UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 5

Centro de Consultoría y Desarrollo de Arquitecturas Empresariales (CDAE)

Módulo de gestión de notificaciones de conflictos de réplica en el Replicador Reko

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor: Helian Rivera Hernández

Tutor: Dariel Noa Graverán

La Habana, Junio del 2014

DECLARACIÓN JURADA DE AUTORÍA

Declaro ser autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo. Autorizo a dicho centro para que haga el uso que estime pertinente con este trabajo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año 2014.

Firma del autor:

_____

Helian Rivera Hernández.

Firma del tutor:

_____

Ing. Dariel Noa Graverán.

AGRADECIMIENTOS

A mi papá por ser mi guía desde que era un niño siempre respondiendo a todos mis por qué con una sabiduría inigualable. La persona que me introdujo a este mundo de la informática que tanto me gusta. Por ser ese padre que no cambiaría por nada en este mundo y por muchas cosas más, muchas gracias.

A mi mamá por estar siempre ahí para mí cuando lo he necesitado, incondicionalmente. Por ser quien eres y por darme todo tu amor y tu comprensión en todas las decisiones que he tomado. Por tus regaños que siempre me enseñan un poco más de la vida. Por ayudarme a ser quien soy y por mucho más te estoy eternamente agradecido.

A Mima y a mis otros abuelos Pipo, Juanito y Alla que ya no están conmigo. Por ser ejemplos de excelentes personas y enseñarme cómo de unida debe ser una familia. Muchas gracias por su apoyo cuando lo he necesitado.

A mi tía cacha, mi segunda mamá, por su todo cariño y su preocupación.

A mis amigos de pinar Tony, Erdin, Yuniesky, Raidel y demás. Ustedes son los hermanos que nunca tuve, gracias por preocuparse y por constantemente preguntarme cómo iba la tesis.

A mis compañeros de aula, en especial a Ramírez, Yeikel, Jesús, Armando y a toda la gente del 88. De ustedes aprendí siempre algo en mi estancia en la universidad. Ah y gracias también por acompañarme en el doble.

A Adrián Hernández Velozo, que sin ser compañeros de tesis hemos trabajado en conjunto y me ha ayudado mucho.

A mi tutor por ayudarme desde el primer día, por estar siempre al tanto de todo y por servirme de guía para la realización de este trabajo.

A Frank y a toda la gente del proyecto por su ayuda y apoyo.

En fin a todos los que de cierta forma han contribuido a mi formación profesional.

DEDICATORIA

A mi mamá y a mi papá por ser mi ejemplo a seguir, por enseñarme a ser una mejor persona y un mejor profesional. Sé que están muy orgullosos de mí y todo esto se lo debo a ustedes.

RESUMEN

Uno de los procesos fundamentales en la replicación de datos en Sistemas de Bases de Datos Distribuidos es la detección y resolución de conflictos de réplica. El presente trabajo de diploma detalla las características del módulo de notificaciones de conflictos del Replicador Reko perteneciente a la Universidad de las Ciencias Informáticas, un sistema de réplica para Sistemas Distribuidos. El funcionamiento del módulo desarrollado se enfoca en el envío y recepción de notificaciones de conflictos contando con un servidor de mensajería ActiveMQ utilizando el protocolo JMS[1]. Además, se encarga de la publicación de dichas notificaciones en un monitor de réplica para proveer información al usuario administrativo.

La creación del módulo está dada por la escasez de comunicación que existía en el Replicador Reko entre dos nodos durante la ocurrencia de conflictos y los problemas que esto ocasionaba. En la investigación se describen diversos aspectos como: la problemática inicial que dio origen al desarrollo del módulo, la utilización de métodos científicos investigativos y una comparación con otros sistemas actuales con características o funcionalidades similares a las presentes en Reko. Además, empleando la metodología de desarrollo OpenUP se describe el proceso de diseño e implementación del módulo, el cual fue creado con herramientas Open Source y librerías de clases con licencias gratuitas.

Como resultado se obtuvo una solución que garantiza la comunicación entre dos nodos en conflicto para mantener la consistencia de la información en un Sistema de Bases de Datos Distribuidos.

Palabras claves: Conflictos de Réplica, Réplica de Datos, Sistema de Bases de Datos Distribuidos

[1] Servicio de Mensajería de Java, del inglés Java Message Service

ABSTRACT

One of the fundamental processes of data replication on Distributed Database Systems is the detection and solution of replication conflicts. This thesis details the main characteristics of the Conflict Notifications' Module on Reko Replicator, a replication system for Distributed Database Systems which belongs to the University of Informatics' Sciences. The operation of the developed module focuses on the delivery and reception of conflict notifications through an ActiveMQ messaging server using the JMS protocol. In addition, the module is concerned with the publication of said notifications on a replication monitor to provide information to the administrating user.

The necessity of this module is given by the lack of communication that existed on the Reko Replicator between two nodes upon conflict occurrences and the issues caused by this situation. Along this research several features are described regarding the investigation, among which can be found: the description of the initial problematic that gave origin to the development of the module, the utilization of diverse scientific investigation methods and a comparison with other current systems with similar functionalities or characteristics to those present in Reko. Moreover, by using OpenUP as the development methodology the design and implementation process of the module is described, as well as its creation with Open Source tools and class libraries with free licenses.

The final result guarantees the communication between two conflicting nodes in order to maintain the information consistency on a Distributed Database System.

Keywords: Distributed Database System, Data Replication, Replication Conflicts

Contenido

INTRODUCCIÓN

Con el auge de las tecnologías en el mundo actual se han desarrollado numerosas aplicaciones desplegadas sobre diversas plataformas, orientadas a gestionar información proveniente de los usuarios para brindar servicios variados. Esta información es necesario procesarla para su posterior uso ya sea como estadística, estudios demográficos o simplemente para configurar preferencias personales de un usuario específico, por lo cual es necesario almacenar grandes cantidades de datos. Una de las formas utilizadas para hacer esto es a través de un Sistema de Bases de Datos Distribuido donde se ubica la información en bases de datos en diferentes puntos geográficos de manera que el usuario pueda acceder a la información desde cualquier parte como si de datos locales se tratasen.

En un sistema distribuido, cada base de datos se representa como un nodo[2] y debido a que almacenan la información en múltiples computadoras, se puede mejorar el rendimiento en las estaciones de trabajo al permitir que las transacciones sean procesadas en varias máquinas, en vez de estar limitadas a solo una (1).

Este tipo de sistema ofrece ventajas como la modularidad, ya que es posible añadir o eliminar bases de datos sin afectar los módulos restantes. Proveen autonomía y continuidad de las operaciones aún si un nodo está fuera de línea ya que los restantes se encargan de suplir las funcionalidades del nodo fuera de servicio. La protección física de los datos se garantiza, en la mayoría de los casos, ante desastres naturales debido a que la información se encuentra distribuida en diferentes ubicaciones. También se incrementa la disponibilidad de la información y se facilita la expansión del sistema.

Existen dos procesos que garantizan que los Sistemas de Bases de Datos Distribuidos mantengan su información actualizada: la replicación y la duplicación.

La duplicación, teniendo menor complejidad, básicamente se encarga de determinar una base de datos maestra y después duplicar la misma. Este proceso suele configurarse para que se realice en un momento específico, durante horas de la noche. En el proceso

[2] Un nodo en un Sistema de Base de Datos Distribuidas puede ser un cliente, servidor o ambos en dependencia de la situación

de duplicación los usuarios solamente pueden cambiar los datos existentes en la base de datos maestra.

La replicación, por otro lado, utiliza herramientas de software especializadas que detectan cambios en el sistema distribuido. Una vez que los cambios se identifican, el proceso de replicación se encarga de que todas las bases de datos contengan la misma información. La replicación puede ser un proceso complicado en dependencia del tamaño y de la cantidad de bases de datos además de que consume tiempo y recursos computacionales. (1)

En la Universidad de las Ciencias Informáticas (UCI), específicamente en el Centro de Consultoría y Desarrollo de Arquitecturas Empresariales (CDAE) se cuenta con una solución para el proceso de réplica en Sistemas Distribuidos de Datos llamado Reko. La misma cubre las principales necesidades de réplica en sistemas de bases de datos relacionales incluyendo transferencia, protección y recuperación de la información, la distribución y replicación entre los gestores más utilizados como PostgreSQL, SQL Server, MySQL y Oracle.

Reko surge a raíz de la necesidad de la distribución de datos presentada por el Sistema de Gestión Penitenciario Venezolano (SIGEP) en el año 2007 donde se usa actualmente. Se ha adoptado además como solución de réplica en el Sistema de Gestión de Hospitales Alas-His, DATAFAR de las Fuerzas Armadas Revolucionarias de la República de Cuba y el proyecto de Informatización de Tribunales, el sistema de Fiscalía. Reko se presenta como un software robusto que ha sido probado en numerosos escenarios obteniendo resultados satisfactorios, además debido a que está construido sobre arquitectura en tecnologías libres constituye una solución para la sustitución de importaciones.

Durante el proceso de réplica en Reko pueden suceder numerosos acontecimientos inesperados que deben ser tratados de lo contrario la información no será homogénea en todo el sistema. Entre ellos se encuentran los conflictos de réplica de datos, comunes en los Sistemas de Bases de Datos Distribuidos y que pueden ocurrir por diversas razones como errores durante la introducción de la información, errores provocados por la existencia de diferentes estructuras entre las bases de datos u otras causas que surgen de la desigualdad de la información existente en las bases de datos.

Partiendo de los conflictos que puede generar el proceso de réplica entra en vigor el método utilizado para su resolución, el cual se debe encargar de eliminar las inconsistencias y lograr que se continúe fluidamente con la replicación de los datos. Mientras ocurre la resolución de conflictos es necesario tener comunicación todo el tiempo con los procesos relacionados a la réplica que suceden simultáneamente para que no se persista información incorrecta innecesariamente. Se debe informar también al administrador del sistema de la ocurrencia del conflicto, de su método de solución, del completamiento de la solución y del estado del proceso de réplica, así como de cualquier otro inconveniente que se presente.

Reko utiliza un servidor de mensajería JMS para enviar y recibir los datos de réplica de los diferentes nodos a los que se esté registrado. Los datos a enviar se determinan por el mecanismo de captura, el cual realiza esta operación mediante el uso de disparadores[3] y tablas espejos auto generadas por Reko en la base de datos para detectar cambios en la información y replicarla. Una vez capturado el cambio se envía como una transacción, o grupo replicable como se le denomina en Reko, la cual se encapsula en un mensaje. Estas acciones se muestran en el Monitor de Réplica en conjunto con otra información adicional como por ejemplo nodo destino, tipo de cambios sobre la base de datos (INSERT, UPDATE, DELETE) y datos de la tupla afectada. Este mensaje se captura en el nodo destino y se procede a persistir su contenido en la base de datos. Cada acción replicable contenida en el grupo replicable se procesa una a continuación de la otra.

En relación a la detección y resolución de conflictos, Reko posee un sistema de notificación encargado de mostrar información referente a los conflictos. Este sistema tiene deficiencias en cuanto a la propagación de las notificaciones por el entorno de réplica. Ante la detección de un conflicto en un nodo que recibe datos, la información pertinente al mismo así como el aviso de su ocurrencia, no se muestra en el nodo que envía dichos datos. De igual manera no se informa de la solución a aplicar, o los cambios realizados por la misma. Esta falta de comunicación con el nodo remoto previene que se pueda presentar una solución automatizada al conflicto la cual involucre enviar datos de regreso al nodo que replica para modificar los valores en la tabla en que se originaron.

[3] Traducción del inglés Triggers

Partiendo de varios elementos emitidos anteriormente se define la **situación problemática** de la siguiente manera:

Existe una necesidad de mejor fluidez en la comunicación entre dos nodos durante la detección y resolución de un conflicto de réplica. Esto trae como problemas la reducción de la transparencia en el proceso de resolución. Además se desconoce en el nodo que origina la réplica de la ocurrencia del conflicto y de cuáles fueron los objetos o tuplas que lo provocaron. Una vez encontrada la solución a aplicar, se desconocen sus características y su estado en ambos nodos, por lo que aun cuando el problema ha sido tratado apropiadamente, no se advierten los cambios que este puede haber ocasionado. La falta de claridad en la comunicación en ambos sentidos puede traer consigo la reincidencia en contrariedades similares durante el proceso de réplica, suponiendo que su origen es debido a errores humanos. En adición a esto, se afecta la monitorización de la réplica enfocada al tratamiento de los conflictos, y se entorpece el proceso de administración de un nodo por parte de un usuario por la falta de información.

Tomando como punto de partida lo antes expuesto, se puede formular el **Problema de Investigación** en: ¿Cómo garantizar la comunicación entre dos nodos en conflicto en Reko para mantener la consistencia de la información en un Sistema de Bases de Datos Distribuidos?

La investigación se desarrolla en torno al **objeto de estudio**: La resolución de conflictos durante la réplica de datos.

Enmarcado en el **campo de acción**: El proceso de notificación en la ocurrencia de conflictos de réplica de datos.

Teniendo como **objetivo general de la investigación:** Desarrollar un módulo para gestionar las notificaciones de conflictos en el replicador Reko para mantener la consistencia de la información en un Sistema de Bases de Datos Distribuidos.

El cual se puede desglosar en los siguientes **objetivos específicos**:

1. Realizar un estudio teórico sobre las notificaciones de los conflictos de réplica de datos en Sistemas Distribuidos.

2. Diseñar un módulo para la gestión automática de las notificaciones de los conflictos de réplica en el replicador Reko.
3. Implementar un módulo para gestionar las notificaciones de conflictos de réplica en el replicador Reko.

Para dar cumplimiento a los objetivos enunciados se plantean las siguientes **tareas de investigación**:

1. Realización del marco conceptual para esclarecer los principales conceptos utilizados a lo largo de la investigación.
2. Selección de metodología, tecnologías, herramientas de software a utilizar y lenguajes de programación para el desarrollo del sistema informático con calidad
3. Diseño del módulo para gestionar automáticamente las notificaciones de los conflictos de réplica en Reko
4. Implementación del módulo para gestionar las notificaciones de conflictos de réplica en el replicador Reko.
5. Validación del módulo para gestionar las notificaciones de conflictos de réplica en el replicador Reko, mediante la realización de pruebas de caja blanca y caja negra.

Esta investigación se apoya en los métodos científicos de investigación teóricos y empíricos para dar cumplimiento a las tareas planteadas anteriormente.

Métodos Teóricos:

- **Analítico – Sintético**: Se emplea en el momento de buscar la esencia del tema en cuestión, también permite realizar análisis de teorías, de documentos, entre otros; realizando una síntesis de la misma posibilitando así encontrar los elementos más importantes que se relacionan con el proceso de notificación en conflictos de réplica
- **Modelación:** Utilizado para representar cómo ocurren los procesos que se desean automatizar, además para elaborar los diferentes modelos definidos en la metodología escogida y que sirven de guía durante el desarrollo de la solución.

Métodos Empíricos:

- **Observación:** Utilizado para detectar las necesidades existentes y posibles mejoras que se le pueden añadir al sistema en desarrollo, con el propósito de obtener una aplicación mejor estructurada.
- **La consulta de información en todo tipo de fuentes:** Utilizado para llevar a cabo la elaboración del marco teórico de la investigación, así como para abordar sobre las herramientas, lenguajes y metodología a utilizar.

A partir de esta investigación se espera como posible resultado la creación de un módulo de notificación entre nodos durante la ocurrencia de conflictos que le permita al software Reko tener un control más elaborado de la situación problemática presentada para así darle solución a estos inconvenientes garantizando la mayor consistencia de los datos posible en el proceso de réplica.

El documento está estructurado de la forma siguiente:

En el **Capítulo 1 "Fundamentación Teórica"** se desarrolla el marco teórico de la investigación. Se describen los conceptos principales relacionados con el dominio del problema. Se realiza un estudio del arte de los replicadores de datos y sus diferentes métodos de solución de conflictos y más estrechamente cómo los mismos envían notificaciones a los usuarios o administradores. La descripción de Reko, la metodología de desarrollo de software y la definición de las herramientas a utilizar durante el proceso de desarrollo también son elementos referidos en este capítulo.

En el **Capítulo 2 "Propuesta de Solución"** se incluyen las características, descripción, diseño del módulo y análisis de la solución que se brinda para darle respuesta a la problemática planteada. Se definen los requisitos funcionales y no funcionales. Se realiza el modelo de casos de uso y además se describen los artefactos generados a través del proceso de desarrollo del sistema.

El **Capítulo 3 “Implementación y Pruebas”** contiene parte del código fuente y la descripción del flujo de implementación. Los mismos se usan para mostrar el estilo de codificación utilizado y describir la operación de las principales funcionalidades. Además se realiza una descripción de las pruebas que le serán aplicadas al sistema para demostrar la robustez del mismo.

CAPITULO 1. FUNDAMENTACIÓN TEÓRICA

En el presente capítulo se abordarán los conceptos que sirven como base a la investigación sustentados en la realización del marco teórico. Se hará un estudio acerca de las bases de datos, los sistemas de bases de datos distribuidos, los sistemas de réplica, conflictos de réplica, notificaciones y sistemas homólogos a Reko. Esto permite orientar la investigación conceptualmente basándose en teorías y enfoques realizados por expertos utilizando los medios bibliográficos disponibles.

