ISSN 1870-4069

Hacia transacciones distribuidas coordinadas por agentes para la arquitectura de microservicios

Xavier Limón[1], Alejandro Guerra-Hernández[2], Angel J. Sánchez-García[1], Juan Carlos Peréz Arriaga[1], Juan Luis López Herrera[1]

[1] Universidad Veracruzana, Facultad de Estadística e Informática, Xalapa, México
[2] Universidad Veracruzana, Centro de Investigación en Inteligencia Artificial, Xalapa, México
{hlimon, aguerra, angesanchez, juaperez}@uv.mx, jlopezh73@gmail.com

Resumen. La arquitectura de microservicios ha surgido como respuesta al rápido cambio tecnológico, la preocupación de mayor extensibilidad, escalabilidad y a la necesidad de ciclos de entrega de software cada vez más cortos. En esta arquitectura, el sistema, usualmente distribuido, es descompuesto en una serie de servicios altamente cohesivos e independientes. Cada microservicio puede contar con tecnologías de implementación y persistencia de datos diferente, obteniéndose así sistemas distribuidos heterogéneos, donde cada microservicio puede ser desarrollado y mantenido por equipos de trabajo diferentes. Dada su heterogeneidad y naturaleza distribuida, uno de los retos latentes en esta arquitectura es el manejo adecuado de transacciones distribuidas que se expanden por varios microservicios. En este trabajo se propone una aproximación basada en agentes para la arquitectura de microservicios, dicha aproximación puede verse como una capa autónoma que coordina las transacciones distribuidas del sistema. A diferencia de aproximaciones existentes, la propuesta es de alto nivel, lo que facilita su futura implementación en una tecnología de agentes específica, e integra aspectos de confiabilidad y robustez basados en coordinación entre agentes.

Palabras clave: sistemas multi-agente, microservicios, arquitecturas de software, transacciones distribuidas.

Toward Distributed Transactions Coordinated by Agents for the Microservices Architecture

Abstract. The microservices architecture has arisen as a response to fast technological changes, more extensibility and scalability concerns, and the necessity of shorter software delivery cycles. In this architecture, the system, usually distributed, is decomposed into a series of highly cohesive and independent services. Each microservice can have different implementation and data persistence technologies, thus obtaining heterogeneous distributed systems, where each microservice can be developed and maintained by different work teams. Given its heterogeneity and distributed nature, one of the latent challenges in this architecture is

the proper management of distributed transactions that are expanded by several microservices. In this paper we propose an agent-based approach for the microservices architecture, this approach can be seen as an autonomous layer that coordinates the distributed transactions of the system. Unlike existing approaches, the proposal is of a high level, which facilitates its future implementation in a specific agent technology, and integrates aspects of reliability and robustness based on coordination between agents.

Keywords: multi-agent systems, microservices, software architectures, distributed transactions.

1. Introducción

Los microservicios son un estilo arquitectónico de software inspirado en el cómputo orientado a servicios que en tiempos recientes ha gozado de gran popularidad [4]. La idea general de esta arquitectura es descomponer el sistema en servicios independientes, usualmente distribuidos, que hacen la menor cantidad posible de tareas relacionadas, esto es, servicios pequeños con alta cohesión que eventualmente colaboran entre si. Al ser independientes, cada microservicio cuenta con su propio ciclo de vida, lo que permite que sean desarrollados por diversos grupos de trabajo en momentos diferentes y utilizando diferentes tecnologías heterogéneas [11]. Todas las características antes mencionadas facilitan la integración e incluso experimentación de nuevas tecnologías, así como la extensibilidad y escalabilidad misma del sistema.

Los microservicios nacen en un momento donde existen diversas formas para lidiar con problemas de interoperabilidad derivados de componentes heterogéneos, siendo el ejemplo más relevante los servicios web RESTful [6], los cuales promueven el uso de formatos neutrales estándar, principalmente JSON, para el intercambio de datos. La combinación entre servicios web RESTful y microservicios es una de las más comunes en la actualidad [11] ya que brinda facilidad de distribución, flexibilidad de despliegue, tolerancia a fallos y velocidad de entrega.

Considerando su naturaleza independiente, abierta y heterogénea, los micro servicios suelen seguir un patrón de diseño conocido como base de datos por servicio [14], el cual establece que cada microservicio cuente con su propia base de datos, lo que puede incluir diversas tecnologías de persistencia SQL o no SQL.

Dada la potencial diversidad de tecnologías de persistencia de datos en microservicios, un problema abierto es el manejo de transacciones distribuidas que involucran a varios microservicios. Una transacción básicamente es un conjunto de operaciones que se deben de ejecutar de forma atómica, esto es, o se ejecutan todas o no se ejecuta ninguna, de esta forma se asegura la integridad de datos si algo sale mal en cualquier paso. La coordinación necesaria para realizar cada paso de la transacción, y potencialmente limpiar la transacción en caso de error, mientras al mismo tiempo se trata de no degradar significativamente el

rendimiento y escalabilidad del sistema, es un problema no trivial que requiere por si mismo de tecnologías y patrones de diseño propios.

