CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL
UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

Un sistema flexible y extensible de entidades replicadas para aplicaciones cooperativas distribuidas

Tesis que presenta:

Act. Maria Guadalupe Pérez Medina Montaño

Para obtener el grado de:

Maestra en Ciencias en Computación

Directores de Tesis:
Dr. José Guadalupe Rodríguez García /
Dr. Dominique Decouchant

México, D. F. Marzo, 2010

Agradecimientos

Agredecezco al Consejo Nacional de Ciencia y Tecnología por el apoyo ecoómico que me permitió realizar la maestría en ciencias de la computación.
También deseo agradecer al Cinvestav por brindarme la oportunidad de continuar con mis estudios y a sus investigadores por compartir sus conocimientos conmigo durante mi estancia en esta institución.
Especialmente, deseo agradecer a mi asesor, el Dr. Dominique Decouchant por sus consejos y enseñanzas durante todo el periodo de estudio de la maestría, por ayudarme cada vez que tenía problemas y guiarme cuando perdía el rumbo, siendo no sólo un gran maestro sino un buen amigo.
Quiero agradecer a mi asesor, el Dr. Guadalupe Rodríguez García por sus comentarios y ayuda, haciendo posible concluir mis estudios.
Agradezco a mis revisores, la Dra. Sonia Guadalupe Mendoza Chapa y la Dra. Lizbeth Gallardo por sus críticas tan asertivas, las cuales me permitieron mejorar considerablemente este trabajo de tesis.
Nunca debe faltar un agradecimiento a Sofia Reza por su apoyo y su trato siempre amable.

Dedico este trabajo de investigación a mis padres: Rafael Pérez Medina Ollivary y Jovita Montaño Ceseña, no sólo por otorgarme el don de la vida, sino por impulsarme a cumplir mis sueños y apoyarme de manera incondicional en su realización.
Con mucho amor dedico esta tesis a mis hijos: Alejandro y Ernesto, por hacer que valga la pena el esfuerzo y darle sentido a mi vida. Por rodearme con sus brazos cuando estaba cansada y decirme: “tu puedes mamá”.
Quiero dedicar este trabajo a mis hermanos: Jovita, Adriana, Miriam y Rafael, por compartir las experiencias de la vida. Quienes a pesar de todas las discusiones y desacuerdos normales entre hermanos, no dejan de ser increibles.

No quiero dejar de aprovechar la oportunidad sin agradecer a mis amigos, con quienes he pasado momentos maravillosos y algunos no tan agradables. Principalmente dedico esta tesis a Leo, amiga de toda la vida. A Mario con quien he compartido tanto momentos agradables como difíciles. A Moni quien a pesar de la distancia se mantien cerca de mí. A Pam por la confianza y amistad que me brinda. A Beto por hacer interesante mi vida con sus locuras. A Pau y Johny por su agradable compañia. A Migue por su cariño y buen humor. A Pepe, quien a pesar de todo, siempre está cuando se le requiere.

Simplemente con mucho cariño,

Gracias.

Contents

Índice de Figuras **1**

Resumen **1**

Abstract **3**

1 Introducción **5**

2 Antecedentes y motivación **13**
2.1 Definiciones y principios de la colaboración asistida por computadoras . 14
2.1.1 Definición de *groupware* o software de trabajo colaborativo 14
2.1.2 Control de concurrencia 16
2.1.3 Distribución/notificación de las modificaciones de datos compartidos 20
2.1.4 Protocolos de comunicación 27
2.2 Aplicaciones colaborativas y la colaboración en la Web 29
2.2.1 Juegos cooperativos 29
2.2.2 Mensajería instantánea 33
2.2.3 Agendas colaborativas 34
2.2.4 Editores cooperativos 37
2.2.5 Aplicaciones cooperativas para la educación a distancia 43
2.2.6 Síntesis de tipos de aplicaciones cooperativas 46
2.2.7 Soluciones utilizadas actualmente 48
2.3 Una arquitectura de trabajo cooperativo flexible y extensible 49
2.3.1 Arquitectura general de la plataforma propuesta 50

3 Objetos flexibles y actualizaciones dinámicas de software **55**
3.1 Técnicas que permiten hacer los objetos flexibles 56
3.1.1 La programación orientada a aspectos 56
3.1.2 Patrones de diseño 57
3.2 Actualizaciones dinámicas de software 62
3.2.1 El problema de actualizar un sistema en tiempo de ejecución . . . 62
3.2.2 Actualizaciones dinámicas y objetos flexibles 64
3.2.3 Control de concurrencia de las instancias de las clases 64

3.2.4 Transformación del estado 66
3.2.5 Actualizaciones dinámicas en sistemas distribuidos 66
3.2.6 Control de la actualización 66
3.2.7 Actualizaciones dinámicas como un aspecto 67
3.2.8 El patrones de diseño vs. la programación orientada a aspectos . 67

