UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MÓDULO PARA LA SINCRONIZACIÓN AUTOMÁTICA DE DOCUMENTOS XML

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

CRISTHIAN ANDRÉS MOYA BASCUR

PROFESOR GUÍA:
SERGIO OCHOA DELORENZI
MIEMBROS DE LA COMISIÓN:
CLAUDIO GUTIÉRREZ GALLARDO
PATRICIO NELLEF INOSTROZA FAJARDIN

SANTIAGO DE CHILE
AGOSTO 2007

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: CRISTHIAN MOYA B.
FECHA: 15/08/2007
PROF. GUIA: Dr. SERGIO OCHOA

"MÓDULO PARA LA SINCRONIZACIÓN AUTOMÁTICA DE DOCUMENTOS XML"

Esta memoria aborda el escenario de trabajo donde las personas necesitan desplazarse entre distintas ubicaciones físicas, a fin de llevar a cabo su tarea. Estos trabajadores utilizan dispositivos móviles, tales como Laptops y PDA's, para llevar a cabo sus actividades y compartir recursos (información). La información que comparte cada individuo corresponde a documentos XML. Cada persona puede modificar a su gusto las copias locales de los documentos compartidos, aún si estos individuos se encuentran temporalmente desconectados del servidor y de sus compañeros. Por esa razón, se pueden generar inconsistencias entre las copias. Para alcanzar nuevamente un estado consistente; los trabajadores tienen que sincronizar sus copias a bajo costo.

El principal objetivo de este trabajo de título es generar una aplicación que sea capaz de sincronizar documentos XML en forma automática. El método de sincronización se enfocó al escenario de uso de las aplicaciones colaborativas móviles que ejecutan en PDAs, y utilizan redes inalámbricas como soporte de comunicación.

Se analizaron varias aplicaciones ya desarrolladas que reconcilian documentos XML y que ejecutan sobre PDA's. Ninguna de ellas presentaba una real solución a los requerimientos que planteaba el escenario en el que se enmarca este trabajo de título. Por esta razón, se decidió desarrollar una aplicación que automatizara lo más posible el proceso de reconciliación de documentos XML. Dada la flexibilidad en el formato de los documentos XML, se definieron ciertas restricciones sobre los documentos XML a sincronizar por la aplicación. Estas restricciones se vieron reflejadas en la incorporación de atributos a los nodos de los documentos XML a sincronizar.

Se diseñó un esquema de sincronización extensible a través de clases C#. Este esquema involucra dos pasos. Primero se genera el árbol de diferencias entre los documentos XML a sincronizar. Luego, se resuelve cada una de las diferencias encontradas. Para resolver las diferencias se desarrollaron clases llamadas Resolutores. Estos resolutores definen políticas de reconciliación que se traducen en operaciones aritméticas, lógicas, de orden, etc. Cuando los resolutores encuentran diferencias entre los documentos XML, aplican las políticas de reconciliación para determinar cuál será el valor final del nodo reconciliado. Las políticas de reconciliación de los resolutores son fácilmente programables y extensibles. Con la creación de políticas de reconciliación de diferencias se logró automatizar el proceso de sincronización.

En resumen, se considera que esta memoria alcanzó el objetivo planteado, dado que se logró desarrollar una aplicación capaz de sincronizar diferencias entre documentos XML de manera automática. Esta aplicación ejecuta tanto en PDA's como en PCs.

Agradecimientos

Principalmente agradezco a mi Dios, Jehová por permitirme esta oportunidad. A mi familia sobremanera y a mis amigos que me apoyan siempre. Al profesor Dr. Sergio Ochoa por la ayuda entregada en la confección de esta memoria.

Índice de Contenidos

1. Introducción

El uso de dispositivos móviles ha crecido mucho en los últimos años. Las conocidas PDA's (Personal Digital Assistant) y notebooks (laptops) se han masificado y han resultado una importante herramienta de trabajo en muchas ocasiones. Los dispositivos móviles están siendo utilizados, por ejemplo, para recopilar datos e información en terreno, y transmitirlos a un servidor central.

Desde un punto de vista técnico, los retos que ha generado la masificación de dispositivos móviles no son triviales. De hecho se habla de una nueva rama de la ingeniería de software, específica para aplicaciones móviles [Roman, 2000]. En primer lugar, las aplicaciones móviles tienen que ejecutar sobre redes inalámbricas, que generalmente son de menor calidad que las cableadas, debido a su limitado ancho de banda, rango de comunicación y por sobre todo, a la alta tasa de desconexiones [Kleinrock, 1996].

En contraste con las redes cableadas, donde los computadores se conectan a una sola red, una aplicación móvil tiene que lidiar con distintos tipos de conexiones de diversa calidad, y situadas en distintos lugares. Por otra parte, aun cuando los dispositivos móviles están rápidamente haciéndose más poderosos, sus capacidades de memoria y procesamiento son todavía muy limitadas. Esto sin considerar que dichos dispositivos son sensibles al consumo de su batería [Musolesi, 2002].

Dado todo lo ya mencionado, ha resultado necesario crear aplicaciones móviles que funcionen en forma distribuida, que sean autónomas y que tengan la capacidad de comunicar y compartir datos con otras aplicaciones móviles. Debido a eso, los ingenieros de software se han visto enfrentados a un nuevo problema causado por la movilidad de los usuarios y por las limitaciones de hardware de los dispositivos computacionales y de conectividad de las redes inalámbricas.

En un escenario móvil, es esencial crear réplicas locales, para que sea posible que cada individuo pueda acceder y modificar los datos mientras no se pueda contactar al servidor. De esa manera cada nodo aumenta su autonomía de trabajo. Así, durante una desconexión, los usuarios deben actualizar las copias locales de los datos compartidos. El resultado de la actualización son copias locales, distintas a la versión que maneja el servidor. Las copias distribuidas no son necesariamente consistentes, puesto que muchos usuarios las modifican en paralelo, mientras están desconectados. Surge así la necesidad de la “sincronización”, también llamada “reconciliación” [Mascolo, 2002]. En el presente trabajo de título se utiliza indistintamente el término “sincronización” y “reconciliación” para referirse al mismo proceso.

La “reconciliación” se define como el proceso de analizar 2 registros para resolver discrepancias entre ellos. Es decir, a partir de 2 documentos distintos se intenta conseguir un tercero, que sea lo más parecido posible a los 2 primeros. Los sistemas existentes para realizar esta operación no proporcionan mecanismos automáticos y eficaces para manejar las inconsistencias generadas por las copias locales. El comportamiento más común es elegir una copia considerando una política predefinida, usar timestamp sin implementar una

política particular o bien, transferirle el criterio de sincronización al usuario [Musolesi, 2002]. Muchas veces, este timestamp debe ser reemplazado por otro tipo de indicador, que permita sincronizar documentos por distintos criterios, como por ejemplo, el rol del usuario que actualizó los datos a sincronizar, o bien el número de sincronizaciones previas que tienen los documentos.

Para compartir información entre distintos dispositivos móviles, es muy conveniente que la representación de datos esté en XML. De hecho, se ha dicho que XML es el estándar para compartir información entre sistemas de datos heterogéneos [Varadero, 2006]. XML es un lenguaje extensible de etiquetas. Principalmente se propone como un estándar para el intercambio de información semi-estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable. Tiene un papel muy importante en la actualidad, ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil. Uno de los usos más interesantes de XML es la capacidad para comunicar los datos y la semántica de los mismos. Si la información se representa en XML, cualquier aplicación podría escribir un documento de texto plano con los datos que estaba manejando, y otra aplicación podría ser capaz de recibir esta información y trabajar con ella. En resumen, XML representa un formato estándar de representación de datos, que soporta la interoperabilidad entre aplicaciones, aunque éstas estén escritas en distintos lenguajes [O'Reilly, 2007]. Por esa razón este trabajo de memoria utiliza archivos XML como fuente de datos para realizar las sincronizaciones.

En el escenario de las aplicaciones colaborativas móviles, compartir datos es vital para apoyar el trabajo en grupo. Por lo tanto, en este escenario de uso se vuelve absolutamente necesario contar con un sincronizador, que sea capaz de funcionar en forma transparente para el usuario final, y que permita sincronizar utilizando diferentes criterios. Ese es exactamente el objetivo de esta propuesta de memoria.

1.1. Justificación

Este trabajo se enmarca en un escenario en el cual, trabajadores móviles se desplazan entre distintas ubicaciones físicas, a fin de llevar a cabo su tarea. Estos trabajadores utilizan dispositivos móviles tales como Laptops y PDA's. La información que comparte cada individuo perteneciente a la red son documentos XML. Cada individuo puede modificar a su gusto estos documentos, aún si ellos se encuentran temporalmente desconectados del servidor y de sus compañeros. Debido a que ellos pueden modificar sus copias locales, se pueden generar inconsistencias entre las copias. Para alcanzar nuevamente un estado consistente; los trabajadores tienen que re-sincronizar sus copias.

Como ya se ha analizado, la desconexión en escenarios móviles es la norma por sobre la excepción. Por lo tanto, se deben proveer mecanismos que permitan a las aplicaciones acceder y modificar los datos compartidos en diversas condiciones de trabajo. Entonces, se hace necesario crear un método de sincronización o reconciliación de copias, en este caso, para documentos XML.

Debido a la falta de una estructura fija por parte de los documentos XML, la sincronización se vuelve una tarea nada trivial. De manera similar a cómo se mezclan

archivos de texto, los documentos XML resultantes de una sincronización, pueden llevar a diferencias conocidas como “ambigüedades”. En ese caso se requiere de la participación de “resolutores” que son los encargados de resolver las ambigüedades a la hora de sincronizar dos archivos XML [Mascolo, 2002]. Para especificar mejor este problema, se presenta el siguiente ejemplo:

```
<inventario>
  <producto>
    <descripcion>lapiz pasta</descripcion>
    <cantidad>1000</cantidad>
  </producto>
  <producto>
    <descripcion>goma de borrar</descripcion>
    <cantidad>120</cantidad>
  </producto>
</inventario>
```

Stock realizado por trabajador 1

```
<inventario>
  <producto>
    <descripcion>lapiz pasta</descripcion>
    <cantidad>990</cantidad>
  </producto>
  <producto>
    <descripcion>goma de borrar</descripcion>
    <cantidad>120</cantidad>
  </producto>
</inventario>
```

Stock realizado por trabajador 2

Figura 1: Copias locales de inventario

Un supermercado administra su inventario en bases de datos XML. Para actualizar el inventario, se tienen trabajadores que cuentan con dispositivos móviles. Estos trabajadores analizan el stock real que existe en las bodegas, cuentan los productos y luego ingresan los datos en sus dispositivos móviles, guardando la información en copias locales de la base de datos XML que contiene el inventario general. Mientras los trabajadores realizan su labor de actualización de stock, los dispositivos están desconectados de la base de datos central. Si dos trabajadores, por casualidad, actualizan en sus copias locales el inventario de un mismo producto, pero no coinciden las cantidades que ambos han contado, se produce una ambigüedad. Al momento de consolidar la información de todos los dispositivos de los trabajadores, se deberá resolver la ambigüedad decidiendo qué cantidad del producto efectivamente se debe ingresar al inventario. En la figura 1 se muestran dos documentos XML que representan los inventarios locales realizados por dos trabajadores diferentes.

La sincronización de los documentos con ambigüedades, como los del ejemplo, debe ser realizada bajo políticas específicas que se planteen en cada aplicación particular. Muchas de estas ambigüedades incluso necesitan la intervención del usuario para ser resueltas. Desafortunadamente, si se requiere intervención humana, entonces la usabilidad de la aplicación se degrada notablemente.

Considerando ese factor, este trabajo de memoria pretende diseñar un método y una aplicación de software capaz de automatizar el proceso de sincronización de los documentos

XML. Esta sincronización debe ser lo más automática posible para evitar al máximo la intervención del usuario en la misma. Este es el principal desafío a abordar dentro del trabajo propuesto.

En la actualidad existen algoritmos ya diseñados que resuelven algunas de estas ambigüedades de los documentos, al definir políticas de sincronización. Uno de los más conocidos es el algoritmo de sincronización implementado en XMIDDLE [Mascolo, 2002]. Sin embargo, este método no es capaz de resolver ambigüedades, considerando algunos de los elementos típicos que están presentes en un sistema colaborativo móvil. Como por ejemplo: la presencia de usuarios con roles, la disponibilidad de mecanismos de control de piso, etc. Haciendo uso de estas particularidades de las aplicaciones colaborativas móviles, este trabajo de memoria pretende generar un método de sincronización y un conjunto de resolutores que permitan automatizar este proceso.

1.2. Objetivos del Trabajo

El principal objetivo de este trabajo de título es generar una aplicación que sea capaz de sincronizar documentos XML en forma automática. El método de sincronización se enfocó al escenario de uso de las aplicaciones colaborativas móviles que ejecutan en PDAs, y utilizan redes inalámbricas como soporte de comunicación. Los objetivos específicos que se desprenden del objetivo general son los siguientes:

1. Diseñar un método que permita la sincronización automática de archivos XML, para ser utilizado como soporte de aplicaciones colaborativas móviles. Este método tiene que permitir múltiples criterios para la sincronización de los datos.

2. Diseñar e implementar una herramienta de software que ejecute en PDAs y notebooks, que sea capaz de ejecutar el método de sincronización diseñado. Esta herramienta fue escrita en el lenguaje C# y utiliza .Net Compact framework.

La herramienta permite la sincronización de documentos entre PDAs o entre una PDA y un PC/notebook (por ej. un servidor). Vale aclarar que las limitaciones de recursos de hardware que poseen las PDAs, hacen que este proceso deba implementarse de forma tal que la performance de la solución no tenga un impacto negativo sobre la solución.

1.3. Plan de Trabajo

El plan de trabajo que se definió para dar cumplimiento a los objetivos de la memoria es el siguiente:

1. Familiarización con el algoritmo de sincronización propuesto para XMIDDLE y análisis de su utilización en el ambiente que se plantea para el presente trabajo de título.

2. Estudio de políticas que define XMIDDLE y análisis de las mismas, aplicándolas al ambiente de uso antes especificado. De ser necesario, definición de nuevas políticas de sincronización.

3. Diseño del método de sincronización para dos documentos XML.

4. Familiarización con lenguaje .Net C#.

5. Desarrollo de aplicación de sincronización para dos documentos XML. Inicialmente la solución fue implementada para que funcione en notebooks.

6. Adaptación del método de sincronización creado, para que sea capaz de sincronizar N documentos XML.

7. Extensión de aplicación desarrollada, para N documentos y para que ejecute en PDA's.

8. Prueba de la aplicación desarrollada, análisis de resultados y ajuste en caso de ser necesario.

9. Escritura del documento final del trabajo de título.

A continuación se presenta el cronograma de actividades realizado para alcanzar los objetivos planteados.

Cronograma de actividades:

Mes / Tarea	1	2	3	4	5	6	7	8
1	XXXX	XX						
2		XXX						
3		X	XXXX					
4			XXXX					
5			X	XXXX				
6				XX	XXXX			
7					XXX	XXXX	X	
8							XXX	X
9				XXX			XX	XXXX

2. Trabajos Relacionados

Entre los trabajos más importantes realizados por otros investigadores en el ámbito de la memoria propuesta, están los siguientes:

2.1. Harmony

El sistema Harmony [Harmony, 2006] es un framework genérico escrito en el lenguaje ML, para reconciliar actualizaciones de datos heterogéneos, replicados y guardados en formato XML. La arquitectura básica de la solución que propone este framework involucra a las dos copias que se sincronizarán y a un "archivo" que representa el estado pasado común de estas reproducciones. Cuando se ejecuta Harmony, el software realiza un recorrido de árbol sobre las dos reproducciones y el archivo. Luego observa dónde ha cambiado cada réplica con respecto al archivo base, para poder propagar los cambios a la otra reproducción. Finalmente se actualiza el archivo para que refleje el nuevo estado sincronizado.

Internamente, Harmony representa todos los datos de la misma forma: como árboles desordenados. El algoritmo de sincronización de Harmony es simple: compara los nodos de los árboles desde la raíz:

- Cualquier nodo que esté presente en el archivo y en una de las reproducciones, pero no en la otra se considera borrado. Por lo tanto, el nodo correspondiente se suprime de la reproducción donde existe y también del archivo.

- Inversamente, nodos que están presentes en una reproducción y ausentes del archivo y de la otra reproducción, se consideran agregados. Estos nodos (y los sub-árboles debajo de ellos) se copian al archivo y a la otra reproducción.

- Para los nodos que están presentes en ambas reproducciones, el algoritmo procede recurrentemente en los sub-árboles correspondientes.

La única excepción a este comportamiento ocurre cuando el algoritmo se da cuenta que el árbol "mezclado" que ha construido queda mal formado con respecto al esquema de la sincronización. El esquema deseado para la sincronización se proporciona como otra entrada a Harmony. Para hacer un buen trabajo, el sincronizador tiene que poder alinear la información en las dos reproducciones (y el archivo) según un cierto criterio o esquema. Bajo este concepto, aparece la parte más interesante de Harmony, los lenses.

Los lenses son transformaciones bidireccionales entre un cierto formato de datos concreto (codificado como árbol de Harmony) y un formato más "abstracto" que "se alinee para la sincronización". Es decir, se cambia la información de modo que el algoritmo recurrente de sincronización encuentre las "mismas partes" en los sub-árboles de las dos reproducciones al mismo tiempo. Los lenses son bidireccionales por la necesidad de que cada reproducción realmente pueda ser transformada dos veces: una vez a fin de prepararla para la sincronización, y entonces otra vez cuando se acaba la sincronización, con el objetivo de transformar el "árbol abstracto actualizado" de nuevo a su formato concreto original.

Formalmente, un lens entre el formato concreto C y el formato abstracto A es un par de funciones, una que va de C a A y la otra que va de A x C a C.

Los lenses definen políticas de reconciliación entre los documentos XML y son programables. Además se pueden agregar lenses a Harmony para personalizar la reconciliación de los archivos. Existen también otros proyectos de sincronización relacionados, como “Unison File Synchronizer” [Unison, 2006] o Multisync [Multisync, 2006].

2.2. XMIDDLE

XMIDDLE es un middleware para compartir datos en computación móvil. Le permite a los dispositivos móviles compartir datos codificados en XML con otros móviles. Además pueden tener acceso a los datos compartidos cuando están desconectados, y reconciliar cualquier cambio que haga en los datos otro dispositivo.

El objetivo es asegurarse que todos los dispositivos eventualmente tendrán una versión consistente de los datos compartidos. XMIDDLE es muy liviano y rápido, y está pensado especialmente para el comportamiento de constantes desconexiones que muestran los dispositivos móviles actualmente.

El algoritmo que implementa XMIDDLE se basa en la representación de árbol de los documentos XML. XMIDDLE Basa la reconciliación de los documentos en el árbol de diferencias entre ellos.

XMIDDLE además es flexible al permitir a los programadores de aplicaciones móviles, definir sus propias políticas de reconciliación a través de Resolutores. Los Resolutores son clases que contienen un método abstracto que define el modo de reconciliación de un nodo específico de los árboles a reconciliar. El modo de uso de estos Resolutores es “marcar” el nodo de cada uno de los árboles a reconciliar con el atributo (resolutor = “nombreResolutor”) para así forzar a XMIDDLE a usar tal resolutor para la reconciliación de ese nodo en caso de ambigüedades.

XMIDDLE está programado en el lenguaje orientado al objeto, JAVA. Para tener un más acabado entendimiento del funcionamiento de XMIDDLE, a continuación se presenta un resumen de las clases más importantes que XMIDDLE implementa [Musolesi, 2002].

2.2.1. Clases: Métodos y Variables Relevantes de XMIDDLE

A continuación se describen brevemente los principales métodos y variables relevantes presentes en la plataforma.

LevelTreeDiff

Esta clase, permite comparar dos documentos XML, y retornar un archivo “diff”, el cual es un árbol XML que contiene las diferencias entre ellos. Estas diferencias tienen una notación operacional. Dados dos documentos, treeA y treeB, el archivo de diferencias especifica las operaciones que deben ser realizadas en treeA para obtener treeB.

El método principal de esta clase es compute(); que dados dos documentos, produce su diferencia. Los nodos de este árbol de diferencias pueden ser de los tipos:

- addsubtree: Se debe agregar un sub-árbol.
- delsubtree: Se debe eliminar un sub-árbol.
- changeattr: Se debe cambiar un atributo en el árbol.
- changepcdata: se debe cambiar el valor de un atributo del árbol.

Variables de instancia:

```
XmlDocument difftree: Árbol donde se guardarán las diferencias.
```

Métodos:

```
XmlDocument compute (XmlDocument base_Renamed, XmlDocument changed): Procesa las diferencias de los documentos y genera el árbol de diferencias diff utilizando los métodos a continuación descritos.
```

Métodos que generan el árbol de diferencias, según el tipo de diferencia encontrada en los archivos:

```
void attributeDiff(XmlNode base_Renamed, XmlNode changed)

void textDiff(XmlNode parent, String oldval, String newval)

void addSubtree(XmlNode parent, XmlNode subtreeroot, XmlNode leftsibling, int order)

void deleteSubtree(XmlNode subtreeroot)
```

LevelTreeMerge

El documento 'diff' que es producido por el método compute() de la clase LevelTreeDiff es usado por la clase LevelTreeMerge. Esta clase es capaz de generar el documento modificado 'treeB' que ha sido usado como argumento del algoritmo LevelTreeDiff a partir de 'treeA' y 'diff'. En otras palabras, esta clase se usa para reconstruir un documento XML desde un documento base, conociendo los cambios que han sido hechos en él.

La figura 2 ejemplifica el uso de las clases *LevelTreeDiff* y *LevelTreeMerge.*

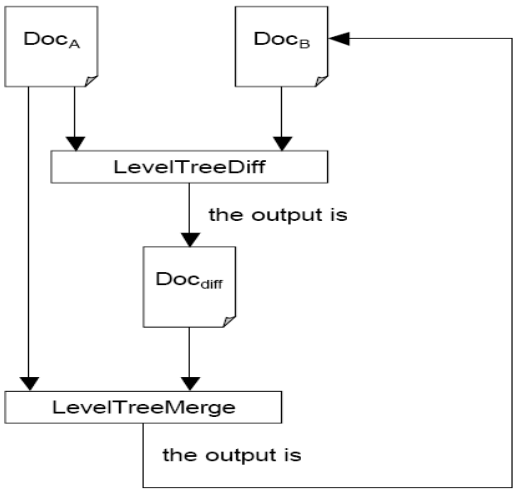


Figura 2: Uso de las clases LevelTreeDiff y LevelTreeMerge

LevelTreeReconcile

Esta clase es responsable de implementar el algoritmo de reconciliación. Su método más importante es reconcile(). Los argumentos de este método son los dos documentos XML que tiene que reconciliar, y el último documento común entre ambos. La salida es el documento reconciliado.

Usa el algoritmo de LevelTreeDiff para encontrar las diferencias entre los dos documentos XML. Los nodos del árbol de diferencias pueden ser de los tipos: agregación, eliminación, modificación de texto o modificación de atributos. Estos nodos definen qué acciones son necesarias realizar sobre el árbol A para obtener el árbol B.

Variables de Instancia:

`ArrayList resolutors: Arreglo de Resolutores para aplicar en la reconciliación.`

Métodos:

`void addResolutor(String className): Agrega Resolutores al arreglo de Resolutores.`

`XmlDocument reconcile(XmlDocument treeA, XmlDocument treeB, XmlDocument latestCommon, XmlDocument schema): Aplica la reconciliación a los árboles treeA, treeB, basándose en el`

árbol latestCommon como base.

Modo de operación:

- Obtiene la diferencia de los árboles treeA y treeB.
- Recorre el árbol de diferencias. Para cada nodo del árbol de la diferencia:
  - Si el nodo es del tipo '*addsubtree*': agrega el nodo a un vector de adiciones que serán procesadas luego.
  - Si el nodo es '*changepcdata*' o '*changeattr*' (las diferencias son del tipo modificaciones), el nodo se procesa de inmediato con el método *applyChangePCData* o *applyChangeAttr* según corresponda.
  - Si el nodo es del tipo 'delsubtree': se procesa con el método *applyDelete*.
- Se procesan las adiciones con el método *applyAdd*.

Métodos relevantes de la clase LevelTreeReconcile:

```
void applyAdd(XmlDocument treeA, XmlDocument latestCommon,
XmlElement addCommand, bool orderIsImportant):
```

treeA es el documento A, *latestCommon* es el último documento común, *addCommand* es el nodo generado por una diferencia de tipo agregación, en el árbol de diferencias.
Basándose en la información referida en la Tabla 1: Si *orderIsImportant* es *false*, o sea es un árbol desordenado y si *addCommand* está en *treeA*, pero no está en *latestCommon*: se agrega el nodo en el documento reconciliado.

```
void applyDelete( XmlDocument treeA, XmlDocument treeB,
XmlDocument latestCommon, XmlElement deleteCommand, bool
orderIsImportant)
```

deleteCommand es el nodo en el árbol de diferencias, generado por una eliminación.
Basándose en la información referida en la Tabla 2:
- Si *orderIsImportant* es true, o sea es un árbol ordenado:
  - Si addCommand está en treeA, y está en latestCommon: se elimina el nodo.
- Si *orderIsImportant* es false, árbol desordenado:
  - Si addCommand no está en treeB, pero está en latestCommon: se elimina nodo.

```
void resolveNode(XmlNode n1, XmlNode n2, XmlNode nOld)
```

Aplica el método doReconcile para resolver ambigüedades entre dos nodos. Entrega al método *doReconcile* la política con la que reconciliar los nodos. Esta política la captura del nodo n1, el cual debe contener un atributo que defina el resolutor a usar para su resolución.

`String doReconcile(String first, String second, String old, String policy)`

Crea un resolutor del tipo policy y aplica este para resolver los valores first y second. Se toma también en cuenta el valor old, que representa el valor de ese nodo en el documento común.

Resolutor

Es la clase que se encarga de resolver las diferencias de los nodos al momento de sincronizar. Permite la definición de nuevas políticas de reconciliación. Se hereda la clase Resolutor para generar un nuevo Resolutor y se define la forma de resolver una ambigüedad en el método reconcile. Este método recibe los valores de atributos de los nodos a reconciliar y devuelve un valor reconciliado.

Variables de instancia:

`String Name`: Nombre que se usará para invocar a este Resolutor en los nodos que generen ambigüedades.

Métodos:

`String reconcile(String local, String remote, String common, String type)`: Realiza la reconciliación de los nodos local y remote, basándose en el nodo common (nodo del árbol que contiene la última versión común entre local y remote). El tipo de reconciliación que se realizará para estos nodos se define en type y debe estar preseteado para el Resolutor.

2.2.2. Métodos que procesan nodos del árbol de diferencias

Para tener un entendimiento más acabado de cómo se lleva a cabo el algoritmo de resolución de ambigüedades que implementa XMIDDLE, a continuación se detallará este procedimiento.

Para este propósito resulta necesario definir el concepto de árboles ordenados y desordenados. Se define un árbol ordenado si el orden relativo entre sus ramas y sus hojas es importante. Se define un árbol desordenado si el orden relativo entre sus ramas no es importante.

Sean treeA y treeB árboles a reconciliar, y sea latestCommon la última versión común conocida entre ellos. Si T es sub-árbol de treeB (se recorre todo el árbol treeB), el algoritmo de resolución de conflictos para árboles desordenados, que implementa XMIDDLE puede definirse en la Tabla 1:

Sub-árbol T presente en treeA	Sub-árbol T presente en latestCommon	Operación a realizar
Sí	No	Eliminar sub-árbol T de treeA
Sí	Sí	Nada
No	No	Agregar sub-árbol T a treeA
No	Sí	Nada

Tabla 1: Algoritmo de reconciliación para árboles desordenados

El algoritmo de resolución de conflictos para árboles ordenados, que implementa XMIDDLE puede definirse en la Tabla 2:

Sub-árbol T presente en treeA	Sub-árbol T presente en treeB	Sub-árbol T presente en latestCommon	Operación a realizar en el Documento Reconciliado
Sí	No	Sí	No agregar sub-árbol T
No	Sí	Sí	No agregar sub-árbol T
Sí	No	No	Agregar sub-árbol T
No	Sí	No	Agregar sub-árbol T

Tabla 2: Algoritmo de reconciliación para árboles ordenados

En la tabla 2, la columna de la derecha muestra la operación que se debe realizar en el documento final reconciliado, para llevar a cabo la reconciliación.

2.3. OMA Data Synchronization y SyncML

La “Open Mobile Alliance” (Alianza Móvil Abierta) es un grupo de estándares para la industria de la telefonía móvil. OMA es un foco de desarrollo de especificaciones de servicios móviles, para apoyar la interoperabilidad de dispositivos móviles. OMA hace interfaces para que dispositivos móviles puedan interactuar independientemente de las redes a las que se encuentren conectados y las plataformas que utilicen. Dentro de sus proyectos más interesantes se encuentra SyncML. SyncML es una especificación para un framework de sincronización de datos basados en formato XML. SyncML está diseñado para usarse entre dispositivos móviles conectados intermitentemente con la red y los servicios de red disponibles. SyncML está particularmente diseñado para casos donde se almacenan datos en diversos formatos o son utilizados por diversas aplicaciones. El protocolo de

representación de SyncML está definido por un sistema de mensajes bien definidos que se transportan entre las entidades participantes en una operación de sincronización de datos. Los mensajes se representan como un documento XML [OMA, 2007].

Paquetes, Mensajes y Comandos SyncML

En SyncML, las operaciones de sincronización de datos conceptualmente se encapsulan en un ***paquete SyncML***. El paquete SyncML es simplemente uno o más ***mensajes SyncML*** que se requieren para transportar un conjunto de comandos de sincronización de datos.

Un mensaje SyncML es un documento XML bien formado, pero no necesariamente válido. El documento consiste en una cabecera, especificada por el elemento *SyncHdr,* y un cuerpo, especificado por el elemento *SyncBody.* La cabecera contiene información acerca de la ruta y versión del mensaje SyncML que se está enviando. El cuerpo es un contenedor de uno o más ***comandos SyncML***. Los comandos SyncML son elementos específicos que describen las operaciones que se deben realizar para la sincronización e incluyen los datos a sincronizar. Quien envía el mensaje se denomina "originador" y quien lo recibe, "receptor". Los mensajes se clasifican en "peticiones" y "respuestas"

Algunos comandos SyncML embebidos en mensajes "peticiones" pueden son los siguientes:

- Add: permite al originador realizar una petición para que el receptor agregue el dato que está enviando.
- Alert: permite enviar una alarma al receptor.
- Atomic: indica que el originador desea que, de los comandos que envíe, se deben realizar todas o ninguna de sus operaciones semánticas.
- Copy: indica que el originador requiere que se copie el dato que se envía.
- Delete: permite al originador pedir que el receptor elimine la información que se envía.
- Exec: indica que el dato que se envía debe ser ejecutado.
- Get: indica que el originador pide que se le envíe cierto dato.