Los sistemas Multi-Agente, MAS por su siglas en inglés, facilitan la realización de tareas coordinadas complejas en sistemas distribuidos, ya que son, por definición, sistemas distribuidos y cuentan con el nivel de abstracción adecuado para facilitar la comunicación entre componentes, así como la flexibilidad, modularidad, robustez, autonomía y escalabilidad necesarios [8].

Dadas las características mencionadas, es natural pensar en la posibilidad de unir los sistemas Multi-Agente con la arquitectura de microservicios. Está unión, aunque en la actualidad no muy común, puede brindar posibilidades interesantes como la que se presenta en este trabajo, donde se propone una aproximación basada en sistemas Multi-Agente para resolver el problema de transacciones distribuidas en la arquitectura de microservicios. Los objetivos principales de la aproximación propuesta son los siguientes:

1. Escalabilidad. Apropiada para sistemas de gran escala, con diversos nodos computacionales y una gran cantidad de microservicios.
2. Confiabilidad. Es posible adoptar estrategias para mantener la integridad de datos incluso en situaciones de fallo.
3. Heterogeneidad. Abierta a cualquier tipo de tecnología de implementación y persistencia de datos en los microservicios.
4. Alto nivel. El modelo de la propuesta, su flujo de trabajo, sus conceptos y estrategias asociadas pueden ser entendidas en términos de alto nivel gracias al uso de agentes. Este aspecto facilita el uso y configuración por parte de los desarrolladores de microservicios, así como la implementación de la propuesta en una tecnología de agentes específica.

El resto de este trabajo está organizado como sigue. En la sección 2 se amplía el contexto del problema de transacciones distribuidas en la arquitectura de microservicios introducido anteriormente, incluyendo aproximaciones existentes para resolverlo parcialmente. En la sección 3 se detalla la propuesta, describiendo la idea general, un modelo para representar transacciones, un formato de configuración para describir el flujo de una transacción y finalmente un conjunto de estrategias para el manejo de errores. Finalmente, en la sección 4 se presentan diversas conclusiones del trabajo acompañadas del trabajo futuro por realizar.

2. Contexto

Las transacciones distribuidas no son un tema nuevo en ingeniería de software, son un problema inherente de los sistemas distribuidos y existen varios esfuerzos propuestos en la literatura para tratarlas, tales como el coordinador de transacciones distribuidas de Microsoft (MSDTC) de la plataforma .NET [7], así como diversas implementaciones de manejadores de bases de datos como Oracle y MySQL que utilizan un protocolo de manejo de transacciones conocido como commit de dos fases (two-phase commit) [9].

Sin embargo, las soluciones tradicionales, como las antes mencionadas, suelen adolecer de los siguientes problemas en el contexto de microservicios [11]:

1. Homogeneidad. Cada parte del sistema debe ser tecnológicamente compatible, ya sea en tecnología de persistencia o implementación, para que el mecanismo funcione.
2. Baja escalabilidad. Muchas soluciones trabajan de forma síncrona, esto es, los componentes pueden detener su trabajo o bloquear tablas de la base de datos a la espera de la respuesta de otros componentes.

Otra posibilidad, en el contexto de microservicios, es que los microservicios mismos se coordinen de forma síncrona para realizar las transacciones. Esta aproximación, aunque directa y heterogénea, puede adolecer del problema de escalabilidad antes mencionados, siendo difícil de implementar de forma adecuada y confiable, sobre todo en el aspecto de manejo de errores.

A raíz de los problemas mencionados, especialmente el referente a escalabilidad, en tiempos recientes se ha adoptado un modelo de persistencia conocido como consistencia eventual [1], dicho modelo no requiere de bloqueos y permite el trabajo concurrente sobre datos de transacciones incompletas y datos parcialmente actualizados. El modelo establece que de no haber más actualizaciones futuras, la base de datos eventualmente se vuelve consistente. La consistencia eventual permite un modelo de trabajo asíncrono, esto es, no se requiere de coordinadores en la transacción ni de esperar a que otro componente termine su trabajo, cada componente puede trabajar de forma independiente.

La consistencia eventual puede considerarse de consistencia débil y conlleva ciertos riesgos. Para que pueda ser implementado de forma adecuada, el desarrollador debe tener en cuenta las situaciones de concurrencia que pudieran generar inconsistencias, complementándolas con código de dominio [11]. Por ejemplo, en una aplicación para realizar compras, en vez de directamente substraer dinero de la cuenta del cliente, primero se reserva el crédito necesario para realizar una compra y si la transacción sale bien, se substrae el dinero de la cuenta como paso final.

