UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

Facultad 5

Centro de Consultoría y Desarrollo de Arquitecturas Empresariales (CDAE)

Módulo de resolución automática de conflictos para el Replicador de Datos Reko.

Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas.

Autor: Adrian Hernández Velozo.

Tutora: Ing. Katia Roselló Díaz.

La Habana, Junio de 2014

MINISTERIO DE EDUCACIÓN SUPERIOR

UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

Declaro ser autor del presente trabajo de diploma y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales del mismo, 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 ____.

_____

Adrian Hernández Velozo

Firma del Autor

_____

Ing. Katia Roselló Díaz

Firma del Tutor

A mi madre, mi padre, mis hermanos y a Leydis.

A mis amigos, que aunque no son ni muchos ni pocos, son los suficientes.

Agradecimientos

A mi madre que me lo ha dado todo en el mundo y mejor no puede ser.

A mi padre que a pesar de no estar juntos forma parte de mi vida.

A mi hermano Héctor por mostrarme uno de los caminos a seguir en la vida y por el conocimiento y cariño entregado.

A mi hermano Osdelito que me vuelve loco todo el tiempo, me termina la paciencia de forma precipitada pero igual siempre lo voy a querer con el corazón.

A Leydis no solo por ser mi novia sino gran parte de mi vida, por cada día robarme un pedacito de mi corazón y hacerlo suyo, por hacer de mi vida algo maravilloso, por compartir largos ratos frente a este trabajo y aprender sobre los "conflictos", por hacerme orgulloso de su persona y por tantas cosas más que me sería imposible expresarlo en estas pocas líneas.

A la familia de Leydis, a sus padres y sus abuelos por dejarme formar parte de su mundo y entregarme su cariño.

A mis hermanos de la Lenin: Beny, ReY, Mario, Racendy, Alexis y Carrillo por llegar a ser como una familia y por ser amigos de toda la vida.

A mis amigos Lissainet, Beatriz y Julio por vivir momentos inolvidables junto a Leydis y a mí y por compartir aventuras por toda Cuba, sin contar las que nos faltan.

A la gente de la UCI que han formado parte de mi vida en esta etapa: Alexis, Ramírez, Susana, Miosotis, Alfredo, Alieski, Carlos, Yorgüy y Katerín.

A Pupo por dedicarme su tiempo, por su forma de ser y por siempre ver primero la parte buena de este mundo y hasta que se demuestre lo contrario.

A Helián por ser casi como un compañero de tesis, solo faltó oficializarlo, por aportar tanto a este trabajo y formar parte del resultado obtenido.

Al grupo de trabajo del proyecto: Albín, Frank y Gloria por aportarme su conocimiento sobre Reko.

A Vilma por darme la oportunidad de superarme profesionalmente.

A todas las personas que de una forma u otra han aportado a mi formación como profesional y como persona.

A todos…muchas gracias.

Velozo.

RESUMEN

Durante la réplica de datos en Sistemas de Bases de Datos Distribuidos ocurren conflictos debido a la inconsistencia de la información, deteniendo el proceso de réplica hasta que se solucionen. El Replicador de Datos Reko es un software multiplataforma que permite la réplica de datos entre distintos gestores y brinda un módulo de resolución de conflicto, el cual presenta limitantes frente a la resolución automática de conflictos. El mecanismo de resolución automática solo permite realizar dos acciones automáticas sobre los conflictos de unicidad y eliminación que se detectan. Como consecuencia, no garantiza que se resuelvan correctamente los conflictos de unicidad para distintos escenarios, y no asegura la consistencia de los datos, dejando la mayor responsabilidad de esta tarea al mecanismo de resolución manual.

Se propone un módulo de resolución automática de conflictos para el Replicador de Datos Reko que permita capturar, clasificar y resolver de forma automática los conflictos que ocurran durante el proceso de réplica. Dicho módulo tiene como objetivo brindar variantes de soluciones automatizadas para los distintos tipos de conflictos que puedan ocurrir durante el proceso de réplica, asegurando en todo momento la consistencia de la información.

Se emplea la metodología OpenUp para guiar el diseño e implementación del módulo, el cual fue desarrollado completamente con herramientas libres y librerías de clases con licencias gratuitas. Se muestra un análisis de las pruebas realizadas sobre el módulo desarrollado. Se realiza una comparación en cuanto a cantidad de funcionalidades y eficiencia, con la versión anterior del Replicador de Datos Reko.

Palabras claves: OpenUp, resolución automática de conflictos, réplica de datos, Sistema de Bases de Datos Distribuidos.

Contenido

INTRODUCCIÓN 1
CAPÍTULO 1. FUNDAMENTACIÓN TEÓRICA DE LOS CONFLICTOS DE RÉPLICA 5
1.1 Marco conceptual 5
1.1.1 Bases de Datos (BD) 5
1.1.2 Sistemas de Bases de Datos Distribuidas (SBDD) 5
1.1.3 Réplicas de Datos 6
1.1.4 Características de las Réplicas de Datos 6
1.1.5 Conflictos en la réplica de datos 7
1.2 Replicador de Datos Reko 8
1.3 Análisis de sistemas de réplica 12
1.3.1 SymetricDS 12
1.3.2 Oracle Streams 13
1.3.3 Daffodil Replicator 13
1.3.4 Slony-I 14
1.3.5 Microsoft SQL Server Developer Network (MSDN) 14
1.4 Resultados 15
1.5 Herramientas y lenguajes 16
1.5.1 Plataforma de desarrollo 16
1.5.2 Lenguaje de programación 16
1.5.3 Entornos de desarrollo 16
1.5.4 Frameworks (Marco de trabajo) 17
1.6 Metodología de desarrollo. 17
1.6.1 OpenUp 17
1.7 Consideraciones parciales. 19
CAPÍTULO 2. CARACTERÍSTICAS Y DISEÑO DEL SISTEMA. 20
2.1 Descripción del proceso a automatizar. 20
2.2 Modelo de Dominio. 21
2.2.1 Diagrama de Clases del Dominio. 21

2.2.2 Descripción de las Clases del Modelo de Dominio. 22
2.3 Modelo del Sistema. 22
2.3.1 Requisitos Funcionales. 22
2.3.2 Requisitos No Funcionales. 23
2.3.3 Definición de Casos de Uso del Sistema 23
2.3.4 Diagrama de Casos de Uso 23
2.3.5 Descripción de los Actores del Sistema 24
2.3.6 Descripción de los Casos de Uso del Sistema 24
2.4 Descripción de la arquitectura del Replicador de Datos Reko. 30
2.4.1 Patrones arquitectónicos. 31
2.4.2 Patrones de diseño. 32
2.5 Modelo de diseño. 34
2.5.1 Diagramas de Clases del Diseño. 34
2.5.2 Descripción de las clases. 36
2.5.3 Diagramas de Interacción de Diseño. 39
2.6 Modelo de despliegue. 42
2.7 Consideraciones parciales del capítulo. 42
CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS 43
3.1 Modelo de implementación. 43
3.1.1 Diagrama de componentes. 43
3.2 Código fuente. 44
3.3 Modelo de prueba. 47
3.3.1 Camino Básico. 48
3.3.2 Framework JUnit. 50
3.3.3 Pruebas de Sistema. 51
3.3.4 Conteo de Funcionalidad. 51
3.3.5 Eficiencia. 52
3.4 Consideraciones parciales del capítulo. 56
Conclusiones generales. 57

Recomendaciones 58
Referencias Bibliográficas 59

Índice de Tablas

TABLA 1 CARACTERÍSTICAS DE LOS REPLICADORES 15

TABLA 2 TIPOS DE SOLUCIONES EMPLEADAS PARA TRATAR LOS CONFLICTOS 15

TABLA 3: ACTORES DEL SISTEMA 24

TABLA 4: DESCRIPCIÓN DEL CU EVITAR, CAPTURAR Y CLASIFICAR LOS CONFLICTOS. 24

TABLA 5: DESCRIPCIÓN DEL CU RESOLVER DE FORMA AUTOMÁTICA LOS CONFLICTOS 25

TABLA 6: DESCRIPCIÓN DEL CU CONFIGURAR EL MÓDULO DE CONFLICTOS. 30

TABLA 7: DESCRIPCIÓN DE LA CLASE APPLICATORTRIGGERDAOIMPL. 36

TABLA 8: DESCRIPCIÓN DE LA CLASE RECEIVERMANAGER. 36

TABLA 9: DESCRIPCIÓN DE LA CLASE CONFLICTMANAGER. 37

TABLA 10: DESCRIPCIÓN DE LA CLASE CONFLICT. 38

TABLA 11: DESCRIPCIÓN DE LA CLASE CONFLICTREPLICABLEACTION. 38

TABLA 12: RESULTADOS DE LA PRUEBA DE CAJA BLANCA 48

TABLA 13: CASOS DE PRUEBA 49

TABLA 14: COMPARACIÓN DE FUNCIONALIDADES ENTRE LAS DOS VERSIONES DEL REPLICADOR DE DATOS REKO. 52

TABLA 15: RESOLUCIÓN AUTOMÁTICA Y VARIANTES. 55

Índice de Figuras

Figura 1: Principales componentes del Replicador de Datos Reko. Fuente: (1) 9
Figura 2: Diagrama de proceso del tratamiento de conflictos en el Replicador de Datos Reko. 11
Figura 3: Ciclo de vida definido por OpenUp 18
Figura 4: Diagrama de Clases del Dominio 21
Figura 5: Diagrama de Casos de Uso. 24
Figura 6: Vista de los principales componentes del Replicador de Datos Reko. 32
Figura 7: Diagrama de Clase de Diseño para el CU Evitar, capturar y clasificar los conflictos. 34
Figura 8: DC para el CU Resolver de forma automática los conflictos 35
Figura 9: DC para el CU Configurar el módulo de Conflictos. 36
Figura 10: DS para el CU Evitar, capturar y clasificar conflictos 39
Figura 11: DS para el CU Resolver de forma automática los conflictos de eliminación. 40
Figura 12: DS para el CU Resolver de forma automática los conflictos de unicidad. 40
Figura 13: DS para el CU Resolver de forma automática los conflictos de actualización. 41
Figura 14: DS para el CU Resolver de forma automática los conflictos de desconocidos. 41
Figura 15: Diagrama de Despliegue. 42
Figura 16: Diagrama de componentes para el subsistema Distribuidor de cambios. 43
Figura 17: Diagrama de componentes para el subsistema Aplicador de cambios 44
Figura 18: Clases de prueba. 50
Figura 19: Resultados de las pruebas unitarias 51
Figura 20: Escenario de réplica 52
Figura 21: Tabla de la BD2 53
Figura 22: Tabla de la BD1 53
Figura 23: Tabla de la BD2 después de la resolución de los conflictos. 54
Figura 24: Tabla de la BD1 después de la resolución de los conflictos. 54
Figura 25: Tabla de la BD1 después de la resolución de los conflictos. 54
Figura 26: Tabla de la BD2 después de la resolución de los conflictos. 54

INTRODUCCIÓN

Desde los orígenes de la humanidad, el crecimiento constante y acelerado de la información ha traído como consecuencia que con el paso del tiempo la forma tradicional de almacenarla y consultarla se convirtiera en una actividad demasiado costosa. Sin embargo, desde hace unas décadas, con el desarrollo de las Tecnologías de la Informática y las Comunicaciones (TIC), han surgido un conjunto de herramientas que permiten realizar este proceso de una forma menos engorrosa. Una de las tecnologías existentes son las Bases de Datos (BD), las cuales permiten guardar gran cúmulo de información de manera organizada para que luego se pueda encontrar y utilizar fácilmente. (1)

Originalmente la información se almacenaba de forma centralizada, pero con el transcurso del tiempo, la necesidad de acceder a los datos aumentó y a su vez surgieron inconvenientes, a los cuales no era posible dar una solución eficiente desde un modelo centralizado. Estos problemas dieron paso a la creación de los Sistemas de Bases de Datos Distribuidas (SBDD). Estos responden a la relación que existe entre diferentes BD las cuales se encuentran en ordenadores ubicados geográficamente distantes e interconectados por una red de comunicaciones. (2)

El uso de los sistemas distribuidos ha propiciado que las BD que lo conforman tengan la necesidad de mantener la actualización y sincronización de los datos entre ellas. Para lograr esto fue necesario desarrollar herramientas que permitieran el copiado y la distribución de objetos de una BD hacia otra, manteniendo la coherencia y consistencia de la información. Este proceso se dio a conocer como réplica de datos. (3)

Un sistema de réplica debe cubrir las principales necesidades relacionadas con la distribución de datos como: protección, recuperación, sincronización de datos, transferencia de datos entre diversas localizaciones y centralización de la información en una única localización. Las soluciones robustas en esta temática son privativas y por tanto muy costosas, mientras las existentes en software libre están incompletas y son muy difíciles de configurar. (4)

El Replicador de Datos Reko es un software que surge en la Universidad de la Ciencias Informáticas (UCI) como solución a las necesidades de réplica del proyecto Sistema de Gestión Penitenciario Venezolano (SIGEP). El desarrollo de esta herramienta comenzó en el año 2007 y en la actualidad se lleva a cabo en el Centro de Consultoría y Desarrollo de Arquitecturas Empresariales (CDAE) para una continua expansión y mejora. El Replicador de Datos Reko tiene como objetivo brindar un software multiplataforma que le permita al usuario, con el menor esfuerzo, cubrir las necesidades fundamentales de replicación.

Durante el despliegue de la herramienta en disímiles escenarios se detectaron conflictos relacionados fundamentalmente a la inconsistencia en la información existente en el ambiente de réplica. Se define como conflicto en un proceso de réplica cualquier irregularidad originada en la

aplicación de la información que impide que la transacción pueda ser aplicada, deteniendo completamente el proceso de réplica hasta que se solucione. (5)

Para este tipo de situaciones el Replicador de Datos Reko cuenta con un módulo de resolución de conflictos que los clasifica en tres tipos: conflictos de unicidad, conflictos de eliminación y conflictos desconocidos. Además cuenta con mecanismos que permiten la captura, resolución automática y resolución manual de los conflictos que se genera durante el proceso de réplica. El módulo descrito presenta las siguientes limitantes:

- Mecanismo de captura: Durante la detección de conflictos solo diferencia los conflictos de unicidad y de eliminación y los demás los clasifica como conflictos desconocidos para el Replicador de Datos Reko.
- Mecanismo de resolución automática: Solo permite realizar dos acciones automáticas sobre los conflictos de unicidad y eliminación que se detectan:
  1- Ignorar los conflictos detectados.
  2- En presencia de un conflicto de unicidad permite convertir la acción de inserción que llega en una acción de actualización.

El módulo de resolución de conflictos antes expuesto no garantiza que el mecanismo de resolución automática resuelva correctamente los conflictos de unicidad para distintos escenarios y no asegura la consistencia de los datos. La mayor responsabilidad de esta tarea recae en el mecanismo de resolución manual. Esto trae como consecuencia una posible afectación al tiempo de respuesta del sistema debido a que depende de la intervención directa del usuario para resolver el conflicto.

Lo expuesto anteriormente constituye la situación problemática de la presente investigación, a partir de la cual se pretende resolver el siguiente problema:

¿Cómo resolver de manera eficiente los conflictos que ocurren durante el proceso de réplica en el Replicador de Datos Reko, para mantener la consistencia de los datos?

A partir del problema enunciado se define como **objeto de estudio:** el proceso de réplica en los SBDD.

Todo lo anterior precisa como **campo de acción:** el proceso de detección y resolución de conflictos durante la réplica de datos.

Para dar solución al problema planteado se propone el siguiente **objetivo general:** Desarrollar un módulo de resolución automática de conflictos en el Replicador de Datos Reko, para mantener la consistencia de los datos.

El objetivo ha sido desglosado en los siguientes **objetivos específicos:**

1. Analizar los principales conceptos relacionados con la detección y solución de los conflictos de réplica en SBDD.
2. Implementar un módulo para la detección y solución de forma automática de los conflictos de réplica para el Replicador de Datos Reko.
3. Validar el módulo obtenido a través de pruebas que evalúen la solidez del código y la eficiencia con respecto a la versión anterior.

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

1. Elaboración del marco teórico para precisar los principales conceptos que se emplean en el proceso de réplica de datos.
2. Estudio del estado del arte sobre los sistemas de réplicas que presentan propuestas para la resolución de los conflictos durante el proceso de réplica de datos.
3. Realización del análisis y diseño del componente para la obtención de los artefactos del sistema, como son: los diagramas de casos de uso, clases y componentes, con sus respectivas descripciones.
4. Implementación de los componentes para la solución automática de los conflictos de unicidad, eliminación, actualización y desconocidos para el Replicador de Datos Reko.
5. Implementación del componente para la configuración de la gestión de conflictos.
6. Realización de pruebas en entornos de producción que generen conflictos pre-analizados para validar el producto implementado.

Los métodos utilizados en la presente investigación se desglosan a continuación:

- **Métodos teóricos:**
  - Analítico-Sintético: permitió que a través del estudio de las necesidades y problemas originados en determinados proyectos, se analizaran y sintetizaran las ideas fundamentales para mejorar la solución existente.
  - Histórico-Lógico: a partir de un estudio de la evolución y uso de los sistemas de réplicas se fundan las bases necesarias para definir la solución definitiva.
- **Métodos empíricos:**
  - Observación: permitió detectar las necesidades existentes y posibles mejoras que se le pueden añadir al sistema en desarrollo, con el propósito de extender las funcionalidades del mismo a una aplicación mejor estructurada.

De esta investigación se espera como **posible resultado** un módulo automático de detección y resolución de conflictos para el Replicador de Datos Reko que permita la solución automatizada de los conflictos que se generen durante el proceso de réplica.

El presente documento consta de tres capítulos, así como conclusiones, recomendaciones y referencias bibliográficas.

El **Capítulo 1. Fundamentación teórica de los conflictos de réplica.** Presenta un análisis del estado del arte sobre replicadores de datos. Se destacan las tendencias y tecnologías actuales sobre las cuales se apoya la propuesta. Se describen la metodología de desarrollo de software, herramientas y lenguajes a emplear en el desarrollo de la solución.

El **Capítulo 2. Características y diseño del sistema.** Incluye la descripción, diseño y análisis de la solución propuesta para darle respuesta a la problemática planteada. Se especifican los requisitos funcionales y los no funcionales. Se realiza el modelo de casos de uso. Se describen los estilos arquitectónicos y patrones de diseño aplicados. Se describen los diagramas de clases de diseño y secuencia y el modelo de despliegue.

El **Capítulo 3. Implementación y pruebas.** Muestra la implementación del módulo. En él se analizan los estilos de codificación y se brinda una solución a los requisitos especificados. Se describen los diagramas de componentes, se muestra el código fuente de las principales clases y se aplican pruebas al módulo para demostrar su eficiencia.

CAPÍTULO 1. FUNDAMENTACIÓN TEÓRICA DE LOS CONFLICTOS DE RÉPLICA

En el presente capítulo se precisan un conjunto de conceptos, definiciones y fundamentos que componen el marco teórico relacionado con el objeto de estudio definido en la investigación. Se destacan las principales características de los elementos asociados a la temática de la investigación así como las principales metodologías y herramientas que servirán de apoyo para la búsqueda de la solución a la problemática planteada.