Algunos comandos SyncML embebidos en mensajes "respuestas" pueden ser:

- Status: Indica el estado de una operación o que ha ocurrido un error al procesar cierta "petición".
- Results: retorna el resultado de una petición Get.

Los Comandos SyncML por sí solos no definen completamente la semántica de las operaciones SyncML. Por ejemplo, la adición de un documento a una base de datos con el comando "Add" puede tener una semántica muy diferente a la de agregar un requerimiento a una cola con el mismo comando. La semántica de una operación de SyncML queda determinada por el tipo de datos que se estén sincronizando. Esto significa que es posible que un originador solicite una operación a un receptor particular, que no tenga ningún sentido para él. En ese caso, el recipiente debe devolver un código de estado de error.

El protocolo de sincronización de SyncML define Roles de Cliente y Servidor:

SyncML Client: Este dispositivo contiene un agente que envía sus modificaciones al servidor. Este cliente es típicamente un teléfono móvil.

SyncML Server: Típicamente este servidor es un PC. El servidor toma el papel de centro de información actualizada para los clientes.

SyncML define distintos tipos de sincronización para el modelo cliente-servidor:

- Sincronización de 2 vías: El cliente y el servidor intercambian sus modificaciones. El cliente envía sus modificaciones primero.
- Sincronización lenta: Es una sincronización de 2 vías pero cada dispositivo envía todos sus datos, no sólo las modificaciones.
- Sincronización de 1 vía: El cliente envía sus modificaciones al servidor, pero el servidor no responde con sus modificaciones.
- Sincronización sólo de cliente: Todos los clientes envían sus modificaciones al servidor y este sincroniza. No envía información sincronizada de vuelta a los clientes.
- Sincronización sólo de servidor: El servidor envía sus modificaciones a todos los clientes, pero ellos no devuelven sus modificaciones.

Se puede notar que la sincronización definida por SyncML es más bien pensada en un modelo centralizado de información [OMA, 2007].

3. Solución Propuesta

A continuación se muestra en detalle el algoritmo implementado por la solución propuesta. Se definen ciertas restricciones o requisitos que deben cumplir los documentos a reconciliar, se describen las clases implementadas y sus métodos principales. Además se detalla la forma en que el proceso de sincronización es llevado a cabo.

3.1. Requisitos de la Solución

Para llevar a cabo una correcta sincronización, los documentos a sincronizar deben respetar cierta estructura. En particular se definen atributos que debe tener cada nodo de los documentos a sincronizar. Estos atributos son protegidos y son usados sólo por la aplicación que sincroniza, no son modificables ni accesibles por el usuario (a excepción de _resolutor, esto se detalla más adelante). Los atributos protegidos se indican con un _ (underscore) antes del nombre del atributo. A continuación se describen en detalle los atributos protegidos:

3.1.1. Atributos protegidos necesarios para la sincronización:

_id:

Indica el identificador del nodo. Cada nodo de los árboles (documentos) a sincronizar, tiene un identificador único para el árbol donde está. Estos identificadores marcan un camino, parecido a lo que describe XPATH. Para explicar mejor los identificadores de los nodos y los caminos que marcan se muestra un ejemplo en la Figura 3.

```
<inventario _id="1">
   <producto _id="1.1">
    <descripcion _id="1.1.1">lápiz pasta</descripcion>
    <cantidad _id="1.1.2">982</cantidad>
   </producto>
   <producto _id="1.2">
    <descripcion _id="1.2.1">coca cola 500cc</descripcion>
    <cantidad _id="1.2.2">10003</cantidad>
   </producto>
   ...
</inventario>
```

Figura 3: Identificador, cuerpo y niveles de nodos

En la figura 3 se muestra que todo nodo de un árbol sincronizable tiene un atributo _id. Este atributo será llamado "**identificador**". El nodo raíz tiene como identificador un número natural. Para cada nodo no raíz, su identificador está compuesto del identificador de su padre, un punto "." y un número natural que será llamado "**cuerpo**" del nodo. En el caso particular del nodo raíz, por definición su cuerpo es su identificador. En la figura 3 se destacan en gris los cuerpos de nodos.

Así, se puede marcar un camino entre la raíz y cualquier nodo del árbol. Cada nodo contiene en su identificador el “cuerpo” de cada uno de sus ancestros separados por puntos, ordenados de izquierda a derecha partiendo por la raíz. Además, la cantidad de puntos “.” del identificador de un nodo marca su profundidad o nivel en el árbol, pues dice cuántos ancestros tiene hasta su raíz.

El algoritmo de sincronización desarrollado en este trabajo de título es válido sólo para árboles “desordenados”, es decir, en los que no importa la posición (izquierda-derecha) de sus nodos en un mismo nivel. Lo que sí importa para los árboles desordenados, es el nivel de cada nodo en el árbol.

_timeStamp:

Indica el timestamp del nodo más nuevo con el cual se ha sincronizado este nodo. Se explica la manera cómo se asignan timestamp a los nodos en la figura 4.

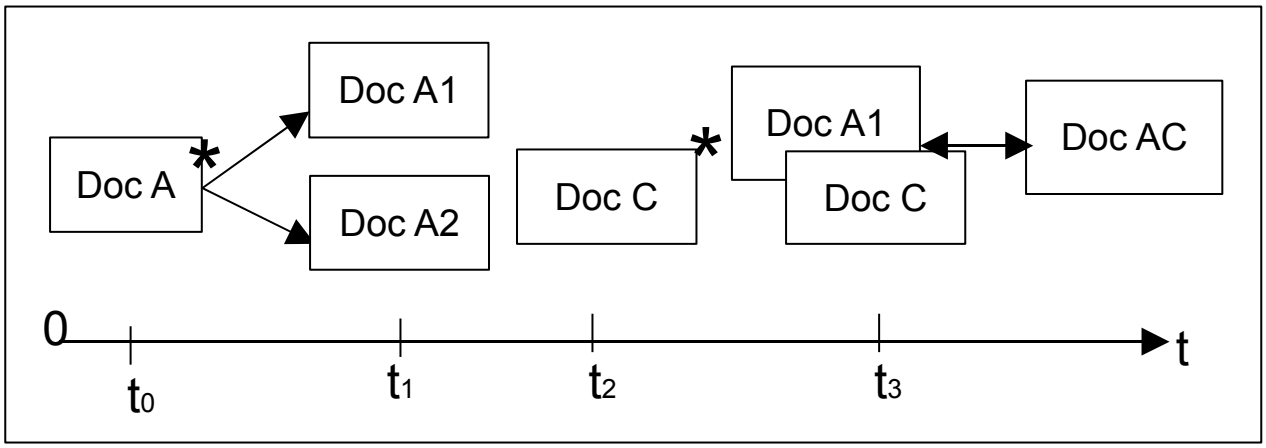


Figura 4: Asignación de timestamp

En t0 se crea DocA. Por lo tanto este documento tendrá _timeStamp=”t0”. Luego, en t1 se divide DocA en dos copias para la distribución a algunos trabajadores móviles. DocA1 y DocA2 heredan _timeStamp=”t0” de su padre DocA, pues no se les han hecho modificaciones aún. En t2 se crea DocC por lo que este documento tendrá _timeStamp=”t2”. Y finalmente en t3 se sincronizan DocA1 y DocC, resultando de esta sincronización el documento DocAC. DocAC tendrá _timeStamp=”t3” pues hereda el mayor timestamp de los documentos que se reconciliaron, o sea el timestamp del documento más nuevo. Se utiliza el timestamp para tener una idea de cuán antiguas son las modificaciones hechas a los nodos y así tener una base para la sincronización.

_role:

Indica el nivel de autoridad sobre la aplicación que tiene el usuario que creó el nodo. El _role se relaciona intuitivamente a la probabilidad que tiene un nodo a esparcir las modificaciones que se realicen sobre él. Cuando se crea un nodo en un documento, éste

hereda el rol del usuario que lo está creando. Si se sincronizan dos nodos con distinto _role el nodo sincronizado heredará el mayor rol.

_syncTimes:

Indica las veces que el nodo ha sido sincronizado. Cuando se crea un nodo, éste naturalmente tiene _syncTimes="0". Cuando se sincronizan dos nodos, el *syncTimes* resultante será el mayor *syncTimes* + 1. Es decir, se heredan el *syncTimes* del documento que ha sido más sincronizado y se le agrega a éste la sincronización actual. Este proceso se plasma en la figura 5.

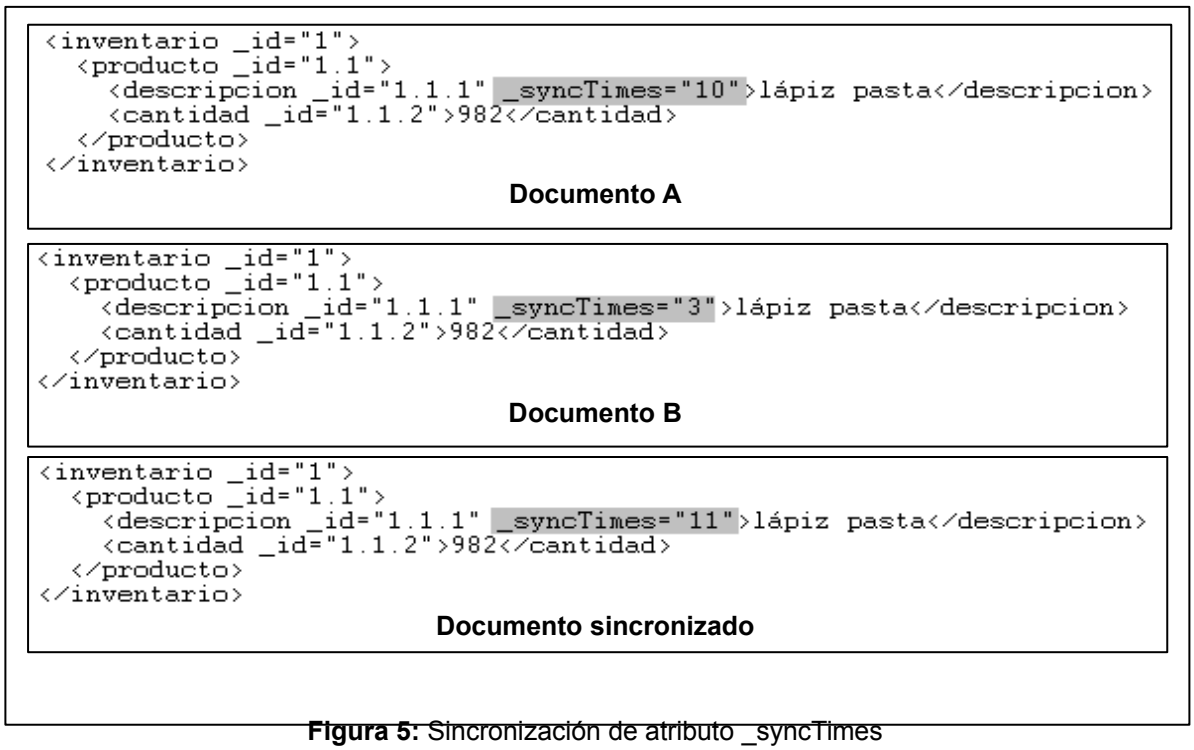


Figura 5: Sincronización de atributo _syncTimes

_visible:

El atributo _visible es usado para marcar la eliminación de los nodos. El algoritmo diseñado en este trabajo de título no considera la eliminación directa de nodos. En vez de eliminar un nodo, se le marca como "no visible" con el atributo _visible="false". Por defecto los nodos al ser creados son visibles, o marcados como _visible="true". Se utilizan resolutores para sincronizar este atributo, el nodo resultante de una sincronización, será o no visible según el resolutor que se decida utilizar.

Cuando un nodo está marcado para eliminación (_visible="false") todos sus hijos también se marcan para eliminación. Se muestra un ejemplo de un documento con algunos nodos marcados para eliminación en la figura 6.

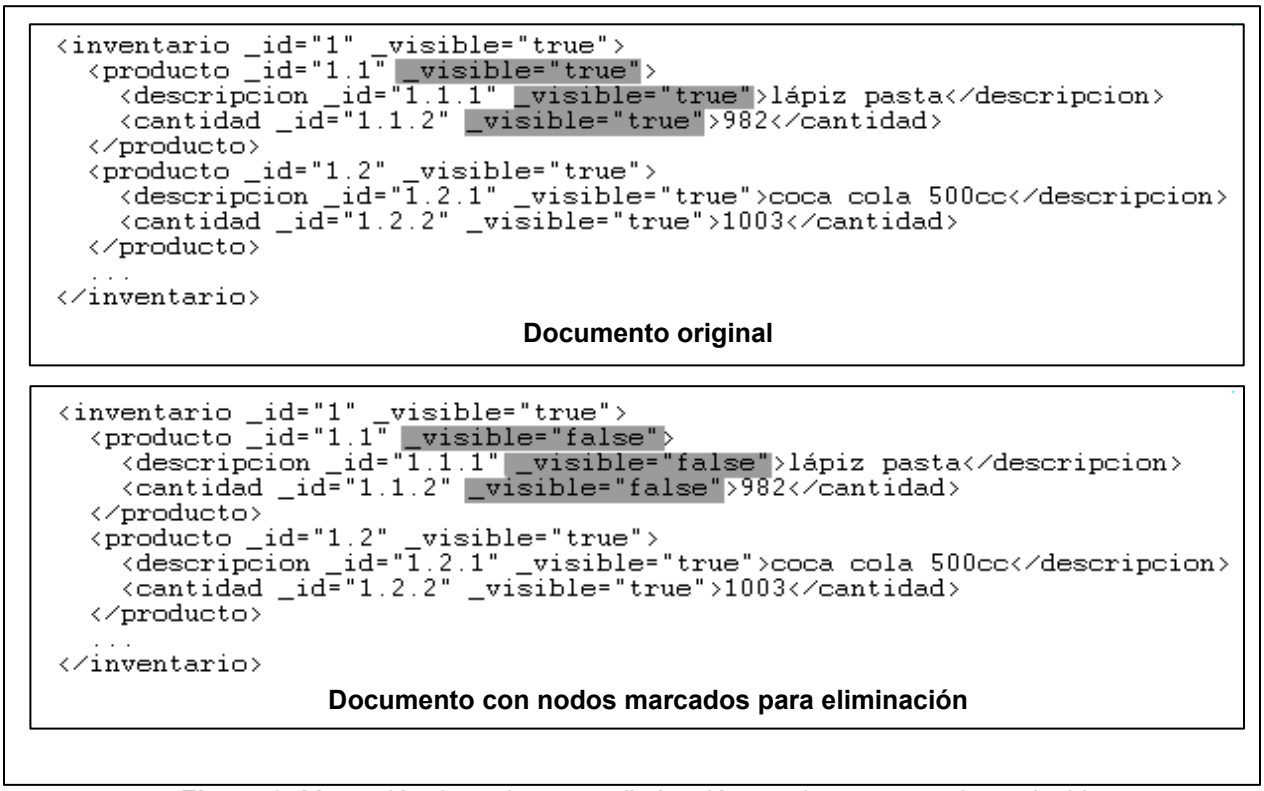


Figura 6: Marcación de nodos para eliminación, en documentos sincronizables

Como se puede ver en la figura 6, se marca para eliminación el nodo _id=”1.1” del documento original. Por consiguiente, sus hijos también son marcados para eliminación.

Se propone un método “pack” que recorra un árbol y borre sus nodos marcados para eliminación. Sin embargo, se debe proceder con mucha cautela pues éste método puede interferir con el algoritmo de sincronización propuesto por este trabajo de título. A modo de ejemplo, supóngase que se tienen dos documentos XML **docA** y **docB** que comparten un nodo (se puede extender a N documentos, bajo el mismo criterio y el problema que se presenta es el mismo). Sea **sharedNode** el nodo compartido por docA y docB. Para simplificar el lenguaje, se define como “dueño” de un documento, quien modifica el mismo. Se pueden presentar dos escenarios que generan inconsistencias en la sincronización si se ejecuta el método pack.

1. El dueño de docA marca “no visible” sharedNode, mientras que en docB todavía está “visible”: Si ejecuta pack sobre docA, se eliminará sharedNode. En una próxima sincronización entre docA y docB éste nodo será agregado al árbol reconciliado como “visible” pues sólo está presente en docB y hereda la visibilidad que tenía en él. (véase el capítulo 3.2.1 del presente trabajo de título). Si el dueño de docA tenía más privilegios que el de docB, el que sharedNode siga “visible” en el documento reconciliado es un error, dado que debería haber sido eliminado. Se podría decir que docB nunca se entera que sharedNode era un nodo compartido con docA por lo que no lo sincroniza. Entonces, la decisión de eliminar o no este nodo, no debería ser arbitraria, sino el resultado de una sincronización entre documentos.

2. El dueño de docA marca sharedNode como "no visible" y el dueño de docB también. Si se ejecuta pack sólo en uno de estos documentos, por ejemplo en docA, cuando se vuelvan a sincronizar docA y docB, se agregará sharedNode al documento reconciliado. Esto puede llevar a error, pues si ambos dueños marcaron sharedNode como eliminado, efectivamente este nodo debería haberse eliminado. Si el dueño del documento reconciliado, vuelve a marcar como "visible" sharedNode, entonces se tendría una inconsistencia.

_resolutor:

Indica el resolutor que se debe utilizar en la sincronización de este nodo. Al crear el nodo se define el resolutor que se utilizará asignando el atributo _resolutor="nombre", donde "*nombre*" es el nombre del resolutor a utilizar. Sin embargo, el atributo ***_****resolutor* puede ser modificado por usuarios con ciertos privilegios. Si se sincronizan dos nodos con distinto resolutor, se utiliza el resolutor de mayor prioridad y se hereda éste en el documento reconciliado.

En el presente trabajo de título se definieron 3 resolutores: timeStamp, role y syncTimes. Se explicará la función de cada uno, más adelante. Sin embargo, la clase Resolutor se construyó de tal manera que sea muy simple la creación de nuevos resolutores dadas las circunstancias específicas de cada ambiente de trabajo. Si se crea un nuevo resolutor, para utilizarlo en la sincronización de cierto nodo, simplemente se le asigna *_resolutor="nombre nuevo resolutor"* a tal nodo. Para mayor claridad, se muestra el uso de los resolutores creados en el presente trabajo de título en la figura 7.

_resolutor: timeStamp.

Cuando se usa este resolutor entre dos nodos, la sincronización asignará el nodo más recientemente modificado al documento reconciliado. Es decir, el resolutor *timeStamp* compara los atributos _timeStamp de los nodos a sincronizar y elige el que tenga un timeStamp mayor.

En la figura 7 se aprecian dos copias locales de un mismo documento original. Esta figura se enmarca el escenario que se había representado anteriormente, de un supermercado que lleva su inventario en una base de datos XML (Véase figura 1). Se puede apreciar que el trabajador A revisa el stock de inventario en el timeStamp ***99075***, y el trabajador B en el ***5827***. Por lo tanto, como el nodo cantidad _id="1.1.2" usa el resolutor ***timeStamp***, entonces la sincronización de ambos documentos decide que el nodo sincronizado tendrá el valor del documento A, en este caso ***982***. El atributo ***_****timeStamp* es sincronizado según la manera que se explicó anteriormente (véase la figura 4). Intuitivamente esto quiere decir que como el trabajador A revisó el stock del supermercado después de B, tiene información más actual.

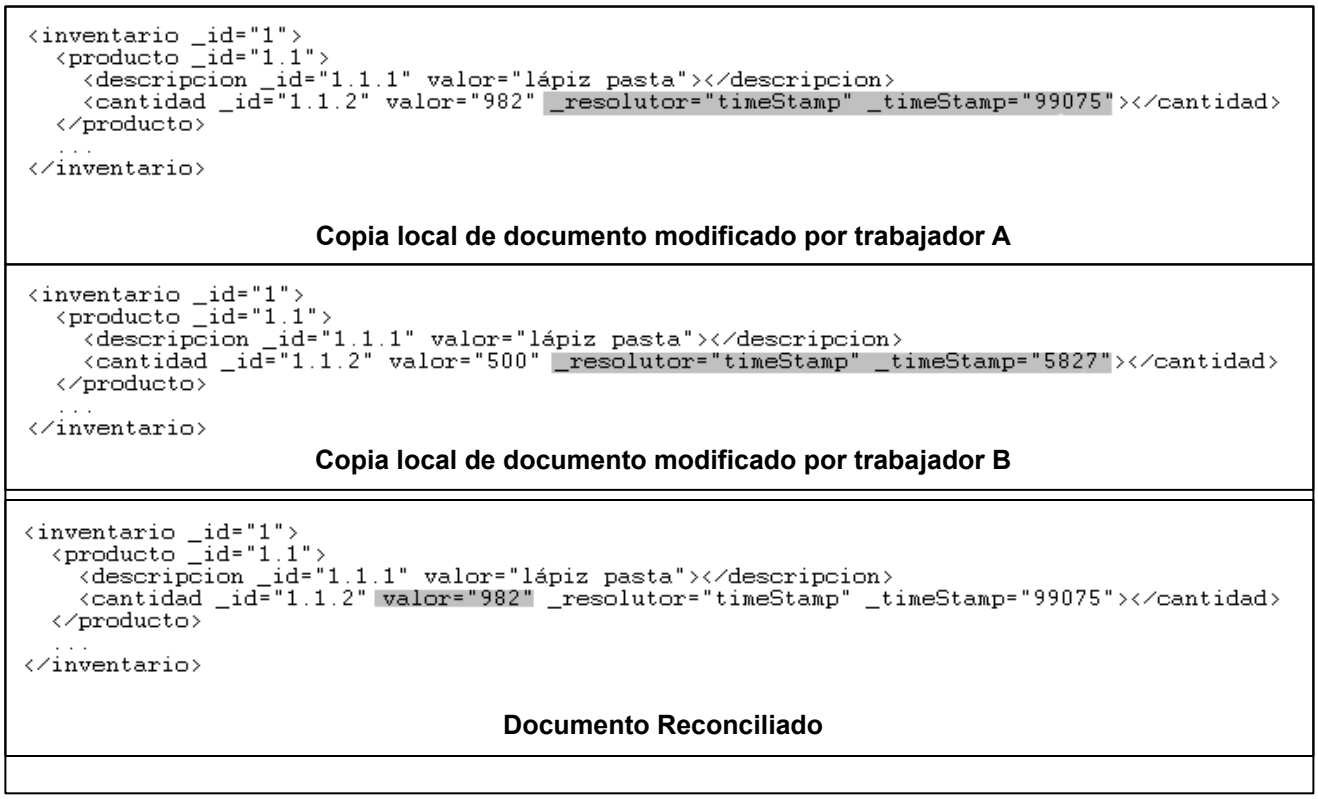


Figura 7: Uso de resolutor “timeStamp”

_resolutor: role.

Cuando se usa este resolutor entre dos nodos, la sincronización asignará el nodo con más alto rol al documento reconciliado. Es decir, el resolutor *role* compara los atributos _role de los nodos a sincronizar y elige el que tenga un role mayor.

_resolutor: syncTimes.

La sincronización de acuerdo a resolutor *syncTimes* asigna el nodo más veces sincronizado al documento reconciliado. Es decir, el resolutor *syncTimes* compara los atributos _syncTimes de los nodos a sincronizar y elige el que tenga un *syncTimes* mayor.

3.1.2. Otros atributos protegidos

Para que un documento sea sincronizable, cada nodo del árbol que lo define debe contener todos los atributos protegidos descritos. Además, se define otro atributo protegido que será usado en la sincronización, pero no es necesario para que un documento sea sincronizable.

_visited:

Indica que el nodo ha sido visitado en la presente sincronización.

3.1.3. Otras restricciones sobre documentos sincronizables

Inserción de nodos al modificar documentos XML:

Dado el esquema de identificación de nodos (véase figura 3), únicamente se permite insertar nodos finales en los documentos XML (nodos hoja). No se deben insertar nodos intermedios, pues ésto afectaría enormemente el sistema de notación e identificación de nodos. Además sólo se pueden insertar nodos bajo padres existentes. En otras palabras no se permite insertar una nueva raíz en un documento XML, pues esto haría que el documento dejara de ser sincronizable.

Atributo identificador de los nodos de documentos sincronizables:

No se permite modificar el valor del atributo identificador "_id" en los nodos. Además, no se debe cambiar el nombre a los nodos, dado que esto haría que el nodo dejara de ser sincronizable.

3.2. Diseño de la Solución Propuesta

La aplicación desarrollada se escribió en el lenguaje C#. Se basa en el algoritmo basado en XMIDDLE para la sincronización de documentos XML. Dadas las características del algoritmo para XMIDDLE, se llegó a la conclusión que ésta solución era la más idónea para satisfacer los requisitos planteados en el presente trabajo de título. En principio se intentó modificar el código de la solución XMIDDLE, para hacerlo aplicable al ambiente colaborativo móvil. Sin embargo, este planteamiento resultó demasiado engorroso. En definitiva, no era una solución viable la modificación de tal algoritmo.

Para la identificación de modificaciones hechas a las copias locales de los documentos, XMIDDLE utiliza un documento base, el cual sirve de plantilla para otras dos copias locales que se analizarán en base a él. Esta solución resulta absolutamente general para árboles ordenados y desordenados. En particular, permite sincronizar árboles ordenados, ya que identifica claramente el orden de los hermanos de un nodo.

Sin embargo, en el ambiente para el cual el presente trabajo de título es útil, no es necesario que los árboles ordenados sean sincronizables. Esto pues todos los árboles que sean sincronizados, serán desordenados. Es decir, no importaría la posición relativa de los hermanos de un nodo. Sólo importaría la profundidad de un nodo en particular.

Al notar este requerimiento menos estricto que para XMIDDLE, se decidió que una marcación de nodos con identificadores únicos para todos los documentos sincronizables, simplificaría mucho la sincronización. Es así como se llegó a la creación del atributo "_id" para cada nodo de los documentos sincronizables. Utilizando este identificador, ya no sería necesario tener un documento base que sirviera de plantilla para notar los cambios realizados a las copias locales, puesto que si un nodo era modificado o agregado, esto lo

acusaría la comparación del mismo nodo (con el mismo identificador) en la copia local de otro trabajador. Además, se utilizó el atributo "_id" para marcar "caminos" desde la raíz de un documento hasta sus hojas. Para referencias de la construcción de estos caminos véase la figura 3 anteriormente citada.

La introducción de este identificador tuvo algunas implicancias importantes en los requisitos que se exigirían a los documentos para ser sincronizables. Por ejemplo, no es posible sincronizar un nodo que no contenga el atributo _id. Esto inmediatamente hizo imposible sincronizar árboles con nodos de sólo texto. Se muestran este tipo de nodos en la figura 8.

```
<inventario _id="1">
   <producto _id="1.1">
      <descripción _id="1.1.1">lápiz pasta</descripcion>
      <cantidad _id="1.1.2">14</cantidad>
   </producto>
</inventario>
```

Figura 8: Nodos de sólo texto

En la figura 8 se puede notar que los nodos que están marcados son nodos de sólo texto, pues contienen ningún atributo. Sin embargo, esta situación no es un gran problema dado que todo documento que tenga nodos de sólo texto, puede ser trasformado a documento sincronizable. Una posible transformación puede ser borrar el nodo de sólo texto y agregar su texto como un atributo en el nodo que lo contenía. En la figura 9 se muestra este proceso. Sin embargo, este proceso de transformación no es abordado por el presente trabajo de título.

```
<inventario _id="1">
  <producto _id="1.1">
    <descripcion _id="1.1.1">lápiz pasta</descripcion>
    <cantidad _id="1.1.2">982</cantidad>
  </producto>
</inventario>
```

Documento original

```
<inventario _id="1">
  <producto _id="1.1">
    <descripcion _id="1.1.1" valor="lápiz pasta"></descripcion>
    <cantidad _id="1.1.2" valor="982"></cantidad>
  </producto>
</inventario>
```

Documento sincronizable

Figura 9: Transformación de nodos de sólo texto para documento sincronizable

Otro problema que se genera al introducir el atributo ***_id***, se produce cuando se

insertan nodos a los documentos sincronizables. Cuando un trabajador móvil actualiza su copia local de un documento y le agrega un nodo, se debe etiquetar ese nodo con un identificador único. Es necesario tener certeza (o alta probabilidad) de que el identificador para el nuevo nodo sea efectivamente único dentro de todo el grupo de trabajo.

Para resolver este problema se planteó la siguiente alternativa de solución. Como ya se ha explicado, cada identificador está compuesto del identificador de su padre y su "cuerpo" (ver figura 3 para más detalle). Por lo tanto, se puede escoger aleatoriamente el cuerpo del nodo dentro de un rango más o menos grande. Mientras más grande el rango que se escoja, más improbable será que este identificador esté repetido.

Sea W un grupo de trabajo que comparte documentos XML. Sea T={T_1... T_M} el conjuntos de árboles que representan a los documentos XML de W. Sea Ti perteneciente a T y **n** algún nodo no raíz de Ti. Sea además **nPadre,** padre de **n**. Suponiendo sin pérdida de generalidad que **D** árboles de T comparten **nPadre**, **n** tiene profundidad **p**, los árboles de T tienen grado promedio **g**, y que además el cuerpo de **n** está en el rango {1...**N**}; entonces para que exista un nodo **m** en los árboles de T que tenga el mismo identificador que **n**, se debe cumplir que:

i. **m** es hijo de **nPadre**, en cualquiera de los **nPadre** compartidos por los D árboles.
ii. **m** tiene el mismo cuerpo que **n**.