Dado el panorama antes planteado, surge la necesidad de contar con formas concretas para lidiar con transacciones distribuidas en el contexto de microservicios. En la sección siguiente, se presenta una serie de patrones de diseño relacionados que intentan resolver el problema planteado.

2.1. Patrones de diseño relacionados

Posiblemente el patrón de diseño más importante y conocido para el manejo de transacciones distribuidas con consistencia eventual sea el patrón Saga [5]. Una saga es una secuencia de transacciones locales donde cada transacción actualiza datos de un solo microservicio. La primera transacción es iniciada por una petición externa, luego cada paso subsecuente es disparado tras haber completado el paso previo, si algo sale mal en cualquier paso, se ejecuta una serie de acciones de compensación que intentan limpiar la transacción. Este patrón puede ser implementado de dos formas generales:

- Orquestada. Existe un coordinador central de transacciones, quien orquesta cada paso de la transacción y coordina las operaciones de compensación en

caso de fallos. Cada vez que un paso se completa, el microservicio le avisa al coordinador que ha terminado. Esta orquestación permite que los microservicios se mantengan desacoplados. La orquestación puede ser síncrona o asíncrona.

- Coreografiada. Basada en eventos asíncronos. Cada microservicio produce un evento cada vez que completa un paso de la transacción o un error ocurre, otros microservicios escuchan dicho evento y continúan con el proceso acorde.

La aproximación orquestada tradicional tiene la desventaja de tener un sólo punto central de fallo, además de que conlleva a la centralización de tráfico. Así mismo, el coordinador de transacciones puede ser por si solo un módulo muy complejo de crear y mantener. Otra desventaja de esta aproximación es que, para ser confiable, requiere de atomicidad entre la finalización de una operación local y la notificación al coordinador, de no haber dicha atomicidad se puede caer en casos en los que la operación local finalizó pero por algún error no se le pudo notificar al coordinador este hecho.

Al igual que la aproximación orquestada, la aproximación coreografiada requiere, para ser confiable, de atomicidad entre la finalización de la operación local y la publicación del evento asociado. Una aproximación posible es un patrón de diseño llamado Event Sourcing [14]. Event Sourcing establece la persistencia de entidades de dominio, tales como ordenes y clientes, como una secuencia de eventos de cambio de estado. Cada vez que el estado de una entidad de dominio cambia, un nuevo evento se agrega a su cola de eventos. Si se quiere recuperar el estado actual de la entidad de dominio, simplemente se ejecutan sus eventos en orden. Dado que almacenar un evento es una sola operación, esta aproximación es inherentemente atómica.

En Event Sourcing, las aplicaciones persisten eventos en un almacén de eventos, el cuál cuenta con los mecanismos necesarios para agregar y obtener los eventos de una entidad. Cuando se une con el patrón Saga, el almacén de eventos también sirve como intermediario para la coordinación coreografiada de transacciones, permitiendo que microservicios se suscriban a eventos que pueden almacenar otros microservicios.

Al utilizar Saga y Event Sourcing en conjunto, surge el problema de no tener directamente disponible el estado actual (o incluso uno próximo al actual) de una entidad para realizar consultas, para solucionar este problema se utiliza otro patrón de diseño conocido como CQRS [15] (Command Query Responsibility Segregation). En dicho patrón la aplicación se divide en dos partes: el lado de comando y el lado de consultas. El lado de comando maneja operaciones de creación, actualización y borrado, emitiendo los eventos pertinentes; esencialmente este es el lado de Event Sourcing. El lado de consultas maneja consultas, las cuales se ejecutan contra una o más vistas materializadas de la entidad; por ejemplo, una vista materializada puede ser obtenida a través de ejecutar los eventos de la cola de la entidad en el almacén de eventos. Para crear vistas de forma más eficiente, se pueden guardar imágenes periódicas de las entidades en una base de datos, de esta forma la reconstrucción de la entidad también se

simplifica al no tener que ejecutarse todos los eventos históricos para reconstruir el estado actual.

La aproximación coreografiada es una forma natural y descentralizada de implementar el patrón Saga, que promueve bajo acoplamiento entre microservicios, sin embargo, conforme más microservicios participan en la transacción, se vuelve más complejo entender el flujo de la misma y determinar qué servicios escuchan qué eventos, es también necesario evitar dependencias cíclicas que puedan crear bucles infinitos de eventos. Además de esto, para ser confiable y utilizable, requiere ser acompañado de otros patrones de diseño, e.g., Event Sourcing y CQRS, lo cual puede complicar la implementación y traer otros problemas consigo. Por ejemplo, al utilizar Event Sourcing, el almacén de eventos representa un punto central de fallo. Así mismo, a pesar de la mejora de confiabilidad que Event Sourcing establece, sigue sin definirse una forma o recomendación para manejar casos en los que un evento se emite pero el servicio receptor se encuentra inaccesible; de no tratarse casos como este, la transacción queda incompleta y con datos inconsistentes en el almacén de eventos.