1.1 Marco conceptual

1.1.1 Bases de Datos (BD)

Existe un gran cúmulo de definiciones sobre qué es una BD. De acuerdo a algunos autores pueden ser:

Rosa M. Mato declara que es un conjunto de datos interrelacionados, almacenados con carácter más o menos permanente en la computadora, o sea, una colección de datos variables en el tiempo. (6)

El autor Manuel Sierra[1] concluye que una BD es un sistema informático a modo de almacén, donde se guardan grandes volúmenes de información. Permite automatizar el acceso a la información de manera rápida y fácil, además de poder realizar cambios de una manera más eficiente. (1)

Sergio Ezequiel en el libro Bases de datos y su aplicación con SQL conceptualiza una BD como el lugar donde se guardan los datos en reposo (persistentes) y al cual acceden las diferentes aplicaciones. (7)

Para la investigación se consideró que una BD es un conjunto de datos almacenados, variables en el tiempo, relacionados entre sí y utilizados por aplicaciones externas.

1.1.2 Sistemas de Bases de Datos Distribuidas (SBDD)

M. Tamer Özsu y Patrick Valduriez definen un SBDD como una colección de múltiples BD lógicamente relacionadas entre sí y distribuidas en una red informática. (2)

El autor Jofman Pérez considera que un sistema de bases de datos distribuidas es aquel en el cual múltiples sitios de bases de datos están ligados por un sistema de comunicaciones, de tal forma que un usuario en cualquier sitio puede acceder a los datos en cualquier parte como si estuvieran en su propio sitio. (5)

Las definiciones expuestas se consideran válidas y acordes con los objetivos del trabajo y se asume para el desarrollo de la investigación el concepto enunciado por M. Tamer Özsu y Patrick Valduriez.

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

1.1.3 Réplicas de Datos

Conceptos enunciados por varios autores:

La replicación ayuda a garantizar la coherencia de los datos, los cuales, distribuidos entre los diferentes nodos de la red, deben estar sincronizados correctamente para garantizar la consistencia de la información, lo que implica el copiado y mantenimiento de datos de un lugar (el nodo en el que la manipulación de los datos se llevó a cabo) a los otros nodos en la red que se van a actualizar. (8)

La replicación es el proceso de compartir información entre bases de datos (o cualquier otro tipo de servidor) para asegurar que el contenido es coherente entre sistemas. (9)

La replicación es un conjunto de tecnologías destinadas a la copia y distribución de información y objetos entre bases de datos, para mantener su sincronización y coherencia. (3)

La replicación es el proceso de copiar y mantener objetos en varias bases de datos que componen un sistema de BD distribuida. Los cambios aplicados en un sitio se capturan y se almacenan de forma local antes de ser expedido y aplicado en cada uno de las ubicaciones remotas. (10)

El autor concluye con el último concepto expuesto agregándole que durante todo el proceso se debe mantener la coherencia entre las bases de datos.

1.1.4 Características de las Réplicas de Datos

Propósitos de la replicación:

- **Disponibilidad**: los datos son accesibles desde varios sitios, aun cuando algunos sitios han cerrado.
- **Rendimiento**: la replicación permite localizar los datos más cerca de los puntos de acceso de los usuarios reduciendo los tiempos de respuestas.

La replicación de datos se clasifica teniendo en cuenta tres elementos fundamentales:

1- Ambientes de replicación:
   - **Maestro-esclavo** (*master-slave*): en este caso la replicación se realiza en un solo sentido: desde un nodo maestro a uno o varios esclavos.
   - **Multi-maestro** (*multi-master*): se configuran varios nodos como maestros, permitiendo que ocurra la replicación en todos los sentidos.
2- Forma de transmitir los cambios en el entorno de replicación:
   - **Sincrónica**: los cambios ejecutados en un nodo maestro son aplicados instantáneamente en el nodo origen y en los nodos destinos dentro de una misma transacción. En caso de no poder ejecutarse la acción en cualquiera de los nodos, la transacción completa es cancelada. Requiere una alta disponibilidad de recursos de red.

- **Asincrónica**: los cambios ejecutados en una tabla son almacenados y enviados posteriormente al resto de los nodos del entorno cada ciertos intervalos de tiempo.

3- Forma de capturar y almacenar los cambios a replicar:

- **Basada en *triggers***[2]: 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***[3]: se sostiene en la lectura de *logs* de cambios que proporcionan algunos gestores como Oracle.

La réplica de datos puede ocurrir en dos tipos de entornos diferentes:

- **Entorno homogéneo**: los procesos de réplicas ocurren en los diferentes ambientes que usan el mismo gestor de bases de datos.
- **Entorno heterogéneo**: la réplica heterogénea es aquella que es implementada sobre gestores de bases de datos diferentes, por ejemplo: bases de datos funcionando con MySql y otra con Postgres. (5)

Un buen software de replicación de datos debe cubrir las principales necesidades relacionadas con la distribución de datos entre los gestores más populares como:

- Protección.
- Recuperación.
- Sincronización de datos.
- Transferencia de datos entre diversas localizaciones.
- Centralización de la información en una única localización. (4)

1.1.5 Conflictos en la réplica de datos

Se entiende como conflicto durante la réplica de datos cualquier irregularidad originada en el proceso de aplicación de la información que impide que la transacción pueda ser aplicada. Los conflictos normalmente son asociados a inconsistencia en la información existente en un ambiente con relación al otro. (5)

Tipos de conflictos que se han detectado y afectan la integridad de la información:

- **Conflicto de unicidad**: sucede cuando la replicación de un registro intenta violar una restricción de integridad, ya sea por llave primaria o única (*Primary Key* o *Unique*). Por ejemplo, cuando dos transacciones originadas de dos sitios diferentes, cada una inserta un

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

[3] *Log* es un registro oficial de eventos durante un rango de tiempo en particular.

registro en su respectiva tabla replicada con el mismo valor de clave primaria. En ese caso sucede un conflicto de unicidad.

- **Conflicto de eliminación**: un conflicto de eliminación ocurre cuando una transacción intenta borrar o actualizar un registro que no existe.
- **Conflicto de actualización**: este conflicto ocurre cuando dos transacciones originadas desde distintos sitios actualizan el mismo registro, en forma cercana en el tiempo. (10)

La detección de conflictos es el acto de determinar si una inserción, actualización o eliminación genera algún problema con los datos objetivos a modificar. Si esto ocurre estamos en presencia de un conflicto. La resolución de conflicto es el acto de precisar qué hacer cuando se detecta un conflicto. (11)

Cuando se diseñan SBDD se deben tener en cuenta todos los requisitos de aplicación antes de construir la BD. Ejecutar esta actividad permite evitar en gran medida la posibilidad de conflictos de réplicas, para esto existen algunas técnicas como:

- **Propietario Primario de BD** *(Primary Database Ownership)*: permite evitar la posibilidad de conflictos limitando el número de BD en el sistema que tendrán acceso a las actualizaciones simultáneas de las tablas con datos compartidos. Este modelo previene los conflictos porque solo una BD permite las actualizaciones a un conjunto de datos compartidos a la vez.
- **Propietario Compartido de BD** *(Shared Database Ownership)*: se recomienda utilizar cuando el modelo anterior sea muy restrictivo para los requerimientos de la aplicación. Este modelo presupone que pueden ocurrir conflictos para los cuales se recomiendan aplicar estrategias que los eviten como las que se plantean a continuación:
    - Evitar conflictos de unicidad: se pueden evitar los conflictos de unicidad asegurando que cada BD usa identificadores únicos para los datos compartidos.
    - Evitar conflictos de eliminación: en general las aplicaciones que operan con el modelo *Shared Database Ownership* no deben borrar registros usando la sentencia *DELETE*. En su lugar deberían marcar los registros que se deben borrar y configurar el sistema para purgarlos periódicamente.
    - Evitar conflictos de actualización: en un modelo *Shared Database Ownership* no es posible evitar este tipo de conflictos en su totalidad. Por eso se deben identificar los tipos de conflictos posibles y configurar el sistema para que los resuelva si estos ocurren. (12)

1.2 Replicador de Datos Reko

El Replicador de Datos Reko ha sido implementado en la plataforma *Java Enterpise Edition* (JEE). Esta plataforma provee una arquitectura robusta para el desarrollo de aplicaciones empresariales en

el lenguaje Java utilizando un modelo multicapas e incluye una serie de APIs (*Application Programming Interface*), tecnologías, herramientas de desarrollo e implementaciones de referencia de los servicios que brinda.

A continuación se muestran los componentes fundamentales de la herramienta, así como una descripción de los mismos.

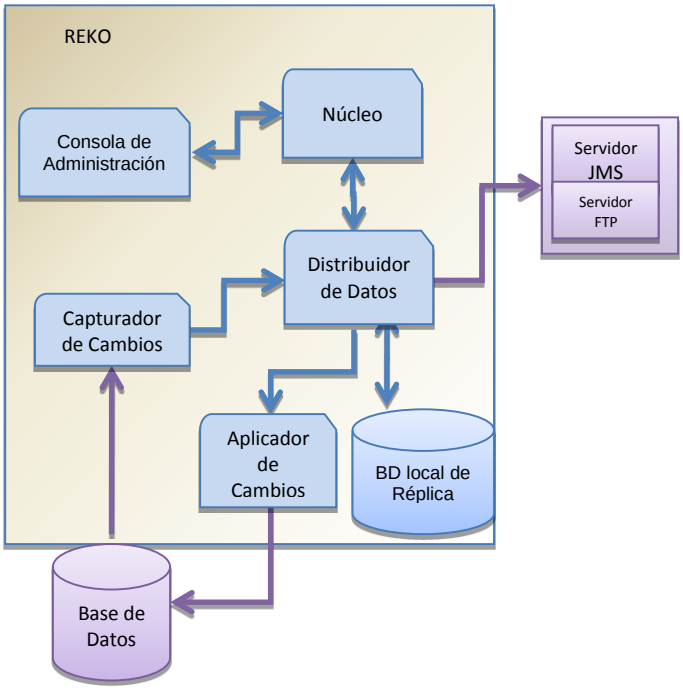


Figura 1: Principales componentes del Replicador de Datos Reko.
Fuente: (1)

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 (*File Transfer Protocol*): 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 BD 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.

Principales funcionalidades que brinda el Replicador de Datos Reko:

- Soporte para réplica de datos en ambientes con conexión y sin conexión.
- Soporte para réplica entre bases de datos con diferentes estructuras.
- Soporte para réplica de ficheros externos a la BD.
- Soporte para la sincronización de datos en ambientes con y sin conexión.
- Selección de los datos de réplica ajustada por filtros.
- Soporte para réplica de datos entre gestores diferentes.
- Monitoreo en tiempo real de los datos de réplica.
- Soporte para programación del momento de captura y envío de los Datos de Réplica.
- Soporte para resolución de conflictos. (4)

Analizando detenidamente el soporte para resolución de conflictos, se determinó que el módulo de conflictos es capaz de clasificar los conflictos en tres tipos fundamentales:

- Conflictos de unicidad: estos ocurren cuando se intenta aplicar una acción de inserción en la BD y ya existe un juego de datos con la misma llave primaria.
- Conflicto de eliminación: ocurre cuando se aplica una acción de eliminación y no existe ninguna tupla con la misma llave primaria.
- Conflictos desconocidos: contiene todos los conflictos que no se puedan agrupar en ninguno de los antes mencionados.

Este módulo cuenta con mecanismos que permiten la captura, la resolución automática y resolución manual de los conflictos que se genera durante el proceso de réplica. Los funcionamientos para el proceso que realiza el módulo son:

- Mecanismo de captura: interviene cuando se están aplicando los datos en la BD, es capaz de capturar y clasificar los conflictos que se generan durante el proceso de réplica.
- Mecanismo de resolución automática: una vez detectado los conflictos permite aplicar soluciones automatizadas para resolver el conflicto y luego aplicar nuevamente en la BD.

- Mecanismo de resolución manual: cuando se detectan conflictos que no se les pueden aplicar soluciones automatizadas, en este caso todos los conflictos desconocidos, el usuario de forma manual debe resolver el conflicto detectado a través de una interfaz web que le permita crear la acción que resolverá el conflicto.

La herramienta cuando detecta un conflicto detiene el proceso de réplica completamente y hasta que no se solucione, ya sea automática o manualmente, no se reanuda la réplica de datos. A continuación se muestra el diagrama de proceso del tratamiento de los conflictos en el Replicador de Datos Reko:

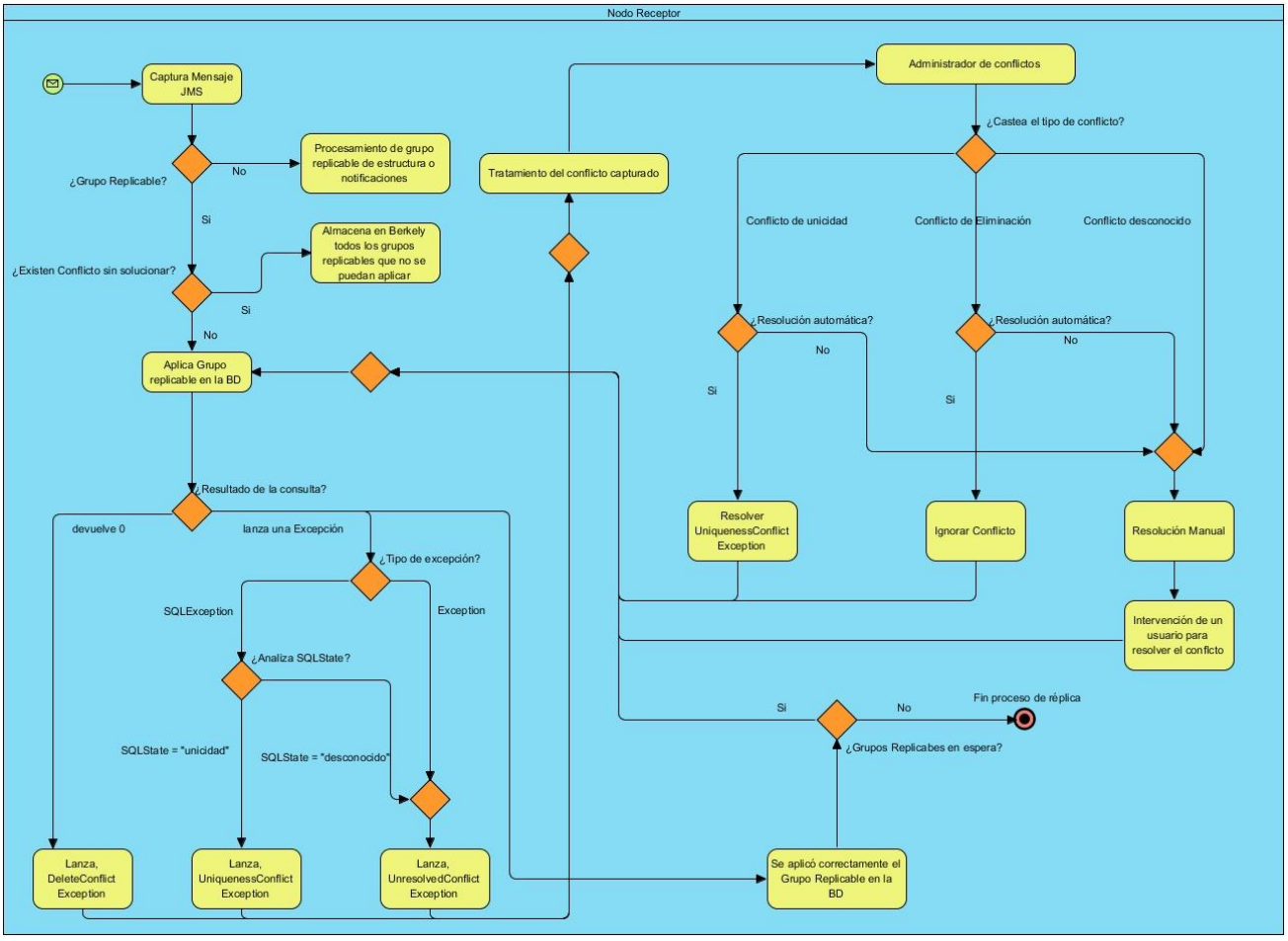


Figura 2: Diagrama de proceso del tratamiento de conflictos en el Replicador de Datos Reko.

Aunque el Replicador de Datos Reko brinda un conjunto de funcionalidades en esta temática presenta serias limitantes las cuales se desglosan en (13):

- Mecanismo de captura: durante la detección de conflictos solo diferencia los conflictos de unicidad y de eliminación y los demás los clasifica como conflictos desconocidos para el Replicador de Datos Reko.
- Mecanismo de resolución automática: solo permite realizar dos acciones automáticas sobre los conflictos que se detectan:

1- Ignorar los conflictos detectados.
2- En presencia de un conflicto de unicidad permite convertir la acción de inserción que generó el conflicto al aplicar en la BD destino en una acción de actualización.

1.3 Análisis de sistemas de réplica

Para un mejor resultado de las funcionalidades a desarrollar, se procedió a un estudio de las herramientas existentes en el ámbito de la réplica de datos. Estas herramientas fueron seleccionadas para su análisis debido fundamentalmente a su capacidad de detectar y solucionar los conflictos que se generan durante la réplica de datos, capacidad de réplica en distintos escenarios y entre diferentes gestores de BD. También se tuvieron en cuenta características necesarias que debe tener un buen software de réplica de datos.

1.3.1 SymetricDS

SymetricDS es un software libre bajo los términos LGPL[4] desarrollado en la plataforma JEE. Entre sus características fundamentales se encuentran:

- Soporte para la sincronización bidireccional de BD.
- Soporte para la replicación multi-maestra.
- Soporte para la replicación asíncrona.
- Soporte para réplica de datos entre gestores diferentes.
- Soporte para la administración remota.
- Soporte para la detección y resolución de conflictos (incorporada en la versión 3.0).
- Soporte para la sincronización de ficheros (incorporada en la versión 3.5). (11)

El funcionamiento de soporte para la detección y resolución de conflictos está basado en tres aspectos fundamentales:

- El usuario en la configuración puede previamente definir cómo debe actuar el sistema frente a distintos tipos de conflictos.
- En caso de que se detecte un conflicto y el usuario no haya definido previamente una solución, el sistema automáticamente, dependiendo del tipo de acción: inserción, actualización o eliminación, lo resuelve de manera predeterminada o lo ignora.
- Resolución manual: en caso que durante la réplica de dato se generen conflictos el usuario manualmente se debe obtener una solución a través de una interfaz. (14)

[4] Licencia Pública General Reducida de Linux, del inglés *Lesser General Public License*. Esta licencia permite que los software que usen (sin modificar) bibliotecas o componentes licenciados con LGPL, puedan ser liberados bajo otras licencias relacionadas o no con el software libre.

1.3.2 Oracle Streams

Oracle Streams es un sistema propietario de réplica avanzada que proporciona una infraestructura flexible que cumple con una amplia variedad de las necesidades de intercambio de información. Entre sus características fundamentales se encuentran: replicación bidireccional y unidireccional, tratamiento de conflictos, captura de datos basado en *logs* y transformación de datos definidos por el usuario.