Sea P_1 la probabilidad de que la proposición i sea cierta, entonces:

$$P_1 = \underbrace{\frac{1}{(D \times g^{p-1})} + \ldots + \frac{1}{(D \times g^{p-1})}}_{D \text{ veces}} = \frac{(1 \times D)}{(D \times g^{p-1})} = \frac{1}{(g^{p-1})}$$

Sea P_2 la probabilidad de que la proposición ii sea cierta, entonces:

$$P_2 = \frac{1}{N}$$

Así, la probabilidad de que exista un nodo **m** con el mismo identificador que **n**, queda determinada por:

$$P_1 \times P_2 = \frac{1}{(N \times g^{p-1})}$$

Se puede notar que al crear un nuevo nodo, la mayor probabilidad de que el

identificador que se le asigne esté repetido, se produce cuando se insertan nodos de poca profundidad. En el caso que se inserta un nodo como hijo de la raíz, la probabilidad queda determinada por P_2. Así, mientras mayor sea el rango en el cual se elige el cuerpo de los identificadores de los nodos, menor la probabilidad que se repitan identificadores en la inserción. Cabe destacar que no está permitido insertar nodos en la raíz de los documentos.

3.2.1. Clases: Métodos y Variables Relevantes

A continuación se presenta una descripción de las clases y métodos más relevantes de la solución desarrollada. En la figura 10 se presenta el diagrama de clases minimal de la solución propuesta. Este diagrama es reducido, simplemente para ver la relación de las clases a modo general. Más adelante se mostrará el diagrama completo de clases, con sus respectivos métodos y variables.

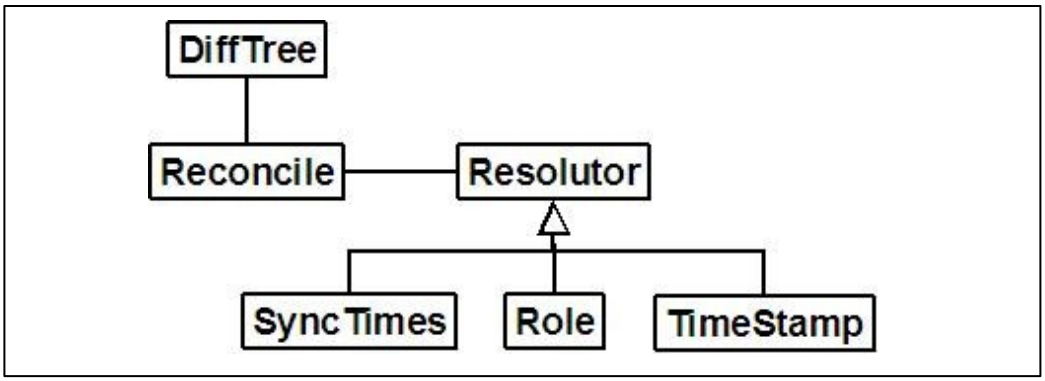


Figura 10: Diagrama de clases solución propuesta

Clase DiffTree

La clase DiffTree es la encargada de generar el árbol de diferencias entre dos documentos sincronizables. En este árbol de diferencias se plasman los nodos que, o bien, han sido agregados en las copias locales, o bien, han sido modificados en ellas. Intuitivamente si se tiene el árbol de diferencias *diff* entre dos documentos *A* y *B*, en diff se registran las acciones que se deben realizar sobre *A* (o indistintamente sobre *B*) para generar el documento reconciliado.

El árbol de diferencias que genera DiffTree se compone de dos tipos principales de nodos:

Árbol de Diferencias: Nodos MODIF del árbol de diferencias

Se dice que un *nodo n* es **"compartido"** por dos documentos XML *A* y *B* (o equivalentemente árboles A y B) si *n* está en *A* y existe otro nodo en *B* con su mismo identificador. Además se dice que *A* **"comparte"** nodos con *B* si existen nodos compartidos por ambos árboles.

Cuando el árbol A comparte nodos con el árbol B, cabe la posibilidad que cada trabajador haya modificado su copia local de estos nodos compartidos. Por lo tanto, si se encuentran nodos compartidos, se buscan diferencias entre ellos. Sea *sharedA* un nodo compartido en el *árbol A,* y *sharedB* el mismo nodo, pero en el árbol *B*. Si existen diferencias *entre* sharedA *y sharedB* se genera un nodo MODIF, y se agregan a él los siguientes atributos protegidos:

- _id: identificador del nodo compartido.
- _timeStampA y _timeStampB: timestamp de *sharedA* y *sharedB.*
- _roleA y _roleB: roles de *sharedA* y *sharedB.*
- _syncTimesA y _syncTimesB: synctimes de *sharedA* y *sharedB.*
- _visibleA y _visibleB: visibilidad de *sharedA* y *sharedB.*

Se dice que un atributo *"genera diferencias"* cuando tal atributo está presente en sharedA y *sharedB*, es no protegido *y* tiene un valor distinto en *sharedA* y *en sharedB*. Se muestra el concepto de un atributo que genera diferencias en la figura 11.

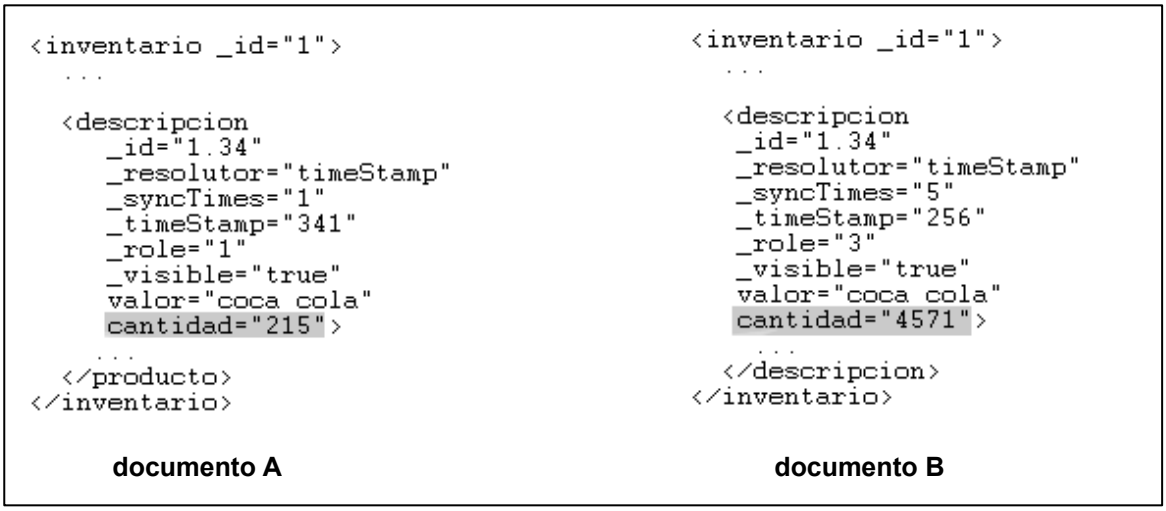


Figura 11: Atributos que generan diferencias

En la figura 11 se ve que el atributo *"cantidad"* genera diferencias, pues está en el nodo *_id="1.34"* de ambos árboles y además tiene valores distintos en cada uno de estos nodos.

Luego de los atributos protegidos, se agregan los atributos que sólo estén presentes en *sharedA* o bien en *sharedB.* También se agregan los atributos que no generan diferencias. Finalmente, los atributos que generan diferencias se insertan como hijos del nodo MODIF.

Estos hijos de MODIF tienen la siguiente forma:

- El nombre del atributo compartido se utiliza como nombre del nodo hijo.
- El valor en A y en B del atributo compartido son atributos del nodo hijo llamados *_valueA* y *_valueB*.

Para una mejor comprensión de los nodos MODIF del árbol de diferencias, se ejemplifica la creación de uno de ellos en la figura 12.

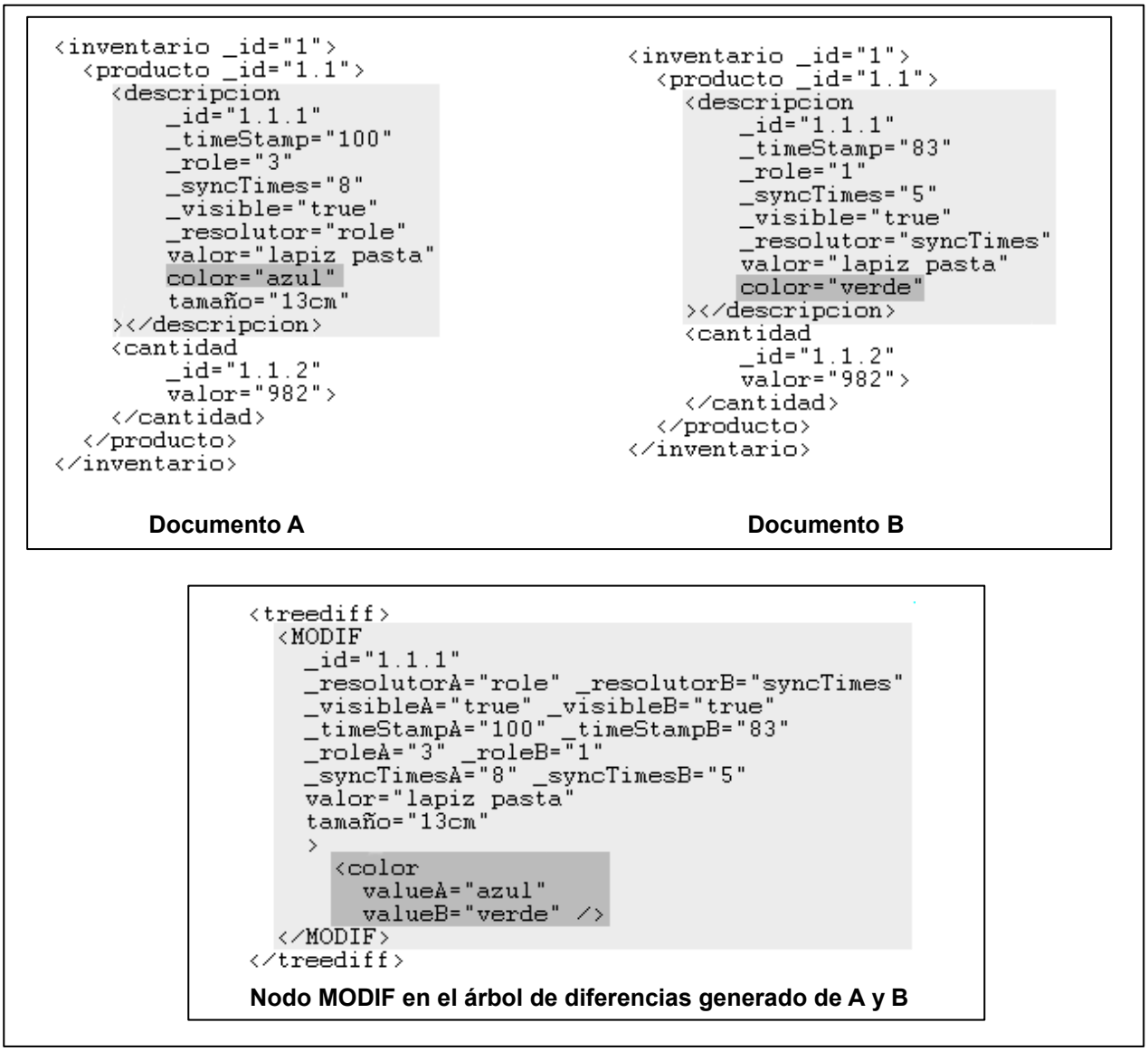


Figura 12: Nodo MODIF, atributos e hijos

En la figura 12 se aprecian dos documentos que comparten los nodos *descripcion* con _id=”1.1.1”, *cantidad* con _id=”1.1.2” y *producto* con _id=”1.1”. De éstos, el único nodo que tiene modificaciones es *descripción* _id=”1.1.1”, destacado en color gris claro. Este atributo pasa a tomar el papel de *sharedA* en el árbol *A* y *sharedB* en el árbol *B*. Se puede ver que el

único atributo que genera diferencias es *color,* pues tiene distintos valores en *sharedA* y *sharedB*. Por otra parte, el atributo *valor* no genera diferencias, aún cuando es compartido por *sharedA* y *sharedB*, pues tiene el mismo valor en ambos nodos. Finalmente, como el atributo *tamaño* no es compartido por *sharedA* y *sharedB*, sino sólo está presente en uno de estos nodos, tampoco genera diferencias y por lo tanto se agrega a MODIF. Así, el nodo MODIF que se genera en el árbol de diferencias (abajo en la figura), tiene los atributos protegidos de A y B, el atributo *valor* y el atributo *tamaño*.

Como hijo del nodo MODIF queda un nodo con el nombre *color* pues éste es el nombre del único atributo que generó diferencias. El nodo *color* tiene como atributos los valores que toma *color* en los nodos de *A* y *B*. Esto se plasma en gris oscuro en la figura 12.

Árbol de Diferencias: Nodos ADD del árbol de diferencias

Se dice que un nodo *n* es **"no compartido"** por los documentos XML *A* y *B* si n está en A y no se puede encontrar un nodo en *B* con su mismo identificador. Cuando existen nodos no compartidos por los árboles A y B, se generan nodos ADD en el árbol de diferencias. Se crea un nodo ADD por cada nodo no compartido por A y B. Sea *notShared* un nodo no compartido por A y B. Entonces se generará un nodo ADD que tiene un sólo atributo, *_idPadre*. Este atributo *_idPadre* muestra el identificador del nodo que era el padre de notShared antes de la reconciliación. Es bajo este nodo que se debe insertar *notShared* en el árbol reconciliado. Luego, se agrega *notShared* como hijo de ADD. Para más claridad, se muestra la creación de un nodo ADD en la figura 13.

En la figura 13 se ve que el documento B tiene 3 nodos no compartidos con A: *producto* con _id="1.2", *descripción* con _id="1.2.1" y *cantidad* con _id="1.2.2". Por lo tanto, estos 3 nodos generan nodos ADD en el árbol de diferencias.

```
<inventario _id="1">
  <producto _id="1.1">
    <descripcion _id="1.1.1" valor="goma" />
    <cantidad _id="1.1.2" valor="56" />
  </producto>
</inventario>
```

Documento A

```
<inventario _id="1">
  <producto _id="1.1">
    <descripcion _id="1.1.1" valor="goma" />
    <cantidad _id="1.1.2" valor="56" />
  </producto>
  <producto _id="1.2">
      <descripcion _id="1.2.1" valor="cuaderno" />
      <cantidad
        _id="1.2.2"
        _resolutor="timeStamp"
        _timeStamp="5138"
        valor="13"
      />
  </producto>
</inventario>
```

Documento B

```
<treediff>
  <ADD _idPadre="1">
    <producto _id="1.2" />
  </ADD>
  <ADD _idPadre="1.2">
    <descripcion _id="1.2.1" valor="cuaderno" />
  </ADD>
  <ADD _idPadre="1.2">
    <cantidad _id="1.2.2" _resolutor="timeStamp" _timeStamp="5138" valor="13" />
  </ADD>
</treediff>
```

Nodos ADD en el árbol de diferencias generado

Figura 13: Nodos ADD en árbol de diferencias

Compute: Método principal de la clase DiffTree

El método principal de la clase DiffTree es **compute(A,B)** el cual arma el árbol de diferencias entre los documentos *A* y *B*. Compute se puede dividir en dos pasos principales: En el primer paso se recorre el árbol *A* con el método **processTreeA**.

El método processTreeA realiza un recorrido en amplitud del árbol *A*. Sea **nodoA** el nodo del árbol *A* que se está recorriendo, y **nodoB** un nodo en el árbol *B* con el mismo identificador de nodoA (es decir, es el mismo nodoA pero en otro árbol), entonces se puede esquematizar el método processTreeA en la tabla3:

¿nodoA está en árbol B?	¿nodoB es igual a nodoA?	Operación que processTreeA realiza en árbol de diferencias.
Sí	No	Agregar nodo MODIF al árbol de diferencias.
Sí	Sí	No se realiza acción sobre el árbol de diferencias.
No	-	Agregar nodo ADD al árbol de diferencias.

Tabla 3: ProcessTreeA

Es importante notar que cada vez que se visita un nodo en *A*, si se encuentra nodoB correspondiente a él, nodoB se marca como *"_visited"*.

El segundo paso del método compute es **processTreeB**. Puesto que processTreeA ya ha agregado al árbol de diferencias todos los nodos compartidos por A y B (que tenían modificaciones), y además todos los nodos que sólo estaban en A; lo que falta por agregar sólo son los nodos no compartidos que existen en B.

El método processTreeB realiza un recorrido en amplitud al árbol *B.* Sea **nodoB** el nodo que se está recorriendo en el árbol B. Si nodoB está marcado como _visited="true", es decir fue visitado por processTreeA, no genera acción sobre el árbol de diferencias. Por otro lado, si nodoB no ha sido visitado, quiere decir que no existe en el árbol A (no es nodo compartido); por lo tanto genera acción ADD en el árbol de diferencias. Finalmente, se borran los atributos "_visited" de los nodos, dado que sólo se utilizan para el proceso de creación del árbol de diferencias. Se muestra un resumen de processTreeB en la tabla 4.

¿nodoB fue visitado por processTreeA?	Operación que processTreeB realiza en árbol de diferencias.
Sí	Agregar nodo ADD al árbol de diferencias.
No	No se realiza acción sobre el árbol de diferencias.

Tabla 4: processTreeB

Propiedad de Conmutación del árbol de diferencias

Dados dos documentos XML sincronizables *A* y *B*, se dice que el árbol de diferencias entre ellos es ***conmutable*** dado que cumple que diffTree(*A,B*) posee exactamente los mismos nodos y a la misma profundidad que diffTree(*B,A*). Es importante notar que la conmutación no considera el orden relativo entre las ramas del árbol de diferencias, sino sólo

sus nodos y la profundidad de ellos.

Se puede mostrar la *conmutación* del árbol de diferencias por las operaciones que ejecuta el método compute, que genera el árbol de diferencias. Sean los documentos XML *A* y *B* sincronizables. En la figura 14, se muestran los documentos *A* y *B.* Los nodos marcados con la letra “A”, solamente están presentes en el árbol *A;* los marcados con la letra “B” sólo están presentes en *B,* los nodos compartidos que no han sido modificados están en blanco y marcados con “AB”; y los nodos compartidos que han sido modificados se muestran en gris y marcados con “AB”. Se intentará mostrar cómo compute(*A,B*) y compute(*B,A*) van modificando los árboles de diferencias que vayan generando, para mostrar que después de ejecutados, ambos métodos finalizan con el mismo árbol de diferencias. Si efectivamente, los árboles de diferencias de compute(*A,B*) y compute(*B,A*) tienen los mismos nodos a la misma profundidad, entonces se habrá mostrado que el árbol de diferencias generado por *compute* es *conmutable.*

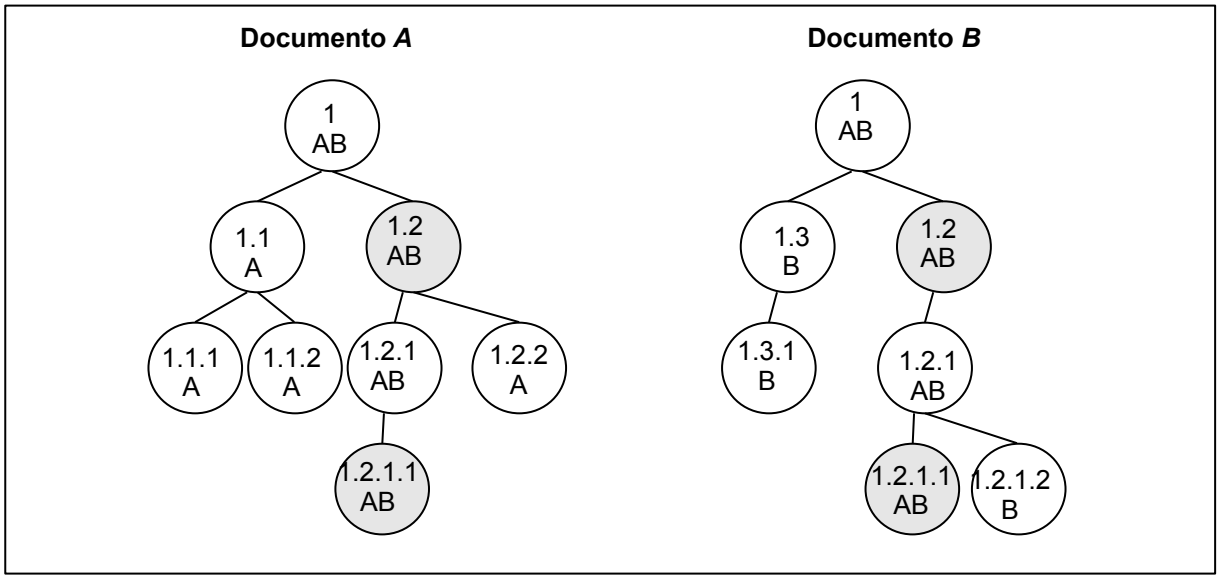


Figura 14: Documentos XML sincronizables *A* y *B*

En la figura 15 se muestra el resultado de aplicar el primer paso de *compute:* processTreeA, sobre los documentos *A* y *B*. Se muestra el estado de los árboles de diferencia que genera la ejecución de compute(*A,B*) y compute(*B,A*). En el caso de compute(*A,B*) processTreeA se ejecuta sobre el documento *A,* pero en el caso de compute(*B,A*) se ejecuta sobre el documento *B*. processTreeA(*A*) agregará al árbol de diferencias todos los nodos que sólo estén en *A* y los nodos compartidos que tengan modificaciones. processTreeA(*B*) agregará los nodos que sólo estén en B y los compartidos que tengan modificaciones.

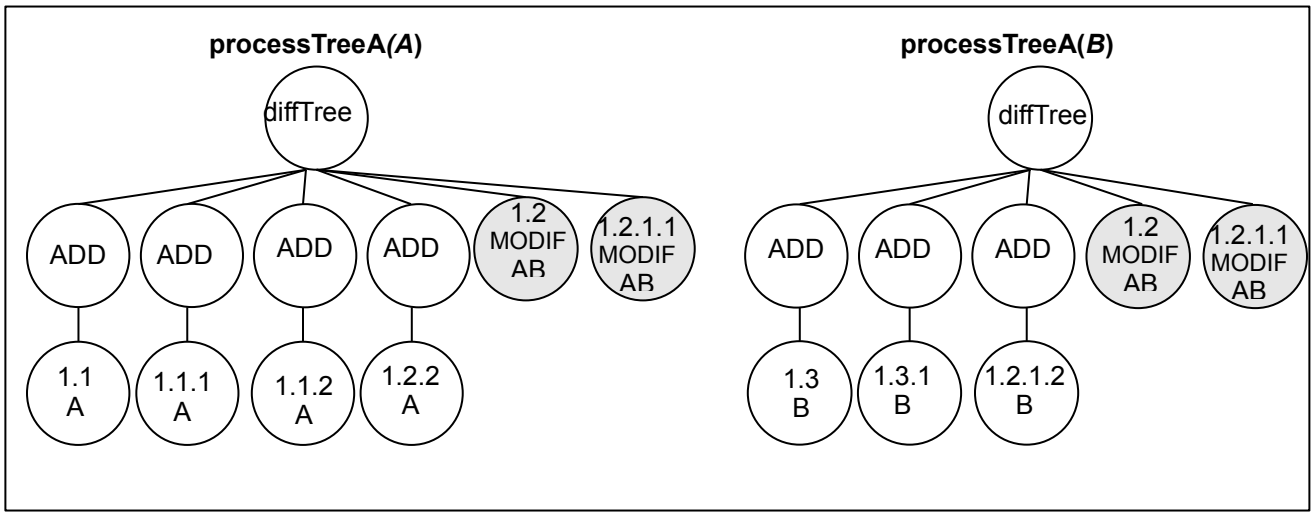


Figura 15: processTreeA en compute(*A,B*) y compute(*B,A*)

El último paso de *compute* es *processTreeB.* En la figura 16 se muestran los árboles de diferencias generados por compute(*A,B*) y compute(*B,A*) al finalizar el procedimiento *processTreeB*. En el caso de compute(*A,B*) processTreeB se ejecuta sobre el documento *B,* pero en el caso de compute(*B,A*) se ejecuta sobre el documento *A*.

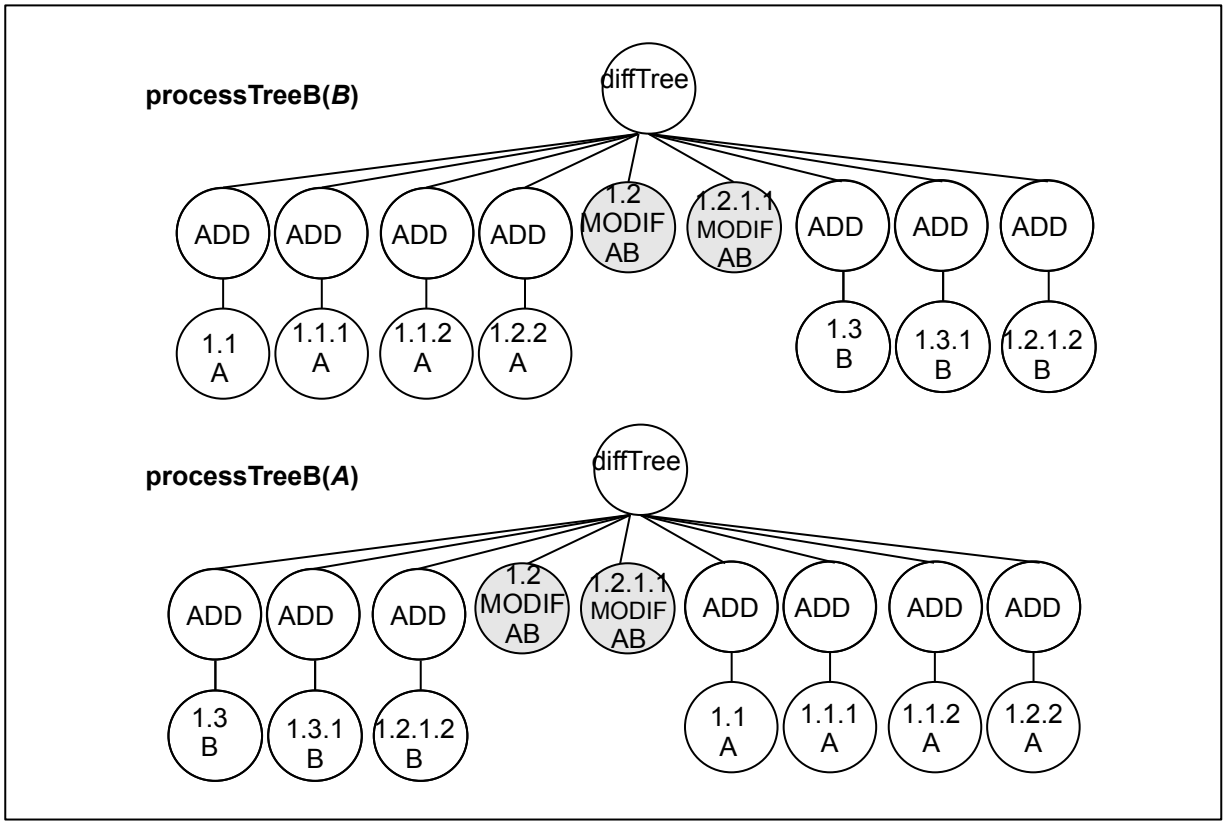


Figura 16: processTreeB en compute(*A,B*) y compute(*B,A*)

Se ve en la figura 16 que processTreeB(B) agrega los nodos que sólo están en el árbol *B* al árbol de diferencias. Y por su parte, processTreeB(A) agrega los nodos que sólo están en el árbol *A.* Se ve que después de terminado el proceso de *compute* los árboles de diferencias producidos por compute(*A,B*) y compute(*B,A*) tienen exactamente los mismos nodos y a la misma profundidad cada uno. Se ve entonces que el árbol de diferencias es *conmutable.*

Clase Reconcile

La clase Reconcile es la encargada de llevar a cabo el proceso de reconciliación o sincronización basado en el árbol de diferencias generado por DiffTree. Posee una lista de resolutores, que utilizará para la reconciliación de los documentos XML. Estos resolutores tienen una política de reconciliación que se verán en detalle en la sección ***Clase Resolutor***.

El método encargado de la reconciliación es **reconcile(A,diff(A,B))** que recibe como parámetros un documento *A* y la diferencia de *A* con el otro documento *B* a reconciliar *diff(A,B)*. El método reconcile recorre *Top-Down* el árbol de diferencias y para cada nodo ADD o MODIF que encuentre, realiza una acción sobre el árbol *A*. Como anteriormente se había dicho, el árbol de diferencias contiene las acciones a realizar en el árbol *A* para obtener el árbol reconciliado entre *A* y B, por este motivo, cuando se hable de *"árbol reconciliado"* se entenderá que se hace referencia al árbol *A* siendo modificado para obtener la reconciliación. A continuación se detallan las acciones a realizar en *A* para nodos de tipo ADD o MODIF:

Procesamiento de Reconcile: Acción nodo ADD en el árbol de diferencias

La presencia de un nodo ADD en el árbol de diferencias, indica que se debe insertar un nodo en el árbol reconciliado. Como ya se había visto, cada nodo ADD en el árbol de diferencias tiene un atributo *_idPadre*. Sea *insertNode* el nodo que debe ser insertado en el árbol reconciliado. Entonces *_idPadre* representa el identificador del nodo bajo el cual insertar *insertNode* en el árbol reconciliado.

Sea *padre* el nodo identificado por *_idPadre* en el árbol reconciliado. Si no existiera *padre* en el árbol reconciliado, habría un problema, pues no se tendría un padre bajo el cual insertar *insertNode.* Sin embargo, no es posible que no exista tal nodo al momento de realizar la sincronización sobre *insertNode.*

Naturalmente *padre* tiene que estar o bien, en el árbol *A,* o bien en el árbol B. No en ambos, pues si hubiese estado en ambos no se habría generado nodo ADD sino MODIF en el árbol de diferencias. Por lo tanto, se pueden dar dos casos:

- *padre* pertenecía al árbol *A* antes de la sincronización: En este caso no existe la probabilidad que no se encuentre *padre* en el árbol reconciliado, pues éste árbol es el mismo que *A*.

