

MAPE-K as a Service-oriented Architecture

Juan Vizcarrondo, Jose Aguilar, Ernesto Expósito, Audine Subias

▶ **To cite this version:**

Juan Vizcarrondo, Jose Aguilar, Ernesto Expósito, Audine Subias. MAPE-K as a Service-oriented Architecture. IEEE Latin America Transactions, 2017, 15 (6), pp.1163-1175. hal-01872138

HAL Id: hal-01872138
https://laas.hal.science/hal-01872138

Submitted on 11 Sep 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MAPE-K as a Service-oriented Architecture

J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias

Abstract— **The reflective middlewares have been used as a powerful tool to cope with inherent heterogeneous nature of distributed systems, in order to give them greater adaptability capacities. Recently, some papers have extended the reflective middlewares with autonomic capabilities based on the autonomic computing paradigm. One of the main component is the MAPE-K (Monitor-Analyze-Plan-Execute plus Knowledge) component of the autonomic manager. In this paper we develop a MAPE-K component as a service of the autonomic managers of a reflective middleware based on autonomic computing. In this way, the MAPE-K is distributed to each service that is part of the composition, and internally each component of MAPE-K is constructed as a separate service, distributed and weakly coupled, giving great flexibility. We test the MAPE-K component as a service in a reflective middleware architecture based on autonomic computing, for the distributed diagnostic of faults in the services composition.**

Keywords— **SOA systems, reflective middleware, autonomic computing.**

I. INTRODUCCIÓN

El DESARROLLO de aplicaciones basadas en la Arquitectura Orientada a Servicios (SOA, por sus siglas en inglés Service Oriented Architecture) consiste en un patrón de desarrollo de software en el que una aplicación es descompuesta en pequeñas unidades, lógicas o funcionalidades, denominadas servicios. SOA permite un despliegue de aplicaciones distribuidas muy flexible, con bajo acoplamiento entre sus componentes de software, los cuales interactúan en ambientes distribuidos heterogéneos. En general, SOA provee un estándar que hace posible la interoperatividad, la reusabilidad entre distintas piezas de software, entre otras cosas. Los Servicios Web son [18] entidades computacionales que se pueden agregar a otros Servicios Web para el desarrollo de aplicaciones más grandes, que pueden, a su vez, ser publicadas, localizadas e invocadas a través de Internet. La integración de servicios facilita la cooperación, cruzando los límites de organizacionales. Este estilo de programación basado en la composición y en la reutilización de servicios (nuevas aplicaciones basadas en servicios existentes), es posible por el apropiado ensamblaje de otros servicios ya existentes.

Por otro lado, la mayoría de los actuales middlewares se conciben como arquitecturas capaces de adaptarse ellas mismas, basadas en una inferencia de cómo es su conducta actualmente y las condiciones que presenta el ambiente en el que se desenvuelven. Uno de estos tipos de middlewares son los reflexivos, que al tener una auto-representación, les es posible realizar reconfiguraciones dinámicamente. De esta manera, logran mejorar el desempeño de las aplicaciones que se ejecutan sobre ellos basados en la optimización de las propiedades no funcionales, tales como tiempo de respuesta, disponibilidad, tolerancia a fallos, escalabilidad, seguridad, entre otros.

En este mismo orden de ideas, la Computación Autonómica representa un modelo de computación de auto-gestión, permitiéndole a una aplicación adaptarse a los cambios en el ambiente sin la intervención humana. En particular, Uno de sus componentes principales es el componente MAPE-K del gestor autonómico, que le permite realizar un proceso de retroalimentación para responder de manera autónoma a las necesidades que se le presentan. La Computación Autonómica ha sido usada para auto-gestionar capacidades como: Auto-configuración, Auto-reparación, Auto-Optimización y Auto-Protección.

En el caso de las Arquitecturas SOA, se han empleado middlewares denominados Enterprise Service Bus (ESB), que permite el despliegue e integración de servicios, a los cuales actualmente se le están incorporando componentes basados en Computación Autonómica (Autonomic Enterprise Service Bus) [9].

El principal problema con todas esas arquitecturas es que han sido implementadas como arquitecturas centralizadas o semi-centralizadas, que en el mejor de los casos están compuestas por componentes locales, distribuidos a través de los servicios que conforman la composición, los cuales son coordinados por un componente central, que obtiene una visión completa del problema para inferir una estrategia de acción. Esto trae como consecuencia el problema de la escalabilidad de las arquitecturas, cuando las aplicaciones contienen un gran número de componentes.

En este trabajo se desarrolla un componente MAPE-K como servicio para los gestores autonómicos de un middleware reflexivo basado en la computación autonómica. De esta manera, MAPE-K puede ser distribuido en cada servicio que forma parte de la composición de una aplicación SOA, e internamente cada componente de MAPE-K se construye como un servicio separado, distribuido y débilmente acoplado, dando una gran flexibilidad. Además, en este trabajo se prueba MAPE-K como servicio en un middleware reflexivo autonómico para la gestión de fallas en aplicaciones basadas en servicios, cuya arquitectura es completamente distribuida, tal que el diagnóstico y la reparación de fallas se realizan a través de la interacción de los MAPE-K presentes en cada servicio.

J. Vizcarrondo, Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL), Mérida, Venezuela, jvizcarrondo@cenditel.gob.ve

J. Aguilar, Investigador Prometeo en la Universidad Técnica Particular de Loja, y en la Escuela Politécnica Nacional, Ecuador, aguilar@ula.ve

E. Exposito, CNRS, LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes), Toulouse, Francia,

A. Subia, CNRS, LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes), Toulouse, Francia, subias@laas.fr

II. ESTADO DE ARTE

Se han propuesto varios middleware reflexivo para la gestión de aplicaciones. Por ejemplo, en [14] se propone un middleware reflexivo llamado SOAR, definido para la gestión de fallas en la composición de servicios, diseñado como una estructura global (centralizado) para controlar y adaptar el sistema completo. El middleware se compone de dos niveles de una torre reflectante: el primero (nivel base) es responsable de la descripción de las características básicas del sistema SOA (Aplicación SOA), y el segundo nivel (nivel meta) es responsable de la supervisión y la adaptación del sistema SOA. Para realizar la reflexión, el nivel meta realiza su conexión causal con las entidades de la aplicación SOA: consumidores, aplicación que sirve de intermediario (Bus), y proveedores. El nivel meta analiza los mensajes intercambiados por los servicios basado en criterios de QoS como cantidad de requerimientos exitosos y tiempo de respuestas (proceso de introspección). En el momento que sucede una degradación de QoS el middleware modifica las conexiones de la aplicación. La intersección del middleware se realiza haciendo uso de la unión de servicios web, al conectar o desconectar dinámicamente los servicios que forman parte de una tarea conjunta. Los experimentos realizados sobre esa arquitectura fueron solo considerando un servicio. [10] presenta un middleware reflexivo para adaptar redes de sensores heterogéneos, desplegados en escenarios dinámicos. Ese middleware maneja las necesidades de los usuarios, como misiones que la red debe cumplir. Estas misiones se traducen luego a parámetros en la red, y son distribuidos a través de procesos de razonamiento acerca de las capacidades de los nodos de la red y las condiciones del entorno. Ellos usan un enfoque multi-agente para llevar a cabo este razonamiento, así como las adaptaciones necesarias durante el tiempo de ejecución. En [2] han propuesto un middleware reflexivo basado en la computación en la nube, llamado CARMiCLOC (por sus siglas en inglés, Context Awareness Middleware in CLOud Computing), cuya tarea es monitorear el contexto para brindar una variedad de funciones y servicios según él. CARMiCLOC puede comportarse como un SaaS (Software as a Service) o como PaaS (plataforma como servicio).

En el mismo orden de ideas, [12] propone una arquitectura centralizada para las reparación de los servicios web, llamada arquitectura self-healing (auto-sanación). Para determinar la reparación del sistema SOA utiliza indicadores de calidad sobre los servicios web, los cuales son: disponibilidad, tiempo de respuesta, escalabilidad, entre otros. La arquitectura consta de tres módulos: el de seguimiento y medición (es responsable de la observación y el mantenimiento de los registros de los parámetros de QoS que son relevantes), el motor de diagnóstico y de estrategias de decisión (detecta la degradación del sistema e identifica los posibles planes de reparación), y finalmente, el módulo de reconfiguración (que implementa el plan de reparación).