Para realizar la réplica se basa en tres componentes:

- Captura: los cambios por acciones Lenguaje de Definición de Datos (DDL) o Lenguaje de Manipulación de Datos (DML) son capturados del registro de rehacer y luego son empaquetados en LCR[5]. Los datos y los eventos pueden ser cambiados o formateados por un conjunto predefinido de reglas antes de ser empaquetados en un LCR.
- Puesta en escena: los LCR se almacenan en el entorno de ensayo hasta que un abonado los recoge para ser utilizado o consumido. El abonado puede ser otro entorno de ensayo o una aplicación de usuario.
- Consumo: durante el consumo, los LCR se recogen y se aplican en una BD. Se modifican los LCR antes de ser aplicados en la BD.

Oracle Streams establece mecanismos de detección de conflictos que se generan durante la replicación e incorpora rutinas para la resolución automática de potenciales conflictos. Además, permite que los usuarios creen sus propias rutinas empleando reglas que satisfagan las necesidades de su negocio. Los conflictos que no se puedan filtrar por los mecanismos antes mencionados se almacenan en registros para un posterior tratamiento manual. (15)

1.3.3 Daffodil Replicator

Es una herramienta de código abierto desarrollada en Java para la integración, migración y protección de datos en tiempo real. Permite bidireccionar datos de replicación y sincronización entre bases de datos homogéneas y heterogéneas. Presenta la capacidad de detectar y permitir la resolución de conflictos, es independiente de plataforma, permite sincronización bidireccional, soporte para la replicación de datos de gran tamaño y réplica entre bases de datos con estructuras iguales.

Daffodil Replicator es capaz de detectar y resolver los conflictos que ocurren entre el publicador y el suscriptor durante la réplica de datos. Clasifica los conflictos en tres tipos: unicidad, eliminación y

[5] Unidad básica de Captura de Cambios, del inglés *Logical Change Records*

actualización. La forma de solución depende de quién tiene la información confiable si el publicador o el suscriptor y dónde se detecta el conflicto. (16, 17)

1.3.4 Slony-I

Es un software de replicación maestro-esclavo, que tiene la capacidad de replicar grandes bases de datos. Slony-I es un sistema para centros de datos y sistemas de seguridad, su modo normal de operación implica que todos los nodos deben estar disponibles todo el tiempo. Ofrece replicación asíncrona usando disparadores para recolectar los cambios ocurridos en la BD.

Las principales desventajas de Slony-I es que solo soporta el servidor de BD PostgreSql, replica solo entre bases de datos con estructuras iguales y no tiene la capacidad de replicar objetos de gran tamaño. Para la resolución de conflictos establece que se deben resolver automáticamente según las políticas que están establecidas y en caso que no exista una política para un conflicto determinado se detiene el proceso de réplica hasta que manualmente se resuelva el problema. (18, 19)

1.3.5 Microsoft SQL Server Developer Network (MSDN)

Es un software que está limitado bajo licencia, además, como es una herramienta de Microsoft SQL Server, la replicación está definida únicamente para sistemas operativos de Microsoft. MSDN cuenta con tres tipos de replicación:

- Replicación de instantáneas: no se realiza un seguimiento de los cambios de datos; cada vez que se aplica una instantánea, esta sobrescribe completamente los datos existentes.
- Replicación transaccional: realiza un seguimiento de los cambios a través del registro de transacciones de SQL Server.
- Replicación de mezcla: realiza un seguimiento de los cambios a través de desencadenadores y tablas de metadatos.

Se analizó el tratamiento a los conflictos que se generan durante la replicación de mezcla ya que es la que más se adapta a la forma de réplica del Replicador de Datos Reko.

El comportamiento de la detección y resolución de conflictos depende de las opciones que brinda MSDN para este caso y según la elección se determina qué puede ser un conflicto y qué tipo de solución se puede aplicar. La replicación de mezcla ofrece solucionadores automáticos que se clasifican en 4 tipos (solucionador de conflictos predeterminado basado en prioridad, controlador de lógica de negocios, solucionador personalizado basado en COM, solucionador basado en COM proporcionado por Microsoft) y un solucionador interactivo que proporciona una interfaz de usuario donde permite elegir manualmente la forma en que se solucionan los conflictos en tiempo de ejecución. (20 - 22)

1.4 Resultados

Se concluye a partir del estudio realizado que hay replicadores que soportan la resolución de conflictos de forma automática pero no tienen el mismo alcance. Esto se pone de manifiesto en dos formas fundamentales:

- Resolución automática básica: contiene solamente la resolución y detección de los conflictos más conocidos (conflictos de unicidad, eliminación, actualización) y solo brinda una o dos variantes para la solución de estos, lo que constituye una limitante debido a que un mismo conflicto puede tener diferentes soluciones según el escenario en que se presente.
- Resolución automática avanzada: contiene la solución básica pero con más variantes para la resolución de los conflictos y además permite al usuario elegir acciones automatizadas previamente o durante la ejecución del proceso de réplica para resolver determinados tipo de conflicto.

Al concluir el análisis del Replicador de Datos Reko y sus homólogos se realiza una comparación entre ellos, como se muestran en las tablas 1 y 2.

Tabla 1 Características de los replicadores

Características/ Herramientas	**Reko**	**SymetricDS**	**Oracle Streams**	**Daffodil Replicator**	**Slony-I**	**MSDN**
Ambientes de replicación soportados						
Maestro-esclavo					X	
Multi-Maestro	X	X	X	X		X
Gestores soportados						
PostgresSQL	X	X		X	X	
MySQL	X	X				
Oracle	X	X	X	X		
Microsoft SQL Server	X	X		X		X
Otros gestores		X		X		

Fuente: elaboración propia a partir de (1,11-22)

Tabla 2 Tipos de soluciones empleadas para tratar los conflictos

Características/ Herramientas	Reko	SymetricDS	Oracle Streams	Daffodil Replicator	Slony-I	MSDN
Resolución manual	X	X	X		X	X

Resolución automática básica	X	X	X	X	X	X
Resolución automática avanzada		X				X

Fuente: elaboración propia a partir de (1,11-22)

1.5 Herramientas y lenguajes

El software Replicador de Datos Reko desde sus inicios siempre ha utilizado herramientas y lenguajes de código abierto para su implementación. Siguiendo este principio y con el objetivo de mantener un estándar en el desarrollo se decide utilizar las mismas herramientas y lenguajes en las que fue desarrollado. A continuación se hace una breve descripción para un mejor entendimiento.

1.5.1 Plataforma de desarrollo

JEE (*Java Enterprise Edition*): es un entorno independiente de la plataforma centrado en desarrollar aplicaciones empresariales basadas en la web. Consta de un conjunto de características que proveen las funcionalidades necesarias para desarrollar aplicaciones basadas en web. JEE simplifica el desarrollo de este tipo de aplicaciones, basándolas en componentes modulares y estandarizados, y proporcionando un conjunto de especificaciones que aseguran la portabilidad de las aplicaciones. (23)

1.5.2 Lenguaje de programación

Java: es un lenguaje de uso general, orientado a objeto, concurrente y uno de los más populares a nivel mundial, particularmente para el desarrollo de aplicaciones cliente-servidor en la web. Está diseñado para tener tan pocas dependencias de implementación como se pueda y su intención es permitir que una vez escrito el programa por los desarrolladores se pueda ejecutar en cualquier plataforma. (24)

1.5.3 Entornos de desarrollo

Eclipse STS (*Spring Tool Suite*): es un potente y completo entorno de desarrollo para construir aplicaciones con Spring debido a que incluye los componentes y las funcionalidades necesarias para potenciar el desarrollo web orientado a una arquitectura empresarial. Es una herramienta libre desarrollada por Spring Source. (25)

Apache ActiveMQ: el más popular y potente servidor de mensajería de código abierto implementado en Java. ActiveMQ es rápido y fue diseñado para ser compatible con muchos protocolos y lenguajes que consuman sus servicios tales como: Java, C++, C#, entre otros. (26)

Apache Tomcat: es un software de código libre con soporte para las tecnologías Java Servlets y JSP (*Java Server Pages*). Está desarrollado en un entorno abierto y participativo. (27)

1.5.4 Frameworks (Marco de trabajo)

SpringMVC: SpringMVC (*Model-View-Controller*) Modelo-Vista-Controlador, construido sobre el núcleo de Spring. Este *framework* es altamente configurable a través de interfaces y permite el uso de múltiples tecnologías para la capa vista, entre las que se encuentran JSP. Potenciado el desarrollo web orientado a arquitecturas empresariales. (28)

Hibernate: es un *framework* gratuito, de código abierto. Se encarga del mapeo de clases de Java a tablas de la BD y la generación de *consultas* a la BD. Hibernate facilita la migración de sistemas entre diferentes motores de bases de datos. Reduce aproximadamente el 95% de las tareas que un programador tenía que hacer para realizar funciones comunes de acceso a datos. Ofrece un marco de trabajo fácil de usar para mapear un modelo orientado a objetos a una tradicional BD relacional. Hibernate soporta la mayoría de los sistemas de bases de datos SQL. Brinda facilidades para recuperación y actualización de datos, control de transacciones, repositorios de conexiones a bases de datos, consultas programáticas y declarativas, y un control de relaciones de entidades declarativas. (29)

Dojo: es un *framework* de código abierto desarrollado en JavaScript destinado a facilitar el rápido desarrollo de JavaScript o de las aplicaciones basadas en Ajax y sitios web. Su idea es la de abstraer al desarrollador de las complejidades del HTML y de las discrepancias existentes entre navegadores, que hacen que el código JavaScript a utilizar sea diferente. Potencia el desarrollo del lado del cliente en una aplicación web. (30)

1.6 Metodología de desarrollo.

1.6.1 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 de RUP 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.

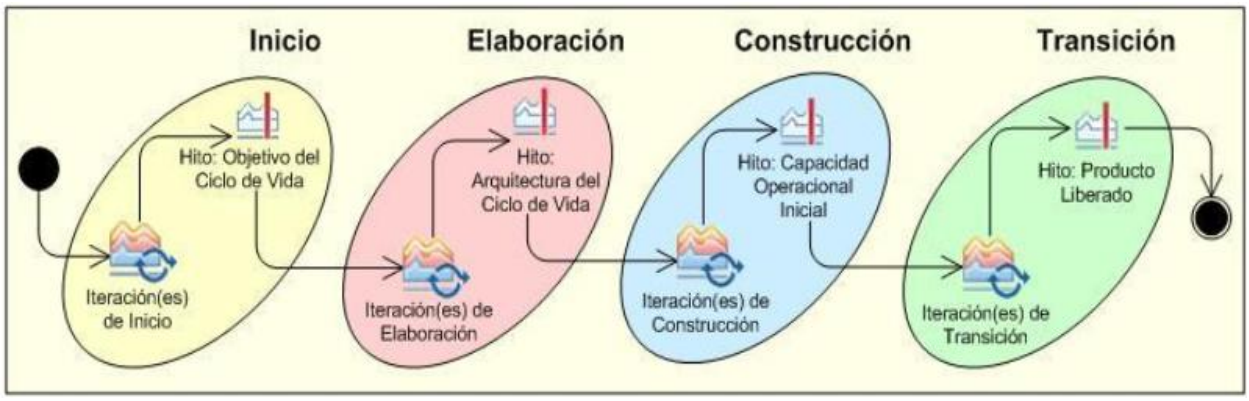


Figura 3: Ciclo de vida definido por OpenUp

En el mismo, 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, etc. Se puede utilizar 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*[6] 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.

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 *stakeholders*, desarrolladores, arquitectos, gestor de proyecto y probadores. El equipo toma sus propias decisiones sobre que se requiere realizar, cuáles

[6] 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»

son las prioridades y cómo abordar los requerimientos y necesidades de los *stakeholders*. Junto con la colaboración intensa la presencia de los interesados en el equipo es esencial para el éxito. (31)

1.7 Consideraciones parciales.

A partir del análisis realizado en este capítulo se concluye que:

- La solución actual que brinda el Replicador de Datos Reko para la resolución automática de los conflictos es mínima y por tanto no cubre todas las necesidades sobre esta temática, dándole la mayor responsabilidad de la resolución de los conflictos al mecanismo manual, el cual depende de la intervención del usuario, es muy costosa en tiempo y propicia errores por parte del usuario, trayendo como consecuencia que demore el proceso de réplica.
- Para la solución del problema no es viable la utilización total o parcial de las herramientas de réplica existentes, pero en el caso de SymetricDS se puede utilizar como referencia la forma en que maneja los conflictos para el desarrollo del módulo propuesto.
- Debido a que el desarrollo de la solución se va a desplegar sobre la arquitectura del Replicador de Datos Reko se decide utilizar las mismas herramientas, lenguajes y metodología que actualmente tiene definido el proyecto, con el objetivo de mantener un estándar de trabajo.

CAPÍTULO 2. CARACTERÍSTICAS Y DISEÑO DEL SISTEMA.

En este capítulo se especifican los requisitos funcionales y los no funcionales, se realiza el modelo de casos de uso, se describen los estilos arquitectónicos y patrones de diseño aplicados, se describen los diagramas de clases del diseño, secuencia y el modelo de despliegue. Esto se realizará tomando como referencia los conceptos y términos definidos en el primer capítulo, y regido por la metodología OpenUp.

2.1 Descripción del proceso a automatizar.

Actualmente el Replicador de Datos Reko permite realizar configuraciones de réplica, enviar, recibir y sincronizar los datos en ambientes con conexión y sin conexión, utilizando los gestores de bases de datos Oracle, MySQL, PostgreSQL y SQLServer. Durante el proceso de réplica en el momento de aplicar los cambios sobre la BD destino pueden ocurrir conflictos debido a la inconsistencia de la información. Para estos casos el sistema no es capaz de gestionar automáticamente todos los conflictos que ocurren.

Una vez desarrollado el módulo de resolución automática de conflictos se podrá brindar variantes de soluciones automatizadas para los distintos tipos de conflictos que puedan ocurrir durante el proceso de réplica, aumentando considerablemente la eficiencia del software. Dichas variantes establecen soluciones que están enfocadas a la prioridad de la información que se establezca en cada nodo a partir de la etiqueta de nodo ganador o no ganador. Quedando definida de la siguiente forma:

Prioridad de la información: se define en cada nodo si es ganador o no. En dependencia, si es ganador entonces la información que llega y entra en conflicto no tiene prioridad sobre la existente en el nodo que recibe. En caso contrario, siendo no ganador, la información que tiene prioridad es la que llega. Cualquier cambio sobre la información del nodo que recibe (no ganador) es viable.

Las variantes para cada tipo de conflictos quedan establecidas de la siguiente forma:

Variantes para cuando el **nodo** se comporta como **ganador** (datos locales con prioridad):

Variantes para resolución de **conflictos de unicidad:**

1. **Nuevo id**: Insertar datos que llegan con un nuevo id generado a partir del rango definido. Al nodo remoto se envía una actualización de los datos sin alterar ya que es el mismo id pero la información es distinta y los datos con el nuevo id se envían como una inserción (Opcional).
2. **Ignorar datos:** Ignorar datos que llegan. Enviar actualización de datos mantenidos al nodo remoto para mantener consistencia (Opcional).

Variantes para resolución de **conflictos de actualización:**

1. **Insertar datos:** Insertar datos que llegan como nueva tupla.
2. **Ignorar datos:** Ignorar datos que llegan. Eliminar datos en el nodo remoto (Opcional).

Variantes para resolución de **conflictos de eliminación:**

1. **Ignorar datos:** Ignorar la acción de eliminación que llega.

Variantes para cuando el **nodo** se comporta como **no ganador** (datos locales sin prioridad):

Variantes para resolución de **conflictos unicidad:**

1. **Nuevo id**: Actualizar datos que llegan sobre la tupla en conflicto y la información de la tupla en conflicto se inserta con un nuevo id generado a partir del rango definido. Al nodo remoto se envía una acción de inserción con la información de la tupla en conflicto con el nuevo id generado (Opcional).
2. **Actualizar datos:** Actualizar datos que llegan sobre la tupla en conflicto.

Variantes para resolución de **conflictos de actualización:**

1. **Insertar datos:** Insertar datos que llegan como nueva tupla.

Variantes para resolución de **conflictos de eliminación:**

1. **Ignorar datos:** Ignorar la acción de eliminación que llega.

En el caso específico de los **conflictos desconocidos** se desarrolla un mecanismo que permita a partir de la primera ocurrencia de un conflicto desconocido automatizar la acción seleccionada manualmente para resolver el conflicto, asegurando que cuando ocurra nuevamente este tipo de conflicto se le aplica la acción automatizada elegida inicialmente.

2.2 Modelo de Dominio.

Un modelo de dominio captura los tipos más importantes de objetos y tiene como objetivo agrupar y describir las clases más importantes dentro del contexto del sistema, ayudando a establecer un vocabulario común entre los usuarios, clientes, desarrolladores y *stakeholders*. (32)

2.2.1 Diagrama de Clases del Dominio.

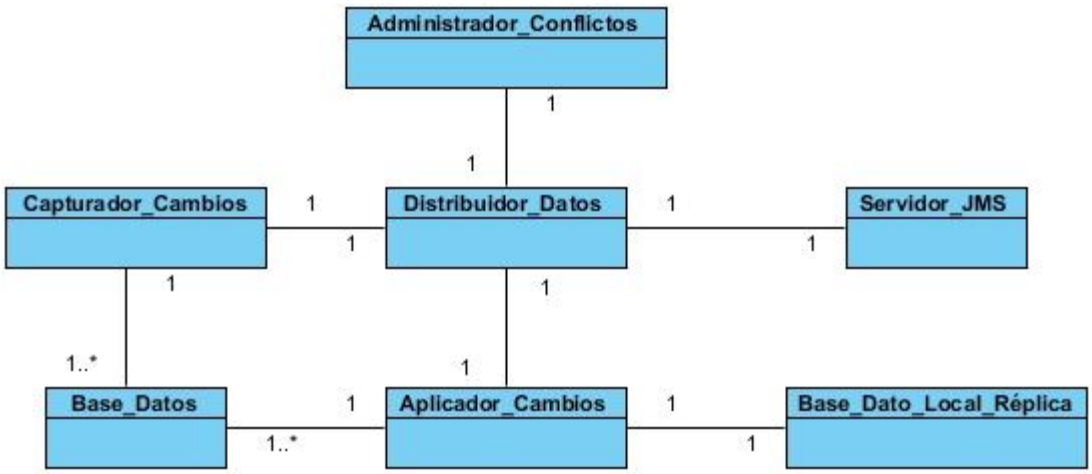


Figura 4: Diagrama de Clases del Dominio

2.2.2 Descripción de las Clases del Modelo de Dominio.

Administrador_Conflictos: soluciona y almacena los conflictos que ocurran durante el proceso de réplica dependiendo de la configuración y el tipo de conflicto.

Capturador_Cambios: captura los cambios que se realizan sobre la BD y se los entrega al distribuidor de datos.

Distribuidor_Datos: determina el destino del cambio realizado en la BD, los envía y se responsabiliza de su llegada.

Aplicador_Cambios: ejecuta sobre la Base_Datos los cambios que sean replicados desde otro nodo además de capturar y clasificar los conflictos que ocurran.

Servidor_JMS: servidor de mensajería bajo la especificación *Java Message Service*, es utilizado como punto intermedio en la distribución de la información enviada bajo JMS.