- *padre* pertenecía al árbol *B* antes de la sincronización: Si éste es el caso, entonces *insertNode* también pertenecía al árbol *B,* pues es hijo de *padre.* Primeramente, hay que tener en cuenta que processTreeB recorre el árbol B en amplitud. En la figura 17 se grafica el recorrido que realiza éste método sobre el árbol *B.*

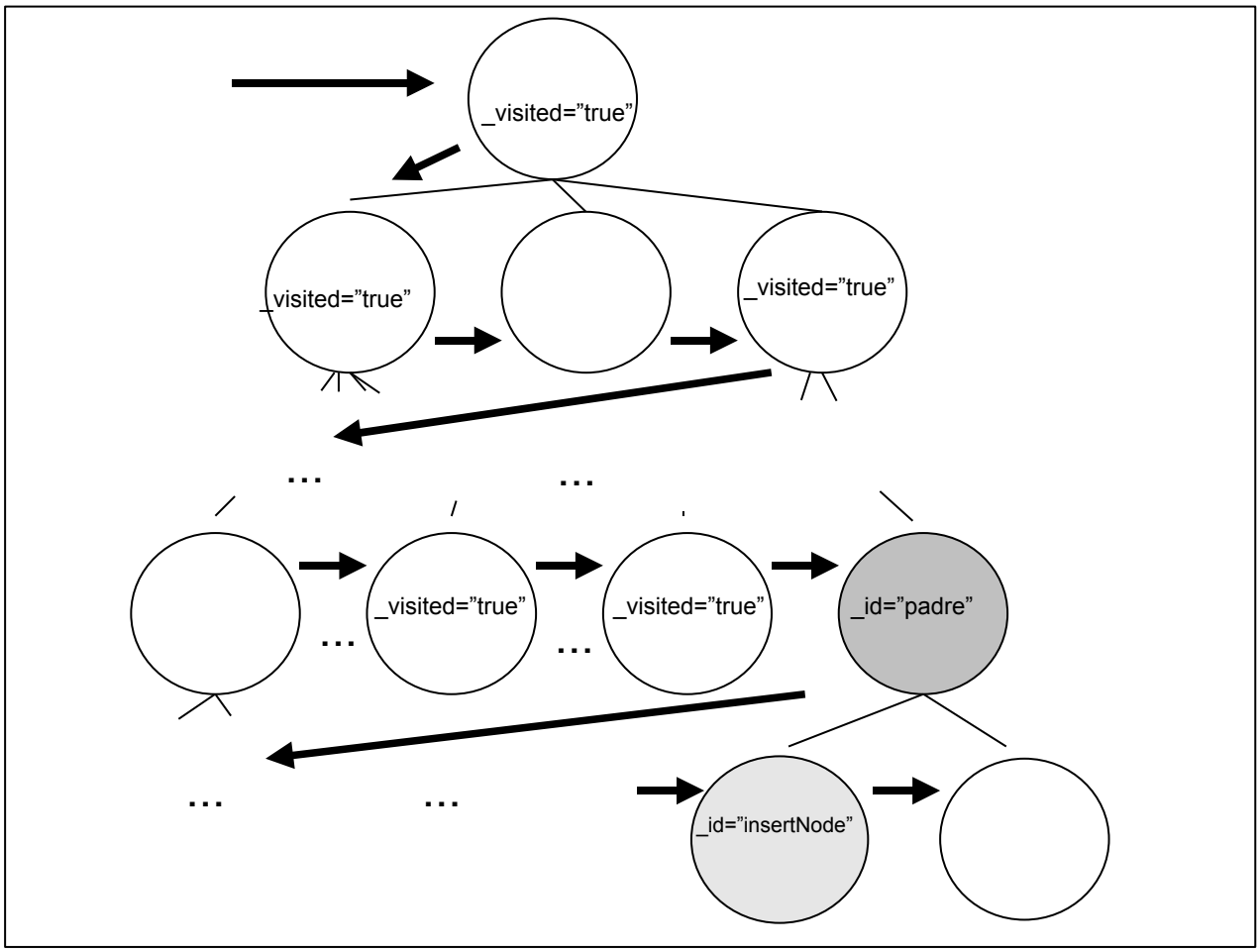


Figura 17: Recorrido en amplitud de processTreeB

Si se da el caso que processTreeB encuentra un nodo no visitado, o no marcado como _visited="true", quiere decir que este nodo no se encontraba en el árbol *A* y por lo tanto genera acción ADD sobre el árbol de diferencias. Se puede ver que los nodos *padre* e *insertNode* en la figura 17, no están marcados como _visited="true", pues sólo pertenencen al árbol *B.* Se nota además que processTreeB procesa el nodo *padre* ***antes*** que *insertNode.* Por consiguiente, en el árbol de diferencias se insertará primero el nodo ADD que referencia a *padre* y después el que referencia a *insertNode*. Se creará un árbol de diferencias de la forma que se plasma en la figura 18.

```
<treediff>

  ...

  <ADD _idPadre="abuelo">
    <padre _id="padre" />
  </ADD>

  ...

  <ADD _idPadre="padre">
    <insertNode _id="insertNode"/>
  </ADD>
  ...

</treediff>
```

Figura 18: Posición relativa entre nodos ADD en el árbol de diferencias

Se ve en la figura 18 que el nodo ADD que referencia a *padre* está antes que el que referencia a *insertNode*. En el árbol, entonces el nodo ADD de *padre* estará más a la izquierda que el de *insertNode.* Como el método reconcile realiza un recorrido Top-Down del árbol de diferencias, se procesará primero el nodo ADD de *padre* que el de *insertNode.* Esto se muestra en la figura 19.

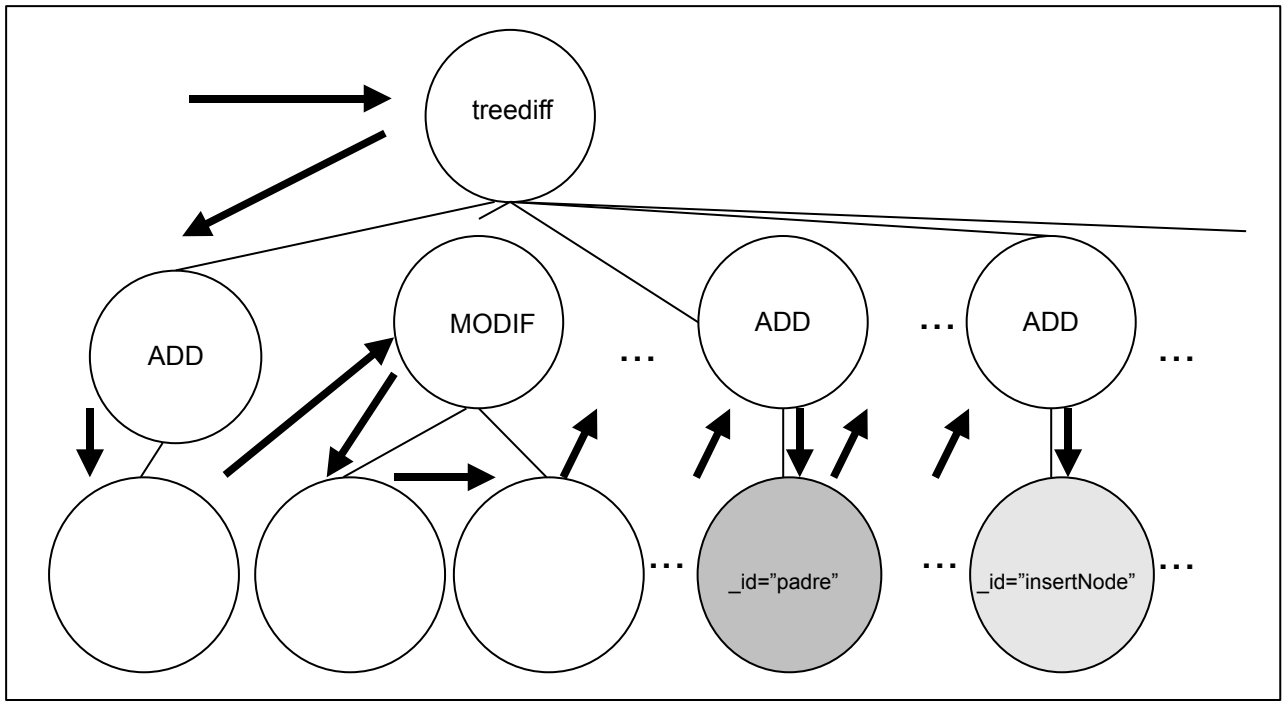


Figura 19: Recorrido Top-Down de reconcile sobre el árbol de diferencias

Se puede ver en la figura 19 que el recorrido Top-Down de reconcile sobre el árbol de diferencias, hace que para cuando se procesa el nodo ADD que referencia a *insertNode* **ya se ha insertado** el nodo *padre* en el árbol reconciliado, pues el nodo ADD que lo referencia ya ha sido procesado. Con todo esto se concluye que *padre* siempre existirá en el árbol reconciliado **antes** de insertar cualquiera de sus hijos en él.

Cuando reconcile se encuentra con un nodo ADD, busca el nodo con identificador *_idPadre* en el árbol reconciliado. Siguiendo con la notación adoptada, sea *padre* el nodo en el árbol reconciliado con identificador *_idPadre* y sea *insertNode* el nodo hijo del nodo ADD que se está recorriendo. Si *insertNode* ya está en el árbol reconciliado, no se inserta. Esto quiere decir que *insertNode* pertenecía al árbol *A* antes de la reconciliación y por tanto no es necesario efectuar cambios. Si *insertNode* no está en el árbol reconciliado, se inserta bajo *padre.* Así finaliza la reconciliación del nodo ADD.

Procesamiento de Reconcile: Acción nodo MODIF en el árbol de diferencias

La presencia de un nodo MODIF en el árbol de diferencias, significa que existe un nodo que está compartido por los árboles *A* y *B* y que por lo tanto, debe ser sincronizado con una política definida por un resolutor.

El método reconcile rescata el atributo *_id* del nodo MODIF, y luego busca en el árbol reconciliado el nodo que posea tal identificador. Este atributo indica qué nodo fue modificado y por tanto debe sincronizarse. Siempre existirá un y sólo un nodo identificado con *_id* en el árbol reconciliado, pues MODIF es generado por nodos compartidos por *A* y *B*. Además los identificadores de nodos son únicos. Sea *shared* el nodo identificado por *_id* en el árbol reconciliado. Es sobre *shared* que se ejecutará la reconciliación. El proceso de sincronización de *shared* se puede dividir en 3 pasos:

- *paso 1:* Primero se agregan a *shared* los atributos no protegidos del nodo MODIF. Cabe notar que todos los atributos no protegidos de MODIF no generan diferencias (véase figura 12). Si alguno de estos atributos ya estaba presente en *shared,* no se agrega.

- *paso 2: S*e sincronizan los atributos protegidos del nodo MODIF. En la tabla 5 se muestra la sincronización de cada uno de los atributos protegidos:

Atributo	Método de sincronización
_resolutor	Se escoge el resolutor con mayor prioridad según prioritizeResolutors
_id	Se deja igual
_timeStamp	Se escoge el mayor _timeStamp
_role	Se escoge el mayor _role
_syncTimes	Se deja el mayor _syncTimes + 1
_visible	Se sincroniza de acuerdo al resolutor utilizado

Tabla 5: Sincronización de atributos protegidos

- *paso 3:* Una vez agregados los atributos protegidos y los que no generan diferencias, se sincronizan los atributos que generan diferencias. En el árbol de diferencias, estos atributos son los hijos del nodo MODIF. Cada hijo del nodo MODIF es un atributo que genera diferencias. El método de la clase Reconcile que realiza la sincronización de estos atributos, es **doReconcile**.

DoReconcile: resolución de conflictos de nodos MODIF

Sea *sharedA* un nodo compartido en el *árbol A,* y *sharedB* el mismo nodo (con el mismo identificador que *sharedA*), pero en el árbol *B*. Para realizar la reconciliación de atributos de *sharedA* y *sharedB* que generan diferencias, doReconcile recibe 5 parámetros:

- *valorA*: Es el valor del atributo en *sharedA*
- *valorB*: Es el valor del atributo en *sharedB*
- *criterioA:* Es el valor que toma el resolutor escogido para la sincronización, en *sharedA*
- *criterioB:* Es el valor que toma el resolutor escogido para la sincronización, en *sharedB*
- *resolutor:* Es la política de resolución que se utilizará para llevar a cabo la presente sincronización.

Lo primero que hace el método doReconcile es instanciar un Resolutor que contenga la política "*resolutor*". En la subsección ***Clase Resolutor*** dentro de este mismo capítulo, se explica cómo se instancian los resolutores de acuerdo a políticas particulares. Una vez instanciado tal Resolutor, doReconcile invoca el método Resolutor.reconcile para que tal Resolutor se encargue de resolver las diferencias entre *valorA* y *valorB*. Finalmente, doReconcile devuelve la respuesta del Resolutor en forma de String, que contiene la sincronización entre *valorA* y *valorB* dados *criterioA* y *criterioB.* Este String es asignado al atributo que generó diferencias entre *sharedA* y *sharedB* para luego insertar este atributo ya sincronizado a *shared* en el árbol reconciliado.

A continuación se ilustra una reconciliación de un nodo MODIF, paso a paso, usando *_resolutor="timeStamp"*. La figura 20 muestra los documentos a reconciliar y el árbol de diferencias generado.

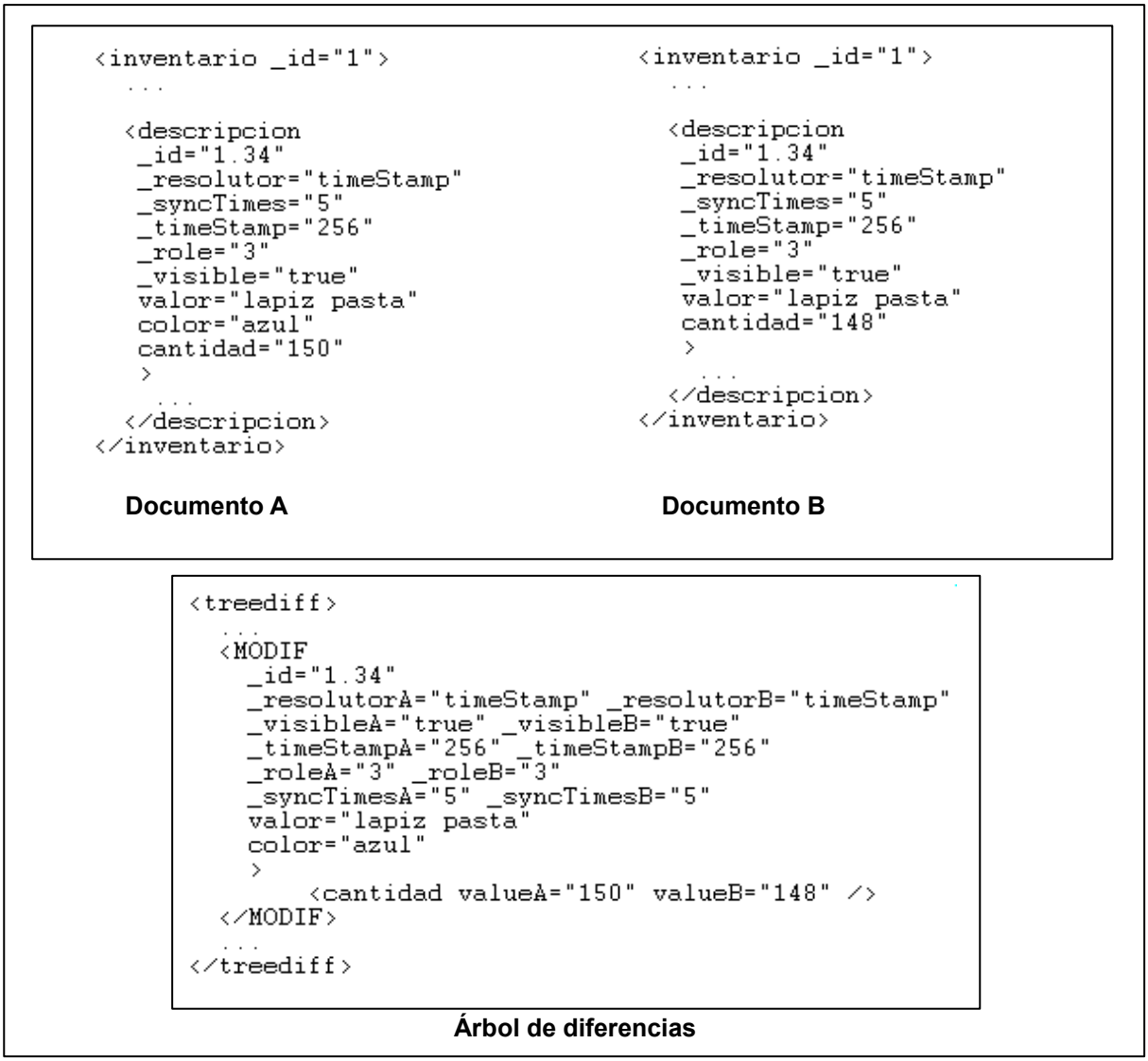


Figura 20: Documentos a reconciliar y árbol de diferencias

En la figura 20 se representa un escenario donde dos trabajadores han realizado un inventario. Se puede apreciar que para el nodo *_id="1.34"* el trabajador *A* ha rescatado más información que el trabajador *B,* al registrar además de la *cantidad* de lápices pasta que encontró, el *color* de éstos. Se nota además que el trabajador *A* ha registrado 150 lápices pasta, mientras el trabajador *B*, sólo 148. Por lo tanto, el atributo *cantidad* genera diferencias.

El primer paso de la reconciliación, como ya se dijo, se realiza sobre los atributos no

protegidos del nodo MODIF. Se agregan estos atributos al nodo reconciliado. En la figura 21 se muestra esto para el nodo *_id="1.34".*

Figura 21: Reconciliación, paso 1. Atributos no protegidos de MODIF

En el segundo paso, se agregan los atributos protegidos del nodo MODIF. Se sincronizan los atributos protegidos y luego se agregan al nodo reconciliado. Esto se ve en la figura 22.

Figura 22: Reconciliación, paso 2. Atributos protegidos de MODIF

Finalmente, se sincronizan los atributos que generan diferencias con el método doReconcile y se agregan al nodo reconciliado. Esto se ve en la figura 23.

Figura 23: Reconciliación, paso 3. Atributos que generan diferencias

La forma en que el Resolutor resuelve las diferencias de los atributos, se verá más adelante, bajo la subsección ***Clase Resolutor***.

Caso particular de reconciliación de nodo MODIF : atributo protegido "_visible"

El atributo *_visible* indica si un nodo es visible o no. Si un nodo está marcado *_visible="false"* se dice que está "marcado para eliminación". Este nodo se considerará borrado del árbol que lo contiene. La sincronización de este atributo no es realizada de la misma manera que los demás atributos protegidos. Este atributo se sincroniza considerándolo como no protegido que genera diferencias. Así, se podría decir que se aplica el paso 3 de la reconciliación sobre *_visible*. Sin embargo, no se crea un hijo en MODIF para la reconciliación de este atributo, sino que se resuelve directamente con *doReconcile* mientras se está ejecutando el paso 2 para atributos protegidos. Así, la llamada a *doReconcile* para reconciliar el atributo *_visible* sería:

doReconcile(visibleA,visibleB,criterioA,criterioB,resolutor)

donde *visibleA* es la visibilidad de *sharedA, visibleB* es la visibilidad de *sharedB*, y *criterioA, criterioB* y *resolutor* los criterios del resolutor a utilizar.

Cuando un nodo es marcado para eliminación, todos sus descendientes en el documento reconciliado deben también ser marcados para eliminación.

Asociación de la reconciliación

El método encargado de generar el documento reconciliado es *reconcile.* Se dice que la reconciliación es ***asociable***, por cuanto se cumple que: para *A* y *B* documentos XML sincronizables, reconcile(*A,diffTree(A,B)*) tiene exactamente los mismos nodos, a la misma profundidad que reconcile(*B,diffTree(A,B)*).

Si se desea reconciliar los documentos XML *A* y *B,* el método *reconcile* recibe como parámetros uno de los documentos a reconciliar *A* o *B* y la diferencia entre ellos. Se intentará mostrar por qué *reconcile* es indiferente a que se le entregue *A* o *B* como primer parámetro. Para explicar esto, se plantea el mismo escenario que el visto en la figura 14. Bajo este escenario, en las figuras que siguen se plasma la ejecución de *reconcile(A,diffTree(A,B))* y de reconcile(*B,diffTree(A,B)*). Al finalizar la ejecución de ambos métodos, debería tenerse dos árboles reconciliados con exactamente los mismos nodos a la misma profundidad.

En la figura 24 se muestran los documentos *A* y *B* a reconciliar, y el árbol de diferencias generado para ellos.

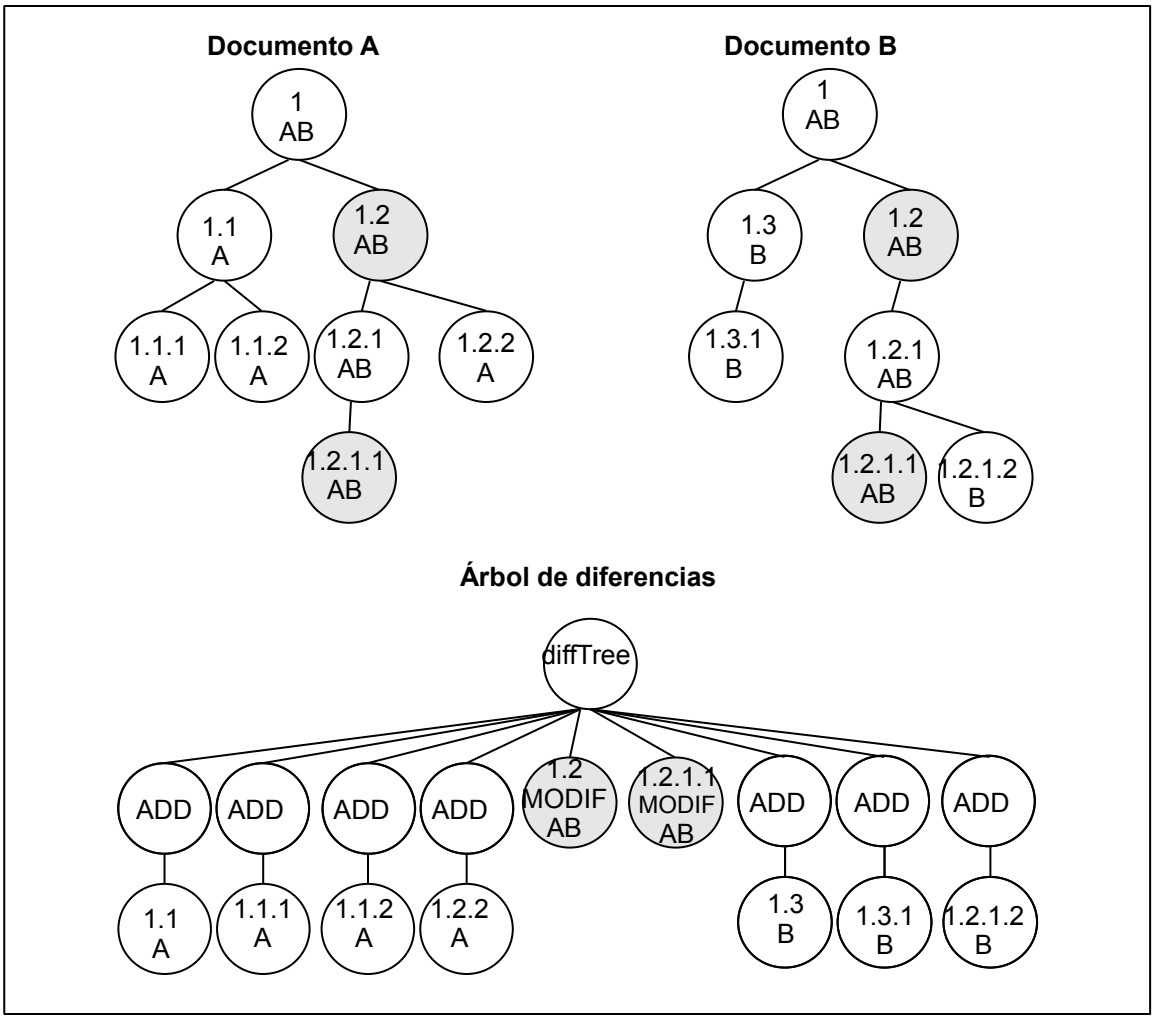


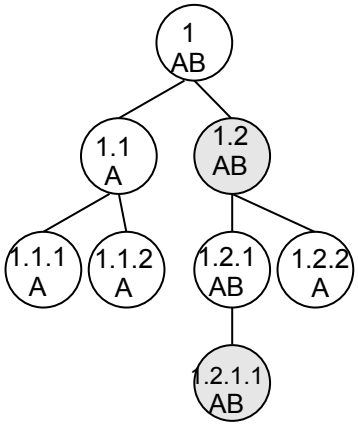
Figura 24: Documentos *A* y *B* sincronizables y su árbol de diferencias

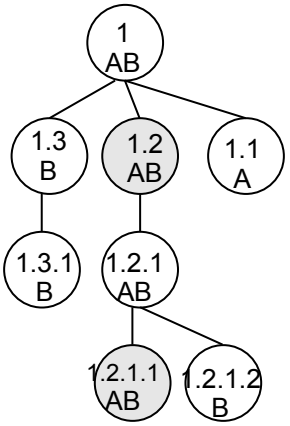
Como se ha analizado, el método reconcile hace un recorrido Top-Down sobre el árbol de diferencias, y para cada nodo que encuentre en él, realiza una acción sobre el documento reconciliado. Quien toma el lugar de documento reconciliado es el documento que se entrega como primer parámetro a *reconcile*. Se pueden encontrar en el árbol de diferencias sólo dos tipos de acciones a realizar: ADD y MODIF. Si se encuentra un nodo ADD, se debe agregar el hijo de éste nodo en el documento reconciliado, sólo en el caso que no exista en este documento. Si se encuentra un nodo MODIF, se deben sincronizar los atributos de tal nodo en el árbol reconciliado con un Resolutor apropiado para tal efecto.

Se plasma en la figura 25 la ejecución de los métodos *reconcile(A,diffTree(A,B))* y *reconcile(B,diffTree(A,B)).* Sea *recA* el árbol reconciliado que genera el método *reconcile(A,diffTree(A,B))* y sea *recB* el generado por *reconcile(B,diffTree(A,B)).* Se muestran las consecuencias que las acciones del árbol de diferencias van produciendo en *recA* y *recB*.

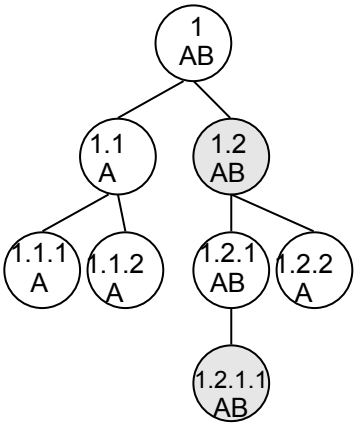
recA **recB**

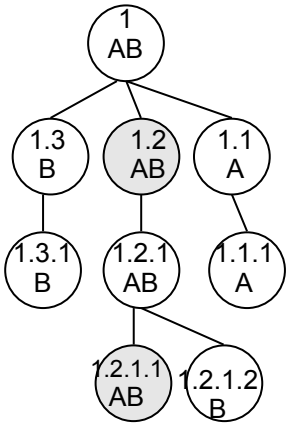
***Acción* nodo ADD con hijo 1.1:**
Se debe agregar el nodo 1.1 en *recA* y *recB*. Como ya está en *recA*, sólo se agrega en *recB*.



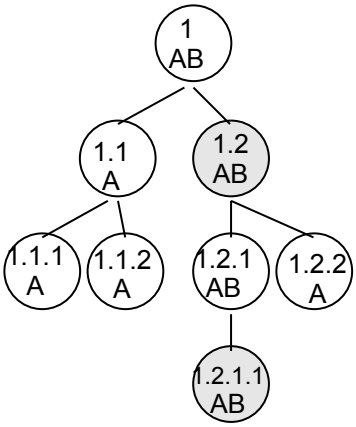


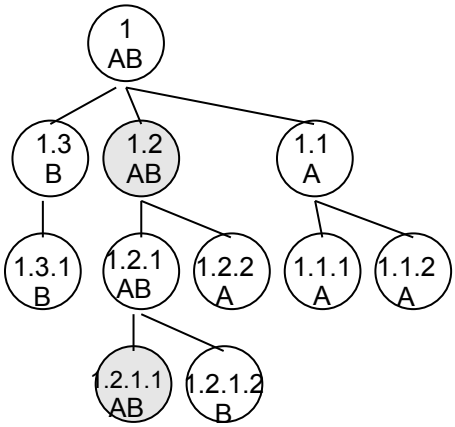
***Acción* nodo ADD con hijo 1.1.1:**
Se debe agregar el nodo 1.1.1 en *recA* y *recB*. Como ya está en *recA*, sólo se agrega en *recB*.





Hasta el nodo ADD con hijo1.2.2 se realizan las mismas acciones. Por lo tanto para cuando haya terminado la acción del nodo ADD con hijo 1.2.2 se tendrán los árboles siguientes:

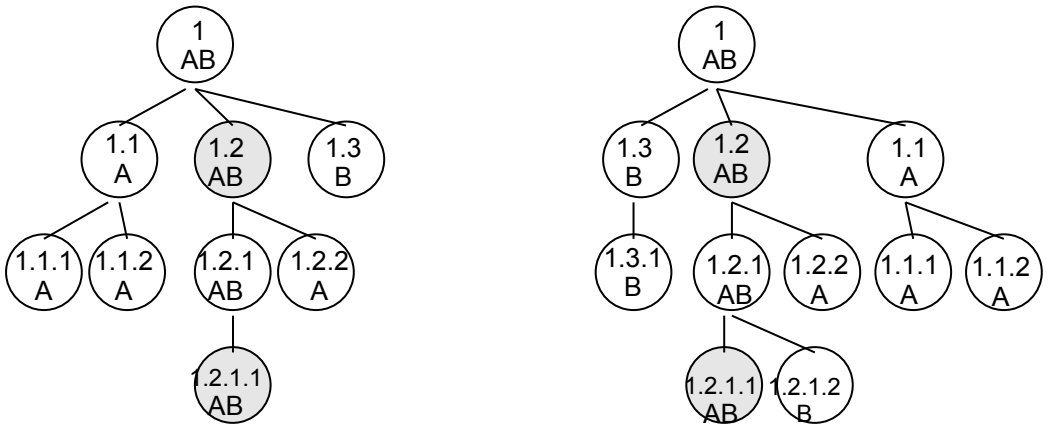




***Acción* nodo MODIF 1.2:**
Se debe sincronizar el nodo 1.1.1 en *recA* y *recB*. No se agregan ni se quitan nodos en éstos árboles, sólo se sincroniza ese nodo en ambos árboles.

