PartyId>
```

```
   </eb:From>
   <eb:To>
    <eb:PartyId
   eb:type="URI">999999</eb:PartyId>
   </eb:To>
   <eb:CPAId>IPCC</eb:CPAId>
   <eb:ConversationId>ABC123</eb:ConversationId>
   <eb:Service>TKT_AsrServicesRQ</eb:Service>
   <eb:Action>TKT_AsrServicesRS</eb:Action>
   <eb:MessageData>
    <eb:MessageId>488827669053840610</eb:MessageId>
    <eb:Timestamp>2019-07-02T18:35:07</eb:Timestamp>
    <eb:RefToMessageId>1000</eb:RefToMessageId>
   </eb:MessageData>
  </eb:MessageHeader>
  <wsse:Security
   xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
   <wsse:BinarySecurityToken
   EncodingType="wsse:Base64Binary"
   valueType="String">Shared/IDL:IceSess\/SessMgr:1\.0.IDL/Common/!ICESMS\/STSB!ICESMSLB\/STS.LB!-
3000398624414709619!1615713!0</wsse:BinarySecurityToken>
  </wsse:Security>
 </soap-env:Header>
 <soap-env:Body>
  <asr:AccountingRS
   isQueryStillRunning="false" version="1.2.1"
   xmlns:asr="http://www.sabre.com/ns/Ticketing/AsrServices/1.0">
   <asr:Header
    messageID="TKTHLI700-28439-371069546-1562092505398-787-asrws">
    <ns2:OrchestrationID
    seq="0" xmlns="http://www.sabre.com/ns/Ticketing/AsrServices/1.0"
    xmlns:ns2="http://services.sabre.com/STL/v01"
    xmlns:ns3="http://services.sabre.com/STL/Catalog/v01"
    xmlns:ns4="http://www.sabre.com/ns/Ticketing/TTL/1.0">TKTHLI700-28439-371069546-1562092505398-787-
asrws</ns2:OrchestrationID>
    <ns2:Results
    xmlns="http://www.sabre.com/ns/Ticketing/AsrServices/1.0"
    xmlns:ns2="http://services.sabre.com/STL/v01"
    xmlns:ns3="http://services.sabre.com/STL/Catalog/v01"
   xmlns:ns4="http://www.sabre.com/ns/Ticketing/TTL/1.0">
    <ns2:Success>
     <ns2:System>T2-ASR</ns2:System>
    </ns2:Success>
   </ns2:Results>
  </asr:Header>
  <AccountingHeader
   xmlns="http://www.sabre.com/ns/Ticketing/AsrServices/1.0"
   xmlns:ns2="http://services.sabre.com/STL/v01"
   xmlns:ns3="http://services.sabre.com/STL/Catalog/v01"
   xmlns:ns4="http://www.sabre.com/ns/Ticketing/TTL/1.0">
```

```
    <IssuingAgentEprCity>RKT</IssuingAgentEprCity>
    <IssuingAgentDieSine>WBT</IssuingAgentDieSine>
    <IssueAgentEmpNum>902606</IssueAgentEmpNum>
    <IssueCity>RKT</IssueCity>
    <IssueStation>86493503</IssueStation>
    <ReportOpenedDate>2019-07-02</ReportOpenedDate>
    <IsReportAutoClosed>false</IsReportAutoClosed>
   </AccountingHeader>
   <TotalsByCurrency
   xmlns="http://www.sabre.com/ns/Ticketing/AsrServices/1.0"
   xmlns:ns2="http://services.sabre.com/STL/v01"
   xmlns:ns3="http://services.sabre.com/STL/Catalog/v01"
    xmlns:ns4="http://www.sabre.com/ns/Ticketing/TTL/1.0">
    <CurrencyCode>AED</CurrencyCode>
    <TotalAmount
     amountText="25142"
     decimalPlace="0">25142</TotalAmount>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/sabre_sonic_apis/soap/ticketing/retrieve_agent_sales_report -->
    </TotalsByCurrency>
  </asr:AccountingRS>
 </soap-env:Body>
</soap-env:Envelope>
```

Características de diseño principales:

- Servicio aparentemente sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación incompleta (falta el convenio).
- Existen campos de fecha en diferentes formatos (*timeStamp* vs *dateTime*).
- Se hallan valores de campos con resultados en texto plano.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.

S16. Nombre de servicio: Origen y destino (*Origin And Destination*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Utilidad.

Estilo: *SOAP*.

Sinopsis: Se utiliza para recuperar la información de origen y destino de un viaje especificando sus segmentos.

Tabla 36: Interfaz:

WSDL (versión *OriginAndDestinationRQ1.0.3*)

```
<!-- La documentación del contrato no se encuentra disponible -->
```

Tabla 37: Ejemplo de solicitud y respuesta:

Solicitud

```
<OriginAndDestinationRQ>
        <Options Mode="Multiple" Pattern="Traveler"/>
        <Segment EndLocation="FRA" SegmentNumber="1" StartLocation="GRU">
                <Flight ArrivalDate="2017-06-22" ArrivalTime="0900" Carrier="LA" DepartureDate="2017-06-21"
DepartureTime="1330" SegmentStatus="HK"/>
        </Segment>
        <Segment EndLocation="MUC" SegmentNumber="2" StartLocation="FRA">
                <Flight ArrivalDate="2017-06-22" ArrivalTime="1030" Carrier="LH" DepartureDate="2017-06-22"
DepartureTime="0975" SegmentStatus="HK"/>
        </Segment>
        <Segment EndLocation="FRA" SegmentNumber="3" StartLocation="MUC">
                <Flight ArrivalDate="2017-06-22" ArrivalTime="1140" Carrier="LH" DepartureDate="2017-06-22"
DepartureTime="1080" SegmentStatus="HK"/>
        </Segment>
        <Segment EndLocation="JNB" SegmentNumber="4" StartLocation="FRA">
                <Flight ArrivalDate="2017-06-23" ArrivalTime="0510" Carrier="LH" DepartureDate="2017-06-22"
DepartureTime="1325" SegmentStatus="HK"/>
        </Segment>
        <Segment EndLocation="GRU" SegmentNumber="5" StartLocation="JNB">
                <Flight ArrivalDate="2017-06-24" ArrivalTime="0990" Carrier="LA" DepartureDate="2017-06-24"
DepartureTime="0660" SegmentStatus="HK"/>
        </Segment>
</OriginAndDestinationRQ>
```

Respuesta

```
<OriginAndDestinationRS>
        <JourneyData Type="International" RuleCarrier="LA" StopoverTime="13">
                <OnDData OriginCity="SAO" DestinationCity="MUC" OriginAirport="GRU"
DestinationAirport="MUC" Rule="5"/>
                <OnDData OriginCity="MUC" DestinationCity="JNB" OriginAirport="MUC"
DestinationAirport="JNB" Rule="3"/>
                <OnDData OriginCity="JNB" DestinationCity="SAO" OriginAirport="JNB" DestinationAirport="GRU"
Rule="9"/>
        </JourneyData>
</OriginAndDestinationRS>
```

Características de diseño principales:

- Servicio aparentemente sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación incompleta (falta el convenio).
- Existen campos de fecha en formatos comunes tanto en la solicitud como en la respuesta (*date* y *time*).
- Se hallan valores de campos con resultados en texto plano.
- Autenticación mediante *token* de sesión.

- Firma digital en el encabezado del mensaje.

S17. Nombre de servicio: Crear *token* de acceso (*Create Access Token*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Manejo de sesión.

Funcionalidad: Utilidad.

Estilo: *SOAP*.

Sinopsis: Se usa para solicitar el *token* de acceso a los servicios *SOAP*.

Tabla 38: Interfaz:

WSDL (versión *TokenCreateRQ1.0.0*)
<!-- La documentación del contrato no se encuentra disponible -->

Tabla 39: Ejemplo de solicitud y respuesta:

Solicitud

```
<SOAP-ENV:Envelope>
        <SOAP-ENV:Header>
                <eb:MessageHeader eb:version="1">
                        <eb:From>
                                <eb:PartyId>Client</eb:PartyId>
                        </eb:From>
                        <eb:To>
                                <eb:PartyId>SWS</eb:PartyId>
                        </eb:To>
                        <eb:CPAId>PCC</eb:CPAId>
                        <eb:ConversationId>1234</eb:ConversationId>
                        <eb:Service>Session</eb:Service>
                        <eb:Action>TokenCreateRQ</eb:Action>
                        <eb:MessageData>
                                <eb:MessageId>1234</eb:MessageId>
                                <eb:Timestamp>2015-01-01T00:00:00</eb:Timestamp>
                        </eb:MessageData>
                </eb:MessageHeader>
                <wsse:Security>
                        <wsse:UsernameToken>
                                <wsse:Username>USER</wsse:Username>
                                <wsse:Password>PASSWORD</wsse:Password>
                                <Organization>PCC</Organization>
                                <Domain>DOMAIN</Domain>
                        </wsse:UsernameToken>
                </wsse:Security>
        </SOAP-ENV:Header>
        <SOAP-ENV:Body>
                <sws:TokenCreateRQ Version="1.0.0"/>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Respuesta

```
<SOAP-ENV:Envelope>
        <SOAP-ENV:Header>
                <eb:MessageHeader eb:version="1">
                        <eb:From>
                                <eb:PartyId>Client</eb:PartyId>
                        </eb:From>
                        <eb:To>
                                <eb:PartyId>SWS</eb:PartyId>
                        </eb:To>
                        <eb:CPAId>PCC</eb:CPAId>
                        <eb:ConversationId>1234</eb:ConversationId>
                        <eb:Service>Session</eb:Service>
                        <eb:Action>TokenCreateRS</eb:Action>
                        <eb:MessageData>
                                <eb:MessageId>1234</eb:MessageId>
                                <eb:Timestamp>2015-01-01T00:00:00</eb:Timestamp>
                        </eb:MessageData>
                </eb:MessageHeader>
                <wsse:Security>
                        <wsse:BinarySecurityToken valueType="String" EncodingType="wsse:Base64Binary">
T1RLAQKvhOegyUujiZpE+uDAjHHmRfRmxRDDuJCPlszyUSmyhKGXWR0JAACgeveXEFWUPWzsmw9+Ihd9BSDYEtp
ikXHi8yJ9iW7vXgJpDNqnktLD4W8P7UP3zdra5szeuNXQB3yNbkjcK+3Vl1Gr/f8g00qU8ZhtzIBVz/PoD48GuaxNH7/Uq7
ZztI1bXu7ve9NEW6tVsp6qxbt9Jatn/B5IXf2t+T7S2l5QJU46kNg3r1H0ndhCp/pDwVT3FIo8sVSnWNZbIvUhrH6gQg**
</wsse:BinarySecurityToken>
                </wsse:Security>
        </SOAP-ENV:Header>
        <SOAP-ENV:Body>
                <sws:TokenCreateRS Version="1.0.0">
                        <sws:Success/>
                </sws:TokenCreateRS>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Características de diseño principales:

- Servicio aparentemente sincrónico con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados.
- Se presenta documentación incompleta (falta el convenio).
- Existen campos de fecha en formatos comunes tanto en la solicitud como en la respuesta (*timeStamp*).
- Autenticación mediante *token* de sesión.
- El *ID* de la conversación y el *token* de acceso de seguridad conforman el *ID* de conexión. Su retorno significa que la conexión al *API Gateway* de *SABRE* se ha establecido y se asigna un token de acceso al sistema de reservas. El cliente debe extraer y almacenar los valores presentes en las secciones *eb:CPAId*,

eb:ConversationId y todo el nodo *wsse:security@wsse:BinarySecurityToken* para su inclusión en solicitudes a servicios *SOAP* posteriores que utilizan la conexión específica.

S18. Nombre de servicio: Encolar mensaje (*Place Queue Message*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Colas de mensajes.

Estilo: *SOAP*.

Sinopsis: Se utiliza para almacenar un mensaje o registro de viaje del pasajero (*PNR*) en la cola designada, normalmente asociada a la estación aérea.

Tabla 40: Interfaz:

WSDL (versión *QueuePlaceLLS2.0.4RQ*)

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"
xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
targetNamespace="https://webservices.sabre.com/websvc">
<!-- sección abreviada, para ver la información completa referirse a:
http://webservices.sabre.com/wsdl/tpfc/QueuePlaceLLS2.0.4RQ.wsdl -->
<wsdl:binding name="QueuePlaceSoapBinding" type="sws:QueuePlacePortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="QueuePlaceRQ">
            <soap:operation soapAction="QueuePlaceLLSRQ"/>
            <wsdl:input>
                <soap:header message="sws:QueuePlaceInput" part="header" use="literal"/>
                <soap:header message="sws:QueuePlaceInput" part="header2" use="literal"/>
                <soap:body parts="body" use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:header message="sws:QueuePlaceOutput" part="header" use="literal"/>
                <soap:header message="sws:QueuePlaceOutput" part="header2" use="literal"/>
                <soap:body parts="body" use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="QueuePlaceService">
        <wsdl:port name="QueuePlacePortType" binding="sws:QueuePlaceSoapBinding">
            <soap:address location="https://webservices.havail.sabre.com/websvc"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Tabla 41: Ejemplo de solicitud y respuesta:

Solicitud
<QueuePlaceRQ Version="2.0.4"> *<QueueInfo>* *<QueueIdentifier Number="400" PrefatoryInstructionCode="11" PseudoCityCode="IPCC1"/>* *</QueueInfo>* *</QueuePlaceRQ>*
Respuesta
<QueuePlaceRS Version="2.0.4"> *<stl:ApplicationResults status="Complete">* *<stl:Success timeStamp="2015-09-07T20:30:23-06:00"/>* *</stl:ApplicationResults>* *<Text>OK 1030 NMIUTF</Text>* *</QueuePlaceRS>*

Características de diseño principales:

- Servicio síncrono (no usa *JMS* para el intercambio de mensajes) con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados. Podría ser diseñado/implementado de manera asíncrona.
- Se presenta documentación.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.

S19. Nombre de servicio: Acceder cola de mensajes (*Access Queue*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Colas de mensajes.

Estilo: *SOAP*.

Sinopsis: Se utiliza para acceder a una cola designada o navegar hacia una particular y visualizar sus mensajes.

Tabla 42: Interfaz:

WSDL (versión *QueueAccessLLS2.0.9RQ*)
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" *xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"* *xmlns:sws_xsd="http://webservices.sabre.com/sabreXML/2011/10" xmlns:sws="https://webservices.sabre.com/websvc"* *xmlns:eb="http://www.ebxml.org/namespaces/messageHeader" xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"* *targetNamespace="https://webservices.sabre.com/websvc">* <!-- sección abreviada, para ver la información completa referirse a: http://webservices.sabre.com/wsdl/tpfc/QueueAccessLLS2.0.9RQ.wsdl --> *<wsdl:binding name="QueueAccessSoapBinding" type="sws:QueueAccessPortType">* *<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>* *<wsdl:operation name="QueueAccessRQ">*