Muchos de los problemas mencionados de la aproximación coreografiada y sus patrones asociados se deben a que la transacción en si no existe como un concepto de alto nivel, la transacción queda oculta a bajo nivel en forma de eventos, lo que complica su tratamiento atómico. En la aproximación que se propone, se intentan remediar, o al menos mitigar, los problemas de aproximaciones existentes, mientras al mismo tiempo se simplifica el proceso al mantener una visión de alto nivel de las transacciones, explotando el nivel de abstracción elevado que los sistemas Multi-Agente permiten.

3. Propuesta

En esta sección se propone una aproximación para implementar transacciones distribuidas para la arquitectura de microservicios, dicha aproximación tiene las siguiente características:

- Basada en agentes. Se explotan conceptos de sistemas Multi-Agente tales como comunicación basada en actos de habla [10] y coordinación.
- Basado en consistencia eventual, dado que es el modelo de persistencia de datos que permite mayor escalabilidad y heterogeneidad.
- La aproximación propuesta puede verse como una posible implementación del patrón Saga introducido en la sección 2.1.
- Sigue un modelo semi-orquestado asíncrono. Es simi-orquestado porque no existe una unidad central de mando sino que cada agente le pide a otro que realice el siguiente paso; es asíncrona ya que los agentes son capaces de comunicar o pedir algo sin necesidad de esperar una respuesta, así mismo los agentes pueden exhibir comportamiento reactivo cuando un evento sucede.
- Define un modelo de transacción que permite describir flujos de trabajo así como configuraciones de transacciones.
- Propone estrategias de manejo de errores para la detección, tolerancia y recuperación de errores, dichas estrategias explotan el nivel de abstracción elevado de los sistemas Multi-Agente.

La idea general de la propuesta es que cada microservicio tenga asociado un agente particular, dicho agente puede estar o no en el mismo servidor. Cada transacción nace en algún microservicio, el cual cuenta con los medios para comunicar, de forma directa o indirecta, el inicio de la transacción a su agente asociado. La forma concreta de comunicación depende de la implementación, por ejemplo, si tanto el microservicio como el agente se encuentran en el mismo servidor, la comunicación puede establecerse de forma simple mediante un pipe, o podrían utilizarse tecnologías de agente concretas tales como CArtAgO [13] que le permiten al agente reaccionar ante señales emitidas por el microservicio. Una vez el inicio de la transacción le es informada al agente, la transacción es manejada por una capa Multi-Agente independiente, la cual se encuentra también a cargo del manejo de errores referentes a transacciones y a la propia capa de agentes. La figura **1** muestra la idea general de la aproximación propuesta.

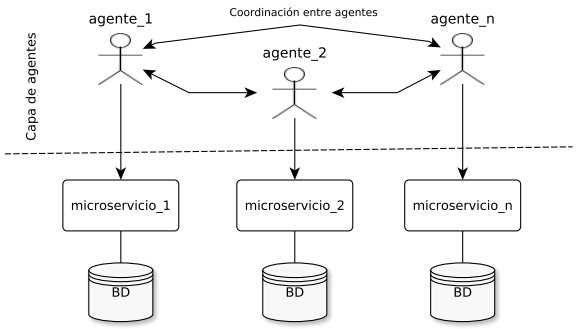

Fig. 1. Modelo general. Cada microservicio tiene asociada una base de datos BD y un agente. Los agentes, en la capa de agentes independiente, se coordinan entre si para realizar transacciones entre microservicios.

3.1. Modelo de transacción

En esta sección se presenta un modelo de transacción propuesto, el cual facilita la configuración y representación a nivel agente de las transacciones, dicho modelo se apoya en conceptos de teoría de grafos.

Definición 1 *Una transacción* $T = \{St_1, St_2, ..., St_n\} n >= 1$ *es un conjunto de caminos simples, i.e., que no pasan más de una vez por el mismo vértice, llamados sub-transacciones.*

Definición 2 *Una sub-transacción* $St = \{sc_1, sc_2, ..., sc_m\} m >= 1$ *es un conjunto de caminos de longitud 1 llamados secciones.*

De esta forma, las transacciones pueden verse como secuencias lineales que son descompuestas en secciones. En cada sección se ejecuta un paso de la transacción, para esto, es necesario contar con información asociada.