En [3] y [7] definen los requisitos de reflexión para sistemas de software auto-gestionados, para adaptarse a las cambiantes condiciones ambientales. Define como la reflexión puede apoyar este tipo de sistemas auto-adaptativos, para conferirles la capacidad de razonar, comprender, explicar y modificar los requisitos en tiempo de ejecución. También, en [4] exploran la función de la reflexión computacional en el contexto de los sistemas de software auto-adaptativos.

Por otro lado, la computación autonómica ha sido usada previamente para realizar la auto-reparación en aplicaciones SOA. En [13] enumeran y clasifican los servicios necesarios en un middleware autonómico. Además, proponen un marco de referencia para generar middleware con bucles autonómicos, adaptable a aplicación en diferentes áreas, con diferentes limitaciones y necesidades. En [24] se propone una arquitectura centralizada basada en Computación Autonómica llamada JOpera project, encargada de realizar la auto-reparación a una composición SOA. Para esto, define un lazo de control MAPE encargado de balancear la carga, realizando una configuración autonómica de la composición. El sistema recolecta la información de toda la aplicación para verificar que se encuentra balanceado. La política de optimización es determinada por un grupo de criterios (por ejemplo, carga de trabajo), con el fin de realizar una auto-configuración del sistema.

En [5] describen un middleware autonómico para sistemas embebidos automotrices. Hacen especial énfasis en la necesidad de permitir una flexible reconfiguración automática en tiempo de ejecución, considerando el medio ambiente (sensible al contexto), y el rendimiento del automóvil. El middleware permite la auto-sanación para manejar tiempo de ejecución de los fallos detectados En [25] se ha propuesto un middleware reflexivo para el diagnóstico de fallas en la composición de servicios en aplicaciones SOA, llamado ARMISCOM (por sus siglas en inglés, Autonomic Reflective MIddleware for Management Service COMposition). ARMISCOM es totalmente distribuido, tal que no es coordinado por ningún diagnosticador global, los diagnóstico de fallas se realizan a través de la interacción de los diagnosticadores presente en cada servicio, y las estrategias de reparación se definen a través del consenso de los reparadores igualmente presentes en cada servicio.

Otra arquitectura centralizada ha sido implementada en un bus de servicios para monitorear QoS [9], la cual puede ser convertida en descentralizada al colocar instancias del ESB para sub-flujos de la composición, logrando de esta forma obtener una federación de ESBs que monitorean sub-flujos. Ahora bien, con el enfoque centralizado no pudieron lograr el diagnóstico como un todo de la aplicación, por los problemas de escalabilidad propios de las arquitecturas centralizadas. En el otro caso, se divide la composición en sub-flujos, y se asignan instancias del ESB para monitorearlos, logrando una federación de ESB que funciona de manera des-centralizadas, contrarrestando los problemas de escalabilidad.

Otras implementaciones distribuidas han sido desarrolladas en [11] y [22], colocando lazos MAPE para auto-reparar cada servicio web, activando o desactivando servicios.

III. BASES TEORICAS

A. *Computación Autonómica*

La Computación Autonómica [15] es un modelo de computación auto-gestionada inspirada en el sistema nervioso del ser humano. Permite crear sistemas que son capaces de auto-administrarse con una gestión de alto nivel, manteniendo la complejidad del sistema invisible para el usuario. Incorpora sensores y efectores a los sistemas para permitir la recogida de datos sobre el comportamiento del sistema y actuar en consecuencia. La Computación Autonómica define una arquitectura compuesta de 6 niveles:

- Los recursos gestionados: puede ser cualquier tipo de recurso (hardware o software), y pueden tener atributos de autogestión incrustados.
- Puntos de contacto: implementa los mecanismos de sensores y/o actuadores en los recursos gestionados.
- Gestor Autonómico: implementa los lazos de control inteligentes que automatizan las tareas de auto-regulación de las aplicaciones. Básicamente, está compuesto por cuatro partes llamadas MAPE: Monitoreo de eventos/datos desde las interfaces de los sensores, Análisis de la información recuperada, Planificación de las acciones a realizar, y Ejecución de las mismas (enviar las órdenes a los componentes del sistema) a través de los actuadores.
- Orquestación de los gestores autonómicos: Proporciona la coordinación entre los Gestores autonómicos locales
- Administrador Manual: crea una interfaz humano-computador para que los gestores autonómicos.
- Fuentes de conocimiento: Proporciona acceso a los conocimientos de acuerdo a las interfaces definidas en la arquitectura.

B. *ARMISCOM*

ARMISCOM (Autonomic Reflective MIddleware for management Service COMposition) es un middleware reflexivo autonómico, cuya capacidad reflexiva le permite controlar y cambiar el comportamiento de las aplicaciones SOA ejecutándose sobre él, en particular, cuando las mismas presentan fallas [25]. De esta manera, ARMISCOM tiene un comportamiento dinámico y adaptativo y puede gestionar programas que son capaces de cambiar de forma dinámica o evolucionar. En general, los dos procesos clásicos de reflexión en este caso consisten en [20]:

- Introspección: consiste en analizar el intercambio de mensajes entre los servicios que forman parte de la composición y los componentes del sistema SOA. Para alcanzar un nivel adecuado de introspección ARMISCOM debe observar tanto el sistema SOA como la aplicación SOA ejecutándose en él. Para lograr esto, se utilizan las especificaciones estándar para servicios web y los servicios SOA: WSDL (por sus siglas en inglés, Web Services Description Language), UDDI (por sus siglas en inglés, Universal Description, Discovery and Integration), y OWL-S (marcado semántico para servicios web).
- Intersección: es la capacidad de ARMISCOM para modificar su propia ejecución. Para esto, modifica el flujo de la composición de los servicios, reescribiendo las especificaciones de la aplicación: WC-DL, enlaces dinámicos (dinamic bidding), entre otros.

En particular, ARMISCOM está distribuido completamente a través de todos los servicios del sistema, con el fin de tener una visión más cercana (local) de la ocurrencia de los eventos que ocurren en los servicios que componen una aplicación. ARMISCOM se divide en los clásicos dos niveles de todo middleware reflexivo (ver Fig. 1):

- Nivel Base: En este nivel se ejecuta la composición de servicios que forman parte de una aplicación SOA, y se sigue el conjunto de normas y definiciones que rigen esas interacciones (Sistema SOA).
- Nivel Meta: En este nivel se encuentra la capacidad reflexiva del middleware. Para ello, en este nivel se hacen las tareas de introspección, y se determinan las operaciones de intersección a realizar. En particular, en este nivel se despliega gran parte de la arquitectura autonómica del middleware, que se presenta en la siguiente sección, para analizar el estado actual y determinar las tareas de tolerancia a fallas a realizar.

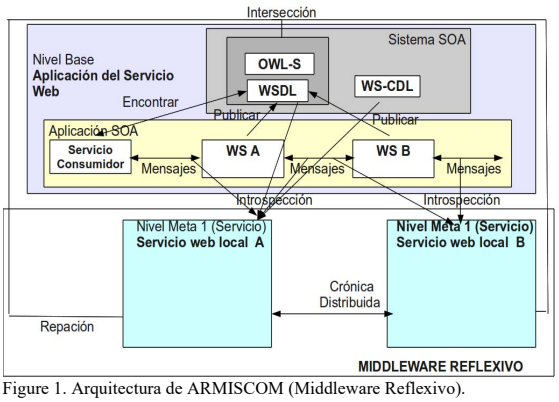


Figure 1. Arquitectura de ARMISCOM (Middleware Reflexivo).

IV. MAPE-K COMO SERVICIO

Nosotros proponemos en este trabajo que todos los elementos del bucle de control del gestor autonómico, sean ofrecidos como servicios. Igualmente, extendemos a MAPE con la base de conocimiento que se esté usando (K), de tal manera de ofrecer como servicio a MAPE-K. En ese sentido, cualquier gestor autonómico del sistema puede solicitar alguna de las funcionales requeridas por el bucle de control.

Ahora bien, los servicios MAPE-K debe ser integrado en los dos niveles de un middleware reflexivo (niveles meta y base), lo cual se realiza de la siguiente manera:

- Todo el componente K y las fases AP de MAPE forman parte del nivel meta. En particular, A analiza el comportamiento del sistema, y en función de ello, P planifica las acciones a realizar. Estas dos tareas son más reflexivas, por eso acontecen en el nivel meta.
- Las fases M y E forman parte del nivel base. M observa el

sistema (introspección) y E ordena la ejecución del plan realizado (intersección). Esas son tareas realizadas sobre el sistema bajo supervisión, por eso acontecen en el nivel base.