```
                    <soap:operation soapAction="QueueAccessLLSRQ"/>
                    <wsdl:input>
                        <soap:header message="sws:QueueAccessInput" part="header" use="literal"/>
                        <soap:header message="sws:QueueAccessInput" part="header2" use="literal"/>
                        <soap:body parts="body" use="literal"/>
                    </wsdl:input>
                    <wsdl:output>
                        <soap:header message="sws:QueueAccessOutput" part="header" use="literal"/>
                        <soap:header message="sws:QueueAccessOutput" part="header2" use="literal"/>
                        <soap:body parts="body" use="literal"/>
                    </wsdl:output>
                </wsdl:operation>
            </wsdl:binding>
            <wsdl:service name="QueueAccessService">
                <wsdl:port name="QueueAccessPortType" binding="sws:QueueAccessSoapBinding">
                    <soap:address location="https://webservices.havail.sabre.com/websvc"/>
                </wsdl:port>
            </wsdl:service>
</wsdl:definitions>
```

Tabla 43: Ejemplo de solicitud y respuesta:

Solicitud

```
<QueueAccessRQ Version="2.0.9" xmlns="http://webservices.sabre.com/sabreXML/2011/10"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <QueueIdentifier Number="200" PseudoCityCode="IPCC1"/>
</QueueAccessRQ>
```

Respuesta

```
<QueueAccessRS xmlns="http://webservices.sabre.com/sabreXML/2011/10" Version="2.0.9"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stl="http://services.sabre.com/STL/v01">
            <stl:ApplicationResults status="Complete">
                <stl:Success timeStamp="2017-09-11T10:00:00-06:00"/>
            </stl:ApplicationResults>
            <Line>
                <UniqueID ID="NGDEZP"/>
            </Line>
            <Paragraph>
                <Text>011  SEE REMARKS</Text>
                <Text>1.1BIDON/GWIDON</Text>
<!-- sección abreviada, para ver la información completa referirse a:
https://beta.developer.sabre.com/docs/soap_apis/management/queue/Access_Queue -->
                <Text>7TZA.7TZA*AW5 0319/06SEP12 NGDEZP H</Text>
            </Paragraph>
</QueueAccessRS>
```

Características de diseño principales:

- Servicio síncrono (no usa *JMS* para el intercambio de mensajes) con una sola función cuyos datos se encuentran estructurados en base a esquemas de representación versionados. Podría ser diseñado/implementado de manera asíncrona.

- Se presenta documentación.
- Existen múltiples datos en formato de texto plano.
- Autenticación mediante *token* de sesión.
- Firma digital en el encabezado del mensaje.

S20. Nombre de servicio: Servicios de viajes corporativos (*Corporate Travel Services*).

Audiencia: Aerolíneas y agencia de viajes.

Categoría: Transporte aéreo.

Funcionalidad: Reserva.

Estilo: *ReST*.

Sinopsis: Conjunto de servicios de dominio que permiten implementar una experiencia de compra y reserva basada en políticas de viaje.

Tabla 44: Interfaz:

YAML de *Swagger* (versión *Corporate Travel Services v1*)

```
swagger: '2.0'
x-project-id: corporatetravelservicesv1
securityDefinitions:
  oauth2_authentication:
    type: oauth2
    tokenUrl: 'https://api-crt.cert.havail.sabre.com/v2/auth/token'
    flow: application
    x-base64-encode-client-credentials: true
security:
  - oauth2_authentication: []
info:
  title: Corporate Travel Services
  version: v1
host: api-crt.cert.havail.sabre.com
basePath: /v1/tex/api
schemes:
- http
- https
paths:
  '/catalogs/{id}':
    get:
      description: |
        Gets a catalog of the shopping options available to the traveler.
      produces:
      - application/json
      tags:
      - Catalog
      operationId: readCatalog
      parameters:
```

```
    - name: id
      in: path
      description: The unique ID of a previously created `Catalog` that is to be retrieved.
      required: true
      type: string
      x-example: c23e107e-2b3a-4793-a442-217fdec4dcb2
    responses:
      '200':
        description: Successful response
        schema:
          $ref: '#/definitions/Catalog'
      '400':
        description: Bad Request
      '404':
        description: Catalog not found
      '422':
        description: Unable to process
      '500':
        description: Internal System Error
  /catalogs:
    post:
      description: |
        Creates a catalog of shopping options available to the traveler. After defining a set of search criteria, it completes the shopping request and assigns a unique ID to the resulting catalog.
      consumes:
      - application/json
      tags:
      - Catalog
      operationId: createCatalog
      parameters:
      - in: body
        name: shopRequest
        required: true
        schema:
          $ref: '#/definitions/ShopRequest'
      responses:
        '201':
          description: Catalog created
          headers:
            Location:
              description: Location
              type: string
        '500':
          description: Internal System Error
  '/carts/{id}':
    get:
      description: |
        Gets the contents of the `Cart`.
      produces:
      - application/json
```

```
# sección abreviada, ver la información completa en <https://beta.developer.sabre.com/sites/default/files/hOviB2jt.yaml>
  PaymentType:
    type: string
    enum:
      - TRAVELER_PAYMENT_CARD
      - SITE_PAYMENT_CARD
    description: Identifies the type of payment used.
  OpenResponseFormatType:
    type: string
    enum:
      - NUMBER
      - DATE
      - TEXT
    description: Identifies the format of the data in the open response.
# sección abreviada, ver la información completa en <https://beta.developer.sabre.com/sites/default/files/hOviB2jt.yaml>
```

Tabla 45: Ejemplos de solicitud y respuesta:

Solicitud ejemplo número 1 (*URI=/carts/{id}/bookings* - verbo *POST*)

```
{
  "travelerId": "site-SiteNameuser-UserName",
  "contactInfo": {
    "emailAddress": "johnsmith@work.com",
    "telephoneNumber": {
      "type": "HOME",
      "number": "2145551234"
    },
    "formattedPhoneNumber": {
      "countryCode": "1",
      "areaCode": "01205",
      "localNumber": "765-676-567",
      "extension": "1111"
    }
  },
  "components": {
    "bookableFlightItinerary": {
      "itineraryId": "fe9d3808-ad10-4bb3-ab5d-a3bf4acc2fe8.1426400367",
      "justifications": [
        {
          "type": "FLIGHT",
          "code": "CU"
        }
      ],
      "paymentCard": {
        "paymentCardId": "1",
        "billingAddress": {
          "addressLine1": "Ellen Ave",
          "addressLine2": "House nr. 10",
          "city": "Dallas",
          "stateProvinceCode": "TX",
          "postalCode": "75063",
```

```
        "countryCode": "US"
       },
       "pciData": {
        "securityCode": "0000",
        "typeCode": "VI",
        "ownerName": "Vernon Bear",
        "number": "4444333322221111",
        "expirationDate": "10/18"
       }
      },
      "paymentVouchers": [
       {
        "voucherCode": "UW2WWG"
       }
      ],
# sección abreviada, ver la información completa en
<https://beta.developer.sabre.com/docs/rest_apis/air/reservation/corporate_travel_services/reference-
documentation#/Booking/bookCart>
}
```

Respuesta ejemplo número 1

HTTP 201 – Booking created

Solicitud ejemplo número 2 (*URI=/sites* – verbo *POST*)