Definición 3 *$IS = \langle tr\_id, st\_name, input\_data, incoming\_action, output\_data, compensation\_action, state, next\rangle$ es un tupla de información asociada a la sección, donde:*

- *tr_id: identificador único de la transacción a la cual pertenece la sección.*
- *st_name: nombre lógico de la sub-transacción a la cual pertenece la sección.*
- *input_data: datos asociados para ser procesados por la sección.*
- *incoming_action: nombre de la acción del microservicio a ser ejecutada en la sección. La acción recibe input_data.*
- *output_data: datos producidos por incoming_action.*
- *compensation_action: nombre de la acción del microservicio a ser ejecutada si incoming_action ya ha sido ejecutada y algo salió mal en cualquier punto de la transacción. La acción recibe output_data.*
- *state: establece el estado de la ejecución de la sección, es un valor del conjunto $\{pendant, processing, processed, pendant\_compensation, processing\_compensation, processed\_compensation\}$.*
- *next: agente que continuará con la sub-transacción st_name para la transacción actual tr_id. También puede establecer un enlace entre la sub-transacción actual st_name y una nueva new_st_name, esto en caso de que se requiera visitar nuevamente a una agente, esto es, crear un camino que no sea simple.*

La aproximación propuesta no tiene lineamientos específicos en cuanto a cómo representar o crear *input_data* y *output_data*, esto depende de los desarrolladores de microservicios y de la implementación especifica del modelo. Por ejemplo, se puede preferir el uso de un formato de texto estándar como JSON donde los desarrolladores pueden elegir utilizar sólo algunos de los datos de entrada para sus operaciones.

Es también interesante notar que el modelo permite la creación de sub-transacciones que trabajan de forma paralela, esto gracias a que es posible ligar una sub-transacción existente con una nueva a través de *next*, siempre y cuando no haya dependencia directa entre las sub-transacciones, en cuyo caso se puede plantear una sola sub-transacciones secuencial.

3.2. Configuración

La mayoría de información asociada a una sección, descrita en la definición 3, puede ser generada de forma dinámica por la capa de agentes, sin embargo, algunos datos deben ser configurados por los desarrolladores de microservicios: *st_name*, *incoming_action*, *compensation_action* y *next*. Se propone un formato de configuración basado en lógica de primer orden el cual tiene la finalidad de representar las transacciones en las cuales los agentes participan. La configuración se realiza por agente, y es, junto con las acciones a ejecutar, los únicos aspectos referentes a transacciones por los cuales debe de preocuparse el desarrollador de microservicios. El formato propuesto contiene los siguientes predicados:

- $incoming\_action(st\_name, action)$: establece la acción *action* que se ejecutará en el microservicio dada la sub-transacción *st_name*. La acción se ejecuta utilizando *input_data* y genera *output_data* como se mencionó en la sección 3.1.
- $compensation\_action(st\_name, action)$: establece la acción *action* a ser ejecutada en caso de que haya algún problema en la transacción y $state = processed$ para la sección correspondiente *st_name*. La acción se ejecuta utilizando *output_data*.
- $next(st\_name, agent)$: establece al siguiente agente *agent* para continuar la sub-transacción *st_name*.
- $next(st\_name, new\_st\_name, agent)$: liga una sub-transacción *st_name* existente con una nueva *new_st_name* y establece el agente *agent* que empezará con dicha sub-transacción. Este predicado es de utilidad para crear caminos no simples en la transacción sin ambigüedad, i.e., cuando el siguiente agente de una sub-transacción ya había sido visitado.

Como ejemplo de configuración, considérese un caso donde un cliente quiere establecer una orden de compra sobre un producto. A continuación se muestra una posible configuración utilizando el formato propuesto.

```
1  //Agente order
2  incoming_action(orderCreation, createPendingOrder).
3  compensation_action(orderCreation, deleteOrder).
4  next(orderCreation, customer).
5  incoming_action(orderApproval, approveOrder).
6  compensation_action(orderApproval, cancelOrder).
```

```
1  //Agente customer
2  incoming_action(orderCreation, reserveCredit).
3  compensation_action(orderCreation, cancelReservation).
4  next(orderCreation, orderApproval, order).
```

El flujo de la transacción descrito en la configuración puede ser representado fácilmente mediante un diagrama similar al de secuencia de UML, el cual se muestra en la figura **2**. Notar que la figura sólo muestra el flujo normal de la transacción, i.e., sin acciones de compensación. Así mismo, por legibilidad, en la figura no se muestran los datos de entrada y salida de cada acción. Esta facilidad de representación es gracias al manejo de alto nivel de las transacciones y muestra la posibilidad de una herramienta que genere automáticamente los archivos de configuración a través de una interfaz gráfica donde se modela el flujo principal y alterno (acciones de compensación) de la transacción.

