Revista Venezolana de Computación
ISSN: 2244-7040
http://www.svc.net.ve/revecom
Vol. 2, No. 1, pp. 1-13, Diciembre 2015
Selección de los Mejores Artículos de CoNCISa 2015

CrESTa: Ejecución Tolerante a Fallas de Servicios Web Compuestos basado en Crónicas

Marcos Grillo[1], Yudith Cardinale[1], José Aguilar[2]
marcos.grillor@gmail.com, yudith@ldc.usb.ve, aguilar@ula.edu.ve

[1] Departamento de Computación, Universidad Simón Bolívar, Caracas, Venezuela
[2] Departamento de Computación, Universidad de Los Andes, Mérida, Venezuela

Resumen: Los Servicios Web Compuestos (SWC) facilitan la definición e implementación de procesos delicados y complejos de una forma confiable y versátil. Una vez que un usuario envía un requerimiento al sistema, durante la fase de composición se realiza el descubrimiento y la composición de varios Servicios Web, que en conjunto darán respuesta a la petición del usuario. Posteriormente un motor de ejecución realizará las invocaciones apropiadas de dichos Servicios Web para finalmente retornar la respuesta al usuario. En el área de motores de ejecución de SWC se han realizado diversas propuestas que incluyen sistemas centralizados o distribuidos, que implementan mecanismos de tolerancia a fallas para garantizar el funcionamiento y consistencia del sistema. En este artículo se propone CrESTa (*ChRonical-based Execution engine for composite web Service with fault TolerAnce*), un motor de ejecución tolerante a fallas, que modela los SWC con Redes de Petri Coloreadas, usa crónicas para detectar fallas y considera parámetros de calidad de servicio. Presentamos la arquitectura de CrESTa y mostramos su funcionamiento con un caso de estudio práctico.

Palabras Clave: Composición de Servicios Web; Tolerancia a Fallas; Crónicas; Calidad de Servicio.

Abstract: Composite Web Services (CWS) facilitates complex and delicate process definition and implementation. A user might query a system for a requirement which is processed and transformed in a CWS. After the processing phase, an execution phase is needed which is executed by an engine, whose only function is to call all the Web Services involved to generate an answer that will satisfy the query. Several proposal has been done in execution engines, which includes both centralized and distributed approaches, and almost everyone implements recovery mechanisms to maintain the system stability and consistency. This article proposes CrESTa (*ChRonical-based Execution engine for composite web Service with fault TolerAnce*), a CWS execution engine with recovery mechanisms, that uses Collored Petri Networks for CWS representation, chronicles to detect failures, and considers quality assurance parameters. We present the architecture, and the execution process through a practical example.

Keywords: Composite Web Services; Recovery Mechanisms; Chronicles; Quality of Service.

I. Introducción

Los Servicios Web (SW) han tenido un impacto contundente en nuestro día a día. Es muy común ver como universidades y empresas proveen servicios a través de la red para mejorar y optimizar procesos, permitiendo a las organizaciones procesar y comunicar datos sin la necesidad de conocer profundamente las tecnologías de cada organización [1]. Los SW han ganado considerable interés en investigaciones del sector académico y de la industria, dado que ofrecen un lenguaje neutral, débilmente acoplado, independiente de las plataformas y estandarizado para integrar aplicaciones y servicios dentro de organizaciones [2].

La composición de servicios web consiste en combinar SW desarrollados por diversas organizaciones que ofrecen diversas propiedades transaccionales, funcionales y calidad de servicio [3]. Diversas razones pueden influir en la necesidad de crear un Servicio Web Compuesto (SWC), tal como la creación de aplicaciones que agreguen diversos SW existentes para satisfacer requerimientos de negocio dinámicos [4]. Por ejemplo, suponga un usuario que desea comprar un boleto de avión y provee información respecto a ciertos parámetros tales como límite de precio, aerolínea favorita y no desea escalas (estos parámetros representan los requerimientos funcionales); se puede establecer un SWC que contenga diversos SW: uno para reservar el boleto, otro que contenga la conexión con su banco para el proceso de pago y otro que le envíe un correo electrónico con la confirmación de su compra. Si existen varios SW que ofrecen las mismas funcionalidades, la selección puede ser guiada por los requerimientos no funcionales del usuario (e.g., tiempo máximo de ejecución, confiabilidad de los SW); asï para nuestro ejemplo, se elige el mejor entre

todos los SW de aerolíneas disponibles (el que reporte menores tiempos de ejecución y tenga mayor confiabiliadad). Con este simple ejemplo se observa cómo se pueden resolver problemas complejos con una composición de SW ya disponibles [1]. El desarrollo de un SWC se puede descomponer en dos fases: composición y ejecución. Primero se debe definir la composición dinámica o estáticamente, dependiendo de los requerimientos del usuario (funcionales y no funcionales). Como resultado se obtiene un SWC conformado por varios SW, representado con una estructura como un grafo, un *workflow* o una red de Petri; esta estructura especifica un orden de ejecución particular establecido por restricciones de flujo de datos o de control. Para el ejemplo anterior, es preciso que primero se ejecute el SW que realiza la reserva del boleto áereo, luego el SW que procesa el pago y finalmente el SW que envía el mensaje de confirmación de compra (es decir, una ejecución secuencial). Para el proceso de definición del SWC se pueden utilizar tecnologías tales como WS-BPEL[1] que es una especificación hecha bajo el precepto de modelar fácilmente procesos de negocio como un SWC o WSDL[2], OWL-S[3] o UDDI[4] para definir, expresar información semántica o descubrir un SW, de tal forma que se pueda elegir una composición que satisfaga los requisitos del usuario.

Luego, en la fase de ejecución, se deben invocar los SW del SWC de acuerdo a la definición realizada en la fase previa de composición. Para ejecutar un SWC es necesario un programa o un sistema que se encargue de invocar los SW de acuerdo al orden establecido en la fase de composición y se asegure de cumplir todas las restricciones que correspondan al SWC. A este tipo de programas los llamaremos *motores de ejecución*. Hay 2 esquemas ampliamente utilizados para implementar motores de ejecución: motores centralizados, en donde un coordinador se encarga de toda la ejecución, y motores distribuidos, donde la ejecución se realiza con la colaboración de diversos participantes sin un coordinador central [1]. Si hay muchos SW en la composición y el flujo de datos o de control permite ejecuciones paralelas de varios SW, se podría tomar ventaja de un modelo distribuido, ya que se podría reducir el tiempo total de ejecución del SWC, siempre y cuando el tiempo que tome la sincronización posterior de los datos sea despreciable.

Durante la ejecución de un SWC, existen diversas situaciones que pueden causar la falla de uno o más SW. En el ejemplo de la compra de boletos mencionado con anterioridad, es posible que el servicio de cobro no esté disponible al momento de la compra, sin embargo, se podría reemplazar por otro SW de cobro que si esté funcionando, reintentar la operación o cancelar la compra completamente. Un SWC tolerante a fallas, es aquel que, aún en presencia de fallas finaliza la ejecución del SWC (e.g., reintentando la invocación, substituyendo o replicando el SW que falle) o abandona la ejecución asegurando un estado consistente del sistema (e.g., haciendo *roll-back* y compensando los SW que se ejecutaron exitosamente antes de la falla). En este sentido, la ejecución confiable de SWC se convierte en un mecanismo clave para enfrentar los retos de las aplicaciones de acceso libre y abierto en un ambiente dinámico y no confiable para asegurar consistencia del sistema aún en presencia de fallas [2].

Así la consideración del manejo de fallas debe atacarse en la fase de ejecución. Son los motores de ejecución los que deben encargarse de establecer mecanismos de recuperación ante fallas para asegurar la ejecución de la composición. En este contexto, las fallas durante la ejecución de un SWC pueden ser reparadas por recuperación hacia atrás (*Backward recovery*) o recuperación hacia adelante (*Forward recovery*). La recuperación hacia adelante implica deshacer todo el trabajo que se realizó exitosamente antes de la falla y retornar al estado inicial consistente (antes de que la ejecución del SWC comenzara), usando técnicas de *roll-back* y compensación. La recuperación hacia adelante trata de reparar la falla y continuar la ejecución; reintentar, replicación y substitución son algunas técnicas usadas. También se pueden utilizar propiedades transaccionales de los SW, que implícitamente describen su comportamiento en caso de fallas y aseguran la clásicas propiedades transaccionales ACID (*Atomicity, Consistency, Isolation, and Durability*).

Las propiedades transaccionales permiten recuperación hacia atrás y recuperacón hacia adelante [5]. Estos enfoques de recuperación aseguran la propiedad todo-o-nada, cuando ocurre una falla. Sin embargo, para algunas consultas de usuarios, obtener respuestas parciales puede tener sentido y utilidad. Por lo tanto, se requieren de estrategias de recuperación alternativas que provean una propiedad intermedia en caso de fallas. La técnica de puntos de control (*checkpointing*) puede ser implementada para retornar las respuestas parciales obtenidas hasta el momento de una falla y poder reiniciar la ejecución posteriormente para finalizar con la ejecución faltante [6][7].