Base_Datos_Local_Réplica: concepto que guardará las configuraciones propias de la réplica, así como acciones sobre la Base_Datos que han provocado conflictos al aplicarse y transacciones que no han llegado a su destino.

Base_Datos: concepto que representa la BD que se está replicando. Los cambios ejecutados sobre ella serán enviados así como serán aplicados otros cambios provenientes de otros nodos de réplica.

2.3 Modelo del Sistema.

Un modelo del sistema describe lo que supuestamente hará un sistema, pero no cómo implementarlo. Idealmente, una representación de un sistema debería mantener toda la información sobre la entidad que se está representando. (32)

2.3.1 Requisitos Funcionales.

Los requisitos funcionales de un sistema describen las capacidades o funciones que el sistema debe cumplir (33). Los requisitos funcionales identificados se describen a continuación:

RF1: Capturar conflictos.

- **RF1.1:** Evitar conflictos.
- **RF1.2:** Clasificar los conflictos detectados.
- **RF1.3:** Detener proceso de réplica.

RF2: Resolver automáticamente los conflictos de unicidad.

RF3: Resolver automáticamente los conflictos de actualización.

RF4: Resolver automáticamente los conflictos de eliminación.

RF5: Resolver automáticamente los conflictos desconocidos.

RF6: Configurar el módulo de resolución automática de conflictos.

2.3.2 Requisitos No Funcionales.

Los requerimientos no funcionales definen las restricciones del sistema, son propiedades o cualidades que el sistema debe poseer. Representan las características del producto. Los requisitos no funcionales del componente a desarrollar son equivalentes a los del software al que se integra, Replicador de Datos Reko. (34)

- Fiabilidad: debe ser capaz de recuperarse ante fallos como errores en el proceso de transformación provocados por configuraciones incorrectas, falta de fluido eléctrico y fallos en la red garantizando la integridad de los datos que se transforman.
- Rendimiento: debe estar concebido para el consumo mínimo de recursos. Los clientes no necesitarán más de 128MB de RAM, lo suficiente para ejecutar un navegador Web.
- Soporte: el diseño del componente tendrá en cuenta patrones GOF y GRASP para garantizar la escalabilidad del sistema.
- Implementación: se implementará utilizando la plataforma JEE y las tecnologías asociadas que se emplean en Replicador de Datos Reko.
- Los recursos de hardware mínimos para el funcionamiento son:
    - 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, 100MB de disco duro y procesador 2.0 GHz como mínimo.
- Software: se requiere para el funcionamiento del sistema disponer de un servidor JMS, un servidor de BD PostgreSQL y navegador Mozilla 3.0 o superior. Se brinda soporte a 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.
- Seguridad: los nodos de replicación manejan credenciales entre ellos para verificar la autenticidad de los datos transferidos. El envío de datos se realiza utilizando protocolos de comunicación como TCP o SSL.
- Legales: las herramientas y componentes empleados en el desarrollo deben ser libres.

2.3.3 Definición de Casos de Uso del Sistema

Los casos de uso proporcionan uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico. (32)

2.3.4 Diagrama de Casos de Uso

Los diagramas de casos de uso son utilizados para especificar la relación entre los actores y los casos de uso en un sistema. Son usados además para ilustrar los requerimientos del sistema.

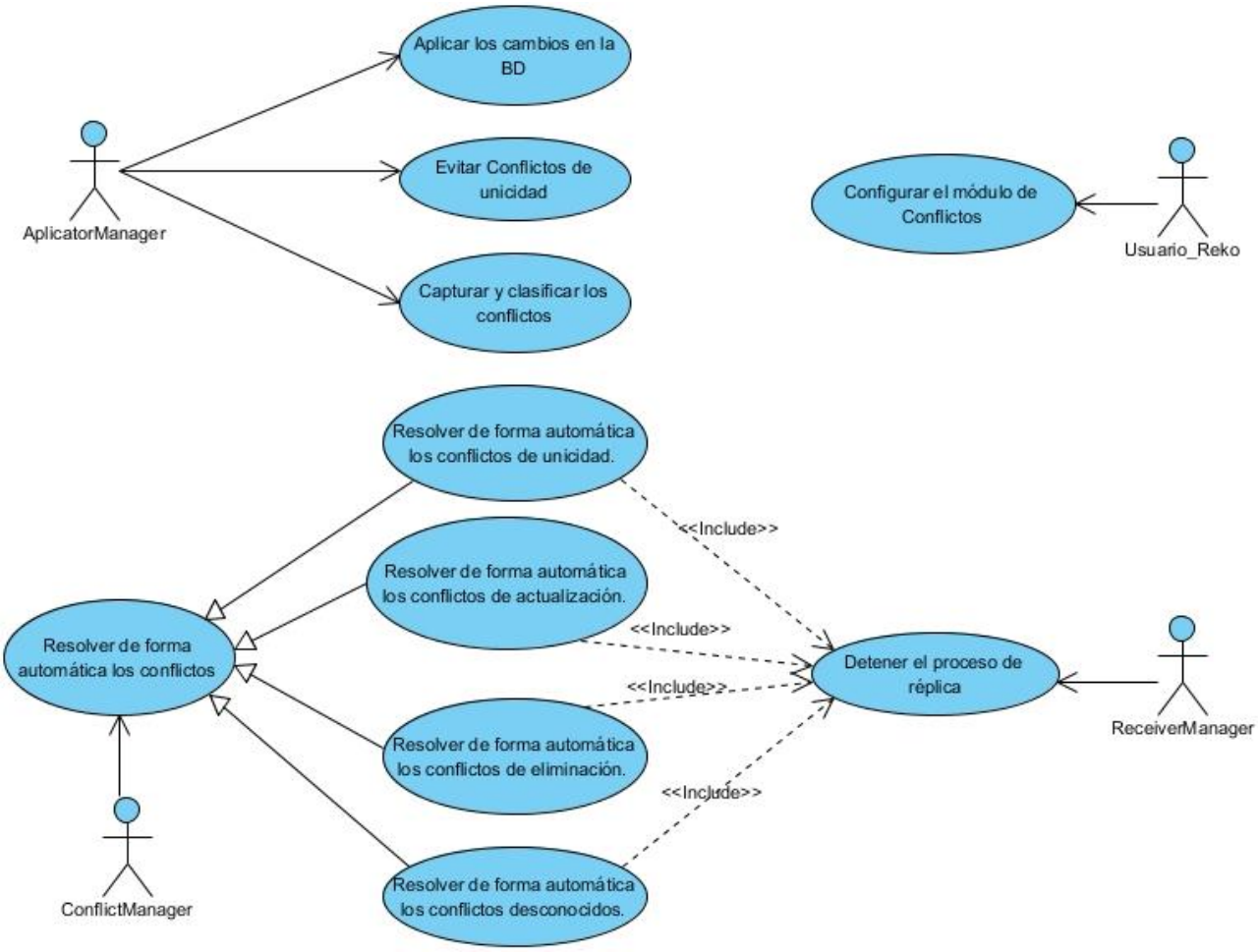


Figura 5: Diagrama de Casos de Uso.

2.3.5 Descripción de los Actores del Sistema

Tabla 3: Actores del Sistema

Actores	Descripción
AplicatorManager	Responsable de ejecutar los cambios realizados sobre la BD en los nodos destinos.
ConflictManager	Responsable de resolver los conflictos que ocurren durante el proceso de réplica.
Usuario_Reko	Responsable de configurar el módulo de resolución de conflictos de forma automática.
ReceiverManager	Responsable de recibir el grupo de estructuras replicables y detener o reanudar el proceso de réplica.

2.3.6 Descripción de los Casos de Uso del Sistema

Tabla 4: Descripción del CU Evitar, capturar y clasificar los conflictos.

Caso de Uso:	Evitar, capturar y clasificar los conflictos
Actores:	Clase AplicatorManger
Resumen:	El caso de uso inicia cuando la clase AplicatorManager intenta aplicar un grupo replicable sobre la BD.
Precondiciones:	Debe estar abierta la conexión a la BD.

Referencias:	RF1, RF1.1, RF1.3, RF1.4.
Prioridad:	Crítico.
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema
1. La clase AplicatorManager intenta aplicar un grupo replicable sobre la BD. 8. La clase ReceiverManager detiene el proceso de réplica.	2. Obtiene las acciones replicables contenidas en la agrupación. 3. Si la acción replicable es de tipo INSERT, crea una consulta a partir del identificador para comprobar si en la BD existe alguna tupla con el mismo identificador, si es el caso entonces se está en presencia de un conflicto de unicidad y se comprueba si la información es diferente para la acción y la tupla existente. Si esto se cumple se continúa con el flujo normal. 4. Se construye a partir de la acción replicable la consulta con su respectivo dialecto y se ejecuta sobre la BD. 5. Si el evento anterior genera algún tipo de excepción entonces se está en presencia de un conflicto. 6. Se clasifica el conflicto según el tipo de excepción. 7. Se lanza la excepción. 9. Se envía el grupo replicable en conflicto a la clase ConflictManager.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
	3.1 Si la información es exactamente igual se descarta la acción replicable para no aplicarla en la BD y evitar que se genere un conflicto de unicidad. 5.1 Si el evento no genera ninguna excepción, para los casos que las acciones son de tipo UPDATE y DELETE se comprueba la cantidad de filas afectadas cuando se aplica en la BD. Si el resultado es 0 entonces se está en presencia de un conflicto, si no, significa que se aplicó correctamente la acción sobre la BD.
Poscondiciones:	Se ha capturado y clasificado un conflicto.

Tabla 5: Descripción del CU Resolver de forma automática los conflictos

Caso de Uso:	Resolver de forma automática los conflictos.
Actores:	Clase ConflictManager, clase ReceiverManager
Resumen:	El caso de uso comienza cuando la clase ConflictManager recibe un conflicto.
Precondiciones:	En la configuración del módulo de conflictos deben activarse las soluciones automáticas sobre los diferentes conflictos.
Referencias:	RF2, RF3, RF4, RF5.
Prioridad:	Crítico.
Sección: Resolver de forma automática los conflictos de unicidad.	
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema

1. La clase ConflictManager recibe un conflicto de unicidad. 2. Si en la configuración está activado la solución automática para este tipo de conflicto procede al flujo automático.	3. A partir de la configuración del módulo selecciona la variante de nueva llave primaria para un nodo ganador. 4. Se genera una nueva llave primaria para resolver la acción replicable en conflicto. 5. Se construye una acción replicable de inserción con la nueva llave primaria y una de actualización con los datos de la BD y se encapsula en un grupo replicable. 6. Se envía al nodo remoto el grupo replicable con los cambios a aplicar para mantener la consistencia de los datos (Depende de la configuración). 7. Se devuelve una lista con las acciones que resuelven el conflicto en el nodo local.
8. Descarta la acción en conflicto por las nuevas acciones sobre el grupo replicable. 9. La clase AplicatorManager intenta aplicar nuevamente el grupo replicable sobre la BD. 10. La clase ReceiverManager reanuda el proceso de réplica.	11. Aplica satisfactoriamente el grupo replicable sobre la BD.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
2.1 Si en la configuración está activado la solución manual para este tipo de conflicto se procede al flujo manual. 3.1 Busca en Berkeley si existe para este conflicto una acción automatizada. 5.1 Almacena en Berkeley el conflicto para que el usuario posteriormente lo resuelva manualmente. 6.1 La clase ReceiverManager mantiene detenido el proceso de réplica hasta que el usuario lo resuelva manualmente.	4.1 No encuentra ninguna coincidencia.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema

7.2 Descarta la acción en conflicto.	3.2 A partir de la configuración del módulo selecciona la variante de ignorar datos que llegan para un nodo ganador. 4.2 Se construye una acción replicable de actualización con los datos de la BD y se encapsula en un grupo replicable. 5.2 Se envía al nodo remoto el grupo replicable con los cambios a aplicar para mantener la consistencia de los datos (Depende de la configuración). 6.2 Devuelve una lista vacía.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
	3.3 A partir de la configuración del módulo, selecciona la variante de nueva llave primaria para un nodo no ganador. 4.3 Se construye una acción replicable de actualización con los datos que llegan y se genera una nueva llave primaria para la información de la BD. 5.3 Se envía al nodo remoto la acción replicable de inserción construida encapsulada en un grupo replicable con los cambios a aplicar para mantener la consistencia de los datos (Depende de la configuración). 6.3 Devuelve una lista con las acciones que resuelven el conflicto en el nodo local.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
	3.4 A partir de la configuración del módulo, selecciona la variante de nueva llave primaria para un nodo no ganador. 4.4 Se construye una acción replicable de actualización con los datos que llegan. 5.4 Devuelve una lista con las acciones que resuelven el conflicto en el nodo local.
Sección: Resolver de forma automática los conflictos de actualización.	
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema
1. La clase ConflictManager recibe un conflicto de actualización. 2. Si en la configuración está activado la solución automática para este tipo de conflicto procede al flujo automático.	3. A partir de la configuración del módulo selecciona la variante de insertar datos para un nodo ganador. 4. Se construye una acción replicable de inserción con los datos que llegan. 5. Devuelve una lista con las acciones que resuelven el conflicto en el nodo local.

6. Descarta la acción en conflicto por las nuevas acciones sobre el grupo replicable. 7. La clase AplicatorManager intenta aplicar nuevamente el grupo replicable sobre la BD. 8. La clase ReceiverManager reanuda el proceso de réplica.	9. Aplica satisfactoriamente el grupo replicable sobre la BD.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
2.1 Si en la configuración está activada la solución manual para este tipo de conflicto se procede al flujo manual. 3.1 Busca en Berkeley si existe para este conflicto una acción automatizada. 5.1 Almacena en Berkeley el conflicto para que el usuario posteriormente lo resuelva manualmente. 6.1 La clase ReceiverManager mantiene detenido el proceso de réplica hasta que el usuario lo resuelva manualmente.	4.1 No encuentra ninguna coincidencia.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
7.2 Descarta la acción en conflicto.	3.2 A partir de la configuración del módulo selecciona la variante de ignorar datos que llegan para un nodo ganador. 4.2 Se construye una acción replicable de eliminación y se encapsula en un grupo replicable. 5.2 Se envía al nodo remoto el grupo replicable con los cambios a aplicar para mantener la consistencia de los datos (Depende de la configuración). 6.2 Devuelve una lista vacía.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
	3.3 A partir de la configuración del módulo selecciona la variante de insertar datos para un nodo ganador. 4.3 Se construye una acción replicable de inserción con los datos que llegan. 5.3 Devuelve una lista con las acciones que resuelven el conflicto en el nodo local.
Sección: Resolver de forma automática los conflictos de eliminación.	
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema

1. La clase ConflictManager recibe un conflicto de eliminación. 2. Si en la configuración está activado la solución automática para este tipo de conflicto procede al flujo automático. 3. Descarta la acción en conflicto por las nuevas acciones sobre el grupo replicable. 4. La clase AplicatorManager intenta aplicar nuevamente el grupo replicable sobre la BD. 5. La clase ReciverManager reanuda el proceso de réplica.	6. Aplica satisfactoriamente el grupo replicable sobre la BD.
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
2.1 Si en la configuración está activado la solución manual para este tipo de conflicto se procede al flujo manual. 3.1 Busca en Berkeley si existe para este conflicto una acción automatizada. 5.1 Almacena en Berkeley el conflicto para que el usuario posteriormente lo resuelva manualmente. 6.1 La clase ReciverManager mantiene detenido el proceso de réplica hasta que el usuario lo resuelva manualmente.	4.1 No encuentra ninguna coincidencia.
Sección: Resolver de forma automática los conflictos desconocidos.	
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema
1. La clase ConflictManager recibe un conflicto desconocido. 2. Busca en Berkeley si existe para este conflicto una acción automatizada. 6. Descarta la acción en conflicto por las nuevas acciones sobre el grupo replicable. 7. La clase AplicatorManager intenta aplicar nuevamente el grupo replicable sobre la BD.	3. Encuentra una coincidencia. 4. Aplica la acción automatizada para resolver el conflicto. 5 Devuelve una lista con las acciones que resuelven el conflicto en el nodo local. 8. Aplica satisfactoriamente el grupo replicable sobre la BD.

9. La clase ReciverManager mantiene detenido el proceso de réplica hasta que el usuario lo resuelva manualmente.	
Flujos Alternos	
Acción del Actor	Respuesta del Sistema
4.1 Almacena en Berkeley el conflicto para que el usuario posteriormente lo resuelva manualmente. 5.1 La clase ReciverManager mantiene detenido el proceso de réplica hasta que el usuario lo resuelva manualmente.	3.1 No encuentra ninguna coincidencia.
Poscondiciones:	Se ha resuelto el conflicto.

Tabla 6: Descripción del CU Configurar el módulo de Conflictos.

Caso de Uso:	Configurar el módulo de Conflictos.
Actores:	Usuario_Reko
Resumen:	El caso de uso inicia cuando el Usuario_Reko accede a la sección de conflicto en la configuración general.
Precondiciones:	El Usuario_Reko debe estar autenticado en el sistema.
Referencias:	RF6
Prioridad:	Crítico.
Flujo Normal de Eventos	
Acción del Actor	Respuesta del Sistema
1. El Usuario_Reko accede a la sección de conflictos en la configuración general 3. El Usuario_Reko selecciona las opciones según las necesidades de su lógica de negocio.	2. El sistema despliega la interfaz de usuario en la sección de conflictos de la configuración general. 4. El sistema almacena la configuración en el archivo replication.properties.
Poscondiciones:	Se ha configurado el módulo de conflictos.

2.4 Descripción de la arquitectura del Replicador de Datos Reko.

La IEEE[7] Estándar 1471-2000 emitió la definición oficial de Arquitectura del Software y expone que: “Es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución”. (35)

[7] Del inglés: Institute of Electrical and Electronics Engineers.

La Arquitectura del Software es el diseño de más alto nivel de la estructura de un sistema y se selecciona y diseña en base a los requerimientos del sistema; consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de trabajo a seguir.

Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de sistemas software. Existen varios tipos de patrones, dependiendo del nivel de abstracción, del contexto particular en el cual aplican o de la etapa en proceso de desarrollo. Algunos de estos tipos son:

- Patrones arquitectónicos: son esquemas fundamentales de organización de un sistema software y especifican una serie de subsistemas y sus responsabilidades respectivas e incluyen las reglas y criterios para organizar las relaciones existentes entre ellos.
- Patrones de diseño: son patrones de un nivel de abstracción menor que los patrones de arquitectura. Están por lo tanto más próximos a lo que sería el código fuente final.
- Idiomas: patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto. (36)

2.4.1 Patrones arquitectónicos.

La arquitectura basada en componentes se enfoca en la descomposición del diseño en componentes funcionales o lógicos que expongan interfaces de comunicación bien definidas. El desarrollo basado en componentes es utilizado en aplicaciones que pueden descomponerse en componentes lógicos o funcionales. A su vez los componentes de software son unidades reutilizables que pueden relacionarse con otros módulos de software por medio de interfaces bien definidas que contienen métodos, eventos y propiedades, y pueden estar escritos en lenguaje funcional, procedural u orientado a objetos. En la especificación UML se define que: "…un componente es una unidad modular con interfaces bien definidas, que es reemplazable dentro del contexto". (37)

El Replicador de Datos Reko está centrado en una arquitectura basada en componentes, debido a que todas las funcionalidades pueden ser encapsuladas en un conjunto de comportamientos que pueden ser reemplazados por otros. Los principales componentes presentes en el software son:

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

Aplicador: ejecuta en la BD 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.