- Por otro lado, si M y E tienen tareas de pre-procesamiento y preparación de datos, estas se realizarían en el nivel meta.

De esa manera, el Gestor autonómico está compuesto por dos módulos (A y P) y una fuente de conocimiento en el nivel meta, y eventualmente, M y E ejecutándose en los dos niveles (ver Fig. 2). Esa es la forma como se integra el lazo de control MAPE-K de la arquitectura de computación autonómica en el middleware reflexivo.

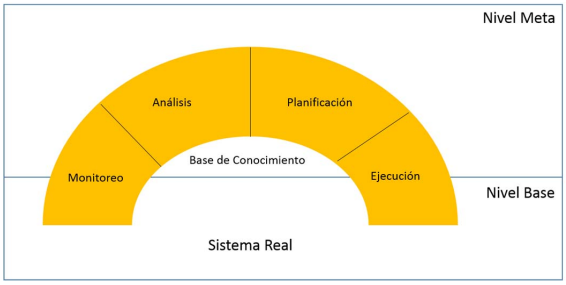


Figure 2. Arquitectura del Middleware basada en Computación Autonómica

Por otro lado, cada gerente autonómico trabaja a nivel local, requiriendo los servicios MAPE-K en un momento dado. Así, todos los componentes funcionales de MAPE-K son desarrollados como servicios, que pueden ser invocados por los gerentes autonómicos y componentes del middleware que conforman sus niveles meta y base, como servicios. Esto facilita la implementación en entornos distribuidos heterogéneos, y al ser débilmente acoplados, permite el desarrollo de los componentes MAPE-K de manera independiente (reusabilidad). Pero además, también facilita sus usos en otras aplicaciones no desplegadas en el middleware.

V. MAPE-K COMO SERVICIO EN ARMISCOM

A continuación vamos a especificar como se usaría MAPKE-K como servicio específicamente en ARMISCOM.

A. Componente Autonómico de ARMISCOM

En ARMISCOM, los diferentes niveles de una arquitectura autonómica tienen las siguientes características:

- Recursos administrados: corresponden a los servicios de la aplicación SOA que se está supervisando.
- Puntos de enlace: ARMISCOM intercepta los mensajes intercambiados en formas de eventos (sensores) y modifica el flujo de la aplicación supervisada, teniendo acceso de escritura a las especificaciones de la composición (actuadores).
- Gestor Autonómico: automatiza las tareas de auto-regulación de la aplicación SOA, para esto, el gestor autonómico se distribuye en cada servicio que forma parte de la composición (un gestor por cada servicio) y demanda los servicios de MAPE-K como servicios.
- Fuentes de Conocimiento: ARMISCOM utiliza las crónicas (son patrones temporales de fallas, ver [27] para más detalles) para realizar el diagnóstico, una ontología para inferir los mecanismos de reparación a aplicar, y una metadata para ejecuta la reparación.
- Orquestador gestores autonómico: Permite la iteración de los gestores autonómicos, en particular la comunicación de las crónicas distribuidas.
- Manual Manager (Administrador manual): provee la interfaz para modificar las crónicas y la manualmente metadata.

Por otro lado, ARMISCOM integra en los dos niveles del middleware (meta y base) los 6 niveles de una arquitectura de computación autonómica, de la siguiente manera (ver Fig. 3):

Los Manejadores de recursos y los puntos de enlace forman parte del nivel base del middleware; el Manejador Autonómico, la Fuente de Conocimiento y el Orquestador de los Gerentes Autonómicos forman parte del nivel meta. En los puntos de enlace se realiza la interacción con los servicios, con el fin de recuperar los datos de seguimiento (introspección) y ejecutar las decisiones de reparación definidas (intersección). Por otro lado, el Manejador Autonómico está compuesto por dos componentes, un diagnosticador y un reparador, los cuales son equivalentes a la estructura MAPE de la arquitectura clásica de computación autonómica. El diagnosticador observa el sistema y analiza las fallas, y el reparador define el plan de reparación y ordena su ejecución.

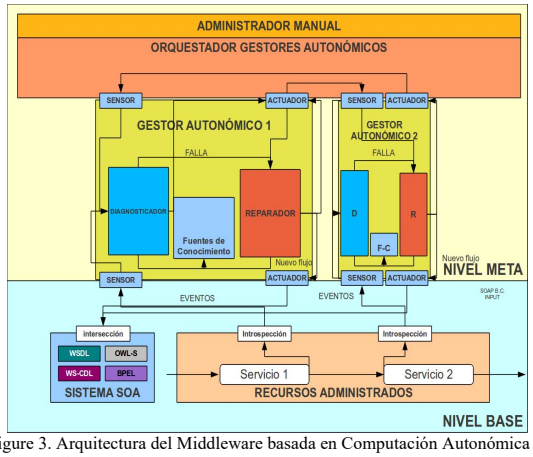


Figure 3. Arquitectura del Middleware basada en Computación Autonómica de ARMISCOM.

B. MAPE-K como servicio en ARMISCOM

Como se comentó en la sección anterior, el Gestor autonómico está compuesto por dos módulos en el nivel meta: el diagnosticador y el reparador, y una fuente de conocimiento, que corresponden a la estructura MAPE-K de la arquitectura de computación autonómica. Cada gerente autonómico trabaja a nivel local (en cada servicio), para analizar las fallas internas del servicio, y a partir de las interacciones con los otros gestores autonómicos, analiza globalmente las fallas en las aplicaciones basadas en servicios. Por lo tanto, este componente debe

comunicarse con el resto de los diagnosticadores de los otros servicios en la composición. Este mismo comportamiento sucede con el reparador.

Por otro lado, OpenESB es un bus de servicios, que permite el despliegue de aplicaciones SOA, facilitando el desarrollo de aplicaciones flexibles con bajo acoplamiento [21]. Además, es *"Open Source"*, y esta implementado según la especificación JSR208 (Java Specification Request, http://www.jcp.org/en/jsr/detail?id=208).

En el caso de ARMISCOM, los componentes que conforman su nivel meta son desarrollos como servicios. A continuación se presenta la implementación de ARMISCOM en OpenESB.

El Servicio Diagnosticador

La tarea principal del componente Diagnosticador es realizar el diagnostico de situaciones anormales que acontecen en la composición de los distintos servicios presentes en una aplicación. Para ello, el Diagnosticador es el encargado de verificar el conjunto de llamadas y respuesta de los servicios, los cuales serán las fuentes de eventos que utilizarán las crónicas distribuidas para el reconocimiento de patrones temporales. Así, el Diagnosticador usa el paradigma de crónicas para caracterizar los patrones de fallas a detectar y para reconocerlos [1] (ver Tabla I). En particular, usa el formalismo de crónicas distribuidas definido en [27], para definir un conjunto de patrones distribuidos para el diagnóstico de fallas en la composición de servicios.

Para el reconocimiento usa el procesador inteligente de eventos (IEP, por sus siglas en inglés) de OpenESB, el cual se basa en el lenguaje declarativo CQL (lenguaje continuo de consulta) para realizar consultas continuas en un flujo de eventos, lo que permite el procesamiento de eventos complejos (CEP) [6], [26].

En particular, el servicio Diagnosticador cuenta con una interfaz de entrada que capta los eventos que ocurren en la composición. Adicionalmente, posee dos interfaces de salida, que permiten la comunicación con los otros Diagnosticadores para poder comunicar eventos, y con los reparadores para informar el diagnóstico de las fallas. A continuación se presenta la descripción del servicio Diagnosticador y de sus interfaces:

TABLA I
OPERACIONES DEL SERVICIO DIAGNOSTICADOR

Operación	Interfaz de Entrada			Interfaz de Salida		
	Protocolo	Variables	Descripción	Protocolo	Variables	Descripción
Diagnoser events	Protocolo de Servicios (SOAP), BC:SOAP	Evento		Protocolo de Servicios (SOAP), BC:SOAP	Evento Enlazador	
		Id	Identificador único que permite diferenciar eventos que no forman parte de la misma operación.		id	Representa el estado final de la composición luego de cambiar su flujo.
		Event	Nombre del evento.		event	Nombre del evento.
		time	Tiempo de ocurrencia del evento.		time	Tiempo de ocurrencia del evento.
		extras	Otros atributos que se consideren necesarios para realizar el reconocimiento del evento.		extras	Otros atributos que se consideren necesarios para realizar el reconocimiento de los eventos.
				Protocolo de archivos BC:FILE	Fault	
					id	Identificador único que permite diferenciar eventos que no forman parte de la misma operación.
					fault	Nombre de la falla.
					faulttype	El tipo de la falla. Aplicable en algunos casos en el que se necesita obtener más información acerca de la falla, tal como QoS, SLA, entre otros.
					time	Tiempo de ocurrencia de la falla.
					extras	Otros atributos que se consideren necesarios para realizar el reconocimiento de los eventos.