El motor de ejecución también debe considerar los requisitos no funcionales que se establecieron al momento de la definición del SWC. Generalmente, estos requisitos no funcionales definen la calidad de servicio solicitada por el usuario en términos de tiempo total de ejecución, costos, confiabilidad, etc. Durante la composición se seleccionan los SW de acuerdo a sus parámetros de calidad reportados previamente por estimación usando técnicas análiticas, de simulación, heurísticas o por trazas de ejecuciones previas [8][9][10]. Estas consideraciones se deben tomar en cuenta durante la ejecución del SWC, de tal forma que se asegure el cumplimiento de las restricciones impuestas por el usuario en términos de calidad de servicio. En particular, el tiempo total de la

[1] Web Service Bussines Process Execution Language, 2007, https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm tomado en Diciembre de 2014. Es una especificación de OASIS que permite modelar SWC como orquestación enfocándose en procesos de negocio.

[2] Web Service Description Language, 2001, http://www.w3.org/TR/wsdl tomado en Diciembre de 2014, se puede ver como una firma de un servicio web

[3] Semantic Markup for Web Services, 2004, http://www.w3.org/Submission/OWL-S/ tomado en Diciembre de 2014, permite describir servicios web semánticos

[4] Universal Description Discovery and Integration, 2004, http://www.uddi.org/pubs/uddi_v3.htm tomado en Diciembre de 2014, es un mecanismo para registrar y localizar servicios web

ejecución y el tiempo promedio en la ejecución de un SW se pueden considerar para ofrecer una mayor compatibilidad con requisitos no funcionales preestablecidos. Estos requisitos se deben considerar incluso cuando hay una falla, ya que el objetivo primordial del motor es realizar la ejecución sin violar ningún tipo de restricciones predeterminadas. Durante la detección de fallas es necesario implementar un mecanismo que sea eficiente al tomar la decisión apropiada para realizar la reparación y que no altere significativamente los parámetros de calidad de servicio, como el tiempo de ejecución total, asegurando así la ejecución bajo los parámetros definidos.

El paradigma de crónicas ha sido usado para determinar fallas en sistemas dinámicos, permitiendo modelar las relaciones temporales entre eventos observables y describiendo los patrones de comportamiento del sistema. Una crónica es un conjunto de eventos relacionados con restricciones de tiempo que representan una interpretación de lo que está ocurriendo en el sistema estudiado en un momento dado [11]. En trabajos previos el paradigma de crónicas ha sido usado para detectar fallas en la ejecución de SWC [12].

Este trabajo propone un motor de ejecución, llamado CrESTa (*ChRonical-based Execution engine for composite web Service with fault TolerAnce*) que permite mejorar los tiempos de ejecución de un SWC utilizando un sistema distribuido en vez de uno centralizado, considerando mecanismos de tolerancia a fallas para la recuperación de la ejecución utilizando propiedades transaccionales y el tiempo de ejecución como aseguramiento de calidad de servicio. Adicionalmente se propone un modelo donde se busca evitar que los tiempos de detección de fallas desmejoren significativamente los tiempos de ejecución totales utilizando crónicas.

El trabajo está estructurado de la siguiente manera: en la Sección II se explica la representación de un SWC utilizando Redes de Petri Coloreadas y la detección de fallas utilizando crónicas; en la Sección III se describe la arquitectura de CrESTa, explicando el funcionamiento de sus componentes; la Sección IV presenta los trabajos relacionados a esta investigación; en la Sección V se da un caso de estudio donde se ejemplifica el funcionamiento de CrESTa; finalmente la Sección VI presenta nuestras conclusiones y trabajo futuro.

II. Tolerancia a Fallas en Servicios Web Compuestos

Algunos estudios recientes en el área de SWC se han centrado específicamente en mecanismos de detección inmediata de fallas en sistemas distribuidos y en medidas de recuperación en presencia de éstas. A continuación se presentarán los conceptos que se utilizarán para proponer la construcción de un motor de ejecución de SWC distribuido tolerante a fallas.

A. SWC Tolerantes a Fallas Basado en Redes de Petri Coloreadas

Algunos motores de ejecución son capaces de manejar fallas, ya sea con manejo de excepciones [13][14], basado en propiedades transaccionales [15][16], usando técnicas de replicación [17][18] o por combinación de varios enfoques [4][19]. En los trabajos iniciales de soporte de tolerancia a fallas de SWC, sólo se manejaban construcciones de programación de bajo nivel como el manejo de excepciones (por ejemplo in WSBPEL). El manejo de excepciones normalmente es especificado explícitamente en tiempo de diseño, definiendo cómo son manejadas las excepciones y qué comportamiento asumen los SWC cuando es recibida una excepción. Más recientemente, la confiabilidad de SWC ha sido manejada a un nivel de abstracción más alto, es decir a nivel de la estructura del flujo de ejecución tales como *workflows*, grafos o Redes de Petri, y con tecnología de métodos independientes como propiedades transaccionales y replicación.

Cardinale et. al. en [3] explica cómo las Redes de Petri son utilizadas ampliamente en el modelaje de SW ya que capturan tanto los flujos de ejecución de los SW como su naturaleza distribuida. Adicionalmente, en el mismo trabajo se extienden las Redes de Petri Coloreadas para modelar las propiedades transaccionales de los SW involucrados en una composición.

Las propiedades transaccionales más usadas para un SW son **pivote**, **compensable** y **reintentable**. Si s es un SW, s es **pivote** (p) si una vez ejecutado sus efectos son permanentes y no se pueden deshacer semánticamente, es **compensable** (c) si es posible deshacer semánticamente la ejecución de s (ya sea por la llamada a otro servicio o al mismo con diversos parámetros, para hacer una regresión), o **reintentable** (r) si garantiza una terminación satisfactoria luego de un número finito de reintentos. Hay combinaciones posibles de propiedades, un servicio **pivote** puede ser **reintentable** (pr), así como un servicio **compensable** puede ser también **reintentable** (cr), lo que quiere decir que las propiedades transaccionales de un servicio s pertenencen al conjunto $\{p, pr, c, cr\}$.

Un SWC puede adquirir una propiedad transaccional agregada en término de las propiedades transaccionales de sus SW componentes. Así, un SWC cs es **atómico** (a) si al terminar la ejecución de todos los SW involucrados los cambios son permanentes y no se pueden deshacer semánticamente, **compensable** (c) si todos los componentes son compensables y **reintentable** (r) si todos sus componentes son reintentables. De nuevo pueden haber combinaciones de propiedades, teniendo SWC **atómicos** y **reintentables** (ar), y **compensables** y **reintentables** (cr), por lo que todas las propiedades transaccionales de cs están en el conjunto $\{a, ar, c, cr\}$

Las técnicas de recuperación soportadas por los propiedades transaccionales de SWC son:

- Recuperación hacia atrás: consiste en restaurar el estado que el sistema tenía al comienzo de la ejecución del SWC; es decir, todos los efectos producidos por el SW que falle y por los SW previamiente ejecutados, son semánticamente deshechos por técnicas de *roll-back* o compesación.
- Recuperación hacia adelante: consiste en reparar la falla para permitir que el SW que falle, continúe su ejecución; reintento y substitución son técnicas usadas para proveer recuperación hacia adelante.

Dada la gran proliferación de SW publicados en Internet, existen servicios equivalentes diseñados/desarrollados inde-

pendientemente por organizaciones diferentes, que pueden ser usados como componentes alternativos reduntantes para ofrecer sistemas tolerantes a fallas. En este contexto, una réplica representa un SW funcionalmente equivalente. Las réplicas pueden ser usadas para replicación de ejecución o para substitución permitiendo recuperación hacia adelante, sin depender de propiedades transaccionales. Estos enfoques de recuperación aseguran la propiedad todo-o-nada, para asegurar la consistencia del sistema cuando ocurre una falla. Sin embargo, para algunas consultas de usuarios, obtener respuestas parciales puede tener sentido y utilidad. Por lo tanto, se requieren de estrategias de recuperación alternativas que provean una propiedad intermedia en caso de fallas. Cuando una falla ocurre, se generan puntos de control que contienen información de un estado avanzado de ejecución (incluyendo resultados parciales). Un SWC parcialmente ejecutado, contiene puntos de control que permite su reanudación posterior a partir de ese estado avanzado de ejecución [20][21].

Nuestro trabajo utiliza la defínción de la extensión de Redes de Petri Coloreadas presentada por Cardinale et. al. en [3], llamada WSDN por sus siglas en inglés de Red de Dependencia de Servicio Web (Web Service Dependency Net). Esta extensión es una tupla (A, S, F, ξ) donde:

- A es un conjunto finito no vacío, que contiene las entradas y salidas de los SW;
- S es un conjunto finito de transiciones, cada transición es un SW;
- $F : (AxS) \cup (SxA) \rightarrow \{0, 1\}$ indica la presencia (1) o ausencia (0) de de arcos entre las entradas/salidas (A) y las transiciones (S);
- ξ es una función de coloración tal que $\xi : C_A \cup C_S$ con $C_A : A \rightarrow \Sigma_A$, $C_S : S \rightarrow \Sigma_S$ y $\Sigma_A = \{I, a, ar, c, cr\}$, $\Sigma_S = \{p, pr, a, ar, c, cr\}$, que representan las propiedades transaccionales del SWC (representado en la coloración de los parámetros de entrada) y las propiedades transaccionales de los SW (representado en la coloración de las transiciones) respectivamente.