Independientemente 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 otra 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.

El Distribuidor, el Aplicador y la Administración son los componentes que han sido extendidos para el desarrollo del módulo propuesto. Para el tratamiento de los conflictos que ocurren durante el proceso de réplica se hace necesaria la integración con el componente Distribuidor y el Aplicador. En el paquete **conflicts**, correspondiente al componente Distribuidor, se realizarán modificaciones, pues hasta el momento su funcionamiento se basa fundamentalmente en una resolución manual y una mínima cobertura al tratamiento automático de los conflictos, demorando el proceso de réplica. Con esta propuesta de solución se extiende su funcionamiento para una cobertura total al tratamiento automático de los conflictos aumentando considerablemente la eficiencia del sistema.

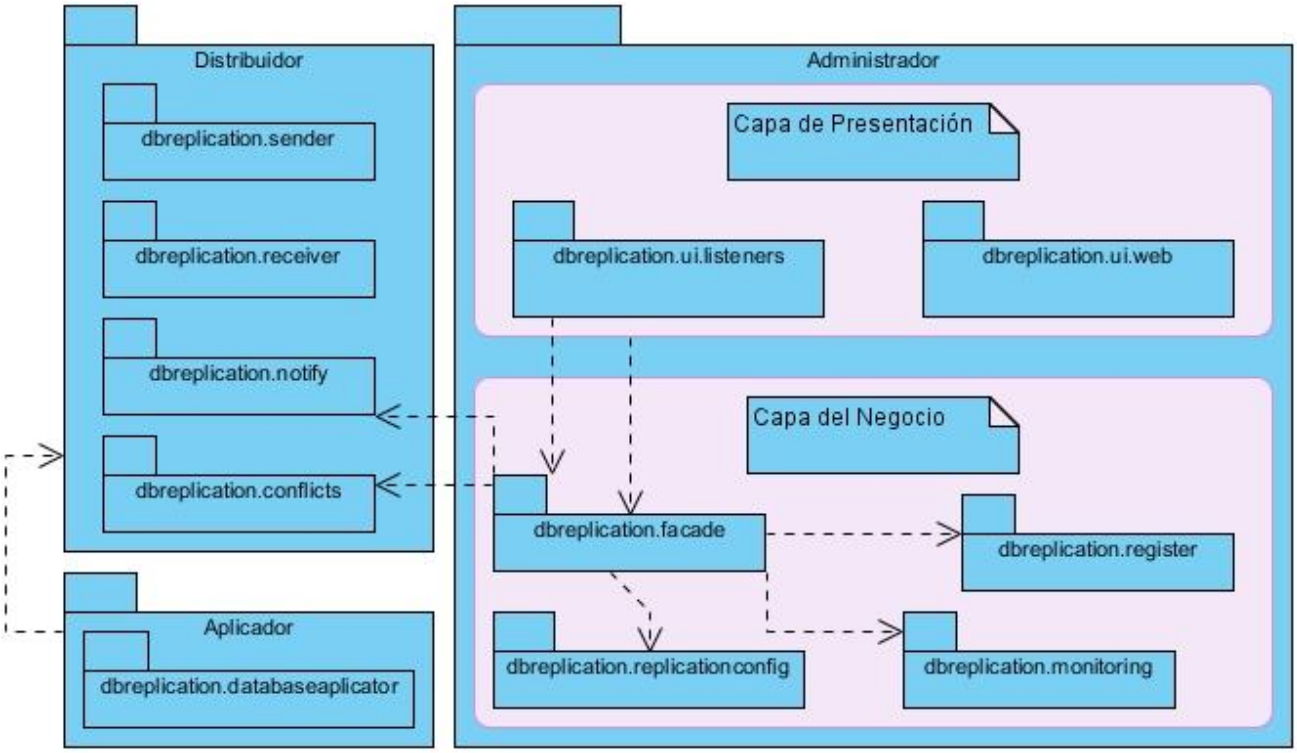


Figura 6: Vista de los principales componentes del Replicador de Datos Reko.

2.4.2 Patrones de diseño.

Los patrones de diseños brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares.

Patrón GRASP[8]

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 considerados

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

buenas prácticas en el diseño de software. 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.

Experto: es el principio básico de asignación de responsabilidades, por tanto la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo.

Creador: el patrón creador nos ayuda a identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases. De manera general, una clase B debería ser responsable 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 ConflictManager, la cual contiene toda la información pertinente a las ConflictReplicableAction y se encarga de gestionarlas.

Controlador: es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado. Ejemplo de este patrón serían las clases ConflictManager, ReceiverManager y AplicatorManager.

Alta Cohesión: es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Los elementos con baja cohesión son más difíciles de comprender, reutilizar, mantener y cambiar.

Bajo Acoplamiento: se relaciona con el patrón de alta cohesión. Es un patrón evaluativo que dicta cómo 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 aquellos para los cuales esta variación ocurre. Esto se logra mediante el uso de operaciones polimórficas. El uso de este patrón se manifiesta en los tipos de conflictos.

Patrones GOF[9]

Patrón Fachada: proporcionar una interfaz unificada de alto nivel que, representando a todo un subsistema, facilite su uso. La "fachada" satisface a la mayoría de los clientes, sin ocultar las funciones de menor nivel a aquellos que necesiten acceder a ellas. El Replicador de Datos Reko utiliza este patrón para ofrecer un punto de acceso para la configuración del módulo de conflicto a través de la clase ReplicationFacade. (36)

2.5 Modelo de diseño.

El modelo de diseño es un modelo de objetos que describe la realización física 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. (32)

2.5.1 Diagramas de Clases del Diseño.

Las clases de diseños representan abstracciones de los subsistemas y componentes de la implementación del sistema, además son directas y representan una sencilla relación entre el diseño y la implementación. (37)

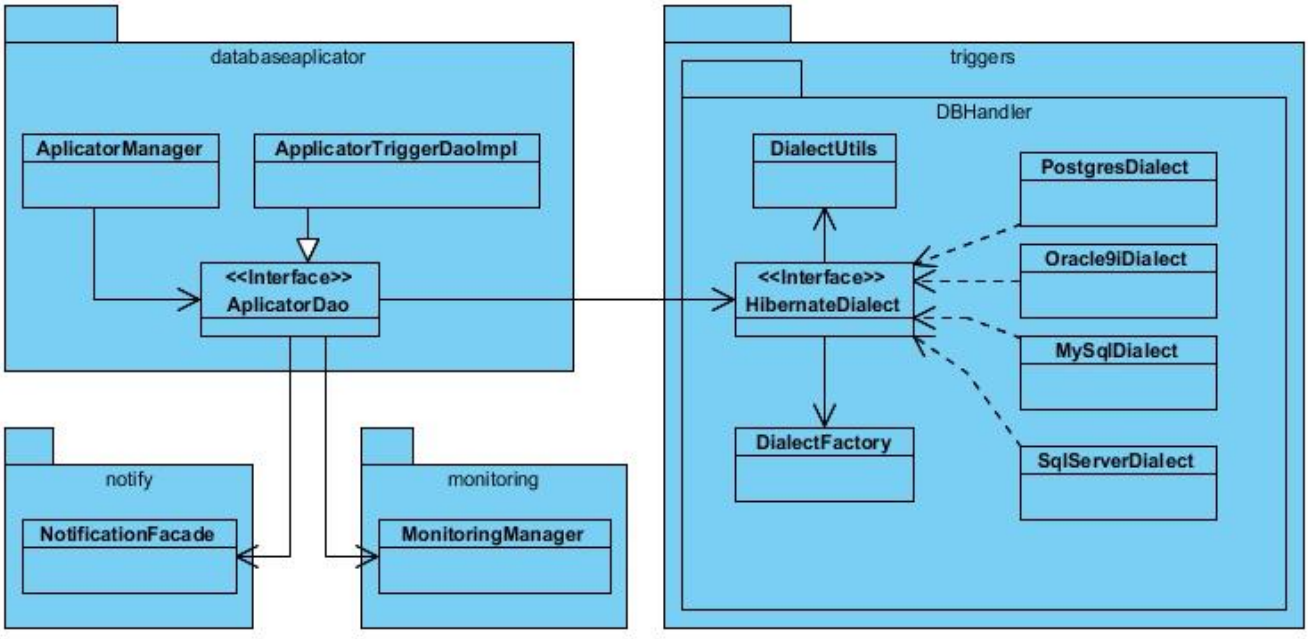


Figura 7: Diagrama de Clase de Diseño para el CU Evitar, capturar y clasificar los conflictos.

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

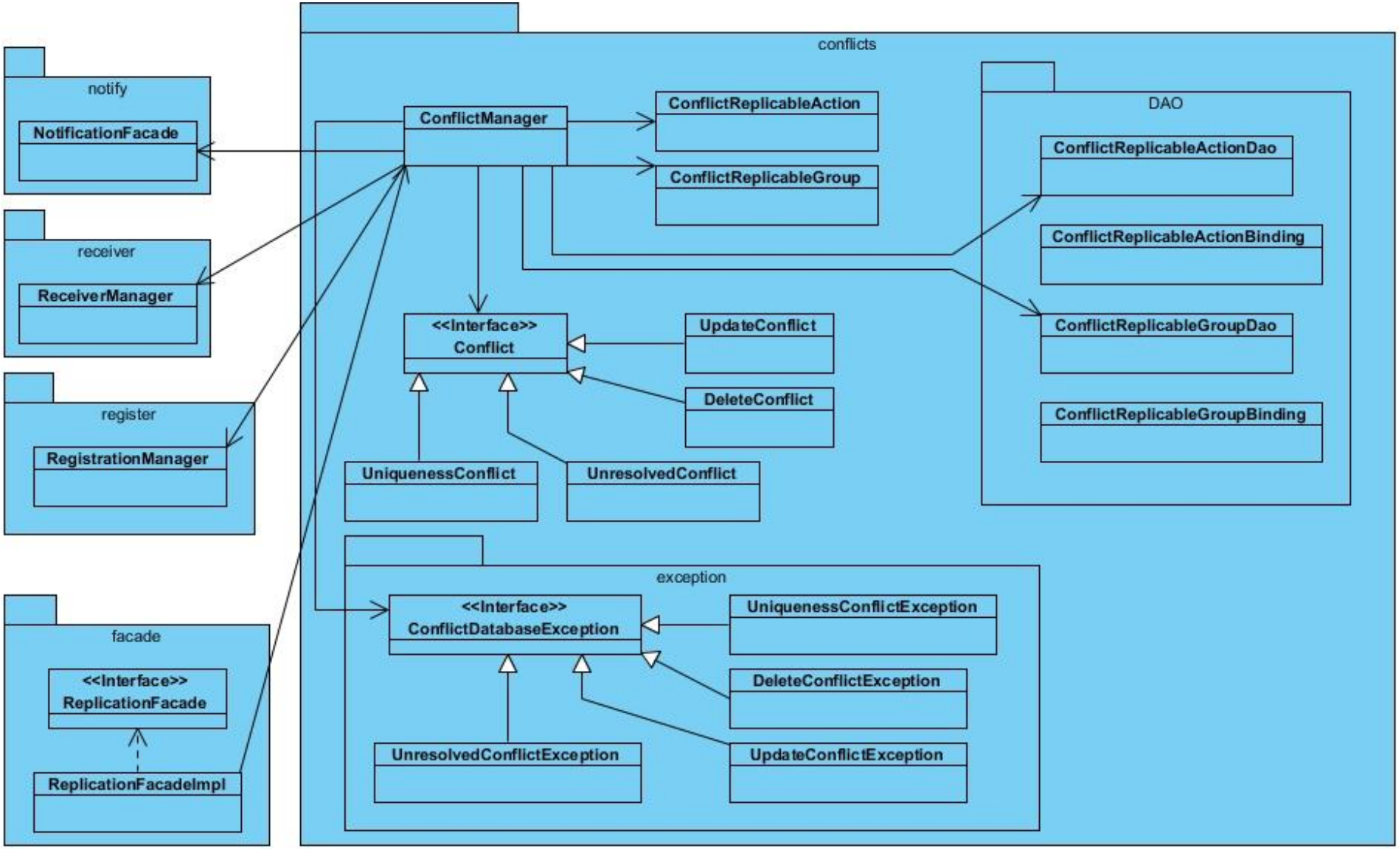


Figura 8: DC para el CU Resolver de forma automática los conflictos

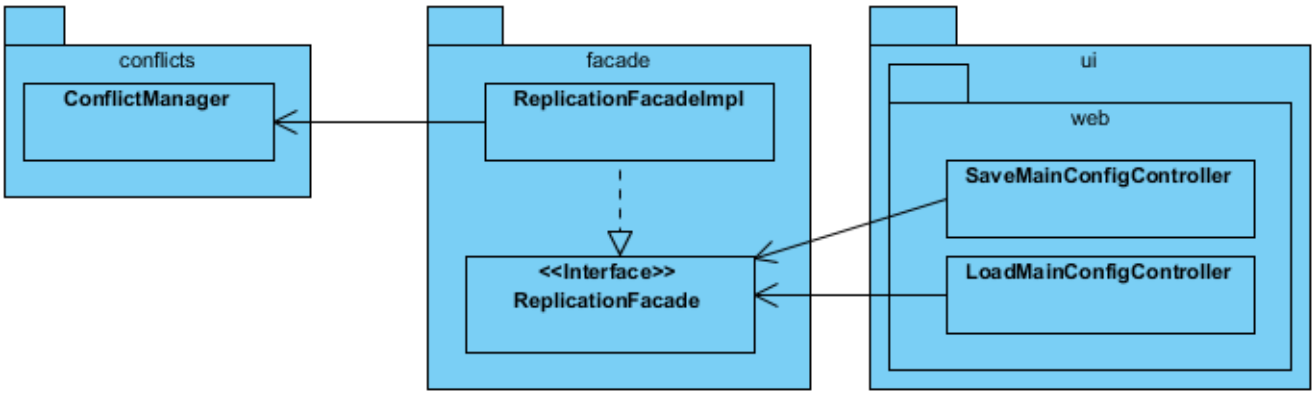


Figura 9: DC para el CU Configurar el módulo de Conflictos.

2.5.2 Descripción de las clases.

Tabla 7: Descripción de la clase ApplicatorTriggerDaoImpl.

Descripción de la clase ApplicatorTriggerDaoImpl	
Nombre: ApplicatorTriggerDaoImpl	
Tipo de clase: Auxiliar	
Atributos	Tipos
monitoringManager	private
notificationFacade	private
Responsabilidades	
Nombre:	DiscardUniquenessConflictWithSameData(action: ReplicableAction, con: Connection, flag: boolean): boolean
Descripción:	Cuando se intenta aplicar una acción de inserción se compruebe si existe alguna tupla con el mismo id y si sucede se revisa cada parámetro de la acción que llega con la original y si todos coinciden se descarta el posible conflicto de unicidad ya que la información es la misma, evitando detener el proceso de réplica.
Nombre:	getPrimaryKeysValues(action: ReplicableAction, table: Table): Map<String, Object>
Descripción:	Devuelve en un mapa las llaves primarias con el nombre de su columna y el valor correspondiente.
Nombre:	getColumnsValues(action: ReplicableAction, table: Table): Map<String, Object>
Descripción:	Devuelve en un mapa las columnas que no son llaves primarias con el nombre de su columna y el valor correspondiente.
Nombre:	Execute(group: ReplicableGroup)
Descripción:	Aplica en la BD las acciones replicable y evita, captura y clasifica los conflictos que ocurran.

Tabla 8: Descripción de la clase ReceiverManager.

Descripción de la clase ReceiverManager	
Nombre: ReceiverManager	
Tipo de clase: Controladora	
Atributos	Tipos
monitoringManager	private
conflictSolving	private
conflictManager	private
Responsabilidades	

Nombre:	onTransactionArrive(ReplicableGroup, boolean)
Descripción:	Es el que controla el proceso de réplica, recibe los grupos replicables y los intenta aplicar en la BD si no existen conflictos resolviéndose o por resolver.
Nombre:	persistConflictResolved(ReplicableGroup)
Descripción:	Intenta aplicar nuevamente en la BD el grupo replicable con el conflicto solucionado y todos los grupos replicables que estaban en espera por la resolución de un conflicto.
Nombre:	conflictDatabaseExceptionTreatment(ConflictDatabaseException, ReplicableGroup)
Descripción:	Proporciona el tratamiento adecuado para los grupos replicables en conflictos y los grupos replicables en espera.
Nombre:	setConflictSolving(boolean)
Descripción:	Cambia el valor de la variable conflictSolving que es la bandera para detener o no el proceso de réplica.
Nombre:	isConflictSolving(): boolean
Descripción:	Devuelve el valor de la variable conflictSolving.

Tabla 9: Descripción de la clase ConflictManager.

Descripción de la clase ConflictManager	
Nombre: ConflictManager	
Tipo de clase: Controladora	
Atributos	Tipos
registrationManager	private
notificationFacade	private
receiverManager	private
metadataManager	private
dataSource	private
conflictReplicableGroupDao	private
ConflictReplicableActionDao	private
Responsabilidades	
Nombre:	solveConflict (replicableGroup: ReplicableGroup, conflictDatabaseException: ConflictDatabaseException): ReplicableGroup
Descripción:	A partir de la configuración del módulo y el tipo de conflicto elige un tratamiento específico para cada conflicto.
Nombre:	getFirstConflictsReplicableGroup():ConflictReplicableGroup
Descripción:	Devuelve el primer grupo replicable en conflicto.
Nombre:	persistConflictReplicableGroup(conflictReplicableGroup: ConflictReplicableGroup)
Descripción:	Persiste en Berkeley el grupo replicable en conflicto.
Nombre:	deleteConflictReplicableGroup(conflictReplicableGroup: ConflictReplicableGroup)
Descripción:	Elimina de Berkeley el grupo replicable en conflicto.
Nombre:	existReplicableGroupConflict(): boolean
Descripción:	Devuelve si existe en Berkeley al menos un grupo replicable en conflictos.
Nombre:	persistConflictReplicableAction(conflictReplicableAction: ConflictReplicableAction)
Descripción:	Persiste en Berkeley las acciones a aplicar para resolver un determinado conflicto de forma automáticamente.
Nombre:	solveManualAuto(replicableActionConflict: ReplicableAction, SenderId: String, variant: String): LinkedList<ReplicableAction>

Descripción:	Aplica las acciones que el usuario crea para resolver el conflicto.

Tabla 10: Descripción de la clase Conflict.