1.1 Marco Conceptual

1.1.1 Base de Datos

Existen diversas definiciones de Base de Datos y de acuerdo a algunos autores son:

Una base de datos es un conjunto de datos persistentes que es utilizado por los sistemas de aplicación de alguna empresa dada. - C.J.Date[4] (1).

"…una serie de datos organizados y relacionados entre sí, los cuales son recolectados y explotados por los sistemas de información de una empresa o negocio en particular" - Damián Pérez [5] (2).

Una base de datos es una aplicación que administra datos y permite rápido almacenamiento y recuperación de los mismos. Existen diversos tipos de bases de datos pero los más populares son las bases de datos relacionales, en estas los datos se almacenan en tablas en donde cada fila contiene el mismo tipo de información. – David Bolton[6] (3)

[4] C.J. Date, Autor, conferencista, investigador y consultor independiente, especializado en la tecnología de bases de datos.

[5] Damian Pérez Valdés, Webmaster, Administrador de Sistemas, con experiencia en desarrollo web y de aplicaciones.

[6] David Bolton, Bachelor of Science (Licenciado en Ciencias) y desarrollador de software en Londres, Inglaterra

Una base de datos es un sistema informático a modo de almacén. En este almacén se guardan grandes volúmenes de información. – Manuel Sierra[7] (4).

El autor, de acuerdo a lo citado anteriormente, coincide en que una Base de Datos se caracteriza por los siguientes aspectos fundamentales:

- Administra datos y permite rápido almacenamiento y recuperación de los mismos.
- Se guardan grandes volúmenes de información.
- Una serie de datos organizados y relacionados entre sí.
- Explotados por los sistemas de información de una empresa o negocio en particular.

1.1.2 Sistemas de Bases de Datos Distribuidas (SBDD)

El autor de la tesis coincide con el concepto definido por M. Tamer Özsu y Patrick Valduriez en su libro:

"Un SBDD es una colección de bases de datos múltiples e interrelacionadas lógicamente distribuidas en una red de computadoras las cuales tienen la capacidad de procesamiento autónomo lo cual indica que puede realizar operaciones locales o distribuidas". (5)

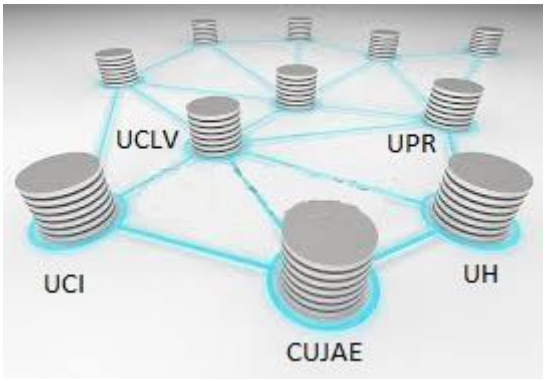


Figura 1. Interconexión en un SBDD

[7] Manuel Sierra, Analista y programador informático con experiencia en la tecnología JAVA, PHP y C#.

En un SBDD, las Bases de Datos se distribuyen en forma de nodos como se muestra en la Figura 1, pudiendo estos ser agregados fácilmente y existiendo además autonomía e independencia entre ellos. Las probabilidades de que un nodo afecte el funcionamiento del sistema son bajas.

Ventajas de los SBDD:

- Reflejan una estructura organizacional
- Autonomía local y habilidad de distribución mejorada
- Mejor confiabilidad
- Mejor rendimiento
- Crecimiento Modular
- Mayor seguridad
- El usuario no sabe que los datos están en diferentes sitios ya que se le presentan como una sola base de datos como si de datos locales se tratase

1.1.3 Replicación

La replicación es un conjunto de tecnologías para copiar y distribuir datos y objetos de una base de datos desde una ubicación hacia otra y posteriormente sincronizar entre ellas para mantener la consistencia. Al usar la replicación, se puede distribuir información hacia diferentes ubicaciones y a usuarios remotos o móviles a través de redes LAN o WAN, conexiones por medio de módems, inalámbricos e Internet. (6)

Replicación es el proceso de copiar y mantener objetos de una base de datos, tales como tablas, en múltiples bases de datos que conforman un sistema distribuido. Los cambios aplicados en un sitio son capturados y almacenados localmente antes de ser enviados y aplicados en cada una de las ubicaciones remotas. - Randy Urbano[8] (7)

La replicación es el proceso de compartir información entre bases de datos para asegurar que el contenido es consistente entre los sistemas. Se utiliza normalmente para

[8] Randy Urbano, Escritor técnico de Oracle.

incrementar el número de servidores de bases de datos disponibles para los clientes, reduciendo así la carga de trabajo en cada uno. - Mike Chapple[9] (8).

El autor asume el concepto definido por Randy Urbano siendo este adecuado para lo que se necesita, además de que se encuentra actualizado y se ajusta al propósito de la investigación.

La replicación ofrece diversas ventajas. Entre las más comunes se encuentran:

Disponibilidad: los usuarios acceden a los datos de los sitios que usualmente se encuentran más cercanos a ellos geográficamente. Si la copia local de los datos replicados no se encuentra disponible, los clientes aún pueden acceder a las copias remotas de la información.

Rendimiento: la replicación provee acceso rápido a datos compartidos ya que balancea su almacenamiento en múltiples sitios. Algunos usuarios pueden acceder un servidor, mientras que otros usuarios acceden a servidores diferentes, de esta forma reduciendo la carga de trabajo en la red.

Paralelismo incrementado: los sitios de réplica que contienen una información específica pueden procesar las consultas relacionadas a esa información en paralelo. Esto conlleva a que el tiempo de ejecución de las consultas sea más rápido.

Respaldo ante fallos: la reparación de un servidor después de un fallo se puede simplificar al copiar los estados de la réplica en vez de reconstruir el servidor a partir de registros

Los procesos de réplica pueden ocurrir en diferentes entornos:

- **Entornos homogéneos:** enmarca los procesos de réplica de información donde los diferentes ambientes usan el mismo gestor de base de datos sobre la misma plataforma
- **Entornos heterogéneos:** se entiende por réplica heterogénea aquella que es implementada sobre diferentes gestores de bases de datos, por ejemplo servidores de bases de datos funcionando con SQL Server y otros con Oracle.

[9] Mike Chapple, Licenciado en Ciencias de la Computación, Máster en Ciencias y Doctor en Ciencias

Para capturar y aplicar los cambios a replicar existen diferentes formas como por ejemplo:

- **Basada en triggers**[10]: Se crean una serie de triggers en la bases de datos que permiten capturar las operaciones de inserción, actualización y eliminación realizadas sobre las tablas a replicar.
- **Basada en logs**[11]: Se sostiene en la lectura de logs de cambios que proporcionan algunos gestores como Oracle

1.1.4 Conflictos de Réplica

Los conflictos de réplica pueden ocurrir en un ambiente de replicación que permite actualizaciones concurrentes de los mismos datos en ubicaciones diferentes. Un ejemplo de esto es que dos transacciones originadas de diferentes lugares actualizan la misma fila aproximadamente al mismo tiempo. (7)

La primera opción al desarrollar un ambiente de replicación debe ser la de diseñarlo de manera que se eviten los conflictos de réplica. Al usar varias técnicas, la mayoría de los sistemas pueden evitar los conflictos en un gran porcentaje de los datos que se replican. Aun así, debido a que se requiere que en estos sistemas existan datos que pueden ser actualizados desde múltiples sitios siempre existe la posibilidad de que surjan conflictos de réplica.

Los conflictos se pueden separar en 3 tipos:

- **Conflictos de actualización:** Ocurren cuando dos transacciones se originan de sitios diferentes, mientras que una elimina una fila la otra transacción actualiza. Esto sucede porque en este caso no existe la fila para actualizar, ya que fue previamente borrada.

[10] Triggers o disparador, procedimiento que se ejecuta cuando se cumple una condición establecida al realizar una operación.

[11] Log es un registro de eventos o sucesos a partir de una acción determinada en un tiempo dado.

- **Conflictos de unicidad:** Ocurren cuando la replicación de una fila intenta violar la integridad de la entidad, como por ejemplo la llave primaria (PRIMARY_KEY) o restricción única (UNIQUE_CONSTRAINT).
- **Conflictos de eliminación:** Ocurren cuando se trata de eliminar una tupla que ya no existe.

1.1.5 Notificaciones

Una notificación es el proceso de comunicar un mensaje determinado a un destinatario. Su propósito fundamental es alertar que un evento en particular ha ocurrido. Un ejemplo sencillo de esto es que se configure un sitio web para enviar un correo a un usuario cuando ocurra una actualización del contenido o un algún otro evento similar suceda. Las notificaciones por lo general se propagan mediante el uso de un sistema de notificación, el cual es un conjunto de protocolos y procedimientos que pueden involucrar tanto componentes humanos como computarizados. El propósito de estos sistemas es generar y enviar mensajes oportunos a una o a un grupo de personas. Los sistemas simples de notificaciones utilizan medios de comunicación tales como correos o mensajes de texto. Sistemas más complejos, diseñados para enviar información crítica utilizan otros medios además de estos y pueden incluir elementos humanos para asegurar que cada persona reciba el mensaje.

En lo que concierne a la investigación, las notificaciones en sistemas distribuidos son alertas que lanza el sistema relacionadas con la réplica de datos o algún cambio administrativo en la configuración. Las mismas pueden informar acerca del completamiento exitoso de la sincronización de los datos, de la existencia de nuevos nodos, o de la ocurrencia de conflictos durante la réplica.

1.2 Estudio de sistemas homólogos

Existe una variedad de soluciones de réplica en el mercado de software, algunas incluidas dentro de su propio Sistema Gestor de Bases de Datos y otras como un software independiente e incluso bajo licencias de libre distribución. A continuación se

muestra un estudio realizado acerca de cómo varias herramientas de réplica tratan los conflictos y principalmente como los detectan y los notifican.

1.2.1 Microsoft SQL Server Merge Replication Agent

Merge Replication Agent es una funcionalidad ofrecida por Microsoft SQL Server que permite enviar cambios realizados de un servidor primario, llamado publicador (de su traducción del inglés, publisher), a uno o más servidores secundarios, llamados suscriptores (de su traducción, subscriber). Es uno de los tres modos disponibles en Microsoft SQL Server para distribuir datos en conjunto con Snapshot Replication y Transactional Replication. Merge Replication es el tipo más complejo de replicación debido a que les permite tanto al publicador como al suscriptor hacer cambios en la base de datos independientemente. Es comúnmente utilizado por usuarios móviles que no pueden estar constantemente conectados con el publicador pero que necesitan llevar una copia de la base de datos a la cual le hacen cambios.

Al usar Merge Replication cualquier cambio sobre un dato se califica como un conflicto o no en dependencia del tipo de detección de conflictos que se defina sobre un artículo.

Si se define detección a nivel de columna, se considera un conflicto los cambios que son hechos sobre la misma columna en la misma fila en más de un nodo de replicación

Si se define detección a nivel de fila, se considera un conflicto si se realizan cambios sobre cualquier columna en la misma fila en más de un nodo de replicación (las columnas afectadas en la fila correspondiente no necesariamente son las mismas)

Si se selecciona detección a nivel de registro, se considera un conflicto si se hacen cambios en cualquier fila al mismo registro lógico en más de un nodo de replicación (las columnas afectadas en la fila correspondiente no necesariamente son las mismas)

Cuando ocurre un conflicto se lanza una alerta, Microsoft SQL Merge Replication Agent viene con varias alertas predefinidas las cuales se pueden configurar para responder a varios eventos desde el Monitor de Réplica bajo la ficha Alertas. Además de eso se pueden crear alertas basadas en el contador del Monitor de Rendimiento **SQLServer:Replication Merge Conflicts/sec**. Para esto se escribe un código que le asigna a ese contador una instancia a monitorear y que de esta forma alerte cuando ocurra un conflicto con dicha instancia. Después que un conflicto se detecta, el Merge

Agent lanza el solucionador de conflictos seleccionado y lo utiliza para determinar el ganador del conflicto. La fila ganadora se aplica en el publicador y en el suscriptor, y los datos de la fila que pierde se persisten en una tabla de conflictos.

Los conflictos se pueden ver usando el Visualizador de Conflictos de Réplica (Replication Conflict Viewer) disponible en SQL Server Management Studio. El Visualizador de Conflictos y el Solucionador Interactivo son herramientas similares, pero el Solucionador Interactivo permite resolver conflictos mientras ocurre la sincronización, en cambio el Visualizador está diseñado para mostrar los conflictos después que han sido solucionados. (9)

Microsoft SQL Server Merge Replication Agent aunque presenta un funcionamiento similar al deseado no constituye una solución viable ya que es una herramienta propietaria. Además de eso, ignorando la limitante anterior, no es compatible con el lenguaje de desarrollo de Reko ni con las plataformas de despliegue ya que solo funciona sobre sistemas operativos de Microsoft.

1.2.2 Slony-I

Slony-I es un sistema de replicación de tipo “master to multiple slaves” (maestro a múltiples esclavos) que soporta promoción de esclavos y replicación en cascada. En general es un sistema maestro-esclavo que incluye el tipo de aptitudes necesarias para replicar grandes bases de datos a un número limitado razonable de sistemas esclavos. “Razonable”, en este contexto, está en el orden de docenas de servidores. Si el número de servidores crece más allá de eso, el costo de las comunicaciones crece prohibitivamente y los beneficios incrementales de tener varios servidores decaerán a partir de ese punto.

Slony-I es un sistema diseñado para centros de datos y sitios de respaldo, donde el modo normal de operación es que todos los nodos están disponibles todo el tiempo y además que todos los nodos pueden estar asegurados. Si existen nodos que son propensos a conectarse y desconectarse de la red regularmente, o nodos que no se pueden mantener con seguridad entonces Slony-I no es la solución de réplica ideal para ese ambiente. (10)

Para cada base de datos en el sistema de réplica se crea un nodo daemon (demonio) llamado Slon. Este daemon es el motor de réplica en sí y consiste en un programa híbrido con funcionalidad maestra y esclava.

Todos los cambios en el sistema como añadir nodos, suscriptores, acciones relacionadas con la réplica, mensajes de sincronización y conflictos se comunican a través del sistema como eventos y son manejados por el daemon. Un evento se genera al insertar la información pertinente al mismo en una tabla y notificándole a todos los oyentes (listeners).

Cada daemon establece conexiones remotas de bases de datos con los nodos desde donde recibe los eventos. Los daemons usan el mecanismo de PostgreSQL LISTEN/NOTIFY para informarse entre sí de la generación de eventos. (11)

Slony-I, aunque siendo una solución *Open Source*, no presenta un sistema de notificaciones adecuado que se ajuste al problema planteado. Slony-I no es un software diseñado para para soportar grandes cantidades de operaciones entre una variedad de servidores con bases de datos de réplica, sino que su propósito fundamental es más bien realizar respaldos de información por lo tanto no se adecua a las necesidades presentadas.

1.2.3 Oracle Streams

Oracle Streams provee una infraestructura flexible que satisface una variedad de necesidades de distribución de información. El mismo habilita la propagación de los datos, transacciones y eventos en el stream (flujo) de datos dentro de una base de datos, o de una base de datos a otra. La flexibilidad respecto a las soluciones tradicionales para la replicación de datos y colas de mensajes, permite a los clientes seleccionar una única solución para compartir información y desplegar soluciones de información en menos tiempo y por menos costo.

El proceso de replicación de información se inicia con el notificador de cambios, configurado con anterioridad y que es el encargado de alertar cuando ha ocurrido un cambio sobre una tabla marcada para replicación, producto de una transacción de negocio en la base de datos. El proceso de notificación de cambios se realiza utilizando la herramienta Oracle Database Change Notification (ODCN) y se configura haciendo un

análisis transaccional de las operaciones que se realizan sobre la BD de captura para notificar solo las tablas necesarias y así ganar en rendimiento.

En caso de errores durante el proceso de captura, dichos errores son notificados al administrador de réplica en el subsistema de monitoreo, el cual tomará una decisión para darle solución al fallo ocurrido. La solución inmediata a los errores en el proceso de captura es reprocesar la transacción. Para poder aplicar la transacción, ésta debe encontrarse en su estado íntegro y las operaciones se aplican teniendo en cuenta el mismo criterio con que fueron aplicadas originalmente en el servidor que las produjo.

