Trabajo de Grado

Hipp

Leandro Quiroga

Directora: Alicia Díaz

Co Director: Alejandro Fernández

Agradecimiento

Agradezco a todos los que en mayor o menor medida me ayudaron tanto en mi desarrollo como profesional como también en mi crecimiento como persona.

Resumen

Los sistemas *groupware* ayudan a grupos de personas a colaborar para alcanzar un objetivo común. Dentro de este conjunto de sistemas, un subconjunto particular es el compuesto por los sistemas *groupware* sincrónicos. En ellos, las personas interactúan en tiempo real en pos de un objetivo común. Uno de los desafíos más interesantes en el desarrollo de *groupware* sincrónico es el mantenimiento de la consistencia de la información presentada a los usuarios.

Hoy en día, la mayoría de los sistemas *groupware* sincrónicos se basan en una arquitectura de coordinación centralizada (cliente - servidor). Utilizando este tipo de arquitectura, la implementación de las técnicas de mantenimiento de consistencia resulta más sencilla. Tanto el uso de bloqueos como el establecimiento de un orden total sobre las operaciones se simplifican teniendo una entidad distinguida encargada de su administración. En una arquitectura de coordinación distribuida (*peer to peer*), el uso de estas técnicas tradicionales resulta complejo y presenta desventajas, lo cual anima al uso de técnicas no tradicionales.

Una técnica de mantenimiento de consistencia no muy utilizada es *Operational Transformation* (*OT*). En esta técnica, los cambios se encapsulan en operaciones. Cada uno de los nodos genera estas operaciones, que son ejecutadas primero localmente y luego propagadas al resto de los pares. Cuando una operación llega a un nodo, antes de ejecutarse, se determina si dicha operación se generó en forma concurrente con alguna otra, en cuyo caso se procederá a transformarla para que tenga en cuenta los cambios realizados por esta última.

En este contexto, se evaluó la posibilidad de encapsular el funcionamiento de un algoritmo de OT en un *framework* que facilite el desarrollo de *groupware* sincrónico con arquitectura de coordinación distribuida. El resultado positivo de esta evaluación permitió definir Hipp, un *framework* basado en OT para el desarrollo de sistemas *groupware* sincrónicos con arquitectura peer to peer. Éste se basa en modelos replicados y posee un diseño en capas que independiza cada una de ellas. Su principal objetivo es el mantenimiento de consistencia de la información que se le presenta al usuario. Por ello delega los detalles de la comunicación peer to peer en un software de comunicación que puede ser intercambiado.

Como complemento del *framework* diseñado e implementado se entrega una guía de uso que incluye también buenas prácticas y un editor colaborativo de *XML* implementado utilizando Hipp a modo de ejemplo de uso y prueba de funcionamiento.

Indice

Capítulo 1

Introducción

En este primer capítulo se pretende introducir al lector en algunos de los conceptos, definiciones y técnicas más importantes para este trabajo de grado. Se plantea su motivación describiendo sin entrar en detalle las problemáticas existentes y se enuncia su objetivo. Por último se describe la estructura de este informe.

Un sistema es *groupware* cuando ayuda a un grupo de personas a colaborar para alcanzar un objetivo común. Así, el propósito del *groupware* es asistir a grupos de personas en la comunicación, colaboración y coordinación de sus actividades [Ellis 2].

Específicamente Ellis, Gibbs y Rein, definen *groupware* como sistemas basados en computadoras que mantienen a grupos de personas acoplados en una tarea común y proveen una interfaz a un ambiente compartido. Las nociones de "tarea común" y "ambiente compartido" son cruciales para esta definición, ya que excluyen a los sistemas multiusuarios, como los sistemas *time-sharing*, cuyos usuarios no comparten una tarea común. Se debe notar también que esta definición no especifica que los usuarios están activos simultáneamente. Los sistemas *groupware* que específicamente soportan actividad simultánea se denominan *groupware sincrónicos*.

En los sistemas *groupware sincrónicos*, las acciones de los usuarios deben ser propagadas rápidamente al resto de los usuarios. Por ejemplo, en un juego de fútbol entre dos usuarios, las acciones que efectúe un usuario deberán ser propagadas lo más rápidamente posible al otro usuario, para que el juego luzca lo más real posible.

Una *sesión* de un sistema *groupware sincrónico* consiste en un grupo de personas conectadas, llamadas usuarios o participantes. La duración de las sesiones solo está determinada por la conexión del primer usuario y la desconexión del último de estos, no teniendo en general una duración típica. La sesión provee a cada participante una interfaz a un contexto compartido, por ejemplo, los participantes pueden observar vistas de datos que evolucionan. El *tiempo de respuesta* del sistema es el tiempo necesario para que una acción de un usuario se refleje en su propia interfaz. Mientras que el *tiempo de notificación* es el tiempo necesario para que una acción de un usuario sea propagada a las interfaces del resto de los usuarios.

Las características más importantes de los sistemas *groupware sincrónicos* son [Ellis 1]:

- Alta interactividad, el *tiempo de respuesta* deberá ser ínfimo.
- Tiempo real, los *tiempos de notificación* deberán ser comparables a los *tiempos de respuesta*.
- Distribución, en general, no se puede asumir que los participantes estarán conectados a la misma computadora o a la misma LAN.
- Volatilidad, los usuarios son libres de entrar y/o salir de una sesión en el momento en que lo deseen.
- Ad-hoc, generalmente los usuarios no seguirán una secuencia de pasos planificada, no es posible decir *a priori* que información será accedida.
- Enfocada, durante una sesión existirá un alto grado de conflictos de acceso, cuando los participantes trabajen sobre los mismos datos.

La mayoría de los sistemas *groupware* de la actualidad se basan en una arquitectura Cliente – Servidor. En esta arquitectura existe una entidad distinguida, el servidor, a cargo de la coordinación del sistema. En ocasiones, el modelo de objetos o datos (según corresponda) se encuentra concentrado en el servidor, siendo éste el encargado de la ejecución de las operaciones que genera cada uno de los clientes y de la propagación del

resultado a todos ellos. La otra posibilidad, es que el modelo se encuentre replicado en los clientes. De esta forma, el servidor sólo tendrá la responsabilidad de recibir los cambios de los clientes y propagarlos al resto de ellos. Existe también la posibilidad de un híbrido entre las alternativas anteriores, lo cual implica que el servidor contenga una parte del modelo y los clientes otra, dividiendo así las responsabilidades sobre estos.

El hecho de tener una entidad distinguida encargada de la coordinación, otorga la ventaja de simplificar los algoritmos de control de concurrencia. Sin embargo, tener un servidor central, implica también tener un punto crítico de fallas. Cualquier fallo en el servidor dejará inútil la totalidad del sistema. Además, la escalabilidad del sistema estará limitada por el desempeño del servidor.

Las aplicaciones *groupware peer to peer*, se basan en una arquitectura distribuida. Esto significa que no requieren de entidades únicas para su funcionamiento como si lo hace la arquitectura Cliente – Servidor. Por el contrario, éstas incrementan la disponibilidad de servicios, distribuyéndolos entre un gran número de nodos, que se supone no fallarán todos a la vez. De esta forma, el modelo de objetos o datos se encontrará replicado en cada uno de los nodos participantes, siendo ellos mismos los encargados de propagar los cambios que se efectúen localmente, al resto de sus pares.

A medida que la Web continúa creciendo, tanto en contenido como en conexiones de dispositivos, los sistemas *peer to peer* empiezan a tener una real preponderancia. Ejemplo de esto son los sistemas para compartir archivos, la computación distribuida y los servicios de mensajería instantánea.

Este tipo de aplicaciones, a pesar que cada una de ellas tiene propósitos diferentes, comparten muchas actividades tales como descubrimiento de nodos, búsquedas y envío de archivos, mensajes o datos. Actualmente el desarrollo de aplicaciones *peer to peer* es ineficiente, debido mayormente a que los desarrolladores resuelven una y otra vez los mismos problemas multiplicando así implementaciones de infraestructura similares.

En los sistemas *groupware* que contienen espacios compartidos, como por ejemplo una pizarra colaborativa, uno de los retos más relevantes es el mantener sincronizada la información que se muestra a los diferentes usuarios. Es decir, que la información

presentada al grupo, sea *consistente*. Ésta, se considera una tarea compleja debido a que las intervenciones de los usuarios en el espacio compartido se realizan en forma concurrente.

La mayoría de los sistemas *groupware* que utilizan modelos replicados, primero aplican localmente los cambios efectuados por los usuarios en el modelo compartido y luego propagan dichos cambios al resto de los usuarios. Generalmente se utiliza esta estrategia debido a que el aplicar los cambios localmente primero, reduce drásticamente el *tiempo de respuesta*. Esta estrategia se conoce como *actualización local optimista*, ya que aplica y muestra los cambios antes de confirmar que estos serán definitivos.

Cuando dos o más usuarios generan cambios simultáneos en el modelo compartido, estos son aplicados en diferente orden en cada nodo. Si estos cambios son conflictivos entre sí, es decir su orden de aplicación no es conmutativo, el resultado que verá cada usuario será diferente y por lo tanto, el sistema se encontrará en un estado *inconsistente*.

Veamos este problema con un ejemplo, consideremos un rompecabezas colaborativo, en el cual un grupo de personas (cada uno en su propia computadora) intenta resolver un rompecabezas compartido moviendo y encastrando piezas concurrentemente. Para simplificarlo, supongamos que el grupo solo está compuesto por dos participantes. Tanto los movimientos como los encastres de las piezas, producidos por cada usuario, se propagarán al otro usuario. En este escenario, una situación conflictiva sucede cuando dos usuarios deciden simultáneamente tomar la misma pieza y moverla a distintos destinos. Cada usuario aplicará el cambio localmente y lo propagará a su compañero. Esto hace que en cada computadora los cambios se apliquen en diferente orden y por lo tanto, en el caso de dos movimientos de piezas, se llegue a resultados divergentes.

La ocurrencia de estos conflictos está directamente relacionada con el *tiempo de notificación*, ya que cuanto mayor sea éste, mayor será la probabilidad de que cambios no exactamente simultáneos se solapen.

El mantenimiento de consistencia, se considera central en este trabajo de tesis y se detalla en el siguiente capítulo.

Los *frameworks* son una técnica de reuso en el paradigma de orientación a objetos. Un *framework* constituye la estructura interna de las aplicaciones, dando la posibilidad de configurar los detalles específicos de esta. Así, un *framework* es una aplicación reusable semi-completa que puede ser especializada para producir aplicaciones específicas.

Existen una gran cantidad de *frameworks* desarrollados para diferentes tipos de aplicaciones. Un subgrupo de ellos está orientado al desarrollo de aplicaciones *groupware*. Algunos ejemplos de estos son COAST [Schuckmann], un *framework groupware* que provee independencia del mecanismo de persistencia y comunicación subyacentes, DyCE [Tietze] un *framework* orientado al desarrollo de componentes *groupware* y Chatblocks [Naso] un *framework* para la construcción de aplicaciones de comunicación textual sincrónica.

La capacidad de reutilización que proveen los *frameworks* permite encapsular el conocimiento del dominio, aprovechando el esfuerzo invertido en su investigación, con el objetivo de evitar el continuo desarrollo de soluciones comunes a requerimientos repetidos.

La creación de un *framework* de manejo de consistencia permitiría el encapsulamiento de la complejidad que éste insume y su reutilización en diferentes dominios.

1. Objetivo y resultados esperados

El objetivo de este trabajo de tesis es contribuir en el desarrollo de aplicaciones *groupware* sincrónicas basadas en una arquitectura de coordinación distribuida. Esta ayuda estará dirigida a evitar que se desarrollen una y otra vez soluciones de mantenimiento de consistencia que podrían ser reutilizadas en diferentes dominios de aplicación.

Dicha contribución permitirá a los desarrolladores enfocarse en los objetivos propios de sus aplicaciones, relegando la atención a los requerimientos generales de los sistemas *groupware*. De esta forma se busca simplificar y acelerar el desarrollo de las aplicaciones prescindiendo de las etapas de diseño, implementación, *testing* y

mantenimiento de los algoritmos de mantenimiento de consistencia de la información compartida y de las actividades básicas de comunicación de toda aplicación *groupware*.

Para ello se dispone crear un *framework* para el desarrollo de sistemas *groupware* sincrónicos con arquitectura de coordinación distribuida (*peer to peer*) el cual se enfocará en el mantenimiento de la consistencia de la información compartida por los usuarios.

El *framework* desarrollado deberá permitir la creación, búsqueda y el acceso de los usuarios a una sesión compartida con otros usuarios y algún mecanismo para el envío y recepción de información entre ellos.

Por otra parte, dicho *framework* deberá generar el soporte necesario para que los sistemas desarrollados cumplan con las características de las aplicaciones *groupware* sincrónicas descriptas anteriormente, alta interactividad, tiempo real, distribución, volatilidad, *ad-hoc* y enfocada.

Como corresponde a todo *framework*, el *framework* desarrollado tendrá consigo una guía de uso del mismo. En esta guía se incluirá el detalle de las extensiones necesarias y optativas para su instanciación, los pasos requeridos para su inicialización y una descripción de cada una de sus funcionalidades.

Por último, se requerirá la implementación de un prototipo de aplicación sencilla que muestre el funcionamiento del *framework* y describa cada uno de los pasos que se siguieron en su desarrollo.

2. Organización del informe

Este trabajo está organizado de la siguiente forma, en el próximo capítulo se analizará el problema de mantenimiento de consistencia, explicando qué significa consistencia y las implicancias de su mantenimiento. Además se estudiará el mantenimiento de consistencia en una arquitectura de coordinación centralizada y en otra distribuida.

Luego, el capítulo 3, está dedicado a *Operational Transformation* (*OT*), que es la estrategia de mantenimiento de consistencia elegida para este trabajo. En él, se describe la historia y objetivos de dicha estrategia, su funcionamiento, una descripción breve de los algoritmos analizados, su comparación y elección para su desarrollo. Seguido de esto, se detalla la necesidad de reutilizar los algoritmos de manejo de consistencia y se muestra que la reutilización de los algoritmos de OT es posible. Además se presentan los *frameworks* como herramientas de reutilización.

El capítulo 4 incluye el objetivo, la descripción de la arquitectura y el diseño del *framework* implementado (*Hipp*). En este capítulo se describe detalladamente cada capa de *Hipp*, como se realiza la interacción entre ellas y se presenta una guía de uso del mismo. Ésta se estructura en tres secciones que describen las extensiones que deben realizársele al *framework* para cada dominio de aplicación, cómo se efectúa la inicialización de *Hipp* y cómo utilizar las funcionalidades que éste provee.

Siguiendo a éste, en el capítulo 5 se presenta un ejemplo de uso de *Hipp*. El ejemplo consiste en un editor colaborativo de *XML*. En este capítulo se enuncia el objetivo de su desarrollo, una descripción de la funcionalidad que presenta, el diseño macro de su modelo, el análisis de sus operaciones y funciones de transformación, su integración con *Hipp*, el proceso de verificación que se llevó a cabo y las conclusiones propias de su desarrollo.

A continuación, se detallan las conclusiones del trabajo y su perspectiva en el capítulo 6. Por último, en una sección aparte, se presentan las referencias a artículos consultados para la confección de esta tesis.

Capítulo 2

Mantenimiento de Consistencia

En este capítulo se detalla la noción de consistencia, se establece la causa fundamental de la generación de inconsistencias en la información compartida y se presentan las dos técnicas más conocidas para su mantenimiento. A continuación se describen las arquitecturas de coordinación centralizada y distribuida y se comentan las ventajas y/o inconvenientes de la implementación de las dos técnicas anteriores en cada una de ellas.

1. Introducción

La Real Academia Española define consistencia como *coherencia entre las partículas de una masa o los elementos de un conjunto*. En el rubro que nos convoca, los sistemas *groupware* basados en modelos replicados, interpretamos el ***conjunto*** como el sistema *groupware* y los ***elementos*** como la información compartida por los nodos de un sistema *groupware*.

Un nodo de un sistema *groupware* consiste en una aplicación, manejada o no por un usuario, la cual interactúa con otros nodos. Esta aplicación, está compuesta a su vez, por un modelo de datos que representa la información compartida y por un conjunto de operaciones que se pueden realizar sobre los datos. Las operaciones constituyen conjuntos de instrucciones que modifican los datos.

Para una definición más formal de *consistencia*, utilizaremos algunas otras definiciones que repasaremos a continuación.

Definición. La **relación de orden causal (→)** [Lamport] establece que dadas dos operaciones O_a y O_b generadas por los nodos i y j respectivamente, decimos que O_a ***precede*** a O_b ($O_a \rightarrow O_b$) si y solo si alguna de las siguientes condiciones se satisface:

- i = j y la creación de O_a ocurrió antes que la creación O_b.
- $i \neq j$ y la ejecución de O_a en el nodo j ocurrió antes de la creación de O_b.
- Existe una operación O_x tal que $O_a \rightarrow O_x$ y $O_x \rightarrow O_b$.

Es importante destacar la diferencia entre la creación de una operación y su ejecución. La creación de una operación es el proceso anterior a la ejecución de la misma en el que se establecen las acciones a realizar, mientras que la ejecución aplica efectivamente dichas acciones en el modelo del sistema. Notar además que la relación de orden causal es un orden parcial, ya que pueden existir operaciones O_a y O_b tales que ni O_a precede a O_b ni O_b precede a O_a.

Definición. La ***propiedad de precedencia*** [Ellis 1] establece que si una operación O_a ***precede*** a otra operación O_b, entonces en cada nodo la ejecución de O_a ocurre antes que la ejecución de O_b.

Definición. Una sesión *groupware* está ***en reposo*** (***quiescent***) [Ellis 1] si y solo si todas las operaciones generadas han sido ejecutadas en todos los nodos, es decir, no hay requerimientos en tránsito o esperando por ser ejecutados en ningún nodo.

Definición. La ***propiedad de convergencia*** [Ellis 1] establece que un sistema groupware es ***convergente*** si el modelo de cada nodo es idéntico en todos los nodos cuando la sesión *groupware* está ***en reposo***.

Definición. Un sistema *groupware* es ***consistente*** [Ellis 1] si y solo si, la ***propiedad de convergencia*** y la ***propiedad de precedencia*** se satisfacen siempre.

El mantenimiento de consistencia de la información compartida en un sistema *groupware* se basa en eso, en hacer que se satisfagan las propiedades de convergencia y de precedencia.

La causa fundamental de la aparición de inconsistencias es que el envío de cambios desde un nodo a otro consume una cantidad impredecible de tiempo. Este tiempo es impredecible pues surge de la latencia de la red, el tiempo de procesamiento, el tiempo de encolado, etc. [Hofte]. Los problemas aparecen cuando se propagan cambios conflictivos entre sí, al mismo tiempo desde diferentes nodos. Esto resultará en distintos órdenes de aplicación de los cambios en los nodos y por lo tanto resultados divergentes.

Por ejemplo, dos personas que comparten una pizarra colaborativa desean cambiar el color de una figura. Mariana decide pintarla de rojo mientras que, al mismo tiempo, Leandro lo hace con blanco. Al propagarse estos cambios a la computadora del otro, sobrescriben el cambio hecho localmente. Lo cual hace que se obtengan resultados diferentes en cada computadora (ver figura 2.1).

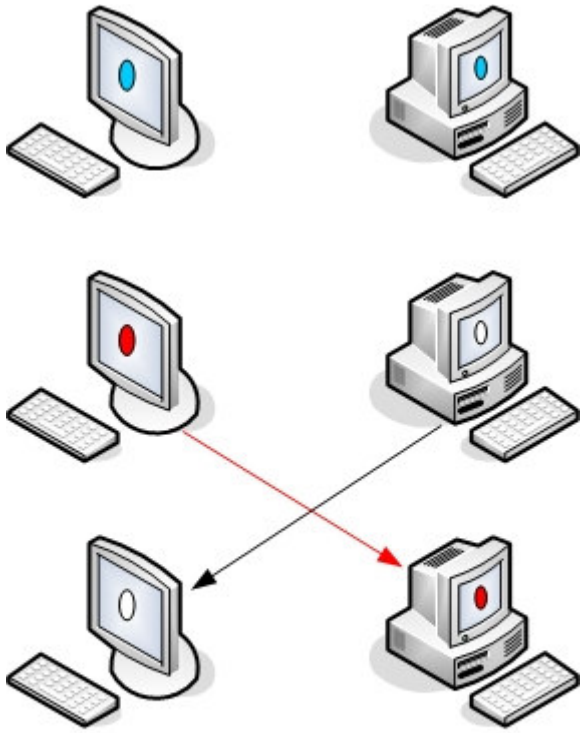

Figura 2.1 Surgimiento de inconsistencias.

Una alternativa de solución a este problema es establecer un orden total global a todas las operaciones. Este se basa en el hecho que si todos los nodos parten de un mismo estado y ejecutan una serie de operaciones en el mismo orden, el resultado final en cada uno de ellos será el mismo.

Otra estrategia es no aceptar operaciones que puedan causar inconsistencias, por ejemplo negándoles ciertas operaciones a ciertos usuarios. Esto puede llevarse a cabo mediante bloqueos. Los bloqueos son una forma de coordinación que le dan temporalmente el privilegio de realizar operaciones que puedan causar inconsistencias a un usuario a la vez y se lo restringe a los otros mientras este lo tenga.

2. Arquitectura de coordinación centralizada

En un sistema *groupware*, una arquitectura de coordinación centralizada es aquella en la que existe una entidad que se distingue sobre las otras y que tiene el rol de coordinar la comunicación entre el resto de las entidades del sistema. Generalmente a esta entidad distinguida se la denomina servidor y al resto de las entidades se las llama clientes.

Mantener consistencia en la información compartida en este tipo de sistemas no suele ser demasiado complejo. Esto se debe a que se cuenta con una entidad distinguida que permite la sincronización de los eventos de comunicación que surgen de los distintos clientes.

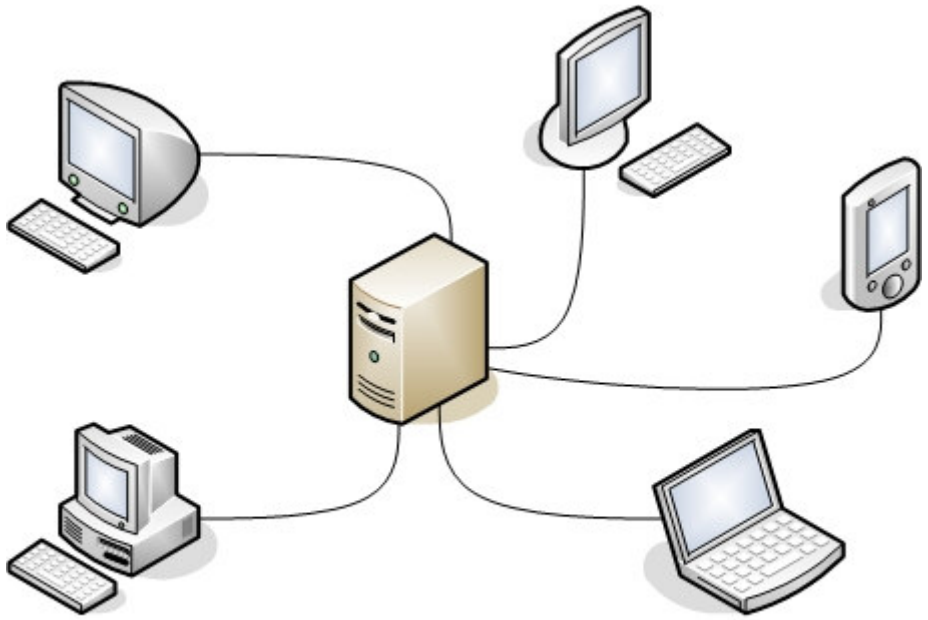

Figura 2.2. Arquitectura de coordinación centralizada.

Cualquiera de las dos soluciones antes presentadas es sencilla de implementar teniendo una entidad distinguida que se encargue de la sincronización. En el caso de querer establecer un orden total global se podrá utilizar un contador en la entidad distinguida, que otorgue turnos a los nodos que los soliciten. Estos turnos serán adjuntados a las operaciones y serán tenidos en cuenta al momento de su ejecución en cada nodo.

Otra alternativa posible estableciendo un orden total, es que cada cliente envíe las operaciones al servidor, este establezca un orden total por arribo y se encargue de distribuir las operaciones a todos los clientes.

Por último, si se opta por el establecimiento de bloqueos, el hecho de tener una entidad distinguida facilita el establecimiento de los mismos. Será el servidor el encargado de permitir o denegar el pedido de modificación de las entidades involucradas.

3. Arquitectura de coordinación distribuida

En un sistema *groupware*, una arquitectura de coordinación distribuida es aquella en la que no son necesarias entidades únicas para su funcionamiento. Las arquitecturas *peer to peer* son un caso particular de las arquitecturas de coordinación distribuida, las cuales se definen como un sistema auto organizado de entidades autónomas e iguales (*peers*), las cuales tienen como objetivo el uso compartido de recursos en un entorno de red, evitando servicios centralizados [Oram]. A diferencia de las arquitecturas centralizadas, se incrementa la disponibilidad de servicios, distribuyéndolos en un gran número de nodos, los cuales, se asume, no fallarán todos a la vez.

En una arquitectura *peer to peer* pura, no existe una entidad distinguida que se encargue de mantener un orden total entre las operaciones que se generan. El hecho de tenerla rompería su pureza. Por lo tanto, el mantenimiento de consistencia entre los estados de los diferentes *peers*, deberá realizarse de forma distribuida.

