“Transacciones distribuidas”

Juan Sebastián Díaz Serrano, Jorge Gómez, Sergio Guzmán Mayorga, Mauricio Neira
Felipe Ramos, Juan Camilo Ruiz
Iteración 5
Universidad de los Andes, Bogotá, Colombia
{js.diaz, ja.gomez10, s.guzmanm, m.neira10, jf.ramos, jc.ruiz}@uniandes.edu.co
Fecha de presentación: Diciembre 10 de 2017

Tabla de contenido

1 Introducción

Con el fin de resolver la mayoría de preguntas en [1] se plantea el siguiente informe para documentar el manejo de transacciones distribuidas, especificando el análisis de su implementación (restricciones de rotondas, lógica del requerimiento RF18 y estrategias para el cumplimiento del caso de uso RF18), la forma en que se implementaron las transacciones y el impacto de diferentes estrategias para satisfacer los requerimientos planteados en la entrega. Por último, se dejan dos espacios para especificar consideraciones adicionales del proyecto y el análisis de resultados del mismo.

2 Análisis de la implementación de transacciones distribuidas

2.1 Restricciones de rotondas

A continuación, se presentan las restricciones planteadas de cada rotonda para su negocio. Se asume que todas las rotondas del sistema utilizan una interfaz XA de X/Open junto con la implementación de RabbitMQ para el manejo de colas de mensajes.

2.1.1 Rotonda A-05

- Para que un restaurante esté registrado en la rotonda debe tener un representante como usuario de la rotonda. Dicho representante presenta un rol de LOCAL.
- Se asume que hay restaurantes que pueden ofrecer el mismo producto al usuario. Sin embargo, lo referente a disponibilidad, precios y costos dependen de cada local.
- Pueden existir dos menús con el mismo nombre en la rotonda, pero la descripción, costo, productos, precio, etc. dependen del restaurante que lo está ofreciendo.
- Una cuenta tiene o no asignada un cliente y a su vez puede o no estar registrada a una mesa. Ahora por la conformación de la "super-rotonda" se plantea que la mesa sea representada por una cadena de caracteres de la forma:

<Número de la mesa>-<Nombre de la zona>-<Nombre de la rotonda>

2.1.2 Rotonda A-17

- La petición de productos se realiza mediante la creación de carritos por lo que todos los datos de compra, tanto usuarios como información de los productos debe ser incluida inicialmente. Igualmente, la información de productos dentro de carritos que fueron correctamente comprados se encuentra en una estructura de Historial.
- En el ámbito de productos, tanto menús como productos individuales son dados como tal. De esa forma, hay una tabla que proporciona la diferenciación necesaria entre productos individuales y menús.
- Cada producto se asocia a un restaurante. De esta manera se genera diferenciación entre productos con un mismo nombre. Cabe resaltar nuevamente que productos incluye tanto productos individuales como menús.
- Cada usuario registrado posee un email, sin embargo, puede que un usuario no registrado también lo posea.
- Dentro de carritos se incluye la información de la zona de petición de productos. Con el fin de evitar ambigüedades, se incluye una columna que indica tanto rotonda como mesa dentro de la zona; todo esto en la tabla de carrito.
- La creación de un restaurante implica la existencia de un usuario administrador del mismo.

2.1.3 Rotonda B-14

- Para que un restaurante se registre en la rotonda, este debe tener asociado una Zona existente.
- Cada producto tiene asociado un restaurante, de modo que varios restaurantes pueden tener un producto con el mismo nombre, pero un restaurante no puede tener dos productos con el mismo nombre. Por producto se incluye tanto platos como menús.
- Con la conformación de la super rotonda, ahora los pedidos tienen una nueva columna llamada ZONA_ROTONDA, la cual corresponde a la zona de la rotonda a la que pertenece.

2.2 Lógica del requerimiento RF18

Dentro de la implementación para el requerimiento de hacer un pedido con productos de diferentes rotondas, tenemos un esquema mediante el cual realizamos la mayor cantidad de

entradas posibles para el pedido en cada rotonda. En este caso, quien recibe todas las entradas iniciales es la Rotonda 1, la cual responde con las entradas que pudo procesar en su base de datos, además de con la información del pedido relacionado con estas entradas. Posteriormente, el administrador filtra las entradas para excluir las ya pedidas por la Rotonda 1, para que la Rotonda 2 realice el mismo proceso que la Rotonda 1, y retorna un consolidado de las entradas que pudo procesar junto con la información del pedido relacionado con estas entradas, el proceso para la Rotonda 3 es análogo.