El servicio Diagnosticador, en la operación llamada "Diagnoser events", recibe los eventos usando la interfaz de entrada *Evento*. En el momento que se produce el arribo de un evento, se pone en funcionamiento el motor de inferencia para el reconocimiento de las crónicas distribuidas. Al realizarse el reconocimiento de una crónica, el Diagnosticador expresa ese reconocimiento. Para esto, el servicio cuenta con una interfaz de salida compuesta por 2 mensajes: *Evento Enlazador* en el caso de que necesite comunicarse con otro Diagnosticador, y *Falla* para informar al reparador local. En la Fig. 4 se presenta la implementación del servicio Diagnosticador en el middleware ARMISCOM.

Como se muestra en la Fig. 4, el Diagnosticador se coloca en cada instancia del gestor autonómico de ARMISCOM (servicio de la aplicación SOA desplegada sobre ARMISCOM), recibe como entradas el conjunto de eventos que están aconteciendo en el servicio local de la composición (paso (1) Eventos) y los "eventos enlazadores" generados por otros Diagnosticadores (paso (4) Evento Enlazador), utilizando para todo esto la interfaz de entrada con el mensaje "event". Cada vez que un nuevo evento llega, el Diagnosticador activa el motor de

inferencia de crónicas usando el módulo IEP. Cada vez que una crónica es reconocida, el Diagnosticador puede tener dos salidas: un mensaje "evento enlazador" (ver paso (3) Evento Enlazador), que corresponde a los eventos que utilizan otros Diagnosticador para poder realizar un reconocimiento local de las fallas (este se ha desarrollado utilizando el protocolo SOAP); o un mensaje "falla" (ver paso (2) Falla), que permite informar a los Reparadores que ha ocurrido una falla (este se ha desarrollado con el protocolo para la gestión de archivos FILE).

El Servicio Reparador

El componente Reparador es el encargado de realizar los ajustes en la composición cuando acontece una falla. El Reparador tiene 2 sub-servicios:

Sub-servicio Planificador.

Usa una ontología llamada "Fault Recovery", que permite relacionar las fallas que pueden ocurrir en la composición de servicios con los posibles mecanismos de corrección que puedan aplicarse según la falla. Posteriormente, con el resultado generado por la ontología, se realiza una búsqueda en una meta-data basada en regiones, que permite al middleware seleccionar los métodos de corrección a aplicar en las regiones problemáticas. Esa metadata ha sido almacenada previamente. Así, el sub-servicio Planificador contiene 2 operaciones, ver Tabla II.

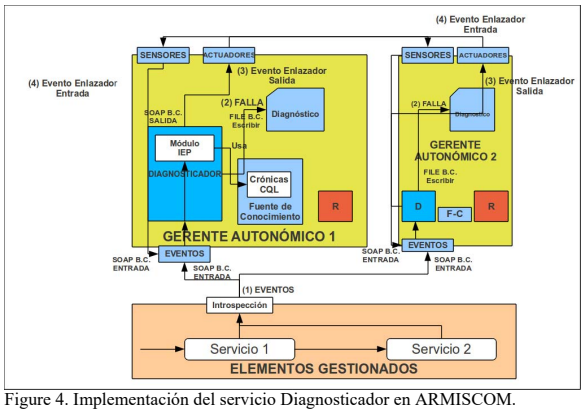


Figure 4. Implementación del servicio Diagnosticador en ARMISCOM.

TABLA II
OPERACIONES DEL SUB-SERVICIO PLANIFICADOR

Operación	Interfaz de Entrada			Interfaz de Salida		
	Protocolo	Variables	Descripción	Protocolo	Variables	Descripción
Repair Method	Protocolo de archivos, BC:FILE	Fault	Nombre de la falla	Protocolo de Servicios (SOAP), BC:SOAP	Possible Methods	Contiene los distintos métodos que se podrían aplicar para corregir la falla suministrada por la interfaz de entrada.
Resolution plan	Protocolo de Servicios (SOAP), BC:SOAP	Possible Methods	Los posibles métodos de corrección que se pueden aplicar a una falla.	Protocolo de Servicios (SOAP), BC: SOAP	Resolution plan	Contiene el conjunto de operaciones a realizar para solventar la falla.
		event_init	El lugar de inicio del flujo que se desea modificar en la composición.			
		event_end	El último servicio que se desea modificar en la composición.			

El sub-servicio planificador, en la operación "Repair Method" recibe el nombre de la falla que se encuentra en un archivo. Con el nombre de esta falla, el sub-servicio infiere el conjunto de métodos de solución que pueden ser aplicados para resolver la falla. La operación "Repair Method" utiliza la ontología "Fault-Recovery" para correlacionar las fallas y los métodos de reparación. Para la implementación del servicio web en OpenESB se realizó un programa en JAVA, utilizando el motor de inferencia FACT++ (es un razonador basado en expresiones en lógica de descriptiva, Cubre los lenguajes OWL y OWL2).

Por otro lado, la operación "Resolution plan" recibe como entrada el conjunto de métodos que se pueden implementar (Possible Methods) y la parte del flujo de la composición que se encuentra afectada por la falla; una vez invocado, la operación realiza la búsqueda de los métodos que se encuentran disponibles haciendo uso del campo operations (operaciones) de la meta-data, y devuelve la acción que mejor se ajuste al requerimiento (Resolution plan). En la Fig. 5 se muestra la implementación del sub-servicio planificador en ARMISCOM.

Como se muestra en la Fig. 5, la operación "Repair Method" recibe como entrada el diagnóstico del diagnosticador (ver paso (1) Fault), que se encuentra en forma de archivo. Para esto, extrae el nombre de la falla (fault) del mensaje de salida Fault de la operación "Diagnoser events" del diagnosticador (es el mensaje de entrada en la operación "Repair Method"). Después, realiza la inferencia de los métodos que se pueden implementar para solventar este tipo de falla, usando el motor de inferencia ontológica FACT++, retornando los métodos posibles a implementar (ver paso (2) Posibles métodos). El resultado de la operación "Repair Method" se suministra como entrada a la operación "Resolution plan", agregando los atributos Flow_init y Flow_end tomados del mensaje generado del diagnóstico de la falla por el Diagnosticador. Con esta información, la operación busca en la meta-data el mecanismo de solución que mejor se ajusta a la falla, y responde con el mensaje de salida "Resolution plan" (ver paso (3) Plan de Resolución).

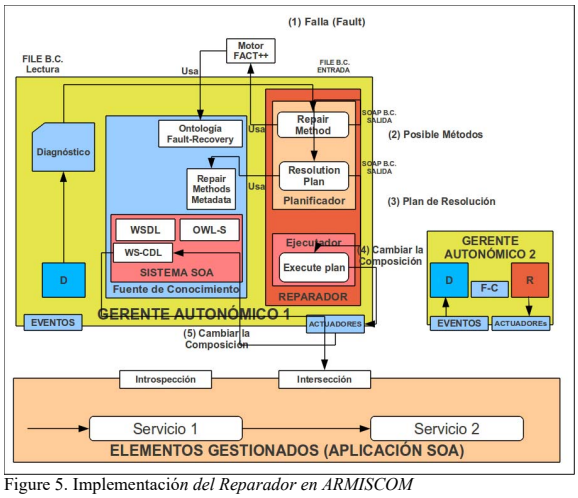


Figure 5. Implementación *del Reparador en ARMISCOM*

Sub-servicio Ejecutador

Ejecutador es el encargado de realizar los cambios en la composición. Para esto, ARMISCOM modifica la aplicación SOA usando el contenido de la nueva composición. El sub-servicio Ejecutador cuenta con una sola operación para realizar eso, llamada execute plan (ver Tabla 3).