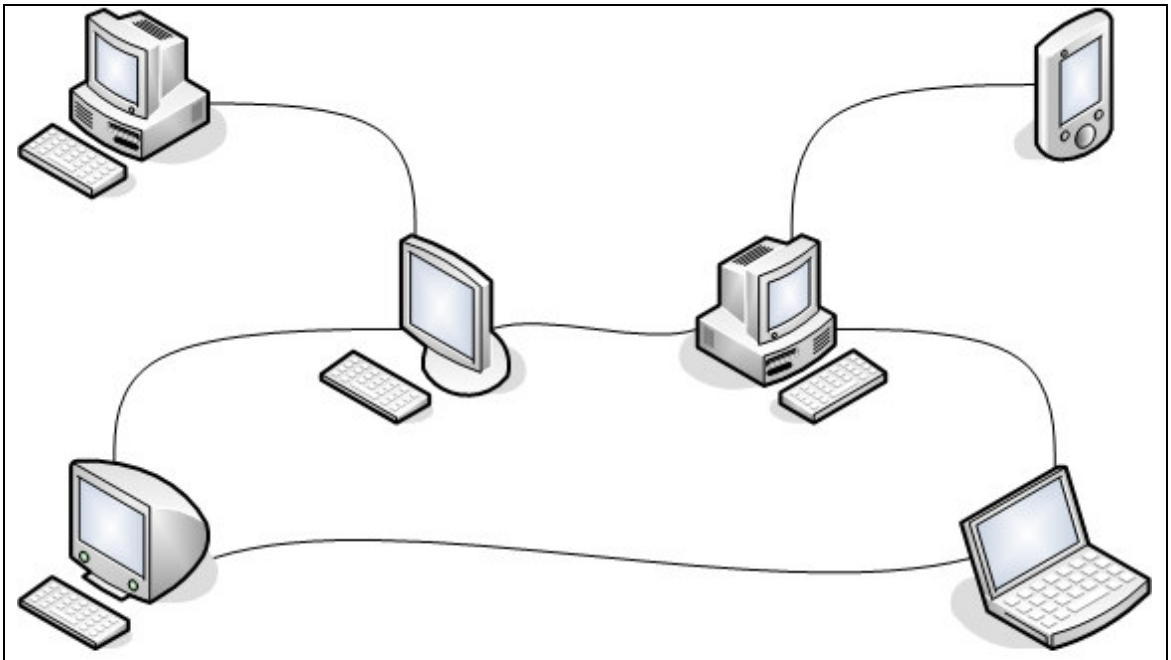

Figura 2.3. Arquitectura de coordinación distribuida.

Si se evalúa la alternativa del establecimiento de bloqueos, se verá que ésta disminuye drásticamente la interactividad entre los participantes. Peor aún, esta disminución será mucho más notoria en una arquitectura *peer to peer*, debido a que se deberán manejar de forma distribuida los bloqueos, lo cual multiplica varias veces el *tiempo de respuesta*.

Una alternativa a estas dos formas de mantener consistencia y la elegida en este trabajo de tesis es *Operational Transformation* y se presentará en el capítulo siguiente.

Capítulo 3

Operational Transformation

Este capítulo tiene el objetivo de describir en detalle la técnica de *Operational Transformation* (*OT*). Así, se comienza describiendo los acontecimientos más importantes en la historia de *OT*. A continuación se enuncian y describen cada uno de los objetivos para los cuales se desarrolló esta técnica. Luego el capítulo se centra en su funcionamiento, describiendo el algoritmo, los conceptos y estructuras de datos que utiliza hasta mostrar los casos en los que falla. Le siguen a esto una serie de definiciones y propiedades que se utilizarán para caracterizar y comparar varios algoritmos diferentes de *OT* a continuación. El capítulo concluye con un análisis de factibilidad de encapsular un algoritmo de *OT* en un *framework* para su posterior reutilización en el desarrollo de aplicaciones de distintos dominios.

1. Historia

Ellis y Gibbs fueron los inventores del algoritmo de *Operational Transformation* (*OT*) en 1989, hace ya más de veinte años. En un *paper* llamado *"Concurrency Control in Groupware Systems"* presentaron el algoritmo *dOPT* [Ellis 1] (por distributed OPerational Transformation), el cual utilizaron en el sistema *Grove*. El algoritmo tenía el propósito de permitir la edición colaborativa de texto, interpretando el objetivo final de los usuarios. Es decir, obtener un resultado convergente en todos los nodos, preservar el orden en que ocurrieron los cambios y preservar la intención de los usuarios en cada acción.

Al poco tiempo de su presentación, se descubrió que en determinadas circunstancias, el algoritmo *dOPT* fallaba. A este escenario se lo llamó "*dOPT puzzle*" [Cormack 1]. En 1996 Ressel propuso un nuevo algoritmo que resolvía el *dOPT puzzle* y lo llamó *adOPTed* [Ressel]. Dos años después, C. Sun desarrolló otro algoritmo, que al igual que el de Ressel, resolvía el *dOPT puzzle* y lo llamó *GOT* [Sun 1].

Más tarde, hace no mucho tiempo, sendos grupos de investigación probaron que ambos algoritmos fallaban en determinados casos y propusieron enmiendas [Imine 1, Imine 2]. En el transcurso de este tiempo se han desarrollado numerosos algoritmos, cuyas características se describirán brevemente más adelante.

2. Objetivos

El algoritmo de *Operational Transformation* tiene como metas el permitir el desarrollo de groupware de tiempo real, distribuido y sin restricciones. El hecho de permitir el desarrollo de groupware de tiempo real implica que el tiempo que se demora en reflejar las acciones locales debe ser extremadamente pequeño, similar al de un sistema monousuario, y el tiempo de respuesta a las acciones remotas debe estar únicamente determinado por la latencia de la red.

Permitir el desarrollo de sistemas distribuidos implica que los diferentes usuarios no estarán todos conectados a una única computadora, ni tampoco a una misma red local. Estos podrán estar distribuidos físicamente alrededor del mundo, siempre y cuando tengan acceso a la red que utiliza el sistema.

Por último, los sistemas no deberían tener restricciones que limiten la interacción entre los usuarios. Es decir, los usuarios podrán editar la información compartida concurrentemente en cualquier momento. Esto derivará en un flujo libre y natural de la información entre los usuarios.

3. Funcionamiento

Los algoritmos de *Operational Transformation* (*OT*) son una alternativa para mantener consistencia en ambientes totalmente descentralizados. Con este método no es necesario el establecimiento de bloqueos ni la ejecución de las operaciones en un orden total.

En los algoritmos de *OT*, los cambios son encapsulados en operaciones. Cada uno de los nodos genera estas operaciones, que son ejecutadas primero localmente y luego propagadas al resto de los pares. Las operaciones que llegan desde otros pares a cada nodo son ejecutadas siguiendo reglas estrictas.

Conceptualmente, cada operación *O* tiene un contexto. Este contexto está compuesto por las operaciones necesarias para llevar el modelo desde su estado inicial hasta el estado en que *O* fue definida. A este último estado se lo denomina *definition context*. El significado de una operación solo puede ser interpretado correctamente en su *definition context*. Al momento de ejecutar *O*, si el contexto actual, denominado *execution context*, es diferente al *definition context*, la ejecución de *O* tendrá que ser retrasada o transformada para poder hacerse en el contexto adecuado.

Los algoritmos de *OT* transforman las operaciones para incluir o excluir los efectos de otras operaciones. Intuitivamente, las transformaciones modifican los atributos de las operaciones antes de su ejecución para que así, incorporen los efectos de las operaciones ejecutadas con anterioridad, pero que no fueron tenidas en cuenta en el momento de su creación.

En la próxima sección se describirán algunos conceptos y estructuras de datos necesarios para el funcionamiento del algoritmo dOPT y posteriormente se detallará el algoritmo.

3.1. Conceptos y estructuras de datos

Para su funcionamiento, el algoritmo dOPT se apoya en una serie de conceptos y estructuras de datos que a continuación se detallan.

Sistema groupware

Ellis y Gibbs definen un sistema groupware como una estructura de la forma:

$G = <S, O>$

Donde:

S es un conjunto de nodos (sites).

O es un conjunto de operadores parametrizados.

Cada nodo (site) está compuesto por un proceso ejecutor (site process), un modelo (site object) y un identificador del nodo (site identifier). El modelo es un objeto de datos pasivo. Como cada usuario estará viendo un modelo distinto, se quiere mantener la consistencia entre estos modelos lo más alta posible (los modelos deberían verse como si fueran el mismo). Los modelos son dependientes del dominio de la aplicación, algunos ejemplos son un documento de texto, una imagen, un plano, etc.

El conjunto O está compuesto por operadores definidos para el modelo, por lo tanto también variarán según el dominio. Por ejemplo para un editor de texto sencillo que tiene un *string* de caracteres como modelo, O estará formado por $\{O_1, O_2\}$ donde:

O_1 = insert[X; P] inserta el carácter X en la posición P.

O_2 = delete[P] elimina el carácter de la posición P.

Las operaciones que pueden aplicarse sobre los modelos son instancias de estos operadores. Un ejemplo de operación es:

$o = O_1[x; 3]$

donde o aplicada al *string* 'abc' da como resultado:

o('abc') = 'abxc'

Matriz de transformación (transformation matrix)

La matriz de transformación es la clave para resolver los conflictos entre operaciones. Ésta contiene las funciones de transformación que, siguiendo reglas definidas para tal fin o acudiendo a prioridades relacionadas con las operaciones, determinarán el resultado de la transformación de una operación con otra.

Dado un sistema groupware G = <S, O>, entonces la matriz de transformación de G, T es una matriz de m x m donde m es la cardinalidad del conjunto O. Cada componente de T es una función que transforma operaciones en otras operaciones. Dadas dos operaciones o_i y o_j, con prioridades p_i y p_j respectivamente y que son instancias de los operadores O_u y O_v, sea:

$o_j' = T_{uv}(o_j, o_i, p_j, p_i)$

$o_i' = T_{vu}(o_i, o_j, p_i, p_j)$

entonces T es tal que lo siguiente se satisface:

$o_j' \circ o_i = o_i' \circ o_j$

siendo ◦ la composición de operaciones y significando que la ejecución sobre dos modelos idénticos de los resultados de la composición a uno y otro lado del igual resultará en modelos idénticos nuevamente.

Vector de estado (state vector)

Sea N la cantidad de nodos en el sistema (se asume que N es constante). Cada nodo tendrá un identificador único que por simplicidad asumiremos que va de 1 a N. El vector de estado del nodo j, es un vector de N posiciones donde la posición i indica cuantas operaciones del nodo i se ejecutaron en el nodo j. Dados dos vectores de estado s_i y s_j se dice que:

$s_i = s_j$ si cada componente de s_i es igual al correspondiente en s_j.

$s_i < s_j$ si cada componente de s_i es menor o igual al correspondiente en s_j y al menos un componente de s_i es menor que el correspondiente en s_j.

$s_i \leq s_j$ si $s_i < s_j$ o $s_i = s_j$.

$s_i > s_j$ si al menos un componente de s_i es mayor que el correspondiente en s_j.

Requerimiento (Request)

Los requerimientos son tuplas de la forma <i, s, o, p> donde i es el identificador del nodo de origen, s es un vector de estado, o es una operación y p es la prioridad asociada a o. Dicho vector de estado corresponde a una copia del vector de estado del nodo origen de la operación al momento de la creación de la operación, es decir constituye el *definition context*.

El vector de estado de los requerimientos se utiliza para asegurar la propiedad de precedencia (Ver capítulo 2). Por ejemplo, en los dos casos mostrados en la figura 3.1 el Nodo 2, usando el vector de estado de r_2, puede determinar que r1 se ha creado con anterioridad a r2 y así demorar la ejecución de este último hasta que r1 haya sido ejecutado.

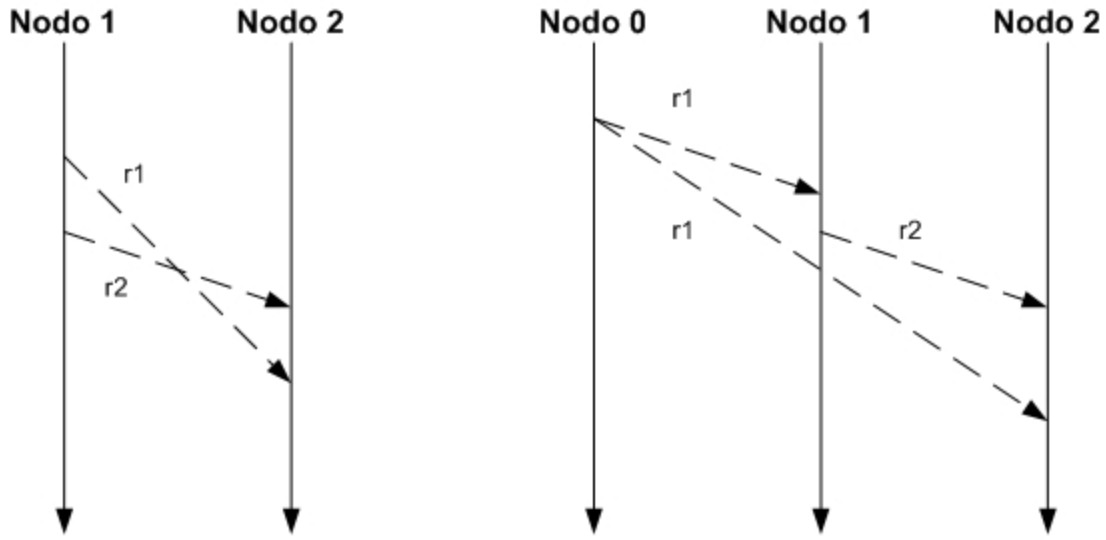

Figura 3.1. Detección de precedencia.

Cola de requerimientos (Request queue)

Los requerimientos que esperan por ser ejecutados en un nodo por el proceso ejecutor son mantenidos en la cola de requerimientos del nodo. Un requerimiento <j, s, o, p> en Q_i (la cola de requerimientos del nodo i) indica que o fue generada por el nodo j mientras estaba en el estado s.

Los requerimientos se agregan a la cola de requerimientos tanto cuando se reciben desde la red como cuando se generan localmente. Los requerimientos se remueven de la cola cuando el proceso ejecutor determina que la operación debe ejecutarse.

A pesar que se utiliza el término *cola* de requerimientos, esto no implica que esta utilice una política *first-in-first-out*.

Bitácora de requerimientos (Request log)

Cada proceso ejecutor mantiene una bitácora de los requerimientos ejecutados en su nodo. Un requerimiento <j, s, o, p> en L_i (la bitácora de requerimientos del nodo i) indica que el nodo i mientras estaba en el estado s ejecutó la operación o (generada por

el nodo j). La bitácora se mantiene ordenada, por lo que es posible encontrar la primera operación ejecutada, la última ó recorrer las operaciones en orden de ejecución.

3.2. El algoritmo dOPT

La siguiente es la especificación del algoritmo dOPT para el proceso ejecutor del nodo i.

Inicialización:

$Q_i \leftarrow$ empty

$L_i \leftarrow$ empty

$S_i \leftarrow <0, 0, \ldots, 0>$

La sección de inicialización, lo único que hace es establecer en vacío la bitácora y la cola de requerimientos e inicializar el vector de estado del nodo.

Generación de operaciones:

Recibir la operación o de la interfaz de usuario.

Calcular la prioridad p de o.

$Q_i \leftarrow Q_i + <i, s_i, o, p>$

Broadcast $<i, s_i, o, p>$ a todos los demás nodos.

Esta sección recibe una operación local, arma un requerimiento, el cual agrega a la cola de requerimientos y hace *broadcast* del requerimiento a todos los demás nodos.

Recibir operaciones:

Recibir $<j, s_j, o_j, p_j>$ de la red.

$Q_i \leftarrow Q_i + <j, s_j, o_j, p_j>$

La sección de recibir operaciones agrega los requerimientos que llegan desde la red a la cola de requerimientos del nodo.

Ejecutar operaciones:

for each <j, s_j, o_j, p_j> ϵ a Q_i donde $s_j \leq s_i$ begin
  $Q_i \leftarrow Q_i$ - <j, s_j, o_j, p_j>
  if $s_j < s_i$
    <k, s_k, o_k, p_k> ← tupla más reciente en L_i donde $s_k \leq s_j$ (o Ø si no existe)
    do while <k, s_k, o_k, p_k> ≠ Ø and $o_j \neq$ Ø
      if $s_j[k] \leq s_k[k]$
        sea u el índice de o_j (o_j es una instancia de O_u)
        sea v el índice de o_k (o_k es una instancia de O_v)
        $o_j \leftarrow T_{uv}(o_j, o_k, p_j, p_k)$
      fi
      <k, s_k, o_k, p_k> ← siguiente tupla en L_i (o Ø si no hay)
    od
  fi
  ejecutar la operación o_j en el modelo del nodo i.
  $L_i \leftarrow L_i$ + <j, s_j, o_j, p_j>
  $s_i \leftarrow s_i$ con el elemento j ésimo incrementado en 1.
end

La última sección y más compleja es la que se encarga de ejecutar las operaciones. Primero se examina la cola por requerimientos rj = <j, s_j, o_j, p_j> que deban ser ejecutados. Cuándo r_j puede ser ejecutado se determina comparando s_j con el vector de estado del nodo i, s_i. Existen tres posibilidades:

Primer caso: $s_j > s_i$

El requerimiento no puede ejecutarse inmediatamente (el nodo j ha ejecutado operaciones que todavía no han sido ejecutadas en el nodo i), con lo cual se deja en la cola.

Segundo caso: $s_j = s_i$

Cuando los dos estados son iguales, o_j se ejecuta inmediatamente sin ninguna transformación.

Tercer caso: $s_j < s_i$

En este caso el requerimiento también puede ejecutarse. Sin embargo, como r_j se refiere a un estado anterior que el estado actual del nodo i, o_j debe ser transformada. Se consulta la bitácora por los requerimientos que no fueron tenidos en cuenta por el nodo j (requerimientos que el nodo i ya ejecutó pero que no fueron ejecutados en el nodo j antes de la generación de r_j). De esta forma, cada uno de estos requerimientos de la bitácora se utilizará uno detrás del otro para transformar o_j, aplicando la función de transformación que correspondiere a cada caso. Esto significa que se transformará o_j con el primer requerimiento, su resultado se transformará con el segundo requerimiento y así con todos los que sean necesarios.

Tanto el segundo como el tercer caso terminan en la ejecución de o_j. Cuando esto ocurre, la operación se aplica sobre el modelo, se agrega a la bitácora y se incrementa el vector de estado del nodo.

3.3. Las funciones de transformación

La matriz de transformación es una matriz de m x m, donde m es la cardinalidad del conjunto de operadores. Recordemos que los operadores, representan las clases de operaciones que pueden ejecutarse en la aplicación groupware. Para el caso del editor de texto colaborativo, se dispone de dos operadores, insert[X_i, P_i] y delete[P_i]. Por lo tanto la matriz de transformación tendrá cuatro funciones de transformación como muestra la figura.

Matriz de Trans.	i_j = insert[X_j, P_j]	d_j = delete[P_j]
i_i = insert[X_i, P_i]	$T_{11}(i_i, i_j, p_i, p_j)$	$T_{12}(i_i, d_j, p_i, p_j)$
d_i = delete[P_i]	$T_{21}(d_i, i_j, p_i, p_j)$	$T_{22}(d_i, d_j, p_i, p_j)$

Figura 3.2. Matriz de Transformación.

Es importante notar que las P (mayúsculas) indican posiciones y las p (minúculas) indican prioridades.

Las funciones de transformación solo se aplicarán a operaciones provenientes de diferentes nodos, ya que las operaciones provenientes de un mismo nodo se procesan en orden. Esto hace que las prioridades de las operaciones a transformar sean siempre diferentes.

A continuación se detalla el algoritmo de cada una de las funciones de transformación de la matriz.

T_{11}(insert[X_i, P_i], insert[X_j, P_j], p_i, p_j) = o_i' where
    if ($P_i < P_j$)
        o_i' = insert[X_i, P_i]
    else if ($P_i > P_j$)
        o_i' = insert[X_i, $P_i + 1$]
    else /* $P_i = P_j$ */
        if ($X_i = X_j$)
            o_i' = noOp
        else if ($p_i > p_j$)
            o_i' = insert[X_i, $P_i + 1$]
        else /* $p_i < p_j$ */
            o_i' = insert[X_i, P_i]

Existen dos casos interesantes en el algoritmo de T_{11}. El primero, cuando los argumentos de las operaciones son iguales, el resultado de la transformación es noOp, que significa la operación vacía. Este caso representa la ocasión en que desde dos nodos diferentes, se propaga al mismo tiempo la misma operación. Es decir, que ambos usuarios tienen intenciones coincidentes. Entonces lo que se debe hacer en cada nodo es obviar una de las dos operaciones, que es lo que se hace devolviendo una noOp.

El segundo caso interesante es cuando sólo las posiciones son coincidentes, es decir que dos usuarios quieren insertar en la misma posición dos caracteres diferentes. En este caso el conflicto se resuelve de acuerdo a la prioridad de cada operación, otorgándole la posición requerida a una u otra operación según cual tenga mayor prioridad.

Es un tema interesante el criterio de establecimiento e interpretación de prioridades ya que estas deben ser siempre diferentes, pero es deseable que no existan privilegios entre los usuarios. Es decir que no sean siempre las operaciones provenientes de un usuario las que tengan mayor prioridad.

Las otras funciones de transformación son más sencillas e intuitivas:

T_{12}(insert[X_i, P_i], delete[P_j], p_i, p_j) = o_i' where
    if ($P_i < P_j$)
        o_i' = insert[X_i, P_i]
    else
        o_i' = insert[X_i, P_i - 1]

T_{21}(delete[P_i], insert[X_j, P_j], p_i, p_j) = o_i' where
    if ($P_i < P_j$)
        o_i' = delete[P_i]
    else
        o_i' = delete[P_i + 1]

T_{21}(delete[P_i], delete[P_j], p_i, p_j) = o_i' where
    if ($P_i < P_j$)
        o_i' = delete[P_i]
    else if ($P_i > P_j$)
        o_i' = delete[P_i - 1]
    else /* $P_i = P_j$ */
        o_i = noOp

3.4. dOPT puzzle

A seis años del invento de Ellis y Gibbs, en Agosto de 1995, Cormack comprobó que el algoritmo dOPT tenía un error [Cormack 1]. Para demostrarlo proveyó el siguiente contraejemplo. Consideremos como modelo un string cuyo estado inicial es "ABCDEFG". El nodo 1 elimina la letra A, mientras que concurrentemente el nodo 2 elimina la letra A y luego la E. Así, el nodo 1 y el 2 obtienen (en forma temporal) el string "BCDEFG" y "BCDFG" respectivamente y propagan, en total, tres requerimientos:

Desde el nodo 1 al 2:

<1, (0,0), delete[1], p1>

Desde el nodo 2 al 1:

<2, (0,0), delete[1], p2>

<2, (0,1), delete[4], p3>

Los valores de las prioridades p1, p2 y p3 no tienen importancia.

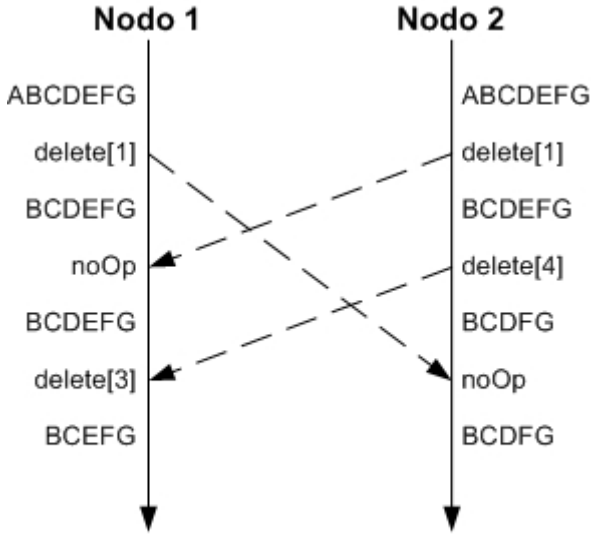

Figura 3.3. dOPT Puzzle.

Cuando el requerimiento destinado al nodo 2 arriba, su operación se transforma en una operación vacía (noOp), obteniendo en dicho nodo, el string "BCDFG". Al mismo tiempo, cuando el primer requerimiento destinado al nodo 1 llega, su operación también se transforma en una operación vacía, dejando el estado de este nodo en "BCDEFG". Al arribar el segundo requerimiento al nodo 1, se transforma su operación en delete[3], lo

cual deja el string en el nodo 1 en "BCEFG". Dado que no hay requerimientos en tránsito y que los estados de los nodos son diferentes, se concluye que el algoritmo es incorrecto.

Esto sucede porque en el nodo 1 se transforma la segunda operación recibida con la operación generada localmente, sin tener en cuenta que la segunda operación recibida ya contiene dicha modificación.

4. Definiciones y propiedades

A continuación se detallan las definiciones y propiedades más importantes, comunes a todos los algoritmos de OT que se utilizarán para caracterizar los algoritmos de OT más conocidos.

4.1. Independencia de operaciones [Lamport]

Dadas dos operaciones O_a y O_b:

- O_b ***depende*** de O_a si y solo si $O_a \rightarrow O_b$.
- O_a y O_b son ***independientes*** (o concurrentes), expresado como $O_a \parallel O_b$ si y solo si no se cumple que $O_a \rightarrow O_b$ ni $O_b \rightarrow O_a$.

4.2. Situaciones de inconsistencia [Sun 1]

Para darnos una idea de los desafíos que se enfrentan en este dominio, consideremos el escenario de la figura 3.4. En ella se muestra una sesión en la que colaboran tres usuarios desde 3 diferentes nodos. Supongamos que las operaciones se ejecutan en la réplica local del modelo ni bien son generadas y luego se las envía al resto de los nodos para que se ejecuten ni bien llegan, en su forma original (sin ser transformadas). Tres diferentes situaciones de inconsistencia pueden identificarse:

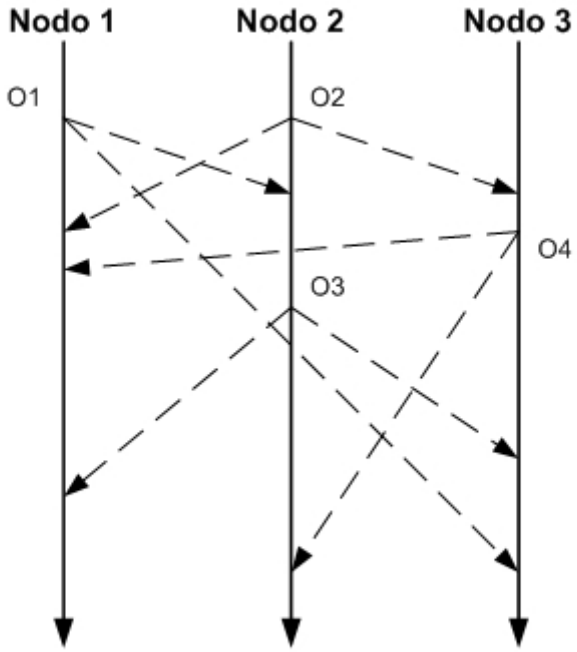

Figura 3.4. Escenario de colaboración en tiempo real.