***Acción* nodo MODIF 1.2.1.1:**
Se debe sincronizar el nodo 1.2.1.1 en *recA* y *recB*. No se agregan ni se quitan nodos en éstos árboles, sólo se sincroniza ese nodo en ambos árboles.

***Acción* nodo ADD con hijo 1.3:**
Se debe agregar el nodo 1.3 en *recA* y *recB*. Como ya está en *recB*, sólo se agrega en *recA*



Las acciones de los nodos ADD con hijo 1.3.1 y ADD con hijo 1.2.1.1 tienen el mismo efecto que ADD con hijo 1.3. Se deben agregar los nodos 1.3.1 y 1.2.1.1 en *recA* y *recB*. Pero como ya están en *recB*, sólo se agregan en *recA*. Los árboles resultantes son:

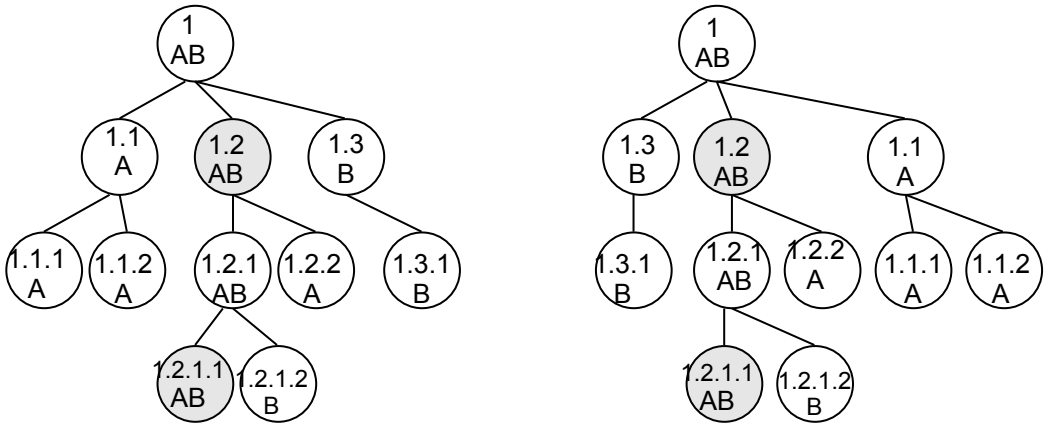


Figura 25: Comparación de reconcile(A,diff(A,B)) y reconcile(B,diff(A,B)) a través de recorrido paso a paso del árbol de diferencias

Se puede ver abajo en la figura 25, que *recA* tiene exactamente los mismos nodos que *recB* y a la misma profundidad cada uno. Sólo difieren en el orden de sus ramas, pero esto no es importante para el algoritmo de sincronización, dado que está ideado para árboles desordenados. Se muestra así, que la reconciliación de documentos sincronizables es *asociable.*

Clase Resolutor

Esta clase es la encargada de resolver discrepancias entre atributos que generan diferencias. Está pensada como una clase abstracta, para dejar la posibilidad abierta a la creación de nuevos resolutores heredando esta clase.

Cada Resolutor posee una política de sincronización que lo *caracteriza*. Podría decirse que es el "nombre" del Resolutor, pero más bien se hace referencia a las operaciones que realiza este Resolutor para la resolución de conflictos.

Al reconciliar dos nodos *sharedA* y *sharedB*, se indica qué política se debe utilizar en la sincronización, a través del atributo *_resolutor="nombrePolítica*" presente en cada uno de los nodos. Una vez reconocida la política que se usará, la clase ***Reconcile*** busca algún Resolutor que sea caracterizado por ésta política (Resolutor.policy = "nombrePolítica"). Cuando se encuentra tal Resolutor se crea una nueva instancia de éste y se invoca su método ***reconcile***.

Si ocurre que *sharedA* y *sharedB* usan distintas políticas de reconciliación, entonces se instancian los dos resolutores caracterizados por esas políticas y luego se elige uno de ellos según su ***prioridad***. La prioridad de los resolutores es un número entero que simplemente indica qué Resolutor debe ser usado por sobre otro en la sincronización. Se utiliza el Resolutor con más alta prioridad.

El método *reconcile* es el más importante de la clase Resolutor, pues es este método el que define las operaciones a realizar para resolver los conflictos dada su política característica. Este método es invocado con 4 parámetros: *valorA, valorB, criterioA y criterioB*. Los parámetros *valorA* y valorB son los valores que toma el atributo que generó diferencias en *sharedA y sharedB*, *criterioA* y *criterioB* son los valores que toma la política de resolución escogida en *sharedA* y *sharedB*. Así, el método reconcile realiza ciertas operaciones entre *valorA* y *valorB* dado *criterioA* y *criterioB*, para conseguir el valor reconciliado y devolverlo. Para mayor claridad, en la figura 26 se muestra una resolución de conflictos.

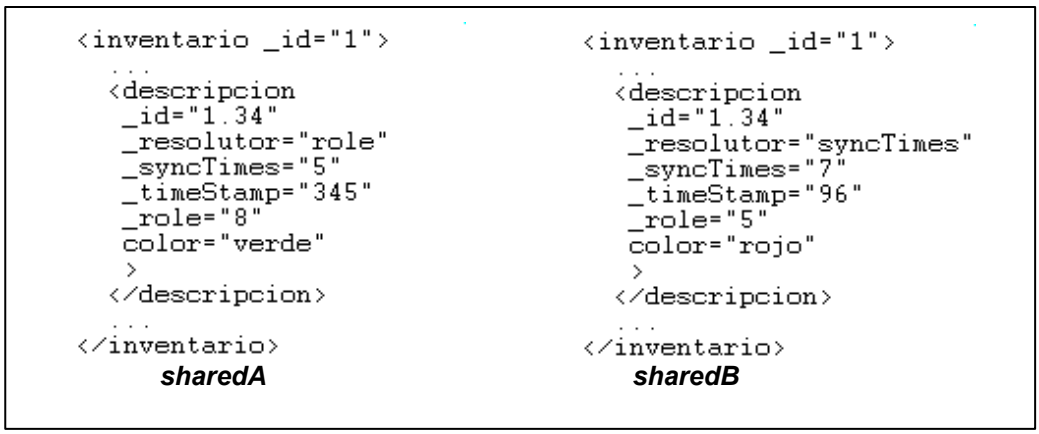


Figura 26: Uso de Resolutor para resolución de diferencias

En el nodo *sharedA* se invoca la política *_resolutor="role"*, mientras que en el nodo B *_resolutor="syncTimes".* Como ambas políticas son distintas se procede a instanciar dos resolutores: Role, caracterizado por "role" y SyncTimes por "syncTimes". Luego se escoge uno de estos resolutores dada su prioridad. En este caso, la prioridad de Role es mayor a la de SyncTimes, por lo tanto se escoge Role. Así, los criterios para la sincronización serán los valores que tome *_role* en cada nodo. En *sharedA* *_role* toma el valor "8", por lo tanto *criterioA* será "8"; de la misma forma, *criterioB* será "5". Como el atributo que está generando diferencias es *color* entonces *valorA="verde"* y *valorB="rojo".* Por lo tanto se invocará Role.reconcile("verde","rojo","8","5") y será éste método el que resuelva la diferencia.

3.2.2. Modelo de sincronización de dos documentos XML

Para la sincronización de dos documentos XML *A* y *B*, se genera el árbol de diferencias entre ellos con la clase DiffTree, a través del método compute. Una vez generada la diferencia, se utiliza el método reconcile de la clase Reconcile para crear el documento reconciliado utilizando la diferencia y alguno de los dos documentos XML. Reconcile recorre el árbol de diferencias y ejecuta las operaciones que éste indica sobre el documento *A* o *B* para transformarlo en el documento reconciliado. Dadas las propiedades de *conmutación* del árbol de diferencias y *asociación* de la reconciliación, puede utilizarse *A* o *B* para obtener el documento reconciliado.

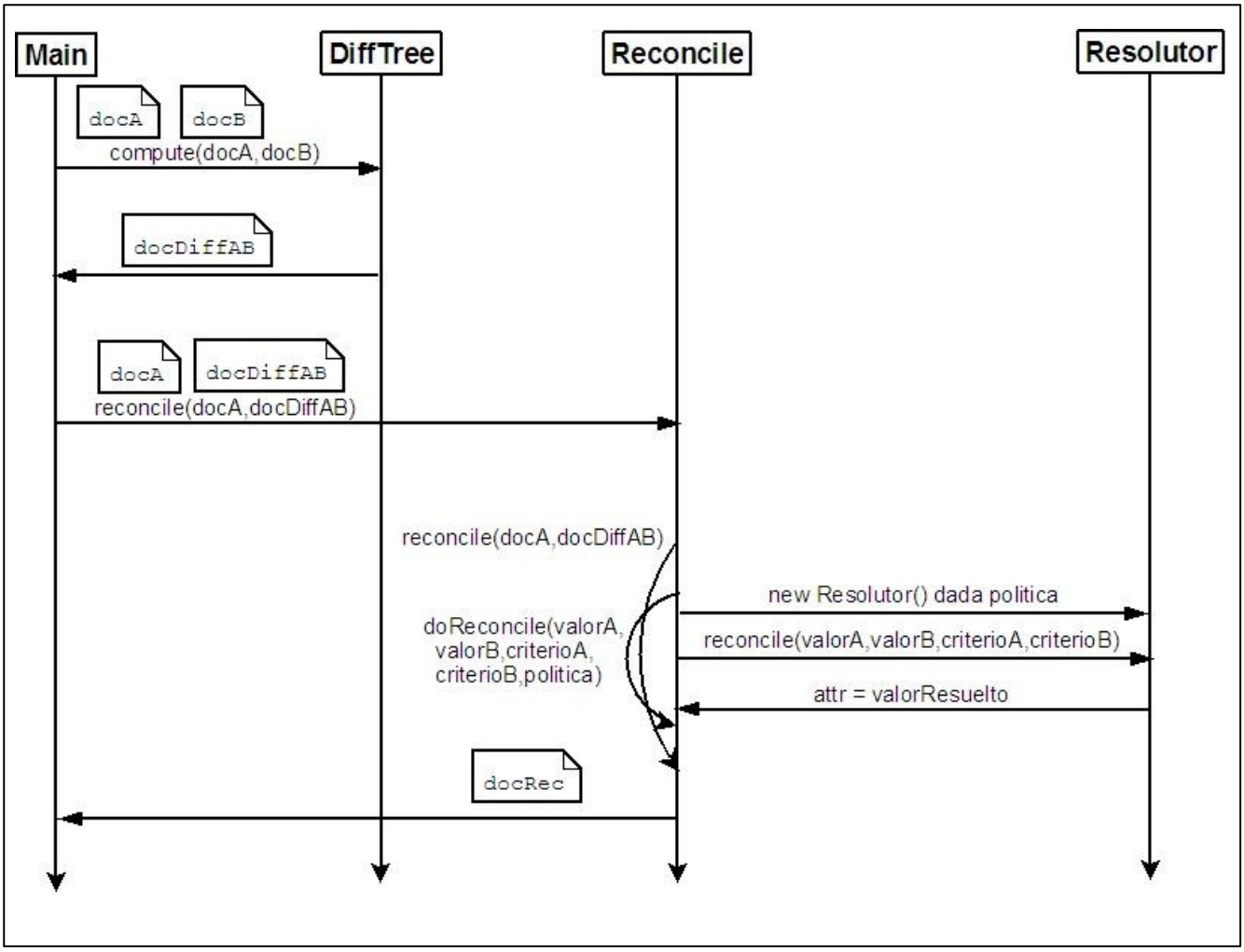


Figura 27: Diagrama de secuencias. Sincronización de dos documentos XML

En la figura 27 se presenta un diagrama de secuencias que muestra el proceso de reconciliación entre dos documentos XML sincronizables.

3.2.3. Sincronización extendida para N documentos

La sincronización de N documentos XML sincronizables se lleva a cabo reconciliando parejas de documentos. Se inicia con una pareja de documentos, y el resultado de esa reconciliación se reconcilia a su vez con otro de los N documentos. Antes de mostrar en detalle el proceso de sincronización de N documentos XML se mostrará una propiedad interesante que hace posible este proceso.

Transitividad de la reconciliación

Se dice que la reconciliación entre N documentos XML sincronizables es *transitiva*, puesto que para todo conjunto de 3 documentos sincronizables *A, B y C* se cumple que el documento reconciliado por:

i. reconcile (A , diff (A , reconcile (C , diff(C,B)))

tiene exactamente los mismos nodos y a la misma profundidad que el documento reconciliado por:

ii. reconcile (B , diff (B , reconcile (A , diff(A,C)))

y exactamente los mismos nodos y a la misma profundidad que el documento reconciliado por:

iii. reconcile (C , diff (C , reconcile (B , diff(A,B)))

O, dicho de otra manera, no importando de qué forma se formen las parejas para reconciliar los 3 documentos, el documento reconciliado resultante entre los 3 documentos es el mismo; salvo el orden relativo de sus ramas. Esta propiedad es intuitivamente cierta pues cada reconciliación, como ya se vio en la figura 25, agrega todos los nodos no compartidos de cada uno de los documentos al árbol reconciliado. En el árbol reconciliado, por otro lado, ya están los nodos compartidos por los documentos y estos nodos no se eliminan. Por lo tanto, no importando la forma en que se escojan las parejas para la reconciliación, siempre se tendrán en el documento reconciliado, ***todos los nodos no compartidos de todos los documentos*** que se reconciliaron, ***más los nodos compartidos*** entre ellos, y lo único que cambiará será el orden relativo entre sus ramas. Es decir, para una reconciliación entre los documentos *A, B y C;* en el documento reconciliado siempre se tendrán todos los nodos no compartidos de *A, B y C* y los nodos compartidos entre: *A - B, B - C, A - C y A - B - C* no importando las parejas que se formen para la sincronización.

Para mostrar la transitividad de la reconciliación, a continuación se muestra gráficamente la ejecución de los métodos de reconciliación dados por las expresiones ***i, ii*** *y* ***iii.*** El escenario que se ve en la figura 28, presenta los documentos *A, B y C* sincronizables. Se puede notar que cada subárbol de los documentos *A, B y C* tienen letras. Estas letras indican si los nodos que están en ese subárbol son o no compartidos por los documentos. Por ejemplo, si un subárbol tiene la letra “A”, esto indica que sus nodos no son compartidos, y sólo pertenecen al documento *A.* Pero si un subárbol tiene las letras “*BC*” esto indicaría que

sus nodos son compartidos por los documentos *B y C.* Y, demás está decir que si un subárbol tiene las letras “ABC”, esto indica que sus nodos son compartidos por los documentos *A, B y* C.

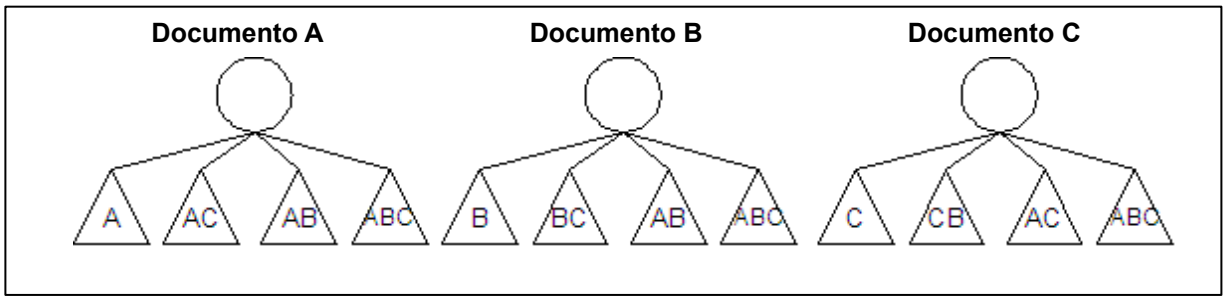


Figura 28: Documentos *A, B y C* sincronizables

En la figura 29 se muestra la ejecución de la reconciliación dada por la expresión ***i.***

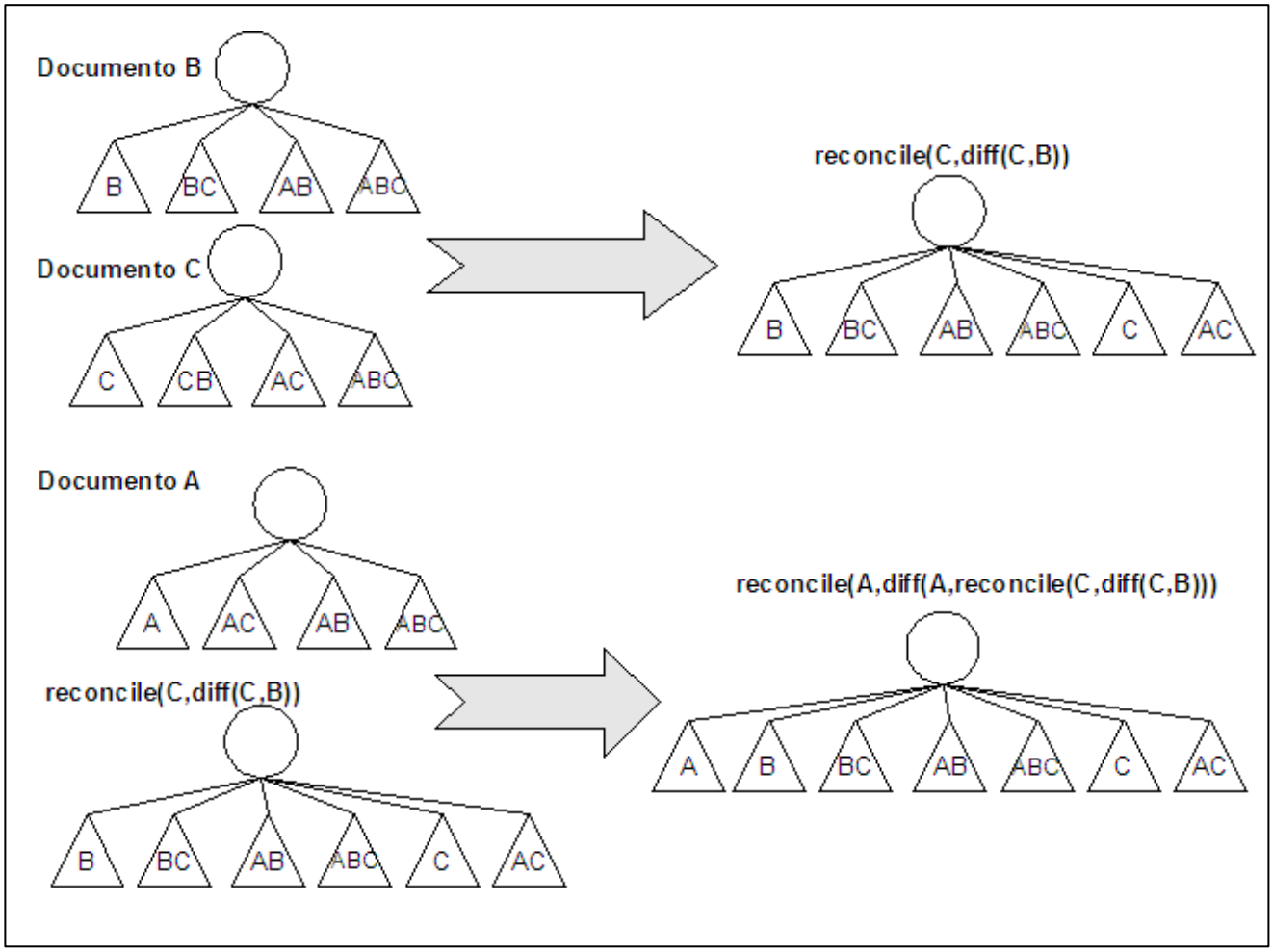


Figura 29: Ejecución de la reconciliación dada por reconcile (A , diff (A , reconcile (C , diff(C,B)))

En la figura 30 se muestra la ejecución de la reconciliación dada por la expresión ***ii***.

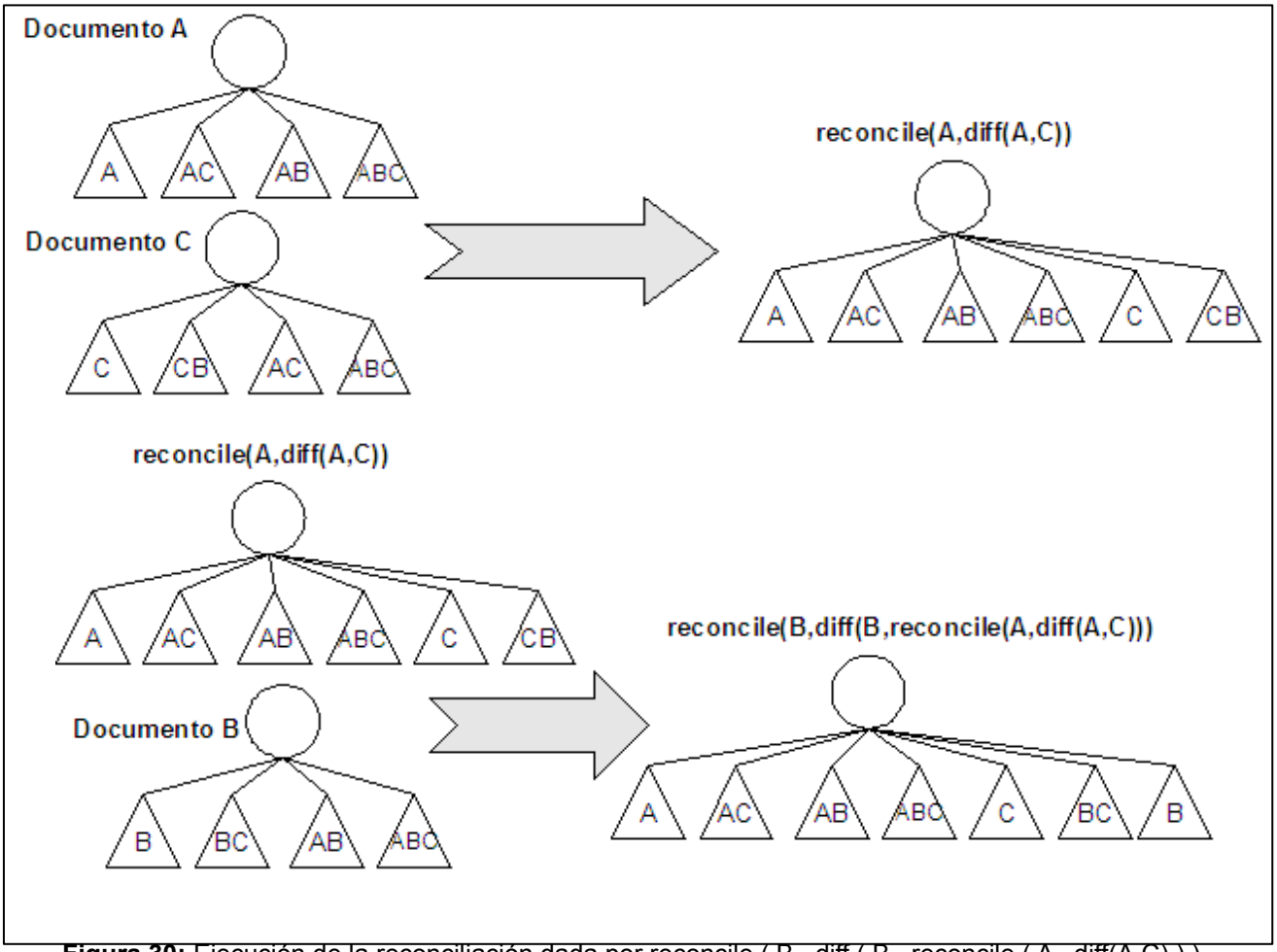


Figura 30: Ejecución de la reconciliación dada por reconcile (B , diff (B , reconcile (A , diff(A,C)))

En la figura 31 se muestra la ejecución de la reconciliación dada por la expresión ***iii***.

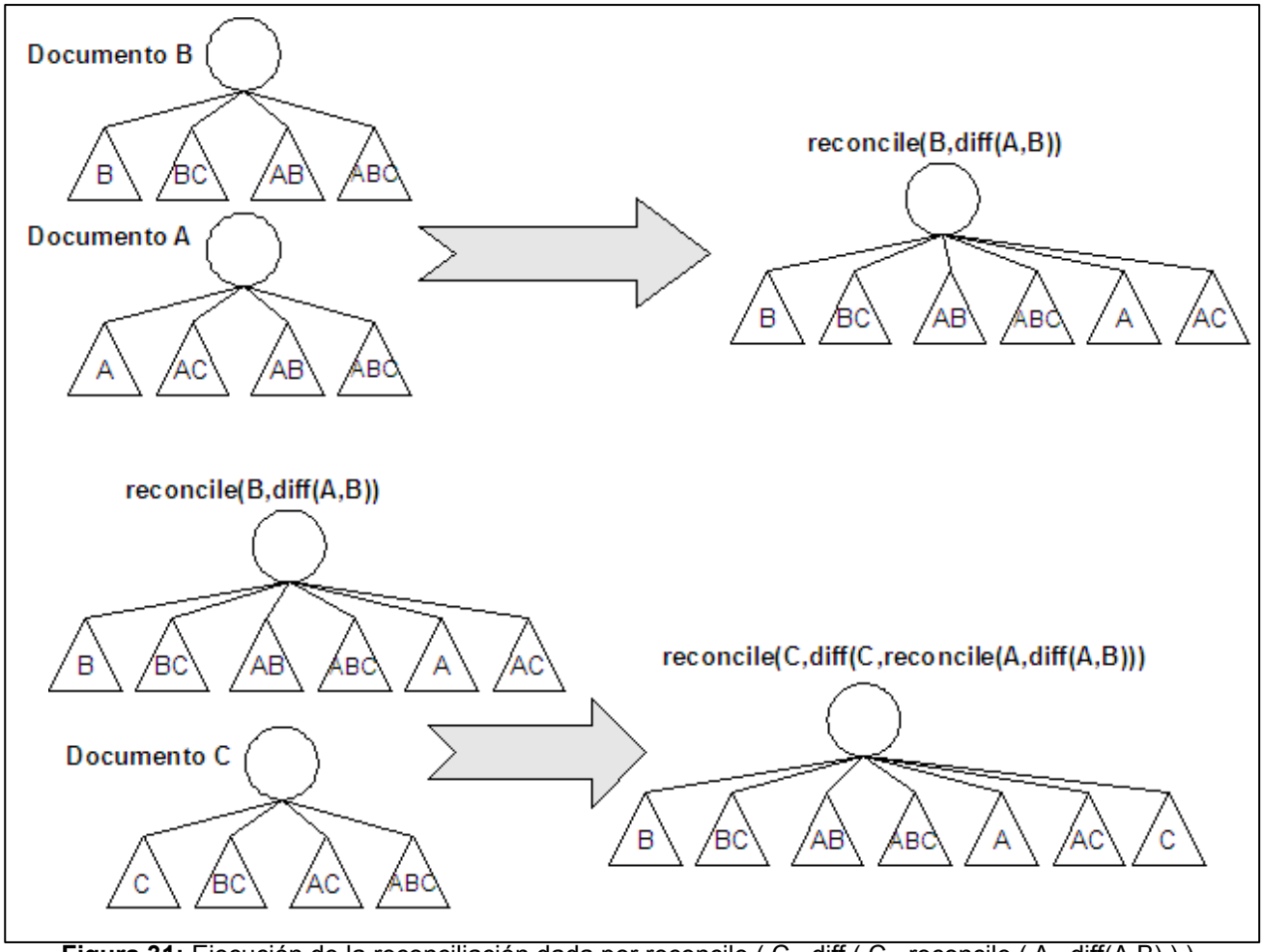


Figura 31: Ejecución de la reconciliación dada por reconcile (C , diff (C , reconcile (A , diff(A,B)))

Se puede notar que las operaciones de reconciliación dadas por las expresiones ***i***, ***ii*** y ***iii*** dan como resultado árboles que tienen subárboles iguales, salvo el orden de sus ramas. Cabe destacar que el contenido de los nodos reconciliados también será el mismo no importando el orden de las parejas en que se reconcilien los documentos, principalmente porque las operaciones matemáticas y de orden utilizadas por los resolutores en sus métodos *reconcile* definidos son transitivas. Sin embargo, se debe tener muy en cuenta que esta propiedad se cumple sólo mientras se usen ***operaciones transitivas*** para la resolución de conflictos en todos los resolutores. Vale decir, si un resolutor utiliza alguna operación matemática, de orden o de cualquier otra índole *no transitiva* para resolver los conflictos de los atributos en su método *reconcile*, la propiedad de transitividad de la reconciliación podría perderse.

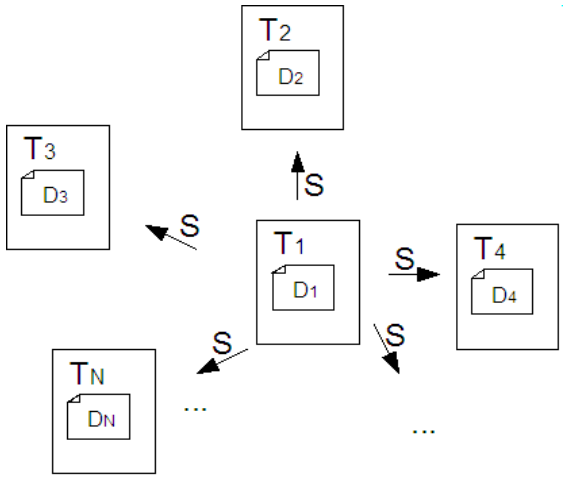
Se muestra así que la reconciliación es *transitiva*.