En caso de que, durante el proceso de aplicación, la transacción que se está ejecutando produzca un error originado por alguna inconsistencia de datos, se registra un conflicto en la tabla “TRANSACCION_ERROR”, el cual tendrá que ser resuelto por el administrador de réplica de forma manual. (12)

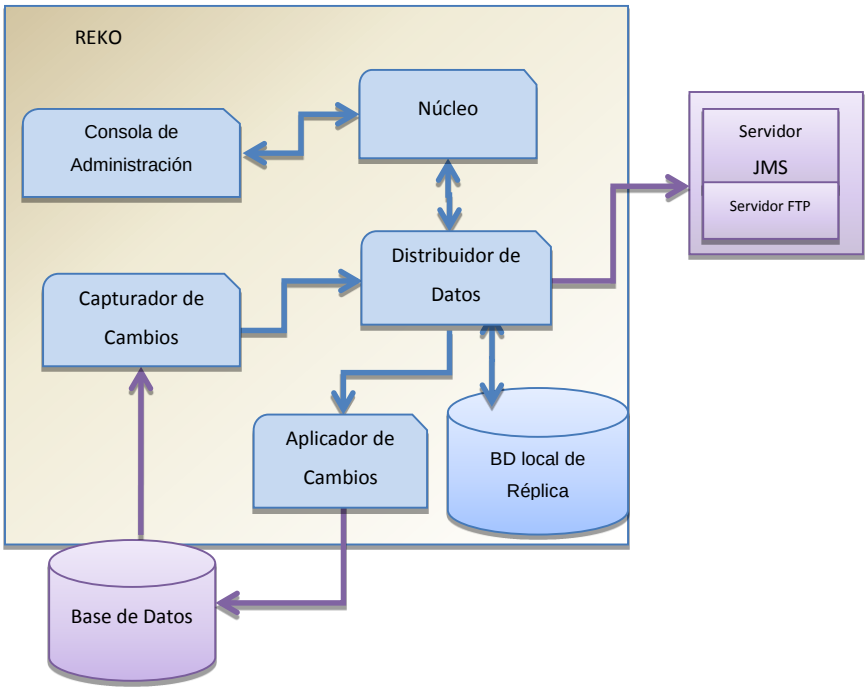
Oracle Streams es una solución privativa no siendo posible acceder a su arquitectura ni a los métodos y algoritmos que puedan dar una solución similar al problema presentado no siendo por tanto una opción a seleccionar.

1.2.4 Replicador Reko

REKO es un software de réplica de datos entre Bases de Datos que pretende cubrir las principales necesidades relacionadas con la distribución de datos entre los gestores más populares. Ha sido diseñado para permitir la réplica y sincronización de datos con conexión y sin conexión, la selección de los datos a replicar y la definición de filtros de replicación. Además, permite replicar datos entre bases de datos con estructuras heterogéneas y entre gestores heterogéneos.

En el siguiente diagrama se muestran los principales módulos que contiene así como una descripción de los mismos.

Figura 2. Módulos de Reko



Núcleo: Maneja las configuraciones fundamentales del software y agrupa las principales funcionalidades de procesamiento información.

Capturador de Cambios: Captura los cambios que se realizan sobre la BD y se los entrega al Distribuidor de Cambios.

Aplicador de Cambios: Ejecuta sobre la BD los cambios que sean replicados hacia la BD.

Distribuidor de Cambios: Determina para dónde debe ser enviado cada cambio realizado en la BD, los envía y se responsabiliza de que cada cambio llegue a su destino.

BD Local de Réplica: Es utilizada para guardar las configuraciones propias de la réplica, las acciones sobre la BD que han dado conflicto al aplicarse y las acciones o transacciones que no han podido llegar a su destino.

Consola Web de Administración: Representa la interfaz del software. Permite realizar las configuraciones principales del software como el registro de nodos, configuración de las tablas a replicar, datos a replicar y el monitoreo del funcionamiento del software.

Servidor JMS (Java Message Service): Servidor de Mensajería bajo la especificación, es utilizado como punto intermedio en la distribución de la información enviada bajo JMS.

Servidor FTP: El servidor FTP es utilizado de forma opcional en el funcionamiento del software. La función principal de utilizar un servidor de FTP es para enviar grandes

archivos de réplica y que se pueda resumir la transmisión de los mismos en caso de problemas en la conexión.

Base de Datos: Es la base de datos que se está replicando, el software de réplica envía los cambios que se realizan sobre ella y aplica los cambios que provienen de otros nodos de réplica.

Reko posee un sistema de notificación que hace uso del servidor de mensajería JMS y que está contenido dentro del componente Distribuidor mencionado anteriormente. Este sistema se encarga de informar acerca del envío y recepción de datos en los diferentes nodos a modo de proveer monitorización de la réplica. También se encarga de mostrar información referente a los conflictos ante su detección.

El sistema de notificación se encuentra limitado en cuanto a la comunicación entre nodos ante la ocurrencia de conflictos. Las notificaciones tienen un ámbito local pues solo se muestran en el nodo en donde ocurre dicho conflicto provocando en adición problemas para mantener la consistencia de los datos por la falta de comunicación con los nodos remotos en el momento de detección y resolución de los conflictos.

Por los aspectos analizados se determina que el sistema existente no suple las necesidades ni resuelve los problemas dados por la situación problemática

1.4 Metodologías de desarrollo

1.4.1 OpenUp

El OpenUP es una plataforma de proceso de desarrollo de software anteriormente llamado BUP (Basic Unified Process o Proceso Unificado Básico) que cubre un conjunto amplio de necesidades y toma un acercamiento ágil al desarrollo de software. Es ligero y promueve las buenas prácticas, haciéndolo un proceso pequeño y extensible si es necesario (híbrido, capaz de incluir partes de otros modelos).

Mantiene características esenciales del RUP[12] como el desarrollo iterativo, desarrollo conducido por casos de usos y escenarios, administración de riesgos y un acercamiento céntrico a la arquitectura. Contiene fundamentalmente un conjunto de simplificaciones de roles, actividades, artefactos y guías.

[12] Proceso Unificado de Desarrollo, del inglés Racional Unified Process

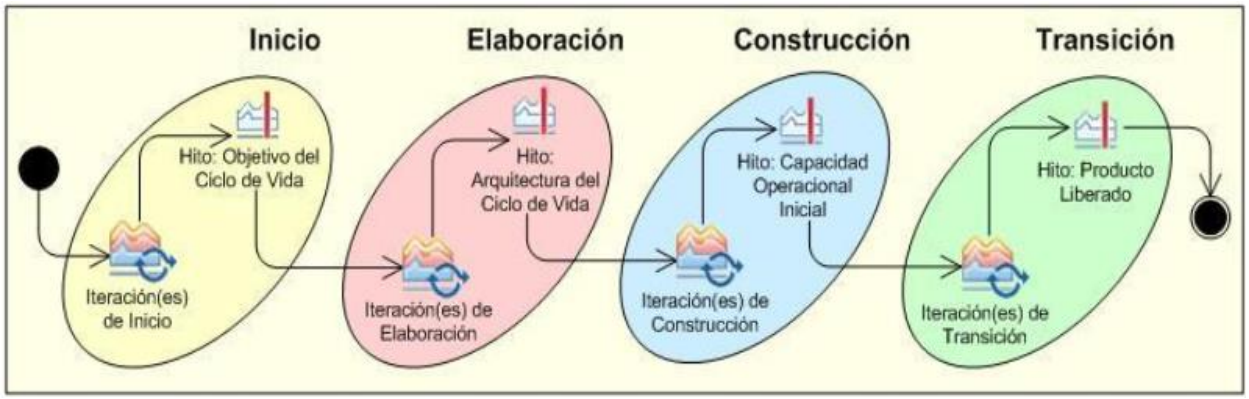


Figura 3. Ciclo de vida del desarrollo en OpenUP

En esta metodología solo el contenido fundamental se incluye, por lo tanto no provee guía en algunos temas que los proyectos enfrentan como por ejemplo equipos de trabajo de gran tamaño, situaciones contractuales, aplicaciones destinadas a procesos críticos. Se puede usar como una base sobre la cual el contenido se puede añadir o ajustar en la medida que se necesite. La mayoría de las prácticas ágiles se idean para que los equipos de trabajo se comuniquen entre sí, proporcionando un entendimiento compartido del proyecto. Los métodos ágiles han atraído la atención debido a la importancia que tiene coordinar el entendimiento y beneficiar a los stakeholders[13] sobre entregas improductivas y la formalidad. OpenUP tiene las características esenciales de un proceso unificado que aplica acercamientos iterativos e incrementales dentro de un ciclo de vida estructurado.

OpenUP está dirigido por los siguientes principios:

- Colaborar para alinear los intereses y compartir el entendimiento
- Equilibrar las prioridades para maximizar el valor de los stakeholders
- Enfocarse en la arquitectura temprana para minimizar los riesgos y organizar el desarrollo
- Evolucionar para continuamente obtener retroalimentación y mejorar

[13] Stakeholder es un término inglés utilizado por primera vez por R. E. Freeman en su obra: “Strategic Management: A Stakeholder Approach”, (Pitman, 1984) para referirse a «quienes pueden afectar o son afectados por las actividades de una empresa»

OpenUP disminuye los riesgos y puede ser utilizarlo tanto en proyectos pequeños como en proyectos grandes y si es manejado con cuidado y con profesionalismo se puede desarrollar un software de gran calidad a pesar de que se diseñe en poco tiempo y con poca documentación. Es recomendable para equipos pequeños, donde sus miembros se encuentren trabajando en un mismo sitio interactuando cara a cara. Un equipo incluye interesados, desarrolladores, arquitectos, gestor de proyecto, y probadores. El equipo toma sus propias decisiones sobre que se requiere realizar, cuales son las prioridades y cómo abordar los requerimientos y necesidades de los interesados. Junto con la colaboración intensa la presencia de los interesados en el equipo es crítica para el éxito.

Se decide optar por la utilización de OpenUP por restricciones existentes en el diseño, además de que la solución será desplegada sobre la arquitectura de Reko y por lo tanto utilizará la misma metodología de desarrollo definida para el mismo. En adición, esto proporciona una menor curva de aprendizaje ya que el equipo de desarrollo no requiere instruirse en una metodología totalmente nueva que genera artefactos desconocidos.

1.5 Herramientas y lenguajes para el desarrollo

1.5.1 Plataforma de desarrollo

Java Platform, Enterprise Edition o **Java EE**, es una plataforma de programación — parte de la Plataforma Java — para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Permite utilizar arquitecturas de N capas distribuidas y se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones. Java EE tiene varias especificaciones de API[14] y define cómo coordinarlos, además de eso también configura algunas especificaciones únicas para Java EE de sus componentes y varias tecnologías de servicios web. Ello permite al desarrollador crear una aplicación de empresa portable entre plataformas y escalable, y al mismo tiempo integrable con tecnologías anteriores. (13)

[14] **Interfaz de programación de aplicaciones** (**IPA**) o API (del inglés *Application Programming Interface*) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

1.5.2 Lenguaje de Programación

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo ya que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. Además de esto gestiona la memoria automáticamente, posee mecanismos de seguridad incorporados y otras características que lo califican como un lenguaje de programación robusto. (14)

1.5.3 Lenguaje de modelado

Lenguaje Unificado de Modelado (LUM o **UML**, por sus siglas en inglés, *Unified Modeling Language*) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocio, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados. (15)

1.5.4 IDE[15] de desarrollo

Spring Tool Suite es un ambiente de desarrollo basado en Eclipse que está personalizado para desarrollar aplicaciones de Spring. Provee un ambiente listo para implementar, debuguear, ejecutar y desplegar aplicaciones Spring. Soporta aplicaciones diseñadas para usos locales y para uso en servidores virtuales o basados en nubes. Está libremente disponible para su uso como medio de desarrollo y con acceso a sus operaciones internas de negocio, completamente de código abierto y bajo los términos de la Licencia Pública de Eclipse. (16)

[15] Entorno de Desarrollo Integrado, del inglés, Integrated Development Enviroment

1.5.5 Herramientas

Apache ActiveMQ es un bróker de mensajería[16] de código abierto (bajo licencia Apache 2.0) que implementa plenamente la especificación de Java Message Service 1.1 (JMS)[17]. Ofrece "Características empresariales" tales como clustering, múltiples almacenes para mensajes, así como la capacidad de emplear cualquier administrador de base de datos como proveedor de persistencia JMS. Aparte de poder usarse en Java, ActiveMQ también puede emplearse en .NET, C/C++ o Delphi o desde lenguajes de script como Perl, Python, PHP y Ruby a través de diversos "clientes cross-language" además de conectarse a numerosos protocolos y plataformas. Entre estos últimos se incluyen los protocolos estándar a nivel de cable, además de un protocolo propio llamado OpenWire. (17)

Apache Tomcat es una implementación de código abierto de las tecnologías Java Servlet y JavaServer Pages. Es desarrollado bajo un ambiente abierto y participativo, y liberado bajo la Licencia de Apache versión 2. (18)

1.5.6 Frameworks

Spring MVC: El paradigma Modelo/Vista/Controlador (MVC) es comúnmente un acercamiento aceptado para construir aplicaciones web de manera tal que la interfaz de usuario está separada de la lógica de la aplicación. Aunque Spring[18] se integra con varios Frameworks MVC populares, también viene con uno propio y muy capaz que promueve

---

[16] Del inglés **message broker** - es un programa intermediario que traduce los mensajes de un sistema desde un lenguaje a otro, a través de un medio de telecomunicaciones.

[17] La API **Java Message Service** (en español *servicio de mensajes Java*), también conocida por sus siglas **JMS**, es la solución creada por Sun Microsystems para el uso de colas de mensajes. Este es un estándar de mensajería que permite a los componentes de aplicaciones basados en la plataforma Java2 crear, enviar, recibir y leer mensajes. También hace posible la comunicación confiable de manera síncrona y asíncrona

[18] Framework de peso ligero, basado en inyección de dependencias y orientación a aspecto, promoviendo el bajo acoplamiento pues los objetos reciben pasivamente sus dependencias en lugar de crearlas o buscar los objetos dependientes por sí mismos

el uso de las técnicas de bajo acoplamiento de Spring en la capa web de una aplicación. (19)

Java Message Service (**JMS**, en español *servicio de mensajes Java*) es la solución creada por Sun Microsystems para el uso de colas de mensajes. Este es un estándar de mensajería que permite a los componentes de aplicaciones basados en la plataforma Java2 crear, enviar, recibir y leer mensajes. También hace posible la comunicación confiable de manera síncrona y asíncrona. El servicio de mensajería instantánea también es conocido como *Middleware*[19] *Orientado a Mensajes* (MOM por sus siglas en inglés) y es una herramienta universalmente reconocida para la construcción de aplicaciones empresariales. (20).

1.6 Consideraciones Parciales

A partir del análisis e investigación de los aspectos abordados en este capítulo se puede concluir que:

- ✓ Se seleccionaron las definiciones de Sistema de Bases de Datos Distribuido de M. Özsu Tamer y Patrick Valduriez, además de Replicación y Conflictos de Réplica de Randy Urbano, en el análisis del Marco Teórico para usarlos como base de la investigación.
- ✓ Para la solución del problema no es viable la utilización total o parcial de herramientas de replicación ya desarrolladas.
- ✓ Se expusieron y justificaron las diferentes herramientas y metodología que se usarán en el desarrollo de la módulo, las cuales guiarán el proceso de desarrollo. Emplear la metodología OpenUP y las mismas herramientas utilizadas para el desarrollo del replicador Reko desde su versión 1.0, disminuyen la curva de aprendizaje y tiempo de desarrollo.
- ✓ El desarrollo de un módulo para la gestión de las notificaciones durante la ocurrencia de conflictos de réplica en Reko, permitirá una mejor comunicación y

[19] Software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, software, redes, hardware y/o sistemas operativos. Éste simplifica el trabajo de los programadores en la compleja tarea de generar las conexiones que son necesarias en los sistemas distribuidos

fluidez durante el proceso de resolución del conflicto, así como también ayudará a mantener la consistencia y homogeneidad de la información

CAPÍTULO 2. PROPUESTA DE SOLUCIÓN

2.1 Descripción del proceso a optimizar

Reko, mediante un mecanismo de captura, detecta los cambios realizados en la base de datos a partir de tablas espejos[20] que se crean por cada entidad de la cual se quieran replicar datos. Dichos cambios se capturan mediante el uso de triggers o disparadores. Cada cierto intervalo de tiempo, Reko encapsula y agrupa los cambios en objetos llamados ReplicableGroup o Grupo Replicable compuestos por un conjunto de operaciones llamadas ReplicableAction o Acción Replicable, y los envía a los nodos donde deben ser aplicados. Una vez que se reciben en el otro extremo estos grupos, se procede a persistirlos en la base de datos, realizando al mismo tiempo la detección y captura de cualquier conflicto de réplica que surja. Sin embargo, ante la ocurrencia de un conflicto la aplicación se encuentra muy limitada. Esto se debe a que posee un solo tipo de solución automática, lo cual reduce el número de posibilidades que la aplicación puede tratar por sí sola. Esto provoca además, que se detenga el proceso de replicación muy a menudo, que es lo que sucede ante la detección de un conflicto. Conjuntamente, la propagación de notificaciones ante este tipo de situaciones es escasa, ya que solo se muestra una notificación en el nodo que captura el conflicto y no describe en su totalidad sus características. Posterior a la solución del conflicto, ya sea por vía manual o automática, no se informa a los nodos involucrados del estado del proceso de réplica ni de los mecanismos u operaciones que necesitaron realizarse para solucionar los problemas que existían.

Esta limitante afecta la monitorización la réplica enfocada a la ocurrencia y tratamiento de los conflictos. Además, la eliminación de este problema apoyaría en cierta medida el proceso de administración de un nodo por parte de un usuario, ya que el mismo no desconocería los detalles de los inconvenientes surgidos durante el proceso de sincronización o replicación.

[20] Es una tabla de control que se encuentra en la base de datos. Almacena el tipo de acción, nombre del esquema, tabla y columna a replicar, así como el usuario que ejecuta la acción.

2.2 Modelo de dominio

Los modelos de dominio son representaciones de un dominio de aplicación que puede ser utilizado para una variedad de metas funcionales en soporte a tareas o procesos específicos de la ingeniería de software.

El objetivo del modelo de dominio es comprender y describir las clases más importantes dentro del contexto del sistema, ayudando a utilizar un vocabulario común entre los usuarios, clientes, desarrolladores y demás interesados. También contribuye a la comprensión de los requisitos del sistema que se desprende de este contexto. (21).