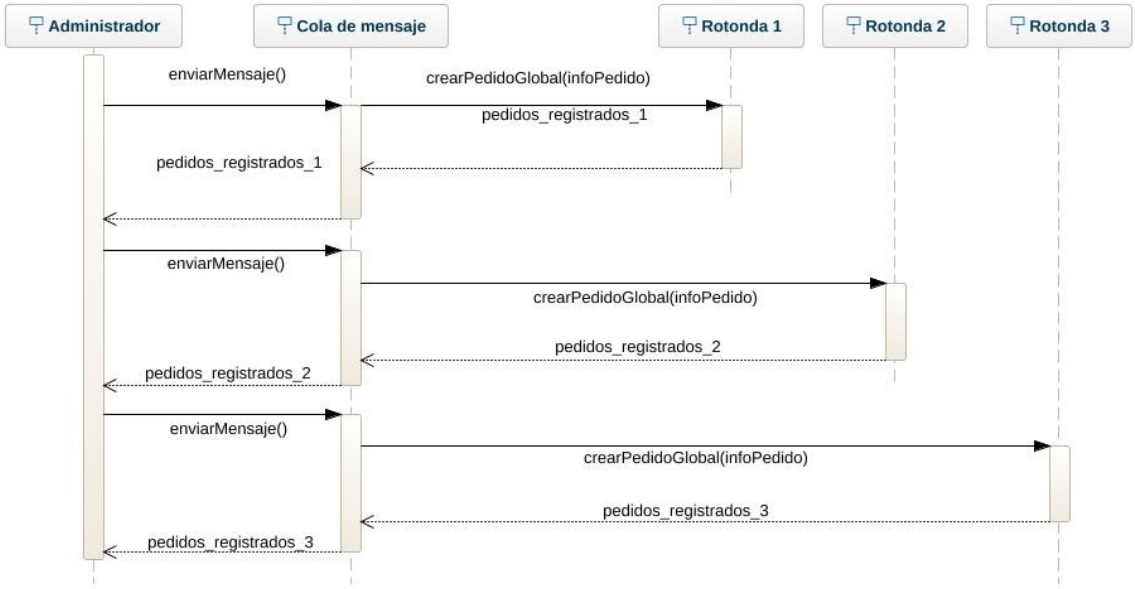

Figura 1. Comunicación entre rotondas desde el diagrama de secuencia.

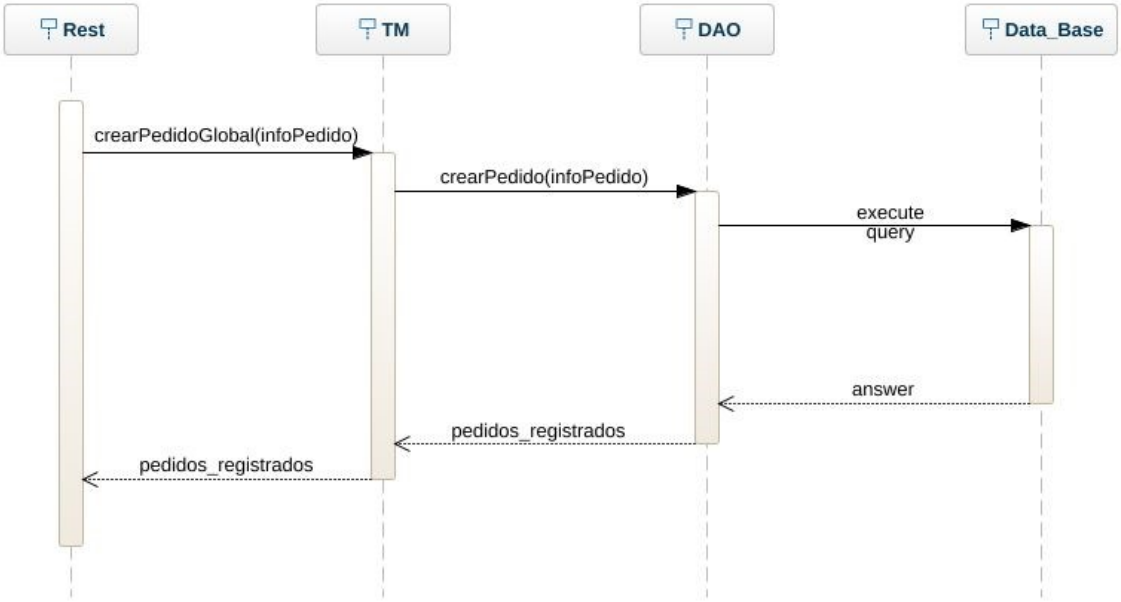

Figura 2. Protocolo de comunicación desde una perspectiva de capas de la aplicación.