Ilustremos con un ejemplo un SWC basado en esta definición. Supongamos que tenemos un SWC que modela una compra de uno o varios productos utilizando 2 tarjetas de débito. Como los servicios de conexión con los bancos suelen tardar, se desea crear un SWC que maneje las 2 operaciones de cobro simultáneamente. La Tabla I muestra la descripción de los SW y parámetros de este ejemplo.

Tabla I: Ejemplo de SWC

SW	Entrada	Salida	Prop. Trans.
s1	Tarjeta1, Orden de Compra	Token1	cr
s2	Tarjeta2, Orden de Compra	Token2	cr
s3	Tarjeta1, Token1	Comprobante1	pr
s4	Tarjeta2, Token2	Comprobante2	pr
s5	Comprobante1, Comprobante2, Orden de Compra	Factura, Orden de entrega	pr

El proceso sería el siguiente: los servicios $s1$ y $s2$ pertenecen a bancos distintos y se encargan de verificar si la información proporcionada de las tarjetas de débito es correcta, también generan un token en caso de que se tengan los fondos necesarios para la operación. De ser así estos fondos son retenidos por una cantidad de tiempo definida por cada banco, este token permite identificar la transacción que reservó el dinero. Como estos servicios no generan cambios irreversibles, poseen la propiedad transaccional **compensable-reintentable**. Los servicios $s3$ y $s4$ reciben información de la tarjeta, el token que identifica la transacción y el dinero reservado en el banco; se encargan de descontar el monto correspondiente y emiten un comprobante de la transacción. Estos servicios son **pivotes-reintentables**, ya que una vez realizada la operación de cobro, ésta no se puede deshacer. Finalmente $s5$ toma los comprobantes y genera la factura de todo el proceso de compra; dado que la factura debe ser única y genera una orden de despacho, es un servicio tipo **pivote-reintentable**. Por las propiedades transaccionales de sus SW, el SWC sería ar, dado que todos sus componentes son reintentables.
Se puede tener una representación gráfica de este SWC, tal como se muestra en la Figura 1.

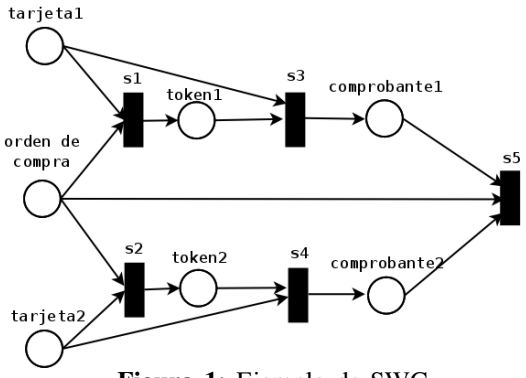

Figura 1: Ejemplo de SWC

De esta forma se puede representar de una forma robusta y sencilla todo lo necesario para la ejecución del SWC.

B. Detección de Fallas

En un escenario donde haya distintos elementos distribuidos ejecutando partes de un SWC, si uno de los elementos falla, es necesario que se active algún mecanismo de sincronización para que todos las partes se pongan de acuerdo sobre cuál va a ser el procedimiento a seguir para la recuperación de la falla. En este sentido, es necesario encontrar una forma de detectar inmediatamente la falla y disparar acciones que se encarguen de manejar la situación. Para este fin se pueden utilizar las **crónicas**.

Una crónica es un conjunto de eventos relacionados con restricciones de tiempo que representan una interpretación de lo que está ocurriendo en el sistema estudiado en un momento dado [11]. Las crónicas se basan en lógica temporal y son muy útiles para modelar sistemas dinámicos y con dependencia de ocurrencia de eventos. En [12], Vizcarrondo et. al. explican cómo cada crónica representa una situación normal o anormal, algún escenario o patrón de comportamiento. Estos eventos que conforman la crónica son observables y en el momento de su ocurrencia pueden generar acciones u otros eventos. Le Guillou et. al. en [22] definen una crónica C como un par

(E, T) donde E es el conjunto de eventos y T es el conjunto de restricciones entre los tiempos de ocurrencia de los eventos. Estas restricciones son las que permiten establecer un orden respecto a lo que debe ocurrir primero y lo que debe ocurrir después para detectar la crónica.

En [12] una crónica se representa con el siguiente modelo:

```
Cronica{
  Eventos{
    evento(e1, T1),
    evento(e2, T2),
    evento(e3, T3)
  }
  Restricciones{
        T2 − T1 < C1,
        T3 − T2 < C2
  }
  acciones{
        accion1,
        accion2
  }
}
```

Donde:

- e_i representan los eventos para el reconocimiento de las crónicas.
- T_i son los puntos en el tiempo en la ocurrencia de los eventos.
- Restricciones: conjunto de restricciones entre los T_i.
- C_i Constantes que representan la diferencia entre la ocurrencia de dos puntos en el tiempo.
- Acciones: conjunto de acciones a ser ejecutados al momento de reconocer la crónica.

Supongamos que queremos definir una crónica que nos permita detectar la ocurrencia de un error en la ejecución de un SW. Definamos el conjunto de eventos E:

$$E = \{inicio, llamada, culminacion, error\}$$

Donde $inicio$ corresponde a la inicialización del SW, $llamada$ corresponde a la ejecución del SW, $culminacion$ indica la terminación correcta, y $error$ representa la falla en la ejecución. Definamos la crónica de la siguiente forma:

```
Cronica{
  Eventos{
    evento(inicio, T1),
    evento(llamada, T2),
    evento(error, T3)
  }
  Restricciones{
    T2 − T1 > 0,
    T3 − T2 > 0
  }
  acciones{
     atenderFalla
  }
}
```

Esta crónica permite reconocer una ejecución fallida de un SW si ocurre un error luego de la $llamada$. T_1, T_2 y T_3 establecen el orden de los eventos, y al detectarse la crónica se llamará a $atenderFalla$. Esta acción, entonces decidirá, por ejemplo, de acuerdo a las propiedades transaccionales del SW, si el SW se puede reinvocar (si es **reintentable**) o hay que hacer *roll-back* (si es **pivote**); o simplemente informar sobre el error ocurrido.

De forma análoga, se puede definir una crónica que detecte un funcionamiento correcto de una ejecución de SW de la siguiente forma:

```
Cronica{
  Eventos{
    evento(inicio, T1),
    evento(llamada, T2),
    evento(culminacion, T3)
  }
  Restricciones{
    T2 − T1 > 0,
    T3 − T2 > 0
  }
  acciones{
    culminacionExitosa
  }
}
```

En este caso se llama a $culminacionExitosa$ al reconocer la crónica, y aunque no es común este tipo de detección, en un sistema distribuido podría servir para sincronizar toda la información entre los SW participantes, obtener el resultado final de la ejecución y entregar los parámetros de salida finales al usuario.

Vizcarrondo et. al. en [12] definen una extensión de este concepto para permitir reconocer eventos en un entorno distribuido, a esto lo llama **crónica distribuida**, y consiste en la descomposición de una crónica con eventos $E = \{E_1, \ldots, E_N\}$ y restricciones T en subcrónicas, donde cada evento local pertenece al proceso o sitio de estudio. Formalmente: sea SC_i una subcrónica con eventos $E_i = \{E_{i1}, \ldots, E_{iM}\}$ y restricciones T_i, donde $E_i \subset E$ y $T_i \subset T$, se tiene que para todos los procesos o sitios i se cumple que:

$$C(E, T) = UNION_{i=1\ldots N}(SC_i(E_i, T_i))$$

Donde $UNION$ representa la unión de todas las subcrónicas cumpliendo las condiciones $E = \bigcup_{i=1}^{N} E_i$ y $T = \bigcup_{i=1}^{N} T_i$. Para este caso, el reconocimiento de la crónica global se puede llevar a cabo en las crónicas locales a través de **eventos enlazadores**, que una vez reconocidos comunican a los otros sitios o procesos la detección de la subcrónica.

Ilustremos con un ejemplo este concepto. Supongamos que tenemos una crónica C con los eventos: $E = \{e_1, \ldots, e_7, EE_{sc1}\}$. Supongamos que esta crónica está conformada por 2 subcrónicas SC_1, $SC2$, $C = UNION(SC_1, SC_2)$ donde $E_1 = \{e_1, e_2, e_6, e_7\}$ y $E_2 = \{e_3, e_4, e_5, EE_{sc1}\}$. EE_{sc1} es el evento enlazador entre el sitio 1 y el sitio 2. La definición de estas subcrónicas se presentan en la Tabla II.

Tabla II: Definición de Subcrónicas

$SC_1\{$
Eventos{
evento(e_1, T_1),
evento(e_2, T_2),
evento(e_6, T_6),
evento(e_7, T_7)
}
Restricciones{
$T_2 - T_1 > 0$,
$T_6 - T_2 > 0$,
$T_7 - T_6 > 0$
}
acciones{
emitir EE_{sc1} a SC_2
}
}
	$SC_2\{$
	Eventos{
	evento(e_3, T_3),
	evento(e_4, T_4),
	evento(e_5, T_5),
	evento(EE_{sc1}, T_{sc1})
	}
	Restricciones{
	$T_4 - T_3 > 0$,
	$T_5 - T_4 > 0$,
	$T_{sc1} - T_5 > 0$
	}
	acciones{
	culminacionExitosa
	}
	}

En la Figura 2 se puede observar un gráfico de la crónica distribuida. La ocurrencia de los eventos sigue siendo determinada por los T_i, solo que, para este ejemplo, la subcrónica 2 (SC_2) es reconocida luego de que haya ocurrido EE_{sc1}.