Divergencia

Las operaciones pueden llegar a los distintos nodos en distinto orden, pudiéndose generar diferentes resultados. Como se muestra en la figura 3.4, el orden de ejecución en el nodo 1 es: O_1, O_2, O_4 y O_3, en el nodo 2: O_2, O_1, O_3 y O_4 y en el nodo 3: O_2, O_4, O_3 y O_1. A menos que estas operaciones tengan la propiedad de ser conmutativas, lo cual no ocurre generalmente, el resultado en cada nodo divergirá.

Violación de causalidad

Debido a que el tiempo de latencia de la red es no determinístico, las operaciones pueden llegar y ser ejecutadas en los nodos en un orden que no es el mismo orden en el que fueron generadas. Como se muestra en la figura 3.4, la operación O_3 fue generada en el nodo 2 luego del arribo de O_1, es decir que el efecto que causó O_1 en el documento compartido fue tenido en cuenta por el usuario del nodo 2 al momento de generar O_3. Por lo tanto O_3 ***depende*** de O_1. Sin embargo, en el nodo 3, O_3 llega y se ejecuta antes que O_1, lo cual en determinados casos puede provocar errores. Por ejemplo, supongamos que la operación O_1 crea cierta entidad A con determinadas características y que la operación O_3 lo que hace es modificar alguna de estas características. Así, O_3 no podrá ejecutarse antes que O_1 pues no podrá hacer referencia a la entidad A.

Violación de intensión

Debido a que las operaciones se generan de manera concurrente, el efecto real de una operación en el momento de su ejecución puede diferir del efecto deseado en el momento de su creación. Como se muestra en la figura 3.4, la operación O_1 se genera en el nodo 1 sin ningún conocimiento de la operación O_2 generada en el nodo 2, por lo cual O_1 será ***independiente*** de O_2 y viceversa. En el nodo 1, O_2 se ejecuta en un estado del documento compartido que ha cambiado debido a la ejecución de O_1. Por lo tanto la ejecución de O_2 no ocurrirá sobre el mismo estado del documento en el que fue generada. Así, puede suceder que el efecto que realiza O_2 no sea el mismo que el efecto que se buscaba cuando se creó. Es decir, se está violando la intensión original de O_2. Por ejemplo, supongamos que el documento compartido inicialmente contiene el string "ABCDE", que O_1 = insert["12", 1] con la intensión de insertar el string "12" en la posición 1 (es decir entre la A y la B) y que O_2 = delete[2] con la intensión de eliminar la letra en la posición 2 (es decir, la C). Luego de la ejecución de estas dos operaciones en todos los nodos, el resultado esperado (y el que conserva la intensión de las operaciones) es "A12BDE". Sin embargo el resultado en el nodo 1 luego de la ejecución de O_1 seguida de O_2 es "A1BCDE", el cual viola la intensión de ambas operaciones.

Es importante destacar que estos tres problemas de inconsistencia son independientes entre sí. Es decir que la ocurrencia de alguno/s de ellos no implica la ocurrencia de el/los otro/s.

4.3. Transformaciones [ACE]

La ***Transformación de Inclusión (IT)*** transforma una operación O_a con otra operación O_b de tal forma que la operación resultante O_a' incluya el impacto de las modificaciones de O_b.

La ***Transformación de Exclusión (ET)*** transforma una operación O_a con otra operación O_b de tal forma que la operación resultante O_a' excluya el impacto de las modificaciones de O_b.

4.4. Propiedades de las funciones de transformación [Ressel]

Se ha demostrado que las funciones de transformación deben cumplir con las siguientes dos propiedades:
Sea O_a' el resultado de la transformación de O_a con O_b y sea O_b' el resultado de transformar O_b con O_a,

TP1 establece que el efecto de ejecutar O_a seguida de O_b' es el mismo que ejecutar O_b seguida de O_a'.

TP1 es suficiente y necesaria para asegurar que un sistema groupware basado en OT con dos usuarios es correcto. La propiedad TP2 asegura la corrección para sistemas de más de dos usuarios.

TP2 establece que la operación resultante de transformar O_c con O_a y su resultado con O_b' es igual a la operación resultante de transformar O_c con O_b y su resultado con O_a'.

4.5. Tipos de comunicación [ACE]

De acuerdo a la forma de comunicación entre los nodos de un sistema groupware, este puede ser clasificado en dos grandes tipos, los que utilizan comunicación ***unicast*** o los prefieren comunicación ***multicast***. En el primer grupo, las aplicaciones que utilizan comunicación ***unicast***, los nodos solo se comunican con un servidor central y es este el encargado de reenviar la información requerida al resto de los nodos. Por otra parte, en los sistemas que adoptan comunicación ***multicast***, los nodos se comunican directamente entre ellos sin la necesidad de un servidor central.

5. Algoritmos [ACE]

A continuación se presenta una breve descripción y un resumen de las características más importantes de los algoritmos de Operational Transformation más conocidos.

5.1. dOPT [Ellis 1]

Fue el primer algoritmo de Operational Transformation en inventarse, sus autores Ellis y Gibbs lo hicieron en 1989. Su nombre surge de Distributed Operational Transformation. Al tiempo de publicarse, se descubrió que tiene un error, al cual llamaron dOPT Puzzle. Este error ocurre cuando un nodo envía más de una operación concurrente a otra operación de otro nodo.

Principales caracterísiticas:

- El algoritmo es incorrecto.
- Utiliza vectores de estado para determinar relaciones de orden causal (operaciones dependientes o independientes).
- Utiliza un historial lineal, llamado Request Log.
- Comunicación multicast.

5.2. CCU [Cormack 2]

CCU (Calculus for Concurrent Update) deriva del algoritmo dOPT y fue desarrollado por Cormack en 1995 en el momento que descubrió el error en dOPT. El algoritmo especifica un modelo concurrente basado en un modelo secuencial enriquecido con todos los pares posibles de operaciones (insert, delete). A pesar que existe una descripción del algoritmo, existen algunos detalles de implementación sin precisar. Curiosamente no se han encontrado referencias a este algoritmo en otros trabajos de investigación.

Principales características:

- Su corrección nunca fue confirmada por otros autores.
- Utiliza vectores de estado para determinar relaciones de orden causal.
- Comunicación unicast.

5.3. Jupiter [Nichols]

Júpiter es un mundo virtual multimedia y multiusuario enfocado en sesiones de colaboración largas. En particular, este software permite compartir documentos y herramientas y opcionalmente comunicaciones de audio y video.
El algoritmo de Júpiter deriva del algoritmo dOPT pero resuelve el dOPT puzzle. Para ello utiliza un estado bidimensional en vez de una bitácora unidimensional para guardar los requerimientos.

Principales características:

- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza múltiples protocolos de sincronización de dos vías para crear un protocolo de n vías.
- Libre de TP2.
- Algoritmo simple.
- Estado bidimensional.
- Comunicación unicast.

5.4. NetEdit [Zafer]

NetEdit es un editor de texto colaborativo que utiliza una arquitectura replicada con procesamiento y datos distribuidos entre los nodos. Su algoritmo de consistencia está basado en el algoritmo de Júpiter.

Principales características:

- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza múltiples protocolos de sincronización de dos vías para crear un protocolo de n vías.
- Libre de TP2.
- Comunicación unicast.

5.5. adOPTed [Ressel]

adOPTed es una versión mejorada del algoritmo dOPT. El corazón de este, es un modelo multidimensional de interacción concurrente, el cual permite comunicación directa de los n nodos. El algoritmo es conceptualmente similar al de Júpiter, pero extiende la comunicación en 2 vías de este último por una comunicación multivía. Al igual que Júpiter utiliza un espacio para almacenar los requerimientos pero en el caso de adOPTed este espacio es multidimensional.

Principales características:

- Está probado que es correcto si sus funciones de transformación cumplen TP2 [Lushman].
- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza solo IT.
- Es posible extenderlo para operaciones de *undo* [Gunzenhäuser].
- Modelo de interacción multidimensional.
- Comunicación multicast.

5.6. GOT [Sun 1]

El algoritmo GOT (Generis Operation Transformation) fue el primer trabajo en definir las tres propiedades de consistencia: convergencia, preservación de la causalidad y preservación de la intensión.

Principales características:

- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza IT y ET.
- Establece un orden global utilizando un esquema de undo/do/redo para lograr convergencia.
- Libre de TP1 y TP2, debido al uso de un orden global.
- Algoritmo complejo (necesita muchas transformaciones).
- Comunicación multicast.

5.7. GOTO [Sun 2]

GOTO (GOT Optimized) es una optimización del algoritmo GOT, la cual hace que las IT y ET deban satisfacer TP1 y TP2. Esta optimización resulta en un algoritmo más simple, que utiliza menos IT y ET en comparación a GOT.

Principales características:

- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza IT y ET.
- Las funciones IT y ET deben satisfacer TP1 y TP2.
- Comunicación multicast.

5.8. SOCT2 [Suleiman 1 y 2]

El algoritmo SOCT2 puede ser utilizado en entornos centralizados o distribuidos.

Entornos centralizados

En un entorno centralizado, un sistema colaborativo que utiliza SOCT2, está compuesto por un conjunto de nodos usuarios conectados a un nodo servidor. El nodo servidor es el responsable del mantenimiento de los objetos compartidos por los usuarios. Cada uno de estos objetos puede ser modificado por los usuarios mediante operaciones específicas que deben enviarse al servidor para ejecutarse. Los estados por los que pasa una operación son: generación, envío, recepción y ejecución. Luego de la ejecución de una operación en el servidor, se supone que los cambios que realiza dicha operación sobre los objetos, pueden ser vistos inmediatamente por los usuarios.

Cuando se envía una operación al servidor, este le aplica una transformación de inclusión con el resto de las operaciones concurrentes a ella (si las hubiera) obtenidas del historial de ejecución. Para determinar si dos operaciones son concurrentes se utilizan marcas de tiempo (*timestamps*). Luego, se ejecuta la operación resultante en el servidor.

Entornos distribuidos

La gran diferencia con el entorno centralizado es que los objetos compartidos están distribuidos y replicados, lo cual hace que el algoritmo sea más complejo.

En este entorno, cada nodo participante es un nodo usuario y servidor a la vez. Los nodos participantes contienen una copia de los objetos compartidos, por lo que cada operación generada deberá ejecutarse en todos los nodos participantes. Así, las operaciones deberán enviarse desde el nodo que la generó hacia todo el resto de los nodos. Los estados por los que pasa una operación en este entorno son: generación, difusión, recepción y ejecución.

El hecho de utilizar un entorno distribuido hace que se necesiten agregar ciertos parámetros en las operaciones y utilizar transformaciones de exclusión, lo cual vuelve al algoritmo más complejo.

Principales características:

- En entornos centralizados solo usa transformaciones de inclusión, mientras que en entornos distribuidos utiliza transformaciones de inclusión y exclusión (sin deshacer y rehacer las operaciones).
- En entornos centralizados utiliza *timestamps* mientras que en entornos distribuidos utiliza vectores de estado para establecer relaciones de causalidad.
- Utiliza un historial de operaciones lineal.
- Los agregados en las operaciones para entornos distribuidos las hace difíciles de manejar e ineficientes.
- No se provee un mecanismo de undo.
- Está probado que las funciones de transformación son incorrectas [Imine 2].
- Comunicación unicast en entorno centralizado.
- Comunicación multicast en entorno distribuido.

5.9. SOCT3 [Vidot]

Verificar que un conjunto de funciones de transformación cumple con TP2 no es una tarea sencilla, se deben verificar alrededor de cien casos dependiendo de los parámetros de estas. Es por esto que los inventores de SOCT3 y SOCT4 decidieron ir en otra dirección, haciendo que no sea necesario satisfacer TP2.

Ellos propusieron la implementación de un orden de serialización global sobre las operaciones. Este orden es establecido por un secuenciador, el cual entrega números Naturales crecientes llamados *timestamps*.

Principales características:

- Utiliza vectores de estado para determinar relaciones de orden causal.
- Utiliza un historial de operaciones lineal.
- Utiliza un orden global para no tener que cumplir con TP2.
- No se conoce algoritmo de *undo*.
- Comunicación *multicast*.

5.10. SOCT4 [Vidot]

SOCT4 es similar a SOCT3, pero a diferencia de este, en SOCT4 las transformaciones no se efectúan en el receptor de las operaciones sino en el emisor. El procedimiento es así, al generarse una operación localmente, esta es ejecutada inmediatamente, luego se acude al secuenciador por un *timestamp* para la operación. No se hace *broadcast* de la operación sino hasta haber recibido y ejecutado todas las operaciones con *timestamp* inferior al obtenido del secuenciador para dicha operación.

Principales características:

- No utiliza vectores de estado.
- Utiliza un orden global para no tener que cumplir con TP2.
- No necesita ET.
- No se conoce algoritmo de undo.
- Comunicación multicast.

6. Comparación de los algoritmos

A continuación se presentará una tabla comparativa de los algoritmos anteriormente descriptos con los siguientes datos:

- Año: especifica el año de publicación.
- Corrección: especifica si el algoritmo o alguna de sus partes es correcta.
- Comunicación: especifica el tipo de comunicación utilizado.
- Información disponible: especifica si existe información suficiente para su implementación.
- *Undo*: especifica si se encuentra desarrollada la funcionalidad de deshacer (*undo*) para el algoritmo.

	dOPT	CCU	Jupiter	adOPTed	GOT	GOTO	SOCT2	SOCT3	SOCT4
Año	1989	1995	1995	1996	1998	1998	1997	2000	2000
Corrección	No	Indefinida	Si	Algoritmo de control	Si	Algoritmo de control	No	Si	Si
Comunicación	Multicast	Unicast	Unicast	Multicast	Multicast	Multicast	Unicast y Multicast	Multicast	Multicast
Información disponible	Si	No	Si	Si	Si	Si	Si	No	No
Undo	No	No	No	Si	No	Si	No	No	No

6.1. Criterios de selección

Para la selección del algoritmo de Operational Transformation a utilizar se tendrán en cuenta los siguientes criterios:

- **Corrección**: es el criterio más importante, ya que si un algoritmo no es correcto, no podrá ser implementado.
- **Tipo de comunicación**: se desea que el tipo de comunicación sea *multicast* para que así el flujo de información no dependa de una única entidad en el sistema.
- **Información disponible**: se requiere que exista suficiente información para la implementación del algoritmo.
- **Posibilidad de deshacer** (*undo*): es una funcionalidad deseada pero no determinante.

6.2. Selección del algoritmo

Los algoritmos que superan el criterio de **corrección** son:

- Jupiter
- adOPTed: aunque está probado que sus funciones de transformación no son correctas, también está probado que si se las sustituye por las presentadas en [Imine 2], el algoritmo es correcto [Lushman].
- GOT
- SOCT3
- SOCT4

Si de este grupo de algoritmos tomamos los que cumple el criterio de **tipo de comunicación**, nos quedan:

- adOPTed
- GOT
- SOCT3
- SOCT4

Aplicando el criterio de **información disponible** descartamos SOCT3 y 4 ya que no se dispone de la información necesaria para la implementación de los secuenciadores. Así los algoritmos que superan además este criterio son:

- adOPTed
- GOT

Por último, deseamos que el algoritmo implementado tenga la posibilidad de ser extendido para soportar deshacer operaciones (*undo*), con lo cual el algoritmo de Operational Transformation elegido para implementar es:

- **adOPTed**

7. Reutilización de algoritmos de Operational Transformation

Cuando se construyen sistemas *groupware peer to peer*, en los cuales pueden surgir conflictos de consistencia, se tiende a desarrollarlos mezclando la lógica del negocio junto con la lógica de preservación de la consistencia. Esto sucede por ejemplo cuando se incluye información de las operaciones precedentes dentro de las abstracciones específicas del dominio o cuando el control de bloqueos está desperdigado por gran parte de la aplicación. Esta no es una práctica recomendada, ya que dificulta el diseño, la implementación y la depuración del sistema y reduce al mínimo la posibilidad de reutilización de parte del mismo.

Ahora, si el sistema se desarrolla separando la lógica del negocio de la de preservación de la consistencia, además de requerir un proceso de desarrollo más sencillo, se obtendrá una herramienta con partes reutilizables y en consecuencia más fácil de mantener.

Como se ha justificado anteriormente, la estrategia para la preservación de la consistencia seleccionada en este trabajo será la de un algoritmo de *Operational Transformation*. Los algoritmos de OT no son sencillos de entender, implementar ni depurar. Una medida de esta complejidad puede percibirse si se considera que los errores en varios trabajos publicados, fueron descubiertos bastante después de su divulgación [Ellis 1, Ressel, Suleiman 1 y 2]. Con lo cual, no se desea que cada vez que se encara el desarrollo de un nuevo sistema basado en OT se deban diseñar, implementar y depurar partes del mismo que podrían haberse reutilizado.

En la próxima sección, se mostrará que aunque se quieran desarrollar sistemas *groupware peer to peer* pertenecientes a diferentes dominios, utilizando un algoritmo de OT la mayor parte de la lógica de preservación de la consistencia es idéntica. Esto genera un contexto ideal para la definición de una capa de manejo de consistencia que provea el comportamiento común y permita configurar los aspectos específicos del dominio.

7.1. Factibilidad

Básicamente, los algoritmos de OT se componen de tres partes, las operaciones, los transformadores o funciones de transformación y un algoritmo de integración. Las operaciones no son más que comandos específicos de la aplicación en cuestión que modelan acciones. Por ejemplo, en un rompecabezas colaborativo, éstas podrían ser el mover o encastrar piezas.

Los transformadores o funciones de transformación son las entidades responsables de modificar las operaciones a ejecutar si estas se encuentran en conflicto con operaciones ejecutadas previamente. Como se explicó anteriormente en este capítulo, las funciones de transformación adaptan las operaciones para hacerlas ejecutables, si su *definition context* difiere del *execution context*.

Por último el algoritmo de integración es el que está a cargo de la coordinación del proceso preservación de la consistencia. Así, es el encargado de recibir las operaciones, decidir cuando ejecutarlas, determinar si deben ser transformadas antes de ejecutarse, ejecutarlas y propagarlas al resto de los nodos.
Si analizamos cada una de estas tres partes con respecto a su posible reutilización veremos que las operaciones están altamente ligadas al dominio de la aplicación. Por ejemplo, en un editor de texto colaborativo, las operaciones serán *insertar carácter* y *eliminar carácter*, mientras que como se mencionó anteriormente, en un rompecabezas colaborativo estas serán *mover pieza* y *encastrar pieza*. Este comportamiento específico no puede ser reutilizado entre dominios. Sin embargo el diseño general de las operaciones y una implementación base si pueden ser reutilizados, por ejemplo, siguiendo el patrón *Command* [Gamma].

Examinando las funciones de transformación, nos daremos cuenta que las modificaciones que estas hacen sobre las operaciones, no son más que modificaciones sobre los atributos que poseen las operaciones. Con lo cual, como las operaciones dependen del dominio de la aplicación, las funciones de transformación también lo harán. Así, las funciones de transformación no podrán ser reutilizadas entre dominios, pero al igual que pasa con las operaciones, el principio general de diseño de estas, sí podrá ser reutilizado.

Por último, si exploramos el algoritmo de integración nos daremos cuenta que este si puede ser reutilizado completamente, ya que:

- La recepción y el envío de operaciones puede ser tratado como una capa independiente abstraída del dominio de la aplicación.
- Para decidir el momento de ejecución de las operaciones y si estas deben ser transformadas para ejecutarse, el algoritmo de OT sólo necesita comparar su *definition context* con el *execution context*, lo cual no depende del dominio de la aplicación.
- La ejecución de las operaciones puede ser generalizada utilizando el patrón *Command* [Gamma].

Por lo tanto, como gran parte del comportamiento de un algoritmo de OT es común entre dominios, concluimos que es posible proveer una implementación reusable de éste que pueda ser especializada para el desarrollo de aplicaciones de diferentes dominios.

7.2. Los frameworks como herramientas de reutilización

Los *frameworks* son una técnica de reuso orientada a objetos. Estos comparten una gran cantidad de técnicas de reuso en general y particularmente las de orientación a objetos. La definición de qué es un *framework* no está consensuada. Una definición frecuente es "un *framework* es diseño reusable de todo o parte de un sistema, representado por un conjunto de clases abstractas y la forma en que sus instancias interactúan" [Johnson]. Mientras que otra definición habitual es "un *framework* es el esqueleto de una aplicación que puede ser configurado por un desarrollador de aplicaciones" o "un *framework* es una aplicación reusable semi-completa que puede ser especializada para producir aplicaciones específicas". Estas definiciones no son excluyentes entre sí, ya que la primera describe la estructura de un *framework* mientras que las dos últimas describen su propósito [Fayad].

Construcción y utilización de un *framework*

Un *framework*, además describe cómo un sistema se descompone en un conjunto de objetos que interactúan. Algunas veces el sistema es una aplicación entera, mientras que otras, es sólo un subsistema. El *framework* describe los objetos que componen el sistema o subsistema especificando sus interfaces y cómo estos objetos interactúan

definiendo el flujo de control entre ellos. En síntesis, un *framework* especifica la forma en que se corresponden las responsabilidades y los objetos de un sistema [Johnson, Wirfs-Brock].

La parte más importante de un *framework* es la forma en que se divide un sistema en sus componentes [Deutsch]. Los *frameworks* también permiten reutilizar implementación, pero esto es menos importante que el reuso de las interfaces del sistema y la forma en que la funcionalidad se divide entre los componentes. Este diseño de alto nivel es el principal componente intelectual de una aplicación y los *frameworks* son un medio para su reuso.

Actualmente, los *frameworks* se implementan con lenguajes orientados a objetos. En ellos, determinados objetos son definidos por clases abstractas. Las clases abstractas son clases que no poseen instancias, con lo cual sólo son usadas como superclases [Wirfs-Brock]. Una clase abstracta usualmente tiene, al menos, una operación abstracta, cuya implementación es responsabilidad de las subclases de esta. Dado que las clases abstractas no tienen instancias, estas no serán usadas como *templates* para la creación de objetos, sino como *templates* para la creación de subclases. Las clases abstractas se utilizan en los *frameworks* para diseñar los componentes, ya que definen una interfaz de los componentes y proveen un esqueleto que puede ser extendido para la implementación de los componentes. A las clases abstractas que permiten que el usuario del *framework* las subclasifique se las llama *hot spots*.

En los lenguajes que permitan la definición de interfaces, los *frameworks* pueden ser definidos en términos de éstas. Sin embargo, de esta forma, sólo se podrá especificar la parte estática de una interfaz y no el modelo de colaboración de los objetos. Por ello, es común en estos lenguajes utilizar ambas, una interfaz y una clase abstracta, para la definición de un componente.

Además de proveer una interfaz, las clases abstractas proveen parte de la implementación a sus subclases. Por ejemplo un *template method* [Gamma] define el esqueleto de un algoritmo en una clase abstracta, delegando algunos pasos a sus subclases. Cada paso se define como un método que puede ser redefinido por las subclases, así las subclases pueden redefinir pasos individuales del algoritmo sin

redefinir su estructura. La clase abstracta puede dejar los pasos individuales sin implementar (definidos como métodos abstractos) o bien proveer una implementación por defecto (llamados *hook methods*) [Pree]. Una clase concreta deberá definir todos los métodos abstractos de su superclase abstracta y podrá definir los *hook methods* que requiera.

Una de las características más importantes de los *frameworks* es la inversión de control. Tradicionalmente, un desarrollador reutiliza componentes de una librería escribiendo un programa que invoca los componentes de ésta cuando lo necesita. Así, el programador decide cuando llamar al componente y es el responsable de la estructura general y del flujo de control del programa. Por el contrario, en un *framework*, el programa es reutilizado y el desarrollador decide qué componentes conectar a éste para completarlo. Estos componentes que se conectan pueden ser provistos por el *framework* o desarrollados por el propio programador. Así, el código desarrollado por el programador es invocado por el código del *framework*. Con lo cual, el *framework*, es el que determina la estructura general y el flujo de control del programa.

Principales beneficios de los *frameworks*

Los beneficios principales que provee la utilización de *frameworks* son:

- **Modularidad**: Los *frameworks* realzan la modularidad encapsulando los detalles de implementación detrás de interfaces estables. La modularidad de un *framework* ayuda a incrementar la calidad de un sistema haciendo que los cambios de diseño o implementación impacten sólo localmente. Esta *localización* de los cambios reduce el esfuerzo requerido para entender y mantener sistemas existentes.

- **Reusabilidad**: las interfaces estables provistas por los *frameworks* acrecientan la reusabilidad ya que permiten definir componentes genéricos que pueden ser reusados para la creación de aplicaciones nuevas. La capacidad de reutilización que proveen los *frameworks* permite encapsular el conocimiento del dominio y el esfuerzo inicial de desarrolladores experimentados, con el objetivo de evitar recrear y revalidar soluciones comunes a requerimientos y desafíos de diseño recurrentes.

- **Extensibilidad**: Los *frameworks* amplifican la extensibilidad proveyendo *hook methods* explícitos que permiten a las aplicaciones extender sus interfaces. Los *hook methods* desacoplan sistemáticamente las interfaces y los comportamientos de un dominio de aplicación, de las variaciones requeridas por las instanciaciones de una aplicación en un contexto particular. La capacidad de extensión de un *framework* es esencial para asegurar que los nuevos servicios y características se desarrollen a tiempo.