```
{
 "companyName": "Sabre",
 "administratorUserName": "subSiteAdministrator",
 "administratorEmail": "subSiteAdministrator@sabre.com",
 "subSite": {
  "subSiteName": "newsubsitename",
  "mainSiteName": "mainsitename",
  "templateSiteName": "templatesitename"
 }
}
```

Respuesta ejemplo número 2

HTTP 201 – Site created

Características de diseño principales:

- *Hub* de servicios de dominio con múltiples direcciones y métodos (varias *URIs* y verbos) para realizar diferentes acciones de negocio vinculadas a la gestión de políticas, reserva y compra de pasajes aéreos.
- Documentación existente.
- Existen campos de fecha en diferentes formatos (date vs date-time).
- Admiten comunicación mediante dos protocolos (*HTTPS y HTTP*).
- Método de autenticación *OAuth2*.
- La interfaz es realmente extensa ya que el contrato *CTS* conforma un *hub* de *APIS* de servicios (3699 líneas de código). En este caso si se establecen referencias comunes

a esquemas compartidos (ejemplo, *$ref: '#/definitions/Money'*) anque también se definen secciones propias del convenio principal (métodos de pago).

- Se admite y se procesa un solo contenido de mensaje (*application/json*).
- Este conjunto de *APIs* modernas puede ser utilizado tanto en experiencias móviles como de escritorio.
- Simplifican la administración de preferencias y políticas de viaje para hacer foco en la reserva y compra de experiencias personalizadas de acuerdo con distintas audiencias.

8.3. Evaluación de *QoS* en *SABRE* y propuesta de mejora

Para proceder con la valoración de *QoS* se establece el siguiente orden: Primero, evaluación de los patrones del inventario; segundo, valoración a nivel específico de las implementaciones teniendo en cuenta la muestra de servicios esenciales (ver secciones S1 - S10). Finalmente, se establece una propuesta basada en las fórmulas de cálculo de *QoS*.

Tabla 46: Patrones de diseño asociados al inventario de servicios en *SABRE*:

Patrones del inventario	Aplicación	Valoración de *QoS*
Inventario global de servicios en la empresa	Existencia del catálogo del producto donde se tratan las 335 *APIs* de *SABRE* como un todo dentro de la compañía, fomentando la reutilización de cada uno de los servicios.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1,5))]
Capas de servicios	Se agrupan a los servicios del inventario por funcionalidad común favoreciendo el reúso. Por un lado, existen los servicios de utilidad que se usan en los servicios de transporte aéreo y también en aquellos vinculados a hotelería, por otro lado, existen los de tareas que realizan acciones de negocio como hacer una reserva de vuelo, establecer el precio y emitir el boleto aéreo y, por último, existen los de entidad como aquel que representa el mapa de asientos o *seat map* de un avión en particular (manifiesto de la aeronave). En	[Formula cualitativa (capas)] + [Formula cualitativa (utilidades + tareas + entidades)] => {[Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]} + {[Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+

	consecuencia, debido a que se aplican los patrones "abstracción de utilidades, tareas empresariales y entidades de negocio" es necesario considerar la fórmula de cada uno de éstos también.	interoperabilidad (+1))] + [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))] + [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]} = [Rendimiento (-1) + confiabilidad (+3) + seguridad (+-0) (+ interoperabilidad (+4))]
Normalización de servicios	Los componentes en la nube se clasifican según la categoría de negocio, función comercial, audiencia a la que pertenecen y el estilo, lo que dentona que el inventario de servicios se diseña con énfasis en el alcance de las responsabilidades individuales de éstos.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Centralización de procesos de dominio	Mediante la presencia de tareas de orquestación como la reserva de vuelo orquestada y el cambio de reserva (ver servicios S9 y S10). También, la interfaz *CTS ReSTful* se puede considerar como el resultado de la aplicación de este patrón, aunque posiblemente no se haya implementado mediante un motor *BPEL*.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Centralización de esquemas	En los servicios *SOAP* y en los *ReSTful* en menor medida ya que los primeros poseen definiciones de esquemas *XSD* para evitar redundancia en convenios, mientras que en los segundos se observan servicios aislados y aunque en *CTS* existen referencias también hay redundancia en las definiciones (como en S10). En conclusión, el patrón de diseño se cumple para los servicios *SOAP* y en el caso de los *ReSTful* parcialmente, por consiguiente, se debe restar una parte representativa referente a los servicios nuevos.	[Formula cualitativa original (centralización de esquemas)] – [Formula cualitativa restando el faltante de los *ReSTful*] => [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+2))] - [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (-0.5))] = [Rendimiento (+-0) +

		confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1.5))]
Recursos de infraestructura homogéneos	El *blueprint* de arquitectura de *SABRE* evidencia que existe infraestructura uniforme para sortear la repetición de tecnología y evitar la sobre arquitectura. Se definen las tecnologías *SOAP* y *ReSTful* para los servicios principales (los segundos principalmente en las orquestaciones), y una infraestructura de mensajería para los de notificación de eventos (*ENS*).	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (-1))]
Protocolo dual	Ya que en los servicios *ReSTful* se admiten los protocolos de comunicación *HTTP* y *HTTPS* mientras que los *SOAP* sólo este último, siendo la tecnología de intercambio de mensajes segura el protocolo primario para los entornos operativos.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))]
Único punto de entrada del inventario	Presencia del *API Gateway* de seguridad como único punto de entrada del inventario para proteger el acceso a los servicios publicados en el catálogo de *SABRE*.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+1))]
Grillas de servicios	Existencia de réplicas de servicios, en los casos donde es necesario mantener información de estado, para lograr escalabilidad y tolerancia a fallas.	Formula cualitativa = [Rendimiento (+1 -1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]
Repositorio para gestionar información de estado	Para los casos donde es necesario almacenar información de estado durante períodos de tiempo prolongados como en los servicios que acceden a colas de mensajes de las estaciones aéreas (ver servicios S18 y S19).	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Versionado estándar	Todos los servicios de *SABRE* siguen una semántica de versionado X.Y.Z, donde la primera letra significa un cambio mayor o sustancial en el contrato o la implementación, la segunda representa un cambio menor normalmente vinculado a funcionalidad y la última letra representa un parche o arreglo sobre una operación defectuosa que no implica cambios en la interfaz.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Centralización de	Existe un catálogo de servicios oficial con	Formula cualitativa =

metadatos	documentación y ejemplos para usar cada una de las *APIs* de *SABRE*.	[Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Total	Evaluación cualitativa global de los patrones de diseño vinculados al inventario de servicios de *SABRE*.	Formula cualitativa del inventario = [Rendimiento (+4 -4) + confiabilidad (+6) + seguridad (+1) (+ interoperabilidad (+15 - 2))]

Tabla 47: Patrones de diseño de la muestra de servicios esenciales en *SABRE*:

Patrones de los servicios principales	Aplicación	Valoración de *QoS*
Capacidad agnóstica al contexto	Los servicios S1 - S20 poseen capacidades independientes del ambiente subyacente ya que no exponen funcionalidad relacionada con las tecnologías que los conforman, como conexión a una base de datos particular, pero en S2, S5 – S11, S13 - S16 y S19 existe cierto grado de acoplamiento con la plataforma *SABRE* debido a que se observan valores de campos con información perteneciente a este contexto base sin modelar, por ejemplo, información en formato texto plano tal como los provee el sistema de reserva heredado. En conclusión, el patrón de diseño se cumple parcialmente, por consiguiente, se debe restar una parte representativa referente a los servicios de la muestra que no son agnósticos.	[Formula cualitativa original (capacidad agnóstica al contexto)] – [Formula cualitativa restando los servicios con cierto grado de dependencia] => [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))] - [Rendimiento (-0.5) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+0.5))] = [Rendimiento (+0.5) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+0.5))]
Descomposición funcional	En S1 - S19 se denota la descomposición funcional ya que cada uno de éstos representan unidades lógicas pequeñas, algunas relacionadas entre sí, que aportan a la solución de negocio mayor del *CRS*. Se excluye de este patrón el servicio *hub* de *CTS* (S20) ya es una interfaz general que enlaza a *URIs* de todos los servicios de viajes corporativos, por lo tanto, su	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]

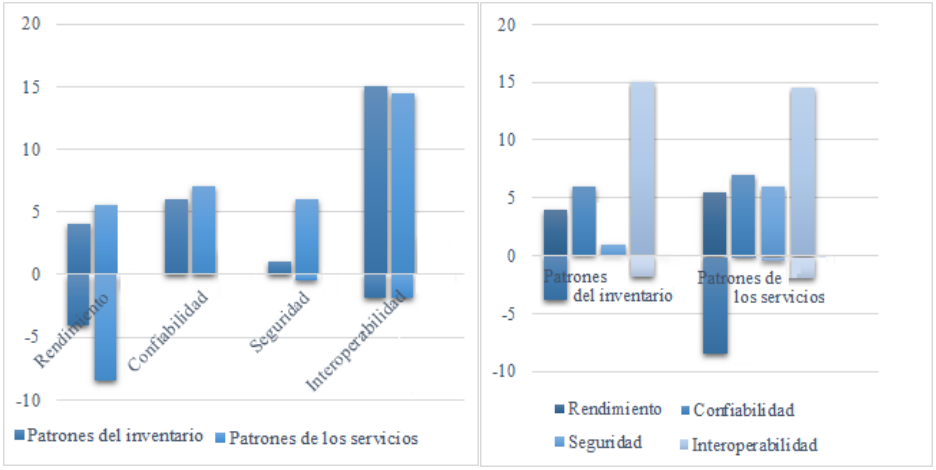
	nivel de descomposición es realmente bajo.	
Encapsulación de servicio	Todos los servicios de la muestra (S1 – S20) encapsulan lógica, accedida a través de *APIs*, que puede ser reutilizada en diferentes soluciones empresariales. Por ejemplo, el servicio que se utiliza para reservar un hotel (S5) puede formar parte de una composición asociada a la venta de un paquete de viaje o de una reservación independiente.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]
Implementación redundante	Existencia de réplicas de servicios S1 – S20 (implementaciones redundantes) que se utilizan para la recuperación ante errores o *failover*, obteniendo alta disponibilidad.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))]
Servicio de protección perimetral	Se evidencia la presencia de un servicio de cortafuegos o *WAF* con reglas de seguridad para evitar ataques *DoS* ya que cuando se ejercitan las *APIs* de los servicios S1 – S20 de manera recurrente con credenciales inválidas éstos quedan inactivos por un período determinado de tiempo desde las direcciones *IP* de dichas cuentas.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+2) (+ interoperabilidad (+-0))]
Subsistema de confianza	Existencia del servicio *API Gateway* para proteger a los servicios (S1 – S20) de accesos por parte de terceros no autorizados. La creación de los *tokens* esta descompuesta a través del servicio S17 en el caso de los servicios clásicos mientras que en los servicios nuevos el método de autenticación seleccionado es *OAuth2*.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+2) (+ interoperabilidad (+-0))]
Contratos concurrentes	En S1 - S9 y S11 - 19 se evidencian múltiples versiones de convenios para los mismos servicios con el fin de atender las necesidades de distintos clientes. Por ejemplo, el servicio S1 cuya versión más reciente es la *OTA_AirFlifoLLS2.1.0RQ* también admite conectividad con las versiones 2.0.2, 2.0.1, 2.0.0 y 1.5.1 para el caso de clientes que no pudieron actualizarse contra la versión más nueva.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))]
Centralización en contratos	Los consumidores se comunican con los servicios través de contratos, *WSDL* para S1 - S9 y S11 - S19 o *YAML* para S10 y S20, los cuales acceden a los recursos de la empresa evitando acoplamiento entre las implementaciones y los clientes.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+2))]

Desnormalización de contratos	Los servicios S10 y S20 (sobre todo el primero) evidencian desnormalización medida de los contratos proveyendo operaciones con cierta redundancia funcional, lo cual mejora el rendimiento operativo en desmedro del mantenimiento de la interfaz.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (-1))]
Desacoplamiento de contrato	Las interfaces técnicas de los servicios S1 - S20 se encuentran físicamente aisladas de las implementaciones evitando así dependencias indeseadas entre los requisitos de negocio y el entorno subyacente.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Paquete heredado	Las *APIS* de *SABRE* S1 - S20 empaquetan las operaciones del CRS exponiendo interfaces que encapsulan la lógica subyacente, disminuyendo así el acoplamiento con el sistema heredado.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Descomposición de servicios	En S1 y S17 - S19 se observa descomposición de servicios genéricos en más granulares para mejorar la interoperabilidad. Por ejemplo, el servicio granular S17 sólo es responsable de crear el *token* de acceso para los servicios *SOAP* lo cual representa una parte específica del servicio genérico *API Gateway*.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Identificación de versión	Los servicios S1 – S9 y S11 – S19 se encuentran versionados, la mayoría incluyendo documentación, para que los clientes estén al tanto de cambios efectuados que puedan alterar el comportamiento esperado. Los servicios S10 y S20 presentan solo una versión explicitada en la sección inicial de los documentos *YAML*.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))]