2.3 Estrategias para el cumplimiento de casos de uso

2.3.1 Colas de mensajes

Como el protocolo AMQP para colas de mensajes no soporta nativamente transacciones distribuidas, específicamente atomicidad, esta se debe asegurar por medio de la lógica de la aplicación. Para ello, tenemos dos casos, si el caso de *rollback* de la operación es una salida usual (con en el RF18) o si solamente se da en caso de error (como en el RF19).

Para el primer caso, la estrategia diseñada fue tener la operación usual y la operación de *rollback* como dos operaciones transaccionales diferentes. La primera operación se ejecuta normalmente, esperando a que el pedido sea atendido, y se define un tiempo máximo de espera. Si al final de este (cuando se genera un *timeout*) no se ha logrado atender la totalidad del pedido, esta transacción termina, y se ejecuta la transacción de *rollback*. Esta también puede tener fallos, pero solo se dan en casos excepcionales, por lo que se hace uso de la otra estrategia para asegurar ACID, que se explicará más adelante. Esta estrategia es óptima para el RF18 ya que, además de lo ya dicho, es naturalmente acoplado en el tiempo, se debe atender el pedido casi en el mismo momento en el que se pide. Esto hace que el concepto de *timeout* sea consistente con la lógica de la aplicación.

El RF19, y el *rollback* del RF18 no comparten estas características. No requieren de sincronización en el tiempo, y sus casos de error son inesperados. Luego, para ellos se maneja una estrategia de consistencia eventual (el estado de la SuperRotonda no es consistente en todo momento, pero lo será eventualmente), por medio de *retries*. En estos, si alguna Rotonda presenta un error al llevar a cabo la operación, se intenta hacer de nuevo hasta que finalmente se logre. Lo mismo sucede (en el RF19) si la Rotonda no se encuentra conectada, el mensaje se queda en cola hasta que esta se conecte y ejecute la operación necesaria. Afortunadamente, las operaciones de estos requerimientos son naturalmente nilpotentes (aplicar la operación varias veces es lo mismo que aplicarla una sola vez), así que una estrategia de *retries* no causará problemas en la base de datos.

2.3.2 Two-Phase Commit

El protocolo *Two-Phase Commit*, como está diseñado específicamente para la ejecución de transacciones distribuidas, requiere menos modificaciones a la lógica de la aplicación que el uso de colas de mensajes. Para ambos requerimientos, la estrategia de ejecución diseñada consiste en que cada Rotonda ejecuta la operación pedida, pero no hace *commit*, y después de esto da su aceptación al manejador de transacciones, y la segunda fase consiste solo de ejecutar *commit* o *rollback*, dado sea el caso, sobre la operación ejecutada. Esta estrategia sigue al pie de la letra los requerimientos del protocolo, y por tanto resulta, demostrablemente, aceptable en la ejecución correcta de los requerimientos.

2.3.3 Comparación

En términos generales, el *Two-Phase Commit* da mayor seguridad de la ejecución correcta de los requerimientos, mientras que las colas de mensajes permiten que estas operaciones sean más eficientes y desacopladas. Esto último es especialmente importante para el RF19, ya que este requerimiento puede ser desacoplado en el tiempo; sería una restricción innecesaria pedir que solo se pueda ejecutar un retiro de un restaurante solamente cuando los sistemas de las tres rotondas se encuentran conectados. Luego, para este requerimiento es muy preferible el uso de Colas de Mensajes como protocolo. El único caso en que esto sería debatible es si los candados en la base de datos son por tupla, ya que entonces el *Two-Phase Commit* puede demorarse indefinidamente sin poner el funcionamiento de la aplicación en riesgo.