La operación "Execute plan" se encarga de realizar las modificaciones en el flujo de la composición de los servicios. Para esto, recibe como entrada el conjunto de operaciones (plan) que se desea ejecutar, modifica la composición de los servicios según lo establecido en las operaciones de la metadata, y luego retorna un mensaje con el contenido previuosstate y currentstate, describiendo las modificaciones realizadas. En la Figura 5 se muestra que la operación "Execute plan" recibe como entrada el mensaje obtenido previamente por la operación "Resolution plan" del sub-servicio Planificador (ver paso (3) Plan de Resolución), y ejecuta las operaciones para modificar el flujo de ejecución de la composición (modificar el documento WS-CDL, ver pasos (4) y (5) Cambiar la Composición).

TABLA III
OPERACIONES DEL SUB-SERVICIO EJECUTADOR

Operación	Interfaz de Entrada			Interfaz de Salida		
	Protocolo	Variables	Descripción	Protocolo	Variables	Descripción
Execute plan	Protocolo de Servicios (SOAP), BC:SOAP	Resolution plan	Contiene el conjunto de operaciones a realizar para modificar la composición de servicio, esta información es derivada de la metadata (ver Figura 3.6).	Protocolo de Servicios (SOAP), BC:SOAP	previuosstate	Describe el estado en que se encontraba el flujo antes de realizar la modificación.
					currentstate	Representa el estado final de la composición luego de cambiar su flujo.

El Servicio Fuente de Conocimiento

La fuente de conocimiento está compuesta por una *ontología* que contiene, entre otras cosas, la taxonomía de fallas y de mecanismos de reparación para aplicaciones SOA; una *base de crónicas distribuida*, que almacena los patrones genéricos de fallas; y *metadata* que describe los mecanismos de reparación disponibles en cada sitio

El marco ontológico de ARMISCOM permite la gestión de todo el conocimiento requerido por el middleware, y está compuesto por (ver Fig. 6):

- La plataforma SOA: en particular, por
  - Web Services Description Language (WSDL): Proporciona un modelo para describir cómo los servicios pueden ser llamados, que parámetros son esperados, y qué funcionalidades ofrecen.
  - Web Services Choreography Description Language (WS-CDL): define un modelo para describir la Coreografía de Servicios Web.
  - Semantic Markup for Web Services (OWL-S): es una ontología que describe semánticamente a los servicios Web [23], a partir de la cual se automatizan las tareas de descubrimiento, invocación, composición, y seguimiento de los servicios web.
- Base de Datos de Crónicas Distribuidas: almacena al conjunto de crónicas que definen los patrones de fallas de los servicios web. Es utilizada principalmente por el Diagnosticador.
- La Ontología Fault-Recovery: define al conjunto de mecanismos de recuperación de fallas, estableciendo las relaciones entre las fallas y los métodos de reparación. Esta ontología se basa en los trabajos [8] y [19].
- La metadata "Service repair methods": es usado para almacenar los métodos de reparación disponibles para un servicio o flujo de la composición.

El componente Fuente de Conocimiento caracteriza todas las fuentes de conocimiento que utiliza ARMISCOM. En lo que se refiere a su implementación, se usa el lenguaje CQL para representar las crónicas, y Protege (interfaz para definir ontologías, escrita en el lenguaje Java) para representar la ontología "Fault Recovery".

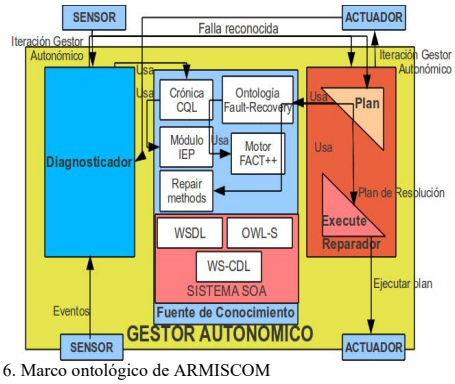


Figure 6. Marco ontológico de ARMISCOM

V. CASO DE ESTUDIO

A. Descripción General del caso de estudio

El escenario seleccionado para poner a prueba nuestra propuesta es la industria del mueble (Furniture Manufacturing) [16]. La industria del mueble tiene como objetivo la optimización de las redes de suministro y producción bajo demanda para el sector de la decoración [17]. La industria del mueble no funciona de una manera única, y en cada nivel de la cadena de suministro hay empresas que trabajan con diferentes procesos. La descripción de los actores que intervienen son:

- Comprador Final (End Customer): Cualquier persona que tiene la intención de amoblar una habitación o una casa.
- Tienda (Retail): tiendas de muebles, que reciben solicitudes del cliente final y envían solicitudes a uno o más Fabricante de muebles.
- Fabricante de Muebles (Furniture Manufacturer): Es el fabricante de muebles, recibe una orden de un minorista (tienda de muebles), y no vende directamente al cliente final. La Tienda puede tener varios Fabricante de muebles que reciben sus requisitos, y selecciona uno de acuerdo con las respuestas emitidas sobre los acuerdos en los criterios de calidad de los productos, costo y tiempo de entrega.

Las interacciones derivadas de la composición de servicios para el caso de estudio, se muestran en la Fig. 7:

(1) Enviar Requerimiento: el comprador envía la lista de productos requerida a la tienda.

(2) Petición de Productos: la tienda envía la petición de productos al fabricante. Esto involucra las órdenes de todos los clientes.

(3) Recibir Productos: la tienda recibe los productos del fabricante.

(4) Enviar Productos: la tienda envía los productos requeridos al comprador.

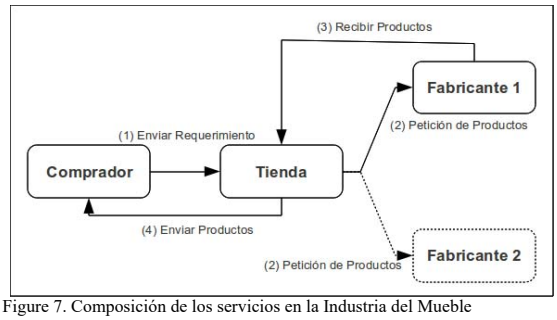
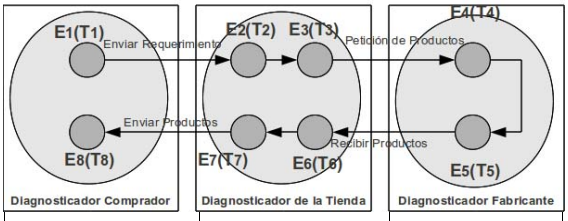


Figure 7. Composición de los servicios en la Industria del Mueble

Ahora se caracterizan la distribución de los eventos entre los diferentes diagnosticadores (sitios) que forman parte de la composición, para construir una crónica genérica para esta aplicación (conectando todos los eventos que pueden ocurrir, ver Fig. 8). Desde esta crónica genérica se deriva cada crónica específica, que describa cualquier situación anormal que se quiera detectar. Por razones prácticas, se considera que el tiempo es medido en segundos, y el retraso en las comunicaciones y el tiempo de reconocimiento de crónicas son insignificantes. La secuencia de eventos de esta crónica genérica se muestra a continuación:



Comprador	Tienda	Fabricante
• E1: Envía el requerimiento de productos a la Tienda • E8: Recibe los productos de la Tienda.	• E2: Recibe el requerimiento de los productos del Comprador. • E3: Envía la solicitud de los productos al Fabricante. • E6: Recibe los productos terminados del Fabricante. • E7: Envía los productos al Comprador,	• E4: Recibe la lista de productos requeridos de la Tienda. • E5: Una vez que los productos son terminados son enviados a la Tienda.

Figure 8. Eventos de la Composición de la Industria del Mueble por diagnosticador

B. Implementación de la aplicación de la Industria del Mueble en ARMISCOM

Para verificar el funcionamiento de ARMISCOM en el servicio Fabricante, se agregaron las operaciones adicionales siguientes para inducir fallas de SLA de Calidad (Quality):

- tuneQualityBehavior: Usada para generar la falla en la operación E5. Si está activada (tuneQualityBehavior = TRUE), la operación E5 produce menos productos de calidad que los que se necesitan (PRvar = PSvar - 1); en caso contrario, la operación E5 funciona normalmente.
- turnprovider: Operación propia del servicio Tienda para definir el servicio Fabricante a invocar. Si la operación

turnprovider es invocada con un valor FALSE, la composición es invocada con Fabricante 1, en caso contrario (valor TRUE) se invoca al Fabricante 2 (se supone solo dos fabricantes). El valor inicial de esta operación es FALSE (invoca por defecto al Fabricante 1).