3.3. Flujos de trabajo

Además de un modelo de transacción y una forma de configuración, es necesario que la capa de agente realice diversas labores que permitan dar soporte a las transacciones. En esta sección se presenta, desde la perspectiva de la capa de agente, el flujo de trabajo de una transacción, incluyendo los casos cuando la

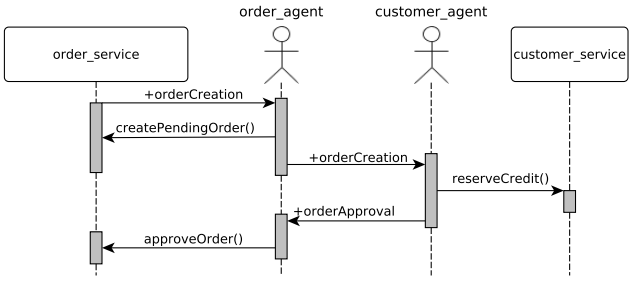

Fig. 2. Flujo normal de la transacción del ejemplo de ordenes y clientes. El símbolo "+" se utiliza para denotar una nueva creencia.

transacción se realiza de forma normal y cuando se genera algún fallo. El flujo normal de una transacción es el siguiente:

1. Un microservicio comunica el inicio de la transacción a su agente asociado, entregando datos de entrada *input_data* y el nombre de la sub-transacción inicial *st_name*.
2. El agente establece qué agentes participarán en la transacción y qué sub-transacciones forman parte de la transacción. Esta información debe ser conocida por todos los agentes y dependiendo de la implementación, podría conocerse de antemano o ser descubierta.
3. Un identificador único *tr_id* es creado para la transacción.
4. En cada sección el agente correspondiente ejecuta el *incoming_action* apropiado, pasando *input_data* como entrada y generando *output_data* como salida.
5. Utilizando la información de *next*, el agente de la sección pasa *output_data* y un *st_name* al siguiente agente.
6. Cuando una sub-transacción finaliza, el último agente en el extremo final se lo comunica a los demás agentes de la transacción. Una sub-transacción finaliza cuando ya no tiene asociado un *next* en la sección actual, o cuando en *next* se liga una sub-transacción nueva.
7. Cuando todas las sub-transacciones finalizan, la transacción finaliza.

El flujo alterno de compensación sólo considera problemas que pudieran darse a nivel de la acción *incoming_action*, por ejemplo, en el contexto de ordenes y clientes, un error de este tipo puede ser que el cliente no cuente con crédito suficiente, en dicho caso la acción falla y por lo tanto toda la transacción. Una posibilidad de manejo de errores derivados de perdida de conectividad y caída de agentes se específica en la sección siguiente. El flujo de compensación ante un fallo generado en una acción es el siguiente:

1. Un *incoming_action* de una sección falla y el agente asociado lo percibe.

2. El agente envía un mensaje a los demás agentes de la transacción para que detengan su trabajo relacionado con la transacción, esto en caso de que haya sub-transacciones paralelas.
3. Los agentes consideran el estado *state* de sus secciones y ejecutan las acciones *compensation_action* en caso de ser necesario, utilizando el *output_data* apropiado para la sección. La ejecución de los *compensation_action* se realiza en el orden inverso a los de *incoming_action*.
4. Cuando un agente termina de ejecutar sus acciones de compensación, se lo comunica a los demás agentes de la transacción.
5. Cuando todos los agentes terminan, la transacción termina.

3.4. Estrategias de manejo de errores

En un sistema distribuido es importante considerar el fallo independiente de componentes y establecer estrategias de detección, tolerancia y recuperación de errores apropiadas [3]. En este apartado, se propone una serie de posibles estrategias a implementar para el manejo de errores, éstas sólo consideran problemas que pueden surgir en la capa de agente ya que los problemas derivados de microservicios pueden ser manejados a través del flujo alterno de compensación presentado en la sección 3.3. Estas estrategias son de vital importancia pues un mal manejo de éstas puede crear inconsistencias de datos y por consiguiente pérdida de integridad. Para el manejo de errores, se consideran los dos tipos de errores más generales que pueden ocurrir:

1. Pérdida de conectividad entre agentes. Se refiere a problemas de red, los cuales pueden ser temporales y afectan la coordinación entre agentes.
2. Caída de algún agente. Agrupa problemas tales como la generación de una excepción no controlada hasta fallos de hardware que provocan la detención abrupta del proceso del agente.