Un modelo de dominio preciso puede contribuir como un aspecto esencial para la implementación de la solución dentro de un ciclo de desarrollo de software, ya que los elementos del modelo que caracterizan el dominio del problema pueden servir como puntos clave para la construcción del código.

2.2.1 Diagrama de clases del dominio

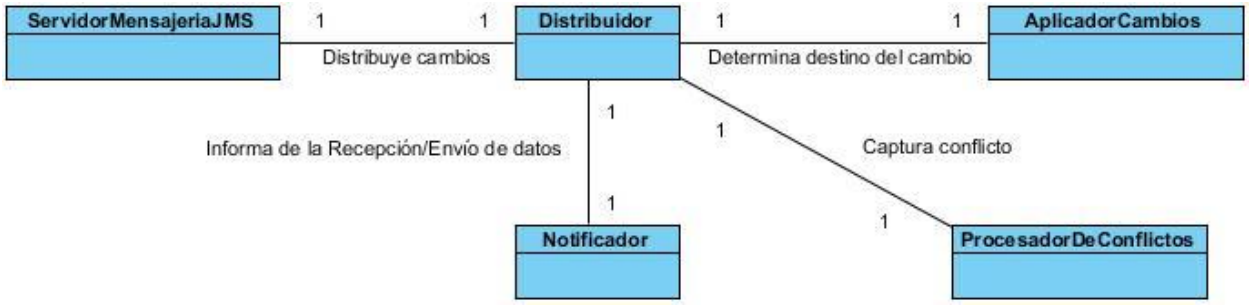


Figura 4. Diagrama de clases del dominio

2.2.2 Descripción de las clases del modelo de dominio

ServidorMensajeriaJMS: concepto que representa al servidor de mensajería que se utiliza para la distribución y recepción de la información.

Distribuidor: concepto que se encarga de repartir los grupos de datos en la aplicación para que los demás elementos realicen su responsabilidad.

AplicadorCambios: concepto que se encarga de ejecutar sobre la base de datos los cambios que se repliquen hacia ella. También detecta y captura los conflictos cuando se intenta aplicar dichos cambios.

Notificador: concepto encargado de publicar determinadas informaciones respecto al funcionamiento de la aplicación.

ProcesadorDeConflictos: concepto que se encarga de darle solución a los conflictos detectados.

2.3 Listado de requerimientos

Requisitos funcionales

Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones específicas. Estos requisitos dependen del tipo de software que se desarrolle, de los posibles usuarios del software y del enfoque general tomado por la organización al redactar requisitos. (22).

RF 1 Generar Notificaciones de Ocurrencia de Conflicto

- RF 1.1 Construir Notificaciones de Ocurrencia de Conflicto
- RF 1.2 Enviar Notificaciones de Ocurrencia de Conflicto

RF 2 Generar Notificaciones de Conflicto Evitado

- RF 2.1 Construir Notificaciones de Conflicto Evitado
- RF 2.2 Enviar Notificaciones de Conflicto Evitado

RF 3 Generar Notificaciones de Conflicto Solucionado

- RF 3.1 Construir Notificaciones de Conflicto Solucionado
- RF 3.2 Enviar Notificaciones de Conflicto Solucionado

RF 4 Generar Notificaciones de Conflicto Solucionado con Datos de Réplica

- RF 4.1 Construir Notificaciones de Conflicto Solucionado con Datos de Réplica
- RF 4.2 Enviar Notificaciones de Conflicto Solucionado con Datos de Réplica

RF 5 Publicar Notificaciones Locales

- RF 5.1 Publicar Notificaciones de Ocurrencia de Conflicto
- RF 5.2 Publicar Notificaciones de Conflicto Solucionado
- RF 5.3 Publicar Notificaciones de Conflicto Evitado
- RF 5.4 Publicar Notificaciones de Conflicto Solucionado con Datos de Réplica

RF 6 Publicar Notificaciones Remotas

- RF 6.1 Publicar Notificaciones de Ocurrencia de Conflicto

RF 6.2 Publicar Notificaciones de Conflicto Solucionado

RF 6.3 Publicar Notificaciones de Conflicto Evitado

RF 6.4 Publicar Notificaciones de Conflicto Solucionado con Datos de Réplica

Requisitos no funcionales

Los requisitos no funcionales, tal y como sugiere el nombre, no están implicados con funcionalidades específicas que brinda el sistema. Se pueden relacionar con propiedades del sistema tales como confiabilidad, tiempo de respuesta, y espacio de almacenamiento. Alternativamente, pueden definir restricciones en el sistema tales como la capacidad de los dispositivos de entrada/salida y las representaciones de datos en las interfaces. (23)

Implementación: Se implementará utilizando la plataforma JEE y las tecnologías asociadas que se emplean en Reko.

Soporte: El diseño del módulo tendrá en cuenta patrones GOF y GRASP para garantizar la escalabilidad del sistema.

Rendimiento: Debe estar concebido para el consumo mínimo de recursos. Los clientes no necesitaran más de 128MB de RAM, lo suficiente para ejecutar un navegador Web.

Fiabilidad: Deber ser capaz de recuperarse ante fallos como: pérdida en la comunicación entre dos nodos, falta del fluido eléctrico, garantizando el envío y recepción de las notificaciones.

Software: se requiere para el funcionamiento del sistema disponer de un servidor JMS, un servidor de base de datos SQL Server y navegador Mozilla 3.0 o superior. Soporta cualquier sistema operativo con la máquina virtual de Java funcionando en una versión 1.6 o superior y no necesita de un ambiente gráfico en el servidor para su funcionamiento.

Hardware:

- Para las estaciones de trabajo: Se requiere tengan tarjeta de red, al menos 128 MB de memoria RAM, al menos 100MB de disco duro y procesador 800 MHz como mínimo.
- Para los servidores: Se requiere tarjeta de red, al menos 512MB de RAM, al menos 100MB de disco duro y. procesador 2.0 GHz como mínimo.

2.4 Propuesta del sistema

Para darle solución a la situación problemática que da surgimiento a la investigación, se realiza una propuesta que operará de la siguiente manera: una vez capturado el conflicto se extraen los datos pertinentes al mismo. Estos datos describen las características del conflicto como su tipo, el código SQL a ejecutar, el nodo origen, el destinatario, la fecha y hora en que se recibió la acción de réplica y la excepción lanzada por el gestor SQL.

La información se encapsula dentro de notificaciones, que pueden ser de diversos tipos y varían en dependencia del escenario presentado. Ejemplo de ello sería que para informar de la ocurrencia de conflictos se genera un tipo de notificación específico, mientras que para informar de la solución del conflicto se genera otro tipo. Es válido mencionar que estas notificaciones no solo informan acerca de la ocurrencia de conflictos o de su solución, sino que pueden además portar datos a aplicar en el nodo que origina la réplica. Estos datos forman parte de la solución automatizada brindada al conflicto en cuestión en el nodo destino. Son necesarios pues la información que permanece en la base de datos receptora, después de aplicado un método de solución, fue transformada para darle solución al conflicto que existía. Esta información, siendo nueva, necesita ser aplicada en la base de datos original a manera de mantener la consistencia de la información en el escenario de réplica.

La notificación creada necesita mostrarse en el monitor de réplica del nodo actual, o sea, el nodo donde ocurrió el conflicto. Para ello, el **`MonitoringManager`**[21] se encargará de construir acciones de monitoreo empleando la información pertinente a los conflictos contenida en las notificaciones que se envían y se reciben. Estas acciones de monitoreo se utilizan para transportar la información a las vistas mediante la publicación de diferentes eventos asociados a los tipos de notificaciones. Los eventos se capturan en tiempo de ejecución y se utilizan para que la información se muestre en la capa de la interfaz visual en el monitor de réplica en vivo. Al mismo tiempo, utilizando la base de

[21] Clase controladora encargada de gestionar la monitorización de la réplica. Para más información referirse a su descripción en la tabla 8.

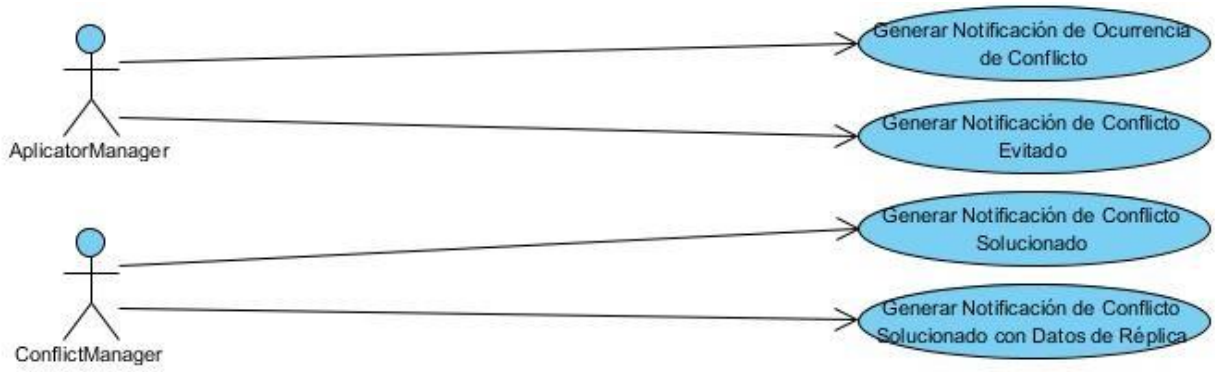
datos local de Berkeley[22], se almacenan las diversas notificaciones existentes para su uso en los historiales. Posterior a la publicación en el nodo actual, se procede al envío de las notificaciones hacia el o los nodos involucrados en el conflicto.

La clase **`NotifyManager`**[23] será la encargada de enviar y recibir todas las notificaciones que arriben a un nodo o surjan de él. Para ello hará uso de los mensajes JMS y del servidor de mensajería mediante el cual se propagan. Una vez recibida una notificación se extraen los datos contenidos en su interior. De manera similar se hace uso de esta información y del conocimiento del tipo de notificación recibido para mostrar en el monitor de réplica del nodo remoto de la existencia de conflictos en el nodo destino que recibió los datos de replicación.

Las funcionalidades de notificación podrán ser accedidas y llamadas en el momento de captura de un conflicto desde el **Aplicador de cambios**, o después de haber sido proveída una solución, por el **Procesador de Conflictos**.

2.5 Modelo de casos de uso del sistema

2.5.1 Diagrama de casos de uso



[22] Berkeley DB es una librería de manejo de base de datos con API para C, C++, Java, Perl, Python, Ruby, Tcl y muchos otros lenguajes.

[23] Clase controladore encargada del envío y recepción de notificaciones. Para más información referirse a su descripción en la tabla 7

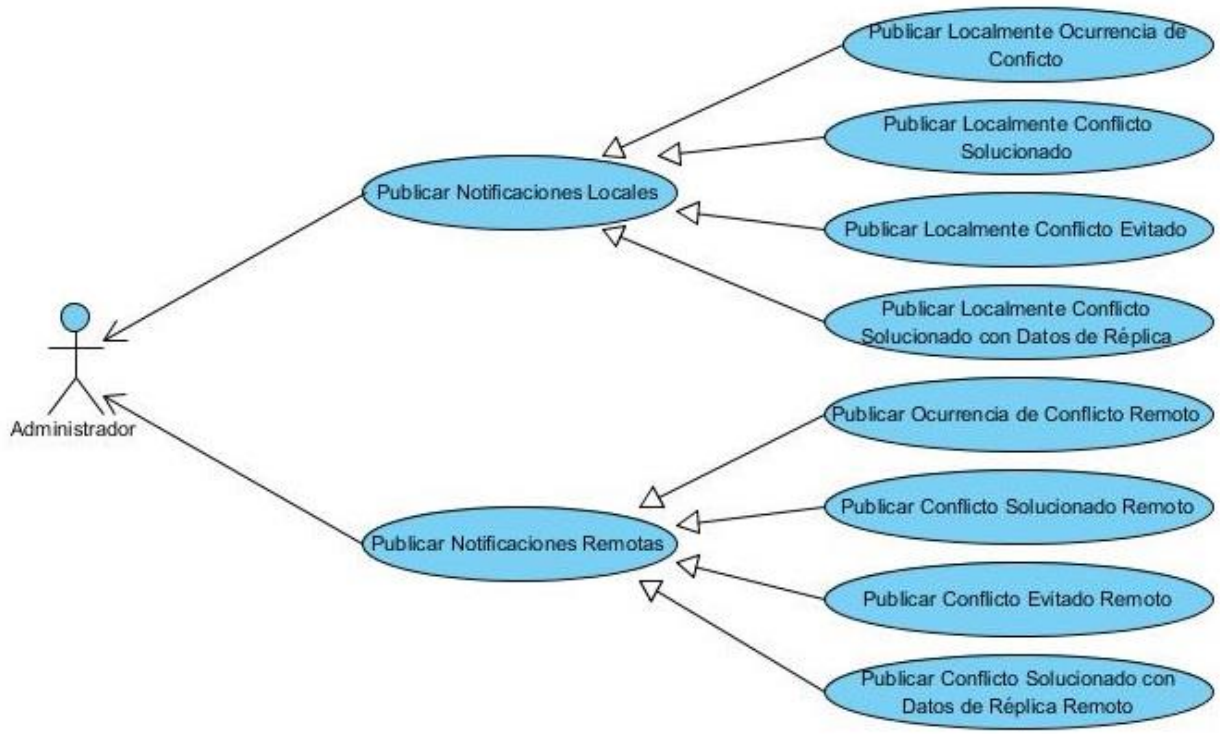


Figura 5. Diagrama de Casos de Uso del Sistema

2.5.2 Descripción de los actores

Tabla 1. Descripción de los actores del sistema

Actores	Descripción
AplicatorManager	Responsable de la aplicación de cambios en la base de datos, además, de la captura y detección de conflictos.
ConflictManager	Responsable de procesar los conflictos y proveer una solución para los mismos
Administrador	Usuario de la aplicación. Es quien se retroalimenta y se informa a partir de las notificaciones en el monitor de réplica.

2.5.3 Descripción de los casos de uso

Tabla 2. Descripción del CU Generar Notificacion de Oucrrencia de Conflicto

Nombre:	CU_GenerarNotificaciónOcurrenciaConflicto
Objetivo:	Construye y envía notificaciones de ocurrencia de conflictos
Actores:	AplicatorManager

Resumen:	El caso de uso se inicia cuando el AplicatorManager detecta un conflicto y envía la información pertinente al mismo ya que se desea informar a los nodos involucrados.
Precondiciones:	Debe estar activo el servidor de mensajería
Referencias:	RF 1.1, RF 1.2
Prioridad:	Crítico
Flujo Normal de Eventos	
Acción del actor	**Respuesta del sistema**
1. El AplicatorManager envía la información pertinente al conflicto.	2. Construye la notificación de tipo "**`ConflictNotification`**". 3. Coloca en la notificación los datos de la acción en conflicto. 4. Coloca en la notificación los datos del nodo donde ocurrió el conflicto. 5. Coloca en la notificación el mensaje de la excepción SQL lanzada por el gestor de la base de datos. 6. Coloca como destino de la notificación, el nodo que originó la acción de réplica. 7. Crea un mensaje de java. 8. Ajusta el destino a partir de la notificación. 9. Encapsula la notificación en el mensaje 10. Envía mensaje por canal de mensajería al destino.
Poscondiciones:	Se ha creado y enviado una notificación de ocurrencia de conflicto utilizando el servidor de mensajería.

Tabla 3. Descripción del CU Publicar Noticaciones Locales

Nombre:	CU_PublicarNotificacionesLocales
Objetivo:	Publica en el monitor de réplica las notificaciones de conflictos ocurridos en el nodo local.
Actores:	Administrador

Resumen:	El caso de uso se inicia cuando el sistema necesita publicar una notificación en el monitor de réplica relacionada con los conflictos, que sirva de información para el administrador.
Precondiciones:	Debe existir una notificación a publicar
Referencias:	RF 5
Prioridad:	Crítico
Flujo Normal de Eventos	
Acción del actor	**Respuesta del sistema**
	1. Crea la acción de monitoreo (MonitoringAction). 2. Asigna a la MonitoringAction los datos de la acción de réplica contenida en la notificación. 3. Asigna a la dirección de la MonitoringAction el valor de “entrada”. 4. Se ejecutan una de las siguientes acciones: i. Si la notificación es de tipo “Ocurrencia de Conflicto” ir al CU_PublicarLocalOcurrenciaConflicto ii. Si la notificación es de tipo “Conflicto Evitado” ir al CU_PublicarLocalConflictoEvitado iii. Si la notificación es de tipo “Conflicto Solucionado” ir al CU_PublicarLocalConflictoSolucionado iv. Si la notificación es de tipo “Conflicto Solucionado con Datos de Réplica” ir al CU_PublicarLocalConflictoSolucionadoDatosReplica 5. Asigna la lista de MonitoringTarget a la MonitoringAction 6. Crea el evento relacionado al tipo de notificación. 7. Encapsula la MonitoringAction en el evento 8. Publica el evento 9. Envía MonitoringAction al MonitoringDao
Poscondiciones:	Se ha publicado una notificación que se mostrará en el monitor de réplica.