Descripción de la clase Conflict	
Nombre: Conflict	
Tipo de clase: Interface	
Atributos	Tipos
registrationManager	protected
notificationFacade	protected
replicableActionConflict	protected
dataSource	protected
SenderId	protected
Responsabilidades	
Nombre:	changeIdInsertAction(Table, MetadataManager): ReplicableAction
Descripción:	Cambia la llave primaria por otra que genera el sistema de una acción de inserción.
Nombre:	changeInsertActionToUpdateAction(Table, MetadataManager) : ReplicableAction
Descripción:	Transforma una acción de *insert* a una acción de *update.*
Nombre:	changeUpdateActionToInsertAction(Table, MetadataManager) : ReplicableAction
Descripción:	Transforma una acción de *update* a una acción de *insert.*
Nombre:	createActionById(int, Table, MetadataManager) : ReplicableAction
Descripción:	Crea una acción de *insert* a partir de una tupla en la BD.
Nombre:	createDeleteActionById(Table, MetadataManager) : ReplicableAction
Descripción:	Crea una acción de *delete* a partir de otra acción replicable.
Nombre:	getNewReplicableGrupId(): String
Descripción:	Crea un id para el grupo replicable que se envía al nodo remoto.
Nombre:	getPkAutobuild(dataType: String): String
Descripción:	Devuelve la llave primaria construida a partir de la configuración.
Nombre:	getPrimaryKeysValues(ReplicableAction, Table): Map<String,Object>
Descripción:	Devuelve en un mapa las llaves primarias con el nombre de su columna y el valor correspondiente.
Nombre:	solveConflict(boolean, String, Table, MetadataManager, String, String, boolean): LinkedList<ReplicableAction>
Descripción:	Devuelve una lista con acciones que resuelve el conflicto.

Tabla 11: Descripción de la clase ConflictReplicableAction.

Descripción de la clase ConflictReplicableAction	
Nombre: ConflictReplicableAction	
Tipo de clase: Modelo	
Atributos	Tipos
Id	private
replicableAction	private
automaticActionType	private
Exception	private

listReplicableAction	private
Responsabilidades	
Nombre:	getAutomaticActionType(): int
Descripción:	Obtiene la acción automatizada para esa acción replicable en conflicto
Nombre:	getException(): Exception
Descripción:	Obtiene la exception del conflicto.
Nombre:	getId(): String
Descripción:	Obtiene el id.
Nombre:	getListReplicableAction(): LinkedList<ReplicableAction>
Descripción:	Obtiene una lista de acciones replicables.
Nombre:	getReplicableAction(): ReplicableAction
Descripción:	Obtiene la acción replicable en conflicto.
Nombre:	setAutomaticActionType(int)
Descripción:	Modifica el tipo de acción automatizada.
Nombre:	setException(Exception)
Descripción:	Modifica la exception
Nombre:	setId(String)
Descripción:	Modifica el id.
Nombre:	setListReplicableAction(LinkedList<ReplicableAction>)
Descripción:	Modifica la lista de acciones replicables.
Nombre:	setReplicableAction(ReplicableAction)
Descripción:	Modifica la acción replicable en conflicto.

2.5.3 Diagramas de Interacción de Diseño.

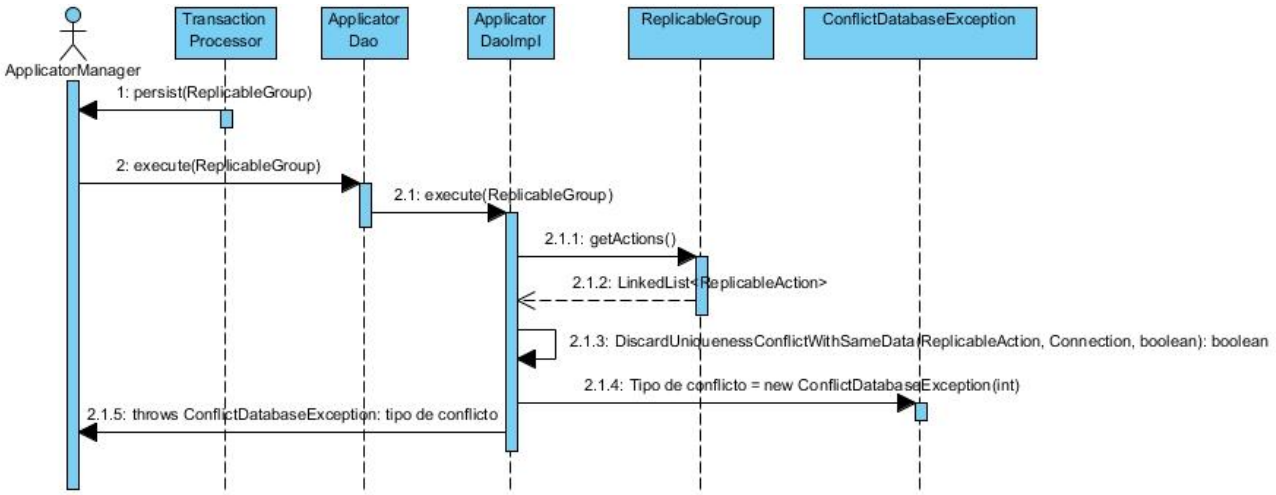


Figura 10: DS para el CU Evitar, capturar y clasificar conflictos

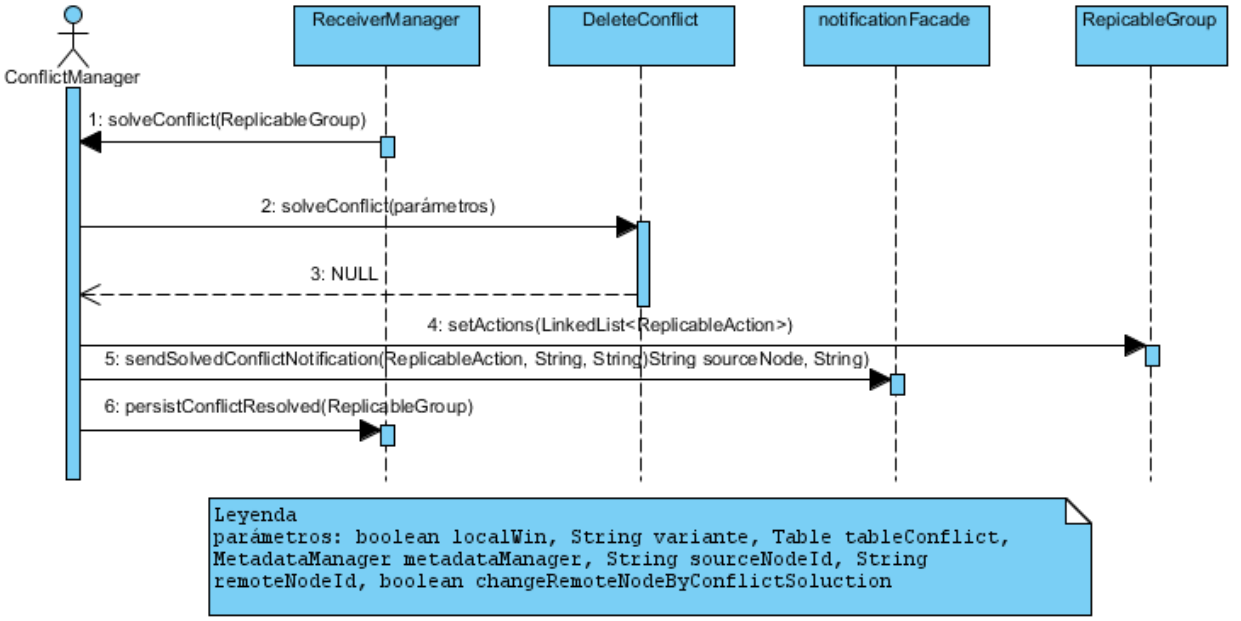


Figura 11: DS para el CU Resolver de forma automática los conflictos de eliminación.

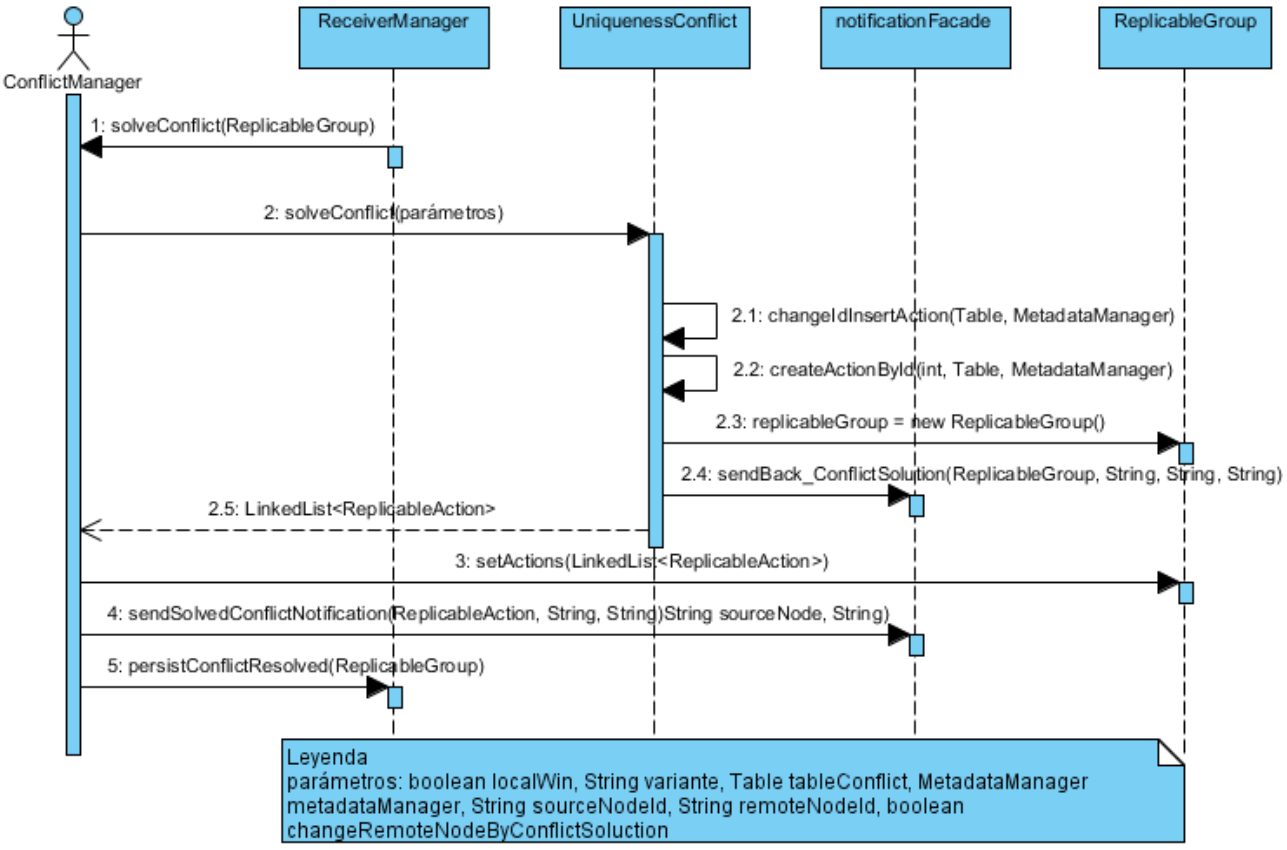


Figura 12: DS para el CU Resolver de forma automática los conflictos de unicidad.

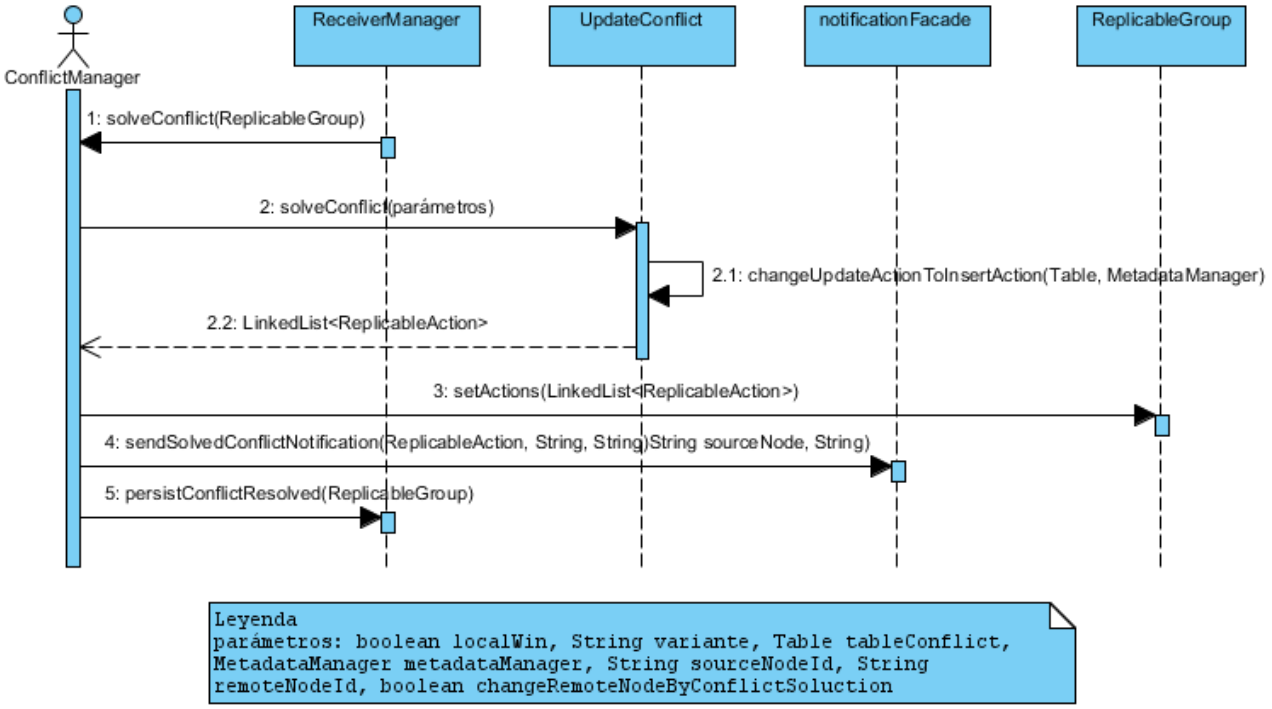


Figura 13: DS para el CU Resolver de forma automática los conflictos de actualización.

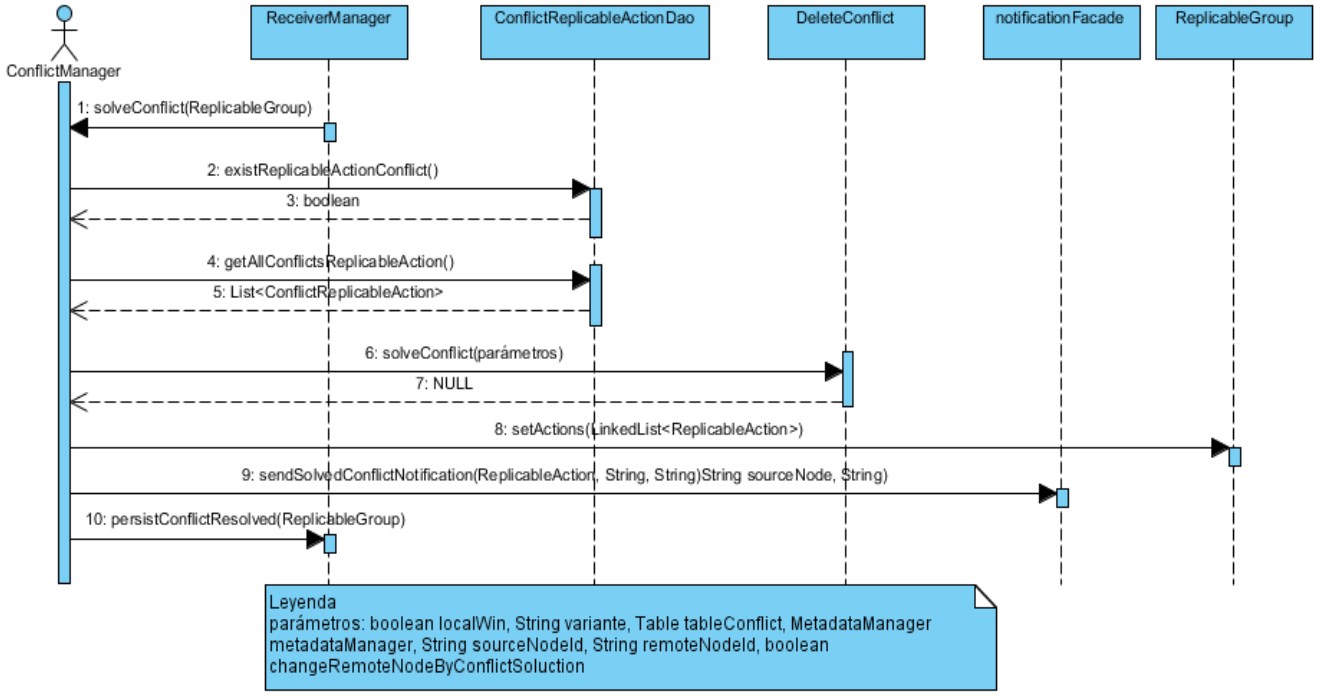


Figura 14: DS para el CU Resolver de forma automática los conflictos de desconocidos.

2.6 Modelo de despliegue.

Un diagrama de despliegue define la arquitectura física del sistema por medio de nodos interconectados, estableciendo la relación fisca que existe entre ellos. Un nodo es un recurso de ejecución tal como un computador, un dispositivo o memoria. (32)

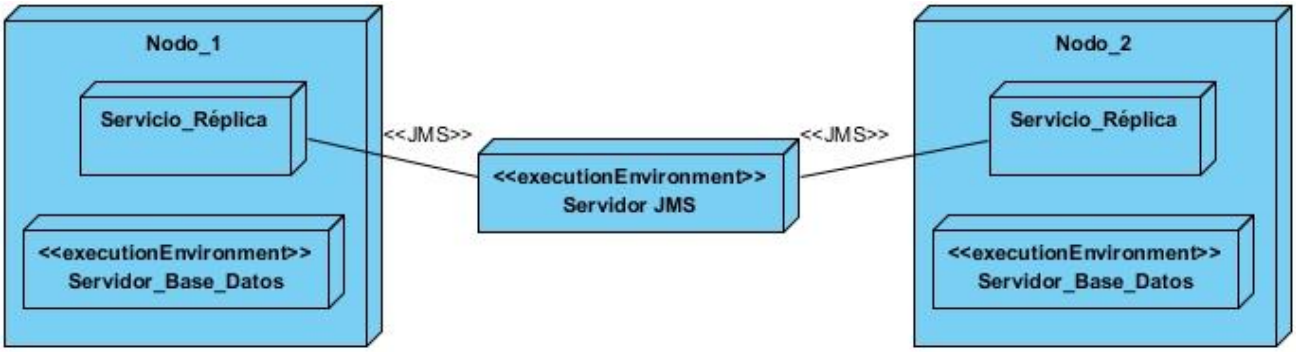


Figura 15: Diagrama de Despliegue.

2.7 Consideraciones parciales del capítulo.

A partir de realizarse el análisis y el diseño del sistema se obtuvieron las siguientes consideraciones:

- La utilización de la metodología OpenUp permitió obtener los diferentes artefactos que describen las funcionalidades del sistema.
- Se hace necesario extender las funcionalidades de los componentes Distribuidor, Aplicador y Administración pues hasta el momento su funcionamiento se basa fundamentalmente en una resolución manual y una mínima cobertura al tratamiento automático de los conflictos.

CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS

En este capítulo se procede a implementar las clases y a validar que el sistema implementa las funcionalidades descritas en los casos de usos y que supera a la versión anterior del Replicador de Datos Reko en cuanto a la eficiencia en la resolución de conflictos de forma automática.