Capacidad de composición	Los servicios S1 – S8 y S11 – S19 poseen la capacidad de formar parte de una o más composiciones, independientemente de la complejidad que esto atañe, lo cual permite resolver problemas mayores de los que resuelven individualmente. Por ejemplo, S17 puede ser utilizado para proveer el *token* de acceso a una composición de servicios vinculada a la reserva de autos o a otra asociada a la venta de pasajes aéreos.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Mensajería dirigida por eventos	Los servicios de notificación de eventos (*ENS*) evidencian la instauración de un sistema de mensajería basado en el modelo publicar/subscribir	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) +

	para el envío de eventos asíncronos entre servicios, lo cual es adecuado para el caso donde múltiples consumidores reciben el mismo evento. Con esta plataforma de *SABRE* los clientes pueden subscribirse para recibir mensajes *XML* que proporcionan información sobre cambios de la reserva en tiempo real, permitiendo que las aplicaciones de los consumidores trabajen de manera más eficiente, ahorrando tiempo y dinero en consultas.	seguridad (+-0) (+ interoperabilidad (+-0))]
Mensajes con metadatos	Existencia de metadatos de seguridad como parte del encabezado de los servicios esenciales de la muestra. Por un lado, en la sección *wsse:Security* de los mensajes *XML* en los servicios S1 – S9 y S11 – S19 y, por otro lado, en el encabezado *HTTP* de autorización o *Authorization* para las *APIs* S10 y S11.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Controlador agnóstico de subtareas	Los servicios orquestadores S9 - S10 y el *hub* S20 son controladores de subtareas con capacidades agnósticas que fomentan la recomposición de las abstracciones de servicios que los conforman.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Transacción atómica de servicio	El servicio S12 posee el campo *SabreTransactionID* el cual se distribuye entre otras actividades en tiempo de ejecución para simplificar el seguimiento de las tareas de negocios y, también, reiniciar las acciones incompletas en caso de fallas. Como resultante, este servicio se puede utilizar para consultar la información de pago de una sesión de trabajo vinculada con la venta de pasajes, paquetes de viaje, hoteles, autos u otra actividad.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))]
Autenticación vía intermediario	Presencia del *API Gateway* como agente intermediario para autenticar a los consumidores y permitirles que se comuniquen con las *APIs* de *SABRE*.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+-0))]
Origen auténtico de los datos	Los mensajes de los servicios clásicos S1 – S9 y S11 - S19 se encuentran firmados digitalmente en la sección del encabezado garantizando la autenticidad del origen y descartando la alteración de datos en	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+1 -0,5) (+

	tránsito por parte de intermediarios inválidos.	interoperabilidad (+-0))]
Puente entre protocolos	El servicio *API Gateway* de *SABRE* hace de puente entre consumidores que utilizan diferentes protocolos (esquemas de comunicación *HTTP* y *HTTPS*) y los servicios S1 – S20. Cabe aclarar que, sobre la base de este componente intermediario, también se aplica el patrón protocolo dual.	Formula cualitativa = [Rendimiento (-1,5) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Negociación de contenido	En S10 y S20 las interfaces *YAML* admiten la negociación de múltiples formatos de contenido durante el intercambio de los mensajes con los consumidores. En concreto las secciones *produce* y *consume* pueden aceptar el intercambio de mensajes de diferentes tipos de contenido, aunque actualmente sólo se encuentre definido *application/json*.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Idempotencia	El *blueprint* de arquitectura de las *APIs* de *SABRE,* donde se indica el procesamiento de más de 10.000.000.000 transacciones por año, y los servicios S1 – S20, los cuales pueden incluir el número de transacción de una sesión de trabajo, evidencian el diseño de servicios idempotentes que permiten la recepción de mensajes repetidos sin consecuencias negativas sobre la operativa del negocio.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]
Total	Evaluación cualitativa de patrones de diseño de la muestra de servicios principales en *SABRE* (S1 – S20).	[Rendimiento (+5.5 -8.5) + confiabilidad (+7) + seguridad (+6 -0.5) (+ interoperabilidad (+14.5 - 2))]

Gráficos de valoración cualitativa por categorías:



Figuras 72 y 73: Evaluación de *QoS* en *SABRE*. Fuente propia elaborada con *Excel*.

Propuesta de patrones de diseño para mejorar *QoS* : Los resultados de la evaluación resaltan que los servicios en *SABRE* son altamente interoperables, confiables y seguros pero poseen cierta lentitud, por lo cual, la propuesta para mejorar *QoS* hace foco en el aumento de rendimiento teniendo en cuenta las tres categorías de patrones ya expresadas.

Tabla 49: Propuesta de patrones de diseño del inventario para mejorar *QoS* en *SABRE*.

Patrones del inventario	Aplicación	Mejora de *QoS*
Esquemas de datos univoco	Contratos de servicios diseñados para evitar transformaciones de los modelos de datos mediante el uso de esquemas de datos unívocos los cuales son compartidos entre los componentes del inventario. De esta manera la gestión de contenido se centraliza resultando en *APIs* menos redundantes y consumidores que no necesitan de adaptaciones respecto al formato de la información, por lo tanto, esto podría permitir que las fechas se representen de una misma manera para diferentes servicios reduciendo la inconsistencia entre los clientes de S1 - S5, S9 – S11, S15 – 17 y S20 y, de igual manera, se podría eliminar la presencia de datos en texto plano como los *remarks* en S2, S5 – S11 y S13 – S19 para que los clientes de estos últimos no tengan que implementar expresiones regulares con el fin de extraer información específica.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Servicios que mantienen información de estado	Establecimiento de servicios de utilidad particulares que mantienen información de estado con el propósito de sortear que las composiciones de servicios de dominio guarden este tipo de información, por ejemplo, mediante servicios de sesión los cuales podrían gestionar la información de estado del *CRS* desde un lugar central mejorado, utilizando cache con una infraestructura altamente performante (*hardware* dedicado) y tolerante a fallas.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))]
Total	Mejora cualitativa global de los patrones de diseño vinculados al inventario de servicios de *SABRE*.	Formula cualitativa = [Rendimiento (+2) + confiabilidad (+-0) + seguridad (+-0) (+

		interoperabilidad (+1 -1))]

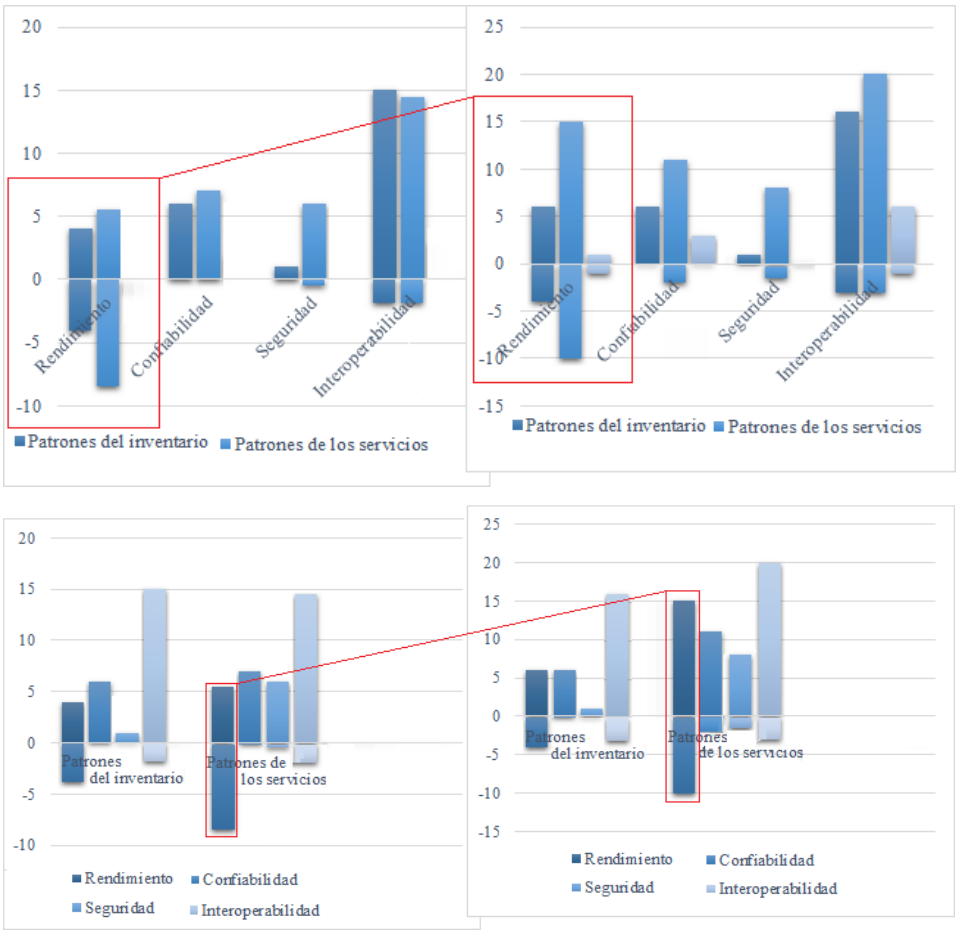
Tabla 50: Patrones de diseño de los servicios para mejorar *QoS* en *SABRE*:

Patrones de servicios	Aplicación	Mejora de *QoS*
Aplazamiento parcial de estado	Aún en los casos donde es necesario mantener la información de estado en los servicios, se puede diferir temporalmente un subconjunto de datos de este tipo. Esto mejoraría el rendimiento de los orquestadores S9 y S10 ya que se reducen las interacciones con los servicios de tareas coordinados mediante *lazy load* de los datos envés de *eager load,* es decir, el detalle de la información de estado se difiere para cuando es necesario recuperarla retornando un breve resumen para la mayoría de los escenarios.	Formula cualitativa = [Rendimiento (+2) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+0))]
Validación parcial de datos	Validación de información de ingreso relevante para lograr desacoplamiento y evitar redundancia en la verificación de datos poco significativos, en los servicios clásicos (S1 – S9 y en S11 – S19) mediante el uso de *XPath* y en los de estilo *ReST* (S10 y S20) a través del uso de *JSONPath*.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Refactorización de servicios	Refactorización/modernización de los servicios en cuanto a la implementación de tecnologías más performantes y seguras. Esto se evidencia en S9 y S20 aunque sin cumplir con un protocolo como *JSON API,* por lo cual, si estas *APIs* se actualizaran de forma adecuada, con enlaces a entidades relacionadas, se disminuiría la redundancia de lógica e incrementaría el rendimiento.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+1) (+ interoperabilidad (+-0))]
Encolamiento asíncrono de mensajes	Por su naturaleza, los servicios S18 y S19 deberían ser implementados de manera asíncrona, por ejemplo, estableciendo de un mecanismo intermedio de encolamiento punto a punto o *Point to Point (P2P)* para el envío de mensajes, lo cual mejora la performance percibida ya que el cliente no debe *lockear* sus recursos mientras espera por la respuesta.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+-0))]
Mensajería confiable	Aunque se reduzca la performance en cierto grado, los servicios de notificación de eventos (*ENS*) deberían establecer un mecanismo intermediario de confianza para garantizar la entrega de los mensajes.	Formula cualitativa = [Rendimiento (-1) + confiabilidad (+1) + seguridad (+-0) (+

	Esto sirve para asegurar la entrega de las notificaciones por parte de componentes que tienen que informar respecto a las valijas perdidas en tránsito o el estado del clima.	interoperabilidad (+-0))]
Devolución de llamada de servicio (*callback*)	En aquellos servicios que se presentan detalles de respuesta extensos o requieren procesamientos prolongados como S2 para mostrar los vuelos disponibles, los orquestadores de reserva aérea y actualización S9 y S10 como así también el *hub* S20, se debería responder con mensajes que incluyan dirección e información correlativa devolviendo las llamadas con las respuestas en un tiempo posterior.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (-1) + seguridad (+-0) (+ interoperabilidad (+-0))]
Mensajería de servicios	Utilización de tecnología de mensajería para el intercambio de mensajes entre servicios internos con el fin de evitar la comunicación a través de protocolos que imponen conexiones persistentes. En el caso de los orquestadores de tareas S9 y S20 esta plataforma (por ejemplo, *Kafka*) mejoraría la performance en cuanto a reducir las llamadas persistentes de componentes intrínsecos mediante un canal de mensajería al cual éstos se subscribirían y reaccionarían ante mensajes dirigidos de manera independiente.	Formula cualitativa = [Rendimiento (+1,5 -0,5) + confiabilidad (-1) + seguridad (-1) (+ interoperabilidad (+1,5))]
Enlaces granulares	Las interfaces de los servicios S10 y S20 tendrían que especificar vínculos livianos a capacidades granulares o entidades de negocio específicas impulsando el potencial de recomposición, por ejemplo, siguiendo el protocolo *JSON API*.	Formula cualitativa = [Rendimiento (+1) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1 -1))]
Entidades vinculadas (*HAETOAS*)	Además del catálogo del producto, tanto los servicios de estilo *ReST* (S10 – S20) como los clásicos deberían exponer en sus interfaces los vínculos con otras entidades de la composición para facilitar el descubrimiento de las interrelaciones por parte de los clientes.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+1) + seguridad (+-0) (+ interoperabilidad (+1))]
Contrato reusable	Los servicios pertenecientes a S20 (*CTS*) deberían ser diseñados con funcionalidad relevante de negocio y baja frecuencia de cambio para evitar que los clientes tengan que actualizar su funcionalidad de manera recurrente.	Formula cualitativa = [Rendimiento (+-0) + confiabilidad (+-0) + seguridad (+-0) (+ interoperabilidad (+1))]
Excepciones	En los errores de los servicios S1 - S20 se deberían	Formula cualitativa =

blindadas	omitir detalles internos con el objetivo de evitar problemas de seguridad asociados a la exposición de datos sensibles, eludiendo exhibir información sensitiva del *CRS* como errores transaccionales vinculados a pagos o de acceso a la base de datos subyacente (por ejemplo, para sortear ataques de inyección *SQL*).	[Rendimiento (-1) + confiabilidad (+-0) + seguridad (+1) (+ interoperabilidad (+-0))]
Total	Mejora cualitativa de los patrones de servicios en *SABRE*.	Formula cualitativa = [Rendimiento (+9.5 -1.5) + confiabilidad (+4 -2) + seguridad (+2 -1) (+ interoperabilidad (+5.5 - 1))]

Gráficos comparativos entre la valoración actual y los patrones para mejorar *QoS* en *SABRE*:



Figuras 74 y 75: Gracias a la propuesta de patrones de diseño para mejorar *QoS* se mejora el rendimiento de manera sustancial logrando un balance adecuado con las restantes características de calidad asociadas a las *APIs* de *SABRE*. Fuente propia hecha con *Excel*.