En esta sección se pondrá a prueba la capacidad de ARMISCOM para realizar la auto-reparación en la composición de servicios. Para esto, es necesario conectar los distintos servicios MAPE-K de la arquitectura autonómica en su implementación en OpenESB.

Para probar el mecanismo de auto-reparación de ARMISCOM, se toma la crónica de "violación de SLA para la calidad de los productos" (todos los productos solicitados por la Tienda deben tener una calidad establecida y debe ser igual a la calidad de los productos recibidos por parte del fabricante) de la Fig. 9.

```
Subchronicle Tienda calidad(SLA)
{
 Events{
  event(E3, T3)
  event(E6, T6)
  event(BESLAQuality,
TSLAQuality)
 }
 Constrains{
  T6 - T3 > 0
  T6 ≤ TSLAORQoS
 }
 When recognized{
  repair invoke(Tienda, 'Service
Level Agreement')
 }
}
```

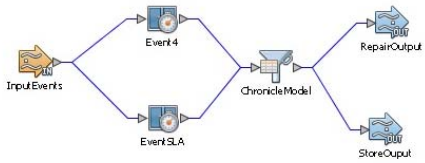
```
Subchronicle Fabrica calidad(SLA) {
 Events{
  event(E4, PSvar T4)
  event(E5, PRvar, T4)
 }
 Constrains{
  T5 - T4 > 0
  E4.PSvar > E5.PRvar
 }
 When recognized{
  emit event(BESLAQuality,
TSLAQuality) to Tienda diagnoser
 }
}
```

Figure 9. Patrón de Crónica distribuida para la aplicación Fábrica de Muebles para violación de SLA de la calidad de los productos

La implementación de la crónica del diagnosticador en el componente IEP de OpenESB se muestra en la Fig. 10.

Como se muestra en la figura anterior, el Diagnosticador de la Tienda maneja dos relations de eventos E4 y ESLA (evento enlazador generado en la sub-crónica del Fabricante), y genera 2 eventos al momento del reconocimiento de la crónica en forma de archivos: RepairOutput (utilizado por el reparador para solventar la falla) y StoreOutput (usado como log y permite mostrar los resultados del reconocimiento). Por otro lado, el Diagnosticador del Fabricante reconoce los eventos E5 y E6, y genera al momento de su reconocimiento el evento enlazador BESLA ToRetail (provee una llamada SOAP al Diagnosticador en la Tienda), y el mensaje ManufacturerOutput (usado como un registro de incidencias o log). Por otro lado, el único método de reparación de los servicios disponibles en cada sitio en este caso de estudio, y su meta-data se muestra en la Tabla 4 y Fig. 11.

Diagnosticador en la Tienda usando el componente IEP



Diagnosticador en el Fabricante usando el componente IEP

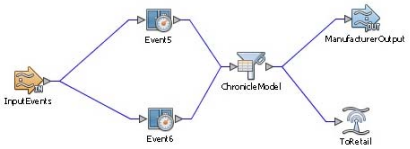


Figure 10. Diagnosticador de la Tienda y el Fabricante usando el componente iep de OpenESB

TABLA IV
MÉTODOS DISPONIBLES PARA REPARAR LA APLICACIÓN DE LA INDUSTRIA DEL MUEBLE

Métodos de Reparación Disponibles						
Id	Weight	RepairMethod	Transition	Event_init	Event_end	Operations
1	1	substituteflow	{Ø}	E4	E5	turnprovider

El método de reparación "substituteflow" en la meta-data consiste en considerar al servicio del Fabricante 2 en la composición (sustituir al servicio por defecto Fabricante 1) cuando sucede una falla en Fabricante 1. Para esto, utiliza la operación disponible en la aplicación de la Industria del Mueble en OpenESB. La implementación del componente Reparador de la instancia de la Tienda en OpenESB se muestra en la Fig. 10; su funcionamiento se ejecuta en tiempo real: lee el diagnóstico de la falla en el puerto newWSDL_inboundPort, luego realiza la consulta al servicio ontología con el contenido de la falla usando el puerto OntologyPort. Seguidamente, realiza la consulta de la metadata internamente, y genera la corrección invocando la operación turnprovider (puerto RetailPort), almacenando la reparación en un archivo usando el puerto outputfile_OutboundPort.

C. Aplicación de la Prueba de Auto-sanación en OpenESB

Para verificar el funcionamiento del mecanismo de auto-reparación de ARMISCOM, se utilizó la operación llamada tuneQualityBehavior dentro de cada servicio Fabricante 1 y 2 (Manufacturer 1 y 2), para inducir la falla de violación de SLA de Calidad de los Productos. Se ejecuta la aplicación la Industria del Mueble siete veces (7), añadiendo un atributo id para diferenciar cada invocación. En la Tabla 5 se muestran las

diferentes invocaciones, con los diferentes valores utilizados para id, Quality, y tuneQualityBehavior.

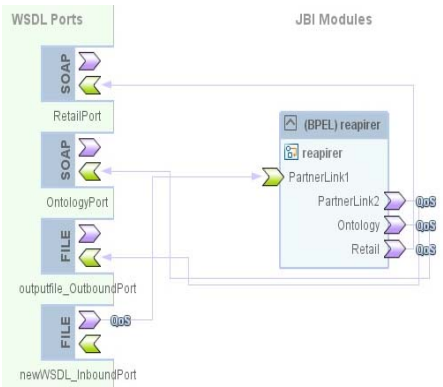


Figure 11. Métodos disponibles para reparar la aplicación de la Industria del Mueble.

TABLA V
INVOCACIONES DE LA APLICACIÓN INDUSTRIA DEL MUEBLE

ID	tuneQualityBehavior		Resultado de la Operación (Calidad)								
	Manuf1	Manuf2	E1	E2	E3	E4 (PSvar)	E5 (PSvar) Manuf1	E5' (PSvar) Manuf2	E6	E7	E8
1	FALSE	FALSE	7	14	14	14	14	-	14	14	14
2	FALSE	FALSE	1	7	7	7	7	-	7	7	7
3	TRUE	FALSE	5	5	5	5	4	-	4	4	4
4	FALSE	FALSE	16	16	16	16	-	16	16	16	16
5	FALSE	TRUE	14	14	14	14	-	13	13	13	13
6	FALSE	FALSE	2	2	2	2	2	-	2	2	2
7	FALSE	FALSE	10	10	10	10	10	-	10	10	10

Para cada sub-crónica que es reconocida, los eventos enlazadores y el diagnóstico de fallas generados se almacenan en los archivos (ManufacturerOutput.xml y RepairOutput.xml). Adicionalmente, el componente de reparación genera un archivo llamado RetailRepairResult.xml, donde almacena el resultado de aplicar el mecanismo de auto-reparación. Así, el contenido de los archivos generados por el reconocimiento de fallas por cada sub-crónica, y el método de reparación implementado para corregir las fallas, para la aplicación Industria del Mueble, se muestra en la Fig. 12. Para facilitar la lectura de la información generada por los eventos enlazadores, el diagnóstico de las fallas y la reparación de las fallas, el contenido de los archivos se describe a continuación:

- Evento enlazador (primera o tercera fila, y primera columna, de la Fig. 12):
  - id: Corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 4).
  - event: Indica el evento enlazador que es generado cuando la sub-crónica es reconocida.
  - time: Corresponde al tiempo en milisegundos cuando es generado el evento enlazador.
  - timestamp: corresponde al tiempo cuando el evento enlazador es generado, expresado en formato de tiempo entendido por los humanos.
  - lp: La lista de productos (lp) cuando el evento enlazador es generado.
- Diagnóstico de fallas (primera o tercera fila, y segunda columna, de la Fig. 12):
  - id: corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 3).
  - fault: Nombre de la falla que es diagnosticada (ej: ServiceLevelAgreement), y que es usada como fuente para el servicio gestor de la ontología.
  - faulttype: corresponde al tipo de falla, es usado generalmente para agregar información en el diagnóstico (ej: Quality).
  - time: corresponde al tiempo en milisegundos cuando es generado el diagnostico.
  - lppre: La lista de productos (lp) en un evento previo al diagnóstico, y añade información al diagnóstico (ej: el lp en el evento E5).
  - lp: La lista de productos (lp) que tiene el evento enlazador que género el diagnóstico.

- event_init: El evento inicial del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la meta-data (Ej: event_init = E4).
- event_end: El evento final del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la meta-data (Ej: event_end = E5).