Tabla 4. Descripción del CU Publicar Ocurrencia de Conflicto Local

Nombre:	CU_ PublicarOcurrenciaConflictoLocal
Objetivo:	Publica en el monitor de réplica una notificación de ocurrencia de conflicto

Actores:	Administrador
Resumen:	El caso de uso se inicia cuando el sistema necesita publicar una notificación en el monitor de réplica relacionada con los conflictos, que sirva de información para el administrador.
Precondiciones:	Debe existir una notificación a publicar
Referencias:	RF 5.1
Prioridad:	Crítico
Flujo Normal de Eventos	
Acción del actor	**Respuesta del sistema**
	1. Crea el(los) objeto(s) destino(s) (MonitoringTarget). 2. Establece como id de MonitoringTarget el id del nodo a donde se envió la acción de réplica y ocurrió el conflicto 3. Establece como estado de los MonitoringTarget el valor de la constante: *STATUS_CONFLICT.* Regresar a la acción 5 del CU_PublicarNotificacionesLocales
Poscondiciones:	Se ha publicado una notificación conflicto local que se mostrará en el monitor de réplica.

Tabla 5. Descripción del CU Publicar Notificadiones Remotas

Nombre:	CU_PublicarNotificacionesRemotas
Objetivo:	Publica en el monitor de réplica las notificaciones de conflictos ocurridos en nodos remotos.
Actores:	Administrador
Resumen:	El caso de uso se inicia cuando el sistema necesita publicar en el monitor de réplica una notificación que se recibe, relacionada con un conflicto de un nodo remoto, que sirva de información para el administrador.
Precondiciones:	Debe existir una notificación a publicar
Referencias:	
Prioridad:	Crítico
Flujo Normal de Eventos	

Acción del actor	Respuesta del sistema
	1. Se recibe un mensaje 2. Se extrae la notificación del mensaje y se envía al NotifyManager. 3. Se extraen datos de la notificación. 4. Guardar en el log, confirmación de notificación recibida. 5. En dependencia del tipo de notificación se escogen una de las siguientes opciones: i. Si es de tipo “Ocurrencia de conflicto” ir al CU_PublicarOcurrenciaConflictoRemoto ii. Si es de tipo “Conflicto Evitado” ir al CU_PublicarConflictoEvitado iii. Si es de tipo “Conflicto Solucionado” ir al CU_PublicarConflictoSolucionado iv. Si es de tipo “Conflicto Solucionado con Datos de Réplica” ir al CU_PublicarConflictoSolucionadoDatosReplica.
Poscondiciones:	Se ha publicado una notificación que se mostrará en el monitor de réplica.

Tabla 6. Descripción del CU Publicar Ocurrencia de Conflicto Remoto

Nombre:	CU_PublicarOcurrenciaConflictoRemoto
Objetivo:	Publica en el monitor de réplica las notificaciones de conflictos ocurridos en nodos remotos.
Actores:	Administrador
Resumen:	El caso de uso se inicia cuando el sistema necesita publicar en el monitor de réplica una notificación de “ocurrencia de conflicto” que se recibe, relacionada con un conflicto de un nodo remoto, que sirva de información para el administrador.
Precondiciones:	Debe existir una notificación a publicar
Referencias:	
Prioridad:	Crítico
Flujo Normal de Eventos	
Acción del actor	**Respuesta del sistema**

	1. Crea la acción de monitoreo (MonitoringAction). 2. Asigna a la MonitoringAction los datos de la acción de réplica contenida en la notificación. 3. Asigna a la dirección de la MonitoringAction el valor de “salida” 4. Crea el(los) objeto(s) destino(s) (MonitoringTarget). 5. Establece como id de MonitoringTarget el id del nodo a donde se envió la acción de réplica y ocurrió el conflicto 6. Establece como estado de los MonitoringTarget el valor de la constante: *STATUS_REMOTE_CONFLICT.* 7. Asigna la lista de MonitoringTarget a la MonitoringAction 8. Crea el evento relacionado al tipo de notificación. 9. Encapsula la MonitoringAction en el evento 10. Publica el evento. 11. Envía MonitoringAction al MonitoringDao
Poscondiciones:	Se ha publicado una notificación de ocurrencia de conflicto en el nodo remoto que se mostrará en el monitor de réplica.

2.6 Descripción de la arquitectura del software de réplica Reko

Las técnicas metodológicas desarrolladas con el fin de facilitar la programación se engloban dentro de la llamada Arquitectura de Software o Arquitectura lógica. Se refiere a un grupo de abstracciones y patrones que nos brindan un esquema de referencia útil para guiarnos en el desarrollo de software dentro de un sistema informático. Así, los programadores, diseñadores, ingenieros y analistas pueden trabajar bajo una línea común que les posibilite la compatibilidad necesaria para lograr el objetivo deseado.

El estándar 1471-2000 de la IEEE define a la arquitectura de software como: “La organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente, y los principios que orientan su diseño y evolución.” (24).

2.6.1 Estilo arquitectónico

Un estilo arquitectónico se define como: “una lista tipos de componentes que describen los patrones o las interacciones a través de ellos. Un estilo afecta a toda la arquitectura de software y puede combinarse en la propuesta de solución”. (25)

El desarrollo Basado en componentes es un estilo arquitectónico utilizado en aplicaciones que pueden descomponerse en componentes lógicos o funcionales y se comunican a través de interfaces bien definidas que contienen métodos, eventos y propiedades. Este estilo arquitectónico permite alcanzar un mayor nivel de reutilización de software y que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo, además de esto también simplifica el mantenimiento del sistema.

Reko presenta una arquitectura que puede definirse como basada en componentes, debido a que todas sus partes encapsulan un conjunto de comportamientos que pueden ser reemplazados por otros. Los principales componentes presentes en el software son:

Capturador_Cambios: encargado de capturar los cambios que se realizan en la base de datos y entregarlos al distribuidor.

Distribuidor: determina el destino de cada cambio realizado en la base de datos, los envía y se responsabiliza de su llegada.

Aplicador: ejecuta en la base de datos los cambios que son enviados hacia él desde otro nodo de réplica.

Administración: permite realizar las configuraciones principales del software como el registro de nodos, configuración de las tablas a replicar y el monitoreo del funcionamiento.

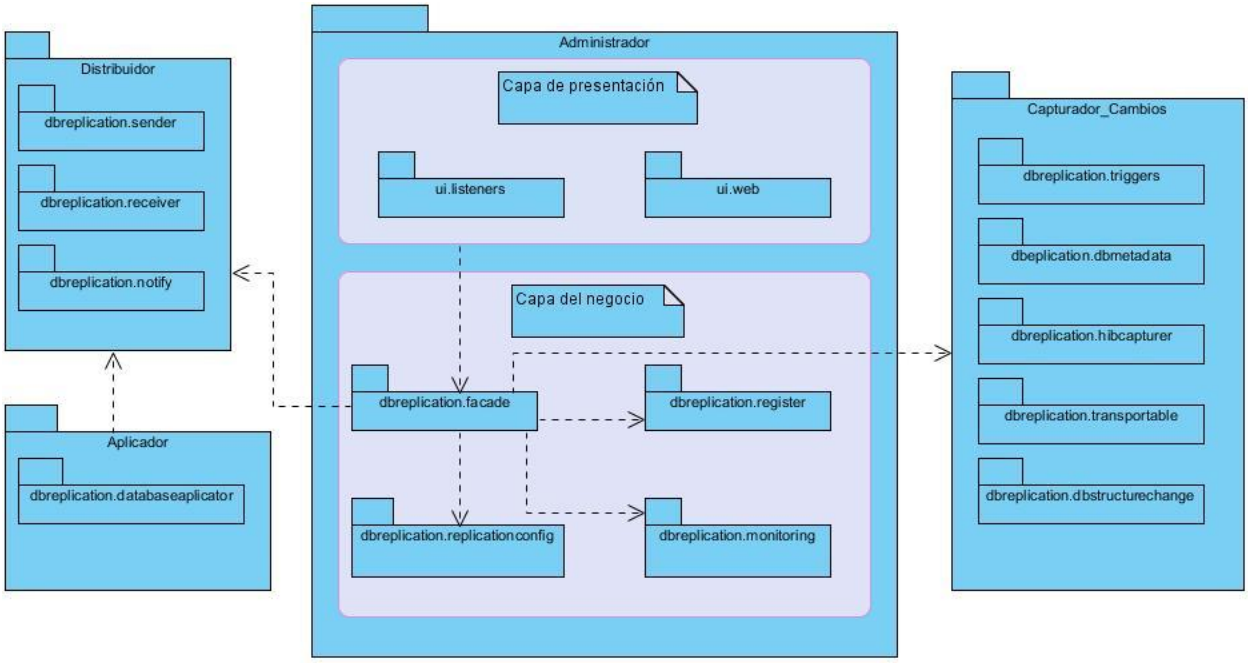


Figura 6. Vista de los principales componentes de Reko

Además de lo antes expuesto, el componente **Administración** responde a un modelo multicapas, donde cada capa tiene funcionalidades y objetivos precisos, así su implementación se encuentra desacoplada de la programación de cualquier otro componente y la comunicación con una capa inferior ocurre a través de interfaces. Además de estar separadas lógica y estructuralmente, las capas se encuentran separadas de manera física.

La **capa de presentación** es la que el usuario ve en su ordenador, es donde se tratan los datos que se van a mostrar. Esta capa se comunica únicamente con la capa de negocio.

La **capa de negocio** es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina además como lógica del negocio porque es aquí donde se establecen todas las reglas que deben cumplirse.

2.6.2 Patrones de diseño

Un patrón de diseño es una solución general reusable de un problema que ocurre en un contexto dado en un software. No es un diseño final que se transforma directamente en

código fuente. Es una descripción o plantilla que determina cómo resolver un problema y que puede ser utilizado en diferentes situaciones. Los patrones de diseño orientados a objetos típicamente muestran las relaciones e interacciones entre clases y objetos, sin especificar las clases que existen en la aplicación final u algún objeto en concreto. Los patrones residen en el dominio y la utilización de módulos e interconexiones entre clases o componentes presentes en un software. En un nivel superior están los patrones arquitecturales que son más grandes en cuanto al alcance y que usualmente describen un comportamiento o filosofía utilizada por un sistema completo. (26)

Patrón GRASP[24]

Consisten en un conjunto de guías para la asignación de responsabilidades a clases y objetos en el diseño orientado a objetos. Los diferentes patrones y principios usados en GRASP son: Controlador, Creador, Experto en Información, Alta Cohesión, Bajo Acoplamiento, Polimorfismo, Variaciones Protegidas y Pura Fabricación. Todos estos patrones responden algún problema de software y casi en todos los casos estos problemas son comunes para casi todos los proyectos de desarrollo de software.

Larman afirma que “la herramienta crítica de diseño para el desarrollo de software es una mente bien educada en principios de diseños. No es UML ni cualquier otra tecnología.” (27) Así, GRASP es realmente un conjunto de herramientas mentales, una ayuda de aprendizaje para guiarnos en el diseño de software orientado a objetos.

Experto: (También experto en información o principio experto) Utilizado para determinar donde delegar la responsabilidad. Entre estas se incluyen métodos, campos, etc. Experto en información conllevará a colocar la responsabilidad en la clase con la mayor cantidad de información necesitada para cumplir dicha responsabilidad. Ejemplo de esto es la clase **`JMSDeliveryReceiverListener`**, la cual cuenta con la información necesaria para cumplir la responsabilidad de recibir las notificaciones.

[24] Patrones Generales de Software de Asignación de Responsabilidades, del inglés General Responsibility Assignment Software Patterns.

Creador: La creación de objetos es una de las actividades más comunes en un sistema orientado a objetos. Simplemente, el patrón creador es responsable de la creación de un objeto de una clase. De manera general, una clase B debería ser responsble de crear instancias de la clase A si uno, o preferiblemente más, de los siguientes aspectos se cumplen:

- Instancias de B contienen o se componen de instancias de A
- Instancias de B persisten instancias de A
- Instancias de B utilizan muy intrínsecamente instancias de A
- Instancias de B tienen la información de inicialización para instancias de A y las pasan en su creación

Ejemplo de patrón creador sería la clase **`MonitoringManager`**, la cual contiene toda la información pertinente a las **`MonitoringAction`** y se encarga de crearlas.

Controlador: Este patrón asigna la responsabilidad de tratar con eventos de sistema en una clase sin interfaz visual que representa el sistema de manera general o un escenario de caso de uso. Ejemplo de este patrón serían las clases **`MonitoringManager`**, **`NotifyManager`** y **`AplicatorManager.`**

Alta Cohesión: Es un patrón evaluativo que intenta mantener los objetos apropiadamente enfocados, administrables y comprensibles. Alta Cohesión significa que las responsabilidades de un elemento dado están fuertemente relacionadas y altamente concentradas. Separar los programas en clases y subsistemas es un ejemplo de actividades que incrementan las propiedades cohesivas de un sistema. Alternativamente, baja cohesión es una situación en la cual un elemento dado tiene muchas responsabilidades no relacionadas. Elementos con baja cohesión son más difíciles de comprender, reutilizar, mantener y cambiar. (27) Como cada clase trabajada en la aplicación tiene un conjunto de funcionalidades directamente relacionadas con la entidad que representan se pueden clasificar como que poseen una alta cohesión.

Bajo Acoplamiento: Se relaciona con el patrón de alta cohesión. Es un patrón evaluativo que dicta como asignar responsabilidades para soportar:

- Pocas dependencias entre clases
- Cambios en una clase tiene poco impacto en las demás.
- Alto potencial de reutilización.

Las clases tienen solamente las dependencias necesarias para realizar sus respectivas funcionalidades, tratando de maximizar la reutilización y el mantenimiento del código.

Polimorfismo: La responsabilidad de definir la variación de comportamiento basado en tipos se asigna a los tipos para los cuales esta variación ocurre. Esto se logra mediante el uso de operaciones polimórficas. Se ve reflejado en el uso de diferentes tipos de notificaciones que realizan diferentes tipos de comportamiento y se agrupan bajo un supertipo en común.

Patrones GOF[25]

Patrón Fachada: Comúnmente utilizado en la programación orientada a objetos. Una fachada es un objeto que provee una interfaz simplificada para un grupo de código mayor, como por ejemplo una librería de clases. Una fachada es capaz de:

- Hacer que una librearía de software sea más fácil de usar, entender y probar, ya que la fachada tiene métodos convenientes para tareas comunes.
- Reducir las dependencias de código externo con respecto al funcionamiento interno de una librería, ya que la mayor parte del código utiliza la fachada, permitiendo así flexibilidad en el desarrollo del sistema.
- Encapsular una colección de API[26]s pobremente diseñadas dentro de una sola API con un mejor diseño.

En Reko se posee la **`NotificationFacade`** en el paquete notify que ofrece un punto de acceso a las funcionalidades de notificación disponibles en el mencionado paquete.

2.7 Modelo de Diseño

El modelo de diseño describe la realización de los casos de uso y sirve como una abstracción del modelo de implementación y el código fuente. Es usado como una entrada inicial en las actividades de implementación y prueba.

[25] Banda de los Cuatro, del inglés Gang-Of-Four. Nombre con el que se conoce comúnmente a los autores del libro.