9. Conclusiones y posibles estudios futuros

9.1. Conclusiones

La contribución principal de esta tesis es el inventariado de patrones de diseño con énfasis en la mejora de *QoS* el cual se basa primordialmente en los modelos de *Richardson*, *Erl* y *Amundsen* teniendo en cuenta los diferentes estilos de servicios y tecnologías de la actualidad. A cada patrón de este catálogo se le asignó una nueva fórmula cualitativa la cual puede ser aplicada tanto a servicios independientes como a colecciones indicando el impacto en el incremento de calidad. Se considera que este diferenciador cualitativo y la comparativa entre los modelos de diseño y los diferentes estilos de *APIs* conforman un aporte sustancial y único a tener en cuenta durante el diseño y actualización de arquitecturas orientadas a servicios y microservicios.

Para alcanzar lo antes mencionado fue necesario:

- Comprender la ontología de los servicios *Web*, estilos existentes, tendencia y análisis de componentes sobre las arquitecturas *SOA* y *MSA*.
- Relevar los estudios previos relacionados a servicios *Web* y arquitecturas en la nube vinculados a patrones de diseño y *QoS*.
- Listar los principios y restricciones de diseño de los servicios.
- Establecer una nómina sintetizada de antipatrones de diseño y su relación con el desarrollo y consumo de servicios.
- Conceptualización de *QoS* y categorización en base a investigaciones recientes.
- Relevamiento, valoración y propuesta de patrones de diseño en las *APIs* de *SABRE,* con lo cual se demuestra que se pueden utilizar las fórmulas cualitativas de los patrones para la mejora de *QoS* en un sistema referente de la actualidad.

9.2. Posibles trabajos a futuro

A continuación, se destacan los posibles estudios a realizar sobre la base del catálogo de patrones de diseño para el incremento de *QoS*:

- Análisis cuantitativo de *QoS* haciendo hincapié en la performance debido a que en este trabajo se omite el análisis numérico del rendimiento, siendo ésta la única

variable de *QoS* que podría ser evaluada de esta forma. La razón por la cual se sorteó este tipo de análisis se basa sobre algunos estudios detallados en la sección trabajos previos, los cuales demuestran la dificultad de una valoración de este estilo sobre todo considerando que excede el alcance de la tesis. Por consiguiente, se podría evaluar el rendimiento de las *APIs* de *SABRE* mediante pruebas de stress y condiciones de red fluctuante.

- Disminución de *QoS* mediante la aplicación de antipatrones en cuanto a que en esta investigación se establece la relación entre cada uno de éstos y los servicios *Web* sin desarrollar la evaluación cualitativa al respecto. Por lo tanto, se podrían analizar las guías establecidas a través de las fórmulas de valoración específicas al antipatrón clase Dios.
- Nómina de patrones de diseño para aumentar la calidad de los microservicios ya que en esta tesis se hace referencia a los modelos esenciales de una arquitectura *MSA*, como contenedores y micro tareas, por lo tanto, se tendría que ahondar en una investigación detallada respecto a este estilo de arquitectura *SOA*. En consecuencia, se podría analizar el impacto de *QoS* en los patrones de diseño asociados al establecimiento de contenedores, como por ejemplo, la selección del nodo líder y como éste influye en la disponibilidad de las instancias mediante actualizaciones del servicio en tiempo de ejecución.

10. Bibliografía

[1] S. Potts y M. Kopack, “Part 1: Introducing Web Services,” *SAMS teach yourself web services in 24 hours*. Indianapolis: Sam publishing, 2003, p. 11.

[2] E. Marks y M. Werrell, “preface,” *Executive's Guide to Web Services*. New Jersey: Wiley & Sons, 2003, pp. XI - XII.

[3] D. Box, J. Vadoss y K. Horrocks, “Service Orientated Architecture,” *Service Orientated Architecture (SOA) in the Real World.* Microsoft, 2007, p. 9. [E-book] Disponible: MSDN e-book.

[4] T. Erl, “Understanding Layers with Services and Microservices,” *Service-Oriented Architecture: Analysis and Design for Services and Microservices 2nd edition.* New Jersey: Prentice Hall, 2017, p. 123. [E-book] Disponible: Amazon e-book.

[5] R. Marset, “¿Qué es un Servicio *Web*?,” *ReST vs. Web Services*. ELP-DSIC-UPV, 2007, p. 3. [Online]. Disponible: http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf. [Accedido Oct. 10, 2012].

[6] E. Gamma, R. Helm, R. Johnson y J. Vlissides, “What is a Design Pattern?,” *Design Patterns CD*. Reading, MA: Wesley, 1998, pp. 2 - 3.

[7] L. Zhang, “WSMoD: A Methodology for Qos-Based Web Services Design,” *Web Services Research for Emerging Applications: Discoveries and Trends*. China: IGI Global, 2010, pp. 16 - 17.

[8] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris y D. Orchard, “What’s a Web Service?,” *Web Service Architecture W3C Working Group Note***.** W3C. NOTE-ws-arch-20040211, Feb. 11, 2004.

[9] D. Crockford, *JavaScript: The Good Parts*. Google Tech Talks, 2009. [Online]. Disponible: https://www.youtube.com/watch?v=hQVTIJBZook&t=2405. [Accedido Mar. 7, 2020].

[10] J. Bean, “Web Services and other Service Types and Styles,” *SOA and Web Services Interface Design—Principles, Techniques, and Standards*. Burlington, MA: MK/OMG Press, 2010, p. 43. [E-book] Disponible: Amazon e-book.

[11] J. Bean, “Web Services and other Service Types and Styles,” *SOA and Web Services Interface Design—Principles, Techniques, and Standards*. Burlington, MA: MK/OMG Press, 2010, p. 44. [E-book] Disponible: Amazon e-book.

[12] E. Christensen, F. Curbera, G. Meredith y S. Weerawarana, *Web Services Description Language (WSDL) 1.1*. W3C. NOTE-wsdl-20010315, Mar. 15, 2001.

[13] R. Chinnici, J. Moreau, A. Ryman y S. Weerawarana, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C. REC-wsdl20-20070626, Jun. 26, 2007.

[14] D. Fallside y P. Walmsley, *XML Schema Part 0: Primer 2nd edition.* W3C. REC-xmlschema-0-20041028, Oct. 28, 2004.

[15] R. Fielding, "Architectural Styles and the Design of Network -Based Software Architectures," Ph. D. disertación, California Univ., Irvine, 2000.

[16] L. Richardson, "Act Three: The Maturity Heuristic," *Justice Will Take Us Millions Of Intricate Moves*. San Francisco: QCon, 2008. [Online]. Disponible: https://www.crummy.com/writing/speaking/2008-QCon/act3.html. [Accedido Mar. 8, 2020].

[17] R. Fielding, "Architectural Styles and the Design of Network -Based Software Architectures," Ph. D. disertación, California Univ., Irvine, 2000, pp. 81 – 82.

[18] M. Amundsen, *RESTful Web Clients: Enabling Reuse Through Hypermedia*. Sebastopol, CA: O'Reilly, 2017. [E-book] Disponible: Amazon e-book.

[19] S. Mizell, M. Foster, K. Fuller y J. Richard, "Representor Pattern," *The Hypermedia Project*. San Francisco: Github, 2014. [Online]. Disponible: https://github.com/the-hypermedia-project/charter#representor-pattern. [Accedido Mar. 8, 2020].

[20] M. Reinbold, "The world through an API," *REST API Notes for 2018-04-19*. Atlanta: Mailchimp, 2018. [Online]. Disponible: https://tinyletter.com/NetAPINotes/letters/rest-api-notes-for-2018-04-19. [Accedido Mar. 8, 2020].

[21] B. Christudas, M. Barai y V. Caselli, "The Mantra of SOA," *Service Oriented Architecture with Java: Using SOA and Web Services to Build Powerful Java Applications*. Birmingham: Packt Publishing, 2008, p. 5.

[22] K. Gabhart y B. Bhattacharya, "SOA Primer," *Service Oriented Architecture (SOA) Field Guide for Executives*. New Jersey: Wiley & Sons, 2008, pp 3 - 27.

[23] A. Arsanjani, "An architectural template for a SOA," *Service-oriented modeling and architecture*. IBM, 2004. [Online]. Disponible: www.ibm.com/developerworks/webservices/library/ws-soa-design1/. [Accedido Mar. 8, 2020].

[24] *Web Services for Management (WSManagement) Specification*, DMTF estándard 1.0.0, 2008.

[25] D. Box, J. Vadoss y K. Horrocks, "The Evolution of SOA," *Service Orientated Architecture (SOA) in the Real World.* Microsoft, 2007, p. 12. [E-book] Disponible: MSDN e-book.

[26] T. Salah, M. Jamal Zemerly, Chan Yeob Yeun, M. Al-Qutayri y Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, Barcelona, 2016, pp. 318 - 325.

[27] Y. Jayawardana, R. Fernando, G. Jayawardena, D. Weerasooriya y I. Perera, "A Full Stack Microservices Framework with Business Modelling," *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka, 2018, pp. 78 - 85.

[28] A. Souri, A. M. Rahmani y N. J. Navimipour, "Formal verification approaches in the web service composition: A comprehensive analysis of the current challenges for future research," *International Journal of Communication Systems*, vol. 31, no. 17, May 2018.

[29] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang y X. Liu, "SmartVM: a SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275 - 293, May 2018.

[30] F. Amato y F. Moscato, "Exploiting Cloud and Workflow Patterns for the Analysis of Composite Cloud Services," *Future Generation Computer Systems*, vol. 67, pp. 255 - 265, 2017.

[31] A. K. Tripathy, M. R. Patra, M. A. Khan, H. Fatima and P. Swain, "Dynamic Web Service Composition with QoS Clustering," *2014 IEEE International Conference on Web Services*, Anchorage, AK, 2014, pp. 678 - 679.

[32] G. Brito, T. Mombach and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019, pp. 140 - 150.

[33] H. Wada, J. Suzuki and K. Oba, "A Model-Driven Development Framework for Non-Functional Aspects in Service Oriented Grids," *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, Silicon Valley, CA, 2006, pp. 30 - 30.

[34] G. Athanasopoulos and M. Pantazoglou, "Interoperability Among Heterogeneous Services: The Case of Integration of P2P Services with Web Services," *International*

Journal of Web Services Research (IJWSR), 2008. [Online]. Disponible: https://www.igi-global.com/article/interoperability-among-heterogeneous-services/3129. [Accedido Mar. 9, 2020].

[35] W. K. Chan, S. C. Cheung, and K. R. Leung, "A Metamorphic Testing Methodology for Online SOA Application Testing," *Advances in Web Services Research Web Services Research for Emerging Applications*. Hershey, PA: IGI Global/Information Science Reference 2010, pp. 45 - 66.

[36] J. Hogg, *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0: Patterns & Practices*. Microsoft Press, 2006.

[37] Microsoft Patterns & Practices Team, "Service Layer Guidelines," *.NET Application Architecture Guide, 2nd Edition*. Redmond, WA: Microsoft, 2009, pp. 115 - 133.

[38] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures.* Cambridge: Cambridge University Press, 2011.

[39] R. Daigneau y I. Robinson, "Design Solutions for SOAP/WSDL and RESTful Web Services," *Service Design Patterns - Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. [Online]. Available: http://www.servicedesignpatterns.com/. [Accedido Mar. 9, 2020].

[40] T. Erl, *Service-Oriented Architecture: Analysis and Design for Services and Microservices 2nd edition.* New Jersey: Prentice Hall, 2017. [E-book] Disponible: Amazon e-book.

[41] B. D. Martino, G. Cretella and A. Esposito, "Semantic and Agnostic Representation of Cloud Patterns for Cloud Interoperability and Portability," *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Bristol, 2013, pp. 182 - 187.

[42] Z. Zheng and M. R. Lyu, *QoS management of web services*. Berlin: Springer, 2015.

[43] T. Aruna y V. Vinod, "Performance Evaluation of Dynamic Web service Composition using Priority Allocation method," *International Journal of Pure and Applied Mathematics*. Chennai: Dr. MGR Educational and Research Institute University, 2017, pp. 353 - 368.

[44] S. Usmani, N. Azeem y A. Samreen, "Service Composition in SOA and QoS Related Issues," *International Journal of Computer Applications in Technology*.

Pakistan: Federal Urdu University of Arts, Science & Technology junto a University of Karachi Dynamic, 2011.

[45] Y. Li, X. Yao y J. Zhou, "Multi-objective Optimization of Cloud Manufacturing Service Composition with Cloud-Entropy Enhanced Genetic Algorithm," *Strojniški vestnik - Journal of Mechanical Engineering*, vol. 62, no. 10, pp. 277 - 290, 2016.

[46] S. Singh y I. Chana, "QoS based Workload Design Patterns in Cloud Computing: A Literature Review," *International Journal of Cloud-Computing and Super-Computing*, vol. 2, no. 2, pp. 37 - 46, 2015.

[47] Y. Yuan, W. Zhang, X. Zhang, y H. Zhai, "Dynamic Service Selection Based on Adaptive Global QoS Constraints Decomposition," *Symmetry*, vol. 11, no. 3, p. 403, 2019.

[48] W. Wang, L. Wang y W. Lu, "An Intelligent QoS Identification for Untrustworthy Web Services via Two-Phase Neural Networks," *2016 IEEE International Conference on Web Services (ICWS)*, San Francisco, CA, 2016, pp. 139 - 146.

[49] S. Saluja y U. Batra, "Assessing Quality by Anti-pattern Detection in Web Services," *SSRN Electronic Journal*, 2019.

[50] Microsoft Patterns & Practices Team, "Service Layer Guidelines," *.NET Application Architecture Guide, 2nd Edition*. Redmond, WA: Microsoft, 2009, p. 115.