- **Inversión de control**: La arquitectura en tiempo de ejecución de un *framework* está caracterizada por la inversión de control. Esta arquitectura permite que los pasos de procesamiento de una aplicación sean especializados por objetos manejadores de eventos que serán invocados por el mecanismo reactivo del *framework*. Cuando ocurren determinados eventos, el *framework* reacciona invocando *hook methods* sobre manejadores pre-registrados, los cuales llevan a cabo procesamiento específico sobre los eventos. La inversión de control permite que sea el *framework*, en vez de la aplicación, el que determina cual conjunto de métodos específicos de la aplicación invocar en respuesta a eventos externos.

Capítulo 4

Hipp

Este capítulo está dedicado a la descripción detallada del *framework Hipp*. Se comienza en la primera sección estableciendo el objetivo del mismo, justificando las elecciones de su arquitectura y la técnica de mantenimiento de consistencia que utiliza para finalizar detallando las características para las que está diseñado. A continuación se describe su arquitectura, se muestra su diseño en capas y se describe detalladamente cada una de ellas. Por último se presenta una guía de uso del *framework*, la cual incluye también una serie de buenas prácticas.

1. Introducción

Hipp [Quiroga], por *Highly interactive peer to peer*, es un *framework* para el desarrollo de aplicaciones *groupware sincrónicas* con arquitectura *peer to peer*, que presenten un alto grado de situaciones en las que se pueden producir inconsistencias. Un ejemplo de estas aplicaciones, puede ser un rompecabezas colaborativo. Como contrapartida, un ejemplo de aplicación *groupware sincrónica* que no tiene un alto grado de situaciones en las que se pueden producir inconsistencias, es el chat.

Hipp utiliza modelos replicados sobre una arquitectura *peer to peer*. Esto significa que cada uno de los nodos contará con una copia del modelo de objetos de la aplicación y que no existirán entidades distinguidas en su arquitectura. El uso de modelos replicados permite que los cambios puedan ser aplicados localmente antes de ser distribuidos al

resto de los nodos. Esto hace que el *tiempo de respuesta*, uno de los aspectos más importantes en el desarrollo de aplicaciones *groupware sincrónicas*, se reduzca considerablemente.

Para la elección de la arquitectura del *framework* se analizaron los aspectos de estabilidad y escalabilidad del sistema y la facilidad de desarrollo de los algoritmos de manejo de consistencia. Comparando en cuanto a la estabilidad las dos alternativas de arquitectura (*peer to peer vs.* cliente – servidor*),* se observa mucho más ventajosa la primer alternativa. Esto es así, debido a que la arquitectura cliente - servidor depende del servidor para su funcionamiento, cualquier inconveniente en él pondrá en jaque la disponibilidad del sistema. Del otro lado, las arquitecturas *peer to peer*, distribuyen los servicios entre una gran cantidad de nodos, los cuales se supone no fallarán todos a la vez. Con lo cual, el sistema no se verá afectado por la caída individual de cualquiera de sus participantes.

Si se analizan las dos arquitecturas con respecto a la escalabilidad del sistema, se obtendrá el mismo resultado. Al igual que con el estudio de estabilidad, el servidor es el que representa el punto crítico. En cuanto aumente la cantidad de usuarios conectados al sistema, en el servidor aumentará la tasa de procesamiento y también la cantidad de conexiones a atender. Se sabe, estas dos variables tienen límites, los cuales imponen una cota superior a la escalabilidad del sistema. Mientras tanto, en una arquitectura *peer to peer*, si el aumento de usuarios se distribuye de forma aproximadamente uniforme entre la red de *peers*, el impacto en el sistema será mucho menor, permitiendo una escalabilidad mayor.

Por último, si consideramos la facilidad de desarrollo de los algoritmos de manejo de consistencia que implica cada arquitectura, el resultado que se obtiene, es contrario a los anteriores. Al igual que en los dos casos previos, el factor determinante, es el hecho que las arquitecturas cuenten o no con una entidad distinguida. Contar con el servidor en la arquitectura cliente – servidor hace que se simplifiquen los algoritmos de manejo de consistencia. Esto se debe a que permite la centralización de la toma de decisiones cuando surgen posibles inconsistencias. En las arquitecturas *peer to peer*, estas decisiones también deberán ser tomadas, pero individualmente en cada uno de los

nodos. La dificultad entonces, radica en que se tomen decisiones que hagan arribar al mismo resultado a todos los nodos.

Para el mantenimiento de consistencia, *Hipp*, utiliza un algoritmo de *Operational Transformation* (*OT*). La elección de *OT* por sobre las estrategias de orden total y bloqueos, se basa en los argumentos expuestos en el capítulo 2. Los cuales evidencian que en una arquitectura *peer to peer* pura no existe una entidad distinguida que se encargue de mantener un orden total entre las operaciones que se generan. Y, que la alterativa de establecimiento de bloqueos disminuye drásticamente la interactividad entre los participantes. La implementación de un algoritmo de *OT* y su configuración a cada dominio (como se verá más adelante en este capítulo) no es una tarea sencilla, pero permite una alta interactividad en arquitecturas descentralizadas.

Para encapsular el comportamiento común del algoritmo de manejo de consistencia y proveer puntos de configuración para determinados aspectos, *Hipp* se construyó como un *framework*. Esta decisión tiene el objetivo de paliar las dificultades de diseño, implementación y *testing* de los algoritmos de *OT*, proveyendo una implementación probada y configurable al usuario del *framework*. Así, el único aspecto (de los tres analizados) que parecía desventajoso en primera instancia, ya no lo es.

Hipp está diseñado para proveer:

- Alta interactividad, el *tiempo de respuesta* se ha minimizado.
- Tiempo real, los *tiempos de notificación*, aunque dependen de la latencia de la red, se han optimizado.
- Distribución, no se asume que los participantes estarán conectados a la misma computadora o a la misma LAN.
- Ad-hoc, no es necesario que los usuarios sigan una secuencia de pasos planificada, no se debe establecer *a priori* que información será accedida.
- Enfoque, el *framework* está dispuesto a tratar con un alto grado de conflictos de acceso, cuando los participantes trabajen sobre los mismos objetos.

2. Arquitectura

La arquitectura de *Hipp* es *peer to peer*. Ésta se compone por nodos iguales distribuidos en una red, la cual permite su comunicación. La comunicación entre los nodos se realiza por medio de mensajes, los cuales provienen de un emisor y se dirigen a todo el resto de los nodos. El *framework* supone que todos los nodos tienen conexión directa entre sí, es decir que no requieren de nodos intermedios para su comunicación. Esta suposición permite el envío de mensajes desde un nodo hacia el resto de ellos de forma más sencilla. Será responsabilidad del componente de comunicación (más adelante en este capítulo) dar el soporte para dicha suposición.

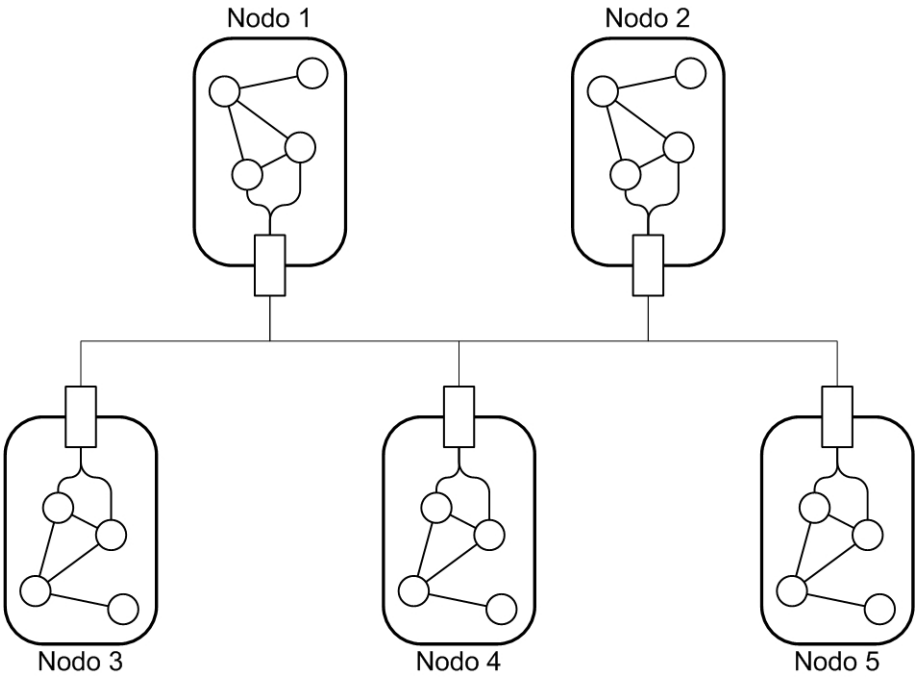

Figura 4.1. Arquitectura *Hipp*.

Cada uno de los nodos cuenta con una copia del modelo de dominio. Así, cada cual será responsable de aplicar en él, los comandos generados localmente y los recibidos desde otros nodos. Los modelos serán idénticos cuando el sistema se encuentre en reposo, es decir, cuando no existan comandos viajando por la red o a la espera de ser ejecutados en algún nodo.

Internamente en cada nodo, *Hipp* se organiza en cuatro capas con diferente objetivo cada una. Las capas se comunican con sus adyacentes vía interfaces o clases abstractas. La *capa de abstracciones de dominio* es la que se encuentra más "cerca" de la

aplicación a desarrollar. Sus componentes abstraen el comportamiento común de los objetos de dominio que necesita Hipp para su funcionamiento. Siguiendo a esta capa se encuentra la *capa de mantenimiento de consistencia*. Ésta alberga la implementación del algoritmo de *Operational Transformation*. A continuación se encuentra la *capa de interfaz de comunicación*. El objetivo de esta última, es concentrar las abstracciones que se encargan de relacionar el algoritmo de *OT* con el software de comunicación. Última de todas aparece la *capa de comunicación*, la cual encapsula el software de comunicación. En la próxima sección se describirá en detalle cada capa.

Figura 4.2. *Hipp,* diagrama de capas.

3. Diseño

En esta sección se describirá en detalle cada una de las cuatro capas que componen el framework haciendo incapié en la estructura y el comportamiento de cada una de las clases e interfaces que las componen.

3.1. Capa de abstracciones de dominio

La capa de abstracciones de dominio está compuesta básicamente por operaciones y transformadores. Las operaciones son subclases de la clase abstracta Operation. Ellas

redefinirán el método abstracto *execute()* y otros dos métodos más de los que se hablará más adelante, en esta misma sección. El método *execute()* recibe un objeto como parámetro y efectúa alguna acción sobre este objeto. Típicamente, dicho objeto será un objeto de dominio o un Facade [Gamma] al modelo de dominio. Los nodos de la aplicación *groupware* se comunicarán enviándose operaciones. Todas las acciones que deban propagarse entre los nodos deberán encapsularse en operaciones.

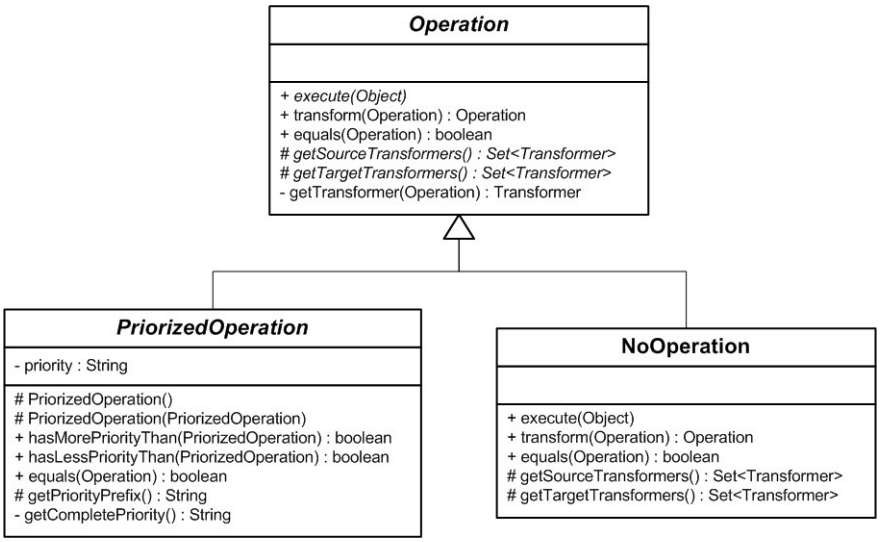

Figura 4.3. Jerarquía Operation.

En el ejemplo del rompecabezas colaborativo, las acciones de mover y encastrar pieza deberán encapsularse en operaciones. Así, existirá una subclase de la clase abstracta Operation por cada tipo de acción del dominio. Es decir, se contará con las clases MovePieceOperation y FitPieceOperation.

Como se puede observar en la Figura 4.3, la clase Operation, tiene una subclase abstracta llamada PriorizedOperation. Ésta representa a las operaciones priorizadas y a pesar que oculta los valores de prioridad al resto de las clases, provee toda la funcionalidad requerida para su uso. Al crear una instancia de una subclase de PriorizedOperation, si no se especificara ningún parámetro, se inicializará la prioridad de la operación con un *String* compuesto por una parte aleatoria y una parte fija. Por otra parte, si en la construcción de una instancia de una operación priorizada se pasa como parámetro otra operación priorizada, se tomará de esta última el valor de prioridad.

Las prioridades de dos operaciones priorizadas pueden ser comparadas utilizando dos métodos, *hasMorePriorityThan()* y *hasLessPriorityThan()*. El primero requiere que las prioridades de las operaciones sean diferentes (en caso contrario lanzará una excepción) mientras que el segundo, no. Por último, la clase ofrece la posibilidad que sus subclases especifiquen un prefijo para la prioridad de la operación, por medio de la redefinición del método *getPriorityPrefix()*. Esto resulta particularmente útil cuando se quieren establecer diferentes niveles de prioridad entre las operaciones, de manera explícita.

La creación del *String* con una parte aleatoria y otra fija utilizado en la inicialización de la prioridad de una operación recién creada, es responsabilidad del Priorizer. Éste es un Singleton [Gamma] generador de prioridades. Dado un identificador en la inicialización del Priorizer, que deberá ser único en toda la red de nodos, éste genera prioridades cuyos valores serán únicos, concatenando un número aleatorio e irrepetible con dicho identificador. Para obtener un número aleatorio e irrepetible, el Priorizer llena una lista ordenada con los números del 0 al 100, de la cual irá removiendo números de forma aleatoria. Cada vez que esta lista se vacíe, el Priorizer la volverá a llenar con los siguientes cien números.

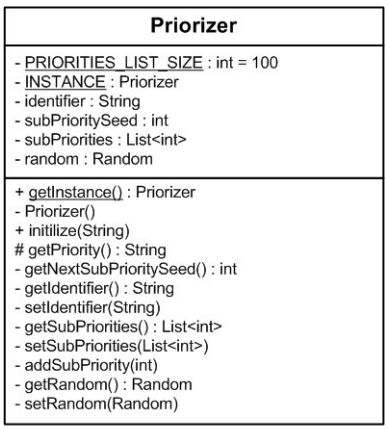

Figura 4.3.1.
Clase Priorizer.

Los transformadores son subclases de la clase abstracta Transformer. Como se dijo en el capítulo 3, los transformadores adaptan las operaciones cuyo *definition context* difiere del *execution context*, para hacerlas ejecutables. Los transformadores redefinen el método *transform()*, el cual recibe como parámetro dos operaciones y devuelve otra como resultado de la transformación. Las dos operaciones recibidas son la operación

original a transformar y la operación con la cual se desea transformarla. Además, cada transformador debe ser un Singleton [Gamma].

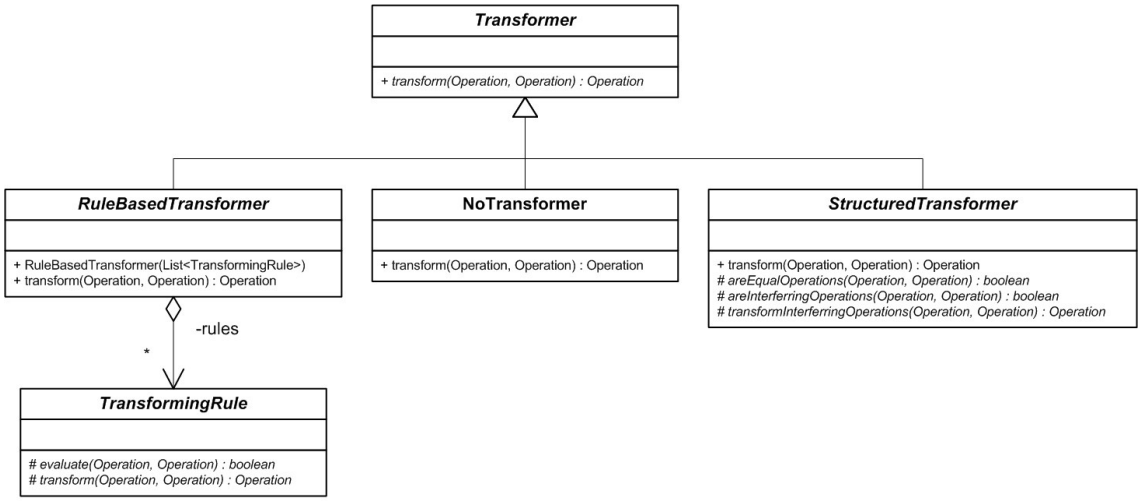

Figura 4.4. Jerarquía Transformer.

Por cada subclase de la clase abstracta Operation, deberá existir un transformador y otros dos por cada combinación binaria de dichas subclases. Así, si se tienen dos tipos de operaciones, A y B, deberá existir un transformador para transformar una A dada otra A, otro para una B dada otra B, otro para una A dada una B y otro para una B dada una A.

Si volvemos al ejemplo del rompecabezas colaborativo, en él, tendremos cuatro transformadores. Uno para transformar operaciones de encastre entre sí, otro para transformar operaciones de movimiento entre sí, otro para transformar operaciones de encastre con operaciones de movimiento y por último, otro para transformar operaciones de movimiento con operaciones de encastre.

Aquellos otros dos métodos a los que se hizo referencia en la descripción de la clase Operation son *getSourceTransformers()* y *getTargetTransformers()*. El primero devuelve el conjunto de los transformadores que toman como operación original una operación de la misma clase en donde está definido el método. Es decir, que si tenemos una subclase de la clase Operation llamada A, en ella la redefinición del método *getSourceTransformers()* devolverá un conjunto formado por todos los transformadores que tomen como operación original, operaciones de tipo A. El método

getTargetTransformers() tiene la misma esencia. Sólo se diferencia del anterior en que devuelve el conjunto de los transformadores que toman como operación con la cual se desea transformar, una operación de la misma clase en donde está definido el método.
Hipp provee además dos subclases abstractas de la clase Transformer, las cuales resultan ventajosas para modelar transformadores complejos. Una de ellas es la clase StructuredTransformer. Este transformador, utilizando el patrón *Template Method* [Gamma], estructura la transformación en tres casos típicos. El primero de ellos, es el caso en que las operaciones a transformar son iguales. Si esto sucede, ya que la operación con la cual se desea transformar ya fue ejecutada, se devolverá una instancia de la clase NoOperation. El segundo caso ocurre cuando las operaciones a transformar no interfieren entre sí, es decir que sin importar el orden en que se ejecuten y sin la necesidad de ser transformadas, al finalizar la ejecución de ambas, el modelo resultante es el mismo. El resultado a devolver en este caso será la operación a transformar, sin ninguna modificación.

El último caso sucede cuando las operaciones a transformar efectivamente interfieren entre sí, es decir que ineludiblemente se requiere una transformación. En este caso, se invocará el método abstracto *transformInterferringOperations()*, el cual deberá ser implementado por las subclases. Para determinar si las operaciones son iguales o interfieren entre sí el StructuredTransformer invocará los métodos abstractos *areEqualOperations()* o *areInterferringOperations()* según corresponda. Ambos métodos deberán ser implementados por las subclases.

La otra subclase de la clase Transformer que resulta útil para la implementación de transformadores complejos, es la clase RuleBasedTransformer. Ésta permite dividir las reglas de transformación correspondientes a los diferentes casos, en distintas clases. Para ello cuenta con una lista de instancias de la clase abstracta TransformingRule, que se recorre cuando se invoca el método *transform()*. Cada TransformingRule de la lista se evaluará (mediante el método *evaluate()*) y en caso de satisfacerse, se ejecutará mediante el método *transform()*. La clase TransformingRule define abstractos estos dos métodos para que sean implementados por cada una de las reglas específicas.

Estos dos grupos de clases, operaciones y transformadores, son las únicas abstracciones específicas de la aplicación, necesarias para garantizar el mantenimiento de consistencia.

Además, en esta capa, se proveen dos clases útiles para el desarrollo de aplicaciones. Estas son NoOperation y NoTransformer. La primera, corresponde a una operación que no realiza ninguna acción sobre el modelo ni sobre ningún otro objeto. Así, su ejecución, es inocua. La NoOperation es particularmente útil en los transformadores, cuando la operación resultante de la transformación no deba realizar ninguna acción.

El NoTransformer tiene una lógica similar, devuelve siempre la operación original como resultado del método *transform()*. Es decir, dadas dos operaciones *a* y *b* al método *transform()*, el NoTransformer devolverá siempre *a*.

La clase Operation define además los métodos *getTransformer()* y *transform()*. El primero es un método privado que recibe la operación con la cual se transformará la operación original y devuelve el transformador adecuado. Para obtener dicho transformador, este método intersecta los *sourceTransformers* de la operación original con los *targetTransformers* de la operación pasada como parámetro. Si esta intersección resulta vacía, se devuelve una instancia de NoTransformer. En caso contrario se devuelve el primer transformador de la intersección.

El método *transform()* es el encargado de devolver la transformación de la operación original con la operación pasada como parámetro. Para ello, utilizando el método *getTransformer()*, obtiene el transformador adecuado e invoca sobre éste, el método *transform()* pasándole como parámetro ambas operaciones.

3.2. Capa de mantenimiento de consistencia

La capa de mantenimiento de consistencia es en donde se encuentra la implementación del algoritmo de *Operational Transformation*. Su principal objetivo es transformar y ejecutar las operaciones (tanto locales como remotas) de forma tal, que el modelo resultante obtenido sea único, independientemente del orden de arribo de las operaciones, pero respetando la intención de los usuarios.

Existe la posibilidad (y es bastante grande) que una serie de operaciones arribe a un nodo en distinto orden del que fueron generadas. Peor aún, es muy probable que el orden de arribo de estas mismas operaciones, sea diferente en distintos nodos. Esta divergencia se debe a cuestiones relacionadas con los tiempos de envío, como por ejemplo la latencia de la red.

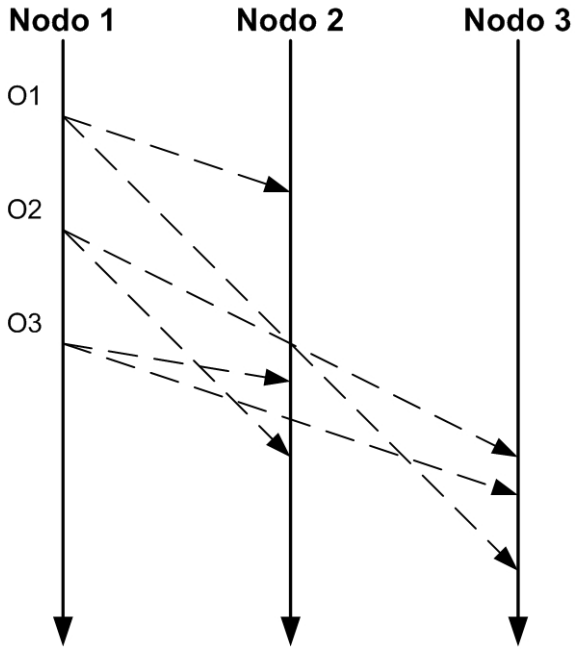

Figura 4.5. Orden de generación y orden de arribo.

Así, el propósito de esta capa es hacer que el *framework* sea independiente del orden de arribo de las operaciones y, respetando las intenciones de los usuarios, obtenga en cada uno de los nodos el mismo modelo resultante.

Algoritmo de mantenimiento de consistencia

Entre todos los algoritmos de *Operational Transformation* investigados se eligió implementar adOPTed debido a que:

- Está probado que su algoritmo de integración es correcto [Lushman].
- Existe la información suficiente para su implementación [Ressel].
- Puede extenderse para soportar operaciones *undo* [Gunzenhäuser].
- Permite una arquitectura replicada y multicast.

- Aunque está probado que sus funciones de transformación son incorrectas [Imine 1], no es necesario utilizarlas. Y si así lo fuera, se pueden utilizar otras [Imine 2], que si lo son.

Gran parte del algoritmo adOPTed está basada en el algoritmo dOPT. Las diferencias más importantes son que adOPTed introduce un modelo de interacción (Interaction Model) multidimensional y la función doblemente recursiva Translate Request (Figura 4.6).