3.1 Modelo de implementación.

Un diagrama de implementación muestra las dependencias entre las partes de código del sistema, en términos de componentes, ficheros de código fuente, ejecutables, etcétera. (32)

3.1.1 Diagrama de componentes.

Los diagramas de componentes muestran tanto los componentes software (código fuente, binario y ejecutable) como las relaciones lógicas entre ellos en un 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. Estos componentes pueden ser:

- Componentes: librerías dinámicas, ejecutables, páginas web.
- Interfaces.
- Relaciones de dependencia, generalización, asociación y realización. (37)

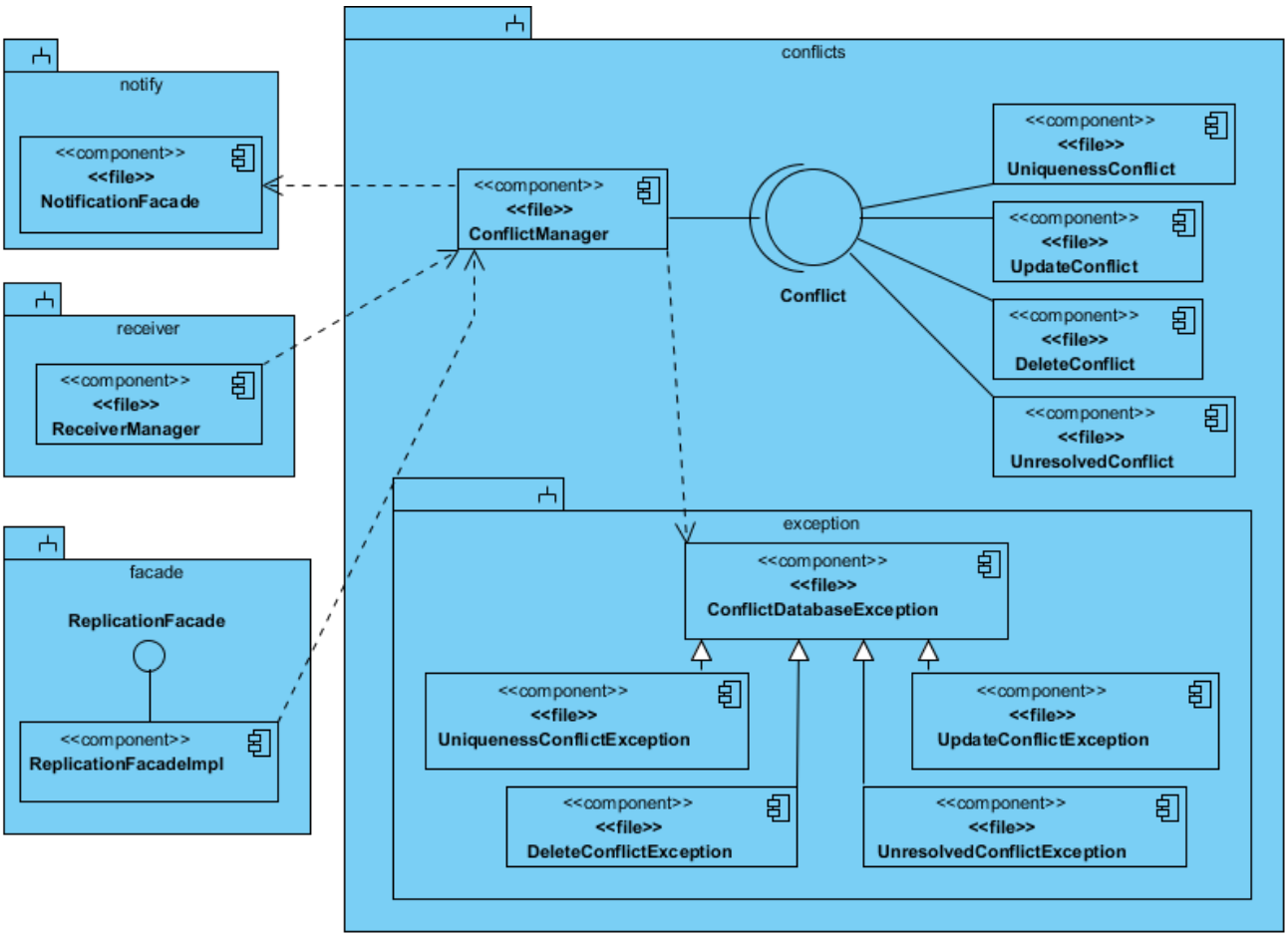


Figura 16: Diagrama de componentes para el subsistema Distribuidor de cambios.

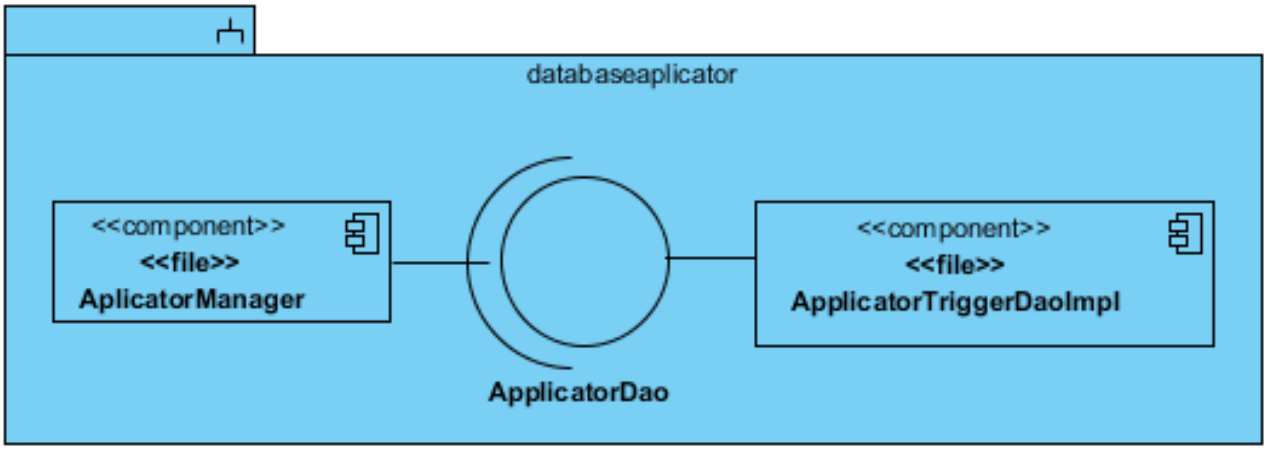


Figura 17: Diagrama de componentes para el subsistema Aplicador de cambios

3.2 Código fuente.

El siguiente fragmento de código pertenece a la clase **`ApplicatorTriggerDaoImpl`**. El método se encarga de aplicar los datos replicados sobre la BD y es el responsable de capturar, clasificar y evitar los conflictos que ocurran.

ApplicatorTriggerDaoImpl

```
public void execute(ReplicableGroup group) throws Exception,
        ConflictDatabaseException {
    long time = 0;
    LinkedList<ReplicableAction> actions = group.getActions();
    time = System.currentTimeMillis();
    Connection con = dataSource.getConnection();
    int actionsForRemove = 0;
    for (ReplicableAction action : actions) {
        try {
            if (action.getActionType() == ReplicableAction.INSERT) {
                if (DiscardUniquenessConflictWithSameData(action, con,
                        false)) {
                    logger.info("Conflicto evitado debido a existencia de
                                tupla idéntica, aplicando la acción: "
                            + action.toString()
                            + " time: "
                            + (System.currentTimeMillis() - time) + " ms");
                    notificationFacade.notifyConflictAvoided(action,
                            group.getSenderId(), GeneralConfig.getNodeId(),
                            group.getId());
                    actionsForRemove++;
                    continue;
                }
            }
            con.setAutoCommit(false);
            String sql = getSql(action);
            PreparedStatement ps = con.prepareStatement(sql.toString());
            setParamsValuesInPrepareStatement(con, action, ps,
                    inputStreamToClose);
            int result = ps.executeUpdate();
            // se captura el conflicto de eliminaci
            if (action.getActionType() == ReplicableAction.DELETE
                    && result == 0) {
```

```
                ConflictDatabaseException conflictDatabaseException = new
                                    DeleteConflictException(actionsForRemove);
                throw conflictDatabaseException;
            }
            // se captura el conflicto de actualizaci
            if (action.getActionType() == ReplicableAction.UPDATE
                    && result == 0) {
                ConflictDatabaseException conflictDatabaseException = new
                        UpdateConflictException(actionsForRemove);
                throw conflictDatabaseException;
            }
            ps.close();
        } catch (DeleteConflictException deleteConflictException) {
            con.rollback();
            con.close();
            logger.info("Error aplicando acción: "
                    + action.toString()
                    + " time: "
                    + (System.currentTimeMillis() - time)
                    + " ms, la acción eliminación sobre una tupla que no
                       existe");

            notifyOnConflict(group, action, "Conflicto de eliminación");
            throw deleteConflictException;
        } catch (UpdateConflictException updateConflictException) {
            con.rollback();
            con.close();
            logger.info("Error aplicando acción: "
                    + action.toString()
                    + " time: "
                    + (System.currentTimeMillis() - time)
                    + " ms, la acción de actualización es sobre una tupla
                       que no existe");
            notifyOnConflict(group, action, "Conflicto de actualización”);
            throw updateConflictException;
        } catch (SQLException sqlException) {
            ConflictDatabaseException conflictDatabaseException = dialect
                    .getConflictDatabaseException(sqlException);
            con.rollback();
            con.close();
            logger.info("Error aplicando acción: " + action.toString()
                    + " time: " + (System.currentTimeMillis() - time)
                    + " ms, " + sqlException.getMessage());
            conflictDatabaseException.setException(sqlException);
            conflictDatabaseException.setActionsForRemove(actionsForRemove);
            notifyOnConflict(group, action, sqlException.getMessage());
            throw conflictDatabaseException;
        } catch (Exception e) {
            e.printStackTrace();
            con.rollback();
            con.close();
            logger.info("Error aplicando acción: " + action.toString()
                    + " time: " + (System.currentTimeMillis() - time)
                    + " ms, " + e.getMessage());
            ConflictDatabaseException conflictDatabaseException = new
             UnresolvedConflictException(actionsForRemove, e.toString());
            conflictDatabaseException.setException(e);
            notifyOnConflict(group, action, e.getMessage());
            throw conflictDatabaseException;
        } finally {
            GeneralConfig.applyingData = false;
        }
```

```
            actionsForRemove++;
        }
        con.close();
    }
```

El método que se muestra a continuación pertenece a la clase **UpdateConflict** y se utiliza para resolver los conflictos de actualización a partir de la configuración establecida.

UpdateConflict

```
public LinkedList<ReplicableAction> solveConflict(boolean nodoGanador,
            String variante, Table tableConflict,
            MetadataManager metadataManager, String sourceNodeId,
            String remoteNodeId, boolean changeRemoteNodeByConflictSoluction) {
        LinkedList<ReplicableAction> listReplicableAction = new
                                                LinkedList<ReplicableAction>();
        if (nodoGanador) {
            if (variante.equals("1")) {
                // Insertar datos que llegan como nueva tupla.
                listReplicableAction.add(changeUpdateActionToInsertAction(
                        tableConflict, metadataManager));
                return listReplicableAction;
            } else if (variante.equals("2")) {
                 // Ignorar datos que llegan y
                 //eliminar datos en nodo remoto
                 if (changeRemoteNodeByConflictSoluction) {
                    ReplicableGroup replicableGroup = new ReplicableGroup(
                            getNewReplicableGrupId(),
                            ReplicableGroup.REPLICATOR_TYPE_TRIGGERS,
                            sourceNodeId);
                    replicableGroup.addSentence(createDeleteActionById(
                            tableConflict, metadataManager));
                    notificationFacade.sendBack_ConflictSolution(
                            replicableGroup, remoteNodeId,
                            replicableActionConflict.toString(),"Explicación");
                }
                return null;
            }
        } else {
            if (variante.equals("1")) {
                // Insertar datos que llegan como nueva tupla.
                listReplicableAction.add(changeUpdateActionToInsertAction(
                        tableConflict, metadataManager));
                return listReplicableAction;
            }
        }
        return null;
    }
```

Este método es el responsable de aplicar nuevamente en la BD luego de que un conflicto ha sido solucionado ya sea manual o automáticamente y si aplica correctamente va a realizar una llamada recursiva del mismo hasta que se vacíe la cola de grupo replicables retenidos por la ocurrencia de conflictos.

ReceiverManager

```
public void persistConflictResolved(ReplicableGroup replicableGroup) {
        try {
            persistInLocalDB(replicableGroup);
            if (conflictManager.existReplicableGroupConflict()) {
                ConflictReplicableGroup firstConflictReplicableGroup =
                        conflictManager.getFirstConflictsReplicableGroup();
                conflictManager.deleteConflictReplicableGroup
                (firstConflictReplicableGroup);
                persistConflictResolved(firstConflictReplicableGroup
                        .getReplicableGroup());
            } else {
                conflictSolving = false;
            }
        } catch (ConflictDatabaseException e) {
            conflictSolving = true;
            conflictDatabaseExceptionTreatment(e, replicableGroup);
        }
    }
```

3.3 Modelo de prueba.

Las pruebas de software son una actividad en la cual un sistema o componente es ejecutado bajo condiciones específicas, y los resultados son observados, registrados y analizados. La prueba de un sistema se define como el proceso de ejercitar o evaluar el sistema por medios manuales o automáticos, para verificar que satisface los requerimientos o para identificar diferencias entre los resultados esperados y los que producen el sistema.

Para lograr el objetivo de las pruebas de software es necesario planear y ejecutar una serie de pasos (pruebas de unidad, integración, validación y sistema). Las pruebas de unidad e integración se concentran en la verificación funcional de cada componente y en la incorporación de componentes en la arquitectura del software. La prueba de validación demuestra el cumplimiento con los requisitos del software y la prueba del sistema valida el software una vez que se ha incorporado a un sistema mayor.

Existen dos categorías diferentes de técnicas de diseño de casos de prueba:

- **Las pruebas de caja blanca:** se concentran en la estructura de control del programa. Los casos de prueba se derivan para asegurar que todas las instrucciones del programa se ejecuten por lo menos una vez durante la prueba, y que todas las condiciones lógicas se ejerciten.
- **Las pruebas de caja negra:** están diseñadas para validar requisitos funcionales sin importar el funcionamiento interno de un programa. Los casos de prueba se derivan mediante la partición de los dominios de entrada y salida de un programa, en forma tal que proporcione cobertura completa. (33)

Se diseñan y ejecutan una serie de pruebas con el objetivo de validar el módulo desarrollado en cuanto a los objetivos de las pruebas antes mencionadas. Además se busca corroborar la eficiencia del módulo propuesto y demostrar que supera a la versión anterior.

3.3.1 Camino Básico.

Esta técnica de caja blanca permite obtener una medida de la complejidad de un diseño procedimental y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución, diseñando casos de prueba que garanticen que cada camino se ejecuta al menos una vez.

Para la ejecución de esta técnica existen cuatro pasos básicos:

1. Obtener la complejidad ciclomática del grafo de flujo.
2. Definir el conjunto básico de caminos independientes.
3. Determinar los casos de prueba que permitan la ejecución de cada uno de los caminos anteriores.
4. Ejecutar cada caso de prueba y comprobar que los resultados son los esperados. (33)

Tabla 12: Resultados de la prueba de caja blanca

UpdateConflict	
0	`public LinkedList<ReplicableAction> solveConflict(boolean nodoGanador, String variante, Table tableConflict, MetadataManager metadataManager, String sourceNodeId, String remoteNodeId, boolean changeRemoteNodeByConflictSoluction) {`
1	`LinkedList<ReplicableAction> listReplicableAction = new LinkedList<ReplicableAction>();`
2	`if (nodoGanador) {`
3	`if (variante.equals("1")) {`
4	`listReplicableAction.add(changeUpdateActionToInsertAction(tableConflict, metadataManager));`
4	`return listReplicableAction;`
	`} else if (variante.equals("2")) {`
5	`if (changeRemoteNodeByConflictSoluction) {`
6	`ReplicableGroup replicableGroup = new ReplicableGroup(getNewReplicableGrupId(), ReplicableGroup.REPLICATOR_TYPE_TRIGGERS, sourceNodeId);`
6	`replicableGroup.addSentence(createDeleteActionById(tableConflict, metadataManager));`
6	`notificationFacade.sendBack_ConflictSolution(replicableGroup, remoteNodeId, replicableActionConflict.toString(),"Explicación");`
7	`}`
8	`return null;`
9	`}`
	`} else {`
10	`if (variante.equals("1")) {`
11	`listReplicableAction.add(changeUpdateActionToInsertAction(tableConflict, metadataManager));`
11	`return listReplicableAction;`

```
12              }
13          }
14            return null;
    }
```

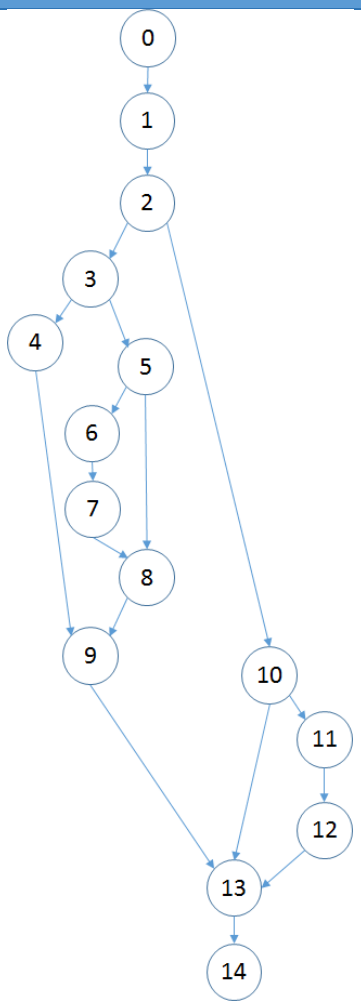
Construcción del grafo de complejidad ciclomática correspondiente al código	
	**Complejidad ciclomática:** A = 18(Aristas) N = 15 (Nodos) V(G) = A – N + 2 V (G) = 18 – 15 + 2 = 5 **P = 3 (Nodos predicados)** V (G) = P + 1 V (G) = 4 + 1 V (G) = 5 **R (Regiones)** R = V (G) R = 5 1. 0, 1, 2, 10, 13, 14 2. 0, 1, 2, 10, 11 3. 0, 1, 2, 3, 4 4. 0, 1, 2, 3, 5, 7, 8 5. 0, 1, 2, 3, 5, 8

Tabla 13: Casos de prueba

No.	Caso de prueba	Resultado esperado
1	No hay una variante configurada para resolver los conflictos de actualización.	Descarta la acción en conflicto.
2	Variante **insertar datos** para un **nodo no ganador**.	Convierte la acción de actualización en conflicto en una acción de inserción y la aplica en la BD.
3	Variante **insertar datos** para un **nodo ganador**.	Convierte la acción de actualización en conflicto en una acción de inserción y la aplica en la BD.

4	Variante **ignorar datos** para un **nodo ganador afectando al nodo remoto**.	Descarta la acción en conflicto y envía al nodo remoto una acción de eliminación con el id de la acción en conflicto.
5	Variante **ignorar datos** para un **nodo ganador sin afectar al nodo remoto**.	Descarta la acción en conflicto.