[51] S. Abeysinghe y S. D. Mangarole, "Introduction to ReST," *RESTful PHP Web Services: Learn the Basic Architectural Concepts and Steps Through Examples of Consuming and Creating RESTful Web Services in PHP*. Packt Publishing, 2008, pp. 7 - 21.

[52] J. Bean, "Core SOA Principles," *SOA and Web Services Interface Design—Principles, Techniques, and Standards*. Burlington, MA: MK/OMG Press, 2010, pp. 25-41. [E-book] Disponible: Amazon e-book.

[53] C. Alexander, S. Ishikawa y M. Silverstein, *A pattern language: towns, buildings, construction*. New York: Oxford University Press, 1977.

[54] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns— Elements of reusable object-oriented software*. Reading, MA: Wesley, 1994.

[55] F. Buschman, R. Meunier, H. Rohnert, P. Sommerland y M. Stal, *Pattern-Orientated Software Architecture— A system Of Patterns*. Chichester: John Wiley & Sons, 1996.

[56] H. Gomaa, “Software Design and Architecture Concepts,” *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge: Cambridge University Press, 2011, pp. 45 - 60.

[57] P. Kuchana, *Software architecture design patterns in Java*. Boca Ratón, FL: Auerbach, 2004.

[58] T. Erl, *Service-Oriented Architecture (SOA) Concepts, Technology, and Design***.** New Jersey: Prentice Hall, 2006.

[59] W. Brown, R. Malveau, H. McCormick III y T. Mowbray, “What is an AntiPattern?,” *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York: John Wiley & Sons, 1998, pp. 6 - 7.

[60] W. Brown, R. Malveau, H. McCormick III y T. Mowbray, “Templates for Patterns and AntiPatterns,” *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York: John Wiley & Sons, 1998, pp. 30 - 36.

[61] W. Brown, R. Malveau, H. McCormick III y T. Mowbray, “Software Architecture AntiPatterns,” *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York: John Wiley & Sons, 1998, p. 88.

[62] C4ISR Architecture Working Group (AWG), “Architecture Views – Definitions, Roles, and Linkages,” *C4ISR Architecture Framework Version 2.0*. Washington DC: Department of Defense, 1997, p. 2.

[63] Group of SOA Practitioners, “Abstract,” *SOA Blueprint Reference architecture version 1.1*. SOA Alliance, 2006, p. 1.

[64] F. Brooks, *The mythical man-month: essays on software engineering*. Reading, MA: Wesley, 1995.

[65] J. Edwards y D. Devoe, “10 Tips for Three-Tier Success,” *D.O.C. Magazine*, pp. 39 - 42, 1997.

[66] S. Khoshafian, “Service Quality and Management,” *Service Oriented Enterprises*. Boca Raton, FL: Auerbach, 2007, pp. 309 - 357.

[67] L. KangChan, J. JongHong, L. WonSeok, J. Seong-Ho y P. Sang-Won, *QoS for Web Services: Requirements and Possible Approaches.* W3C, 2003.

[68] S. Ran, “A Model for Web Services Discovery with QoS,” *ACM SIGecom Exchanges*, vol. 4, no. 1, pp. 1 - 10, En. 2003.

[69] S. Rajesh y D. Arulazi, “Quality of Service for Web Services-Demystification, Limitations, and Best Practices,” *Learning together ... xi xi*, May. 20, 2013. [Online].

Disponible: https://yogipoltek.wordpress.com/2013/05/20/quality-of-service-for-web-services-demystification-limitations-and-best-practices/. [Accedido: Mar. 9, 2020].

[70] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, IEEE Std. 610, pp. 1 - 217, 1991.

[71] N. J. Gunther, *The Practical Performance Analyst*. San Jose: Authors Choice Press, 2000.

[72] A. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez y Ü. Yalçinalp, *Web Services Policy 1.5 - Framework*. W3C, 2007. [Online]. Disponible: https://www.w3.org/TR/2007/REC-ws-policy-20070904/. [Accedido Mar, 10, 2020].

[73] "Sabre APIs," *Sabre*. [Online]. Disponible: https://www.sabretravelnetwork.com/home/solutions/products/sabre_APIs. [Accedido Mar, 10, 2020].

11. Anexos

11.1. Plantilla de antipatrones principales

Índice. Nombre del antipatrón: Identificador del al antipatrón.
Sinopsis: Se declara de manera sintetizada cuál es la razón de ser del antipatrón, la intención y a que problema se asocia.
Nombre alternativo: Otros nombres de referencia si los hubiere.
Escala más frecuente: Lugar donde ocurre habitualmente, por ejemplo, en la aplicación o la empresa.
Nombre de la solución refactorizada: Identificador de la solución recomendada para resolver el antipatrón.
Tipo de solución refactorizada: Categoría a la cual pertenece la solución.
Causas raíz: Motivo por el cual se origina el antipatrón.
Desequilibrio de fuerzas: Consecuencias de la presencia del antipatrón.
Evidencia anecdótica: Relato o prueba informal del acontecimiento si lo hubiere.
Vínculo con los servicios *Web*: Relación entre el antipatrón y los servicios *Web* incluyendo ejemplos resumidos.
[Figura: Diagrama de diseño opcional donde se demuestra la presencia del antipatrón y los componentes que lo conforman.]

11.2. Plantilla de antipatrones principales sintetizados

Índice. Nombre del antipatrón resumido: Identificador del antipatrón.
Síntesis: Reseña del antipatrón y nexo con el problema al cual se atribuye su presencia.
Solución refactorizada: Resumen de la solución sugerida para resolver el antipatrón.
[Vínculo con servicios los *Web*: Conexión entre el antipatrón y los servicios *Web* añadiendo ejemplos breves si existieran.]

11.3. Plantilla de patrones de diseño de los servicios *Web* para mejorar *QoS*

Índice. Nombre: Identificador del patrón de diseño dentro del catálogo (incluye nombre del autor entre paréntesis).
Sinopsis: Se declara brevemente que hace el patrón, la intención y a que problema se

vincula.

Nombre alternativo: Nombres adicionales de referencia si los hubiere.

Servicios afectados: Categorizaciones principales de servicios afectadas mediante la aplicación del patrón de diseño.

- Por funcionalidad: Clasificación de servicio por funcionad (de tareas, entidad y/o utilidad).
- Por estilo: Tipo de servicio (*SOAP* y/o *ReSTful*).

Problema: Conjunto de hechos o circunstancias de diseño que dificultan la consecución de algún fin.

Solución: Escenario que ilustra como se debe resolver el problema planteado.

Aplicación: Se definen cuáles son las situaciones en las que se puede aplicar el modelo de diseño, cuáles son ejemplos de diseños problemáticos que se pueden tratar y, adicionalmente, en el caso de que existiera, que tecnologías se pueden implementar para resolver el problema.

Efectos: Se establece como funciona el patrón en el cumplimiento de sus objetivos, cuáles son las ventajas y desventajas.

Principios soportados: Principios esenciales que se encuentran beneficiados mediante la implementación del patrón de diseño (acoplamiento débil, reusabilidad, descubrimiento dinámico, autonomía, abstracción, servicio sin estado, contrato estándar y capacidad de composición).

Componentes de arquitectura involucrados: Participantes de la arquitectura afectados (inventario, composiciones y/o servicios).

[Diagrama de arquitectura: Para la mayoría de los casos se presenta el diagrama de diseño donde se establecen las estructuras y componentes que conforman al modelo que resuelve el problema planteado. Las representaciones posibles se seleccionan de acuerdo al tipo de patrón, es decir, para el caso de aquellos vinculados a inventarios se utilizan diagramas de composición de estructuras, y para los de servicios, principalmente diagramas de secuencia, actividad y transición de estado en menor medida.]

Mejora de *QoS*: Sección principal de esta tesis donde se elabora la formula cualitativa y se detalla el impacto en los principios fundamentales que conforman *QoS* (rendimiento, confiabilidad, seguridad e interoperabilidad[56]). La mejora cualitativa se determina a partir

[56] La interoperabilidad es el único principio que se vincula directamente con los servicios *Web* ya que en otras áreas de desarrollo de *software* no aplicaría. Es considerada la característica distintiva de los componentes en *SOA*.

del signo positivo ⊕ mientras que la disminución con el signo opuesto ⊖ y, adicionalmente, se debe tener en cuenta que salvo en el caso de ajustes de fórmulas por aplicación parcial de algún patrón compuesto, los valores con distinto signo no se pueden restar ya que suelen representar atributos independientes, por ejemplo, la aplicación de un patrón vinculado a la seguridad disminuye el rendimiento debido al cifrado de los datos y la implementación conjunta de otro patrón de servicio para mejorar la performance de respuesta mediante mecanismos asíncronos son mas performantes que los protocolos de comunicación persistentes pero no van a eliminar el tiempo de encriptación del contenido.

11.4. Plantilla de evaluación de las *APIs* esenciales en *SABRE*

Índice. Nombre de servicio: Identificador del servicio en el catálogo de *SABRE* (identificador oficial en inglés).
Audiencia: Público del servicio.
Categoría: Área de negocio a la cual pertenece la *API*.
Funcionalidad: Clasificación funcional del servicio en el negocio.
Estilo: Protocolo o arquitectura de implementación del servicio (*SOAP* o de estilo *ReST*).
Sinopsis: Reseña de la naturaleza y el objetivo principal de la *API*.
Tabla índice: Interfaz:

Tecnología del contrato (versión)
Detalle de la *API* oficial si se encuentra documentada.

Tabla índice: Ejemplo de solicitud y respuesta:

Solicitud
Detalle del/los ejemplo/s de contenido a enviar (la tecnología de intercambio de mensajes depende del estilo de servicio).
Respuesta
Detalle del ejemplo del contenido de la respuesta.

Características de diseño principales: Particularidades principales de diseño de la *API* a tener en cuenta durante la evaluación de *QoS*.

12. Apéndice

12.01. Muestra[57] de *anti-patterns* y sus características vinculadas con servicios *Web*

Antipatrones de desarrollo:

12.01. Nombre del antipatrón: Flujo de lava.
Sinopsis: El código muerto y la información de diseño olvidada congelan la evolución constante del diseño. Esto es análogo a un flujo de lava con glóbulos de endurecimiento de los materiales rocosos. La solución de rediseño incluye un proceso de gestión de configuración que elimine el código muerto y evolucione o *refactoring* de diseño para aumentar la calidad.
Nombre alternativo: Código muerto.
Escala más frecuente: Aplicación.
Nombre de la solución refactorizada: Gestión de configuración arquitectónica.
Tipo de solución refactorizada: Proceso.
Causas raíz: Avaricia, codicia y pereza.
Desequilibrio de fuerzas: Manejo de funcionalidades, rendimiento y complejidad.
Evidencia anecdótica: Una implementación antigua, mal documentada, de la cual no se sabe bien que funcionalidad implementa y los desarrolladores no están seguros de eliminarla.
Vínculo con los servicios *Web*: En *SOAP* una operación definida en un *WSDL* sin documentación fomenta la existencia de código muerto ya que puede ser difícil de interpretar en un futuro, y en *ReST* se debe usar *Swagger* tanto para asegurar la presencia de documentación como testeo continuo de la *API*. La falta de documentación respecto a la *API* de un servicio hace que en un futuro este código se convierta en una pieza muerta por lo cual la mitigación es documentar.

12.02. Nombre del antipatrón: Descomposición funcional.
Sinopsis: Este antipatrón es el resultado de la inclusión de desarrolladores experimentados no acostumbrados a trabajar usando el paradigma orientado a objetos quienes diseñan e implementan una aplicación *OO*. El código resultante se asemeja a un

[57] Se omiten los antipatrones descriptos en la sección 5.

lenguaje estructural (por ejemplo, *Pascal, Fortran*) en la estructura de las clases.

Nombre alternativo: No OO.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: *OO refactoring*.

Tipo de solución refactorizada: Proceso.

Causas raíz: Avaricia, codicia y pereza.

Desequilibrio de fuerzas: Manejo de complejidad y cambio.

Evidencia anecdótica: Los desarrolladores se encuentran confortables usando una rutina principal la cual llama a múltiples subrutinas, e intentan crear una clase por cada subrutina ignorado los principios orientados a objetos.

Vínculo con los servicios *Web*: Para *SOAP* es implementar múltiples clases diferentes que resuelvan cada operación de un mismo *skeleton* (vinculado a un servicio definido en el *WSDL)* sin tener en cuenta principios de programación *OO* como herencia y por tal motivo no hacer reúso de comportamiento común. Aunque para *ReST* y un enfoque de microservicios pueda parecer algo esperado esto en realidad es incorrecto, ya que ocurre si se habla de dos recursos que duplican funcionalidad lo cual debería incurrir en la creación de otro, a nivel granular, que resuelva esta situación (en un servicio de nivel 3 del *RMM* se podría especificar el vínculo hipermedia a este recurso común entre los dos servicios en disputa, evitando así duplicación de código y responsabilidades).

12.03. Nombre del antipatrón: Proliferación de clases fantasma.

Sinopsis: Éstas son clases con roles muy limitados y ciclos de vida pequeños (complejas y difíciles de mantener). A menudo sólo comienzan los procesos de otros objetos. La solución refactorizada incluye una reasignación de las responsabilidades hacia objetos de larga vida para una posterior eliminación del componente fantasma.