De los dos tipos de errores generales mencionados, el más severo es la caída de agentes puesto que puede requerir de un reinicio manual y una recuperación de error más compleja, sin embargo, desde el punto de vista de los agentes, el efecto de cualquiera de los dos errores mencionados es no poder establecer alguna comunicación con los agentes afectados. Así mismo, la recuperación de errores de caída tiene elementos en común de la recuperación de error derivada de problemas de conectividad. De esta forma, la estrategia de detección de errores propuesta se basa sólo en pérdida de conectividad y se propone una estrategia general de recuperación de errores que aplica a errores de conectividad y parcialmente a errores de caída, posteriormente se propone una extensión para la recuperación de errores de caída. La estrategia de detención, tolerancia y recuperación de errores general es la siguiente:

- Como parte de la capa de agentes, cada agente mantiene un intercambio periódico de información de conectividad con uno o dos agentes. Esta conectividad es lineal, de esta forma, el intercambio puede ser de izquierda a derecha y de derecha a izquierda. Los agentes de cada extremo sólo mantienen

intercambio con un agente que le da información sobre la conectividad de todos los agentes en el extremo opuesto; mientras que los demás agentes mantienen intercambio con otros dos agentes, del agente de su izquierda reciben información de conectividad de los demás agentes del lado izquierdo, mientras a su vez le informan sobre la conectividad de los agentes que están hacía la derecha, esto mismo ocurre del lado derecho de forma inversa. El proceso descrito anteriormente se muestra en la figura **3**.

- Con la información de conectividad, los agentes pueden determinar si tiene sentido o no iniciar una nueva transacción o continuar alguna en curso. Este aspecto puede verse como una forma de tolerancia a fallos.
- Si un agente pierde conectividad, este hecho es reportado a todos los agentes por uno de los agentes que esperaba un mensaje del agente afectado. Durante esta perdida, es posible que algún agente haya intentando contactar con el agente afectado, esto como parte del trabajo de una transacción, si este es el caso, entonces la transacción queda en pausa.
- Los agentes no afectados esperan un periodo de tiempo de gracia antes de realizar alguna recuperación de error, por si una reconexión ocurre.
- Durante el periodo de gracia, los mensajes de conectividad se intentan enviar de forma normal.
- Si la reconexión ocurre durante el periodo de gracia, entonces los agentes que tenían pendiente contactar con el agente afectado reinician su trabajo.
- Si la reconexión no ocurre durante el periodo de gracia, los agentes con conectividad ajustan su intercambio de información de conectividad con los agentes disponibles y continúan trabajando, todo esto como medida de tolerancia a errores, al mismo tiempo, siguen intentando contactar con los agentes no alcanzables. Los agentes afectados por transacciones incompletas inician un proceso similar al descrito en el flujo alterno de compensación mencionado en la sección 3.3, con la salvedad de que la transacción no se considera como terminada sino hasta que eventualmente haya una reconexión y se pueda realizar una limpieza completa.

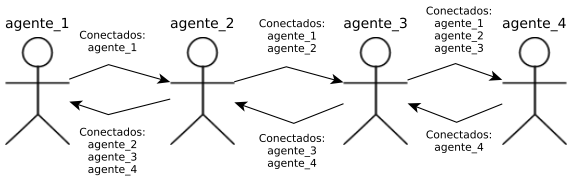

Fig. 3. Intercambio de información de conectividad entre agentes. Cada agente comparte su información con uno o dos agentes para mitigar carga de red.

Es importante notar que al ocurrir un problema de conectividad es posible que varios agentes en un mismo segmento de red pierdan conectividad con los

agentes de otro segmento, en la estrategia mencionada, cada segmento lleva a cabo los pasos antes descritos de forma independiente. La estrategia general de recuperación de error ante una reconexión que sucede después del tiempo de gracia es la siguiente:

- Cuando uno o varios agentes se reconectan se intenta recrear el intercambio de conectividad como estaba antes de la perdida de conexión.
- Si todos los agentes que forman parte de alguna transacción incompleta se reconectan, la transacción se da por finalizada, considerando que dichos agentes ya ejecutaron sus acciones de compensación.

La extensión de la estrategia antes mencionada para caídas es la siguiente:

- Cada agente mantiene una bitácora persistente del estado de cada una de sus transacciones.
- Cuando el agente se reinicia, éste verifica su bitácora y ejecuta las acciones de compensación adecuadas.
- El agente se reconecta con los demás agentes.

Para que el proceso de recuperación de errores derivado de caídas sea confiable, es necesario que existe atomicidad entre la escritura de la bitácora y la realización de las acciones que cambian el estado de la transacción, esto puede lograrse de diversas formas a nivel de implementación, por ejemplo, utilizando transacciones de código tradicionales de forma local.

4. Conclusiones y trabajo futuro

La aproximación presentada en este trabajo promueve un modelado de alto nivel de transacciones distribuidas escalables y heterogéneas, proveyendo una definición flexible de transacción, la cual es posible representar como conocimiento de agente a través del formato de configuración propuesto. El utilizar sistemas Multi-Agente permite la definición de alto nivel de flujos de trabajo y estrategias de manejo de errores, dejando a su vez abiertas oportunidades para mejorar aspectos del proceso a través de esquemas más sofisticados de negociación y razonamiento de agente que exploten técnicas de Inteligencia Artificial.