Protocolo de reconciliación para N documentos XML sincronizables

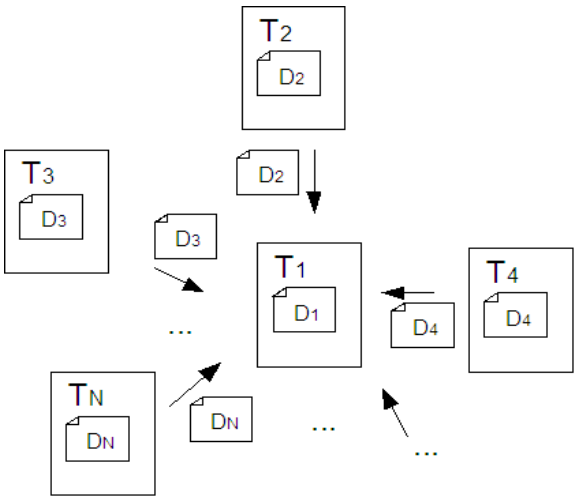
Aprovechando las propiedades de transitividad y asociación de la reconciliación, y de conmutación del árbol de diferencias se puede definir un protocolo de sincronización entre N documentos XML sincronizables. A continuación se define el protocolo de sincronización.

Sea *R* una red de trabajo definida entre N trabajadores con dispositivos móviles que comparten un documento XML sincronizable. Si un trabajador perteneciente a la red *R* necesita sincronizar la copia local del documento XML que comparte, y en ese momento tiene acceso a *R*, envía una señal *S* de petición de reconciliación a todos los trabajadores que estén accesibles en *R* al momento. Sea T_1 el trabajador que generó *S*, D_1 el documento XML de T_1, y $D_2...D_N$ los documentos de los demás trabajadores accesibles en ese momento a través de la red *R*. Cada trabajador que reciba *S*, enviará su copia local del documento compartido a T_1. Así, el dispositivo móvil de T_1 tendrá ahora acceso **local** a los documentos $D_1...D_N$. Paso siguiente, T_1 comienza a reconciliar $D_1...D_N$ por parejas. Primero, realiza la reconciliación de D_1 - D_2 lo que produce el documento reconciliado D_1D_2, luego D_1D_2 se reconcilia con D_3 y se obtiene así $D_1D_2D_3$ y así sucesivamente, hasta llegar a reconciliar los N documentos, obteniendo el documento reconciliado $D_1D_2D_3...D_N$.

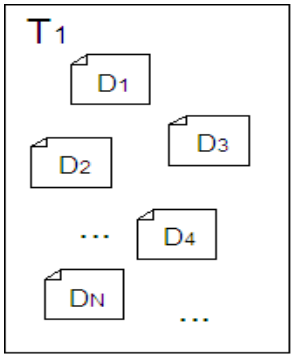
Dada la transitividad de la reconciliación, no importa qué parejas de documentos se reconcilien, ni en qué orden se realice la sincronización; el resultado final siempre será el mismo, siempre y cuando los N documentos formen parte de la reconciliación al menos 1 vez. Se esquematiza el protocolo planteado en la figura 32.



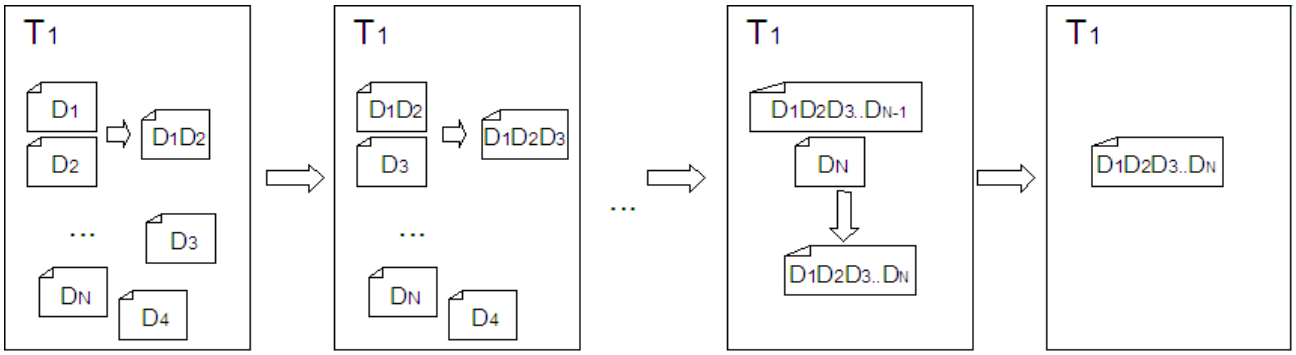
El trabajador T_1 envía la señal de petición de reconciliación *S* a todos los dispositivos accesibles por la red *R*



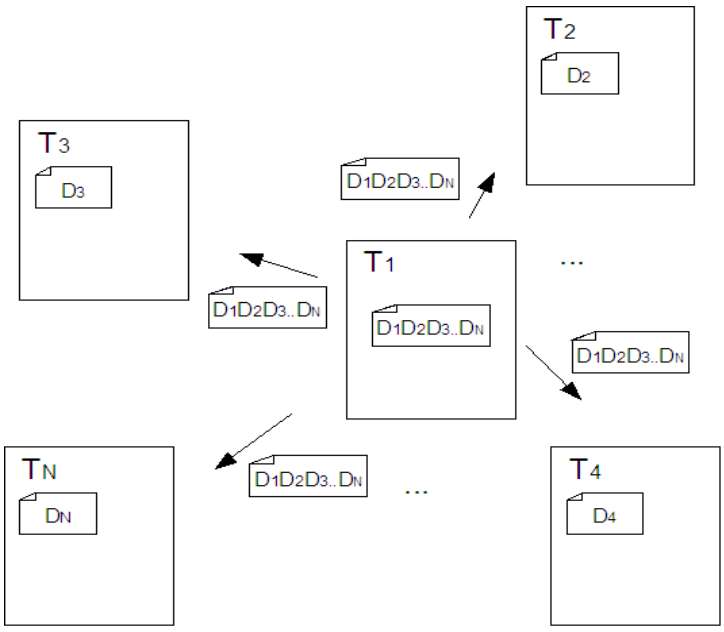
Todos los dispositivos que reciben la señal de *T1,* le envían las copias locales de sus documentos



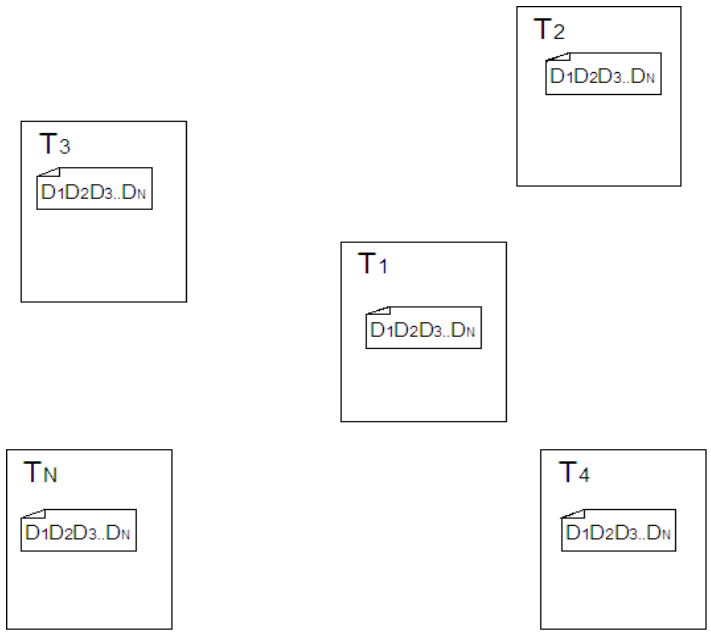
T1 tiene acceso local a los documentos de todos los dispositivos que respondieron a la señal



T1 realiza consecutivas reconciliaciones por parejas de documentos, hasta conseguir el documento reconciliado de las N copias del documento compartido de los N dispositivos que respondieron a la señal



T1 envía una copia del documento reconciliado a todos los dispositivos que respondieron a la señal



Finalmente, todos los dispositivos que reciben exitósamente la copia del documento reconciliado enviada por T1, sobrescriben su copia local del documento compartido con este documento reconciliado

Figura 32: Esquema del protocolo de reconciliación de N documentos sincronizables

Puede pensarse que el protocolo de sincronización planteado en este trabajo de título

está centralizado, o sigue el modelo de cliente-servidor. Sin embargo, esto no es así puesto que cada dispositivo móvil tiene la capacidad de reconciliar desde dos a N documentos XML. Así, cualquier trabajador que se encuentre conectado a la red y que necesite sincronizar su copia local del documento XML compartido, lo puede hacer en cualquier instante iniciando el proceso de sincronización de la manera que ya se ha analizado. Dado que todos los dispositivos móviles están facultados para iniciar el protocolo de sincronización planteado, el procedimiento de reconciliación no está centralizado. Por lo tanto, no es imprescindible para la sincronización que *todos* los dispositivos móviles de la red *R* se encuentren disponibles en el momento en que alguno de ellos inicia el proceso de reconciliación. Solamente se sincronizarán los dispositivos que estén disponibles. En vista de esta situación, se considera importante el planteamiento de un control de versiones de los documentos compartidos. Este punto no se toca en la presente memoria, pero puede subsanarse con la incorporación adecuada de Resolutores y atributos protegidos.

4. Solución Implementada

En este capítulo se analizarán detalles del módulo de sincronización automática de documentos XML desarrollado para dispositivos móviles PDA's y notebooks. Se intentará explicar el código desarrollado a través de pseudocódigo y diagramas de clases.

4.1. Pseudocódigo

A continuación se presenta el pseudocódigo de las dos principales clases de la solución implementada: DiffTree y Reconcile. El objetivo de presentar el pseudocódigo es que se logre una mejor comprensión de la manera como se lleva a cabo el algoritmo de reconciliación de dos documentos XML.

Clase DiffTree

```
DiffTree(treeA,treeB){

  diffTree = null; // es sobre diffTree que se armará el árbol de diferencias.

  void compute(treeA,treeB) {
    processTreeA(treeA,treeB);
    processTreeB(treeB);
  }

  void processTreeA(treeA,treeB){
    foreach (nodeA in treeA){ // se recorre en amplitud el árbol A y se buscan diferencias con el árbol B.
      nodeB = findNode(nodeA,treeB); // ¿está nodeA en el árbol B?
      if (nodeB != null){  // nodeB es el nodo con el mismo _id de nodeA en el árbol B
        nodeB.setAttribute("_visited","true"); //se marca nodeB como visitado
        bool equal = areNodesEqual(nodeA,nodeB); // hay modificaciones?
        if (!equal){
          modifNode = creaModif(nodeA,nodeB); //se crea un nodo MODIF
          diffTree.appendChild(modifNode); // se agrega el nodo MODIF al árbol de diferencias
        }
        else{ // nodeA está en el árbol B, pero no existen diferencias entre nodeA y nodeB
          //no se realiza acción sobre diffTree
        }
      }
      else{ // nodeA no está en el árbolB.
        addNode = creaAdd(nodeA); //se crea un nodo ADD
        diffTree.appendChild(addNode); // se agrega el nodo ADD al árbol de diferencias
      }
    } // end recorre A en amplitud
  }
```

```
  void processTreeB(treeB){
    foreach (nodeB in treeB){ // se recorre el árbol B en amplitud
      if(nodeB.isTagged("_visited","true"){ // si el nodoB fue visitado por processTreeA no se realiza acción
      }
      else{ // si no fue visitado
        addNode = creaAdd(nodeB); //se crea un nodo ADD
        diffTree.appendChild(addNode); // se agrega el nodo ADD al árbol de diferencias
      }
    }
  }//end processTreeB

  node creaModif(nodeA,nodeB){
    modif = new Node("MODIF");
    foreach (attrA in nodeA.Attributes){ // se recorren los atributos de nodeA
      if (!attrA.Name.startsWith("_")){ // atributos no protegidos
        if ((attrB = nodeB.Attributes.GetAtribute(attrA.Name))== null){ // attrA está sólo en nodeA y no en nodeB
          modif.Attributes.add(attrA); // se agrega atributo no compartido
        }
        else{  // el atributo es compartido por nodeA y nodeB
          if ( attrA.Value.Equals(attrB.Value) ) { // si attrA y attrB son iguales, no generan diferencias
            modif.Attributes.add(attrA); // se agrega attrA
          }
          else{ // attrA y attrB generan diferencias
            modifSon = new Node(attrA.Name); // se crea un nodo hijo de MODIF con el nombre del atributo
            modifSon.Attributes.add(new Attribute("valueA",attrA.Value); //valueA y valueB tienen los valores del
            modifSon.Attributes.add(new Attribute("valueB",attrB.Value); // atributo en sharedA y sharedB
            modif.appendChild(modifSon);
          }
        }
        else{ //atributos protegidos
          // para cada atributo protegido se agregan los valores que tienen en sharedA y sharedB.
          modif.Attributes.add(new Attribute(attrA.Name+"A",attrA.Value));   // ej: se agrega _roleA = _role + "A";
          attrB = nodeB.Attributes.GetAtribute(attrA.Name);
          modif.Attributes.add(new Attribute(attrB.Name+"B",attrB.Value));
        }
      }
      foreach (attrB in nodeB.Attributes){
        if ((attrA = nodeA.Attributes.GetAtribute(attrB.Name))== null){ // attrB está sólo en nodeB y no en nodeA
          modif.Attributes.add(attrB); // se agrega atributo no compartido
        }
      }
    }
  }

  node creaAdd(node){
    add = new Node("ADD");
    String idPadre = (node.Attributes.GetAttribute("_id")).substring(0,LastIndexOf(".")); //identificador del padre
    add.Attributes.add(new Attribute("_idPadre",idPadre);
    add.appendChild(node);
  }
}
```

Clase Reconcile

Reconcile(documento,diferencias){

```
  Resolutor resolutors;
```

```
reconcile(documento,diferencias){
  foreach(diffNode in diferencias){ // se realiza recorrido Top-Down sobre el árbol de diferencias.
    if(diffNode.Name.Equals("MODIF")){ // reconciliación de nodo MODIF
      //primero se busca el nodo identificado con _id de MODIF en documento
      nodeRec = findNode(documento,diffNode);
      foreach( attr in diffNode.Attributes ) {
        if(!attr.NameStartsWith("_")){
          //paso 1: se agregan al nodo reconciliado los atributos no protegidos de MODIF
          nodeRec.Attributes.add(attr);
        }
        else {
          // paso 2: se sincronizan atributos protegidos
          nodeRec.Attribute.add(reconcileProtAttr(attr));
        }
      }
      // paso 3: se reconcilian los atributos que generan diferencias, los hijos de MODIF
      foreach (modifSon in diffNode.Children) {
        //el resolutor se busca en nodeRec, pues ya se han sincronizado los atributos protegidos en este nodo
        resolutor = nodeRec.Attributes.GetAttribute("_resolutor").Value;
        criterioA = modifSon.Attributes.GetAttribute(resolutor+"A").Value;
        criterioB = modifSon.Attributes.GetAttribute(resolutor+"B").Value;
        valueA = (modifSon.Attributes.GetAttribute("valueA")).Value;
        valueB = (modifSon.Attributes.GetAttribute("valueB")).Value;
        // se obtiene el valor reconciliado para el atributo que generó diferencias
        valorFin = doReconcile(valueA,valueB,criterioA,criterioB,resolutor);
        // y se actualiza el nodo reconciliado con el nuevo valor resultante de la sincronización
        nodeRec.Attributes.Set(modif.Name,valorFin);
      }
    }
    else if(  diffNode.Name.Equals("ADD")){  //reconciliación de nodo ADD
      idPadre = diffNode.Attributes.GetAttribute("_idPadre").Value;
      padre = findNode(documento,idPadre);
      hijo = diffNode.FirstChild;
      if( null == findNode(documento,hijo))
        padre.appendChild(hijo);   // se inserta el nodo ADD sólo si estaba ya en el documento reconciliado
    }
  }
}

doReconcile(valorA,valorB,criterioA,criterioB,resolutorP){
  bool found = false;
  Resolutor resol = null;
  foreach( res in resolutors ){
    if ( res.containsPolicy(resolutorP) ) { // se busca un resolutor que contenga la política requerida
      found = true;
      resol = res;
    }
  }
  if (found) {
    // se utiliza el resolutor encontrado para resolver las diferencias
    return resol.reconcile( valorA,valorB,criterioA,criterioB,resolutorP);
  }
  else{
    print ("no se encuentra la política en ningún resolutor");
  }
}
}
```

4.2. Diagrama de clases

A continuación, en la figura 33 se presenta el diagrama de clases del módulo para sincronización automática de dos documentos XML desarrollado.

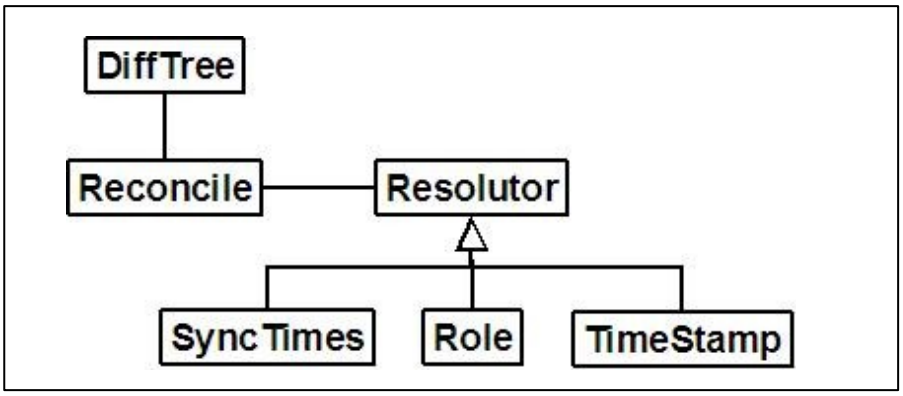


Figura 33: Diagrama de clases módulo de sincronización automática de documentos XML

Clase DiffTree

Se presenta en la figura 34 un esquema de la clase DiffTree con sus métodos y variables de instancia.

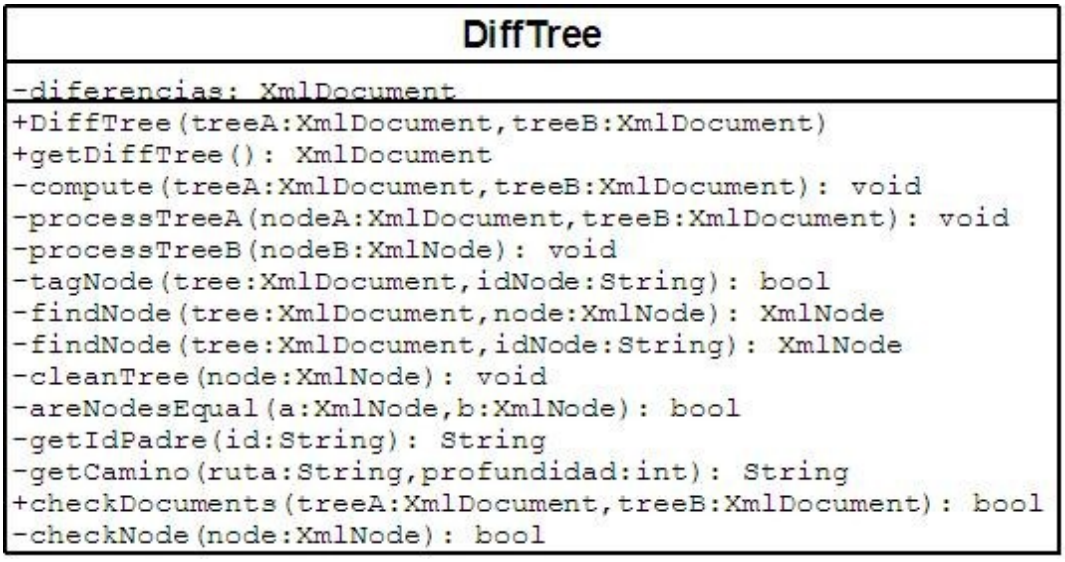


Figura 34: Clase DiffTree

Métodos relevantes de la clase DiffTree

Los métodos más relevantes de la clase DiffTree se presentan en la tabla 6.

Método	Descripción
void compute(treeA, treeB)	Arma el árbol de diferencias entre *treeA* y *treeB*
Node findNode(tree,node)	Busca *node* dentro del árbol *tree.* Si lo encuentra lo devuelve, si no lo encuentra devuelve null
bool areNodesEqual(nodeA,nodeB)	Compara dos nodos y retorna true si son iguales, o false si son distintos

Tabla 6: Métodos de la clase DiffTree

Clase Reconcile

Se presenta en la figura 35 un esquema de la clase Reconcile con sus métodos y variables de instancia.

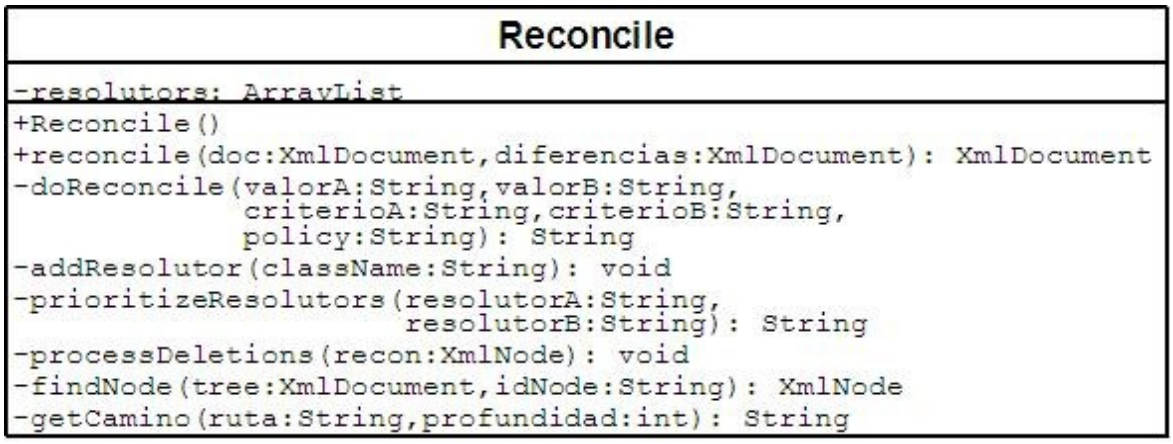


Figura 35: Clase Reconcile

Métodos relevantes de la clase Reconcile

Los métodos más relevantes de la clase Reconcile se presentan en la tabla 7.

Método	Descripción
document reconcile(doc, diferencias)	Aplica el proceso de reconciliación entre *doc* y *diferencias* y entrega el documento reconciliado
void addResolutor(className)	Agrega el resolutor de nombre *className* a la lista de resolutores que se podrán usar para la reconciliación
string prioritizeResolutors(resA,resB)	Entrega el resolutor de mayor prioridad entre *resA y resB*

Tabla 7: Métodos de la clase Reconcile

Clase Resolutor

Se presenta en la figura 36 un esquema de la clase Resolutor con sus métodos y variables de instancia.

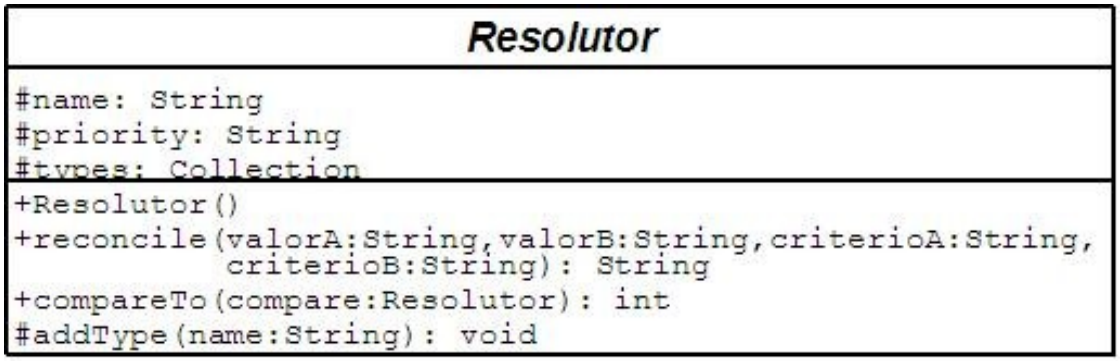


Figura 36: Clase Resolutor

Métodos relevantes de la clase Resolutor

Los métodos más relevantes de la clase Resolutor se presentan en la tabla 8.

Método	Descripción
string reconcile(valorA,valorB,criterioA,criterioB)	Resuelve las diferencias entre *valorA* y *valorB* de acuerdo a los criterios de reconciliación *criterioA y criterioB.* Devuele un string con el valor reconciliado
int compareTo(compare)	Entrega -1 si el resolutor this tiene menor prioridad que el resolutor *compare,* 0 si tiene la misma y 1 si tiene una prioridad mayor

Tabla 8: Métodos de la clase Resolutor

5. Resultados Obtenidos y Esperados

Se realizaron variadas pruebas al módulo desarrollado. Se muestran algunas de ellas y los resultados obtenidos en el presente capítulo. También se realizaron análisis de tiempos de sincronización para distintas configuraciones de documentos XML. El objetivo de estos análisis era tener una idea general de cómo se comportaba el módulo desarrollado dependiendo del número de nodos de los documentos XML a sincronizar. También se quería analizar cómo se comportaba el módulo de sincronización para documentos con árboles de distinta profundidad y grado. Se quería establecer alguna relación entre estos factores y el tiempo que se demoraría el módulo en sincronizar tales documentos. En las siguientes figuras, se muestran algunos documentos que se utilizaron para probar la aplicación desarrollada, y los resultados de tales pruebas.

Las pruebas realizadas se dividieron en:

- Pruebas para sincronización de dos documentos XML
- Pruebas para sincronización de N documentos XML
- Análisis de eficiencia

A continuación se muestran detalladamente algunas de las pruebas realizadas.

5.1. Pruebas para sincronización de dos documentos XML

Las pruebas para la sincronización de dos documentos XML se dividieron en:

5.1.1. Documentos sólo con diferencias de tipo modificación de nodos

Como la reconciliación de documentos con diferencias del tipo modificación de nodos utiliza resolutores, se debieron realizar distintas pruebas para cada uno de los resolutores desarrollados. Se presentan en las siguientes figuras las pruebas más representativas.

En la figura 37 se pueden ver los documentos *A* y B utilizados para las pruebas de modificación de nodos. Los nodos de estos documentos tienen varias características que permiten probar varios aspectos de la reconciliación y los resolutores.

```
<root
 valor="rootA" _id="1" _resolutor="role"
 _timeStamp="12" _role="1" _syncTimes="5"
 _visible="true">

 <node
  shared="s" generaDiff="A" _id="1.1"
  _resolutor="timeStamp" _timeStamp="10"
  _role="1" _syncTimes="5" _visible="true">

  <nodeSon
  valor="A" _id="1.1.1" _resolutor="timeStamp"
  _timeStamp="10" _role="1" _syncTimes="12"
  _visible="true"/>

 </node>

 <node
  _id="1.2" _resolutor="role" _timeStamp="14"
  _role="2" _syncTimes="25" _visible="true"/>

</root>
```

Documento A

```
<root
 valor="rootB" _id="1" _resolutor="timeStamp"
 _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node
  shared="s" generaDiff="B" _id="1.1"
  _resolutor="timeStamp" _timeStamp="15"
  _role="2" _syncTimes="4" _visible="true">

  <nodeSon
   valor="B" _id="1.1.1" _resolutor="syncTimes"
   _timeStamp="11" _role="2" _syncTimes="9"
   _visible="true"/>

 </node>

 <node
  _id="1.2" _resolutor="role" _timeStamp="1"
  _role="2" _syncTimes="25" _visible="true"/>

</root>
```

Documento B

Figura 37: Documentos de prueba *A* y *B*

Se puede ver que los documentos *A* y *B* de la figura 37 comparten sus 4 nodos. Se destacan muchos casos interesantes de pruebas:

- Para el **nodo *_id="1"*** por ejemplo, puede notarse que en el documento *A* se tiene *_resolutor="role",* mientras que en *B* *_resolutor="timeStamp".* Esto es una buena

manera de probar si la reconciliación con resolutores distintos funciona. Como el resolutor Role tiene mayor prioridad que TimeStamp, el resolutor en el nodo reconciliado *_id="1"* debe ser *_resolutor="role".* Además, la reconciliación debe llevarse a cabo bajo la política dada por este resolutor. Se ve que los parámetros de reconciliación para *_role* son: en el documento *A _role="1"* y en *B _role="3".* Como *_role* es mayor en el documento *B,* y la política de Role dice que el mayor role tiene prioridad, entonces en el documento reconciliado, el atributo que genera diferencias *valor,* debe heredar el contenido que tiene en el documento *B valor="rootB".* Por otro lado, los atributos _timeStamp, _role y _syncTimes deben ser reconciliados de acuerdo a la tabla 5.

- Para el nodo *_id="1.1"* se tiene un atributo compartido que no genera diferencias *shared* y uno que genera diferencias *generaDiff; shared* debe ser heredado al nodo reconciliado y *generaDiff* reconciliado según la política *_resolutor="timeStamp".* Se prueba de esta manera entonces el resolutor TimeStamp y el tratamiento sobre nodos compartidos que no generan diferencias.

- Con el nodo *_id="1.1.1"* se pretende probar la reconciliación de nodos modificados en niveles inferiores a los de profundidad 1. Dado que la política de reconciliación *resolutor="_syncTimes"* tiene mayor prioridad que *_resolutor="timeStamp"* además se consigue probar una reconciliación con resolutor SyncTimes.

El resultado de la reconciliación entre los documentos *A* y *B* se muestra en la figura 38.

```
<root
 _id="1" _timeStamp="12" _role="3" _syncTimes="6"
 _resolutor="role" _visible="true" valor="rootB">

  <node
   shared="s" _id="1.1" _timeStamp="15" _role="2"
   _syncTimes="6" _resolutor="timeStamp" _visible="true"
   generaDiff="B">

    <nodeSon
    _id="1.1.1" _timeStamp="11" _role="2" _syncTimes="13"
    _resolutor="syncTimes" _visible="true" valor="A" />

  </node>

  <node
  _id="1.2" _timeStamp="14" _role="2" _syncTimes="26"
  _resolutor="role" _visible="true" />

</root>
```

Figura 38: Documento reconciliado entre *A* y *B*

Los resultados que se obtienen de las pruebas planteadas son los siguientes:

- Para el nodo *_id="1"* se ve que en el documento reconciliado se tiene

_resolutor="role". Además se tiene el atributo *valor="rootB".* Por lo tanto, se priorizó correctamente el resolutor a utilizar, en este caso Role y la reconciliación realizada por este resolutor fue correctamente llevada a cabo. Por otro lado, se tiene que *_timeStamp="12"* es efectivamente el mayor timeStamp entre los nodos *_id="1"* de los documentos *A* y *B*, *_role="3"* también fue correctamente heredado el mayor role, *_syncTimes="6"* corresponde al mayor syncTimes + 1. Así, los atributos protegidos también fueron correctamente reconciliados de acuerdo a la tabla 5.