En adOPTed cada nodo contiene las siguientes estructuras de datos locales:

- s, es el estado de la aplicación.
- k, es el contador de los requerimientos generados localmente. Con cada nuevo requerimiento generado se incrementará en uno.
- v, es el vector de estado del nodo (ver capítulo 3).
- Q, es la cola de requerimientos del nodo (ver capítulo 3). En ella tendrán prioridad los requerimientos generados localmente.
- L, es la bitácora de requerimientos (ver capítulo 3).
- G, es el modelo de interacción multidimensional. Su objetivo es almacenar las transformaciones de los requerimientos para no tener que volver a efectuar dichas transformaciones, ya que es posible que se necesite usarlas reiteradamente.

```
(1.1)  Main:
(1.2)    Initialize
(1.3)    while not aborted
(1.4)      if there is an input o
(1.5)        Generate Request(o)      ;locally
(1.6)        Execute Request
(1.7)      else
(1.8)        Receive Request          ;from remote
(1.9)        Execute Request

(2.1)  Initialize:
(2.2)    s ← s0                ;the initial application state
(2.3)    L ← empty set
(2.4)    Q ← empty set
(2.5)    G ← initial interaction model
(2.6)    v ← (0, 0, …, 0)    ;initial state vector
(2.7)    k ← 1               ;initial request id
(2.8)    u ← identification of local participant

(3.1)  Generate Request (o):
(3.2)    r ← (u, k, v, o)
(3.3)    Q ← Q + r
(3.4)    L ← L + r
(3.5)    k ← k + 1
(3.6)  broadcast r to other participants

(4.1)  Receive Request:
(4.2)    if there is a request from network
(4.3)      choose request r
(4.4)      Q ← Q + r
(4.5)      L ← L + r

(5.1)  Execute Request:
(5.2)    if there is r in Q satisfying Executable?(r, v), then
(5.3)      choose executable request r = (j, kj, vj, oj)
(5.4)      Q ← Q − r
(5.5)      r'' ← Translate Request (r, v)
(5.6)      apply operation o(r'') as user j to state s
(5.7)      increment j-th component of v
```

```
(6.1)  Executable?(r, v):
(6.2)    logical value of v(r) ≤ v

(7.1)  Translate Request(r, v)
(7.2)    (j, kj, vj, oj) ← r
(7.3)    if vj = v then
(7.4)      add request r to model G
(7.5)      return r
(7.6)    else if G contains r' for r translated to v
(7.7)      return r'
(7.8)    else
(7.9)      let i be such that
(7.10)         Reachable?(Decr(v, i)) and
(7.11)         vj[i] ≤ v[i] − 1
(7.12)       v' ← Decr(v, i)
(7.13)       ri ← Request(i, v[i])
(7.14)       ri' ← Translate Request(ri, v')
(7.15)       r' ← Translate Request(r, v')
(7.16)       (r'', ri'') ← tf(r', ri')
(7.17)       add requests r'', ri'' to model G
(7.18)       return r''

(8.1)  Reachable?(v):
(8.2)    for every i in U:
(8.3)      v[i] = 0
(8.4)      or w(Request(i, v[i])) ≤ v

(9.1)  Decr(v, i):
(9.2)    return copy of v with i-th component decremented

(10.1) Request(i, j)
(10.2)   return j-th request from user i in L
```

Figura 4.6. El algoritmo adOPTed.

Como se ve en la Figura 4.6, el proceso Main del algoritmo adOPTed, inicializa todas las estructuras de datos (2.2 – 2.8) y luego entra en un ciclo (1.3 – 1.9) que se ejecuta hasta ser abortado. Dentro de dicho ciclo, se aceptan operaciones provenientes del usuario local o requerimientos provenientes de otros usuarios y se los ejecuta.

Para ejecutar un requerimiento, primero se lo obtiene de la cola de requerimientos. Un requerimiento es ejecutable solo si su vector de estado es menor o igual (ver capítulo 3) que el vector de estado del nodo (6.1). Si existe en la cola más de un requerimiento ejecutable, se dará prioridad a los requerimientos locales (5.3). Esto garantizará que el usuario no deberá esperar a la ejecución de las operaciones de otros usuarios para ver sus propios cambios en la vista de la aplicación. El requerimiento elegido para ejecutar, se quitará de la cola (5.4) y se lo pasará a la función *Translate Request* para que lo transforme junto con el vector de estado del nodo.

La función *Translate Request*, transforma el requerimiento pasado como parámetro, desde su vector de estado *v(r)* al vector de estado pasado como parámetro *v*. Dicha transformación la efectúa recursivamente; si ambos estados son iguales (7.3), se agrega la transformación al modelo de interacción (7.4) y el requerimiento se devuelve tal como se recibió (7.5). Sino, se verifica si el modelo de interacción contiene la transformación buscada (7.6), en cuyo caso se devuelve (7.7). En caso contrario, se determina un predecesor inmediato *v'* de *v* (7.9 – 7.12) y se transforma *r* junto con *v'*. La condición (7.10) se asegura que el modelo de interacción contiene a *v'*. La condición (7.11) garantiza que *v'* es un sucesor de v_j, es decir que *v'* puede ser alcanzado desde v_j. Si existe más de uno de estos predecesores puede elegirse cualquiera de ellos. En (7.13) se hace que r_i sea el *v[i]* ésimo requerimiento proveniente del nodo *i*. A continuación se llama recursivamente a la función *Translate Request* para traducir los requerimientos *r* y r_i junto con *v'* (7.14 – 7.15). Los requerimientos *r'* y r_i' obtenidos son pasados a la función de transformación *tf* en (7.16). Finalmente se agregan r'' y r_i'' al modelo de interacción (7.17) y se devuelve *r''*, la versión transformada de *r'* (7.18). En (5.4), la operación del requerimiento *r''*, se ejecuta sobre el estado de la aplicación y se incrementa la posición correspondiente del vector de estado del nodo (5.7).

Implementación

Hipp implementa en esta capa todos los conceptos descriptos en el algoritmo adOPTed. Existe una clase Site cuya principal ocupación es el envío y recepción de operaciones. Esta clase también colabora con y se encarga de mantener:

- El estado de la aplicación (v. i. *state*).
- El identificador del nodo (v. i. *id*).
- La cola de requerimientos (v.i. *queue*).
- La bitácora de requerimientos (v. i. *requestsLog*).
- El ejecutor de operaciones sobre el modelo (v.i. *executor*).
- El generador de números de serie para los requerimientos (v. i. *serialNumberCounter*).
- El *facade* [Gamma] de comunicación (v. i. *siteConnection*). Ver *capa de interfaz de comunicación*.

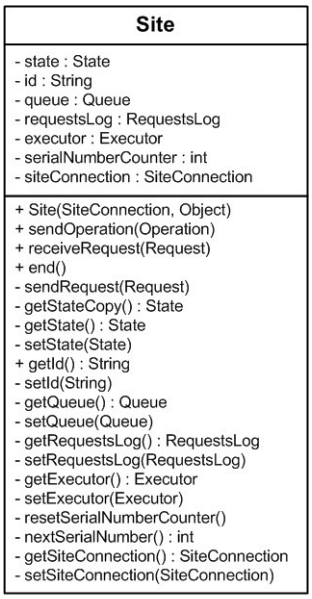

Figura 4.7.
Clase Site.

Para construir una instancia de la clase Site, se deberá especificar el *facade* de comunicación a utilizar y el modelo sobre el cual se ejecutarán las operaciones. En este constructor se guardará el *facade* de comunicación en una variable de instancia, se le asignará a éste, el mismo Site que se está construyendo, se inicializará el identificador del nodo obteniéndolo de dicho *facade*, se inicializará el estado de la aplicación con uno nuevo, se inicializará la bitácora de requerimientos con una nueva, se inicializará la cola

de requerimientos con una nueva, se inicializará el ejecutor, creándolo y pasándole el identificador del nodo, la cola de requerimientos, la bitácora de requerimientos, el estado del nodo y el modelo. Por último se dará comienzo a la ejecución del ejecutor, se inicializará en 1 el generador de números de serie para los requerimientos y se inicializará el Priorizer con el identificador del nodo.

Para el envío de operaciones, la clase Site provee el método *sendOperation()* el cual recibe como parámetro una operación, crea un nuevo requerimiento con el identificador del nodo, un nuevo número de serie, una copia del estado de la aplicación y la operación. A continuación, almacena este requerimiento en la bitácora de requerimientos y lo agrega como requerimiento generado localmente a la cola de requerimientos. Finalmente envía el requerimiento creado al resto de los nodos por medio del *facade* de comunicación.

La clase Site también provee un método para recibir los requerimientos enviados por resto de los nodos de la red. Este método se llama *receiveRequest()* y recibe como parámetro el requerimiento recibido. Al hacerlo, almacena dicho requerimiento en la bitácora de requerimientos y lo agrega a la cola de requerimientos como un requerimiento remoto.

Para finalizar la ejecución de operaciones, la clase Site, provee un método llamado *end()*, el cual interrumpe la ejecución del *thread* donde corre el ejecutor de operaciones.

Antes de que su operación sea ejecutada, los requerimientos se ingresan en la cola de requerimientos. La clase Queue caracteriza dicha cola, la implementa como un monitor [Andrews] y desarrolla el algoritmo que prioriza los requerimientos locales por sobre los remotos. Para ello, cuenta con dos listas de requerimientos. En una almacena los requerimientos generados localmente y en la otra los generados remotamente. Así, cuenta con dos métodos para agregar requerimientos a la cola, *addLocalRequest()* y *addRemoteRequest()*. Ambos son se ejecutan con exclusión mutua (*synchronized*) y siguen la misma lógica, agregan el requerimiento al final de la lista correspondiente y efectúan el *signal* (*notify*).

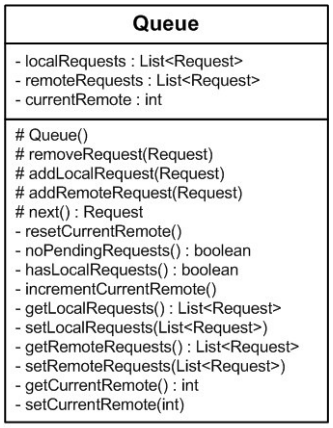

Figura 4.8.
Clase Queue.

Para obtener un requerimiento de la cola se debe invocar el método *next()*, el cual se ejecuta con exclusión mutua (*synchronized*). Este método se invoca solamente desde el ejecutor, el cual trabaja en coordinación con la cola. El método *next()*, mientras la cola no tenga requerimientos ejecutables duerme el *thread*. Cuando la cola si los tenga, se verificará si existen requerimientos locales en ella y si es así, se devolverá el primero. En caso contrario, se obtendrá el siguiente requerimiento remoto, se incrementará el indicador de siguiente y se devolverá dicho requerimiento. El indicador de siguiente requerimiento remoto no es más que un apuntador a una posición en la lista. Se utiliza este indicador pues es posible que el requerimiento remoto devuelto no sea ejecutable, lo cual implicará que el ejecutor intente obtener otro requerimiento. Al pedirse nuevamente un requerimiento, no se debería devolver el mismo, pues ya se verificó que ese requerimiento no es ejecutable y esto desencadenaría en *deadlock*. Este procedimiento no se efectúa con los requerimientos locales, pues éstos siempre llegarán a la cola en un orden tal que todos sean ejecutables.

Luego de comprobar que un requerimiento es ejecutable se deberá removerlo de la cola para ejecutarlo. Así, la cola de requerimientos provee el método *removeRequest()*, el cual recibe como parámetro el requerimiento a remover. En este método se eliminará el requerimiento, ya sea de la lista de requerimientos locales o de la de requerimientos remotos y se reiniciará el indicador de siguiente requerimiento remoto.

El Site es el encargado de invocar el constructor de la cola de requerimientos. En el cual, se inicializan vacías ambas listas y se reinicia el indicador de siguiente requerimiento remoto.

El ejecutor, representado en la clase Executor corre en un *thread* independiente. Su principal ocupación es, como su nombre lo indica, ejecutar las operaciones sobre el modelo. Para ello corre, hasta que el Site lo interrumpa, un ciclo en el cual obtiene un requerimiento de la cola y verifica si este es ejecutable. Si así lo es, remueve el requerimiento de la cola y le delega al requerimiento la responsabilidad de transformarse. Luego ejecuta la operación resultante de la transformación e incrementa el estado de la aplicación. Si el requerimiento obtenido de la cola no es ejecutable, reinicia el ciclo.

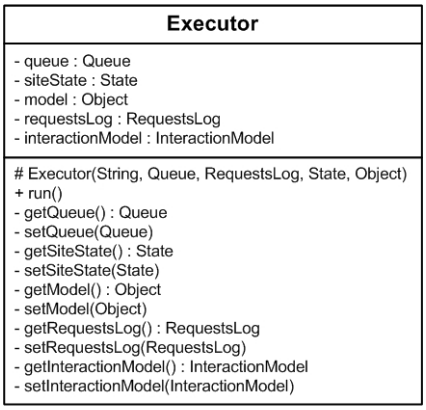

Figura 4.9. Clase Executor.

Como se explicó anteriormente, para la creación del ejecutor, el Site le pasa como parámetro el identificador del nodo, la cola de requerimientos, la bitácora de requerimientos, el estado de la aplicación y el modelo sobre el que se ejecutarán las operaciones. En el constructor del ejecutor, se almacenan todos estos objetos y se crea y almacena también el modelo de interacción.

Para ser enviadas entre los nodos, las operaciones se encapsulan en requerimientos, representados por la clase Request. Los requerimientos, además contienen el identificador del nodo que lo generó, el número de serie de la operación y el *definition context* de la operación.

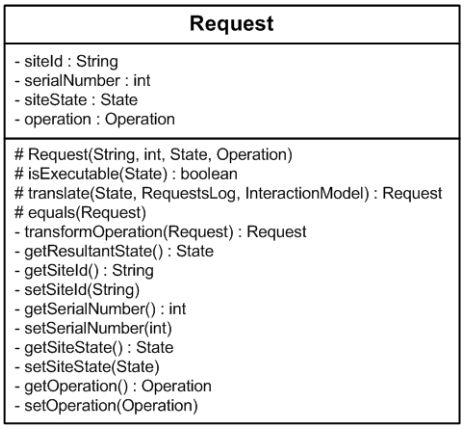

Figura 4.10. Clase Request.

Para determinar si un requerimiento es ejecutable, el ejecutor invoca el método *isExecutable()*, al cual le pasa como parámetro el estado en el cual desea saber si el requerimiento es ejecutable. A dicho estado se lo conoce como *execution context*. Este método verifica que el estado en el que se generó el requerimiento (*definition context*) sea menor o igual que el estado pasado como parámetro y que se haya ejecutado el requerimiento inmediato anterior a este, generado por el mismo nodo. Esta última condición, que no figura en el algoritmo adOPTed, tiene como propósito permitir que las operaciones enviadas por un mismo nodo puedan no llegar en el mismo orden en que se generaron, pero que sí sean ejecutadas en este mismo orden.

El método que determina si es necesario transformar una operación se encuentra en la clase Request. Este método se llama *translate()*, recibe como parámetros el estado al cual se desea transformar el requerimiento, la bitácora de requerimientos y el modelo de interacción y devuelve el requerimiento resultante de la transformación. La implementación de este método sigue los mismos pasos que la función *Translate Request* (7.1) del algoritmo adOPTed.

Tanto el *definition context* como el *execution context* se representan con vectores de estado, los cuales son instancias de la clase State. Esta clase cuenta con una tabla de *hash*, donde mantiene una serie de contadores de las operaciones ejecutadas en cada nodo.

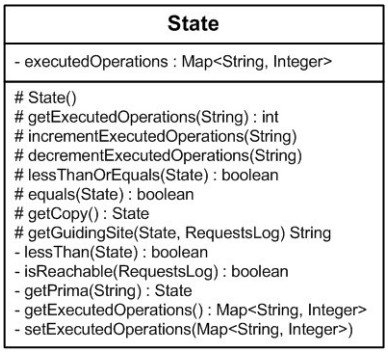

Figura 4.11. Clase State.

La clase State permite obtener, incrementar y decrementar la cantidad de operaciones ejecutadas provenientes de un determinado nodo a través de los métodos *getExecutedOperations()*, *incrementExecutedOperations()* y *decrementExecutedOperations()* respectivamente. Además con los métodos *equals()* y *lessThanOrEquals()* permite determinar si el estado es igual o menor o igual que otro estado pasado como parámetro.

La bitácora de requerimientos está representada por la clase RequestsLog. En ella se almacenan en un diccionario todos los requerimientos, tanto locales como remotos, que salen o llegan al nodo. Este diccionario tiene como clave un objeto, llamado RequestId, conformado por el identificador del nodo origen del requerimiento y el número de serie del requerimiento. Como valor el diccionario almacena el requerimiento en sí.

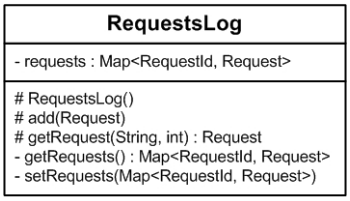

Figura 4.12.
Clase RequestsLog.

La bitácora de requerimientos se utiliza al momento de transformar operaciones, para obtener operaciones ya ejecutadas. Así, tiene dos operaciones típicas, agregar un requerimiento y obtener un requerimiento dado un identificador de nodo y un número de serie. Estas operaciones están modeladas por los métodos *add()* y *getRequest()* respectivamente.

La última clase que compone la capa de mantenimiento de consistencia es la clase InteractionModel, la cual representa el modelo de interacción. Esta se encarga de almacenar los requerimientos resultantes de la transformación de otros requerimientos. Como vimos anteriormente, existen ocasiones en las que se deben realizar las mismas transformaciones sobre las mismas operaciones. Para no repetir una y otra vez estas transformaciones se utiliza el modelo de interacción.

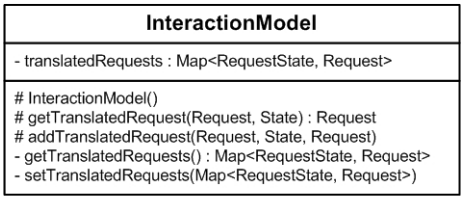

Figura 4.13.
Clase InteractionModel.

El modelo de interacción está compuesto por un diccionario que tiene un objeto RequestState como clave y un requerimiento como valor. El objeto RequestState solo tiene como objetivo relacionar un requerimiento con un estado para almacenarlo en el modelo de interacción. De esta forma, el RequestState, está compuesto por un requerimiento y un estado.

El InteractionModel responde a dos mensajes, *getTranslatedRequest()* y *addTranslatedRequest()*. El primero, dado un requerimiento y un estado, devuelve, si es que lo tiene almacenado, el requerimiento resultante de transformar el requerimiento pasado como parámetro al estado especificado. Mientras que el segundo, dado un requerimiento, un estado y un requerimiento transformado, lo almacena en el diccionario.

A continuación se presenta el diagrama de clases completo de la capa de mantenimiento de consistencia. En él solo se presentan los métodos más importantes de cada clase.

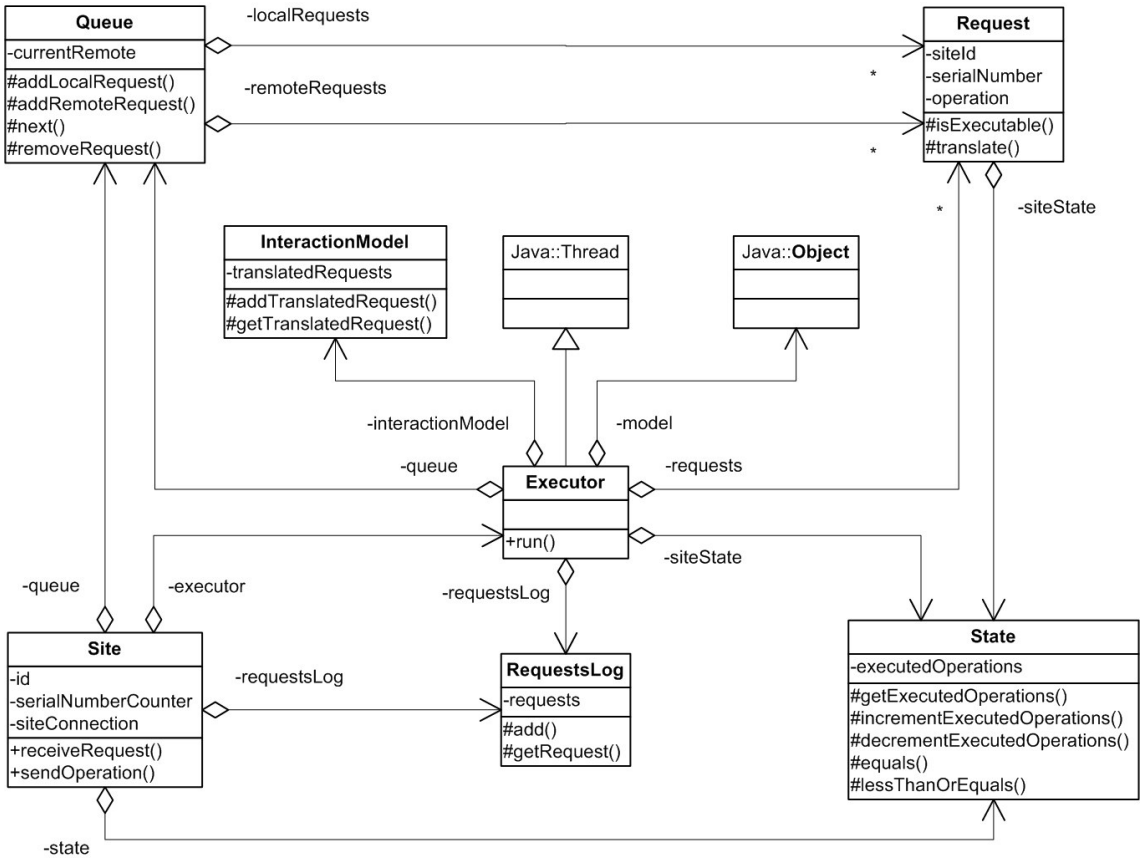

Figura 4.14. Diagrama de clases capa de mantenimiento de consistencia.

3.3. Capa de interfaz de comunicación

El objetivo de esta capa es presentar una interfaz de comunicación de alto nivel independiente del software de comunicación utilizado en la siguiente capa. Para ello cuenta con una serie de clases abstractas e interfaces qué, por medio de su subclasificación e implementación, permiten adaptarse al software de comunicación a utilizar.

Las principales atribuciones de la capa de interfaz de comunicación son el envío y recepción de requerimientos. Además, permite crear grupos de nodos, entrar y salir de ellos y devolver una lista de estos mismos. Los grupos de nodos delimitan el ambiente de colaboración. Es decir, que para que dos usuarios puedan colaborar en una misma aplicación, sus nodos deberán pertenecer al mismo grupo.

Implementación

La capa de interfaz de comunicación está compuesta por tres clases llamadas SiteConnection, GroupManager y Receiver y cuatro interfaces llamadas Sender, Marshaller, RequestsReceiver y Unmarshaller.

La clase SiteConnection es la más importante de esta capa, pues actúa como *facade* [Gamma] de comunicación. Para realizar las actividades inherentes al *facade* de comunicación, esta clase interactúa con instancias de GroupMananger, Receiver, Sender y Marshaller. De esta forma, el SiteConnection colaborará con el GroupManager para obtener la identificación del nodo y con el Sender y Marshaller para el envío de requerimientos. El método *receive()* forma parte de la interfaz RequestsReceiver, la cual es implementada por la clase SiteConnection. Este método, cuando es invocado por el Receiver, pasa al Site el requerimiento recibido.

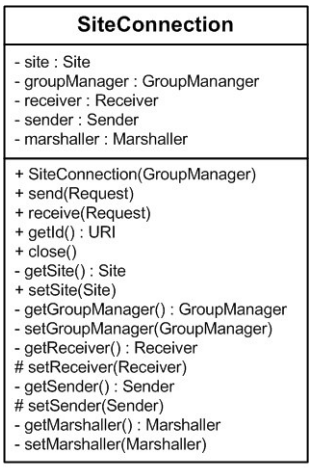

Figura 4.15. Clase SiteConnection.

La clase GroupManager es una clase abstracta cuyo objetivo es proveer una identificación al nodo de la aplicación y realizar las actividades relacionadas con los grupos, como crear un grupo, entrar a un grupo, salir de este y buscar grupos de nodos. Luego de entrar a un grupo, el GroupManager se encarga de crear las instancias de Receiver, Sender, Marshaller y Unmarshaller y pasárselas al SiteConnection. La forma de hacer todas estas actividades, claramente depende del software de comunicación a utilizar, por lo que están representadas por métodos abstractos. Para cada software de comunicación diferente y cada mecanismo de envío y recepción diferente, deberá existir

una subclase de la clase GroupManager que lo represente. La actual implementación de *Hipp* provee una subclase concreta (JxtaGroupManager, ver sección Interacción entre capas, Capa de interfaz de comunicación – capa de comunicación) para el uso de JXTA.

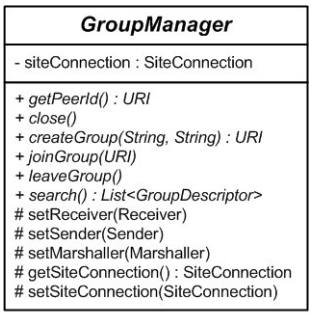

Figura 4.16. Clase GroupManager.

El Receiver es el objeto a cargo de recibir los mensajes, transformarlos en requerimientos utilizando una instancia de Unmarshaller y pasárselos al RequestsReceiver. Para cada forma de recibir los mensajes deberá haber una subclase de la clase Receiver. *Hipp* provee la clase PipeReceiver para el uso de *pipes* de JXTA.

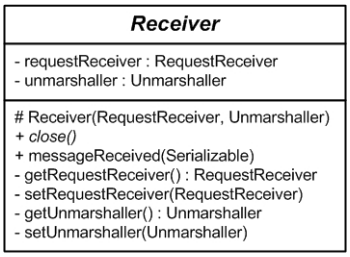

Figura 4.17. Clase Receiver.

La interfaz Unmarshaller solo declara el método *unmarshall()*, el cual toma un objeto serializable (que puede ser enviado por la red) como parámetro y devuelve un requerimiento. Una implementación de esta interfaz, XmlUnmarshaller, para utilizar JXTA se encuentra disponible en esta capa.

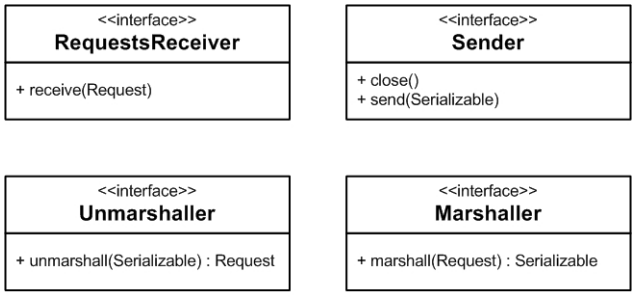

Figura 4.18. Interfaces RequestsReceiver, Sender, Unmarshaller y Marshaller.

Las implementaciones de la interfaz Sender deben implementar dos métodos, *close()* y *send()*. El primero interrumpe las conexiones con los otros nodos. Mientras que el segundo envía al resto de los nodos un mensaje previamente creado por la implementación de la interfaz Marshaller a partir de un requerimiento. Como en los casos anteriores, se provee una clase PipeSender y otra XmlMarshaller.

Si se decidiera utilizar otro software de comunicación diferente a JXTA, será necesario crear las subclases de GroupManager y Receiver y las implementaciones de las interfaces Sender, Marshaller y Unmarshaller correspondientes.