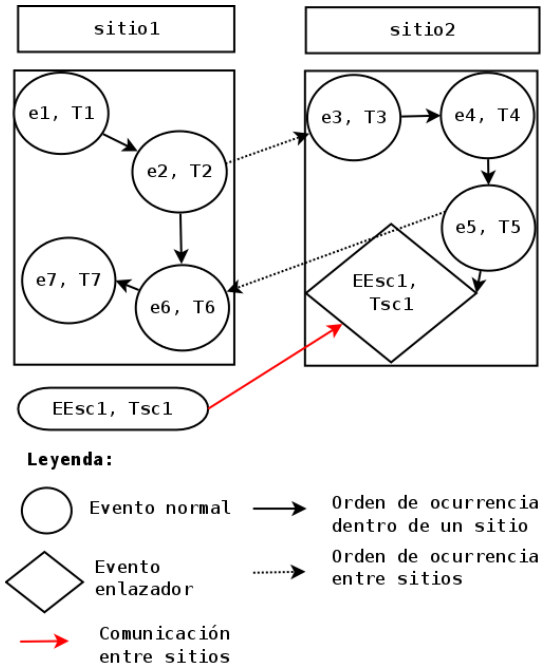

Figura 2: Ejemplo de Crónica Distribuida

Para la ejecución de SWC, se pueden crear crónicas que modelen la falla de los componentes del sistema distribuido, de tal forma que si hay un error en una parte del sistema, inmediatamente se pueda sincronizar la información necesaria y tomar decisiones inmediatas sobre qué mecanismo de recuperación se va a ejecutar. Entonces las acciones relacionadas con detección de falla ($atenderFalla$), implementarían algortimos para decidir de acuerdo a las condiciones actuales de la ejecución (e.g., tiempos de ejecución transcurridos), a las propiedades transaccionales de los SW y a otras técnicas de recuperación disponibles, cuál es la mejor estrategia de reparación aplicable (recuperación hacia adelante o hacia atrás), manteniendo la calidad de servicio exigida inicialmente por el usuario.

III. Arquitectura de CrESTa

CrESTa (*ChRonical-based Execution engine for composite Web Service with fault TolerAnce*) es un motor de ejecución distribuido para SWC, conformado por diversas instancias que pueden ejecutarse en sitios remotos sobre Internet y se comunican para llevar a cabo su funcionalidad. En la Figura 3 se muestra una visión global de CrESTa. Cada instancia de CrESTa, que de ahora en adelante llamaremos Servicio de Ejecución (SE), recibe por HTTP[5] las peticiones tanto de ejecución de SWC como otras necesarias dentro del sistema. Al momento de ejecutar un SWC recibe su representación como una Red de Petri Coloreada, los requisitos no funcionales de calidad de servicio, y de ser necesarias, las definiciones de crónicas distribuidas para la detección de fallas. El SE que haya recibido esta información comenzará con la ejecución y cuando sea necesario enviará esta información a otros SE para ejecutar de forma paralela el SWC. Una vez finalizado el proceso, si no hay fallas, se envía la respuesta.

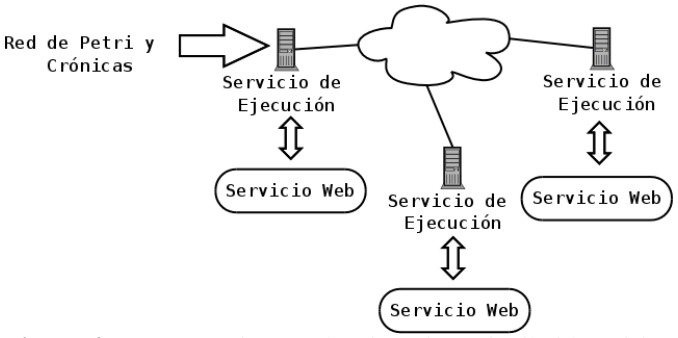

Figura 3: CrESTa: Sistema de Ejecución Distribuido, Visión Global

Cada SE va a estar compuesto por 4 módulos y tendrá una base de datos local donde se guardará información sobre la ejecución que sea necesaria por los módulos y se tenga conocimiento sobre el estado general del servicio en cualquier momento. En la Figura 4 se muestra un gráfico con los módulos propuestos.

Las operaciones que puede ejecutar un SE son las siguientes:

1) Descubrimiento: consiste en detectar otros SE que estén operativos, esto se hace para incrementar la escalabilidad del sistema, pues agregando más SE se podrán manejar más solicitudes de ejecución. Se tiene una lista estática con SE que estén siempre activos, y al conectarse con estos SE se envían la información de los otros SE que ya estén operativos.
2) Chequeo: permite detectar si un SE está operativo.
3) Información de carga: el SE informa al solicitante sobre su estado y cuánto trabajo tiene pendiente.
4) Actualizar información de servicio: permite actualizar la información que se tenga de otros SE que formen parte del sistema.
5) Ejecución de SWC: toma un SWC representado como una Red de Petri Coloreada y lo ejecuta paralelizando

[5]Hypertext Transfer Protocol https://tools.ietf.org/html/rfc7230 tomado en Abril de 2015 protocolo de comunicación

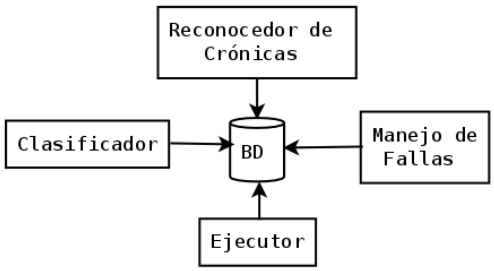

Figura 4: Arquitectura de un Servicio de Ejecución (SE) de CrESTa

peticiones cuando sea posible.

6) Ejecución de tolerancia a fallas: considera las propiedades transaccionales del servicio donde se generó la falla para determinar cuál es el mejor mecanismo de tolerancia a aplicar, ya sea, recuperación hacia atrás (utilizando compensaciones), o recuperación hacia adelante (utilizando substitución o reintento del SW que haya fallado), todo dentro de los parámetros de calidad de servicio establecido por el usuario.
7) Sincronización: permite actualizar la información de un SWC que esté ejecutándose con la colaboración de varios SE al mismo tiempo.

Estas operaciones se desarrollan con la colaboración entre los módulos. A continuación se explica cada componente de un SE y las operaciones que realiza.

A. Módulo de Clasificación

Se encarga de recibir peticiones de servicios relacionados con los puntos 1 al 5, listados en la sección anterior. Atiende las solicitudes relacionadas a información del servicio (operaciones 1, 2, 3 y 4) y en cada caso responde inmediatamente la información solicitada. Si la petición es referente a ejecución de un SWC (operación 5), la agrega a la cola de trabajo del Módulo de Ejecución.

B. Módulo de Ejecución

Revisa la cola de trabajo tan pronto esté libre, toma una representación de SWC (que está modelado como una Red de Petri Coloreada) y comienza la ejecución desde donde se tenga indicado en la definición del SWC para realizar la operación 5. Si durante esta ejecución se detecta que es posible ejecutar SW de forma paralela, se solicita a los demás SE registrados información de carga y se determina cuál o cuáles son los mejores para ejecutar los SW paralelos, luego se instancian las crónicas correspondientes, se envía a los otros SE la ejecución del SWC (indicando uno de los SW paralelos) y por último se continúa con la llamada a alguno de los SW paralelos que no fueron delegados a otros SE. Si no hay SE disponibles, o si el mismo SE puede cumplir con la invocación de todos los SW paralelos, no se instancia ninguna crónica, dado que el mismo SE puede manejar todos los eventos que se generen a partir de la invocación de los SW paralelos (e.g., culminación exitosa, ocurrencia de una falla).

Antes de comenzar y al terminar una llamada a un SW, se establecen puntos de chequeo (*checkpoints*) que contienen la información que va a ser enviada al SW (sus parámetros de entrada) y la información devuelta (sus parámetros de salida) respectivamente. Estos puntos de chequeo mantienen la información necesaria, en caso de ser requerido que se retome posteriormente la ejecución, debido a una falla en el SWC. Adicionalmente, de ser detectado un error por el Módulo de Crónicas, se detiene la ejecución, de tal forma que se pueda sincronizar la información disponible con los otros SE involucrados en la ejecución distribuida y acordar el esquema de recuperación conveniente.

C. Módulo de Tolerancia a Fallas

Toma la información de un SWC cuya ejecución haya fallado en un momento dado, evalúa las propiedades transaccionales de los servicios ya ejecutados, considera las restricciones de calidad de servicio estimadas, decide y ejecuta el mecanismo de recuperación que corresponda, ya sea recuperación hacia atrás o hacia adelante, creación de puntos de control o simplemente devuelve la solución parcial encontrada. Esto corresponde a la operación 6 y es una adaptación al algoritmo definido por Angarita et. al. en [5].