En cuanto al requerimiento RF18, es preferible el uso de *Two-Phase Commit* ya que esta operación es acoplada en el tiempo, y es más importante la realización correcta de las propiedades ACID que el desacoplamiento. Mas aún, la realización de este requerimiento con Colas de Mensajes requiere de mucha lógica adicional, y por tanto está mucho más sujeto a errores.

3 Especificación e implementación de transacciones distribuidas (Protocolos de requerimientos)

3.1 RF18

La cola mandará un mensaje de la forma:

<email_usuario>;<nombre_zona>;<cantidad_productos>(;<nom_producto>,<nom_restaurante>)*

Donde la cantidad de tuplas *(;<nom_producto>,<nom_restaurante>)* es equivalente a la cantidad de productos a pedir. Cada rotonda deberá retornar un mensaje donde especifique cual fue la instancia creada para asegurar el pedido (cuenta, pedido, carrito, etc.) además de una lista (definida como un VO estándar) de los productos que logro agregar al pedido. En caso tal de que exista un *timeout*, o algún otro tipo de excepción, las rotondas que pusieron el pedido, se les mandara por cola de mensajes la información resultante del pedido para que lo desvinculen de la base de datos.

3.2 RF19 v1 (Con colas de mensajes)

La cola mandara un mensaje de la forma *<nombre_restaurante>;<app_destino>*, de tal manera que cada rotonda lo deshabilite. En caso de que se dé un *timeout* en alguna de las rotondas, la cola volverá a mandar el mismo mensaje, para que se vuelva a habilitar el restaurante, hasta que la sentencia pase en cada una de las rotondas. El identificador de la aplicación de destino se incluye para poder verificar que todas las aplicaciones hagan la operación, ya que el protocolo por si solo no soporta esto.

3.3 RF19 v2 (Con *Two-Phase Commit*)

Se realizará un *request* individual para cada rotonda, de tal manera que retorne la sentencia requerida para cumplir el requerimiento, junto con las credenciales para acceder a la base de datos. Posteriormente, al tener todas las sentencias con sus respectivas credenciales, se realizará la ejecución simultanea de estas de tal manera que el *commit* sea global, asegurando un *Two-Phase Commit*.

3.4 RFC13

Se manda un mensaje de la forma:

<fechaInicio>;<fechaFin>;<nombreRestaurante>;<catProd>;<precioMin>;<precioMax>

Internamente cada aplicación lee el mensaje y devuelve un *object* con el resultado de la consulta. Si alguna de estas se deja vacía se asume un ordenamiento, y si se tiene como valor *null*, no se tendrá en cuenta.

3.5 RFC14

Se manda un mensaje de la forma

<fechaInicio>;<fechaFin>;<nombreRestaurante>

Internamente cada aplicación lee el mensaje y devuelve un *object* con el resultado de la consulta. Si alguno de los valores se deja vacío o se tiene como valor *null*, no se tendrá en cuenta.

4 Especificación e implementación de transacciones distribuidas (Análisis de impacto de estrategias)

Para analizar convenientemente las estrategias a utilizar dentro del cumplimiento de los requerimientos planteados, es pertinente resaltar los requerimientos y sus peticiones con el fin de reconocer elementos pertinentes para la resolución de dichos requerimientos. De tal modo, procedemos a mostrar requerimientos de modificación y consulta para reconocer posibles mecanismos y estrategias de solución.

En primer lugar, el requerimiento RFC13 trata de obtener la información completa de todos los productos que puedan estar en cualquier base de datos dentro del sistema. Es por ello que, dentro del proceso de adquisición de esta información, necesitaremos bien sea del mecanismo de colas de mensajes como del mecanismo de *Two-Phase Commit*. Dentro del caso particular de las colas de mensajes, el funcionamiento compete en enviar solicitudes a una cola de mensajes común en donde cada sistema de bases de datos analizará cada mensaje a fin de leer la solicitud y computar si dicha solicitud está dirigida a esa base de datos en particular. A continuación, el proceso de obtener toda información posible de los productos de cada una de las bases de datos culminará al unir todos los datos disponibles en una cola de respuestas común.