A continuación se presenta el diagrama de clases completo de la capa de interfaz de comunicación. En él se muestra como se relacionan las clases e interfaces de esta capa, presentando sólo los métodos más importantes de cada clase o interfaz.

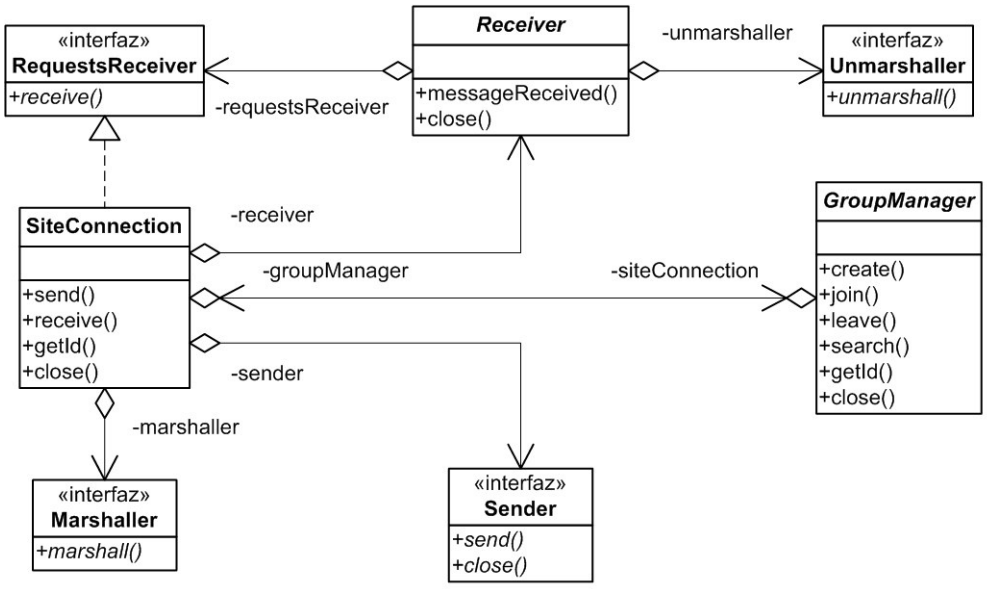

Figura 4.19. Diagrama de clases capa de interfaz de comunicación.

3.4. Capa de comunicación

La capa de comunicación está compuesta por el software de comunicación elegido. En la actual implementación de *Hipp*, sólo se provee soporte para un software de comunicación, JXTA. Se ha elegido este software de comunicación pues es el más popular de los *frameworks* de comunicación *peer to peer* del mercado. Si se deseara utilizar otro software de comunicación solo se necesitará codificar las subclases e implementaciones de interfaces especificadas en la capa anterior.

A continuación se presentará una breve descripción de JXTA.

JXTA

JXTA es una plataforma *peer to peer open source* creada por Sun Microsystems en el año 2001. Esta plataforma está definida como un conjunto de protocolos basados en XML, los cuales permiten que dispositivos conectados a una red, tales como teléfonos celulares, PDAs y PCs, intercambien mensajes entre sí de forma descentralizada.

Los protocolos de JXTA son independientes del lenguaje de programación, lo cual hace que puedan ser implementados desde una gran variedad de estos. Actualmente existen implementaciones *open source* para los lenguajes Java SE, Java ME y C/C++/.Net y

estas implementaciones son ínteroperables, lo cual permite la interacción entre pares implementados en diferentes lenguajes.

JXTA brinda una serie de definiciones sobre los conceptos utilizados en los protocolos. Así, define *peer* como la unidad básica de JXTA. Es decir que cada dispositivo que se conecte a la red JXTA será un *peer*, sin importar si este es un proveedor de servicios, un consumidor de estos o ambos. Los *peers* se auto organizan en grupos de *peers* llamados *peergroups*. Para comunicarse, los *peers* primero deben pertenecer a un *peergroup* y solo podrán comunicarse con los miembros del mismo grupo, aunque un mismo *peer* puede pertenecer a varios grupos a la vez. Además, para que dos peers puedan comunicarse deberán abrir un *pipe* entre ellos. Los *pipes* son el mecanismo básico de comunicación entre *peers*.

Para hacer pública la disponibilidad de un recurso en una red JXTA es necesario crear un *advertisement*. Los *advertisements* son documentos XML que anuncian la presencia de algún recurso. Los recursos pueden ser cualquier cosa que un *peer* pueda utilizar, como por ejemplo otro *peer*, un *peergroup*, *pipes* o servicios provistos por algún *peer*. Para que un *peer* pueda encontrar *advertisements*, deberá ejecutar el *discovery service*. El *discovery service* es un proceso que permite a los *peers* buscar recursos dentro de una red JXTA.

De esta forma, el *framework* JXTA provee el soporte necesario para construir sobre él aplicaciones que requieran formar redes de nodos en las cuales los nodos puedan agruparse según sus necesidades de comunicación, puedan enviar información entre ellos y se permita publicar y buscar recursos y servicios.

3.5. Interacción entre capas

La comunicación entre las capas presentadas anteriormente se da en diferentes formas y entre diferentes objetos. En esta sección se detalla por cada par de capas adyacentes, cada una de las relaciones entre los objetos involucrados.

Capa de abstracciones de dominio – Capa de mantenimiento de consistencia

La comunicación entre las primeras dos capas de *Hipp* involucra a la clase Operation en la capa de abstracciones de dominio y a las clases Site, Executor y Request en la capa de mantenimiento de consistencia. La interacción entre estas dos capas se da de cuatro formas diferentes. La esencia de éstas se puede resumir con cuatro palabras, *envío*, *ejecución*, *transformación* y *encapsulamiento*.

La primera de ellas, *envío*, se da entre el nodo (Site) y las operaciones. El nodo, por medio del método *sendOperation()*, es el encargado del *envío* de las operaciones desde sí mismo hacia el resto de los nodos. El implementador de una aplicación *Hipp* será el que se ocupe de invocar este método cada vez que desee enviar una operación al resto de los nodos.

La segunda palabra, *ejecución*, es lo que hace el Executor sobre las operaciones. En el método *run()*, luego de ser transformadas, el Executor invoca el método *execute()* sobre las operaciones.

Las últimas dos palabras, *transformación* y *encapsulamiento*, involucran a las mismas dos clases, Request y Operation. El requerimiento, en el método *transformOperation()*, es el encargado de invocar el método *transform()* sobre las operaciones. Éste, luego de obtener el transformador adecuado desencadena la transformación. Los requerimientos, además de disparar la transformación de operaciones, las encapsulan para su envío entre los nodos. En este encapsulamiento, los requerimientos, agregan información de vital importancia para la correcta ejecución de las operaciones. Esta información incluye el identificador del nodo origen de la operación, el número de serie de ésta y el estado del la aplicación al momento de la generación de la operación.

Capa de mantenimiento de consistencia – Capa de interfaz de comunicación

La interacción entre las dos capas del medio de Hipp implica a las clases Site y Request de la capa de mantenimiento de consistencia y a la clase SiteConnection y las interfaces RequestsReceiver, Marshaller y Unmarshaller de la capa de interfaz de comunicación. En este caso, la comunicación se puede resumir en cinco conceptos, *codificación*, *decodificación*, *identificación*, *envío* y *recepción*.

La *codificación* es la actividad principal y única del Marshaller sobre los requerimientos. Dado un requerimiento, a través del método *marshall()*, el Marshaller devolverá un mensaje codificado que puede ser transferido por la red. El concepto inverso del anterior, la *decodificación*, es la ocupación del Unmarshaller. Mediante el método *unmarshall()* se decodifica un objeto serializable en un requerimiento.

Para obtener el *identificador* del nodo de la aplicación, el Site invoca el método *getId()* del SiteConnection. El cual delega esta responsabilidad sobre el GroupManager. Este último deberá basarse en un mecanismo de identificación que asegure la unicidad de los identificadores.

El concepto de *envío* involucra a tres clases, Site, Request y SiteConnection. Al momento de enviarse un requerimiento al resto de los nodos, el Site invocará el método *send()* del SiteConnection pasándole como parámetro dicho requerimiento. Este último se encargará de invocar la *codificación* del mismo y le delegará la responsabilidad del envío al Sender.

En la *recepción* el procedimiento es similar. Las implicadas en este caso son las clases Site, Request y SiteConnection y la interfaz RequestsReceiver. Cuando se recibe un mensaje de la red, el Receiver, luego de su *decodificación*, le pasa el requerimiento decodificado al RequestsReceiver vía el método *receive()*. El SiteConnection es la clase que implementa la interfaz RequestsReceiver y en la implementación de dicho método invoca el método *receiveRequest()* sobre el Site pasándole como parámetro el requerimiento recibido.

Capa de interfaz de comunicación – Capa de comunicación

Los sujetos de la interacción, en este caso, dependerán del software de comunicación utilizado. Como se describió en las secciones que detallan cada una de estas dos capas, es posible, en diferentes instanciaciones del *framework*, utilizar distintos softwares de comunicación. Así, para cada uno de estos casos, se utilizarán diferentes subclases e implementaciones de las clases e interfaces descriptas en la capa de interfaz de comunicación.

Sin embargo, el diseño del *framework* hace que las nociones involucradas en la interacción entre estas dos capas no varíen aunque cambie el software de comunicación. Estas nociones implican obtener la identificación del nodo, crear un grupo, entrar a un grupo, salir de este, buscar grupos de nodos, enviar y recibir mensajes y codificar y decodificar requerimientos.

Para la utilización de JXTA como software de comunicación se implementaron cinco clases que dan solución a las nociones anteriores utilizando dicho *framework*. En este contexto, la clase encargada de satisfacer los primeros conceptos se llama JxtaGroupManager. Ésta contiene y colabora con una clase provista por JXTA llamada NetworkManager, la cual, es el punto de entrada a toda aplicación que utiliza JXTA. El NetworkManager permite configurar y echar a correr un nodo JXTA y obtener los objetos necesarios para su utilización. En el constructor del JxtaGroupManager se crea un NetworkManager, se lo configura, se lo hecha a correr y se espera por su conexión con la red.

JxtaGroupManager es una subclase concreta de la clase abstracta GroupManager y por consiguiente, implementa los métodos abstractos definidos en esta última. En estos métodos, por medio del NetworkManager, obtiene la identificación del nodo, crea un grupo, entra a un grupo, sale de este o busca grupos de nodos. Al final del método encargado de entrar a un grupo (*joinGroup()*) se crean y asignan las clases encargadas de enviar y recibir mensajes y codificar y decodificar requerimientos.

El envío y la recepción de mensajes lo llevan a cabo las clases PipeSender y PipeReceiver respectivamente. Ambas envían o reciben mensajes vía *pipes* de JXTA, los cuales obtienen del servicio de *pipes* pasado como parámetro en su constructor. Los mensajes consisten en objetos serializables que para enviarlos se encapsulan en un documento XML y se desencapsulan al recibirse. La clase PipeSender es una implementación de la interfaz Sender mientras que la clase PipeReceiver es una subclase concreta de la clase abstracta Receiver.

Por último, las clases a cargo de la codificación y decodificación de requerimientos son el XmlMarshaller y el XmlUnmarshaller. La implementación de ambas es realmente sencilla, ya que delegan la codificación y decodificación de los requerimientos a una

librería llamada XStream. XStream es una librería hecha en Java cuyo principal objetivo es codificar objetos en XML y viceversa.

4. Utilización de Hipp

En esta sección, se presenta una guía para la utilización de *Hipp*. La guía incluye los pasos necesarios y algunos consejos útiles para la implementación de una aplicación *groupware* utilizando el *framework* desarrollado.

La sección se divide en tres subsecciones, correspondiente cada una de ellas, a una etapa en el desarrollo de una aplicación *groupware* con *Hipp*. Estas etapas fueron nombradas de acuerdo a su relación con el *framework*. Así, en la subsección denominada *Extensión* se detallan las actividades relacionadas con la explotación de los *hot spots*, es decir, el desarrollo de subclases de clases abstractas del *framework* necesarias para su uso. A continuación de ésta, se encuentra la subsección llamada *Inicialización*, en la cual se describen los pasos para crear las instancias de las clases del *framework* necesarias para su funcionamiento. Por último en la subsección nombrada *Uso*, se especifica cómo hacer uso de cada una de las funcionalidades provistas por *Hipp*.

4.1. Extensión

La primera tarea al desarrollar una aplicación con *Hipp* es la especificación de un protocolo de comunicación entre las operaciones que se desarrollarán y el modelo de la aplicación. Esta tarea implica definir en una clase, la cual oficiará como *facade* [Gamma] del modelo de la aplicación, los métodos que invocarán las operaciones al ser ejecutadas. Una instancia de dicha clase, será la que se pasará como parámetro a las operaciones al invocar sobre ellas el método *execute()*.

Una buena práctica antes de implementar las operaciones específicas, es la implementación de una subclase abstracta de la clase Operation que evite tener que hacer *casting* a la clase del modelo específico de cada dominio en cada una de las operaciones. Así, debería definirse el método *execute()* en ésta clase como se muestra a continuación:

```
public void execute(Object model) {
    this.execute((SpecificModel) model);
}
```

Para que este método sea útil (y la aplicación no entre en un *loop* infinito) deberá definirse también el siguiente método abstracto:

```
public abstract void execute(SpecificModel specificModel);
```

De más está decir que se deberá reemplazar SpecificModel por el nombre de la clase del modelo a utilizar.

Al momento de implementar ésta clase abstracta, se deberá analizar si las operaciones involucradas en la aplicación, requieren ser priorizadas, no priorizadas o si existirán operaciones de los dos tipos.

Se requiere que una operación sea priorizada cuando, al determinar que dos operaciones interfieren entre sí, no existe ningún criterio natural que determine cuál es la operación que debe prevalecer. Esto sucede por ejemplo, en el caso del rompecabezas colaborativo, cuando dos usuarios deciden al mismo tiempo mover una misma pieza a dos ubicaciones diferentes. En este caso, no hay ningún criterio natural que indique cual de las dos operaciones debe prevalecer. Las subclases de PriorizedOperation, en estos casos, tienen la posibilidad de comparar sus prioridades y en base a esto determinar cual es la operación que prevalecerá.

En caso de estar implementando una aplicación en la cual existan operaciones que deban ser priorizadas y otras que no, se deberán implementar dos de estas clases abstractas. Una de ellas para las operaciones priorizadas (que extienda de la clase PriorizedOperation) y otra para las operaciones no priorizadas (que extienda de la clase Operation).

Una vez implementada la o las clases anteriores, deberán implementarse las operaciones correspondientes al dominio de la aplicación. Así, se creará una subclase de las clases recién definidas por cada operación requerida. Hasta este momento solo se implementarán los métodos *execute()* de cada operación, más adelante nos ocuparemos del resto de los métodos abstractos que deben redefinirse.

El próximo paso implica crear los transformadores necesarios. Como máximo se deberá crear un transformador por cada combinación binaria de operaciones. Decimos como máximo, pues si las operaciones involucradas en la combinación binaria no son conflictivas (es decir, no interfieren entre sí), no será necesario implementar el transformador. Esto se debe a la forma en la que está diseñado el algoritmo que establece qué transformador utilizar para transformar dos operaciones determinadas. En este algoritmo, si no se encontrara un transformador para las dos operaciones en cuestión, se devolverá una instancia de la clase NoTransformer. Éste al momento de transformar, siempre devolverá la operación a transformar, sin modificación alguna.
Para determinar si dos operaciones son conflictivas, se deben comparar los modelos resultantes luego de ejecutar, sin ninguna transformación, ambas operaciones en los dos órdenes posibles. Si los modelos resultantes son idénticos, las operaciones no son conflictivas.

En muchos casos las operaciones son parametrizadas, es decir poseen variables de instancia con valores que influyen en la ejecución de las mismas. Por ejemplo en una operación de encastre de una pieza con otra, se tendrá especificado el identificador de cada una de las piezas. En estos casos, puede suceder que para algunas combinaciones de valores de las variables de instancia, las operaciones resulten conflictivas y para otras no. En estos casos se considerará que las operaciones son conflictivas y se deberán tener en cuenta las diferentes combinaciones de valores de las variables de instancia en el transformador implementado.

Luego de determinar las combinaciones conflictivas de operaciones, se deberá implementar una subclase de la clase abstracta Tranformer por cada una de estas combinaciones. La clase Transformer define el método *transform()* abstractamente para ser redefinido por cada una de sus subclases. Al momento de implementar dicho método, el desarrollador se encontrará con al menos uno de estos tres casos típicos:

- Las operaciones son iguales. Esto sucede cuando dos usuarios realizan en forma concurrente la misma operación. En este caso, como la operación con la cual se desea transformar ya se ejecutó, la ejecución de la operación a transformar debe ser omitida. Con lo cual, debe devolverse una instancia de la clase NoOperation.
- Las operaciones no son conflictivas. En este caso, la operación que se desea transformar debe devolverse sin modificación alguna.
- Las operaciones son conflictivas. En este caso se deberá realizar una transformación a la operación a transformar. Dicha transformación, como se explicó en el capítulo 3, dependerá exclusivamente del dominio de la aplicación. En el próximo capítulo se ofrecen varios transformadores como ejemplo.

Para los transformadores que presenten estos tres casos, Hipp provee la clase StructuredTransformer. Creando una subclase de ésta e implementando los métodos *areEqualOperations()*, *areInterferringOperations()* y *transformInterferringOperations()*, dicha clase se encargará de devolver la operación correspondiente a cada caso.

Si el algoritmo de transformación de un transformador es un tanto más complejo que en el caso anterior, podrá utilizarse como superclase de éste, la clase RuleBasedTransformer. Ésta, permite dividir las reglas de transformación correspondientes a los diferentes casos, en distintas clases. Para ello cuenta con una lista de instancias de la clase abstracta TransformingRule, que se recorre cuando se invoca el método *transform()*. Cada TransformingRule de la lista, se evaluará (mediante el método *evaluate()*) y en caso de satisfacerse, se ejecutará mediante el método *transform()*. La clase TransformingRule define abstractos estos dos métodos para que sean implementados por cada una de las reglas específicas.

El algoritmo de Hipp que establece qué transformador utilizar para transformar dos operaciones determinadas, requiere que los transformadores devueltos por las operaciones sean iguales. Por lo cual, una buena práctica relacionada con este aspecto, es implementar los transformadores como Singleton [Gamma].

El algoritmo de Operational Transformation utilizado en *Hipp* (adOPTed) requiere para su funcionamiento, que las funciones de transformación cumplan las propiedades TP1 y TP2 (ver capítulo 3). Es muy importante, al desarrollar los transformadores de una aplicación, probar que éstos satisfacen dichas propiedades. Hacerlo, no es una tarea sencilla [Imine 2], pero da la certeza que los transformadores son correctos y evita la generación y permanencia de múltiples errores difíciles de reproducir una vez avanzado el desarrollo. Existen diferentes formas de encarar estas pruebas, algunas de ellas pueden verse en [Imine 1, Imine 3].

Cada operación definida anteriormente, deberá implementar los métodos abstractos *getSourceTransformers()* y *getTargetTransformers()* definidos en la clase Operation. El primero de éstos, deberá devolver el conjunto de los transformadores que toman como operación a transformar una instancia de la clase que implementa dicho método. Es decir, que si se está implementando el método *getSourceTransformers()* en la clase FitPieceOperation, se devolverá el conjunto de todos los transformadores que toman como operación a transformar instancias de la clase FitPieceOperation. Con el método *getTargetTransformers()* sucede lo mismo, pero se devolverá el conjunto de los transformadores que toman como operación con la cual transformar instancias de la clase que implementa el método. Es decir, que si se está implementando el método *getTargetTransformers()* en la clase FitPieceOperation, se devolverá el conjunto de todos los transformadores que toman como operación con la cual transformar, instancias de la clase FitPieceOperation.
Dado que tanto el conjunto de transformadores que devuelve uno u otro método es estático, no será necesario calcularlo cada vez que se lo vaya a utilizar. Por lo que, una buena práctica en este aspecto, es almacenar estos conjuntos en constantes o variables de clase.

Si la aplicación a desarrollar utilizará JXTA para comunicarse, las operaciones y los transformadores creados en los pasos anteriores, son las únicas extensiones de *Hipp* necesarias para que la nueva aplicación funcione. En caso que se desee utilizar otro software de comunicación, se deberán desarrollar las extensiones particulares de la capa de interfaz de comunicación detalladas anteriormente en este capítulo.

4.2. Inicialización

La siguiente tarea, luego de realizar las tareas de extensión, es la creación de las instancias de clases de *Hipp* necesarias para el desarrollo de una aplicación. Solo será necesario crear tres instancias de clases del *framework*.

El primer paso implica crear una instancia de la clase SiteConnection. En la invocación del constructor de ésta, se deberá especificar el GroupManager a utilizar en la aplicación. Si se ha desarrollado alguna subclase de la clase GroupManager en la sección anterior, podrá ser utilizada. Sino, se podrá utilizar la clase JxtaGroupManager, provista por *Hipp*. Utilizando esta última clase, la aplicación se comunicará vía XML sobre *pipes* de JXTA. La clase JxtaGroupManager, al final del método *joinGroup()*, crea las instancias del Sender, el Receiver, el Marshaller y el Unmarshaller, con lo cual no será necesario que el usuario del *framework* lo haga.
La última instancia a crear pertenece a la clase Site. En la invocación del constructor se deberá pasar como parámetro la instancia de la clase SiteConnection creada en el paso anterior y una instancia de la clase *facade* del modelo creada en la sección *Extensión*.

Las tres instancias creadas de las clases de *Hipp* (SiteConnection, GroupManager y Site) serán necesarias durante el uso de la aplicación que se está desarrollando. Por ello, se recomienda almacenarlas en variables de instancia de alguna clase que facilite su acceso. A continuación se muestran las tres líneas de código necesarias para la creación y almacenado de dichas instancias.

```
this.setGroupManager(new JxtaGroupManager());
this.setSiteConnection(new SiteConnection(this.getGroupManager());
this.setSite(new Site(this.getSiteConnection(), this.getModel()));
```

4.3. Uso

Una vez creadas las clases y las instancias necesarias, solo resta hacer uso de ellas. A continuación se presenta una lista de las funcionalidades provistas por *Hipp* con sus respectivas descripciones de uso.

Envío de una operación

Para enviar una operación desde un nodo al resto de ellos se debe invocar el método *sendOperation()* en la instancia de la clase Site creada en la sección anterior. En dicha invocación se pasará como parámetro la operación a enviar. Es importante destacar que para poder enviar operaciones es necesario que el nodo participe de algún grupo de nodos.

Cabe aclarar que no se muestra como funcionalidad la recepción de operaciones debido a que Hipp se encarga automáticamente de esta tarea, sin que el usuario deba invocar ningún método sobre algún objeto.

Obtención del identificador del nodo

Si fuera necesario, se podría obtener el identificador del nodo en el que se está operando. Para hacerlo solo será necesario invocar el método *getId()* del objeto Site creado en la sección anterior. Este método devolverá el identificador del nodo representado por un String.

Creación de un grupo

Para crear un grupo de nodos, se deberá obtener el GroupManager creado en la sección anterior e invocar sobre éste el método *createGroup()*. En dicha invocación, se deberá especificar el nombre y la descripción del grupo a crear. El GroupManager se encargará de la creación del grupo y su publicación en la red para que el resto de los nodos puedan entrar a él. Al finalizar, éste método devolverá el identificador del grupo creado.

Búsqueda de grupos

La búsqueda de grupos se realiza a través del método *search()* invocándolo sobre la instancia del GroupManager creada anteriormente. Este, dispara la búsqueda de grupos y se bloquea hasta obtener el resultado. Cuando lo obtiene, genera una lista de descriptores de los grupos encontrados y la devuelve. Los descriptores de grupo mantienen el identificador, el nombre y la descripción del nodo en cuestión.

Entrada a un grupo

Para entrar a un grupo solo se requiere enviar el mensaje *joinGroup()* al GroupManager instanciado en la sección anterior, especificando el identificador del grupo al que se

desea entrar. Antes de retornar, este método creará las instancias que oficiarán de Sender, Receiver, Marshaller y Unmarshaller. Es importante destacar que en *Hipp*, un nodo solo podrá estar en un grupo a la vez.

Salida de un grupo

Para salir de un nodo se deberá invocar el método *leaveGroup()* sobre la instancia del GroupManager creada en la sección anterior.

Cierre de la aplicación

Al momento de cerrar la aplicación, para evitar que queden hilos corriendo o conexiones abiertas, se deben invocar los métodos *end()* y *close()* de las instancias de las clases Site y SiteConnection creadas anteriormente. Estos métodos aseguran una correcta terminación del ejecutor de operaciones y de las conexiones establecidas por el SiteConnection, el GroupManager, el Sender y el Receiver.

Capítulo 5

Ejemplo de uso: Editor colaborativo de XML

En este capítulo se presenta un ejemplo de uso del *framework Hipp*. A continuación, se enuncia el objetivo de su desarrollo, una descripción de la funcionalidad que presenta, el diseño macro de su modelo, el análisis de sus operaciones y funciones de transformación, su integración con *Hipp*, el proceso de verificación que se llevó a cabo y las conclusiones propias de su desarrollo.

1. Objetivo

El objetivo del desarrollo de este sistema es probar el funcionamiento de *Hipp* con un prototipo sencillo, que presente una cantidad de casos de transformación razonable. Fue determinante al momento de seleccionarse, el hecho de contar con la definición de las operaciones y funciones de transformación [Imine 1].

2. Descripción

IXE (Interactive Xml Editor) es un editor colaborativo de XML implementado sobre el *framework Hipp*. Su propósito es permitir a un grupo de personas la escritura de un documento XML interactivamente. Es decir, que los diferentes usuarios conectados a la aplicación, compartirán un documento XML sobre el cual podrán efectuar directamente

las modificaciones que crean convenientes. IXE se encargará de propagar los cambios que cada uno imprima en el documento y de mantener las copias de éste sincronizadas en cada terminal.