- Reparación de la Falla (segunda o cuarta fila, y segunda columna, de la Fig. 12):
  - id: corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 4).
  - fault: Nombre de la falla que es reparada (ej: SLA), que es usada como fuente para el servicio gestor de la ontología.
  - measure: Agrega información acerca de la métrica de la falla que es reparada (Ej: Quality).
  - methodspossibles: Los posibles métodos de reparación que pueden ser aplicados por el reparador, separados por punto y coma (;), y corresponde a la consulta al servicio que gestiona la ontología (Ej: parametersUpdate;substitute;substituteFlow).
  - methodrepair: Indica el método de reparación que se ha aplicado para corregir la aplicación (Ej: substituteFlow).
  - repair_state: Indica si la falla fue reparada: TRUE si se corrigió y FALSE si no fue posible reparar la aplicación. En algunos casos no es posible reparar la aplicación si los métodos de reparación que se puedan aplicar (methodspossibles) no se encuentran en la meta-data.
  - time: corresponde al tiempo en milisegundos cuando es generado la reparación.
  - previuosstate: Agrega información del estado en que se encontraba la aplicación antes de la reparación.
  - currentstate: Agrega información del estado en que se encuentra la aplicación durante la reparación.

Id	Fabricante (Output)	Tienda (Output)
	Diagnosticador <?xml version="1.0" encoding="utf-8"?> <msgns:ManufacturerOutput_MsgObj xmlns:msgns="ManufacturerChronicle_iep"> <id>3</id> <event>BESLA</event> <time>1410352813133</time> <Timestamp>2014-09-10 T08:10:13.239-04:30</Timestamp> <lp>4</lp> </msgns:ManufacturerOutput_MsgObj> ManufacturerOutput.xml	Diagnosticador <?xml version="1.0" encoding="utf-8"?><msgns:StoreOuput_MsgObj xmlns:msgns="RetailChronicle_iep"> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <faulttype>Quality</faulttype> <lpretail>4</lpretail> <lp>5</lp> <event_init>E4</event_init > <event_end>E5</event_end> <time>1410352813133</time> </msgns:StoreOuput_MsgObj> RepairOutput.xml
3	-	Reparador <?xml version="1.0" encoding="utf-8"?> <msgrepair> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <measure>Quality</measure> <methodspossibles>parametersUpdate;substitute;substituteFlow</methodspossibles> <methodrepair>substitute</method> <repair_state>true</repair_state> <time>1410352815444</time> <previuosstate>Furniture Manufacturer 1</previuosstate> <currentstate>Furniture Manufacturer 2</currentstate> </msgrepair> RetailRepairResult.xml
5	Diagnosticador <?xml version="1.0" encoding="utf-8"?> <msgns:ManufacturerOutput_MsgObj xmlns:msgns="ManufacturerChronicle_iep"> <id>5</id> <event>BESLA</event> <time>1410353325526</time> <lp>13</lp> <Timestamp>2014-09-10 T08:18:45.632-04:30</Timestamp> </msgns:ManufacturerOutput_MsgObj> ManufacturerOutput.xml	Diagnosticador <?xml version="1.0" encoding="utf-8"?><msgns:StoreOuput_MsgObj xmlns:msgns="RetailChronicle_iep"> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <faulttype>Quality</faulttype> <lpretail>13</lpretail> <lp>14</lp> <event_init>E4</event_init > <event_end>E5</event_end> <time>11410353325526</time> </msgns:StoreOuput_MsgObj> RepairOutput.xml
	Reparador	Reparador <?xml version="1.0" encoding="utf-8"?> <msgrepair>

```
<id>5</id>
<fault>ServiceLevelAgreement(SLA)</fault>
<measure>Quality</measure>

<methodspossibles>parametersUpdate;substitute;substituteFlow</m
ethodspossibles >
<methodrepair>substitute</method>
<repair_state>true</repair_state>
<time>1410353328495</time>
<previuosstate>Furniture Manufacturer 2</previuosstate>
<currentstate>Furniture Manufacturer 1</currentstate>
</msgrepair>
```

RetailRepairResult.xml

Figure 12. Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación Industria del Mueble

Como se muestra en la Fig. 12, cuando el funcionamiento de la aplicación de la Industria del Mueble es normal (id = {1, 2, 4, 6, 7}, ver Table 5) ninguna sub-crónica es activada. De lo contrario, en la invocación id = {3, 5} (ver Table 5) se genera el reconocimiento de la falla de violación de SLA para la calidad de los productos. En ambos casos, los eventos generados son almacenados en el archivo ManufacturerOutput.xml (emite el evento enlazador BESLA hacia la Tienda (ver Fig. 12, Diagnosticador Fabricante, id = 3, primera fila y segunda columna), que es leído por el diagnosticador en la Tienda en su sub-crónica, y emite evento con el reconocimiento de la falla (archivo RepairOutput.xml, ver Fig. 12, segunda columna y fila, diagnosticador Tienda, id = 3).

Posteriormente, con el reconocimiento de la falla el componente Reparador en la Tienda obtiene el diagnóstico del archivo generado por el diagnosticador (RepairOutput.xml), y genera las operaciones necesarias para solventar la falla en la composición (operación), guardando el resultado de la reparación en el archivo RetailRepairResult.xml (ver Fig. 12, segunda columna y fila, reparador Tienda, id = 3).

Para hacer más comprensible las reparaciones realizadas por el reparador de la Tienda mostradas en el archivo RepairOutput.xml en ambas invocaciones (id = 3 y 5), la Fig. 13 muestra gráficamente los cambios realizados en la composición. La aplicación comienza con la aplicación con el servicio Fabricante 1, en el instante en que la falla de violación de SLA para la calidad sucede y es reconocida en la ejecución con id = {3} (ver Tabla 5 y Fig. 11), el servicio Fabricante 1 es sustituido por Fabricante 2. Posteriormente, en la invocación con id = {5} el servicio Fabricante 2 es sustituido por el servicio Fabricante 1.

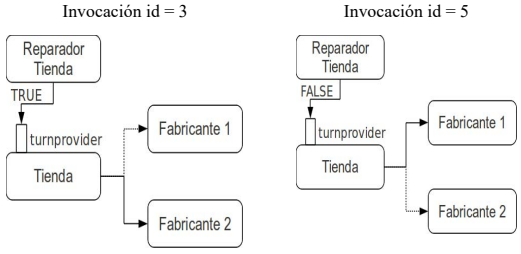


Fig. 13. Reparaciones realizadas por el componente Reparador de la Tienda

D. Aplicación de la Prueba de Auto-sanación en OpenESB

En este experimento se ha mostrado el funcionamiento de todos los componentes de ARMISCOM y su acoplamiento para realizar la gestión automática de fallas en la composición de servicios. Así, en el caso de la Industria del Mueble se ha podido diagnosticar y corregir las fallas de violación de SLA para el caso calidad (Quality) en dos oportunidades, haciendo uso de los patrones de crónicas diseñadas para tal fin. Al momento que el Servicio diagnosticador en la Tienda reconoce la falla, genera un archivo con el contenido del diagnóstico, que es usada por el Servicio reparador en la Tienda para realizar la corrección, que en este caso se ejecuta usando el método de substitución del flujo de la composición.

Así, en este experimento se logró implementar y acoplar todos los componentes del Gestor Autonómico de las distintas instancias de ARMISCOM como servicio, usando OpenESB.

IV. CONCLUSIONES

Al ser ARMISCOM un middleware para la gestión de fallas en aplicaciones SOA, el desarrollo de todas las instancias del gestor autonómico están diseñadas para ser desplegadas como un servicio. Como se mostró, todos los componentes que forman parte de las diferentes instancias locales del gestor autonómico fueron desarrollados e implementados como un servicio web, con interfaces bien definida que puede ser invocadas con el protocolo SOAP. Adicionalmente, los componentes encargados para realizar la interfaz en la consultas a las fuentes de conocimiento (ontología Fault-Recovery y metadata) también siguen la arquitectura de servicios.

En general, las pruebas realizadas en este trabajo sobre la composición de las instancias MAPE-K que componen el gestor autonómico (funcionamiento y acoplamiento de las distintas instancias) fueron exitosas.