- Para el nodo *_id="1.1"* se agregó *shared* al nodo reconciliado y *generaDiff* fue correctamente sincronizado de acuerdo a la política *_resolutor="timeStamp*", puesto que el documento *B* en *_id="1.1"* tiene mayor *_timeStamp* que el documento *A.* Así, *generaDiff* heredó el valor que tenía en el documento *B*. Se prueba de esta manera que el resolutor TimeStamp sincroniza correctamente.

- En el nodo *_id="1.1.1"* se heredó *resolutor="_syncTimes"* que tiene mayor prioridad que *_resolutor="timeStamp*". Por otro lado, el atributo *valor="A"* fue correctamente sincronizado por SyncTimes, dado que *_syncTimes="25"* en el documento *A*, es mayor a _syncTimes="9" del documento *B.* Se muestra así que la reconciliación por resolutor SyncTimes se realiza correctamente.

5.1.2. Documentos sólo con diferencias de tipo 'eliminación' de nodos

La 'eliminación' de nodos, o mejor dicho la *marcación para eliminación* de nodos se realiza de manera muy similar a la reconciliación de nodos modificados vista en la sección 5.1.1. El atributo *_visible* se reconcilia tal como si fuera un atributo no protegido que genera diferencias, por lo que se utilizan resolutores en su sincronización.

La particularidad del atributo *_visible* es que cuando un nodo reconciliado resulta ser marcado como *_visible="false",* todos sus descendientes deben ser marcados como *_visible="false"* pues es lógico que si se marca un nodo como borrado, todos sus descendientes también lo sean. Se muestra una prueba realizada sobre nodos marcados para eliminación en la figura 39.

```
<root
 _id="1" _resolutor="role" _timeStamp="12" _role="1" _syncTimes="5"
 _visible="false">

 <node
  _id="1.1" _resolutor="timeStamp" _timeStamp="10"
  _role="1" _syncTimes="5"
  _visible="true">

  <nodeSon
  _id="1.1.1" _resolutor="timeStamp"
  _timeStamp="10" _role="1" _syncTimes="12"
  _visible="true"/>

 </node>

 <node
  _id="1.2" _resolutor="role" _timeStamp="14"
  _role="2" _syncTimes="25"
  _visible="true"/>

</root>
```

Documento A

```
<root
 _id="1" _resolutor="timeStamp"
 _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node
  _id="1.1" _resolutor="timeStamp" _timeStamp="15"
  _role="2" _syncTimes="4"
  _visible="false">

  <nodeSon
   _id="1.1.1" _resolutor="syncTimes"
   _timeStamp="11" _role="2" _syncTimes="9"
   _visible="true"/>

 </node>

 <node
  _id="1.2" _resolutor="role" _timeStamp="1"
  _role="2" _syncTimes="25" _visible="false"/>

</root>
```

Documento B

Figura 39: Documentos A y B para prueba de marcación de nodos para eliminación

```
<root
 _id="1" _timeStamp="12" _role="3" _syncTimes="6" _resolutor="role"
 _visible="true">

  <node
   _id="1.1" _timeStamp="15" _role="2" _syncTimes="6" _resolutor="timeStamp"
   _visible="false">

    <nodeSon
     _id="1.1.1" _timeStamp="11" _role="2" _syncTimes="13" _resolutor="syncTimes"
     _visible="false" />

  </node>

  <node
  _id="1.2" _timeStamp="14" _role="2" _syncTimes="26" _resolutor="role"
  _visible="true" />

</root>
```

Figura 40: Documento reconciliado entre *A* y *B*

- El nodo *_id="1*" se reconcilia con la política *_resolutor="role"* y en este caso resulta que ésta política tiene mayor prioridad en el nodo *_id="1"* del documento *B*. Como resultado de esto, el atributo *_visible* hereda su valor del documento *B.* Se ve en la figura 40 que el atributo *_visible* el nodo reconciliado *_id="1"* está correctamente sincronizado. Se prueba así que la sincronización de nodos marcados para eliminación se lleva a cabo correctamente.

- El nodo *_id="1.1"* fue reconciliado bajo la política *_resolutor="timeStamp",* y en este caso la política decide correctamente heredar los atributos desde el nodo *_id="1.1"* del documento *B* que es quien tiene mayor prioridad dada la política *timeStamp*. Por lo tanto, el atributo _visible en el nodo reconciliado toma el valor "false", o sea, es marcado para eliminación. Como ocurre esto, todos sus descendientes deben ser marcados para eliminación, aunque la política particular de cada uno de ellos diga otra cosa. En este caso, su único descendiente es *_id="1.1.1".* Por política de reconciliación, este nodo debería ser marcado como *_visible="true",* sin embargo esto estaría incorrecto, pues su padre está marcado como eliminado. Así, el que esté marcado como *_visible="false"* es muestra de que la marcación de nodos como eliminados se realiza correctamente.

Se podría pensar que con estas pruebas se puede decir que toda la reconciliación para eliminación de nodos funciona correctamente. Sin embargo, esto no es así. Cuando un nodo compartido es marcado para eliminación y tiene descendientes en un documento y en el otro no, tales descendientes son agregados en el documento reconciliado pues son nodos no compartidos. Una vez agregados estos nodos deben ser marcados también para eliminación, dado que su ancestro fue marcado para eliminación. Este caso se ve muestra más en detalle en la sección 5.1.4.

5.1.3. Documentos sólo con diferencias de tipo agregación de nodos

En la presente sección se verán algunas de las pruebas realizadas sobre documentos con diferencias sólo de tipo agregación de nodos. En la figura 41 se pueden ver los documentos *A* y *B,* que no comparten nodos (salvo la raíz). Por lo tanto, todos sus nodos deben ser agregados al documento reconciliado.

```
<root _id="1" _resolutor="role" _timeStamp="12" _role="1" _syncTimes="5" _visible="true">

 <node
  _id="1.4" _resolutor="timeStamp" _timeStamp="10"
  _role="1" _syncTimes="5"
  _visible="true">

  <nodeSon
  _id="1.4.1" _resolutor="timeStamp"
  _timeStamp="10" _role="1" _syncTimes="12"
  _visible="true"/>

  <nodeSon
    _id="1.4.2" _resolutor="timeStamp"
    _timeStamp="10" _role="1" _syncTimes="12"
  _visible="true"/>

 </node>

 <node
  _id="1.5" _resolutor="role" _timeStamp="14"
  _role="2" _syncTimes="25"
  _visible="true"/>

</root>
```

Documento A

```
<root
 _id="1" _resolutor="timeStamp"
 _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node
  _id="1.1" _resolutor="timeStamp" _timeStamp="15"
  _role="2" _syncTimes="4"
  _visible="false">

  <nodeSon
   _id="1.1.1" _resolutor="syncTimes"
   _timeStamp="11" _role="2" _syncTimes="9"
   _visible="true"/>

 </node>

 <node
  _id="1.2" _resolutor="role" _timeStamp="1"
  _role="2" _syncTimes="25" _visible="false"/>

</root>
```

Documento B

Figura 41: Documentos *A* y *B* para pruebas de sólo agregación

```
<root _id="1" _timeStamp="12" _role="3" _syncTimes="6" _resolutor="role" _visible="true">

  <node _id="1.1"
   _resolutor="timeStamp" _timeStamp="15" _role="2" _syncTimes="4" _visible="false">

    <nodeSon _id="1.1.1"
    _resolutor="syncTimes" _timeStamp="11" _role="2" _syncTimes="9" _visible="false" />

  </node>

  <node  _id="1.2"
   _resolutor="role" _timeStamp="1" _role="2" _syncTimes="25" _visible="false" />

  <node _id="1.4"
   _resolutor="timeStamp" _timeStamp="10" _role="1" _syncTimes="5" _visible="true">

    <nodeSon _id="1.4.1"
     _resolutor="timeStamp" _timeStamp="10" _role="1" _syncTimes="12" _visible="true" />

    <nodeSon _id="1.4.2"
     _resolutor="timeStamp" _timeStamp="10" _role="1" _syncTimes="12" _visible="true" />
  </node>

  <node _id="1.5"
   _resolutor="role" _timeStamp="14" _role="2" _syncTimes="25" _visible="true" />

</root>
```

Figura 42: Documento reconciliado entre *A* y *B*

Se puede ver que en el documento reconciliado, se agregan todos los nodos no compartidos de los documentos *A* y *B* sin modificar ninguno de sus atributos. Así se prueba que la reconciliación para documentos con sólo agregaciones, funciona correctamente.

5.1.4. Documentos con modificación, eliminación y agregación de nodos

En la presente sección se presentan algunas pruebas realizadas a documentos con diferencias de todos tipos. En la figura 43 se muestran los documentos *A* y *B* utilizados para las pruebas.

```
<root _id="1"
 _resolutor="role" _timeStamp="12" _role="1" _syncTimes="5"
 _visible="true">

 <node
   _id="1.1" _resolutor="timeStamp" _timeStamp="10"
   _role="2" _syncTimes="4"
   _visible="false">

   <nodeSon
    _id="1.1.1" _resolutor="syncTimes"
    _timeStamp="11" _role="2" _syncTimes="9"
    _visible="false"/>

 </node>

 <node
  _id="1.4" _resolutor="timeStamp" _timeStamp="10"
  _role="1" _syncTimes="5"
  _visible="true"/>

</root>
```

Documento A

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node _id="1.1"
  _resolutor="timeStamp" _timeStamp="5" _role="2" _syncTimes="4"
  _visible="true">

  <nodeSon _id="1.1.1"
   _resolutor="syncTimes" _timeStamp="11" _role="2" _syncTimes="9"
   _visible="true"/>

   <nodeSon _id="1.1.2"
    _resolutor="syncTimes" _timeStamp="11" _role="2" _syncTimes="9"
    _visible="true"/>

 </node>

 <node _id="1.2"
  _resolutor="role" _timeStamp="1" _role="2" _syncTimes="25"
  _visible="false"/>

</root>
```

Documento B

Figura 43: Documentos *A* y *B* utilizados para pruebas de agregación, modificación y eliminación

En la figura 44 se muestra el documento reconciliado entre *A* y *B.*

```
<root _id="1"
 _timeStamp="12" _role="3" _syncTimes="6" _resolutor="role" _visible="true">

  <node _id="1.1"
   _timeStamp="10" _role="2" _syncTimes="5" _resolutor="timeStamp"
   _visible="false">

    <nodeSon _id="1.1.1"
     _timeStamp="11" _role="2" _syncTimes="10" _resolutor="syncTimes"
     _visible="false" />

    <nodeSon _id="1.1.2"
     _resolutor="syncTimes" _timeStamp="11" _role="2" _syncTimes="9"
     _visible="false" />

  </node>

  <node _id="1.2"
   _resolutor="role" _timeStamp="1" _role="2" _syncTimes="25" _visible="false" />

  <node _id="1.4"
   _resolutor="timeStamp" _timeStamp="10" _role="1" _syncTimes="5" _visible="true" />

</root>
```

Figura 44: Documento reconciliado entre *A* y *B*

Se pueden ver varios aspectos interesantes en el documento mostrado en la figura 44:

- Se insertaron correctamente todos los nodos no compartidos por ambos documentos. Por ejemplo están los nodos *_id="1.4"*, *_id="1.2"* y *_id="1.1.2".*

- Se reconciliaron correctamente los nodos compartidos que tenían modificaciones. Están los nodos *_id="1.1", _id="1.1.1" y _id="1"* correctamente sincronizados.

- Cabe destacar que el nodo *_id="1.1.2"* en el documento reconciliado se encuentra marcado para eliminación *_visible="false".* En este nodo se da el caso que se mencionaba en la sección 5.1.2. Aunque este nodo originalmente no estaba marcado para eliminación *_visible="true"* y no fue sincronizado por modificaciones, dado que es no compartido; se agrega como marcado para eliminación puesto que su padre (o alguno de sus ancestros en el caso general) en el documento reconciliado está marcado para eliminación. Se ve entonces que la reconciliación de nodos marcados para eliminación está funcionando correctamente

5.2. Pruebas para sincronización de N documentos XML

Se realizaron variadas pruebas de sincronización para N documentos XML, pero se muestran en la figura 45 los documentos *A, B, C y D* utilizados para una de ellas. Estos documentos son pequeños, pero representativos de casos generales de sincronización que se podrían dar.

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node _id="1.1"
  _resolutor="timeStamp" _timeStamp="1" _role="3" _syncTimes="2"
  _visible="true">

  <node _id="1.2" valor="A"
    _resolutor="timeStamp" _timeStamp="5" _role="2" _syncTimes="4"
  _visible="true">

</root>
```

Documento A

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node _id="1.2" valor="B"
  _resolutor="role" _timeStamp="2" _role="1" _syncTimes="3"
  _visible="true">

  <node _id="1.3"
    _resolutor="timeStamp" _timeStamp="5" _role="2" _syncTimes="4"
  _visible="true">

</root>
```

Documento B

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node _id="1.2" valor="C"
  _resolutor="syncTimes" _timeStamp="3" _role="2" _syncTimes="1"
  _visible="true">

  <node _id="1.4" valor="C"
    _resolutor="timeStamp" _timeStamp="5" _role="2" _syncTimes="4"
  _visible="true">

</root>
```

Documento C

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1"
 _visible="true">

 <node _id="1.4" valor="D"
  _resolutor="timeStamp" _timeStamp="10" _role="12" _syncTimes="6"
  _visible="true">

</root>
```

Documento D

Figura 45: Documentos *A, B, C y D* para pruebas

En la figura 46 se muestra el documento reconciliado para *A, B, C y D*.

```
<root _id="1"
 _resolutor="timeStamp" _timeStamp="5" _role="3" _syncTimes="1" _visible="true">

  <node _id="1.4"
   _timeStamp="10" _role="12" _syncTimes="7" _resolutor="timeStamp" _visible="true"
   valor="D" />

  <node _id="1.2"
   _timeStamp="5" _role="2" _syncTimes="6" _resolutor="role" _visible="true"
   valor="A" />

  <node _id="1.3"
   _resolutor="timeStamp" _timeStamp="5" _role="2" _syncTimes="4" _visible="true" />

  <node _id="1.1"
  _resolutor="timeStamp" _timeStamp="1" _role="3" _syncTimes="2" _visible="true" />

</root>
```

Figura 46: Documentos reconciliado entre *A, B, C y D*

Se puede ver que todos los nodos no compartidos son agregados al documento reconciliado. Es el caso de los nodos: *_id="1.1", _id="1.3".* Los nodos compartidos son correctamente sincronizados. Se toma a modo de ejemplo el nodo *_id="1.2"* compartido por los documentos *A, B y C.* El atributo *valor* genera diferencias, por lo que debe ser sincronizado. Se tienen las políticas de resolución para este nodo: En A *_resolutor="timeStamp",* en *B _resolutor="role"* y *en C _resolutor="syncTimes".* La política de resolución con mayor prioridad es *_resolutor="role"* por lo que es escogida para la reconciliación. Se escoge el del atributo en el nodo del documento A, pues es este nodo el que posee un *_role* mayor.

5.3. Análisis de eficiencia

En la presente sección se mostrarán algunas pruebas realizadas con el fin de medir la eficiencia en cuanto a tiempo de ejecución y uso de memoria del módulo creado.

Se intenta determinar un patrón de comportamiento de la aplicación desarrollada cuando se sincronizan dos documentos XML con árboles de distinta profundidad y grado.

Se crearon documentos XML sincronizables con distintas profundidades, desde profundidad 1 hasta 160; y con distintos grados, o cantidad de hijos, desde grado 1 hasta 300. Algunos de los árboles que se crearon se llenaron a su máxima capacidad de nodos y otros se llenaron aleatoriamente de nodos. Luego se procedió a sincronizar los documentos que se habían creado y se tomaron los tiempos de reconciliación. Los resultados obtenidos se muestran en las siguientes tablas y gráficos.

5.3.1. Tiempo de ejecución dependiendo de la profundidad de los árboles

Con distintas profundidades, y grado 2 del documento *A,* y profundidad y grado 1 en el documento *B* se muestran los resultados obtenidos en la tabla 9.

Profundidad A	Grado A	Profundidad B	Grado B	**Tiempo**
1	2	1	1	4,09
2	2	1	1	4,39
3	2	1	1	4,8
4	2	1	1	5,06
5	2	1	1	6,07
8	2	1	1	32,14
10	2	1	1	269,82

Tabla 9: Tiempos para distintas profundidades

El gráfico 1 muestra la curva que se desprende de la tabla 9.

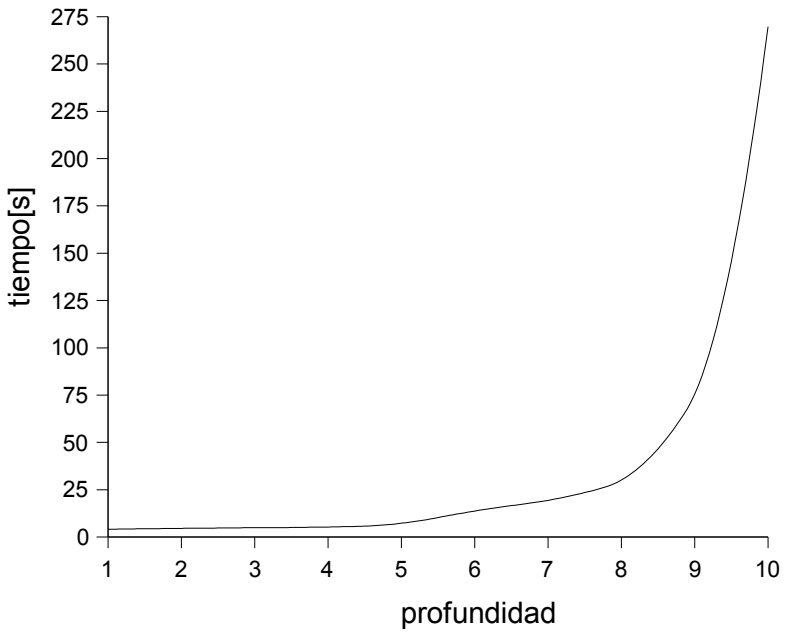


Gráfico 1: Profundidad v/s Tiempo

Se ve claramente que la curva de tiempo para la profundidad de un árbol se parece mucho a una exponencial. Se sabe que la cantidad de nodos que tiene un árbol de profundidad *p* y grado *g* está dada por:

$$G^{P+1} - 1$$

Se podría pensar que la curva representada viene dada por una fórmula como esta, o en otras palabras, que el tiempo de ejecución de la aplicación depende de la cantidad de nodos que posean los árboles a sincronizar. Sin embargo, para asegurarse de que esto es así, y analizar separadamente la influencia de la profundidad del árbol en el tiempo de ejecución, se sincroniza un árbol con grado 1 y profundidad variable.

Profundidad A	Grado A	Profundidad B	Grado B	**Tiempo**
1	1	1	1	4,1
10	1	1	1	4,36
20	1	1	1	5,13
40	1	1	1	8,72
60	1	1	1	18,74
80	1	1	1	31,04
100	1	1	1	53,81
150	1	1	1	298,32

Tabla 10: Tiempos para distintas profundidades con grado 1

El gráfico 2 muestra la correspondencia de la tabla 10.

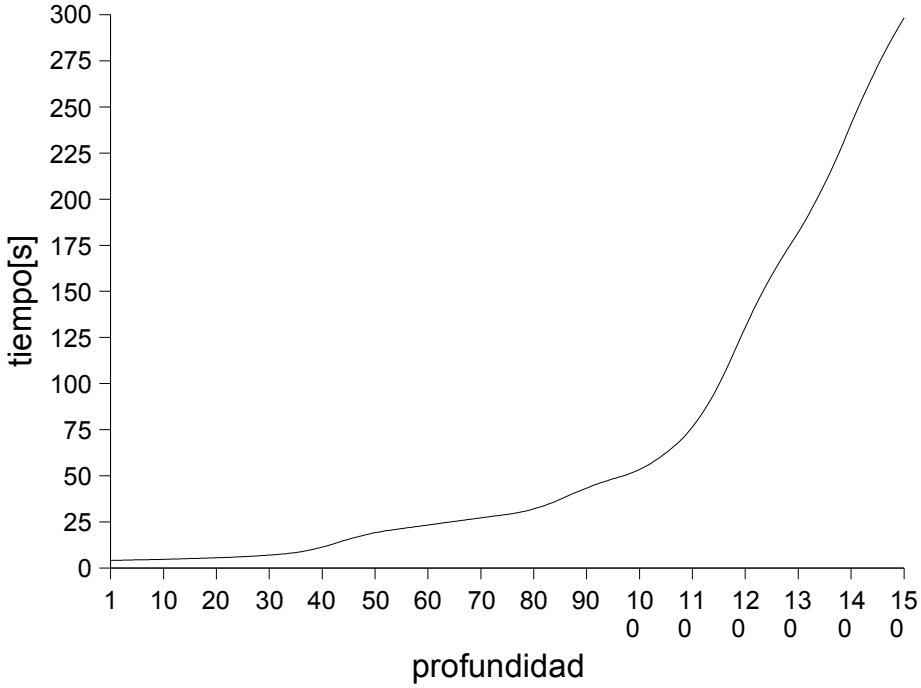


Gráfico 2: Profundidad v/s tiempo

Se puede notar que aunque el grado del árbol *A* es 1, la curva que se dibuja se parece a una exponencial. Como el grado de *A* es 1, en realidad el árbol que se forma para distintas profundidades es simplemente una lista. Así, la cantidad de nodos de los árboles formados con distintas profundidades es la igual a su profundidad. Entonces, se deduce de estos gráficos que la profundidad de los documentos XML a sincronizar implica un tiempo exponencial de la ejecución de la aplicación.

5.3.2. Tiempo de ejecución dependiendo del grado de los árboles

A continuación se mostrará una relación entre el grado de los árboles de los documentos XML y el tiempo de ejecución de la sincronización. Se armaron árboles con distintos grados, o cantidad de hijos por nodo, y se tabularon los tiempos de ejecución de la aplicación.

Profundidad A	Grado A	Profundidad B	Grado B	**Tiempo**
1	1	1	1	4,1
1	10	1	1	4,25
1	20	1	1	4,69
1	40	1	1	6,29
1	60	1	1	8,59
1	80	1	1	8,75
1	100	1	1	10,66
1	150	1	1	18,27
1	200	1	1	27,3
1	300	1	1	53,4

Tabla 11: Tiempos para distintos grados con profundidad 1

El gráfico 3 muestra la correspondencia de la tabla 11.

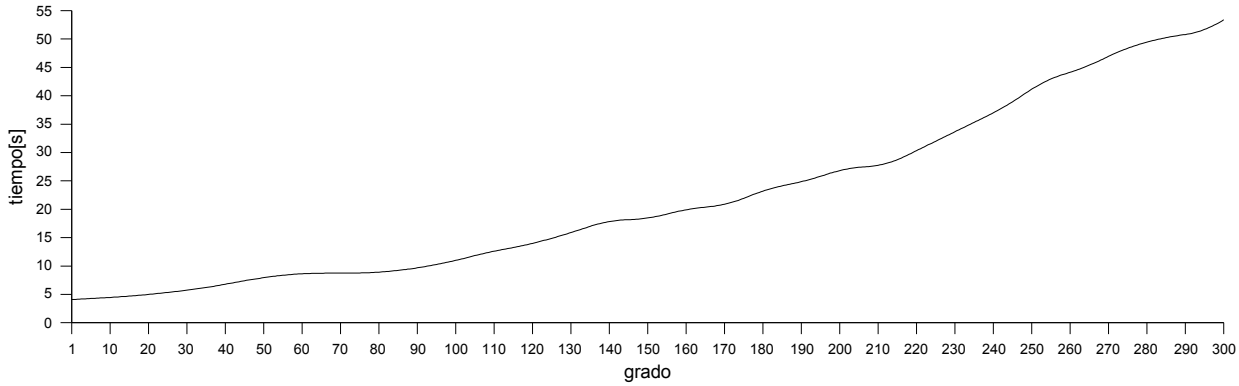


Gráfico 3: Grado v/s tiempo

Se puede ver en el gráfico un comportamiento aproximadamente lineal entre el grado de los árboles sincronizados y el tiempo de ejecución de la aplicación de reconciliación.

5.3.3. Tiempo de ejecución dependiendo de la cantidad de nodos

A continuación se intenta definir una relación entre la cantidad de nodos a sincronizar entre los dos documentos XML y el tiempo de ejecución de la reconciliación. En la tabla 12 se muestran los tiempos de ejecución tabulados para distintos grados y profundidades de los documentos. Se generaron árboles completos para esta prueba, es decir, con la máxima cantidad posible de nodos para cada combinación grado-profundidad.

Profundidad A	Grado A	Profundidad B	Grado B	**Tiempo**
2	2	2	2	6,23
2	3	2	3	4,5
2	4	2	4	4,91
3	2	3	2	4,78
2	5	2	5	5,29
3	3	3	3	6,3
3	4	3	4	11,16
4	3	4	3	10,27
3	5	3	5	12,02
4	4	4	4	20,18
5	4	5	4	128,81

Tabla 12: Tiempos para distintos grados y profundidades de los documentos a reconciliar

Utilizando la fórmula para la cantidad de nodos para árboles completos dada la profundidad y el grado de un árbol, se pueden llevar los datos de la tabla 12 a los que se muestran en la tabla 13, que serán finalmente los que se graficarán.

Cantidad de nodos	**Tiempo**
14	6,23
52	4,5
30	4,78
126	4,91
160	6,3
248	5,29
484	10,27
510	11,16
1248	12,02
2046	20,18
8190	128,81

Tabla 13: Cantidad de nodos entre ambos documentos a sincronizar, y tiempos de ejecución

Los datos que se muestran en la tabla 13 provienen de el cálculo de la cantidad de nodos de los documentos *A y B* dadas las profundidades y grados de ambos. El gráfico 4 muestra la correspondencia de los datos de la tabla 13.

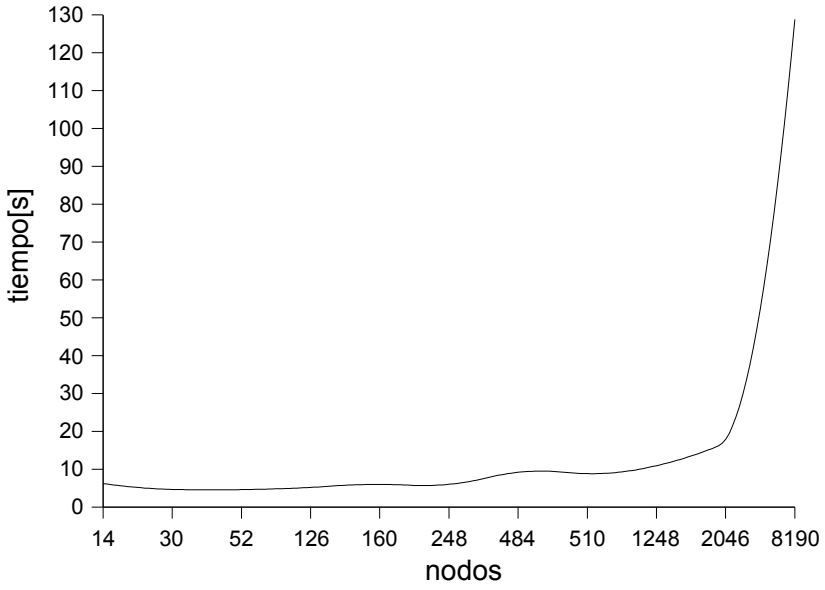


Gráfico 4: Cantidad de nodos v/s tiempo

Se muestra un comportamiento exponencial, sin embargo hay que tener muy en cuenta que los rangos del eje X también tienen comportamiento exponencial. Se linealiza logarítmicamente el rango de X y se vuelve a graficar. El resultado de esta operación se ve en el gráfico 5.

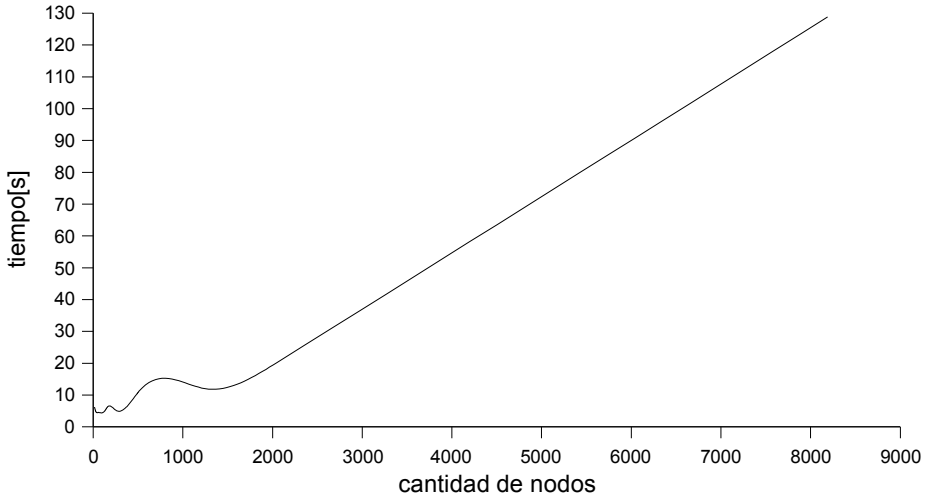


Gráfico 5: Cantidad de nodos v/s tiempo

Se puede ver una relación prácticamente lineal entre la cantidad de nodos a sincronizar y el tiempo que se demora en sincronizar la aplicación de reconciliación desarrollada.

Teniendo en cuenta todos los datos tomados y el análisis de los gráficos presentados se puede concluir que mientras más grado tengan los árboles de los documentos XML a sincronizar, aumentará linealmente el tiempo que tome la reconciliación. Mientras más profundos sean los árboles de los documentos, aumentará exponencialmente el tiempo que tome la reconciliación. Y mientras más nodos tengan los árboles de los documentos XML a reconciliar, aumentará linealmente el tiempo que tome la reconciliación.

6. Conclusiones y Trabajo a Futuro

El presente trabajo de título se enmarcó en un ambiente donde trabajadores móviles comparten documentos XML, y modifican sus copias locales. Utilizan dispositivos móviles para tal fin, en particular PDA's. Las constantes modificaciones de los documentos XML que comparten hicieron necesario el contar con alguna herramienta capaz de sincronizar o reconciliar las copias locales de estos documentos. Se necesitaba además que tal aplicación realizara reconciliaciones que fueran en lo posible, completamente automáticas o sin intervención del usuario.

Se analizaron varias aplicaciones ya desarrolladas que reconcilian documentos XML y que ejecutan sobre PDA's. Ninguna de ellas presentaba una real solución a los requerimientos que planteaba el escenario en el que se enmarca este trabajo de título. Por esta razón, se decidió desarrollar una aplicación que fuera lo más automática posible para la reconciliación de copias de documentos XML. Esta aplicación se basó en el modelo planteado por el algoritmo implementado para XMIDDLE [Mascolo, 2002]. Si bien, la aplicación desarrollada tomó algunas ideas principales del modelo de reconciliación de XMIDDLE, se diseñó completamente desde cero.

Mientras se estaba diseñando la aplicación, se definió un modelo de documentos XML "sincronizables". Fue así como se definieron algunos atributos protegidos que debían tener los nodos de los documentos XML a sincronizar por la aplicación desarrollada. El más importante de estos atributos protegidos fue el llamado *_id*, atributo que identifica únicamente cada nodo de cada árbol XML dentro de la red de trabajadores móviles en la que se enmarca este trabajo de título.

Con la identificación única de nodos dentro de los documentos XML compartidos, se tienen 3 tipos de cambios que los trabajadores móviles pueden hacer a sus copias locales de los XML que comparten: *Modificación:* cuando un trabajador móvil cambia el valor de algún atributo de un nodo ya existente en su copia local del documento XML compartido, *Agregación:* cuando un trabajador móvil inserta un nuevo nodo en la copia local del documento que comparte y *Eliminación:* cuando un trabajador móvil elimina algún nodo de su copia local.

Al reconciliar dos o más copias de documentos XML, los nodos que tengan el mismo identificador *_id* se consideran nodos compartidos. Si los nodos compartidos tienen atributos con distinto valor en las copias compartidas, entonces se sabe que hubo acciones de *modificación.* Así, si un trabajador móvil modifica cierto nodo en su copia local de un documento XML compartido, se podrá identificar qué nodo fue efectivamente modificado y compararlo con *el mismo nodo* de las demás copias locales del documento. Los nodos que no estén compartidos, es decir cuyo identificador no se encuentre en ninguna otra copia de documento XML compartido, se consideran *agregados*. No se implementa la acción de *eliminación,* sino que se realiza una marcación de los nodos que se deseen eliminar.

Otros atributos protegidos definidos tienen que ver con los resolutores. Los resolutores se definieron como entes encargados de resolver discrepancias entre nodos que se hayan modificado. Así cuando dos o más documentos XML se sincronizan, las *modificaciones* que

los trabajadores hayan realizado sobre ellos, se reconcilian usando resolutores. Los resolutores se implementan como clases que heredan de una clase abstracta Resolutor. Se definieron tres resolutores en el presente trabajo de título. Bajo este modelo de implementación, si se necesitan más resolutores, simplemente pueden desarrollarse e integrarse a la aplicación heredando de la clase Resolutor. Cada resolutor define políticas de reconciliación y es la definición de éstas políticas las que logran que el proceso de sincronización sea automático.

El módulo de sincronización fue desarrollado primeramente para dos documentos XML. Sin embargo, se desarrolló pensando en la sincronización de N documentos XML. Una vez concluido el módulo de dos documentos XML se utilizó éste para hacer la sincronización extensiva para N documentos XML. Esta extensión es posible debido a las propiedades de los algoritmos de reconciliación que se definieron.

Se considera cumplido el objetivo del presente trabajo de título dado que la aplicación desarrollada es completamente aplicable al ambiente para el cual fue ideada.

A continuación se presentan algunos puntos donde se considera interesante seguir trabajando a futuro:

- Desarrollo de una aplicación que transforme cualquier documento XML en sincronizable según el criterio de reconciliación del presente trabajo de título. Esto sería muy interesante y no tan complicado de llevar a la práctica. Se pueden tomar en cuenta algunas sugerencias que se dan en el capítulo 3, sección 3.1, principalmente con respecto a la identificación de nodos.

- Diseño más acabado del protocolo de sincronización para N documentos XML planteado en el presente trabajo de título. El protocolo diseñado no considera algunos aspectos que podrían ser muy útiles en ciertos ambientes, como por ejemplo versionamiento de documentos. Puede ser útil para este fin el atributo de *_syncTimes* que indica el número de veces que ha sido sincronizado un nodo.

- Desarrollo de un protocolo de eliminación real de nodos. La aplicación desarrollada en el presente trabajo de título no realiza la eliminación real de nodos. Esto puede que no sea apropiado en algunos ambientes de trabajo donde sea muy común la eliminación de nodos, o se trabaje por mucho tiempo sobre los mismos documentos XML. El no poder borrar nodos realmente, puede resultar en que los documentos XML comiencen a crecer indefinidamente.

- Desarrollo de una aplicación con la cual se puedan fácilmente modificar las copias locales de los documentos XML compartidos. Por ejemplo, tal aplicación podría implementar una interfaz de *agregación* de nodos, que automáticamente identifique únicamente el nodo que se está agregando; otra interfaz de *modificación* de atributos de los nodos; y otra de *eliminación*. Sería interesante que esta aplicación pudiera dar la opción al usuario de cambiar o agregar resolutores a los nodos que esté modificando.

- Adaptación de la aplicación desarrollada en el presente trabajo de título para que las

políticas de resolutores puedan definirse fácilmente para un documento XML completo o para ramas de el mismo, o nodos. En la actualidad, la aplicación sólo permite la definición de resolutores por nodo individualmente. Esto puede resultar un poco engorroso al momento de modificar el documento XML.

7. Bibliografía y Referencias

[Harmony, 2006] Unison. Internet: http://www.cis.upenn.edu/~bcpierce/unison/ Última visita: Noviembre 2006.

[Jiang, 2003] Haifeng Jiang, Hongjun Lu, Wei Wang. XR-Tree: Indexing XML Data for Efficient Structural Joins. 19th International Conference on Data Engineering, 2003. pp 253 – 264. March 2003

[Kleinrock, 1996] Kleinrock, L. Nomadicity: Anytime, Anywhere in A Disconnected World. Mobile Networks and Applications. Mobile Networks and Applications pp. 351 – 357. January 1996.

[Komvoteas, 2003] Kyriakos Komvoteas. XML Diff and Patch Tool. September 5, 2003.

[Lam, 2002] Franky Lam, Nicole Lam, Raymond Wong. Efficient Synchronization for Mobile XML Data. Conference on Information and knowledge management, Virginia, USA. pp. 153 – 160. 2002.

[Lindholm, 2003] Tancred Lindholm. XML Three way Merge as a Reconciliation Engine for Mobile Data. International Workshop on Data Engineering for Wireless and Mobile Access, pp. 93 – 97. 2003.

[Mascolo, 2002] Cecilia Mascolo, Licia Capra, Stefanos Zachariadis and Wolfgang Emmerich. xmiddle: A Data-Sharing Middleware for Mobile Computing. Wireless Personal Communications: An International Journal, pp. 77 – 103. 2002.

[MultiSync, 2006] Internet http://multisync.sourceforge.net/news.php Última visita: Noviembre 2006.

[Musolesi, 2002] Mirco Musolesi. A Middleware for Data-sharing in Ad Hoc Networks. Tesi di Laurea in Ingegneria Elettronica. Facolta' di Ingegneria. University of Bologna. December 2002.

[OMA, 2007] Open Mobile Alliance. http://www.openmobilealliance.org/tech/wg_committees/ds.html. Última Visita: Junio, 2007.

[O'Reilly, 2007] O'Reilly: XML.com. Internet: http://www.xml.com, http://www.ciberaula.com/curso/xml/que_es Última visita: Julio, 2007

[Roman, 2000] Gruia-Catalin Roman, Gian Pietro Picco, Amy L. Murphy. Software engineering for mobility: a roadmap. International Conference on Software Engineering. Limerick, Ireland. pp. 227 – 239. 2000.

[Unison, 2006] Intenet http://www.cis.upenn.edu/~bcpierce/unison/ Última visita Noviembre 2006.

[Varadero, 2006] Varadero Software Factory, S.L. Internet: http://www.interactivanet.com/tecnologia.php. Última visita: Agosto, 2006.

[Wong, 2004] Raymond K. Wong. Collaborative Hypertext Editing in Mobile Environment. Proceedings of the 10th International Multimedia Modelling Conference, pp. 300. 2004.

8. Anexos