La decisión cuál estrategia de recuperación es la apropiada depende de diversos factores, como las condiciones de ejecución actuales, las propiedades transaccionales de los SW, la calidad de servicio demandada por el usuario, la existencia de réplicas, que deben ser evaluados por este módulo para tomar la decisión apropiada de manera dinámica.

D. Módulo de Crónicas

Es una envoltura a un reconocedor de crónicas y tiene un funcionamiento similar al propuesto en [23], es decir, se tiene un analizador de crónicas que se alimenta con la información de éxito o error local (proporcionado por el Módulo de Ejecución), y recibe información de los Módulos de Crónicas de otros SE. Se encarga de avisar al Módulo de Ejecución que ocurrió un error. En la Figura 5 se muestra un ejemplo de interacciones entre distintos SE asociados a una crónica en caso de una falla. El Módulo de Ejecución es el que detecta si una llamada no fue exitosa, dado que recibe un código de respuesta por la llamada HTTP[6], en caso de ser fallida le avisa al Módulo de Crónicas para que se disparen los eventos correspondientes en la crónica distribuida definida en ese momento.

Es posible utilizar las crónicas no sólo para la detección de fallas, sino para la sincronización de los servicios. Es decir, si se define una crónica distribuida para el caso en que todos los SW culminen exitosamente, al terminar todas las ejecuciones esta crónica se dispara y se intercambia toda la información correspondiente a la ejecución, esto corresponde a la operación 7. Posteriormente se decide cuál es el mejor SE para continuar la ejecución y se envía la petición correspondiente.

[6]Hypertext Transfer Protocol https://tools.ietf.org/html/rfc7230 tomado en Abril de 2015 protocolo de comunicación

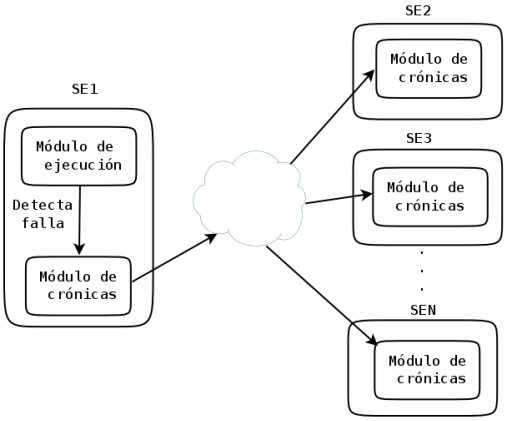

Figura 5: Interacción entre Módulos de Crónicas

IV. Trabajos Relacionados

Los SWC han sido fuente de investigación de diversas universidades en el mundo en los últimos años. Se han hecho esfuerzos en lenguajes para la especificación (Redes de Petri Coloreadas, por ejemplo), donde se establece el qué y cuáles SW conforman el SWC, se han hecho investigaciones en técnicas de análisis, verificación y evaluación para probar y reparar errores de diseño, se han investigado técnicas para selección y especificación automática de SWC y se han propuesto diversos motores de ejecución [1].

En los siguientes puntos se estarán detallando trabajos relacionados sobre los cuales se fundamenta CrESTa, exponiendo sobre las representaciones más utilizadas para SWC, trabajos sobre calidad de servicio, el uso de crónicas en el mundo académico actual y diversas propuestas de motores de ejecución en la actualidad.

A. Representación y Calidad de Servicio de SWC

La representación de SWC se refiere a las estructuras usadas para describir el flujo de ejecución, flujo de datos y restricciones de control entre los SW que lo componen. Existen diversas especificaciones en la actualidad, tales como WS-BPEL[7], WSCI[8], WS-CDL[9]. Sin embargo, en [1] se propone extender las Redes de Petri Coloreadas para incorporar descripciones de propiedades no funcionales de los SW y modelar la automatización de la composición de los servicios. Este enfoque es fundamental para los mecanismos de tolerancia a fallas y por eso es utilizado en CrESTa. En [20] se propone, no sólo el uso de Redes de Petri Coloreadas sino también de ASK-Computational Tree Logic para verificar la correctitud del modelo. Sin embargo, para nuestra propuesta se asume que la representación de la composición ya es correcta.

[7]Web Service Bussines Process Execution Language, 2007, https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm tomado en Diciembre de 2014. Es una especificación de OASIS que permite modelar SWC como orquestación enfocándose en procesos de negocio.

[8]Web Service Choreography Interface, 2002, http://www.w3.org/TR/wsci/ tomado en Diciembre de 2014

[9]Web Service Choreography Description Language, 2005, http://www.w3.org/TR/ws-cdl-10/ tomado en Diciembre de 2014

Si se utilizan mediciones respecto a cuánto tiempo lleva la ejecución de cada SW dentro de la composición y se compara con estimados iniciales, se puede determinar si la ejecución se encuentra dentro de los valores esperados en el marco de la calidad de servicio exigida por el usuario. Si ocurre una falla, estas mediciones permiten a los usuarios del servicio obtener respuestas en una cantidad de tiempo predeterminada, pues se puede elegir entre devolver la solución parcial obtenida, en caso que el mecanismo de tolerancia no cumpla con las especificaciones del usuario o elegir un mejor mecanismo de recuperación que permita mantener la ejecución en tiempos esperados. Considerar al menos el tiempo de ejecución como parámetro de calidad de servicio al momento de diseñar algún motor de ejecución para SWC es fundamental para el usuario. En [24] se propone una estimación de calidad de servicio en términos de parámetros no funcionales, aunque esta estimación se utiliza para elegir una composición en particular, la que mejor se adapte a los requisitos del usuario. En [25] además del tiempo de ejecución estimado de un SW y de un SWC, se considera el tiempo máximo de tolerancia para ejecución de un SWC, entre otros parámetros considerados en el modelo. En [26] se proponen estimaciones de calidad de servicio basadas en regiones.

Si la fase de composición de un SWC tomó en cuenta parámetros de calidad de servicios (requisitos no funcionales) exigidos por el usuario, es imprescindible, que el motor de ejecución del SWC sea garante de que realmente se cumplan dichos parámetros de calidad, aún en escenarios de fallas. Así que la propuesta de CrESTa, incluye la evaluación de parámetros de rendimiento para asegurar que en caso de fallas, la composición cumpla con los requisitos no funcionales del usuario.

B. Crónicas

El modelaje de sistemas reactivos (tales como sistemas operativos, protocolos, entre otros) ha llevado al estudio de la lógica temporal y de las crónicas, para tener un lenguaje mucho más expresivo y elegante para representar restricciones de tiempo y desencadenamiento de eventos. En este sentido, las crónicas permiten modelar un conjunto de eventos restringidos por relaciones temporales cualitativas o cuantitativas, como por ejemplo antes, 100 segundos antes, etc. [21]. En este campo, en [27] se plantea el uso de crónicas que sean equivalentes a Redes de Petri, de tal forma que se puedan inferir situaciones peligrosas en el funcionamiento de, por ejemplo, un aeropuerto. En [28], se extiende el concepto de crónicas para establecer el concepto de crónica distribuida, de tal forma que se puedan reconocer fallas en sistemas distribuidos.

En [27] se propone modelar Redes de Petri Coloreadas con crónicas para simular sistemas complejos y detectar actividades o analizar comportamientos.

En CrESTa, se usan las crónicas para detectar fallas y decidir el mejor esquema de recuperación, así como para modelar el comportamiento correcto de la ejecución de un SWC y lograr la sincronización requerida para respetar el flujo de ejecución

delineado en la estructura que representa el SWC (i.e., la Red de Petri Coloreada).

C. Motores de Ejecución y Frameworks de SWC

Se han hecho muchas propuestas sobre motores de ejecución para SWC que sean tolerantes a fallas. Algunos de ellos forman parte de frameworks que contienen mucho más que el motor de ejecución, pero cada uno presenta algún tipo de limitación que puede ser mejorado. En [4] se propone FACTS, que utiliza WS-BPEL[10] como representación de SWC, y utiliza excepciones para detectar fallas en caso de errores en la ejecución del SWC. El mecanismo propuesto por FACTS agrega una carga extra al motor de ejecución por la evaluación de excepciones, lo cual puede ser mejorado con el uso de crónicas. En [29] se utiliza BPMN[11] para especificar el SWC, y propone modificar su estructura dinámicamente cuando se detecten fallas, dependiendo de configuraciones hechas por el diseñador del servicio. Estos sistemas tienen el problema de utilizar como definición una representación basada en orquestación, que hace difícil la especificación de alguna composición que no requiera un componente central. En [31] se propone reemplazar automáticamente durante la ejecución de un SWC aquellos servicios que presenten fallas dado una lista de servicios de reemplazo proporcionada por el usuario antes de la ejeución.

Utilizando crónicas se puede tener una implementación poco intrusiva y más confiable en la ejecución, pues sólo se reportarán eventos de falla y de culminación exitosa y se basará en un programa de detección de crónicas que ya ha sido probado y utilizado con anterioridad.