El desarrollo de MAPE-K siguiendo el paradigma SOA, en cada instancia del Gestor Autonómico de ARMISCOM, hace que este último funcione con componentes distribuidos débilmente acoplados, lo que le da a ARMISCOM una gran flexibilidad. Así, cada gestor autonómico fue construido para funcionar como una composición de un conjunto de servicios, en específico: Diagnosticador, Reparador y Gestores de Conocimiento, cada uno con propiedades para funcionar independientemente de los demás, permitiendo una fácil adaptación de la arquitectura MAPE-K si se requiere, por ejemplo, cambiar el Diagnosticador si no se está satisfecho con el Diagnosticador basado en crónicas. Así, pueden implementarse otros servicios para el diagnóstico de fallas con otros paradigmas, y solo sería necesario reemplazar el servicio

Diagnosticador por el nuevo componente en la aplicación SOA que define la composición del MAPE-K.

Por otro lado, cada servicio del MAPE-K puede funcionar en una composición distinta al gestor autonómico. Por ejemplo, el servicio diagnosticador puede ser usado independientemente de los demás por una aplicación dada. Para este caso, se puede tener una aplicación SOA y se le interconecta el flujo de eventos en su composición con el sistema de reconocimiento de crónicas del componente Diagnosticador, lo que le permitirá realizar el diagnóstico de fallas.

AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por el Proyecto UTPL titulado: "Medios de Gestión de Servicios (Middleware) Inteligentes para Entornos de Aprendizaje Virtual". Dr. Aguilar ha sido parcialmente financiado por el Proyecto Prometeo del Ministerio de Educación Superior, Ciencia, Tecnología e Innovación de la República de Ecuador).

REFERENCIAS

[1] J. Aguilar "Temporal Logic from the Chronicles Paradigm: learning and reasoning problems, and its applications in Distributed Systems", *Ed. Lambert,* 2011.

[2] J. Aguilar, M. Jerez, E. Exposito, T. Villemur, "CARMiCLOC: Context Awareness Middleware in Cloud Computing", *Proc. Latin American Computing Conference (CLEI),* pp.1-10. 2015.

[3] M. Alaya, T. Monteil, K. Drira, T. Guérout, "A Framework to Create Multi-domains Autonomic Middleware", *Proc. Eighth International Conference on Autonomic and Autonomous Systems*, pp. 14-17, 2012.

[4] J. Andersson, R. Lemos, S. Malek, D. Weyns, "Reflecting on self-adaptive software systems," *Proc. International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 38-47, 2009.

[5] R Anthony, D Chen, M. Törngren, D. Scholle, M. Sanfridson, A. Rettberg,, T. Naseer,, M. Persson, L. Feng, "Autonomic Middleware for Automotive Embedded Systems", En *Autonomic Communication (A. Vasilakos et all. eds.),* pp. 169-210, 2009

[6] A.Arasu, S. Babu and J. Widom, "The CQL continuous query language: semantic foundations and query execution". *International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121-142, 2006.

[7] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, E. Letier. "Requirements reflection: requirements as runtime entities". *Proc.32nd ACM/IEEE International Conference on Software Engineering, vol 2,* pp. 199-202, 2010.

[8] K. S. Chan, J. Bishop, L. Baresi and J. Steyny, "A Fault Taxonomy for Web Service Composition". *Proc. Service-Oriented Computing Workshops,* 2007.

[9] C. Diop "An Autonomic Service Bus for Service-based Distributed Systems". PhD dissertation, Université de Toulouse, Toulouse – France, 2015.

[10] E Freitas, M. Wehrmeister, C. Pereira, A. Ferreira,T. Larsso. "Multi-Agents Supporting Reflection in a Middleware for Mission-Driven Heterogeneous Sensor Networks", *Proc. Eighth International Conference on Autonomous Agents and Multiagent Systems*, 2009.

[11] C Ghedira and Z. Maamar. "Towards a Self-Healing Approach to Sustain Web Services Reliability". *Proc. IEEE Workshops of International Conference on Advanced Information Networking and Applications,* pp. 267-272. 2011.

[12] R Halima, E Fki, K Drira and M. Jmaiel. "Experiments results and large scale measurement data for web services performance assessment". *Proc. IEEE Symposium on Computers and Communications*, pp. 83-88. 2009.

[13] J. Hoffert, A. Gokhale, D. Schmidt, "Timely Autonomic Adaptation of Publish/Subscribe Middleware in Dynamic Environments", *Technical Report,* Vanderbilt University, 2011. Recuperado de: http://www.dre.vanderbilt.edu/~jhoffert/ADAMANT/IJARAS-ADAMANT.pdf

[14] G Huang, X Liu and H. Mei. "SOAR: Towards Dependable Service-Oriented Architecture via Reflective Middleware". *International Journal of Simulation and Process Modelling*, vol. 3, no. 1/2, pp.55-65. 2007.

[15] IBM Corporation. "An architectural blueprint for autonomic computing". Autonomic Computing", Fourth Edition, Recuperado de: http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf, 2006.

[16] Imagine project. 2014. Innovative end-to-end Management of Dynamic Manufacturing Networks. Recuperado de: http://www.imagine-futurefactory.eu/index.dlg.

[17] Imagine project. 2014. Furniture Manufacturing Description. Recuperado de:http://furnituremanufacturing.imagine-futurefactory.eu/articles/furniture-manufacturing-

[18] M. Josuttis. "SOA in Practice The Art of Distributed System Design". O'Reilly, 2007.

[19] A. Liu, Q. Li, L. Huang, and M. Xiao, "FACTS: A Framework for Fault-Tolerant Composition of Transactional Web Services", *IEEE Transactions on Services Computing*, vol.3, no. 1, pp. 46-59, 2010.

[20] P. Maes. "Concepts and Experiments in Computational Reflection". Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 147-155. 1987

[21] OpenESB community web site, 2015. Retrieved October 10, 2015 from: http://www.open-esb.net/

[22] H Psaier, L Juszczyk, F Skopik, D Schall and S. Dustdar. "Runtime Behavior Monitoring and Self-Adaptation". *Proc. 4th IEEE International Conference in Service-Oriented System*s. pp. 164-173, 2010.

[23] S. Poonguzhali, R. Sunitha, and G. Aghila, "Self-Healing in Dynamic Web Service Composition", *International Journal on Computer Science and Engineering*, vol. 3, no. 5. Pp. 2054-2060. 2011.

[24] A Sherif and A. Z. Gurguis. "Towards autonomic web services: achieving self-healing using web services". *Proc. workshop on Design and evolution of autonomic application software*, pp. 1-5. 2005.

[25] J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias, "ARMISCOM: Autonomic reflective middleware for management service composition," *Proc. Global Information Infrastructure and Networking Symposium (GIIS),* pp.1-8, 2012.

[26] J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias, "Building Distributed Chronicles for Fault Diagnostic in Distributed Systems using Continuous Query Language (CQL)". *International Journal of Engineering Development and Research*, vol.3, no. 1, pp. 131-144, 2015.

[27] J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias, "Crónicas Distribuidas para el Reconocimiento de Fallas". *Ciencia e Ingeniería*, vol 36, no 3, pp. 73-84, 2015.



Juan Vizcarrondo, is System Engineer and obtained a Msc in Computer Science at the Universidad de los Andes, Mérida-Venezuela, Currently he's finished his studies PhD in Computer Science at the Universidad de los Andes. He works at the Cenditel since 2007.



Jose Aguilar, is a System Engineer graduated in 1987 from the Universidad de los Andes, Merida, Venezuela. M. Sc. degree in Computer Sciences in 1991 from the University Paul Sabatier-Toulouse-France. Ph. D degree in Computer Sciences in 1995 from the University Rene Descartes-Paris-France. He completed post-doctorate studies at the Department of Computer Science in the University of Houston, researcher at the Microcomputer and Distributed Systems Center (CEMISID) at the same university. Member of the Mérida Science Academy and the International Technical Committee of the IEEE-CIS on Artificial Neural Network.



Ernesto Exposito, earned his engineer degree in computer science from the "Universidad Centro-occidental Lisandro Alvarado" (Venezuela, 1994). He earned his PhD in "Informatique et Télécommunications" from the Institut National Polytechnique de Toulouse (France, 2003). He is Professor in computer sciences at the Institut National des Sciences Appliquées (INSA) of Toulouse.



Audine Subias, received a PhD degree in 1995 and a M.S.degree in 1992 in Informatique Industrielle, both from Paul Sabatier University, in Toulouse, France. Since 1997 she is Associate Professor in control and discrete event systems at the Institut National des Sciences Appliquées (INSA) of Toulouse.