Nombre alternativo: *Poltergeists*.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: *Ghostbusting* para identificar y eliminar todas las clases de este estilo de manera conjunta.

Tipo de solución refactorizada: Proceso.

Causas raíz: Pereza e ignoracia.

Desequilibrio de fuerzas: Manejo de funcionalidad y complejidad.

Evidencia anecdótica: No se está seguro del rol que cumple la clase pero se cree que es relevante su presencia.

Vínculo con los servicios *Web*: En *SOAP* un servicio *backend* compuesto por una sola operación denominada "comenzarProceso..." y para *ReST* es la ejecución del verbo *POST* en un recurso que sólo cumple con este verbo (niveles 0 y 1 de *RMM*) y realiza una única acción que se encuentra vinculada a la composición de otro recurso lo cual quiebra los principios de granularidad fina e interdependencia respecto a servicios de utilidad. Por favor no confundir con servicios que modelan tareas (servicios coordinadores de negocio pertenecientes a la capa superior) los cuales suelen estar limitados a una funcionalidad de negocio motivo por el cual suelen ser menos reutilizables, pero desde el punto de vista de la composición de un inventario son válidos (ver más detalle en la sección 2.4 – clasificación funcional).

12.04. Nombre del antipatrón resumido: Ancla.

Síntesis: Este tipo de antipatrón representa una pieza de *software* o *hardware* que no sirve para ningún propósito útil en el proyecto actual. A menudo, éste es una adquisición costosa, lo que hace la compra aún más irónica. Las consecuencias para los administradores y desarrolladores de software son un esfuerzo importante dedicado a hacer que el producto funcione. Después de una importante inversión de tiempo y recursos, el personal técnico se da cuenta de que el producto es inútil en el contexto actual, y lo abandona por otro enfoque técnico.

Solución refactorizada: Una buena práctica de ingeniería incluye la provisión de un *backup* técnico, un enfoque alternativo que puede ser establecido mediante retrabajo mínimo vinculado al desarrollo de *software*. La selección de un *backup* técnico de seguridad es una importante estrategia de mitigación de riesgos. Los *backups* técnicos deben ser identificados en la mayoría de las tecnologías de infraestructura y en otras tecnologías en zonas de alto riesgo. Se recomienda un enfoque prototipado con licencias de evaluación (disponible por la mayoría de los vendedores) para el caso de los caminos críticos y los *backups* tecnológicos.

Vínculo con los servicios *Web*: La aplicación de una nueva tecnología de desarrollo para servicios *Web* impuesta por un gurú tecnológico vinculado a un proveedor determinado (por ejemplo, *IBM*) sin tener en cuenta que la tecnología usada actualmente cumple con los objetivos necesarios (por ejemplo, utilizando el *framework Axis CXF* el cual es reconocido por brindar soporte especializado en el dominio de los servicios en cuanto a los diferentes estilos).

12.05. Nombre del antipatrón: Martillo de oro.

Sinopsis: Es una tecnología familiar o concepto aplicado obsesivamente a múltiples problemas ligados al *software*. La solución consiste en ampliar el conocimiento de los desarrolladores a través de la educación, capacitación y grupos de estudio con el objetivo de exponer tecnologías y enfoques alternativos.

Nombre alternativo: Martillo de plata.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: Expandir los horizontes mediante capacitación.

Tipo de solución refactorizada: Proceso.

Causas raíz: Ignorancia, orgullo y estrechez de pensamiento.

Desequilibrio de fuerzas: Manejo de transferencia tecnológica.

Evidencia anecdótica: El equipo de desarrollo tiene gran experiencia sobre una tecnología en particular provista por un vendedor. Como resultado, cada nuevo producto o esfuerzo de desarrollo es visto como algo que solo puede ser desarrollado en esta tecnología.

Vínculo con los servicios Web: Los mensajes *XML* basados en *SOAP* o el uso de *JSON* con *ReST* son sinónimo de implementaciones "de facto" respecto a la capa de servicios de una arquitectura orientada a servicios; sin embargo, *SOA* puede ser implementada por otros tipos de servicios dependiendo de las necesidades. Por ejemplo, si el negocio es mediano o pequeño, se podría recomendar *RMI*[58] cuando la tecnología del proveedor y el consumidor está escrita en lenguaje *JAVA* y la comunicación respecto al puerto *RMI* habilitado es posible (el puerto 1099 está habilitado en los colaboradores).

12.06. Nombre del antipatrón: Código espagueti.

Sinopsis: Son estructuras de código altamente acopladas que dificultan el mantenimiento y la reutilización del *software*. La refactorización de código con frecuencia puede mejorar la estructura del software, apoyar el mantenimiento del *software*, y permitir el desarrollo iterativo.

Nombre alternativo: -.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: Limpieza de código o *software refactoring* para mejorar el mantenimiento y el rendimiento.

Tipo de solución refactorizada: *Software*.

[58] *Remote Method Invokation (RMI)* es una *API JAVA* que ejecuta llamadas a métodos remotos entre objetos de manera performante. Para más detalle ver: <https://www.javatpoint.com/RMI>

Causas raíz: Ignorancia y pereza.

Desequilibrio de fuerzas: Manejo de complejidad y cambio.

Evidencia anecdótica: El código spaghetti se presenta como un programa o sistema que contiene una pequeña estructura de *software*. La codificación y extensiones progresivas comprometen la estructura del software de tal manera que la estructura carece de claridad, incluso respecto al desarrollador original. Si se desarrolla utilizando un lenguaje orientado a objetos, el software puede incluir un pequeño número de objetos que contienen los métodos con las implementaciones de gran tamaño que invocan un único flujo o proceso de varias etapas. Además, los métodos de los objetos se invocan de una manera muy predecible, y hay un grado insignificante de interacción dinámica entre los objetos en el sistema. El sistema es muy difícil de mantener y prolongar, y no hay oportunidad de reutilizar los objetos y los módulos en otros sistemas similares.

Vínculo con los servicios *Web*: En *SOAP* cuando un *WSDL* define múltiples *endpoints* de servicios los cuales se encuentran fuertemente acoplados respecto a la implementación de la clase principal *skeleton* (a pesar de la posibilidad de definir múltiples *endpoints* en un *WSDL* esta es una práctica no recomendada para evitar el acoplamiento entre servicios). En *ReST* cuando existen interdependencias entre recursos lo cual afecta la granularidad (ocurre en el nivel 1 del *RMM* debido al no uso de verbos).

12.07. Nombre del antipatrón: Cortar y pegar.

Sinopsis: La reutilización de código mediante la copia de sentencias de código fuente ya existentes solo conduce a problemas de mantenimiento importantes. Este enfoque es muy común, y proviene de la noción que indica que es más fácil de modificar el *software* existente que implementar un programa desde cero. Esto suele ser cierto y representa uno de los principios del paradigma *OO* mediante herencia salvo que en este caso el resultado es la duplicación del código.

Nombre alternativo: Codificación de portapapeles o *sofwtare* clonado.

Escala más frecuente: Aplicación.

Nombre de la solución refactorizada: Reúso de caja negra. La clonación se produce con frecuencia en entornos donde la reutilización de caja blanca o *white box* es la forma predominante para extender la funcionalidad del sistema. En la reutilización de caja blanca, los desarrolladores extienden los sistemas fundamentalmente a través de la herencia. Ciertamente, la herencia es una parte esencial del desarrollo orientado a objetos, pero tiene varios inconvenientes en los sistemas de gran tamaño. En primer lugar, las

subclases que extienden de un objeto requieren un cierto conocimiento respecto a como el objeto se instancia, tales como las limitaciones previstas y los patrones de uso indicados por las clases base heredadas. La mayoría de los lenguajes orientados a objetos imponen muy pocas restricciones, los tipos de extensiones se pueden implementar en una clase derivada, y conducir a un uso no óptimo de las subclases. Además, típicamente, la reutilización de caja blanca es posible solamente en el momento de compilación (para lenguajes compilados, no así en los interpretados); ya que todas las subclases deben estar completamente definidas antes de la ejecución de la aplicación. Por otro lado, la reutilización caja negra tiene un conjunto diferente de ventajas y limitaciones, y es frecuentemente una mejor opción para la extensión funcionalidad de los objetos en sistemas grandes y moderados. Con la reutilización de caja negra, un objeto se utiliza tal cual, a través de su interfaz especificada. El cliente no está autorizado para alterar la interfaz del objeto que instancia. La principal ventaja de la reutilización en este tipo de programación es que la implementación de un objeto componente se puede hacer independiente sin extender la interfaz del objeto con comportamiento común[59] (composición).

Tipo de solución refactorizada: Programación orientada a componentes.

Causas raíz: Pereza.

Desequilibrio de fuerzas: Manejo de recursos y transferencia tecnológica.

Evidencia anecdótica: Un defecto es corregido en una pieza de código y otra pieza que posee un comportamiento similar sigue actuando de manera defectuosa. Existe una gran cantidad de líneas de código agregadas y el diseño se encuentra muy acoplado a diferentes piezas clonadas probablemente creadas por desarrolladores nuevos que se encuentran en una etapa de aprendizaje mediante el ejemplo (se copian el ejemplo funcional y se agregan lo nuevo).

Vínculo con los servicios *Web*: Tanto en *SOAP* como en *ReST* esta situación se evidencia cuando no se modelan las responsabilidades de los servicios de manera adecuada lo cual conlleva a la presencia de *APIs* que duplican comportamiento similar y por lo tanto son engorrosas de mantener. En *ReST,* esto se mitiga mediante el cumplimiento del nivel 3 del *RMM* siguiendo el principio de composición o caja negra.

[59] La distinción entre la reutilización *white box* y *black box* refleja la diferencia entre la programación orientada a objetos (*OOP*) y la programación orientada a componentes o *Component Orientated Programming* (*COP*), donde la creación de subclases de caja blanca son la firma tradicional de la programación orientada a objetos y el enlace dinámico a través de la interfaz es el elemento básico en *COP*.

12.08. Nombre del antipatrón resumido: Ingreso de fallas.

Síntesis: *Software* que se comporta defectuosamente ante sencillas pruebas son un ejemplo de este antipatrón que ocurre cuando se emplean algoritmos ad hoc[60] para el manejo del ingreso de datos. Por ejemplo, si el programa acepta la entrada de texto libre por parte del usuario, un algoritmo ad hoc manejará múltiples entradas diferentes de manera incorrecta. Como consecuencia de este antipatrón el usuario final puede romper la aplicación rápidamente.

Solución refactorizada: Utilizar algoritmos de ingreso de datos con calidad de un entorno de producción. Por ejemplo, el análisis de léxico y *software* de parseo de datos disponibles como *freeware*. Programas tales como *lex* y *yacc* permiten el control robusto de texto compuesto de expresiones regulares y gramáticas libres de contexto del lenguaje. Se recomienda el empleo de estas tecnologías en el desarrollo de *software* de producción de calidad para asegurar el manejo adecuado de ingresos inesperados.

Vínculo con los servicios *Web*: Para *SOAP* respecto a una operación de un *WSDL* la cual posee argumentos de tipo textuales simples que permitan el ingreso libre de datos y para *ReST* el caso de los campos de texto *JSON*. En ambos enfoques se sugiere la utilización de librerías *lex* y *yacc* en la implementación o simplemente la delimitación del capo de texto definido mediante el uso de expresiones regulares en validadores de esquema *XML* o *JSON*.

Antipatrones de arquitectura:

12.09 Nombre del antipatrón: Islas automatizadas.

Sinopsis: Este tipo de antipatrón es caracterizado por una estructura de *software* que inhibe los cambios, la cual evidencia falta de coordinación y planificación a través de un conjunto de sistemas. Múltiples sistemas dentro de una empresa están diseñados de forma independiente en todos los niveles. La falta de uniformidad impide la interoperabilidad entre sistemas, impide la reutilización, y eleva los costos y, además, servicios que carecen de calidad respecto a una estructura de apoyo para lograr adaptabilidad.

Nombre alternativo: Múltiples tubos de concina empresariales.

[60] Algoritmos ad hoc hacen referencia a código hecho en el momento; en este caso el manejo de los datos de entrada es implementado por el desarrollador sin tener en cuenta la reutilización de *software* existente.

Escala más frecuente: Empresa.