En [30] se propone THROWS, que es un motor de ejecución distribuido basado en listas que identifican los SWC a ser ejecutados, usando WS-SAGAS[12] para representar el SWC y los estados de los SW. Ofrece recuperación hacia adelante en caso de fallas, se concentra en el uso de CEP (Current Execution Process) y CEL (Candidate Engine List), que contiene información del progreso de la ejecución y una lista con servicios candidatos para ejecutar el SW. Este planteamiento fue mejorado posteriormente e integrado en un framework en [19], con el nombre FENECIA. Este trabajo tiene desventajas al momento de modelar el SWC. Primero no tiene la profundidad matemática que tienen las Redes de Petri Coloreadas, en segundo lugar, está aglutinando la representación del SWC junto con la información de la ejecución del mismo.

En el campo de la web semántica, en [33] se propone el uso de una extensión de WS-BPEL[13] llamado SPL[14], que le agrega una descripción semántica utilizando RDF4S [34] a el SWC. Utiliza esta definición ontológica como un método para manejo de excepciones que integra SW, agentes y ontologías para definir procesos de negocio y darle contenido semántico. Adicionalmente, en [35] se utiliza OWL-S[15] para descomponer peticiones de usuarios en SW y representarlos. En [36] se propone la generación automática de una Red de Petri Coloreada que representen la composición de un servicio web a través de un mediador que toma una solicitud y utilizando información semántica genera automáticamente la composición. Esto puede ser utilizado posteriormente para extender el trabajo realizado en este proyecto, de tal forma que se pueda tener además del motor, un sistema con definición de la composición.

Se han propuesto diversos mecanismos para manejar problemas de sincronización que se encuentran en la ejecución de SWC distribuidos. En [37] se detecta inconsistencias causadas por ciclos de dependencia y compensaciones, cambiando las dependencias de acuerdo a un conjunto de reglas que aseguran la consistencia de la ejecución. En [38] se propone un algoritmo de control de concurrencia basado en la dependencia de entrada y salida de cada SW dentro de la composición.

En [39] se propone FaCETa, donde se plantea un mecanismo para hacer recuperación hacia atrás (compensación) utilizando una Red de Petri Coloreada modificada, en particular, se invierten las aristas y se colocan SW de compensación que permitirán regresar el estado general de la composición al estado antes de la ejecución del SW original. En [5] y [25] se propone un motor de ejecución distribuido de SWC tolerante a fallas, que considera parámetros de calidad de servicio que toma un grafo que representa a un SWC como entrada, pero que considera un ambiente altamente confiable de ejecución como requisito para el sistema, se basa en el uso de MPI[16] aunque se pueden utilizar tecnogías web para convertir al motor en un servicio para masificar su uso y es posible utilizar un mecanismo confiable poco intrusivo para la detección de fallas (que serían las crónicas).

Vizcarrondo et. al. en [23] proponen un middleware llamado ARMISCOM que utiliza crónicas distribuidas para detectar fallas en composición de servicios. Sin embargo, no consideran las propiedades transaccionales de los SW, no toma en cuenta la calidad de servicio para decidir dináminamente la mejor estrategia de recuperación y es un middleware, no un motor de ejecución compuesto por múltiples instancias distribuidas, como lo que se está proponiendo en este trabajo.

En resúmen, ninguno de los trabajos mencionados con anterioridad utilizan el modelo de crónicas junto con las Redes de Petri Coloreadas para la detección de fallas, se espera

[10]Web Service Bussines Process Execution Language, 2007, https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm tomado en Diciembre de 2014. Es una especificación de OASIS que permite modelar SWC como orquestación enfocándose en procesos de negocio.

[11]Business Process Model and Notation, 2011, http://www.omg.org/spec/BPMN/2.0/ tomado en Abril de 2015. Es una especificación que permite modelar procesos de negocios en base a eventos, actividades y flujos, similar a un diagrama de flujos pero con artefactos adicionales.

[12]Modelo Transaccional para especificación y ejecución de composición de SW, tomado de [32] define un SWC como un conjunto de transacciones que se pueden definir recursivamente para representar la composición

[13]Web Service Bussines Process Execution Language, 2007, https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm tomado en Diciembre de 2014. Es una especificación de OASIS que permite modelar SWC como orquestación enfocándose en procesos de negocio.

[14]Semantic Programming Language

[15]Semantic Markup for Web Services, 2004, http://www.w3.org/Submission/OWL-S/ tomado en Diciembre de 2014

[16]Message Pasing Interface, 2012, http://www.mpi-forum.org/docs/docs.html tomado en Diciembre de 2014

que el uso de crónicas permita una implementación segura, eficiente, confiable y poco intrusiva de un motor de ejecución, así como las Redes de Petri Coloreadas permitirán representar propiedades transaccionales de SW para el manejo de tolerancia de fallas de una forma genérica y precisa sin la necesidad de otras estructuras (como el CEP y el CEL de THROWS) para expresar el flujo de un SWC.

V. Caso de Estudio

En esta sección se ilustrará el funcionamiento y comportamiento de CrESTa, a través de un ejemplo.
Supongamos que una persona desea realizar un viaje de visita a alguna ciudad y quiere que automáticamente se cree una sugerencia de viaje incluyendo costos, fechas y sitios de estadía, considerando cierta cantidad de visitas a lugares de la ciudad. Supongamos que las restricciones del usuario son:

- Ciudad de destino
- Reputación del sitio de estadía
- Fecha aproximada de viaje y duración del mismo
- Presupuesto
- Lugares que quiere visitar
- Preferencia de transporte

Si en la ciudad hay servicios web que dan la disponibilidad de hoteles, transporte y ventas de boletos a sitios turísticos altamente deseados, es posible establecer una composición de servicios web que satisfagan las restricciones del usuario. Supongamos que los SW necesarios para cumplir la petición son los descritos en la Tabla III.

Tabla III: Servicios del Ejemplo Motivador

SW	Entrada	Salida	Prop. Trans.
hospedaje	fecha de reserva, reputación, presupuesto	fecha	cr
visita1	fecha	fecha de visita1	cr
visita2	fecha	fecha de visita2	cr
visita nocturno	fecha	fecha de visita nocturno	cr
visita3	fecha	fecha de visita3	cr
transporte	fecha de visita1, fecha de visita2 fecha de visita nocturno, fecha de visita3	boletos	cr

El servicio de hospedaje recibe como entrada una reputación que define la calidad esperada del hotel (ejemplo: 5 estrellas, 4 estrellas, 3 estrellas), el presupuesto disponible por el usuario y una fecha tentativa de reserva, luego chequea la disponibilidad en los hoteles de la ciudad y finalmente emite una fecha posible de reserva. Los servicios de visitas a sitios turísticos reciben una fecha tentativa de estadía del usuario, revisan su disponibilidad y emiten una fecha que se apegue a los parámetros recibidos. Finalmente el servicio de transporte recibe todas las fechas de visitas y determina los horarios de transporte y boletos para desplazarse entre las visitas. Como solamente se está haciendo una consulta, todos los servicios son **compensables** y es posible reintentar la solicitud, por eso todos tienen la propiedad transaccional **compensable reintentable** (cr). En la Figura 6 se muestra una Red de Petri

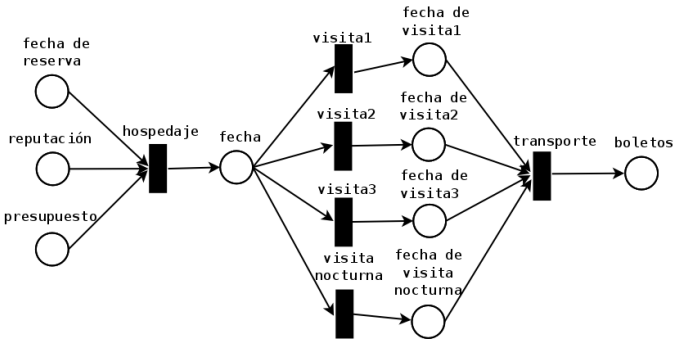

Figura 6: SWC del Caso de Estudio

Coloreada con una posible representación del SWC generado a partir de los servicios y la solicitud del usuario. La generación de esta composición no está considerada en este trabajo, se asume que ya hubo alguna herramienta que se encargó de procesar la solicitud y los requerimientos para generar correctamente el SWC.

El siguiente paso consiste en ejecutar la composición y devolver los resultados obtenidos para ser mostrados al usuario. Supongamos que hay 4 Servicios de Ejecución (SE) de CrESTa activos: SE_1, SE_2, SE_3 y SE_4 y que la ejecución se lleva a cabo de la siguiente manera:

1) SE_1 recibe la representación del SWC, la definición de crónicas distribuidas y los requisitos no funcionales. Esta información es procesada por el Módulo Clasificador, quien determina que es una solicitud de ejecución, por lo que la introduce en la cola de trabajos del Módulo de Ejecución.
2) Cuando el Módulo de Ejecución esté disponible, se comienza la ejecución, llamando al servicio de hospedaje.
3) Con el resultado del SW de hospedaje, se envía a SE_2, SE_3 y SE_4 información para que se realice la llamada de los servicios $visita1$, $visita2$ y $visita\ nocturna$ respectivamente en paralelo, mientras que SE_1 se encarga de $visita3$. Se instancia la crónica distribuida asociada a esta ejecución paralela.
4) Al terminar la ejecución de los servicios mencionados, se concentran las respuestas en uno de los nodos, el último que terminó (supongamos que el SE_1), y luego se llama al servicio de transporte.
5) Se envían las respuestas al usuario.