IXE está basado en las operaciones y funciones de transformación descriptas en el paper "Development of transformation functions assisted by a theorem prover" [Imine 1]. En él, se presentan las operaciones básicas de un editor de XML y se muestra el proceso de modificación que sufrieron las funciones de transformación desde su diseño inicial hasta su corrección para satisfacer las propiedades TP1 y TP2.

Las operaciones utilizadas en el editor son cinco.

- Crear nodo. Dado el identificador del nodo a crear, el identificador del nodo padre de éste y el nombre del *tag* del nodo, crea un nodo en la ubicación correspondiente.
- Eliminar nodo. Dado el identificador del nodo a eliminar, elimina dicho nodo y todos sus hijos.
- Crear atributo. Dado el identificador del nodo donde se creará el atributo y el nombre del atributo a crear, crea el atributo asignándole un valor nulo.
- Eliminar atributo. Dado el identificador del nodo donde se encuentra el atributo a eliminar y el nombre del atributo a eliminar, lo elimina junto con su valor.
- Modificar atributo. Dado el identificador del nodo donde se encuentra el atributo a modificar, el nombre del atributo a modificar y el nuevo valor del atributo, asigna este nuevo valor al atributo.

El editor presenta una interfaz gráfica sencilla que permite observar el documento XML en forma de árbol. Al iniciar la aplicación, el árbol sólo contará con el nodo raíz, a medida que se opere sobre él, se verá su crecimiento. En la parte superior, se presenta una barra de herramientas con un botón para cada una de las operaciones antes descriptas y haciendo *click* derecho sobre el espacio del árbol, se mostrará un menú contextual con las mismas operaciones.

3. Diseño

IXE se diseñó siguiendo el patrón *Model-View-Controller* (*MVC*) [Burbeck]. Así, como se observa en el siguiente diagrama, existe un clase llamada IxeModel a cargo de la representación del documento XML. Ésta contiene el *tag* o nodo raíz (*rootNode*) del documento y un diccionario (*nodes*) cuya clave es el identificador del nodo y cuyo valor es el nodo mismo. En este diccionario se almacenan todos los nodos del documento para permitir un acceso más directo.

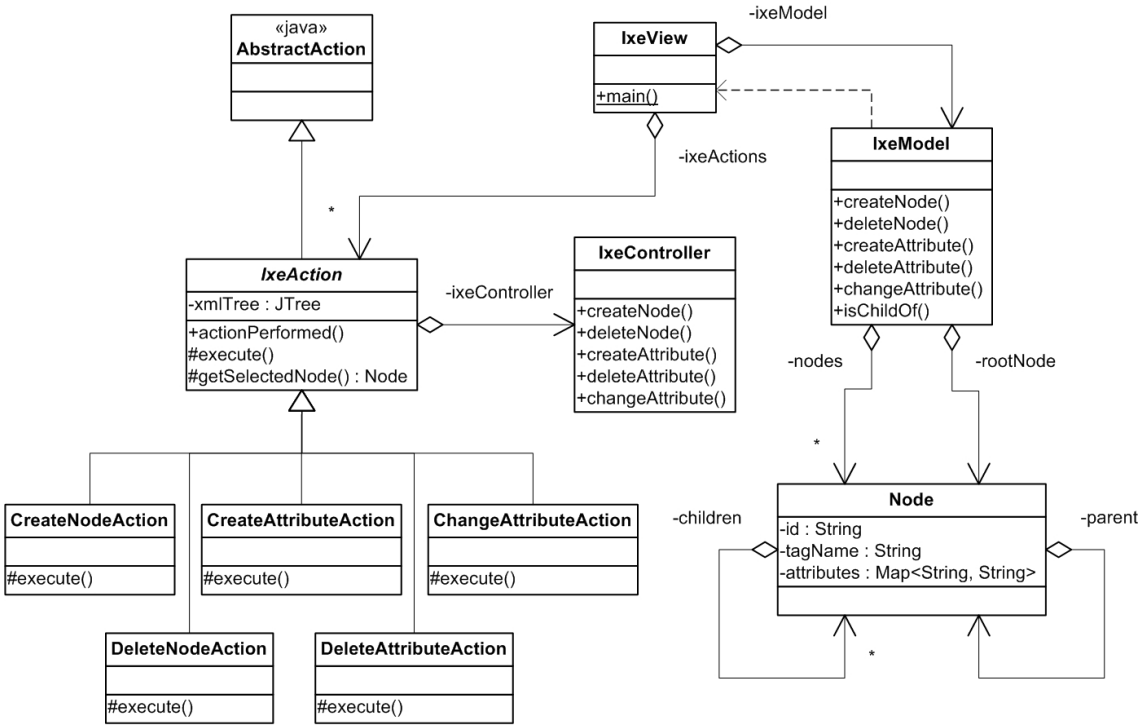

Figura 5.2. IXE - Diagrama de clases.

Los nodos del documento se representan con instancias de la clase Node. Cada nodo posee un identificador (*id*), un nombre (*tagName*) y un diccionario con sus atributos (*attributes*). Además, cada nodo conoce a su nodo padre (*parent*) y a sus nodos hijos (*children*).

La clase IxeModel ofrece métodos para crear y eliminar un nodo, para crear, modificar y eliminar atributos y para saber si un nodo es hijo de otro.

El *Controller*, provee los métodos necesarios para la creación de las operaciones mencionadas en la sección anterior. A diferencia de lo que establece el patrón *MVC*, y como se puede ver en la Figura 5.2, el IxeController no conoce al IxeModel. Esto se debe a que no lo necesita, ya que cuando se creen operaciones, el IxeController no las ejecutará sobre el *Model*, sino que las enviará a través del Site como se verá más adelante en la sección *Integración con Hipp*.

Por último, la clase IxeView es la encargada de la representación de la interfaz gráfica. Ésta, junto con la clase IxeModel, implementan el patrón *Observer* [Gamma]. La vista cumple el rol del observador mientras que el modelo el del observado. Así, cuando se efectúa alguna modificación en el modelo, éste notifica a la vista, vía el mecanismo de dependencias.

La vista, además, se encarga de invocar los métodos de creación de operaciones ofrecidos por el *Controller*. Para cada uno de estos métodos, existe una acción que se ocupa de su invocación. Estas acciones son las que activan los botones de la barra de herramientas o las opciones del menú contextual y se organizan en una jerarquía. En ella, la IxeAction extiende a la clase AbstractAction de Java y conoce al Controller y a la vista del árbol de nodos.

El diseño del esquema de ejecución de las acciones se hizo siguiendo el patrón *Template Method* [Gamma]. Al momento de la ejecución de una acción, en la IxeAction se verifica que alguno de los nodos XML se encuentre seleccionado, en cuyo caso delega a la subclase correspondiente su ejecución.

La clase IxeView se encarga también, en su constructor, de la creación del IxeModel y del IxeController y de registrarse como dependiente del primero.

4. Operaciones

Las operaciones con las que cuenta esta aplicación son cinco. Cuatro de ellas (CreateNodeOperation, DeleteNodeOperation, CreateAttributeOperation y DeleteAtributeOperation) son subclases de la clase abstracta IxeOperation definida según el criterio establecido en la sección *Extensión* del capítulo anterior. La restante (ChangeAttributeOperation) extiende la clase abstracta IxePriorizedOperation siguiendo el mismo criterio.

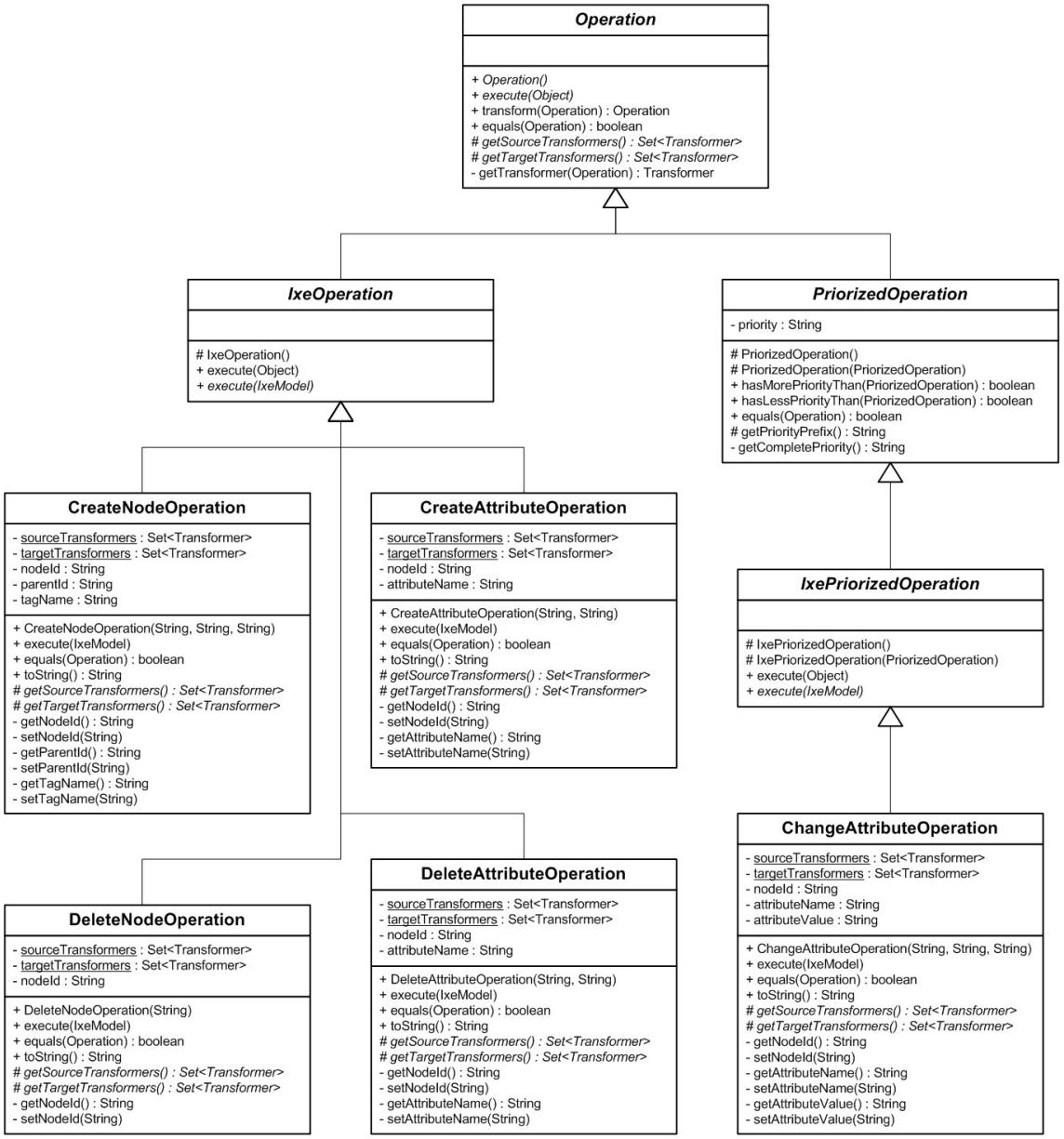

Figura 5.3. IXE - Jerarquía de operaciones.

Se separaron las operaciones en dos jerarquías distintas debido a que existe una única operación que requiere ser priorizada. Sucede esto, pues cambiar el valor de un atributo, es la única acción en la cual se necesita de un valor de prioridad para decidir un conflicto entre operaciones. Se podrá ver una explicación más detallada del caso en la próxima sección.

5. Análisis de las funciones de transformación

Si bien las funciones de transformación utilizadas en IXE fueron desarrolladas basándose en las funciones especificadas en lógica de primer orden en [Imine 1], las analizaremos como si las hubiésemos diseñado desde cero.

El primer paso implica definir las combinaciones conflictivas de operaciones. Para ello se deben analizar todas las combinaciones binarias entre las distintas clases de operaciones. A continuación se describirá cada uno de los casos:

Transformación de CreateNodeOperation(nodeId1, parentId1, tagName1) con:

- CreateNodeOperation(nodeId2, parentId2, tagName2): las operaciones no son conflictivas ya que el identificador del nodo a crear, deberá ser siempre diferente.
- DeleteNodeOperation(nodeId2): las operaciones son conflictivas en el caso en que el nodo a eliminar es ancestro del nodo a crear.
- CreateAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.
- DeleteAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.
- ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2): las operaciones no son conflictivas.

Transformación de DeleteNodeOperation(nodeId1) con:

- CreateNodeOperation(nodeId2, parentId2, tagName2): las operaciones no son conflictivas.
- DeleteNodeOperation(nodeId2): las operaciones son conflictivas en el caso que el nodo a eliminar de la operación ejecutada sea ancestro del nodo a eliminar de la operación a transformar o si las operaciones se refieren al mismo nodo.
- CreateAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.
- DeleteAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.

- ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2): las operaciones no son conflictivas.

Transformación de CreateAttributeOperation(nodeId1, attributeName1) con:

- CreateNodeOperation(nodeId2, parentId2, tagName2): las operaciones no son conflictivas.
- DeleteNodeOperation(nodeId2): las operaciones son conflictivas en el caso que el nodo a eliminar sea ancestro del nodo que contiene el atributo a crear.
- CreateAttributeOperation(nodeId2, attributeName2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y con el mismo nombre de atributo.
- DeleteAttributeOperation(nodeId2, attributeName2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y con el mismo nombre de atributo.
- ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y con el mismo nombre de atributo.

Transformación de DeleteAttributeOperation(nodeId1, attributeName1) con:

- CreateNodeOperation(nodeId2, parentId2, tagName2): las operaciones no son conflictivas.
- DeleteNodeOperation(nodeId2): las operaciones son conflictivas en el caso que el nodo a eliminar sea ancestro del nodo que contiene el atributo a eliminar.
- CreateAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.
- DeleteAttributeOperation(nodeId2, attributeName2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y sobre el mismo atributo.
- ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2): las operaciones no son conflictivas.

Transformación de ChangeAttributeOperation(nodeId1, attributeName1, attributeValue1) con:

- CreateNodeOperation(nodeId2, parentId2, tagName2): las operaciones no son conflictivas.
- DeleteNodeOperation(nodeId2): las operaciones son conflictivas en el caso que el nodo a eliminar sea ancestro del nodo que contiene el atributo a modificar.
- CreateAttributeOperation(nodeId2, attributeName2): las operaciones no son conflictivas.
- DeleteAttributeOperation(nodeId2, attributeName2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y sobre el mismo atributo.
- ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2): las operaciones son conflictivas en el caso que sean sobre el mismo nodo y sobre el mismo atributo.

Combinaciones conflictivas de operaciones	CreateNodeOperation	DeleteNodeOperation	CreateAttributeOperation	DeleteAttributeOperation	ChangeAttributeOperation
CreateNodeOperation	-	X	-	-	-
DeleteNodeOperation	-	X	-	-	-
CreateAttributeOperation	-	X	X	X	X
DeleteAttributeOperation	-	X	-	X	-
ChangeAttributeOperation	-	X	-	X	X

Figura 5.4. Combinaciones conflictivas de operaciones.

En la Figura 5.4 se muestra una tabla en donde están marcadas con una cruz las combinaciones conflictivas entre operaciones. En esta tabla las filas representan los tipos de las operaciones a transformar mientras que las columnas representan los tipos de las operaciones con las cuales se transformará.

5.1. Análisis de casos conflictivos

En la transformación de una CreateNodeOperation(nodeId1, parentId1, tagName1) con una DeleteNodeOperation(nodeId2) las operaciones son conflictivas en el caso en que el nodo eliminado (identificado por nodeId2) haya sido ancestro del nodo a crear (identificador por nodeId1). El conflicto subyace en que no se puede crear un nodo hijo de un ancestro inexistente. Con lo cual, en este caso, no será necesario ejecutar ninguna operación, debiéndose devolver como resultado de la transformación una instancia de la clase NoOperation.

En la transformación de una DeleteNodeOperation(nodeId1) con una DeleteNodeOperation(nodeId2) las operaciones son conflictivas en el caso que el nodo eliminado por la operación ejecutada (identificado por nodeId2) haya sido ancestro del nodo a eliminar por la operación a transformar (identificado por nodeId1) o en el caso en que las operaciones se refieren al mismo nodo. En esta combinación de transformación, el conflicto se debe a que no se puede eliminar un nodo ya eliminado. Para ello, es importante tener en cuenta que las operaciones de eliminación, no solo eliminan el nodo identificado, sino también todos sus hijos. En cualquiera de estos dos casos se deberá devolver una NoOperation como resultado de la transformación.

En la transformación de una CreateAttributeOperation(nodeId1, attributeName1) con una DeleteNodeOperation(nodeId2) las operaciones son conflictivas en el caso que el nodo eliminado (identificado por nodeId2) haya sido ancestro del nodo que contiene el atributo a crear (identificado por nodeId1). Este caso es similar a los dos anteriores, ya que el conflicto proviene de la eliminación del nodo; no se podrá crear un atributo en un nodo que ya no existe. En consecuencia, como en los casos anteriores, se deberá retornar como resultado de la transformación una instancia de la clase NoOperation.

En la transformación de una CreateAttributeOperation(nodeId1, attributeName1) con una CreateAttributeOperation(nodeId2, attributeName2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y con el mismo nombre de atributo (identificados por attributeName1 y attributeName2). A este caso se llega cuando dos usuarios de la aplicación deciden crear concurrentemente el mismo atributo sobre el mismo nodo. Dado que al momento de la

transformación, una de las operaciones ya fue ejecutada, no sería correcto el volver a hacerlo. Por lo cual, el resultado natural de esta transformación, será una instancia de la clase NoOperation.

En la transformación de una CreateAttributeOperation(nodeId1, attributeName1) con una DeleteAttributeOperation(nodeId2, attributeName2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y con el mismo nombre de atributo (identificados por attributeName1 y attributeName2). Esta situación puede darse en el contexto que detalla la Figura 5.5, en la cual, concurrentemente dos usuarios generan la misma operación de creación de atributo, pero, un tercer usuario, al recibir y ejecutar una de estas operaciones, genera una operación de eliminación de dicho atributo. Para este caso se establece que las operaciones de eliminación de atributos prevalecerán por sobre las de creación de los mismos. Consecuentemente y dado que el atributo ya fue eliminado, se devolverá como resultado de la transformación una instancia de la clase NoOperation.

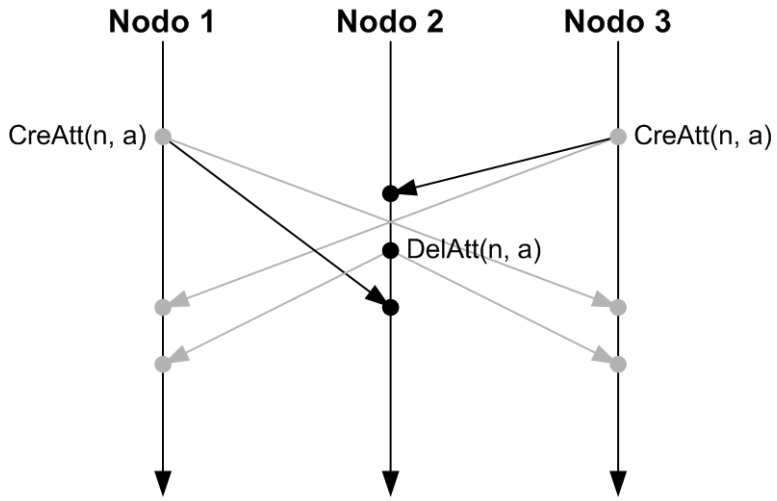

Figura 5.5. Caso conflictivo
CreateAttributeOperation - DeleteAttributeOperation.

En la transformación de una CreateAttributeOperation(nodeId1, attributeName1) con una ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y con el mismo nombre de atributo (identificados por attributeName1 y attributeName2). Este caso es similar al anterior y puede darse en la

misma circunstancia que aquel. Su representación en la Figura 5.6 muestra dos usuarios que generan concurrentemente la misma operación de creación de atributo, pero, un tercer usuario, al recibir y ejecutar una de estas operaciones, genera una operación de modificación de dicho atributo. Para este caso se establece que las operaciones de modificación de atributos prevalecerán por sobre las de creación de los mismos. Consecuentemente y dado que el atributo ya fue modificado, se devolverá como resultado de la transformación una instancia de la clase NoOperation.

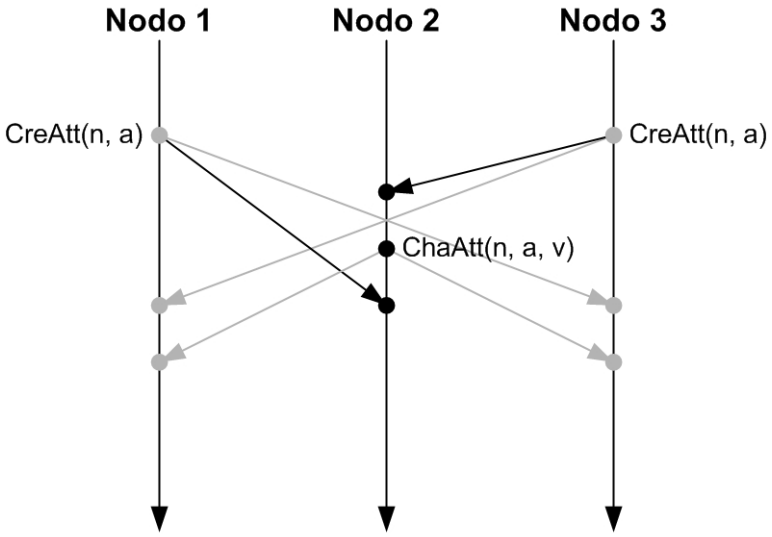

Figura 5.6. Caso conflictivo
CreateAttributeOperation - ChangeAttributeOperation.

En la transformación de una DeleteAttributeOperation(nodeId1, attributeName1) con una DeleteNodeOperation(nodeId2) las operaciones son conflictivas en el caso que el nodo eliminado (identificado por nodeId2) haya sido ancestro del nodo que contiene el atributo a eliminar (identificado por nodeId1). Este caso es similar a los tres primeros casos que analizamos. El conflicto emerge debido a que se pretende eliminar un atributo de un nodo inexistente. La solución natural en este caso, al igual que en los recién mencionados, es la devolución de una NoOperation como resultado de la transformación.

En la transformación de una DeleteAttributeOperation(nodeId1, attributeName1) con una DeleteAttributeOperation(nodeId2, attributeName2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y sobre el mismo atributo (identificados por attributeName1 y

attributeName2). En este caso el conflicto radica en que el atributo que se intenta eliminar con la operación a transformar, ya fue eliminado por una operación idéntica generada concurrentemente a ésta. Como resultado de la transformación, se devolverá una instancia de la clase NoOperation.
En la transformación de una ChangeAttributeOperation(nodeId1, attributeName1, attributeValue1) con una DeleteNodeOperation(nodeId2) las operaciones son conflictivas en el caso que el nodo eliminado (identificado por nodeId2) haya sido ancestro del nodo que contiene el atributo a modificar (identificado por nodeId1). Este caso es muy parecido a los tres primeros casos que se analizaron. El conflicto se debe a que se quiere modificar un atributo que ya no existe, debido a que fue eliminado por la operación antes mencionada. La solución natural en este caso, al igual que en los casos similares anteriores, es la devolución de una NoOperation como resultado de la transformación.

En la transformación de una ChangeAttributeOperation(nodeId1, attributeName1, attributeValue1) con una DeleteAttributeOperation(nodeId2, attributeName2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y sobre el mismo atributo (identificados por attributeName1 y attributeName2). Dado que no se podrá modificar un atributo que ha sido eliminado, el resultado de la transformación será una instancia de la clase NoOperation.

En la transformación de una ChangeAttributeOperation(nodeId1, attributeName1, attributeValue1) con una ChangeAttributeOperation(nodeId2, attributeName2, attributeValue2) las operaciones son conflictivas en el caso que sean sobre el mismo nodo (identificados por nodeId1 y nodeId2) y sobre el mismo atributo (identificados por attributeName1 y attributeName2). Este caso ocurre cuando dos usuarios deciden modificar el mismo atributo de un mismo nodo concurrentemente.

El hecho de que ambas operaciones sean del mismo tipo, impide establecer un criterio de prevalencia entre ellas, lo cual obliga a definir dicho criterio de alguna otra forma. Si para ello se tuvieran en cuenta, por ejemplo, los valores que se pretenden asignar al atributo, se estaría fijando una prioridad implícita entre los valores que pueden tomar los atributos de los nodos. Ésta no es una característica deseada ni tampoco natural.

Una solución alternativa a ésta e independiente de dichos valores u otros de las propiedades esenciales de las operaciones, es el establecimiento de prioridades. Así, como en la sección anterior hemos definido a la ChangeAttributeOperation como una operación priorizada, utilizaremos sus prioridades para definir cuál de los valores será el que prevalezca en el atributo. Dependiendo de ello, se devolverá como resultado de la transformación, una u otra operación.

5.2. Abstracción de casos conflictivos

Como se ha podido observar en la amplia mayoría de los casos conflictivos analizados (todos menos el último), se devuelve como resultado de la transformación una instancia de la clase NoOperation. Con el objetivo de concentrar este comportamiento común en una sola clase abstracta, se creó la clase IxeTransformer. Esta clase extiende la clase abstracta Transformer e implementa el método *transform()*. En éste, utilizando el patrón *Template Method* [Gamma], delega a sus subclases la responsabilidad de definir si devolverá la operación original o una instancia de la clase NoOperation.

En la Figura 5.7 se pueden observar los transformadores que fue necesario crear para la aplicación.