[26] Interfaz de Programación de una Aplicación, del inglés Application Programming Interface,

2.7.1 Diagramas de clases del diseño

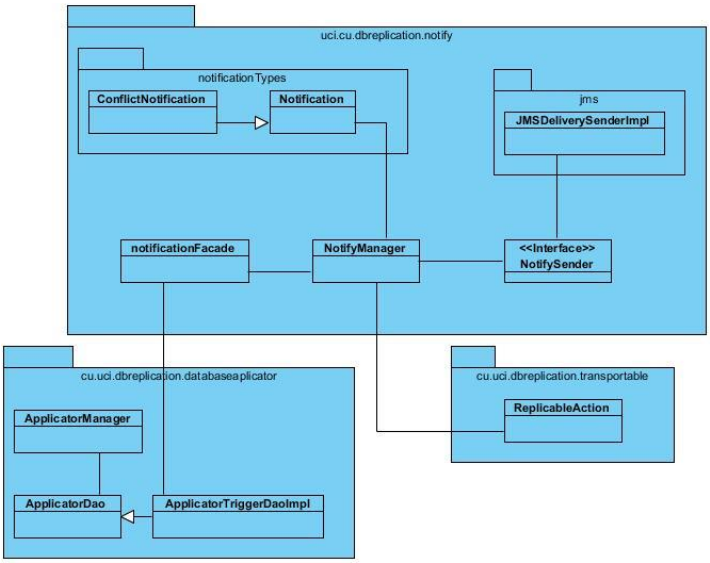


Figura 7. DC para el CU Generar Notificacion Ocurrencia Conflicto

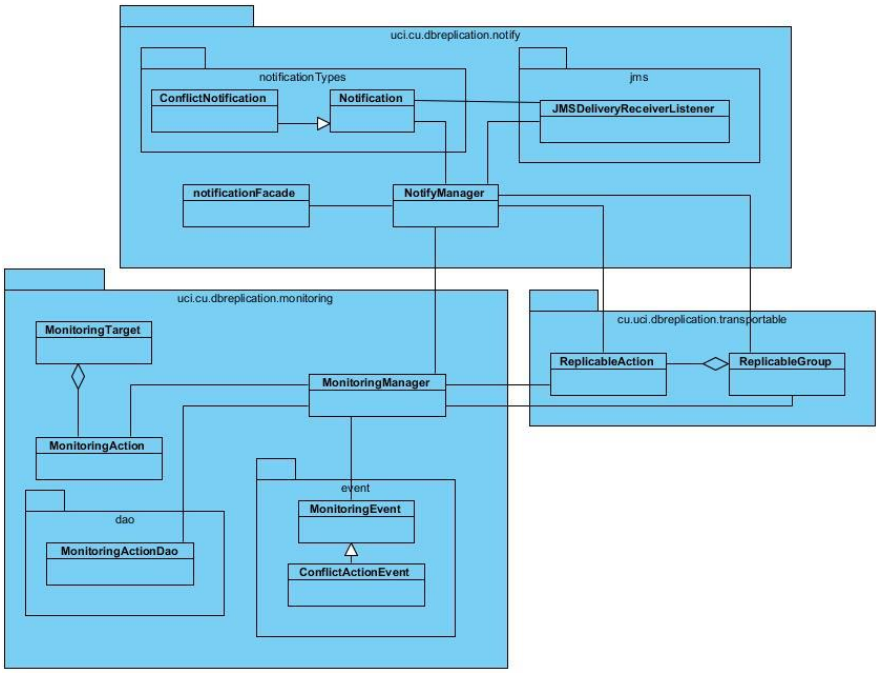


Figura 8. DC para el CU Publicar Ocurrencia de Conflicto Local

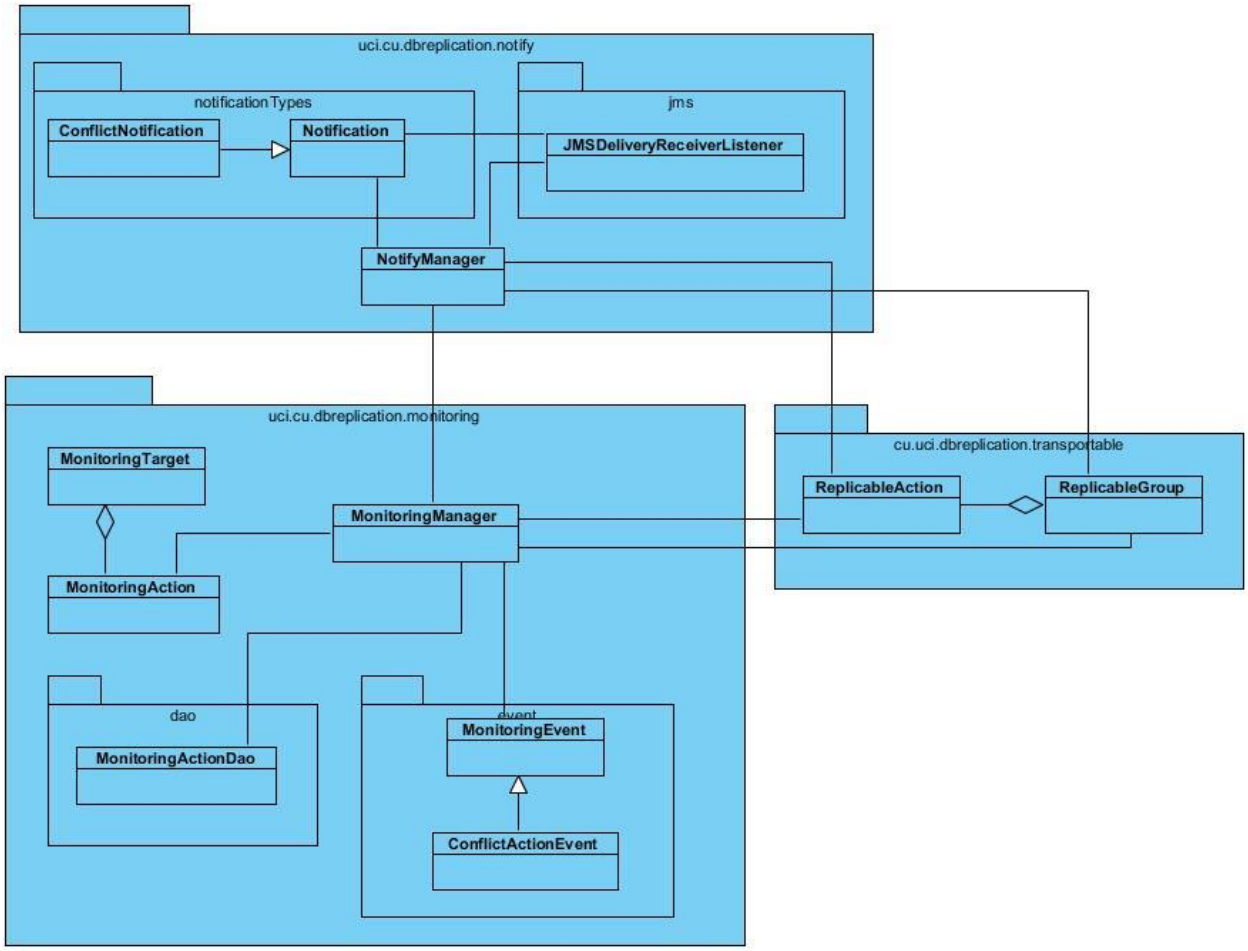


Figura 9. DC para el CU Publicar Ocurrencia Conflicto Remoto

2.7.2 Descripción de las clases

Tabla 7. Descripción de la clase NotifyManager

Descripción de la clase NotifyManager	
Nombre: NotifyManager	
Tipo de clase: Controladora	
Atributos	**Tipos**
logger	protected
deliverySender	private
receiverManager	private
monitoringManager	private
Responsabilidades	

Nombre:	onNotificationReceived(notification: cu.uci.dbreplication.notify.notificationTypes.Notification): void
Descripción:	Decide quién debe procesar las notificaciones recibidas en dependencia de su tipo.
Nombre:	onConflictSolved(notification: cu.uci.dbreplication.notify.notificationTypes.SolvedConflictNotification): void
Descripción:	Procesa notificaciones que se reciben de tipo *conflicto resuelto*. Guarda un registro en el log de confirmación de llegada. Extrae la información de la notificación y envía los datos al **`MonitoringManager`** para que se publique.
Nombre:	onConflictAvoided(notification: cu.uci.dbreplication.notify.notificationTypes.ConflictAvoidedNotification): void
Descripción:	Procesa notificaciones que se reciben de tipo *conflictos de unicidad evitados* por existencia de tupla idéntica. Guarda un registro en el log de confirmación de llegada. Extrae la información de la notificación y la envía al **`MonitoringManager`** para que se publique.
Nombre:	onRemoteConflictOccur(notification: cu.uci.dbreplication.notify.notificationTypes.ConflictNotification): void
Descripción	Procesa notificaciones que se reciben de tipo *conflicto remoto detectado*. Guarda un registro en el log de confirmación de llegada. Extrae la información de la notificación y la envía al **`MonitoringManager`** para que se publique.
Nombre:	onSentBackConflictSolution(notification: cu.uci.dbreplication.notify.notificationTypes.ConflictSolutionContainer Notification): void
Descripción:	Procesa notificaciones que se reciben de tipo *contenedoras de solución* a aplicar para mantener consistencia. Guarda un registro en el log de confirmación de llegada. Extrae datos de la notificación. Envía el grupo

	replicable contenido al aplicador para persistirlo en la BD. Envía datos al **`MonitoringManager`** para publicar la notificación.
Nombre:	sendNotification(notification: cu.uci.dbreplication.notify.notificationTypes.Notification): boolean
Descripción:	Envía una notificación al destino especificado mediante el uso del **`JMSDeliverySenderImpl`**.
Nombre:	sendConflictNotification(action: cu.uci.dbreplication.transportable.ReplicableAction, nodeWithConflict: String, senderNode: String, transactionId: String, info: String): void
Descripción:	Construye una notificación de ocurrencia de conflicto. Provee datos al **`MonitoringManager`** para publicar la notificación en el monitor local. Envía notificación al destino especificado. Guarda registro en el log de confirmación de envío.
Nombre:	sendSolvedConflictNotification(action: cu.uci.dbreplication.transportable.ReplicableAction, source: String, destination: String, transactionId: String, origSQL: String): void
Descripción:	Construye una notificación de conflicto resuelto. Provee datos al **`MonitoringManager`** para publicar la notificación en el monitor local. Envía notificación al destino especificado. Guarda registro en el log de confirmación de envío.
Nombre:	sendConflictAvoidedNotification(action: cu.uci.dbreplication.transportable.ReplicableAction, source: String, destination: String, transactionId: String): void
Descripción:	Construye una notificación de conflicto de unicidad evitado por existencia de tupla idéntica. Provee datos al **`MonitoringManager`** para publicar la notificación en el monitor local. Envía notificación al destino especificado. Guarda registro en el log de confirmación de envío.

Tabla 8. Descripción de la clase MonitoringManager

Descripción de la clase MonitoringManager	
Nombre: MonitoringManager	
Tipo de clase: Controladora	
Atributos	**Tipos**
count1	private
applicationEventPublisher	private
monitoringActionDao	private
Responsabilidades	
Nombre:	onConflictReplicableActionData(action: cu.uci.dbreplication.transportable.ReplicableAction, idGroup: String, senderId: String, info: String): void
Descripción:	Construye una **`MonitoringAction`** para publicar en el monitor de réplica las notificaciones de conflictos ocurridos localmente.
Nombre:	onRemoteConflictAction(action: cu.uci.dbreplication.transportable.ReplicableAction, idGroup: String, nodeWithConflict: String, info: String): void
Descripción:	Construye una **`MonitoringAction`** para publicar en el monitor de réplica las notificaciones de conflictos ocurridos en el nodo remoto.
Nombre:	onConflictAvoidedAction(action: cu.uci.dbreplication.transportable.ReplicableAction, idGroup: String, source: String, destination: String, direction: int, info: String): void
Descripción:	Construye una **`MonitoringAction`** para publicar en el monitor de réplica las notificaciones de conflictos de unicidad evitados por existencia de tupla con datos idénticos.
Nombre:	onSolvedConflictAction(action: cu.uci.dbreplication.transportable.ReplicableAction, idGroup: String, source: String, destination: String, direction: int, info: String, origSQL: String): void

Descripción	Construye una **`MonitoringAction`** para publicar en el monitor de réplica las notificaciones de conflictos resueltos.
Nombre:	onSentBackConflictSolutionAction(action: cu.uci.dbreplication.transportable.ReplicableGroup, origSQL: String, source: String, destination: String, direction: int, info: String): void
Descripción:	Construye una **`MonitoringAction`** para publicar en el monitor de réplica las notificaciones contenedoras de soluciones a aplicar para mantener consistencia de información en ambos nodos.
Nombre:	persistMonitoringAction(monitoringAction: cu.uci.dbreplication.monitoring.MonitoringAction): void
Descripción:	Persiste, en la base de datos local embebida de Berkeley, la acción de monitoreo para mantenerla en los historiales.

2.7.3 Diagramas de Interacción

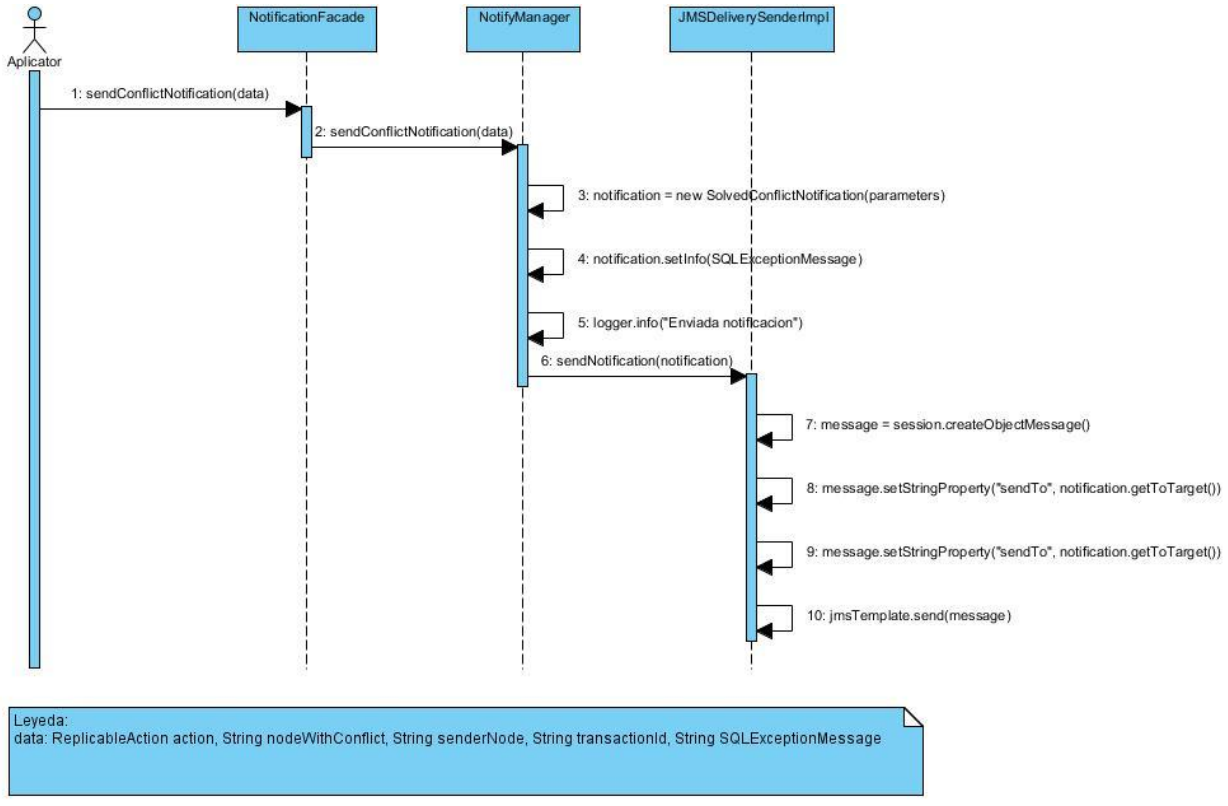


Figura 10. DS para el CU Generar Notificación Ocurrencia Conflicto

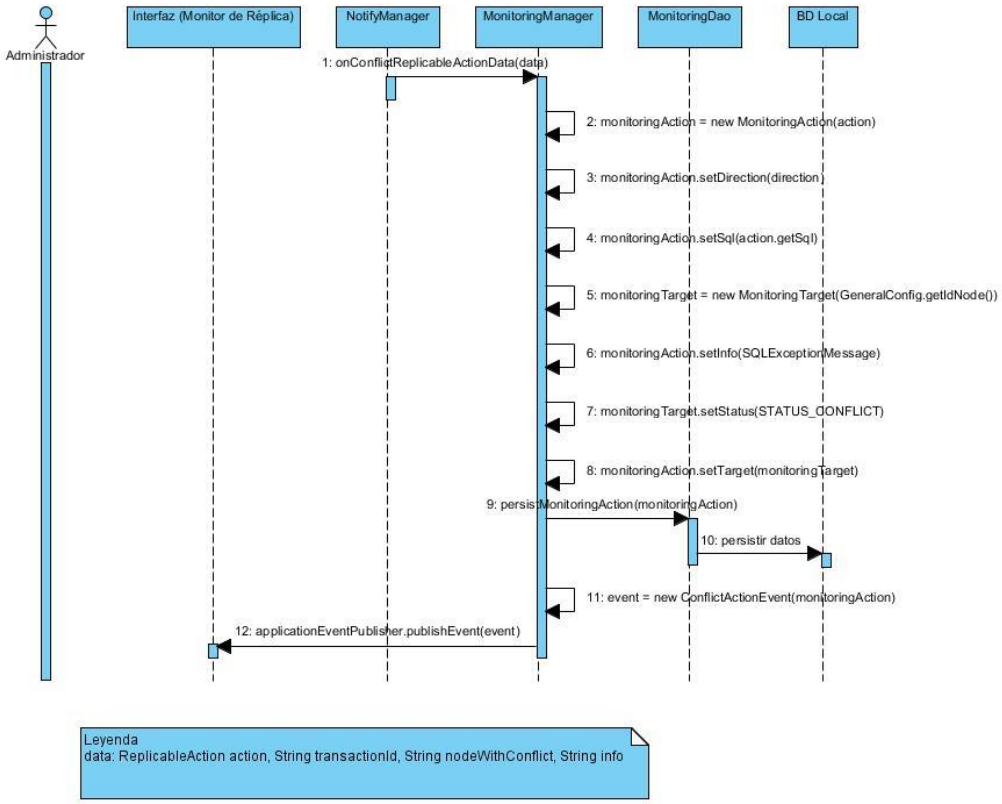


Figura 11. DS para el CU Publicar Ocurrecia Conflicto Local

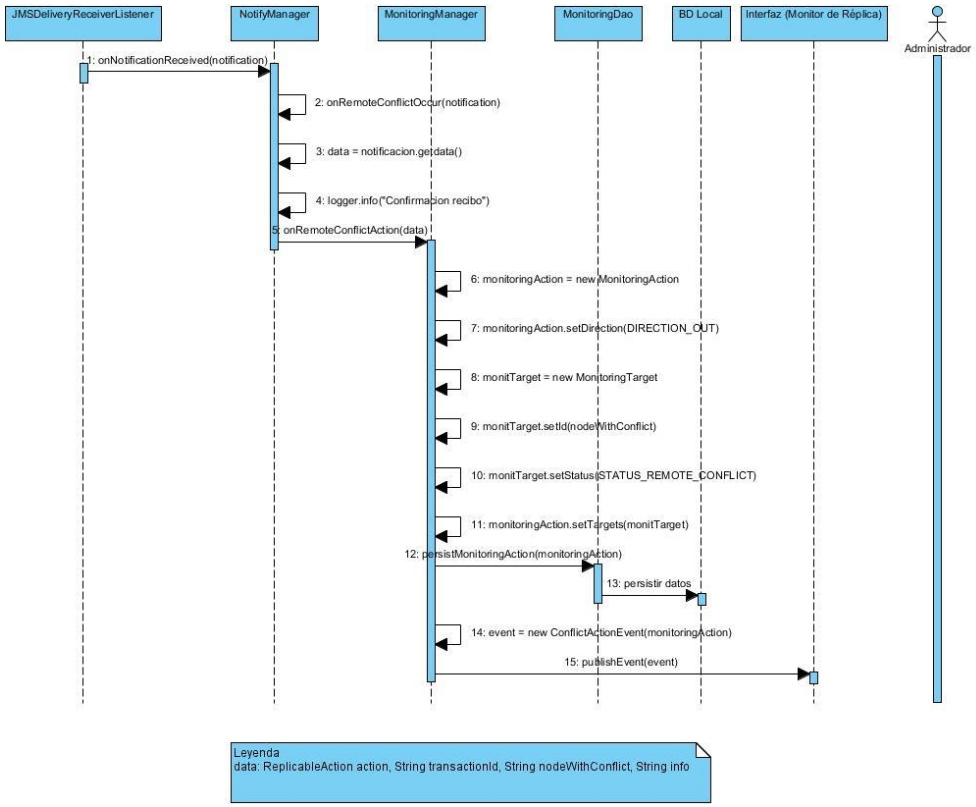


Figura 12. DS para el CU Publicar Ocurrencia Conflicto Remoto

2.8 Modelo de despliegue

Los Diagramas de Despliegue muestran las relaciones físicas de los distintos nodos que componen un sistema y el reparto de los componentes sobre dichos nodos. La vista de despliegue representa la disposición de las instancias de componentes de ejecución en instancias de nodos conectados por enlaces de comunicación. Un nodo es un recurso de ejecución tal como un computador, un dispositivo o memoria. Los estereotipos permiten precisar la naturaleza del equipo:

- Dispositivos
- Procesadores
- Memoria

Esta vista tiene una forma de descriptor y otra de instancia. La forma de instancia muestra la localización de las instancias de los componentes específicos en instancias específicas del nodo como parte de una configuración del sistema. La forma de descriptor muestra qué tipo de componentes pueden subsistir en qué tipos de nodos y qué tipo de nodos se pueden conectar, de forma similar a un diagrama de clases, esta forma es menos común que la primera. Un diagrama de despliegue es un grafo de nodos unidos por conexiones de comunicación. (28)

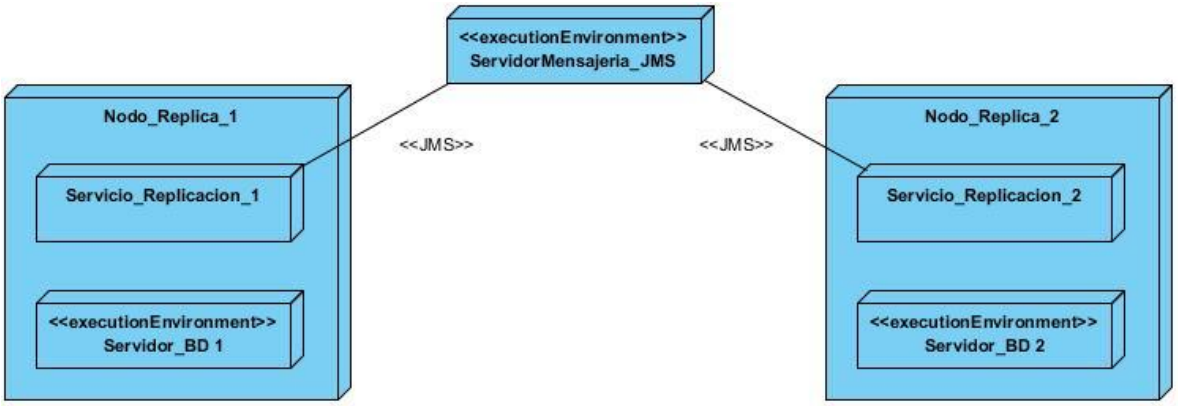


Figura 13. Diagrama de Despliegue

2.9 Consideraciones Parciales

A partir del desarrollo del capítulo se obtienen las siguientes consideraciones:

- ✓ Se abordaron de manera general los aspectos relacionados con el proceso de análisis y diseño del módulo de gestión de notificaciones de conflictos de réplica. Esto servirá como base fundamental para aportar un mejor entendimiento del análisis del siguiente capítulo, el cual abordará acerca del proceso de implementación del módulo.
- ✓ Se brinda una amplia descripción de la arquitectura del replicador de datos Reko y la arquitectura propuesta como solución, enfocada en el módulo de gestión de notificaciones de conflictos de réplica en el replicador Reko.
- ✓ Se detallaron el conjunto de patrones de diseño que se utilizarán para el desarrollo del módulo.
- ✓ Se detalló el modelo de dominio para brindar un entendimiento del sistema y así poder identificar los requisitos funcionales y no funcionales. Esto a su vez conlleva a la definición de los casos de uso y sus relaciones, acompañados de su descripción.
- ✓ Se presentaron los diagramas de clase respectivos a los casos de uso para mostrar las relaciones entre clases. También se realizó una descripción de las clases fundamentales.
- ✓ Se brindaron los diagramas de secuencia para los casos de uso y se elaboró además el diagrama de despliegue para ilustrar cómo estará físicamente distribuido el sistema.

CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS

En este capítulo se elaborará el modelo de implementación donde se obtendrá el diagrama de componentes y se describirán algunos de los estándares de codificación utilizadas. Además, se confeccionará el modelo de pruebas donde se abordarán detalles del diseño, la ejecución y los resultados de las pruebas que le serán aplicadas al sistema.

3.1 Modelo de implementación

El Modelo de Implementación es comprendido por un conjunto de componentes y subsistemas que constituyen la composición física de la implementación del sistema. Entre los componentes se pueden encontrar datos, archivos, ejecutables, código fuente y los directorios. Fundamentalmente, se describe la relación que existe desde los paquetes y clases del modelo de diseño a subsistemas y componentes físicos.

Este artefacto describe cómo se implementan los componentes, congregándolos en subsistemas organizados en capas y jerarquías, señalando sus dependencias. Además, en este modelo se realizan las pruebas de unidad por lo que el desarrollador es el responsable de probar las unidades que produzca y el resultado final es un sistema ejecutable. (29)

3.2 Diagrama de Componentes

Un diagrama de componentes modela la división del sistema en componentes físicos y muestra las dependencias entre estos componentes. Los componentes físicos incluyen archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables, librerías, tablas, paquetes o documentos que formen parte del sistema. Uno de los usos principales es el de identificar qué componentes puede ser compartido entre las partes de un sistema o entre distintos sistemas.

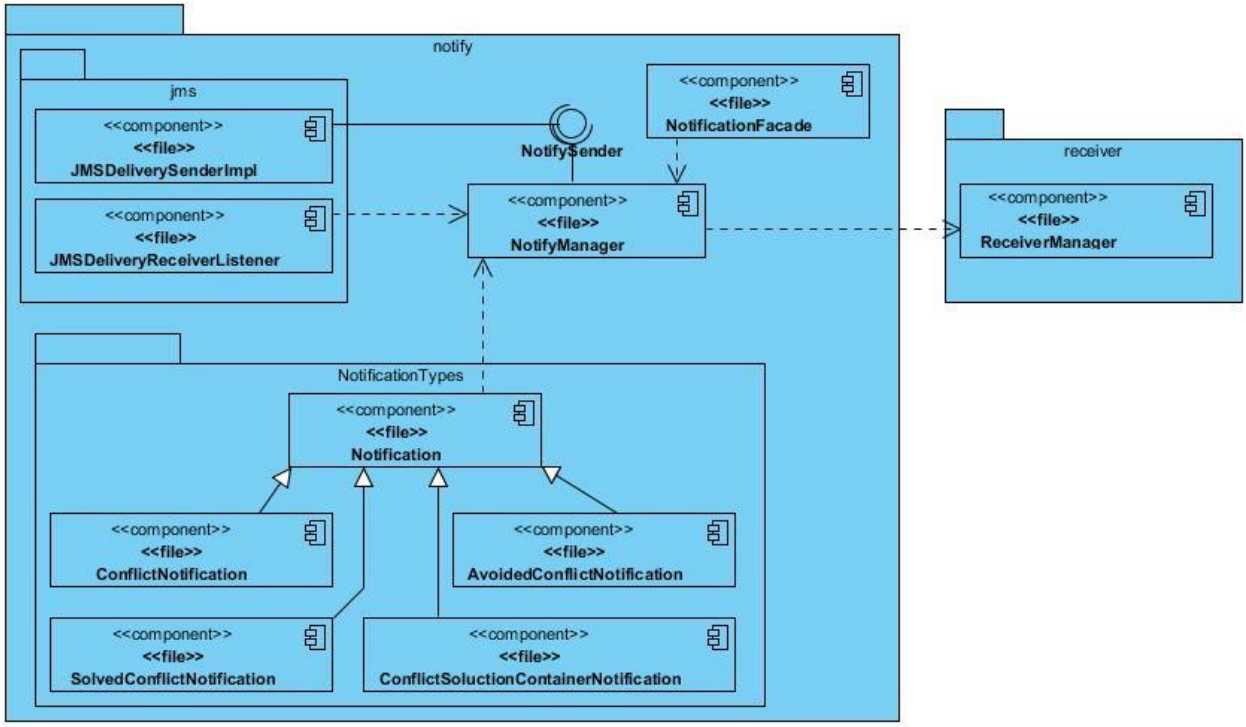


Figura 14. Diagrama de Componentes para el módulo notificaciones en el subsistema Distribuidor de Datos.

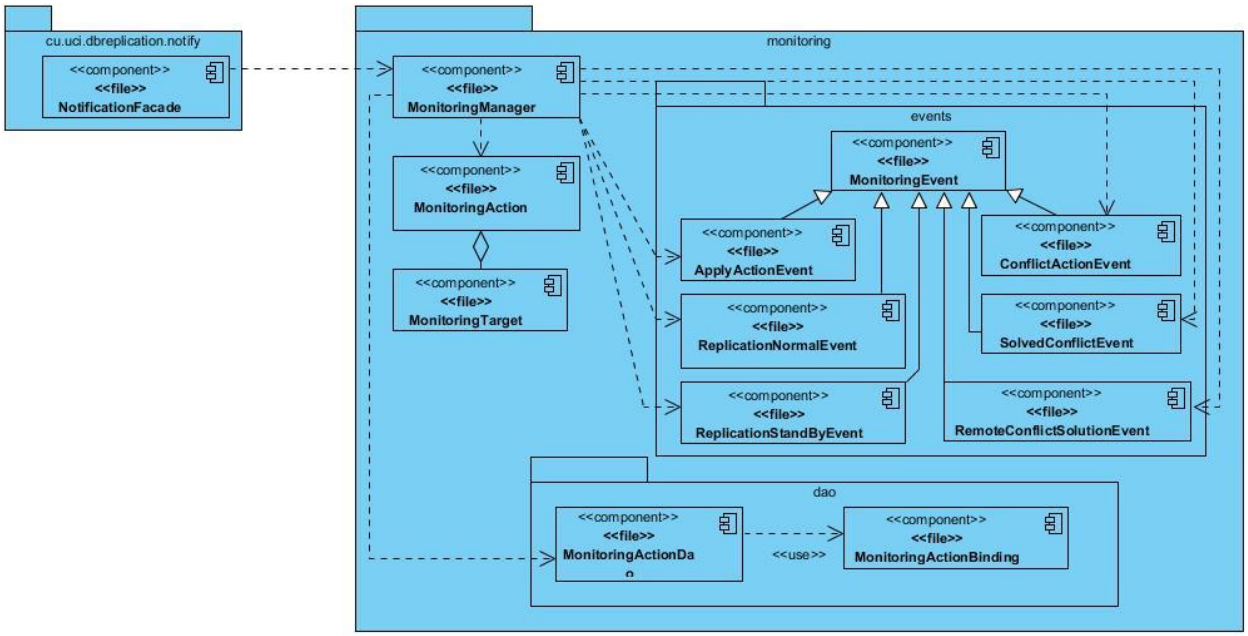


Figura 15. Diagrama de Componentes para el módulo de monitoreo en el subsistema Administrador.

3.3 Código Fuente

El siguiente fragmento de código pertenece a la clase MonitoringManager. El método **onSentBackConflictSolutionAction(ReplicableGroup group, String origSQL, String source, String destination, int direction, String info)** se encarga de construir las acciones de monitoreo correspondientes a las notificaciones que se generan cuando un conflicto se soluciona. En este caso se trata de la solución que requiere enviar datos al nodo que originó la réplica. Estos datos forman parte de dicha solución y se necesitan persistir en la base de datos del nodo origen para garantizar la mayor consistencia en la información posible.

MonitoringManager

```
public void onSentBackConflictSolutionAction(ReplicableGroup group, String
      origSQL, String source, String destination, int direction,
      String info)
      {
            if (count1 == 100000) {
                  count1 = 0;
            }
            ReplicableAction action = group.getActions().get(0);
            List<MonitoringAction> actions = new
            LinkedList<MonitoringAction>();
            MonitoringAction monitoringAction = new MonitoringAction();
            monitoringAction.setId(group.getId() + "-" + count1++);
            monitoringAction.setTable(action.getTable());
            String sql = "<strong>SQL en conflicto: </strong> <br/>" +
                  origSQL + "<br/><strong>SQL Generado: </strong><br/>";
            sql += action.toString();
            monitoringAction.setActionType(action.getActionType());
            if (group.getActions().size() > 1) {
                  monitoringAction.setActionType(4);
                  sql += "<br/>" +
                        group.getActions().get(1).toString();
            }
            monitoringAction.setSql(sql);
            monitoringAction.setDirection(direction);
            monitoringAction.setGroupId(group.getId());
            monitoringAction.setInfo(info);
            //El sender en este caso es el que soluciona el conflicto
            y provee la solucion del mismo
            monitoringAction.setSenderId(source);
            monitoringAction.setDate(getDate());
            List<MonitoringTarget> targets = new
                  LinkedList<MonitoringTarget>();
            MonitoringTarget target = new MonitoringTarget();
            target.setId(destination);
```

```
            if (direction == 1) {
                  target.setStatus(
                  MonitoringTarget.STATUS_REMOTE_CONFLICT_SOLUTION_DATA);
            }
            else {
                  target.setStatus(
                  MonitoringTarget.STATUS_CONFLICT_SOLUTION_DATA);
            }
            targets.add(target);
            monitoringAction.setTargets(targets);
            persistMonitoringAction(monitoringAction);
            actions.add(monitoringAction);
            applicationEventPublisher.publishEvent(new
                  RemoteConflictSolutionEvent(this, actions));
      }
```

El método que se muestra a continuación pertenece a la clase **NotifyManager** y se utiliza para la creación y el envío de una notificación de un conflicto que ha sido solucionado.

NotifyManager

```
public void sendSolvedConflictNotification(ReplicableAction action,
      String source, String destination, String transactionId, String
      solutionSQL)
      {
            SolvedConflictNotification notif = new SolvedConflictNotification(
                        Notification.EXECUTE_SUCCESS, destination, source,
                        transactionId, action);
            notif.setSolutionSQL(solutionSQL);
            String info = "Conflicto solucionado exitosamente";
            notif.setInfo(info);
            //publicar en el nodo actual
            monitoringManager.onReplicationContinue();
            monitoringManager.onSolvedConflictAction(action, transactionId,
                  source, destination, 1, info, solutionSQL);
            //enviar notificacion al nodo remoto para su publicación
            sendNotification(notif);
            //registrar en el logger
            logger.info("Enviada notificacion de conflicto resuelto al nodo: "
                        +"'" + notif.getToTarget() + "'");
      }
```

Este método se encarga de recibir las notificaciones que provienen de nodos remotos, específicamente las notificaciones que contienen información a aplicar en la base de datos para mantener la consistencia.

NotifyManager

```
public void onSentBackConflictSolution(ConflictSolutionContainerNotificaction
      notification)
      {
            try {
                  receiverManager.persistInLocalDB(
                  notification.getReplicableGroup());
                  String oldSql = notification.getOrigSQL();
                  String nodeThatSolvedConflict = notification.getTarget();
                  ReplicableGroup fixedTrasaction =
                  notification.getReplicableGroup();
                  String destination = notification.getToTarget();
                  String info = notification.getInfo();
                  if (monitoringManager.isListenning()){
                        monitoringManager.onSentBackConflictSolutionAction(
                                    fixedTrasaction, oldSql,
                                    nodeThatSolvedConflict,
                                    destination, 1, info);
                  }
                  logger.info("Recibida grupo conteniendo solucion de" +
                        conflicto a aplicar desde '" +
                        notification.getTarget() + "'");
            } catch (ConflictDatabaseException e) {
                  logger.error(e,e);
            }
      }
```

3.4 Modelo de Pruebas

Una vez generado el código fuente, es necesario probar el software para descubrir y corregir la mayor cantidad de errores posibles antes de entregarlo al cliente.