Durante el proceso descrito, en los puntos 2, 3 o 4 pudo ocurrir algún error, ya sea por agotarse el tiempo de espera para respuestas del servicio (quizás el servicio esté muy ocupado) o pudo haber algún error en la conexión. Para la detección de estos errores se utilizan crónicas. A continuación se presentan crónicas para detectar errores en esos puntos.

Para el punto 2 se define la crónica $C_{hospedaje}$:
Eventos: $inicio$, $ejecucion$, $error$.
Crónica:

$C_{hospedaje}$ {
  Eventos {
    evento ($inicio$, T_1),

```
    evento(ejecucion, T2),
    evento(error, T3),
  }
  Restricciones{
    T2 − T1 > 0,
    T3 − T2 > 0
  }
  acciones{
    errorHospedaje()
  }
}
```

Si ocurre alguno de los errores mencionados con anterioridad en SE_1 el Módulo de Ejecución lo informa al Módulo de Crónicas y éste llama a la función $errorHospedaje()$ inmediatamente. Esta función se encuentra dentro del Módulo de Tolerancia a Fallas, donde se evalúan los requisitos no funcionales definidos y se decide si realizar recuperación hacia atrás o hacia adelante.

Para detectar el error en el punto 3 se deben instanciar 4 crónicas distribuidas, una por cada posible error en los SW $visita1$, $visita2$, $visita3$ y $visita\ nocturna$. Como hay 4 SW ejecutándose en 4 SE entonces cada crónica debe estar compuesta por 4 subcrónicas. Ejemplificaremos la definición de la crónica distribuida para detección de error en $visita3$, que llamaremos $C_{visita3}$. Las 4 subcrónicas que la componen las llamaremos: $SC_{visita1}$, $SC_{visita2}$, $SC_{visita3}$ y SC_{visita_noc}. Definamos el conjunto de eventos $E_{visita3}$:

$$E_{visita3} = \{inicio, ejecucion, error\}$$

El conjunto de eventos enlazadores $EE_{visita3}$ viene dado por:

$$EE_{visita3} = \{EESC^{+}_{E12}, EESC^{-}_{E2}, EESC^{+}_{E13}, EESC^{-}_{E3} \\ EESC^{+}_{E14}, EESC^{-}_{E4}\}$$

Los eventos $EESC^{+}_{E12}$, $EESC^{+}_{E13}$, $EESC^{+}_{E14}$ se disparan desde el SE_1 y son los encargados de avisar a SE_2, SE_3 y SE_4 que hubo un error en $visita3$. En la Figura 7 se puede observar la relación entre los eventos en cada uno de los servicios.

La crónica distribuida se puede ver de la siguiente forma:

```
C_visita3{
  Eventos{
    evento(SE1.inicio, T12),
    evento(SE2.inicio, T21),
    evento(SE3.inicio, T31),
    evento(SE4.inicio, T41),
    evento(SE1.ejecucion, T12),
    evento(SE2.ejecucion, T22),
    evento(SE3.ejecucion, T32),
    evento(SE4.ejecucion, T42),
    evento(SE1.error, T13),
    evento(SE1.EESC+E12, T14),
    evento(SE2.EESC−E2, T23),
    evento(SE1.EESC+E13, T15),
    evento(SE2.EESC−E3, T33),
```

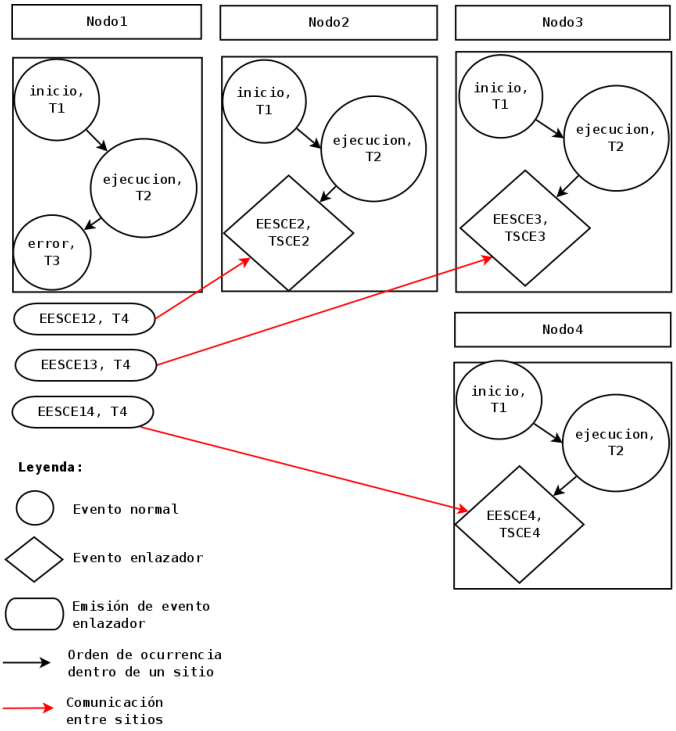

Figura 7: Crónica Distribuida para Detectar un Error en visita3

```
    evento(SE1.EESC+E14, T16),
    evento(SE2.EESC−E4, T43),
  }
  Restricciones{
    T12 − T11 > 0,
    T13 − T12 > 0,
    T14 − T13 > 0,
    T15 − T14 > 0,
    T16 − T15 > 0,
    T22 − T21 > 0,
    T23 − T22 > 0,
    T32 − T31 > 0,
    T33 − T32 > 0,
    T42 − T41 > 0,
    T43 − T42 > 0
  }
  acciones{
    falloDeEjecucion()
  }
}
```

Esa secuencia de eventos vista en la crónica distribuida quiere decir que una vez iniciados los procesos de ejecución en cada SE, si ocurre el error en $SE1$ que está ejecutando $visita3$ se avisa a los demás SE. Una vez detectada la falla utilizando las propiedades transaccionales y considerando parámetros de calidad de servicio en cuanto a ejecución total del SWC y la ejecución estimada de cada SW, se puede establecer qué mecanismo de recuperación (hacia atrás, hacia adelante o puntos de control) se debe activar para cumplir con los requerimientos definidos por el usuario.

Para el punto 4 se define la crónica $C_{transporte}$:
Eventos: $inicio$, $ejecucion$, $error$.
Crónica:

$C_{transporte}$ {
  Eventos {
    evento($inicio$, T_1),
    evento($ejecucion$, T_2),
    evento($error$, T_3),
  }
  Restricciones {
    $T_2 - T_1 > 0$,
    $T_3 - T_2 > 0$
  }
  acciones {
    $errorTransporte()$
  }
}

Y se sigue el mismo procedimiento que para el punto 2.

Note que la definición de las crónicas es independiente de los mecanismos de recuperación que se puedan considerar y de los enfoques de evaluación de condiciones de ejecución y de parámetros de calidad para decidir dinámicamente cuál es el mejor esquema de recuparación (hacia adelante, hacia atrás, puntos de control). De esta manera, se pueden incorporar distintos mecanismos, enfoques y estrategias que se activen de acuerdo a las condiciones evaluadas para ejecutar algoritmos de tolerancia a fallas o implementar sincronizaciones requeridas en una ejecución exitosa del SWC.

VI. Conclusiones y Trabajo Futuro

Con CrESTa se busca tener un sistema robusto, tolerante a fallas y con una alta accesibilidad utilizando una arquitectura orientada a servicio para la ejecución de Servicios Web Compuestos. El uso de crónicas ayuda a este fin con la detección inmediata de fallas y se puede combinar con la evaluación de parámetros de calidad, como el tiempo de ejecución, y la consideración de las propiedades transaccionales de los servicios web para decidir dinámicamente el mejor enfoque de recuperación de fallas, sin que el usuario detecte siquiera que se presentó algún inconveniente.

Este sistema por ser distribuido, presenta un problema que un sistema centralizado no tiene y es la confidencialidad en el intercambio de información entre los SE, pues pueden estar sobre una red insegura y posiblemente se maneje información sensible al momento de contactar los SW de la composición. Actualmente se está trabajando en un esquema de seguridad que permita garantizar la confidencialidad en caso de ser solicitado y adicionalmente la autenticidad de la respuesta, para asegurar que la respuesta de CrESTa no haya sido alterada.

También se trabaja en automatizar el proceso de definición de crónicas, de tal forma que para una ejecución solamente se de la Red de Petri Coloreada y los requisitos no funcionales.

Adicionalmente se someterá el sistema a diversas pruebas de rendimiento para asegurar que el esquema de crónicas efectivamente permite mantener los tiempos de ejecución razonables en situaciones de fallas.

Referencias

[1] Y. Cardinale, J. El Haddad, M. Manouvrier y M. Rukoz, *Web Service Composition Based on Petri Nets: Review and Contribution*, Resource Discovery, serie Lecture Notes in Computer Science. Springer Berlin Heidelberg, vol. 8194, pp. 83–122, Enero 2013.