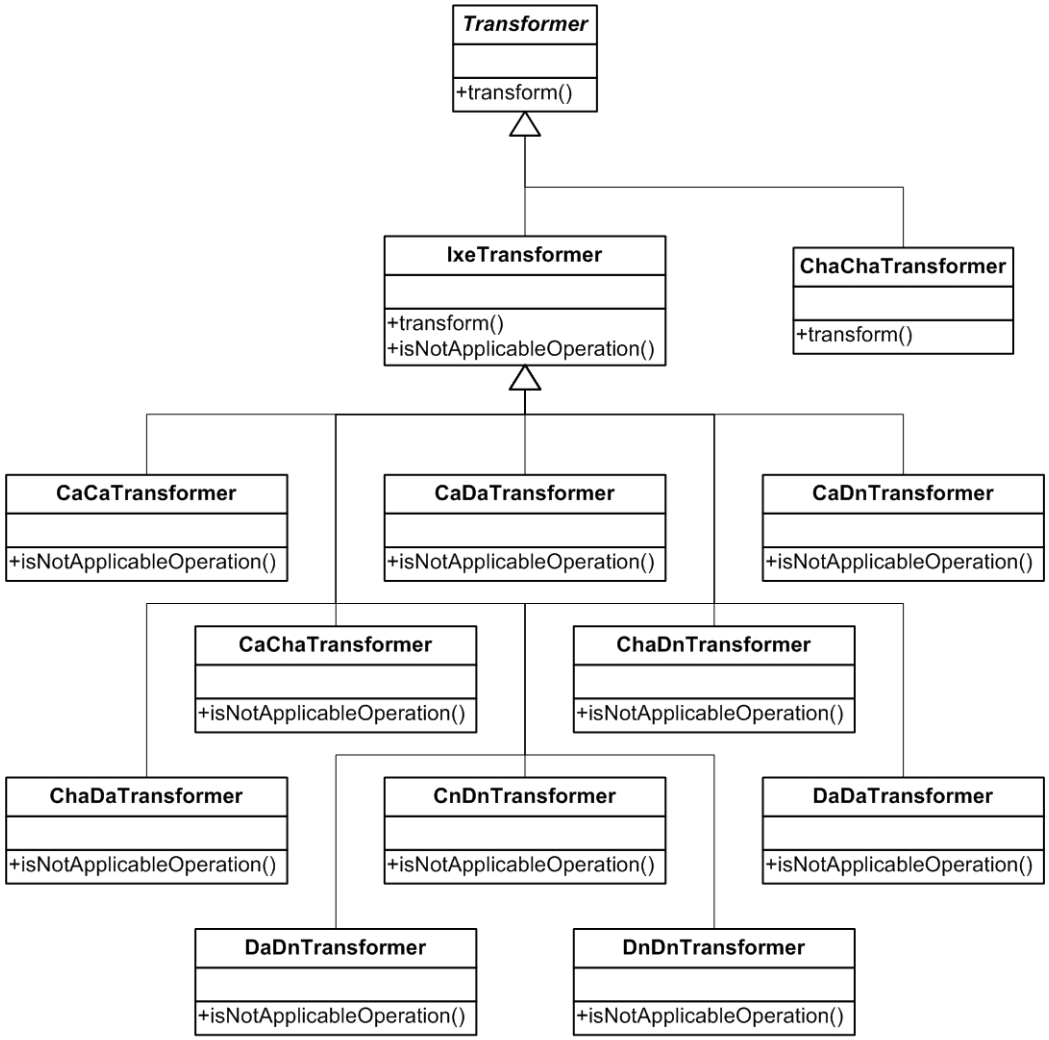

Figura 5.7. Jerarquía de transformadores.

5.3. Combinaciones no conflictivas de operaciones

Cómo se explicó en el capítulo 4, para los casos de combinaciones no conflictivas de operaciones, no se requiere la creación de transformadores. Es por ello que solo se cuenta con once transformadores y no con veinticinco que es el número de combinaciones posibles de tipos de operaciones.

6. Integración con Hipp

Como toda aplicación que se desarrolle sobre *Hipp*, IXE se relaciona con las tres capas públicas que dicho *framework* provee, la Capa de abstracciones de dominio, la Capa de mantenimiento de consistencia y la capa de interfaz de comunicación. La integración entre IXE y estas tres capas se da de diferentes formas y entre diferentes objetos. En esta sección se detallan las relaciones establecidas entre IXE y cada una de dichas capas, describiendo la forma, las clases y los objetos involucrados.

IXE – *Hipp*, Capa de abstracciones de dominio

IXE se integra con la Capa de abstracciones de dominio de Hipp, extendiendo las distintas clases de operaciones y transformadores provistas por este último y definiendo o redefiniendo los métodos necesarios.

Comenzando por las operaciones, la clase IxeOperation extiende la clase Operation según el criterio explicado en el capítulo anterior. En ella, solo se implementa el método abstracto *execute()* y se define uno nuevo para adaptar, el modelo sobre el que se ejecutan las operaciones, al modelo provisto por IXE. Siguiendo el mismo criterio y similar implementación, la clase IxePriorizedOperation extiende a la clase PriorizedOperation de *Hipp*. Alguna de estas dos subclases de operaciones definidas por IXE, será la superclase de las operaciones concretas de la aplicación.

En cada una de las operaciones concretas de IXE, se debió definir el método de ejecución de la operación y los métodos que devuelven los conjuntos de transformadores correspondientes a cada una.

En el rubro de los transformadores, IXE extiende la clase Transformer de Hipp con el transformador abstracto IxeTransformer y con el concreto ChaChaTransformer. El primero define un algoritmo abstracto de obtención de la operación transformada, en el cual participan sus subclases. Mientras que el segundo, por no compartir el dicho algoritmo con el primero, define el suyo propio.

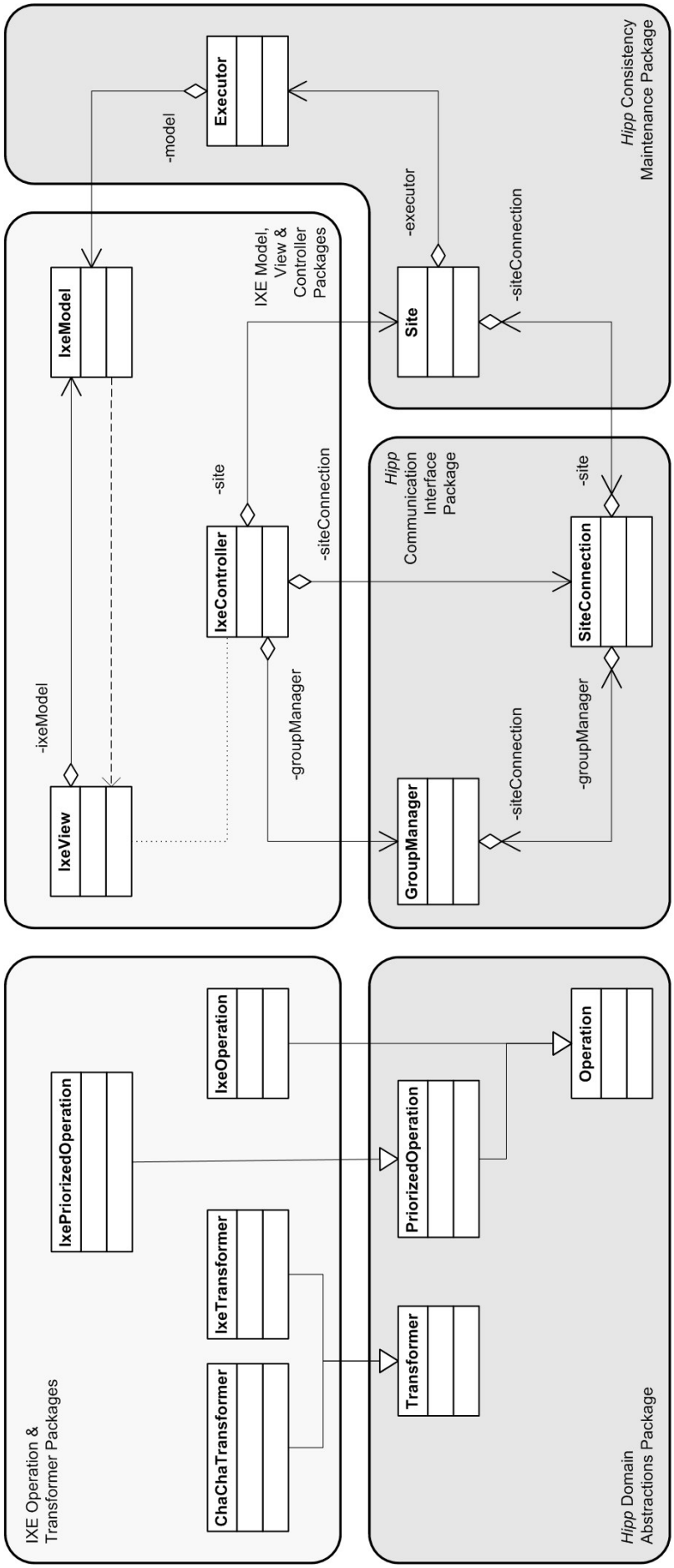

Figura 5.8. IXE - Integración con *Hipp*.

6.1. IXE – *Hipp*, Capa de mantenimiento de consistencia.

La integración entre IXE y la Capa de mantenimiento de consistencia de *Hipp*, involucra a las clases IxeModel e IxeController por el lado de IXE y a las clases Site y Executor por parte de *Hipp*. Las relaciones en este caso son de creación, conocimiento y/o colaboración.

El IxeController, en su constructor, se encarga, entre otras cosas, de la creación del Site. Al hacerlo, debe pasarle como parámetro el modelo sobre el cual se ejecutarán tanto las operaciones locales como las remotas. El modelo pasado como parámetro por el IxeController es una instancia de la clase IxeModel.

Al momento de la ejecución de una operación, el Executor invocará sobre ésta el método *execute()*, pasándole como parámetro la instancia del IxeModel mencionada anteriormente.

Después de crear el Site, el IxeController lo almacena en una variable de instancia, para luego utilizarlo al momento de enviar las operaciones generadas localmente.

6.2. IXE – Hipp, Capa de interfaz de comunicación

La integración entre IXE y la capa de interfaz de comunicación de *Hipp* comprende a la clase IxeController por parte de IXE y a las clases GroupManager, JxtaGroupManager y SiteConnection de parte de *Hipp*. Al igual que en el caso anterior, las relaciones en este caso son de creación, conocimiento y/o colaboración.

En su constructor, el IxeController, crea la instancia del SiteConnection pasándole como parámetro una nueva instancia de la clase JxtaGroupManager, la cual es subclase de la clase abstracta GroupManager. Luego de esto, el IxeController almacena ambos objetos recién creados en dos variables de instancia. Las instancias del SiteConnection y del GroupManager podrán ser utilizadas más tarde para llevar a cabo las funciones de conexión y administración de grupos.

7. Testing

Durante el desarrollo de las funciones de transformación se verificó que estas cumplieran tanto con la propiedad TP1 como con la propiedad TP2. Para ello se realizaron tests de unidad (utilizando el *framework* JUnit) en los que se definieron cada uno de los casos relevantes a cada propiedad.

Con el objetivo de hacer *test friendly* a IXE, se debieron definir una serie de clases y métodos que se detallarán a continuación:

- Se definieron *mocks* de las operaciones y los transformadores que permitieran verificar el funcionamiento de los transformadores sin la necesidad de utilizar el software de comunicación.
- Utilizando *overloading*, se definieron métodos *equals()* que no sobrescribieran el provisto por la clase Object. Estos métodos, en vez de recibir un Object como parámetro reciben un objeto de misma la clase en la cual están definidos.
- Se redefinió el método *toString()* en la clase Node y en cada una de las operaciones para facilitar el *debugging* de la aplicación.

8. Conclusiones

El desarrollo de esta aplicación demostró que es posible la utilización del *framework Hipp* para la implementación de sistemas groupware. Se pudo comprobar con una aplicación simple y cuyas funciones de transformación, se sabe, son correctas, que la utilización de *Hipp* resulta sumamente sencilla.

La simpleza del sistema elegido permitió enfocarse en la verificación del funcionamiento de Hipp y no en los problemas propios del dominio de la aplicación que presenta cada sistema que se desarrolle.

Capítulo 6

Conclusiones y perspectiva

1. Conclusiones

Para cumplir con el objetivo de esta tesis (crear un *framework* de manejo de consistencia para el desarrollo de *groupware sincrónico* con arquitectura de coordinación distribuida), se investigó la temática del *groupware* y su variante sincrónico. Se hallaron sus características deseables, sus posibles arquitecturas y se analizaron las diferentes variantes de mecanismos de mantenimiento de consistencia disponibles. Se pudo establecer que la principal causa del surgimiento de inconsistencias en la información compartida se basa en la latencia de la red y en los tiempos de encolado y procesamiento. Como resultado de esta investigación y análisis, se escogió la utilización de la técnica de *Operational Transformation* para el mantenimiento de la consistencia de la información compartida.

Se realizó un relevamiento del estado del arte de *Operational Tranformation* teniendo en cuenta su historia, objetivo, conceptos, estructuras de datos, funcionamiento, características y una gran variedad de algoritmos. Estos algoritmos fueron comparados y clasificados según sus características particulares con el propósito de elegir uno para su implementación.

Una vez escogido el algoritmo de *Operational Transfromation* a utilizar, se analizó la viabilidad de su encapsulamiento en un *framework* con el objetivo de aprovechar sus ventajas, generalizar su uso en diferentes dominios de aplicación y evitar el desarrollo reiterado de soluciones a los mismos problemas. Dicho análisis resultó satisfactorio, con lo cual, se procedió al diseño de *Hipp*, un *framework* para el desarrollo de aplicaciones *groupware* sincrónicas *peer to peer* enfocado en el mantenimiento de la consistencia de la información compartida.

Hipp utiliza modelos replicados sobre una arquitectura *peer to peer*. Esto significa que cada uno de los nodos contará con una copia del modelo de objetos de la aplicación y que no existirán entidades distinguidas en su arquitectura.

Internamente en cada nodo, *Hipp* se organiza en cuatro capas con diferente objetivo cada una. La primera se encarga de abstraer el comportamiento común de los objetos de dominio que *Hipp* necesita para su funcionamiento. La segunda alberga la implementación del algoritmo de *Operational Transformation* mientras que la tercera hace de interfaz con el software de comunicación. Por último, la cuarta capa está formada por el software de comunicación a utilizar. Estas capas se comunican con sus adyacentes vía interfaces o clases abstractas. El diseño en capas de *Hipp* permite el reemplazo de una o varias de ellas sin afectar el funcionamiento integral del *framework* ni el desempeño de las otras capas.

Tomando el diseño detallado se procedió a realizar una implementación de referencia de dicho *framework*. La implementación se llevó a cabo sobre el lenguaje Java y utilizando diversas herramientas de distribución gratuita (Eclipse, Freepository, etc.). En ella se utilizó JXTA como *software* de comunicación, lo cual demandó la implementación de las subclases adaptadoras para tal efecto.

Para completar el desarrollo del *framework*, se realizó la guía de uso de *Hipp*, en la cual se describe paso a paso como desarrollar una aplicación sobre este *framework*, además de una serie de buenas prácticas. La guía está organizada en tres secciones de acuerdo a las tres etapas definidas para la implementación de estas aplicaciones, Extensión, Inicialización y Uso.

Con el fin de complementar la guía de uso y evaluar el comportamiento de *Hipp*, se diseñó e implementó un ejemplo de uso del *framework*. El ejemplo de uso consiste en un editor colaborativo de XML llamado IXE. En el presente informe se detalla su diseño e implementación indicándose como se aplicó cada uno de los pasos descriptos en la guía de uso de *Hipp*.

IXE fue seleccionado para su implementación con la intención de probar el funcionamiento de *Hipp* con un prototipo sencillo, que presente una cantidad de casos de transformación razonable y que se cuente con la definición de las operaciones y funciones de transformación. Su desarrollo desencadenó la implementación de algunas de las clases que ayudan en la estructuración de los transformadores, contribuyendo así al refinamiento y enriquecimiento de la funcionalidad del *framework*.

2. Perspectiva

A continuación se presentarán una serie de posibles extensiones a este trabajo de tesis. Varias de ellas fueron tenidas en cuenta en el diseño del *framework* para que en caso de considerarlas importantes, su desarrollo sea más sencillo.

2.1. Latecomers

Hipp en esta versión requiere que todos los usuarios compartan el mismo estado inicial de aplicación. Es decir, que los mismos comiencen a interactuar sobre un mismo modelo inicial. Esto implica que los usuarios deban esperar a que todos los miembros de la futura sesión de colaboración inicien sus aplicaciones antes de comenzar a interactuar.

Esta limitación existe debido a que la única comunicación entre los nodos consiste en operaciones que no contienen el estado actual del modelo sobre el que se colabora, sino que solo posee los datos esenciales a la acción a realizar.

Una alternativa de implementación de esta funcionalidad consiste en permitir a los nodos la solicitud y envío de acciones históricas, es decir las enviadas con anterioridad. Así, los nodos de los usuarios que se acoplen a una sesión una vez que ésta ya está

iniciada (*latecomers*), comenzarán con un proceso de sincronización en el cual pedirán al resto de los nodos las operaciones históricas y las ejecutarán como se hace normalmente en *Hipp*. La complejidad de esta alternativa consiste en determinar en qué momento se han recibido todas las operaciones históricas y se finaliza el proceso de sincronización, para permitir al usuario interactuar con el resto de los miembros de la sesión. Esta complejidad da cabida a otra posible extensión llamada *Quiescence* que se detalla más adelante.

La alternativa anterior resultará incómoda si los usuarios se acoplan a las sesiones una vez que se haya enviado una gran cantidad de operaciones desde su comienzo. Para los casos en los que se considere necesario permitirlo, existe otra alternativa de soporte. Ésta consiste en la copia del modelo de un nodo ya incluido en la sesión, hacia el nodo que se está acoplando. Esta alternativa también requiere del establecimiento de un estado de reposo (*quiescent*) en el cual se considere que todos los modelos de todos los nodos incluidos en la sesión son idénticos. En algunos casos, los modelos sobre los que se colabora pueden ser grandes, esto significará una mayor dificultad para su envío por la red.

De todas formas corresponderá al implementador de esta extensión la elección de una de estas alternativas, ambas u otras no incluidas aquí de acuerdo a las características de la problemática que se requiera resolver.

2.2. Deshacer y rehacer operaciones

Resultaría interesante poder contar con las funcionalidades de deshacer y rehacer operaciones en forma nativa en el *framework*. En esta versión, *Hipp* no lo provee, pero como se describió en el capítulo 3, el algoritmo de *operational transformation* seleccionado (*adOPTed* [Ressel]) lo soporta [Gunzenhäuser].

2.3. Integración de componentes básicos

Muchas aplicaciones *groupware* están compuestas por varias herramientas *groupware* básicas, como puede ser un chat, un componente de *awarness*, un repositorio

compartido, etc. Estas herramientas pueden ser desarrolladas como componentes los cuales podrían ser reutilizados en diferentes aplicaciones.

Esta posible extensión propone realizar un estudio de cuáles herramientas *groupware* básicas sería interesante desarrollar como componentes teniendo en cuenta a que nicho de aplicaciones se orienta *Hipp* y desarrollarlas integrándolas con el *framework*.

2.4. Quiescence

Esta posible extensión consiste en desarrollar una funcionalidad que permita determinar cuando una aplicación *Hipp* se encuentra en estado de reposo (quiescent, ver capítulo 2). Que un sistema *Hipp* se encuentre en estado de reposo implica que todas las operaciones generadas han sido ejecutadas en todos los nodos, es decir, no hay requerimientos en tránsito o esperando por ser ejecutados en ningún nodo.
Esta extensión resulta de particular interés debido a que su solución no resulta trivial, ya que el espectro de aplicaciones al que está orientado Hipp propone una gran participación de los usuarios y por consiguiente un gran caudal de operaciones. Esto hace que los estados de reposo duren pequeñas fracciones de tiempo en las contadas ocasiones en las que se producen.

2.5. Soporte de otro software de comunicación

Hipp fue diseñado independiente del software de comunicación a utilizar, pero para su prueba y uso se provee la integración con el *framework* JXTA. La integración con otro software de comunicación permitiría comprobar realmente su independencia, comparar los desempeños y presentar al usuario de *Hipp* alternativas de comunicación que puedan ajustarse a sus requerimientos.

2.6. Evolución en el desarrollo de funciones de transformación

Sería deseable contar con técnicas que permitan simplificar la concepción, implementación y prueba de las funciones de transformación y la factorización de sus casos. Las funciones de transformación se observan como el punto de mayor complejidad en el desarrollo de una aplicación con *Hipp*.

2.7. Desarrollo de aplicación de mayor complejidad

El desarrollo de una aplicación *groupware* de mayor escala que el editor colaborativo de XML que se provee en el capítulo Ejemplo de Uso, permitiría encontrar puntos críticos tanto en el desarrollo como en el funcionamiento de aplicaciones que utilicen *Hipp*.

2.8. Implementación en otros lenguajes de programación

La implementación de *Hipp* en otros lenguajes de programación, permitiría abrir el abanico de posibles usuarios del *framework* ya que posibilitaría que aplicaciones *groupware* ya desarrolladas utilizando otro *framework* puedan reemplazarlo por *Hipp*. Dada su reciente expansión, debería considerarse también el desarrollo de una versión de *Hipp* optimizada para dispositivos móviles.

Referencias

ACE — Mark Bigler, Simon Räss, Lukas Zbinden. ACE a collaborative editor, Report evaluation algorithms.

Andrews — Gregory Andrews, “Foundations of Multithreaded, Parallel, and Distributed Programming” ISBN: 0201357526.

Burbeck — S. Burbeck, "Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)," 1992

Cormack 1 — Cormack, G. (1995): A Counterexample to the Distributed Operational Transform and a Corrected Algorithm for Point-to-Point Communication. University of Waterloo Technical Report, CS-95-08.

Cormack 2 — Cormack G.V., A Calculus for Concurrent Update (Abstract), Proc. 14th ACM Symposium on Principles of Distributed Computing (1995).

Deutsch — Peter L. Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In software Reusability, Volume II. Reading, MA: ACM Press / Adison Wesley 1989.

Ellis 1 C. A. Ellis, S. J. Gibbs. Concurrency control in groupware systems. Proceedings of the 1989 ACM SIGMOD international conference on Management of data, June 1989.

Ellis 2 Clarence A. Ellis , Simon J. Gibbs , Gail Rein, Groupware: some issues and experiences, Communications of the ACM, v.34 n.1, p.39-58, January 1991.

Fayad Fayad, Mohamed; Schmidt, Douglas; Johnson, Ralph. "Building application frameworks: object-oriented foundations of Framework design". New York: John Wiley & Sons, 1999.

Gamma Gamma, E., Helm, R., Johnson, J., Vlissides, J.: Design Patterns. Elements of reusable object-oriented software, Addison Wesley 1995.

Gunzenhäuser Rul Gunzenhäuser Matthias Ressel. Reducing the problems of group undo. Proceedings of the international ACM SIGGROUP conference on Supporting group work, 1999.

Hofte Ter Hofte, H., Working Apart Together – Foundations for Component Groupware, Telematica Instituut, Enschede, 1998.

Imine 1 A. Imine, P. Molli, G. Oster, M. Rusinowitch, Development of transformation functions assisted by a theorem prover, Fourth International Workshop on Collaborative Editing (ACM CSCW'02), Collaborative Computing in IEEE Distributed Systems Online, November 2002.

Imine 2 A. Imine, G. Oster, P. Molli and M. Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. May 2004.

Imine 3 A. Imine, P. Molli, G. Oster, M. Rusinowitch, Proving correctness of transformation functions in real-time groupware, Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work, September, 2003, Helsinki, Finland

Java http://java.sun.com/

Johnson Ralph E. Johnson and Brian Foote. Designing reusable classes. Journal of Object-Oriented Programming, 1988.

JUnit http://www.junit.org/

JXTA https://jxta.dev.java.net/

Lamport Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 1978

Li Rui Li, Du Li, and Chengzheng Sun. A time interval based consistency control algorithm for interactive groupware applications. In ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference on (ICPADS'04), Washington, DC, USA, 2004.

Lushman Lushman, B. and Cormack, G. V. Proof of correctness of Ressel's adOPTed algorithm. Inf. Process. Lett. 86, 6 (Jun. 2003).

Naso F. Naso, F. García, D. De Sogos, R. Tesoriero, A. Fernández. 7mo Congreso Argentino de Ciencias de la Computación. El Calafate. Argentina 2001.

Nichols David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. Proceedings of the 8th annual ACM symposium on User interface and software technology, 1995.

Oram A. Oram, Peer-toPeer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Inc., 2001.

Pree Pree, W. Design Patterns for Object-Oriented Software Development. Reading, MA: Adisson-Wesley, 1995.

Quiroga Leandro Quiroga and Alejandro Fernandez. "A P2P Groupware Framework based on Operational Transformations". In Proceedings of the 27th international Conference on Distributed Computing Systems Workshops (ICDCSW). June 22 - 29, 2007. Toronto, Canada.

Ressel Matthias Ressel , Doris Nitsche-Ruhland , Rul Gunzenhäuser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, Proceedings of the 1996 ACM conference on Computer supported cooperative work, November 16-20, 1996, Boston, Massachusetts, United States

Schuckmann C. Schuckmann, L. Kirchner, J. Schummer, J. Haake. Designing object-oriented synchronous groupware with COAST. Proceedings of Computer Supported Collaborative Work CSCW, 1996.

Suleiman 1 Maher Suleiman, Michele Cart, and Jean Ferrie. Serialization of concurrent operations in a distributed collaborative environment. Proceedings of the international ACM SIG-GROUP conference on Supporting group work: the integration challenge, 1997.

Suleiman 2 Maher Suleiman, Michele Cart, and Jean Ferrie. Concurrent operations in a distributed and mobile collaborative environment. Proceedings of the Fourteenth International Conference on Data Engineering, 1998.

Sun 1 Chengzheng Sun , Xiaohua Jia , Yanchun Zhang , Yun Yang , David Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, ACM Transactions on Computer-Human Interaction (TOCHI), March 1998

Sun 2 Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. Proceedings of the 1998 ACM conference on Computer supported cooperative work, 1998.

Tietze D. A. Tietze, J. Rubart. 2001. http://www.go4teams.com/papers/dyce_gt.pdf.

Vidot Nicolas Vidot, Michelle Cart, Jean Ferrie and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. Proceedings of the 2000 ACM conference on Computer supported cooperative work, 2000.

Wirfs-Brock Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. Communications of the ACM 33(9), 1990.

XStream http://xstream.codehaus.org/

Zafer Ali A. Zafer, Clifford A. Shaffer, Roger W. Ehrich, and Manuel Perez. Netedit: A collaborative editor. 2001.