3.3.2 Framework JUnit.

JUnit es un *framework* de código abierto para pruebas unitarias para el lenguaje de programación Java originalmente escrito por Erich Gamma y Kent Beck. Algunas características de JUnit incluyen:

- Afirmaciones para probar resultados esperados.
- Aparatos de prueba para compartir datos comunes de prueba.
- Corredores de pruebas para pruebas de ejecución. (38)

Se utiliza JUnit para la ejecución de las pruebas automatizadas a 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.

Las pruebas fueron diseñadas guardando relación con las clases a probar permitiendo acceder a los atributos y métodos protegidos de las clases, de tal forma que las funcionalidades encapsuladas en las clases de prueba fueron implementadas según los paquetes de las clases a probar.

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

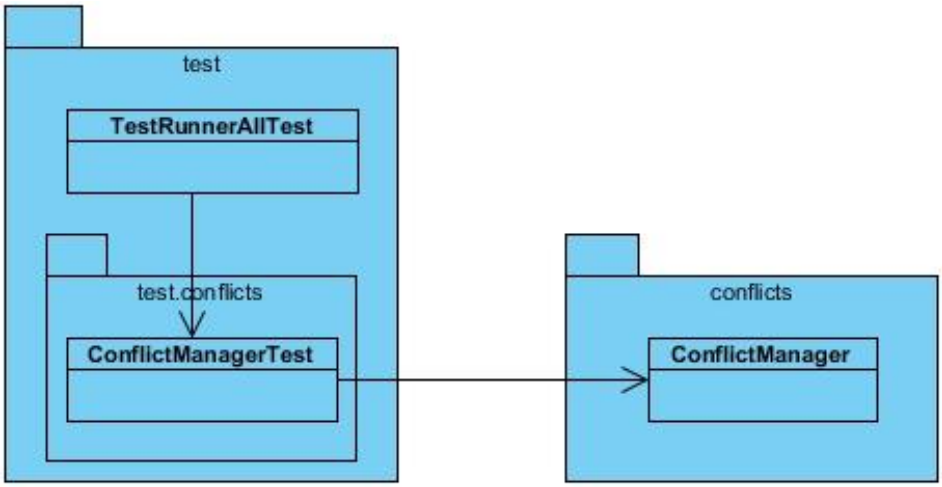


Figura 18: Clases de prueba.

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

```
****** Aplicando pruebas a la clase ConflictManager *****
.   Prueba: testSolveConflict
     Tiempo de ejecución: 253

****** Aplicando pruebas a la clase ConflictManager *****
.   Prueba: testSolveManualAuto
     Tiempo de ejecución: 206

******* Aplicando pruebas a la clase ConflictManager ******
.   Prueba: testPersistConflictReplicableAction
     Tiempo de ejecución: 124

****** Aplicando pruebas a la clase ConflictManager******
.   Prueba: testPersistConflictReplicableGroup
     Tiempo de ejecución: 130

Time: 0.713

OK (4 tests)
```

Figura 19: Resultados de las pruebas unitarias

Como se puede observar, las pruebas realizadas arrojaron resultados positivos así como tiempos de ejecución bajos, lo cual demuestra la solidez del código.

3.3.3 Pruebas de Sistema.

Las pruebas de sistema buscan discrepancias entre el programa y los requisitos, enfocándose en los errores cometidos durante la transición del proceso al diseñar la especificación funcional. Esto hace a las pruebas de sistema un proceso vital de pruebas, ya que en términos del producto, número de errores cometidos, y criticidad de estos, es un paso en el ciclo de desarrollo generalmente propenso a la mayoría de los errores. Las pruebas de sistema tienen como propósito fundamental comparar el sistema o programa con sus objetivos originales. (33) Se diseñaron un conjunto de casos de prueba que permiten validar el cumplimiento de los requisitos funcionales. La ejecución de estos casos de prueba permitió corroborar que los requerimientos establecidos inicialmente se cumplen correctamente sobre el sistema, emitiendo un resultado satisfactorio en la aplicación de las pruebas.

3.3.4 Conteo de Funcionalidad.

El grado de utilidad de ambas versiones de la herramienta puede ser determinado con un simple conteo de sus funcionalidades pues ambas versiones son compatibles entre sí. La nueva versión no elimina ninguna de las funcionalidades de la antigua, solo refina y amplía las existentes. A continuación se muestra una tabla comparativa:

Tabla 14: Comparación de funcionalidades entre las dos versiones del Replicador de Datos Reko.

Funcionalidades	Versión anterior	Versión desarrollada	
	Implementa	Implementa	Mejora
Capturar conflicto de unicidad	x	x	
Capturar conflicto de eliminación	x	x	
Capturar conflicto de actualización		x	
Evitar conflicto de unicidad		x	
Resolución automática de conflicto de unicidad	x		x
Resolución automática de conflicto de eliminación	x		x
Resolución automática de conflicto de actualización		x	
Resolución automática de conflictos desconocidos		x	
Configurar el módulo de conflictos.	x		x

3.3.5 Eficiencia.

Para corroborar la eficiencia del sistema propuesto y demostrar que supera a la versión anterior se decide ejecutar las dos versiones sobre un mismo escenario. Se analizan indicadores como el tiempo de respuesta y la consistencia de la información a partir del tratamiento de la ocurrencia de conflictos pre-analizados. El escenario propuesto consta de dos bases de datos: BD1 y BD2, donde el Nodo_Envía captura la información de la BD1 y la envía hacia el Nodo_Recibe que es el encargado de aplicar los cambios sobre la BD2. Además, el Nodo_Recibe es nodo ganador, por tanto, en la resolución de conflictos su información debe prevalecer sobre la información del Nodo_Envía. (Figura 17).

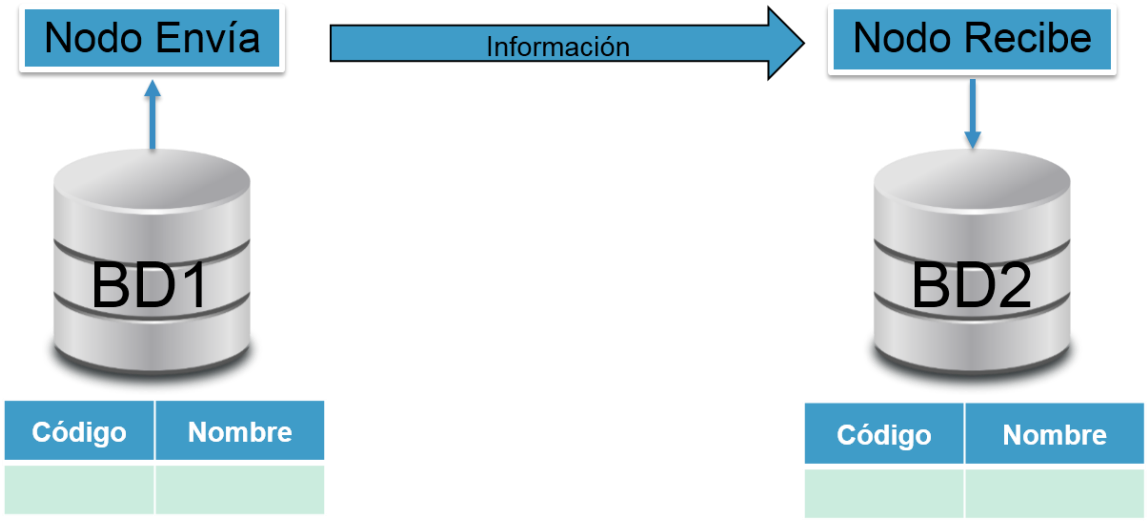


Figura 20: Escenario de réplica

Para la ejecución del escenario se utilizan varios paquetes de datos que contienen distintos tipos de conflictos para analizar la consistencia de la información con respecto a las dos versiones. A continuación se muestra el resultado obtenido:

Estado inicial de las tablas en las bases de datos:

	codigo [PK] character	nombre character varying
1	1	adrian
2	2	helian
3	3	carlos
4	4	luis
5	5	manuel

Figura 21: Tabla de la BD2

	codigo [PK] character	nombre character varying
1	10	
2	11	
3	12	
4	13	
5	14	
6	20	
7	21	
8	22	
9	23	
10	24	

Figura 22: Tabla de la BD1

Se aplica el paquete de datos sobre la BD1:

```
//conflictos de unicidad
INSERT INTO producto(codigo, nombre) VALUES ('1', 'adrian');
INSERT INTO producto(codigo, nombre) VALUES ('2', 'ernesto');
INSERT INTO producto(codigo, nombre) VALUES ('3', 'felipe');
INSERT INTO producto(codigo, nombre) VALUES ('4', 'jose');
INSERT INTO producto(codigo, nombre) VALUES ('5', 'pedro');
//conflictos de actualización
UPDATE producto SET nombre='actualización' WHERE codigo='10';
UPDATE producto SET nombre='actualización2' WHERE codigo='11';
UPDATE producto SET nombre='actualización3' WHERE codigo='12';
UPDATE producto SET nombre='actualización4' WHERE codigo='13';
UPDATE producto SET nombre='actualización5' WHERE codigo='14';
//conflictos de eliminación
DELETE FROM producto WHERE codigo='20';
DELETE FROM producto WHERE codigo='21';
DELETE FROM producto WHERE codigo='22';
DELETE FROM producto WHERE codigo='23';
DELETE FROM producto WHERE codigo='24';
//conflictos desconocido
INSERT INTO producto(codigo, nombre) VALUES ('30', null);
```

Resultado obtenido a partir de la réplica de datos y el tratamiento de conflictos con la versión anterior del Replicador de Datos Reko:

	codigo [PK] character	nombre character varying
1	1	adrian
2	2	ernesto
3	3	felipe
4	4	jose
5	5	pedro

Figura 23: Tabla de la BD2 después de la resolución de los conflictos.

	codigo [PK] character	nombre character varying
1	1	adrian
2	10	actualización
3	11	actualización2
4	12	actualización3
5	13	actualización4
6	14	actualización5
7	2	ernesto
8	3	felipe
9	30	
10	4	jose
11	5	pedro

Figura 24: Tabla de la BD1 después de la resolución de los conflictos.

Como se puede observar, la versión anterior del Replicador de Datos Reko no asegura la consistencia de la información en la resolución de conflictos de forma automática. El mecanismo implementado no tiene en cuenta la relevancia de la información en la resolución de conflictos, debido a que están restringidas las opciones a elegir y no cubren otras formas de solucionar el conflicto.

Resultado obtenido a partir de la réplica de datos y el tratamiento de conflictos con la nueva versión del Replicador de Datos Reko:

	codigo [PK] character	nombre character varying
1	1	adrian
2	10	actualización
3	11	actualización2
4	12	actualización3
5	13	actualización4
6	14	actualización5
7	2	helian
8	3	carlos
9	4	luis
10	5	manuel
11	50	ernesto
12	51	felipe
13	52	jose
14	53	pedro

Figura 26: Tabla de la BD2 después de la resolución de los conflictos.

	codigo [PK] character	nombre character varying
1	1	adrian
2	10	actualización
3	11	actualización2
4	12	actualización3
5	13	actualización4
6	14	actualización5
7	2	helian
8	3	carlos
9	30	
10	4	luis
11	5	manuel
12	50	ernesto
13	51	felipe
14	52	jose
15	53	pedro

Figura 25: Tabla de la BD1 después de la resolución de los conflictos.

En este caso la nueva versión asegura la consistencia de la información y tiene en cuenta la relevancia de la información en la resolución de conflictos de forma automática.

El tiempo de respuesta depende de la cantidad de conflictos que no pueden ser solucionados de forma automática. A continuación se muestra una tabla comparativa entre las dos versiones que contienen los conflictos que se pueden resolver de forma automática y la cantidad de variantes para resolverlos:

Tabla 15: Resolución automática y variantes.

Tipo de conflicto	**Versión anterior**		**Versión desarrollada**	
	Automático	Variantes	Automático	Variantes
Conflicto de unicidad	x	2	x	4
Conflicto de eliminación	x	1	x	1
Conflicto de actualización	-	-	x	3
Conflictos desconocidos	-	-	x	2

La nueva versión permite automatizar la resolución de todos los tipos de conflictos. Además brinda más variantes para resolver los conflictos. Se puede concluir que el tiempo de respuesta de la versión desarrollada es menor que la versión anterior debido a que la mayoría de los conflictos que ocurren en la versión anterior se tienen que resolver de forma manual.

Observando los resultados obtenidos después de ejecutar las dos versiones en un mismo escenario y analizar el tiempo de respuesta, se puede concluir que la nueva versión es más eficiente que la anterior pues mejora, amplía sus funcionalidades y reduce el tiempo de respuesta del sistema.

3.4 Consideraciones parciales del capítulo.

Durante este capítulo se implementaron las funcionalidades con éxito, presentándose el modelo de implementación, conformado por los Diagramas de Componentes. Fueron expuestos los métodos más importantes de la implementación y las relaciones entre ellos. Con el trabajo realizado en este capítulo se pueden arribar a las siguientes consideraciones:

- A partir de la metodología OpenUp se diseñaron los casos de pruebas que permitieron evaluar el funcionamiento del módulo desarrollado.
- La realización de la prueba de software permitió detectar y corregir los errores durante la implementación.
- La versión desarrollada supera a la versión anterior en cuanto a eficiencia y cantidad de funcionalidades.

Conclusiones generales.

Al finalizar el presente trabajo de diploma se logró cumplir de manera satisfactoria el objetivo general planteado, reflejándose en los siguientes resultados:

- El estudio de los principales conceptos relacionados con la réplica de datos permitió identificar características deseables dentro de los sistemas que desarrollan estas funcionalidades.
- Las características de los sistemas de réplica analizados no son apropiadas para reutilizarse dentro del Replicador de Datos Reko.
- La utilización de la metodología OpenUp permitió guiar el proceso de desarrollo obteniéndose artefactos que documentan la implementación del módulo.
- Se implementó un módulo que asegura la resolución de conflictos de manera automática, el cual eleva el tiempo de respuesta y la consistencia de la información en la resolución de conflictos dentro del Replicador de Datos Reko.
- Las pruebas realizadas corroboraron la solidez de la implementación y la eficiencia de la solución propuesta, demostrando que supera a la versión anterior en cuanto a eficiencia.

Recomendaciones

En la versión actual del Replicador de Datos Reko se incluyó la funcionalidad que permite replicar la estructura de las BD. Se plantea como recomendación de este trabajo incluir el mecanismo de resolución automática de conflictos en el proceso de réplica de estructura.

Referencias Bibliográficas

1. Sierra, Manuel. ¿QUÉ ES UNA BASE DE DATOS Y CUÁLES SON LOS PRINCIPALES TIPOS? 2013.

2. Tamer Özsu, M y Valduriez, Patrick . Principles of Distributed Database Systems. s.l. : Springer, 2011.

3. SQL Server Replication. [En línea] http://msdn.microsoft.com/en-us/library/ms151198.aspx.

4. Hernández Suárez, Eivys , y otros. REPLICADOR DE DATOS REKO: HERRAMIENTA DE CÓDIGO ABIERTO PARA SISTEMAS DISTRIBUIDOS. [En línea] 2012. http://repositorio_institucional.uci.cu/jspui/handle/ident/4303.

5. Pérez Tarancon, Jofman y Lavandero García, Jose. Solución integrada de réplica de información. [En línea] 2008. http://repositorio_institucional.uci.cu/jspui/handle/ident/TD_2810_08.

6. Mato García, Rosa María. Informática III. s.l. : Editorial Universitaria, 2009.

7. Ezequiel Rozic, Sergio . Bases de datos y su aplicación con SQL (Manuales Users Series). [En línea] 2014.

8. Applying aspect oriented technology to relational data bases: The replication case. Asteasuain, Fernando y Javed, Adeel . 2009.

9. Chapple, Mike . Replication Definition. [En línea] 2014. http://databases.about.com/cs/administration/g/replication.htm.

10. Urbano, Randy . Oracle Database Advanced Replication, 11gRelease 2 (11.2). 2013.

11. SymmetricDS, Open Source Database Replication. [En línea] 2013. http://www.symmetricds.org/.

12. Oracle8i Replication API Reference. [En línea] http://docs.oracle.com/cd/F49540_01/DOC/server.815/a67793/toc.htm.

13. Ortiz Noriega, Dresky y López Boullon, Carlos Javier. Módulo de Reko para la resolución de conflictos. [En línea] 2010. http://repositorio_institucional.uci.cu/jspui/handle/ident/TD_02982_10.

14. Long, Eric, y otros. SymmetricDS User Guide. 2013.

15. Oracle Database 11g Features: Data Replication & Integration. [En línea] http://www.oracle.com/technetwork/database/features/data-integration/index.html.

16. Daffodil Replicator Console Guide. 2005.

17. Daffodil Replicator Developer's Guide. 2005.

18. PostgreSQL, Slony-I A replication system for. JAN WIEC. s.l. : Afilias USA INC. Horsham, Pennsylvania, USA.

19. Slony-I. [En línea] http://slony.info/documentation/1.2/slonyintro.html.

20. Replicación de SQL Server. [En línea] http://msdn.microsoft.com/es-es/library/ms151198.aspx.

21. Replicación de mezcla. [En línea] http://msdn.microsoft.com/es-es/library/ms152746.aspx.

22. Replicación transaccional. [En línea] http://msdn.microsoft.com/es-es/library/ms151176.aspx.

23. ¿Qué es Java Enterprise Edition ? [En línea] http://www.java.com/es/download/faq/techinfo.xml.

24. Holzner, Steven. La Biblia De Java 2.

25. Hashimi, Sayed, Komatineni, Satya y MacLean, Dave. Spring Persistence with Hibernate. s.l. : Apress, 2010.

26. Apache ActiveMQ ™. [En línea] http://activemq.apache.org/index.html.

27. Apache Tomcat. [En línea] http://tomcat.apache.org/.

28. Walls, Craig. Spring in action. Shelter Island : Manning, 2011.

29. Administración de las Configuraciones y Definiciones de Seguridad del SIGEP. Hernández Suárez, Eivys y Pimentel González, Luis Alberto. 2008.

30. The Dojo Toolkit. [En línea] http://dojotoolkit.org/.

31. BALDUINO, R. Introduction to OpenUP (Open Unified Process). [En línea] 2007. http://www.eclipse.org/epf/general/OpenUP.pdf.

32. Jacobson, Ivar, Booch, Grady y Rumbaugh, James. El Proceso Unificado de Desarrollo de Software.

33. Pressman, R.S. Ingeniería del Software, Un Enfoque Práctico. s.l. s.l. : McGraw-Hil., 2002.

34. Sommerville, Ian. Requerimientos. s.l. 2005.

35. SOCIETY, SOFTWARE ENGINEERING STANDARDS COMMITTEE OF THE IEEE COMPUTER. Recommended Practice for Architectural Description of Software-Intensive Systems. 2000.

36. Martin, Robert C. Design Principles and Design Patterns.

37. Larman, C. UML y Patrones. Prentice Hall, 2005.

38. JOHNSON, R. Expert One-on-One. J2EE Development without EJB. [En línea] 2004. http://read.pudn.com/downloads87/ebook/336085/Wrox%20Press%20Expert%20One-on-One%20J2EE%20Development%20without%20EJB.pdf.