[2] Q. Yu, X. Liu, A. Bouguettaya y B. Medjahed, *Deploying and Managing Web Services: Issues, Solutions, and Directions*, The VLDB Journal, vol. 17, no. 3, pp. 537–572, Mayo 2008.

[3] Y. Cardinale, J. El Haddad, M. Manouvrier y M. Rukoz, *CPN-TWS: a Coloured Petri-net Approach for Transactional-QoS Driven Web Service Composition*, International Journal of Web and Grid Services, vol. 7, no. 1, pp. 91–115, Febrero 2011.

[4] A. Liu, Q. Li, L. Huang y 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, Abril 2010.

[5] R. Angarita, Y. Cardinale y M. Rukoz, *Reliable Composite Web Services Execution: Towards a Dynamic Recovery Decision*, Electronic Notes in Theoretical Computer Science, vol. 302, pp. 5-28, Febrero 2014.

[6] R. Angarita, Y. Cardinale y M. Rukoz, *FACETA*: Checkpointing for Transactional Composite Web Service Execution Based on Petri-Nets*, Procedia Computer Science, vol. 10, pp. 874–879, 2012.

[7] E. Sindrilaru, A. Costan y V. Cristea, *Fault Tolerance and Recovery in Grid Workflow Management Systems*, in proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 475–480, Krakow, Poland, Febrero 2010.

[8] F. Lelli, G. Maron y S. Orlando, *Towards Response Time Estimation in Web Services*, in proceedings of the IEEE International Conference on Web Services (ICWS 2007), pp. 1138–1139, Salt Lake City, USA, Julio 2007.

[9] M. Dharmendra y R. Mahajan, *A Novel Approach for Performance Estimation of Soap-based Web Services*, International Journal of Emerging Technology and Advanced Engineering, vol. 2, no. 2, pp. 1138–1139, Febrero 2012.

[10] X. Zhang, Y. Yin y B. Zhang, *A Service Mining Approach for Time Estimation of Composite Web Services*, Applied Informatics and Communication, serie Communications in Computer and Information Science, vol. 228, pp. 486–492, Agosto 2011.

[11] C. Dousson, *Extending and Unifying Chronicle Representation with Event Counters*, in proceedings of the 15th Eureopean Conference on Artificial Intelligence, pp. 257–261, Lyon, France, Julio 2002.

[12] J. Vizcarrondo, J. Aguilar, E. Exposito y 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, Enero 2015.

[13] D. Sherman, *BPEL: Make Your Services Flow. Composing Web Services into Business Flow*, Journal in Web Services, vol. 3, no. 7, pp. 16–21, 2003.

[14] OASIS, *Web Services Businnes Process Execution Language (WS-BPEL), Version 2.0* http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Abril 2007.

[15] O. Bushehrian, S. Zare y N. Keihani, *A Workflow-Based Failure Recovery in Web Services Composition*, Journal of Software Engineering and Applications, vol. 5, no. 4, pp. 89–95, Febrero 2012.

[16] J. El Haddad, M. Manouvrier y M. Rukoz, *TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition*, IEEE Transactions on Services Computing, vol. 3, no. 1, Abril 2010.

[17] J. Behl, T. Distler, F. Heisig, R. Kapitza y M. Schunter, *Providing Fault-tolerant Execution of Web-service based Workflows within Clouds*, in proceedings of the 2nd International Workshop on Cloud Computing Platforms (CloudCP), Bern, Switzerland, Abril 2012.

[18] W. Zhou y L. Wang, *A Byzantine Fault Tolerant Protocol for Composite Web Services*, in proceedings of the International Conference on Computational Intelligence and Software Engineering (CiSE), pp. 1–4, Wuhan, China, Diciembre 2010.

[19] N. Lakhal, T. Kobayashi y H. Yokota, *FENECIA: Failure Endurable Nested-Transaction based Execution of Composite Web Services with Incorporated State Analysis*, The VLDB Journal, vol. 18, no. 1, pp. 1–56, Enero 2009.

[20] M. Souilah, B. Bérard y M. Boufaida, *Analyzing Behavioral Compatibility for Web Service Choreography Using Colored Petri Nets and ASK-CTL*, in proceedings of the Sixth International Conferences on Advanced Service Computing, pp. 32–39, Venice, Italy, Mayo 2014.

[21] J. Aguilar, *Temporal Logic from the Chronicles Paradigm: Learning and Reasoning Problems, and its Applications in Distributed Systems*, primera edición, LAP Lambert Academic Publishing, 2011.

[22] X. Le Guillou, M. Cordier, S. Robin, L. Rozé y otros, *Chronicles for On-line Diagnosis of Distributed Systems*, in proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008), vol. 8, pp. 194–198, Patras, Greece, Mayo 2008.

[23] J. Vizcarrondo, J. Aguilar, E. Exposito y A. Subias, *ARMISCOM: Autonomic Reflective Middleware for Management Service Composition*, in proceedings of the Global Information Infrastructure and Networking Symposium (GIIS 2012), pp. 1–8, Choroni, Venezuela, Diciembre 2012.

[24] E. Blanco, Y. Cardinale, M. Vidal, *Aggregating Functional and Non-Functional Properties to Identify Service Compositions*, Engineering Reliable Service Oriented Architecture: Managing Complexity and Service Level Agreements, pp. 1–36, Marzo 2011.

[25] R. Angarita, Y. Cardinale, M. Rukoz, *Dynamic Recovery Decision During Composite Web Services Execution*, in proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems (MEDES 2013), pp. 187–194, Luxembourg, Luxembourg, Octubre 2013.

[26] W. Lo, J. Yin, S. Deng, Y. Li y Z. Wu, *Collaborative Web Service QoS Prediction with Location-based Regularization*, in proceedings of the IEEE 19th International Conference on Web Services (ICWS 2012), pp. 464–471, Honolulu, USA, Junio 2012.

[27] C. Choppy, O. Bertrand y P. Carle, *Coloured Petri Nets for Chronicle Recognition*, Reliable Software Technologies–Ada-Europe 2009, pp. 266–281, Mayo 2009.

[28] J. Vizcarrondo, J. Aguilar, A. Subias y E. Exposito, *Distributed Chronicles for Recognition of Failures in Web Services Composition*, in proceedings of the XXXIX Latin American Computing Conference (CLEI 2013), pp. 1–10, Naiguata, Venezuela, Octubre 2013.

[29] K. Fang, J. Li, H. Sun, Y. Zhao y H. Zeng, *Strategy-based Two-Level Fault Handling Mechanism for Composite Service*, in proceedings of the IEEE 2nd International Conference on Software Engineering and Service Science (ICSESS 2011), pp. 494–499, Beijing, China, Julio 2011.

[30] N. Lakhal, T. Kobayashi y H. Yokota, *Throws: An Architecture for Highly Available Distributed Execution of Web Services Compositions*, in proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for e-Commerce and e-Government Applications, pp. 103–110, Boston, USA, Marzo 2004.

[31] S. Gupta y P. Bhanodia, *A Fault Tolerant Mechanism for Composition of Web Services Using Subset Replacement*, International Journal of Advanced Research in Computer and Communication Engineering, vol. 2, no. 8, pp. 3080–3085, Agosto 2013.

[32] N. Lakhal, Y. Kobayashi y H. Takashi, *WS-SAGAS: Transaction Model for Reliable Web Services Composition Specification and Execution*, DBSJ letters, vol. 12, no. 2, pp. 17–20, 2001.

[33] K. Zhao, L. Zhang y S. Ying, *Ontology-Based Exception Handling for Semantic Business Process Execution*, Journal of Software, vol. 7, no. 8, pp. 1791–1798, 2012.

[34] D. Xie, S. Ying, J. Yao y B. Xiao, *A Resource Description Framework for Service*, in proceedings of the 2007 International Conference on Wireless Communications, Networking and Mobile Computing (WiCom 2007), pp. 3351–3354, Shangai, China, Septiembre 2007.

[35] N. Sasikaladevi y L. Arockiam, *Extended WS-FIT Model to Enhance the Fault Tolerance in the Dynamic Composite Web Service*, in proceedings of the 3rd International Conference on Electronics Computer Technology (ICECT 2011), vol. 5, pp. 21–25, Kanyakumari, India, Abril 2011.

[36] A. Marakhimov, J. Yim y J. Joo, *Petri Net Based Semantic Web Service Composition*, International Journal of Multimedia and Ubiquitous Engineering (IJMUE), vol. 9, no. 2, pp. 281–290, 2014.

[37] Z. Zhao, J. Wei, L. Lin y X. Ding, *A Concurrency Control Mechanism for Composite Service Supporting User-Defined Eelaxed Atomicity*, in proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008), pp. 275–278, Turku, Finland, Agosto 2008.

[38] S. Sundar y R. Kanchana, *Handling Concurrency Control Problem in Web Service Compositions*, in proceedings of the Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT 2013), pp. 1–5, Tiruchengode, India, Julio 2013.

[39] R. Angarita, Y. Cardinale y M. Rukoz, *Faceta: Backward and Forward Recovery for Execution of Transactional Composite WS*, in proceedings of the 5th International Workshop on Resource Discovery (RED 2012) at the 9th Extended Semantic Web Conference (ESWC 2012), pp. 89–103, Heraklion, Greece, Mayo 2012.