```
using System;
using System.Xml;
using System.Xml.XPath;

namespace XMLSync.XMLSyncCore.DiffTree
{

   public class DiffTree
   {
      private static XmlDocument difftree = null;

      public XmlDocument getDiffTree()
      {
         return difftree;
      }

      public DiffTree(XmlDocument treeA, XmlDocument treeB)
      {
         difftree = new XmlDocument();
         difftree.AppendChild(difftree.CreateElement("treediff"));
         compute(treeA, treeB);
      }


      private void processTreeA(System.Xml.XmlNode nodeA, System.Xml.XmlDocument treeB)
      {
         XmlNode auxB = null;
         XmlNode auxA = nodeA;
         XmlElement diffRoot = (XmlElement)difftree.DocumentElement;

         if (auxA == null)
            return;

         //recorro el árbol A
         if (!auxA.NodeType.Equals(System.Xml.XmlNodeType.Text))
         {
            while (auxA != null)
            {
               auxB = this.findNode(treeB, auxA); //veo si el nodo está en el árbol B
               XmlElement eAuxA = (XmlElement)auxA;

               if (auxB != null) //sí está.
               {
                  //marco los nodos como _visited para la posterior revisión por processTreeB
                  XmlElement eAuxB = (XmlElement)auxB;
                  eAuxB.SetAttribute("_visited", "_true");
                  eAuxA.SetAttribute("_visited", "_true");

                  bool equal = areNodesEqual(auxA, auxB); //veo si hay modificaciones

                  if (!equal) //hay modificaciones
                  {
                     // NODO MODIF
                     try
                     {
                        XmlNode modif = difftree.CreateNode(System.Xml.XmlNodeType.Element, "MODIF", "");

                        //agrego los atributos que van en el nodo MODIF
                        XmlNode attrId = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_id", "");
                        if (attrId != null)
                        {
                           if (eAuxA.GetAttributeNode("_id") != null)
                           {
                              attrId.Value = (eAuxA.GetAttributeNode("_id")).Value;
                              modif.Attributes.SetNamedItem(attrId);
                           }
                        }
```

```
XmlNode resolutorB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_resolutorB", "");
if (resolutorB != null)
{
   if (eAuxB.GetAttributeNode("_resolutor") != null)
   {
      resolutorB.Value = (eAuxB.GetAttributeNode("_resolutor")).Value;
      modif.Attributes.SetNamedItem(resolutorB);
   }
}
XmlNode resolutorA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_resolutorA", "");
if (resolutorA != null)
{
   if (eAuxA.GetAttributeNode("_resolutor") != null)
   {
      resolutorA.Value = (eAuxA.GetAttributeNode("_resolutor")).Value;
      modif.Attributes.SetNamedItem(resolutorA);
   }
}


XmlNode visibleA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_visibleA", "");
if (visibleA != null)
{
   if (eAuxA.GetAttributeNode("_visible") != null)
   {
      visibleA.Value = (eAuxA.GetAttributeNode("_visible")).Value;
      modif.Attributes.SetNamedItem(visibleA);
   }
}
XmlNode visibleB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_visibleB", "");
if (visibleB != null)
{
   if (eAuxB.GetAttributeNode("_visible") != null)
   {
      visibleB.Value = (eAuxB.GetAttributeNode("_visible")).Value;
      modif.Attributes.SetNamedItem(visibleB);
   }
}

XmlNode timeStampA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_timeStampA", "");
if (timeStampA != null)
{
   if (eAuxA.GetAttributeNode("_timeStamp") != null)
   {
      timeStampA.Value = (eAuxA.GetAttributeNode("_timeStamp")).Value;
      modif.Attributes.SetNamedItem(timeStampA);
   }
}

XmlNode timeStampB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_timeStampB", "");
if (timeStampB != null)
{
   if (eAuxB.GetAttributeNode("_timeStamp") != null)
   {
      timeStampB.Value = (eAuxB.GetAttributeNode("_timeStamp")).Value;
      modif.Attributes.SetNamedItem(timeStampB);
   }
}

XmlNode rolA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_roleA", "");
if (rolA != null)
{
   if (eAuxA.GetAttributeNode("_role") != null)
   {
      rolA.Value = (eAuxA.GetAttributeNode("_role")).Value;
      modif.Attributes.SetNamedItem(rolA);
   }
}
XmlNode rolB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_roleB", "");
if (rolB != null)
{
   if (eAuxB.GetAttributeNode("_role") != null)
```

```
    {
        rolB.Value = (eAuxB.GetAttributeNode("_role")).Value;
        modif.Attributes.SetNamedItem(rolB);
    }
}


XmlNode vecesSincrA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_syncTimesA", "");
if (vecesSincrA != null)
{
    if (eAuxA.GetAttributeNode("_syncTimes") != null)
    {
        vecesSincrA.Value = (eAuxA.GetAttributeNode("_syncTimes")).Value;
        modif.Attributes.SetNamedItem(vecesSincrA);
    }
}
XmlNode vecesSincrB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_syncTimesB", "");
if (vecesSincrB != null)
{
    if (eAuxB.GetAttributeNode("_syncTimes") != null)
    {
        vecesSincrB.Value = (eAuxB.GetAttributeNode("_syncTimes")).Value;
        modif.Attributes.SetNamedItem(vecesSincrB);
    }
}

//agrego los atributos del nodoA a MODIF
for (int i = 0; i < auxA.Attributes.Count; i++)
{
    XmlAttribute attrAddA = auxA.Attributes[i];
    if (!attrAddA.Name.StartsWith("_"))//no debo considerar los atributos protegidos
    {
        //reviso si este atributo está en el nodo de B
        XmlAttribute attrAddB = eAuxB.GetAttributeNode(attrAddA.Name);
        if (attrAddB != null)
        {
            if (!attrAddB.Value.Equals(attrAddA.Value))
            {
                //el atributo está en B y el valor que tiene es distinto al de A
                //entonces lo agrego como hijo del nodo MODIF para que sea sincronizado
                XmlNode addAttrSon = difftree.CreateNode(System.Xml.XmlNodeType.Element, attrAddB.Name, "");
                XmlNode exValueA = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "valueA", "");
                exValueA.Value = attrAddA.Value;
                XmlNode exValueB = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "valueB", "");
                exValueB.Value = attrAddB.Value;
                addAttrSon.Attributes.SetNamedItem(exValueA);
                addAttrSon.Attributes.SetNamedItem(exValueB);
                modif.AppendChild(addAttrSon);
            }
            else
            {
                //el atributo estaba en B, pero tiene el mismo valor que el de A así es que se agrega a MODIF
                modif.Attributes.SetNamedItem(attrAddA);
            }
        }
        else
        {
            //si el atributo no estaba en B debo agregarlo en MODIF
            modif.Attributes.SetNamedItem(attrAddA);
        }
    }

}

//agrego los atributos del nodoB que aún no se han agregado a MODIF
for (int j = 0; j < auxB.Attributes.Count; j++)
{
    XmlAttribute attrAddB = auxB.Attributes[j];
    if (!attrAddB.Name.StartsWith("_"))//no debo considerar los atributos protegidos
    {
        //reviso si este atributo está en el nodo MODIF
        XmlElement eModif = (XmlElement)modif;
```

```
                        XmlAttribute attrModif = eModif.GetAttributeNode(attrAddB.Name);
                        if (attrModif == null) //si el atributo de B no está en MODIF, debo agregarlo
                        {
                            modif.Attributes.SetNamedItem(attrAddB);
                        }
                    }
                }

                /*
                 * debo tener en cuenta que los atributos repetidos en A y B que tenian un valor distinto
                 * no deben agregarse en el nodo MODIF, sino como hijos de él.
                 * Como agregué todos los atributos de B, estos atributos repetidos también se agregaron,
                 * por lo que debo removerlos del nodo MODIF
                 * */
                XmlNode modifHijo = modif.FirstChild;
                while (modifHijo!=null)
                {
                    String atribARemover = modifHijo.Name;
                    XmlElement eModif = (XmlElement)modif;
                    try
                    {
                        eModif.RemoveAttribute(atribARemover);
                    }
                    catch (Exception exp) { }
                    modifHijo = modifHijo.NextSibling;
                    //recordar que el nodo MODIF sólo tiene 1 nivel de hijos por construcción.
                }

                diffRoot.AppendChild(modif);
            }
            catch (Exception el)
            {
                System.Console.WriteLine(el.Message);
            }
            //END NODO MODIF
        }
        else
        {
            //no hay modificaciones, los nodos de A y B son iguales. Se hace nada.
        }
    }

    else
    {
        //el nodoA no estaba en B, se produce un nodo ADD.
        //NODO ADD
        XmlNode add = difftree.CreateNode(System.Xml.XmlNodeType.Element, "ADD", "");
        //debo insertar el atributo que indica el id del padre donde insertar el nodo
        String id = eAuxA.GetAttribute("_id");//identificador del nodo a insertar
        String idPadre = getIdPadre(id); //busco el padre del nodo a insertar
        //como hijo de ADD inserto el nodo que debe agregarse
        try
        {
            add.AppendChild(difftree.ImportNode(auxA,false));
        }
        catch (Exception e) { System.Console.WriteLine(e.Message); }
        XmlNode idP = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_idPadre", "");
        if (idP != null)
        {
            idP.Value = idPadre;
            add.Attributes.SetNamedItem(idP);
        }
        diffRoot.AppendChild(add);
        //END NODO ADD
    }
    processTreeA(auxA.FirstChild, treeB);
    auxA = auxA.NextSibling;
  }
 }
}
```

```
        private void processTreeB(XmlNode nodeB)
        {
            XmlNode aux = nodeB;
            if (aux == null) return;
            XmlElement diffRoot = (XmlElement)difftree.DocumentElement;

            while (aux != null)
            {
                //debo agregar los nodos que no estén marcados como visitados
                //primero rescato el identificador del padre
                XmlElement eAux = (XmlElement)aux;
                String idPadre = eAux.GetAttribute("_id");//identificador del nodo padre
                XmlNode auxHijo = aux.FirstChild;

                while (auxHijo != null)
                {
                    XmlElement eAuxHijo = (XmlElement)auxHijo;
                    //ahora recorro todos sus hijos
                    //revisando si este hijo contiene el atributo _visited
                    XmlAttribute visited = eAuxHijo.GetAttributeNode("_visited");
                    if (visited == null)//si el nodo no había sido visitado
                    {
                        //NODO ADD
                        XmlNode add = difftree.CreateNode(System.Xml.XmlNodeType.Element, "ADD", "");
                        try
                        {
                            add.AppendChild(difftree.ImportNode(auxHijo, false));
                        }
                        catch (Exception e) { System.Console.WriteLine(e.Message); }
                        XmlNode idP = difftree.CreateNode(System.Xml.XmlNodeType.Attribute, "_idPadre", "");
                        if (idP != null)
                        {
                            idP.Value = idPadre;
                            add.Attributes.SetNamedItem(idP);
                        }
                        diffRoot.AppendChild(add);
                        //END NODO ADD
                    }
                    else //si fue visitado
                    {
                        //hago nada.
                    }
                    //paso al siguiente hijo de este nodo
                    auxHijo = auxHijo.NextSibling;
                }

                processTreeB(aux.FirstChild);
                aux = aux.NextSibling;
            }
        }

        private void compute(System.Xml.XmlDocument treeA, System.Xml.XmlDocument treeB)
        {
            XmlNode auxA = treeA.FirstChild;
            processTreeA(auxA, treeB);
            processTreeB(treeB.FirstChild);
            cleanTree(treeA.FirstChild);
            cleanTree(treeB.FirstChild);
        }

    }
}

namespace XMLSync.XMLSyncCore.Reconciliation
{
    public class Reconcile
    {
        private static System.Collections.ArrayList resolutors;

        public XmlDocument reconcile(XmlDocument doc,XmlDocument diferencias) {
```

```
XmlDocument rec = new XmlDocument();
rec = doc;
if (rec != null)
{
   XmlNode auxDiff = diferencias.FirstChild;
   if (auxDiff != null)
   {
      //reviso que el árbol de diferencias esté bien formado
      if (auxDiff.Name.Equals("treediff"))
      {
         //comienzo a recorrer secuencialmente el árbol de diferencias
         auxDiff = auxDiff.FirstChild;
         while (auxDiff != null)
         {
            //SINCRONIZAR MODIF
            if (auxDiff.Name.Equals("MODIF"))
            {
               XmlNode nodoA = findNode(rec, auxDiff);
               XmlElement eNodoA = (XmlElement)nodoA;
               XmlElement eAuxDiff = (XmlElement)auxDiff;
               if (nodoA != null)
               {
                  eNodoA.RemoveAllAttributes();

                  //todos los atributos no protegidos del nodo MODIF deben ser agregados al nodoA
                  //estos atributos no necesitan reconciliarse pues no generaron problemas de sincronización
                  for (int i = 0; i < auxDiff.Attributes.Count; i++)
                  {
                     if (!auxDiff.Attributes[i].Name.StartsWith("_"))
                     {
                        try
                        {
                           XmlAttribute atrib = rec.CreateAttribute(eAuxDiff.Attributes[i].Name);
                           if (atrib != null)
                           {
                              atrib.Value = eAuxDiff.Attributes[i].Value;
                              nodoA.Attributes.Append(atrib);
                           }
                        }
                        catch (Exception exe)
                        {
                           System.Console.WriteLine(exe.Message);
                        }
                     }
                  }

                  //se sincronizan los atributos protegidos y demás atributos no protegidos
                  //que generaron diferencias.

                  //criterios de sincronizacion. Se Busca el resolutor a utilizar
                  XmlAttribute resoA = eAuxDiff.GetAttributeNode("_resolutorA");
                  String resolutorA = "";
                  if (resoA != null)
                  {
                     resolutorA = resoA.Value;
                  }
                  else
                  {
                     resolutorA = "";
                  }

                  XmlAttribute resoB = eAuxDiff.GetAttributeNode("_resolutorB");
                  String resolutorB = "";
                  if (resoB != null)
                  {
                     resolutorB = resoB.Value;
                  }
                  else
                  {
                     resolutorB = "";
                  }
```

```
String resolutor = prioritizeResolutors(resolutorA, resolutorB);

//se obtienen los criterios para sincronizar
XmlAttribute critA = eAuxDiff.GetAttributeNode("_" + resolutor + "A");
String criterioA = "";
if (critA != null)
{
   criterioA = critA.Value;
}
XmlAttribute critB = eAuxDiff.GetAttributeNode("_" + resolutor + "B");
String criterioB = "";
if (critB != null)
{
   criterioB = critB.Value;
}

//comienza la sincronización

//atributos protegidos
//_id: Queda igual

XmlAttribute sincr = eAuxDiff.GetAttributeNode("_id");
if (sincr != null)
{
   String valor = sincr.Value;
   sincr = rec.CreateAttribute(sincr.Name);
   if (sincr != null)
   {
      sincr.Value = valor;
      nodoA.Attributes.Append(sincr);
   }
}

//_timeStamp: queda el mayor timeStamp
String timeA = "",timeB = "";
int timeStampA = 0, timeStampB = 0;
sincr = eAuxDiff.GetAttributeNode("_timeStampA");
if (sincr != null)
{
   timeA = sincr.Value;
}
sincr = eAuxDiff.GetAttributeNode("_timeStampB");
if (sincr != null)
{
   timeB = sincr.Value;
}
try
{
   timeStampA = int.Parse(timeA);
   timeStampB = int.Parse(timeB);
}
catch (Exception ex)
{}
if (sincr != null)
{
   sincr = rec.CreateAttribute("_timeStamp");
   if (sincr != null)
   {
      String valor = "0";
      if (timeStampA >= timeStampB)
         valor = timeStampA + "";
      else
         valor = timeStampB + "";
      sincr.Value = valor;
      nodoA.Attributes.Append(sincr);
   }
}

//_role: queda el mayor
String roleA = "", roleB = "";
int rolA = 0, rolB = 0;
sincr = eAuxDiff.GetAttributeNode("_roleA");
```

```
if (sincr != null)
{
   roleA = sincr.Value;
}
sincr = eAuxDiff.GetAttributeNode("_roleB");
if (sincr != null)
{
   roleB = sincr.Value;
}
try
{
   rolA = int.Parse(roleA);
   rolB = int.Parse(roleB);
}
catch (Exception ex)
{ }
if (sincr != null)
{
   sincr = rec.CreateAttribute("_role");
   if (sincr != null)
   {
      String valor = "0";
      if (rolA >= rolB)
         valor = rolA + "";
      else
         valor = rolB + "";
      sincr.Value = valor;
      nodoA.Attributes.Append(sincr);
   }
}

//_syncTimes: se deja el mayor más 1.
String syncA = "", syncB = "";
int syncTA = 0, syncTB = 0;
sincr = eAuxDiff.GetAttributeNode("_syncTimesA");
if (sincr != null)
{
   syncA = sincr.Value;
}
sincr = eAuxDiff.GetAttributeNode("_syncTimesB");
if (sincr != null)
{
   syncB = sincr.Value;
}
try
{
   syncTA = int.Parse(syncA);
   syncTB = int.Parse(syncB);
}
catch (Exception ex)
{ }
if (sincr != null)
{
   sincr = rec.CreateAttribute("_syncTimes");
   if (sincr != null)
   {
      String valor = "0";
      if (syncTA >= syncTB)
         valor = (syncTA + 1) + "";
      else
         valor = (syncTB + 1) + "";
      sincr.Value = valor;
      nodoA.Attributes.Append(sincr);
   }
}

//_resolutor: se deja el de mayor prioridad
sincr = eAuxDiff.GetAttributeNode("_resolutorA");
if (sincr != null)
{
   String resol = resolutor;
   sincr = rec.CreateAttribute("_resolutor");
```

```
            if (sincr != null)
            {
                sincr.Value = resol;
                nodoA.Attributes.Append(sincr);
            }
        }

        //_visible: se reconcilia de acuerdo al resolutor
        sincr = eAuxDiff.GetAttributeNode("_visibleA");
        String visiA = "", visiB = "";
        if (sincr != null)
        {
            visiA = sincr.Value;
        }
        sincr = eAuxDiff.GetAttributeNode("_visibleB");
        if (sincr != null)
        {
            visiB = sincr.Value;
        }
        String visiFinal = doReconcile(visiA, visiB, criterioA, criterioB, resolutor);
        //si no se pudo resolver la visibilidad se deja visible
        if (visiFinal.Equals(""))
            visiFinal = "true";
        sincr = rec.CreateAttribute("_visible");
        if (sincr != null)
        {
            sincr.Value = visiFinal;
            nodoA.Attributes.Append(sincr);
        }

        //ahora se sincronizan todos los atributos que tenían diferencias.
        for (int i = 0; i < auxDiff.ChildNodes.Count; i++)
        {
            XmlNode auxSon = auxDiff.ChildNodes[i];
            XmlElement eAuxSon =(XmlElement)auxSon;
            String attrName = auxSon.Name;
            XmlAttribute vA = eAuxSon.GetAttributeNode("valueA");
            String attrValueA="";
            if(vA!=null)
                attrValueA = vA.Value;
            XmlAttribute vB = eAuxSon.GetAttributeNode("valueB");
            String attrValueB = "";
            if (vB != null)
                attrValueB = vB.Value;

            String attrValueFin = doReconcile(attrValueA, attrValueB, criterioA, criterioB, resolutor);
            sincr = rec.CreateAttribute(attrName);
            if (sincr != null)
            {
                sincr.Value = attrValueFin;
                nodoA.Attributes.Append(sincr);
            }
        }
    }
}
//END SINCRONIZAR MODIF

//SINCRONIZAR ADD
else if(auxDiff.Name.Equals("ADD"))
{
    XmlElement eAuxDiff = (XmlElement)auxDiff;
    XmlAttribute idPadre = eAuxDiff.GetAttributeNode("_idPadre");
    String _idPadre = "";
    if (idPadre != null)
    {
        _idPadre = idPadre.Value;
    }
    for (int i = 0; i < auxDiff.ChildNodes.Count; i++)
    {
        XmlNode auxSon = auxDiff.ChildNodes[i];
        XmlElement eAuxSon = (XmlElement)auxSon;
        XmlNode padreAdd = findNode(rec,_idPadre);
```

```
                        if (padreAdd != null)
                        {
                            //inserto sólo si el padre existe en el archivo rec
                            //y el nodo a insertar no existe. Para no repetirlo
                            if (findNode(rec, auxSon) == null)
                            {
                                try
                                {
                                    padreAdd.AppendChild(rec.ImportNode(auxSon, false));
                                }
                                catch (Exception ex)
                                {
                                    System.Console.WriteLine(ex.Message);
                                }
                            }
                        }

                    }
                }
                auxDiff = auxDiff.NextSibling;
            }
            //marco como eliminados los hijos de nodos eliminados.
            if (rec != null)
            {
                processDeletions(rec.FirstChild);
            }
        }
        else
        {
            System.Console.WriteLine("ERROR: árbol de diferencias erróneo");
        }
    }
}
else
    System.Console.WriteLine("ERROR: documento a reconciliar está vacío");
return rec;
}

private static String doReconcile(String valorA, String valorB, String criterioA, String criterioB, String resolutor)
{
    Resolutor res = null;
    bool found = false;

    for (System.Collections.IEnumerator i = resolutors.GetEnumerator(); i.MoveNext(); )
    {
        res = (Resolutor)i.Current;
        if (res.policy.Equals(resolutor))
        {
            found = true;
            break;
        }
    }

    if (found)
        return res.reconcile(valorA, valorB, criterioA, criterioB);
    else
    {
        System.Console.Out.WriteLine("Reconcile: No hay resolutor "+resolutor+".");
    }
    return "";
}

}
}
```