Cabe resaltar que dichos datos pueden estar incompletos al no poder rectificarse con anterioridad si alguna base de datos falla en el proceso de obtención de productos. Dado ese inconveniente, entra en escena el uso del protocolo de *Two-Phase Commit*. En este mecanismo, un sistema centralizado verifica con anterioridad tanto posibilidad como disponibilidad de cada base de datos pertinente en la realización del requerimiento. En lo que se conoce como fase de preparación, se alerta sobre la posibilidad de realizar el requerimiento a cada base de datos. En la fase de votación se genera una respuesta acorde al requerimiento sobre la posible realización y dependiendo de la respuesta se escoge la realización o el aborto de las transacciones. Así, en cuanto a transacciones de consulta, la diferencia radica en que con una cola de mensajes es posible obtener información parcial mientras que con el protocolo *Two-Phase Commit* se obtiene de manera binaria: todo o nada. Por ende, el análisis hecho aquí es repetido para RFC14. Finalmente, concluimos que *Two-Phase Commit* asegura consistencia en la información obtenida por lo que es ideal para este tipo de necesidades de información.

Por otro lado, para los requerimientos de modificación de información, aunque el proceso en cuestión para cada mecanismo fue descrito de manera general, por lo que ocurriría de manera similar, el resultado es convenientemente distinto en cuanto a resultados dado que las bases de datos relacionales con los mismos serán filtradas en primera instancia. Siendo así, este proceso será explicado a continuación. En cuanto al requerimiento RF18, en la cola de mensajes no sería posible obtener de antemano la posibilidad de realización del requerimiento, es decir, no podría saberse si un base de datos puede satisfacer un pedido con anterioridad, por lo que nos encontraríamos con resultados vacíos que es aquello que implica un estado de rechazado. Por ende, es conveniente utilizar el protocolo *Two-Phase Commit* y reconocer con anterioridad si un proceso tiene una realización plausible dentro de los términos descritos, es decir, si hay disponibilidad de productos. Así, se logrará saber sobre cuales bases de datos realizar el proceso completo. De igual forma, un caso similar ocurre en cuanto al retiro de un restaurante (RF19): si no se puede realizar el retiro completo de todos los restaurantes relacionados con una franquicia, el retiro de dicha franquicia es imposible, por lo que se recurre al protocolo *Two-Phase Commit* para la realización correcta.

5 Consideraciones Adicionales

Se admite en general que el proyecto no corrió, y esto se debe a que el único sitio donde servía la aplicación era en el computador del integrante Sergio Guzmán Mayorga. Como solo se tenía el trabajo funcional en una máquina se tomó la decisión de implementar todos los métodos locales y que cada grupo probara de alguna forma que al recibir el mensaje esperado se hiciera lo que la aplicación tenía que hacer.

6 Análisis de Resultados

6.1 Aprendizajes y Logros

A lo largo del desarrollo de la iteración 5 se aprendieron dos formas sugeridas para implementar el uso de transacciones distribuidas: el uso de colas de mensajes y de protocolos de dos fases. La verdad, no se pudo evidenciar el desarrollo pleno de la primera modalidad, debido a que la aplicación solo servía en una máquina. Sin embargo, la segunda estrategia de transaccionalidad distribuida se pudo lograr al abrir tres comunicaciones con las bases de datos respectivas para que estuvieran conectadas en un mismo marco y así garantizar las propiedades ACID globales.

6.2 Conclusiones

• Es necesario tener el mismo protocolo de comunicación entre las aplicaciones para que la transaccionalidad distribuida sea eficiente.

• Las colas de mensajes garantizan el envío de instrucciones de forma asincrónica para que cada negocio solo tenga que enviar una solicitud y seguir con su trabajo común y corriente. Sin embargo, no hay garantía alguna de que, si la ejecución de dichas instrucciones llegase a fallar, el estado de la información pueda llegar a ser consistente.

• El protocolo de dos fases (*Two-Phase Commit*) garantiza que la información sea consistente con la realidad en todas las bases de datos involucradas. Sin embargo, es una estrategia transaccional que bloquea todos los recursos hasta que no se termine la transacción.

7 Bibliografía

1. Universidad de los Andes. *Transacciones Distribuidas.* [En línea] Universidad de los Andes. [Citado el: 10 de Diciembre de 2017.] https://sicuaplus.uniandes.edu.co/bbcswebdav/pid-2060664-dt-content-rid-21018979_1/courses/UN_201720_ISIS2304/isis2304-172-Iteracion5-Transacciones%20Distribuidas%281%29.pdf