La fase de pruebas tiene como objetivo verificar el sistema para comprobar si este cumple sus requisitos. Dentro de esta fase pueden desarrollarse diferentes tipos de pruebas en función de los objetivos de las mismas. (30)

Existen generalmente cuatro niveles reconocidos de pruebas: pruebas unitarias, pruebas de integración, pruebas de sistema y pruebas de aceptación. Estos niveles son frecuentemente agrupados por dónde son añadidos en el proceso de desarrollo de software, o por el nivel de especificidad de la prueba. Los principales niveles durante el proceso de desarrollo de software como está definido en la guía SWEBOK[27] son los de

[27] SWEBOK de sus siglas en inglés ***Software Engineering Body of Knowledge***, en español *Conjunto de Conocimiento de Ingeniería de Software*. Estándar internacional ISO/IEC TR 19759:2005 que especifica una guía a los conocimientos generales aceptados de Ingeniería de Software

unidad, *integración* y *sistema* que se distinguen por el objeto a probar sin implicar un modelo de proceso específico. (31) Otros niveles de pruebas se clasifican según el objetivo de las mismas.

Los métodos de prueba de software tradicionalmente se dividen en pruebas de caja blanca y pruebas de caja negra. Estos acercamientos se utilizan para describir el punto de vista que toma un ingeniero cuando diseña los casos de prueba.

Las pruebas de caja blanca se enfocan en la estructura interna o el funcionamiento del programa, en vez de las funcionalidades expuestas al usuario final. En este tipo de pruebas, una perspectiva interna del sistema, así como también habilidades en programación, son usadas para diseñar los casos de pruebas. El probador escoge las entradas para ejercitar caminos a través del código y determinar salidas apropiadas. Mientras que las pruebas de caja blanca se pueden aplicar en los niveles de unidad, integración y sistema, usualmente se realizan en el nivel de unidad.

Las pruebas de caja negra son un método que examina la funcionalidad de una aplicación sin enfocarse en su estructura interna o funcionamiento. Este método de pruebas se puede aplicar a virtualmente cualquier nivel del proceso de desarrollo. Los casos de prueba diseñados, pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa se mantiene. (32)

A partir del análisis de todos los elementos anteriormente expuestos se decidió realizar pruebas unitarias de software, asegurando en gran medida el funcionamiento correcto de la solución propuesta.

Con el objetivo de validar la solución implementada se diseñaron y ejecutaron una serie de pruebas para el módulo desarrollado. Por las características presentadas por el mismo.

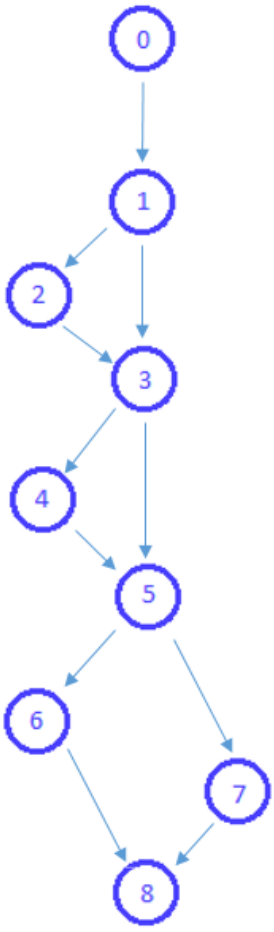
3.4.1 Camino Básico

El camino básico es una técnica de pruebas de caja blanca utilizada para diseñar casos de pruebas cuya intención es examinar todos los posibles caminos de ejecución al menos una vez. El método usa un grafo de control de flujo para convertir el código en un modelo y posteriormente derivar caminos independientes de prueba a partir de él. Cada camino de prueba corresponde a una posible ejecución en el software. Estos caminos a su vez

serán usados para generar casos de pruebas que servirán para verificar cuán correcto es el código.

MonitoringManager

```
0 public void onSentBackConflictSolutionAction(ReplicableGroup group, String
      origSQL, String source, String destination, int direction,
      String info)
      {
1           if (count1 == 100000) {
2                 count1 = 0;
            }
3           ReplicableAction action = group.getActions().get(0);
3           List<MonitoringAction> actions = new
            LinkedList<MonitoringAction>();
3           MonitoringAction monitoringAction = new MonitoringAction();
3           monitoringAction.setId(group.getId() + "-" + count1++);
3           monitoringAction.setTable(action.getTable());
3           String sql = "<strong>SQL en conflicto: </strong> <br/>" +
                  origSQL + "<br/><strong>SQL Generado: </strong><br/>";
3           sql += action.toString();
3           monitoringAction.setActionType(action.getActionType());
3           if (group.getActions().size() > 1) {
4                 monitoringAction.setActionType(4);
4                 sql += "<br/>" +
4                       group.getActions().get(1).toString();
            }
5           monitoringAction.setSql(sql);
            monitoringAction.setDirection(direction);
5           monitoringAction.setGroupId(group.getId());
5           monitoringAction.setInfo(info);
5           monitoringAction.setSenderId(source);
5           monitoringAction.setDate(getDate());
5           List<MonitoringTarget> targets = new
                  LinkedList<MonitoringTarget>();
5           MonitoringTarget target = new MonitoringTarget();
5           target.setId(destination);
5           if (direction == 1) {
6                 target.setStatus(
6                 MonitoringTarget.STATUS_REMOTE_CONFLICT_SOLUTION_DATA);
            }
            else {
7                 target.setStatus(
7                 MonitoringTarget.STATUS_CONFLICT_SOLUTION_DATA);
            }
8           targets.add(target);
8           monitoringAction.setTargets(targets);
8           persistMonitoringAction(monitoringAction);
8           actions.add(monitoringAction);
8           applicationEventPublisher.publishEvent(new
                  RemoteConflictSolutionEvent(this, actions));
      }
```

Construcción del grafo de complejidad ciclomática correspondiente al código	
	**Complejidad Ciclomática:** A = 11 (Aristas) N = 9 (Nodos) V(G) = A – N + 2 V (G) = 11 – 9 + 2 = 4 **P = 3 (Nodos predicados)** V (G) = P + 1 V (G) = 3 + 1 V (G) = 4 **R (Regiones)** R = V (G) R = 4 1. 0, 1, 3, 5, 6 2. 0, 1, 3, 5, 7 3. 0, 1, 2, 3, 4, 5, 7, 8 4. 0, 1, 2, 3, 4, 5, 6, 8

No.	Caso de prueba	Resultado esperado
1	Publica una notificación que indica que se ha recibido una solución con datos a aplicarse en la BD para garantizar consistencia de información.	Mostrar en el monitor de réplica la notificación que indica “Recibida solución con datos”
2	Publica una notificación que indica que se ha enviado una solución con datos a aplicarse en la	Mostrar en el monitor de réplica la notificación que

	BD del nodo remoto para garantizar consistencia de información.	indica “Enviada solución con datos”
3	Publica una notificación que indica que la solución recibida/enviada posee solo una acción de réplica a aplicar	Muestra en el monitor de réplica la notificación y especifica en el campo *Observación* una breve explicación de los cambios a aplicar. Además muestra como tipo de acción Inserción, Actualización o Eliminación (Solo uno de los tres)
4	Publica una notificación que indica que la solución recibida/enviada posee más de una acción de réplica a aplicar	Muestra en el monitor de réplica la notificación y especifica en el campo *Observación* una breve explicación de los cambios a aplicar. Además muestra como tipo de acción Inserción/Actualización.

3.4.2 Framework JUnit

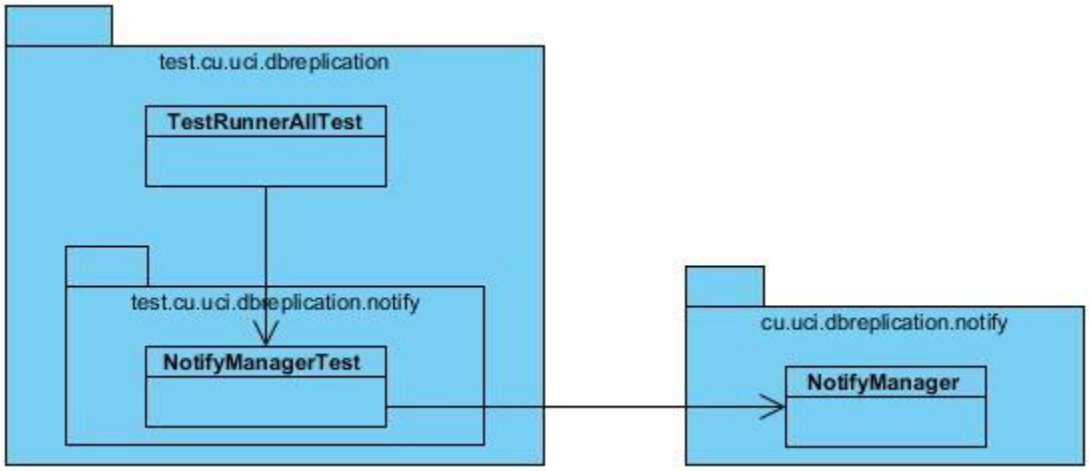
JUnit es un framework de pruebas para el lenguaje de programación Java. Es importante su uso en el desarrollo dirigido por pruebas y es un miembro de la familia de frameworks de prueba de unidad colectivamente conocida como xUnit.

Se escogió JUnit para la aplicación de las pruebas automatizadas al resto de las funcionalidades, por su amplio uso y las facilidades que brinda para recolectar resultados de manera estructurada, reportes, relaciones entre las pruebas y la reutilización de código entre ellas.

Para realizar las pruebas, una buena práctica es usar paquetes paralelos a la estructura de las clases a ser probadas en una carpeta de código distinta, lo cual permite acceder

a los atributos y métodos protegidos de las clases y hace más fácil encontrar las clases de prueba de cualquier clase del sistema.

En la siguiente imagen se muestra la relación establecida entre la clase a probar y las clases de prueba.



Las pruebas a la clase **`NotifyManager`** fueron aplicadas a través de la clase **`TestRunnerAllTest`**. Las mismas arrojaron los resultados que se muestran a continuación.

```
****** Aplicando pruebas a la clase NotifyManager *******
.   Prueba: testSendSolvedConflictNotification
     Tiempo de ejecución: 167

****** Aplicando pruebas a la clase NotifyManager *******
.   Prueba: testSendConflictAvoidedNotification
     Tiempo de ejecución: 206

******* Aplicando pruebas a la clase NotifyManager *******
.   Prueba: testSendBackConflictSolution
     Tiempo de ejecución: 240

****** Aplicando pruebas a la clase NotifyManager *******
.   Prueba: testSendConflictNotification
     Tiempo de ejecución: 148

Time: 0.762

OK (4 tests)
```

Como se puede apreciar las pruebas manifestaron resultados satisfactorios así como también bajos tiempos de ejecución lo cual demuestra la validez del código.

3.4.3 Pruebas de Sistema

Las pruebas de sistema de un software son llevadas a cabo en un sistema completo e integrado para evaluar el cumplimiento de los requisitos especificados por el mismo. Este tipo de pruebas cae en el enfoque de pruebas de caja negra de manera tal que no requiere conocimiento del diseño interno del código o de la lógica de la aplicación. (33)

Se diseñaron un conjunto de casos de pruebas que permiten validar el cumplimiento de los requisitos funcionales, siendo en este caso la publicación de las diferentes notificaciones correspondientes a los conflictos. Estos casos de prueba finalmente permitieron verificar que en efecto se cumple correctamente con los requerimientos, emitiendo resultados satisfactorios.

3.5 Consideraciones Parciales

- ✓ El componente mostró ser completamente funcional, dando solución a todos los requisitos planteados.
- ✓ Los problemas de tratamiento de errores que fueron descubiertos durante el desarrollo del módulo fueron solucionados.
- ✓ Los componentes diseñados en el modelo de implementación permiten la organización del código ejecutable.
- ✓ Las pruebas dieron resultados satisfactorios, mostrando que el componente es funcional y que da solución a los requisitos planteados.
- ✓ La realización de las pruebas constituye la certificación de que el componente soporta las funcionalidades requeridas.

CONCLUSIONES GENERALES

- ✓ El estudio teórico realizado sobre las notificaciones de los conflictos de réplica de datos en sistemas distribuidos fue dirigido por la definición de diversos conceptos fundamentales y el análisis de herramientas existentes de replicación. En lo que respecta a las herramientas se enfocó en cómo gestionan la generación de notificaciones ante la ocurrencia de conflictos. Este análisis sirvió como base teórica para el diseño del módulo.
- ✓ El diseño del módulo para la gestión de las notificaciones de los conflictos de réplica en el Replicador Reko permitió obtener una descripción detallada de sus principales elementos y cómo interactúan entre ellos. Estos elementos fueron utilizados como guía para la implementación de la solución.
- ✓ La aplicación de un conjunto de pruebas permitieron validar el correcto funcionamiento de la solución obtenida. Con la utilización de diversos métodos de prueba orientados a obtener buenos resultados y tiempos de respuesta mínimos, se fueron corrigiendo las no conformidades detectadas en las iteraciones de pruebas. En conclusión se lograron resultados satisfactorios que muestran la correcta operación del módulo.
- ✓ El módulo obtenido gestiona el proceso de notificaciones de conflictos en el Replicador Reko a manera de mantener la consistencia de la información en un Sistema de Bases de Datos Distribuido.

REFERENCIAS BIBLIOGRÁFICAS

1. **Date, C.J.** *Introducción a los Sistemas de Bases de Datos.* México : Pearson Educación, 2001.

2. **Pérez Valdés, Damián.** Maestros del Web. [En línea] 2007. http://www.maestrosdelweb.com/editorial/%c2%bfque-son-las-bases-de-datos/.

3. **Bolton, David.** About.com. [En línea] 2010. http://cplus.about.com/od/glossar1/g/databasedefn.htm.

4. **Sierra, Manuel.** *¿Qué es una Base de datos y cuáles son los principales tipos?* 2013.

5. **Özsu Tamer, M. y Valduriez, Patrick.** *Principles of Distributed Database Systems.* s.l. : Springer, 2011. ISBN 9781441988348.

6. **Microsoft Developer Network.** SQL Server Replication. [En línea] http://msdn.microsoft.com/en-us/library/ms151198.aspx.

7. **Urbano, Randy.** *Oracle Database Advanced Replication, 11g Release 2.* 2013.

8. **Chapple, Mike.** About.com. [En línea] http://databases.about.com/cs/administration/g/replication.htm.

9. **Microsoft.** Microsoft SQL Server Technet Library. [En línea] 2014. http://technet.microsoft.com/en-us/library/ms151257.aspx.

10. **Slony Development Group.** Slony-I Enterprise Level Replication System. [En línea] http://slony.info/documentation/1.2/slonyintro.html.

11. **Wieck, Jan.** *Slony-I A replication system for PostgreSQL.* s.l. : Afilias USA INC. Horsham, Pennsylvania, USA.

12. **Oracle.** Oracle Technology Network. [En línea] http://www.oracle.com/technetwork/database/information-management/streams-fov-11g-134280.pdf.

13. —. Oracle Technology Network. [En línea] http://www.oracle.com/technetwork/java/javaee/overview/index.html.

14. **Weitzenfeld, Alfredo.** *Ingeniería de software orientada a objetos con UML, Java e Internet.* s.l. : Ediciones Paraninfo, 2004. ISBN 9789706861900.

15. **Hamilton, Kim y Russell, Miles.** *Learning UML 2.0.* s.l. : O'Reilly, 2006. ISBN 9780596009823.

16. **Spring.** Spring Tool Suite. [En línea] http://spring.io/tools/sts.

17. **Snyder, Bruce, Dejan, Bosnanac y Davies, Rob.** *ActiveMQ in Action.* s.l. : Manning Publications Company, 2011. ISBN 9781933988948..

18. **Apache Tomcat.** Apache Tomcat Homepage. [En línea] http://tomcat.apache.org/.

19. **Walls, Craig.** *Spring in Action.* s.l. : Manning Publications, 2013. ISBN 9781617291203.

20. **Oracle.** Java Message Service Concepts. [En línea] http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html.

21. **Jacobson, Ivar, Booch, Grady y Rumbaugh, James.** *El proceso unificado de desarrollo de software.* s.l. : Pearson Education, 2000.

22. **Pressman, R.S.** *Ingeniería del Software, Un Enfoque Práctico.* s.l. : McGraw-Hil, 2002. ISBN 8448132149.

23. **Sommerville, Ian.** *Requerimientos.* s.l. : Pearson Education, 2005.

24. **SOFTWARE ENGINEERING STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY.** *Recommended Practice for Architectural Description of Software-Intensive Systems.* New York : The Institute of Electrical and Electronics Engineers, Inc., 2000.

25. **Microsoft Developer Network.** Desarrollo de software basado en componentes. [En línea] 2013. http://msdn.microsoft.com/es-es/library/bb972268.aspx.

26. **Martin, Robert C.** *Design Principles and Design Patterns.* 2000.

27. **Larman, C.** *UML y Patrones. Introducción al análisis y diseño orientado a objetos y el proceso unificado.* México : Prentice Hall, 2005. ISBN 9701702611.

28. **Microsoft.** Microsoft SQL Server Technet Library. [En línea] 2014. http://technet.microsoft.com/en-us/library/ms152746.aspx.

29. —. Microsoft SQL Server Technet Library. [En línea] 2014. http://technet.microsoft.com/en-us/library/ms147839.aspx.