Desde el punto de vista de ingeniería de software, la aproximación propuesta promueve un bajo acoplamiento entre microservicios, ya que las coordinación de las transacciones se encuentra aislada en la capa de agentes. De igual forma, el acoplamiento entre el lado de microservicios y la capa de agentes se mantiene bajo. Desde el punto de vista de los microservicios, no existe la capa de agentes, salvo quizás para resolver cosas como el inicio de una transacción, lo cual es dependiente de la implementación concreta; a su vez, la capa de agentes es genérica, puede adaptarse a cualquier sistema basado en microservicios por lo que no existe una dependencia de dominio entre ambas partes, permitiendo de esta forma que cada parte evolucione de forma separada. Este acoplamiento bajo en todos los niveles tiene como beneficio facilitarle en gran medida a los desarrolladores el manejo de transacciones distribuidas, ya que éste no debe

preocuparse de establecer código especial, como por ejemplo para manejar eventos, sino que la gestión confiable y eficiente de la transacción la lleva a cabo la capa de agentes. Así mismo, no se tiene un sólo punto central de fallo o de concentración de tráfico, como en muchas otras aproximaciones existentes discutidas en la sección 2. El trabajo presentado es un esfuerzo para acercar cada vez más a los sistemas Multi-Agente al desarrollo de software tradicional, mostrando que una aproximación de este tipo tiene argumentos suficientes para ser considerada seriamente en Ingeniería de Software.

La propuesta, más que una guía de cómo llevar a cabo el proceso de transacciones distribuidas, como es el caso de los patrones de diseño presentados en la sección 2.1, da lineamientos más específicos de cómo realizar una implementación funcional que considera también aspectos de manejo de errores. Esta guía es de gran valor para la creación futura de una implementación específica, lo cual se planea hacer como trabajo futuro inmediato. En esta implementación futura será necesario escoger una pila de desarrollo Multi-Agente, teniéndose como principal opción a JaCaMo [2], el cual cuenta con los elementos suficientes para la creación de sistemas Multi-Agente distribuidos basados en el modelo BDI [12] de agencia.

El modelo BDI deja abiertas posibilidades para mejorar aspectos de razonamiento, negociación y coordinación entre agentes. Uno de los objetivos futuros es explorar esquemas más sofisticados en el manejo de transacciones, para que, por ejemplo, los agentes sean capaces de razonar si se debe ejecutar o no una operación sobre datos que están siendo manipulados en otra transacción, con el objetivo de mitigar posibles problemas derivados del uso de consistencia eventual.

Referencias

1. Bailis, P., Ghodsi, A.: Eventual consistency today: Limitations, extensions, and beyond. Queue 11(3), 20 (2013)
2. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming 78(6), 747–761 (2013)
3. Coulouris, G.F., Dollimore, J., Kindberg, T.: Distributed systems: concepts and design. pearson education (2005)
4. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
5. Garcia-Molina, H., Salem, K.: Sagas, vol. 16. ACM (1987)
6. Josuttis, N.M.: SOA in practice: the art of distributed system design. .º'Reilly Media, Inc."(2007)
7. Limprecht, R.: Microsoft transaction server. In: Compcon'97. Proceedings, IEEE. pp. 14–18. IEEE (1997)
8. Moemeng, C., Gorodetsky, V., Zuo, Z., Yang, Y., Zhang, C.: Agent-based distributed data mining: A survey. In: Data mining and multi-agent integration, pp. 47–58. Springer (2009)
9. Mohan, C., Lindsay, B.: Efficient commit protocols for the tree of processes model of distributed transactions. In: Proceedings of the second annual ACM symposium on Principles of distributed computing. pp. 76–88. ACM (1983)

10. Moreira, Á.F., Vieira, R., Bordini, R.H., et al.: Extending the operational semantics of a bdi agent-oriented programming language for introducing speech-act based communication. Lecture notes in computer science pp. 135–154 (2004)
11. Newman, S.: Building microservices: designing fine-grained systems. .ºReilly Media, Inc."(2015)
12. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI architecture. Tech. rep., Australian Artificial Intelligence Institute, Melbourne, Australia (1991)
13. Ricci, A., Viroli, M., Omicini, A.: Construenda est cartago: Toward an infrastructure for artifacts in MAS. Cybernetics and systems 2, 569–574 (2006)
14. Richardson, C.: Microservice architecture patterns and best practices. URL: http://microservices. io/index. html [accessed: 2018-03-17] (2016)
15. Young, G.: Cqrs and event sourcing. feb. 2010. URl: http://codebetter. com/gregyoung/2010/02/13/cqrs-and-event-sourcing (2010)