Nombre de la solución refactorizada: Planeamiento de la arquitectura a nivel empresa o *Enterprise Architecture Planning* para la coordinación de las tecnologías en todos los niveles de la compañía. En un principio, una selección de normas a ser coordinadas a través de la definición de un modelo de referencia de estándares. En modelo de referencia se definen las normas comunes y una dirección de migración para los sistemas de la empresa. El establecimiento de un entorno operativo común sirve para coordinar la selección de productos y controlar la configuración de las versiones del producto. Por último, la definición de perfiles de sistemas, que coordinen la utilización de los productos y las normas, es esencial para asegurar los beneficios de normalización, reutilización e interoperabilidad. Por lo menos, el establecimiento de un perfil del sistema debe definir las convenciones de uso de todos los sistemas.

Tipo de solución refactorizada: Proceso.

Causas raíz: Prisa, apatía y estreches de pensamiento.

Desequilibrio de fuerzas: Manejo del cambio, recursos y transferencia de tecnologías.

Evidencia anecdótica: Cada sistema dentro de la empresa posee la forma de una isla de automatización.

Vínculo con los servicios *Web*: Mediante la aplicación de servicios *Web* relacionados a una arquitectura orientada a servicies también es necesario la definición de perfiles, en *SOAP* establecidos por *WS-I* y en *ReST* cumpliendo con el nivel 3 del *RMM* mediante *JSON API*, con el objetivo de lograr interoperabilidad y coordinación entre los sistemas colaboradores evitando así islas de servicios especializados sin interrelación.

12.10. Nombre del antipatrón: Bloqueo del proveedor.

Sinopsis: Este tipo de antipatrón ocurre en sistemas que se encuentran fuertemente acoplados a arquitecturas propietarias. Cuando las actualizaciones se realizan, existen cambios en el *software* y los problemas de interoperabilidad se producen, por lo cual es necesario el mantenimiento continuo para lograr que el sistema siga funcionando.

Nombre alternativo: Arquitectura sumisa a un proveedor o conspiración de producto.

Escala más frecuente: Sistema.

Nombre de la solución refactorizada: Una capa aislada o *Isolation Layer* separa la tecnología y los paquetes de un vendedor en particular con el objetivo de brindar portabilidad de *software* respecto a interfaces de plataformas específicas propietarias.

Tipo de solución refactorizada: Software.

Causas raíz: Pereza, apatía, orgullo/ignorancia (credulidad).

Desequilibrio de fuerzas: Manejo de transferencia de tecnologías y cambio.

Evidencia anecdótica: Problemas encontrados en proyectos de *software* los cuales aclaman que su arquitectura está basada en algún vendedor en particular o línea de producto. El peor escenario posible está relacionado a una solución *online* la cual deja de ser bridada simplemente por un bloqueo del vendedor.

Vínculo con los servicios *Web*: El uso de servicios *Web,* por definición, evita la generación de bloqueos de vendedores ya que un cliente tiene la potestad de cambiar de proveedor de servicio, sobre todo teniendo en cuenta que las interfaces de una industria suelen poseer un vocabulario en común. Además, en la arquitectura *SOA*, la cual se encuentra dividida en capas, se proveen los niveles intermedios entre la presentación y las interfaces de los servicios respecto a componentes de sistemas operativos tal como se detalla en la sección 2.5 evitando así ligaduras tecnológicas propietarias.

12.11. Nombre del antipatrón resumido: Cortapluma suizo.

Síntesis: Este antipatrón se refiere a las interfaces excesivamente complejas de una clase. Con la intención de lograr que ésta se adecue a todos los casos de uso posibles el diseñador crea un número excesivo de firmas o interfaces. En este patrón son frecuentes en las interfaces de *software* comercial, donde los vendedores están tratando de hacer que sus productos sean aplicables a todas las posibilidades existentes. Este antipatrón es problemático porque ignora la fuerza de la gestión de la complejidad, es decir, es difícil que otros programadores entiendan el objetivo de la clase, incluso en casos sencillos. Otras consecuencias de la complejidad son las dificultades asociadas a la depuración, documentación y mantenimiento.

Solución refactorizada: A menudo, las interfaces complejas y estándares deben ser utilizadas en un proyecto de desarrollo de *software*, por lo tanto, es importante definir convenios para el uso de estas tecnologías de acuerdo con la administración de la arquitectura compleja; de tal manera que la aplicación no se vea comprometida. Esto se llama la creación de un perfil. Un perfil es una convención documentada que explica como utilizar una tecnología compleja. A menudo, un perfil es un plan de ejecución relacionado a la implementación de los detalles de una tecnología. Con los perfiles, dos desarrolladores independientes pueden usar la misma tecnología, con la probabilidad de alcanzar *software* interoperable. Un perfil de una interfaz de *software* define el subconjunto de las firmas que se utilizan, y debe incluir las convenciones para los valores

de los parámetros.

Vínculo con los servicios *Web*: En *SOA* se recomienda el desarrollo de servicios granulares e interoperables para evitar la definición de interfaces complejas que generen interdependencia. En el caso se *SOAP, WS-I* publica los perfiles vinculados a servicios *Web* con el objetivo de lograr interoperabilidad (*WS-I Basic* se presenta como el primer perfil existente relacionado al dominio de servicios e indica las cuatro funcionalidades básicas requeridas para el desarrollo de servicios *Web*[61]). En el caso del estilo *ReST* se sugiere el seguimiento de la especificación *JSON API* en entornos empresariales.

12.12. Nombre del antipatrón: Reinventar la rueda.

Sinopsis: Este antipatrón se presenta cuando un nuevo sistema de *software* es creado sin tener en cuenta funcionalidades existentes en aplicaciones similares.

Nombre alternativo: Diseño de un sistema en vacío.

Escala más frecuente: Sistema.

Nombre de la solución refactorizada: *Architecture Mining* es una manera de crear de forma rápida y con éxito arquitecturas orientadas a objetos robustas, reutilizables y extensibles. *Mining* es un enfoque de diseño *bottom-up* que incorpora conocimiento de implementaciones existentes. Ésta también puede incorporar entradas de diseño de procesos de diseño *top-down*, de tal manera que se incorporan características valiosas de los dos enfoques; aunque este tipo de solución se centraliza primordialmente en el conocimiento de funcionalidades de sistemas *legacy*. *Mining* es aplicable a nivel empresarial y, por obvias razones, de menor manera a nivel global ya que la información se ve restringida debido a ser considerada información competitiva. En una primera etapa, es necesario identificar un conjunto de tecnologías representativas relevantes en el dominio mediante la investigación de conocimiento existente; segundo, se modela cada tecnología representativa necesaria para producir interfaces relevantes de *software*; tercero, los deseos se generalizan para crear especificaciones de interfaces comunes; y finalmente se realiza un refinamiento del diseño.

Tipo de solución refactorizada: Proceso.

Causas raíz: Orgullo e ignorancia.

Desequilibrio de fuerzas: Manejo del cambio y transferencia de tecnologías.

[61] Las cuatro especificaciones originales que conforman el perfil *WS-I Basic* son *XML Schema* 1.0, *SOAP* 1.1, *WSDL* 1.1, y *UDDI* 2.0. Actualmente *WS-I* forma parte de *OASIS* y la última versión disponible de *WS-I Basic* es la 2.0.

Evidencia anecdótica: Se menciona que el *software* a desarrollar posee características únicas y los desarrolladores generalmente poseen un conocimiento mínimo respecto al código de otros desarrolladores.

Vínculo con los servicios *Web*: El paradigma orientado a servicios se basa en la interoperabilidad entre los sistemas con el objetivo de que cada especialista en la industria exponga sus funcionalidades sin necesidad de inventar la rueda generando relaciones de negocio de confianza entre colaboradores. En una *SOA* se deben seleccionar con cuidado las interfaces expuestas y composiciones de servicios con el fin de evitar el solapamiento de implementaciones.

Antipatrones en la gestión de proyectos de *software*:

12.13. Nombre del antipatrón: Parálisis de análisis.

Sinopsis: La lucha por la perfección en la completitud de la fase de análisis a menudo conduce a la parálisis del proyecto generando un excesivo refinamiento de los requisitos y modelos. La solución refactorizada incluye una descripción incremental de los requerimientos y procesos iterativos de desarrollo que difieren del análisis detallado hasta obtener el conocimiento necesario.

Nombre del patrón: Parálisis de análisis.

Nombre alternativo: Análisis en cascada.

Escala más frecuente: Sistema.

Nombre de la solución refactorizada: Desarrollo iterativo e incremental.

Tipo de solución refactorizada: Proceso.

Causas raíz: Orgullo y estrechez de mente.

Desequilibrio de fuerzas: Manejo de la complejidad.

Evidencia anecdótica: Se tiene la sensación de que hasta que no se completen en un 100% el análisis y el diseño no se puede comenzar con la implementación del código del sistema.

Vínculo con los servicios *Web*: -.

12.14. Nombre del antipatrón resumido: Ingeniería centrada en gráficos.

Síntesis: En algunos proyectos, los desarrolladores se atascan o pierden demasiado tiempo en la preparación de gráficos y documentos en lugar del desarrollo de software. La administración no obtiene las herramientas de desarrollo adecuadas, y los ingenieros no

tienen más remedio que utilizar el *software* de oficina existente para producir diagramas seudotécnicos y documentos. Esta situación es frustrante para los ingenieros, que no utilizan su verdadero talento. Muchos de los desarrolladores atrapados en la preparación de gráficos y documentos deben ser redirigidos a la construcción de prototipos.
Solución refactorizada: Sacar fotos a las imágenes de los diseños discutidos en pizarrón y utilizarlas como evidencia en conjunción al uso de los *blueprint* de arquitectura.

12.15. Nombre del antipatrón resumido: Humo y espejitos de colores.
Síntesis: Los sistemas para demostración son herramientas importantes en las ventas, ya que son fáciles de interpretar por los usuarios finales, como representación de la calidad de las capacidades de producción. Un equipo de gestión, ávido en nuevos negocios, a veces (sin querer) puede generar errores de percepción por parte de los usuarios al realizar compromisos más allá de las capacidades de la organización para ofrecer tecnología en funcionamiento. Esto pone a los desarrolladores en una situación difícil debido a que se los presiona en el ofrecimiento las capacidades comprometidas. El perdedor es el usuario final, quien no recibe un sistema con las capacidades previstas en cuanto al costo y en el tiempo comprometido, o cuando se entrega el sistema, el usuario final obtiene a menudo un sistema implementado a toda prisa con muchos defectos.
Solución refactorizada: Es relevante la gestión de las expectativas de los usuarios finales tanto por razones éticas como para establecer credibilidad. Una típica regla de ingeniería de *software* es un sistema cuesta tres veces más que un prototipo.

12.16. Nombre del antipatrón resumido: Mala gestión de proyecto.
Síntesis: Una inadecuada atención en la gestión de los procesos de desarrollo de *software* suele causar falta de dirección y otros síntomas. Comúnmente, las actividades clave se pasan por alto o se minimizan en este antipatrón.
Solución refactorizada: La implementación de un sistema debe ser considerada tan compleja como la creación de una actividad del plan del proyecto, por lo cual el desarrollo de *software* es una tarea de ingeniería tan dificultosa como la construcción de rascacielos respecto a la necesidad de inclusión de pasos y procesos, incluyendo controles y equilibrios. El seguimiento y control de los proyectos de *software* es necesario para que las actividades de desarrollo sean exitosas.

12.17. Nombre del antipatrón resumido: Uso de *e-mail* excesivo.

Síntesis: El uso de correo electrónico es un modo ineficaz de comunicación para temas complejos. Debido al tipo de tecnología y otras características importantes del medio, el correo electrónico está sujeto a una interpretación errónea, porque a menudo los grandes intercambios de correo electrónico tienden a reducir la discusión al mínimo común denominador. Por consiguiente, las largas discusiones a través de este medio consumen demasiado tiempo y requieren mano de obra intensiva.

Solución refactorizada: Se debe usar el correo electrónico de manera cautelosa evitando su uso respecto a mensajes confortativos, críticas, información sensible, tópicos políticamente incorrectos y declaraciones legales. A pesar de que las conversaciones telefónicas, y los debates cara a cara también sean vulnerables a la divulgación, su potencial de daño es menos inminente.