4 Diseño de la plataforma flexible y extensible 69
4.1 Diseño de las entidades replicadas distribuidas 73
4.1.1 Requerimientos de la entidad 73
4.1.2 Atributos de la entidad 74
4.1.3 Métodos de la entidad 77
4.1.4 Proxies de las entidades 80
4.2 Diseño de los módulos funcionales de la plataforma 82
4.2.1 Sistema de actualizaciones dinámicas de aplicaciones `DAUS` (*Dinamical Application Updating System*) 86
4.3 Actualizador de versiones de software 90
4.3.1 Características del actualizador 91
4.3.2 Diseño del actualizador 92
4.3.3 Proceso de actualización 92

5 Caso de estudio: diferentes aplicaciones colaborativas distribuidas 95
5.1 Las aplicaciones colaborativos de tipo chat 95
5.1.1 Entidades de control de acceso 95
5.1.2 Entidades de contactos de los usuarios 98
5.1.3 Entidades de colas de mensajes 100
5.1.4 Proceso en los chats 101
5.2 Editores cooperativos asíncronos de documentos 106
5.2.1 Entidades de control de acceso 106
5.2.2 Entidades de documentos 107
5.2.3 Operaciones de edición y procesos en la plataforma 108
5.3 Agendas colaborativas 114
5.3.1 Entidades de las Agendas 114
5.3.2 Entidades de control de acceso 116
5.3.3 Operaciones y procesos en las agendas 117

6 Implementación y pruebas del modelo 123
6.1 Beneficios de Ruby y su *API* de reflexión 123
6.2 Implementación de la plataforma 127
6.2.1 Plataforma mínima 128
6.2.2 Control de acceso 128
6.2.3 Administrador de entidades 129
6.2.4 *Proxy* de entidades remotas 129
6.2.5 API (*Application Program Interface*) 130

6.2.6 DAUS (*Dinamical Application Updating System*) 130
6.2.7 EUS (*Entity Update System*) 131
6.3 Entidad genérica 132
6.4 Entidades y políticas implementadas 132
6.4.1 Tipos de entidades 132
6.4.2 Políticas de control de acceso 134
6.4.3 Políticas de control de concurrencia del EUS 135
6.5 Desarrollo de una aplicación colaborativa usando la plataforma 136
6.5.1 *Chat room* 136
6.5.2 Pruebas de la entidad de tipo de "archivo" 137
6.6 Implementación del actualizador de versiones de software 138
6.7 Pruebas de flexibilidad y actualizaciones dinámicas 140
6.7.1 Agregación de un nuevo tipo de entidad 140
6.7.2 Modificación de la política de control de acceso de una entidad . 141

7 Conclusiones y trabajo a futuro **143**
7.1 Conclusiones 144
7.1.1 Ventajas del modelo propuestos 145
7.1.2 Deficiencias y limitaciones de la solución propuesta 146
7.2 Trabajo a futuro 146
7.2.1 Plataforma 147
7.2.2 Entidades distribuidas y políticas de administración 147
7.2.3 Aplicaciones 147
7.2.4 Actualizaciones dinámicas 148

Lista de Figuras

2.1 Modificaciones concurrentes 18
2.2 Arquitectura centralizada 21
2.3 Arquitectura completamente replicada 23
2.4 Arquitectura híbrida 24
2.5 Arquitectura de juegos por turnos 30
2.6 Arquitectura de juegos colaborativos en ambiente compartido 31
2.7 Juego *World of Warcraft* 32
2.8 Arquitectura de chat 34
2.9 Mensajería Instantánea 35
2.10 Arquitectura de distribución asociada a una agenda 36
2.11 Calendario de MAPilab 37
2.12 Arquitectura para editores de documentos 38
2.13 Arquitectura de PIÑAS 43
2.14 Educación a distancia: la aplicación Lyceum 45
2.15 Arquitectura de la plataforma 51
2.16 Arquitectura de la entidad 52

3.1 Arquitectura tradicional *vs.* POA figura obtenida de [37] 56
3.2 Patrón de diseño template 58
3.3 Patrón de diseño estrategia 59
3.4 Patrón de diseño *facade* 59
3.5 Diseño de patrón *observer* 60
3.6 Adaptador 61
3.7 Patrón de diseño *proxy* 61
3.8 Actualización de entidades 63
3.9 Actualización de instancias 65

4.1 Proceso de las aplicaciones cooperativas 71
4.2 Integración de los distintos elementos de la plataforma genérica por medio de la plataforma mínima 72
4.3 Estructura de la entidad genérica propuesta 74
4.4 Procesamiento de una llamada remota via un proxy 81
4.5 Diagrama de clases de la plataforma flexible y extensible 83

4.6 Actualización dinámica de las clases 88
4.7 Actualizaciones dinámicas de entidades 90
4.8 Proceso de actualización 93

5.1 Entidad de control de acceso de usuarios en el sitio D 96
5.2 Entidad de control acceso de usuarios en el sitio E 96
5.3 Arquitectura manejada para el control de acceso 97
5.4 Entidad de control de acceso de los administradores 98
5.5 Entidad de contactos de los usuarios 99
5.6 Arquitectura de contactos de los usuarios 99
5.7 Cola de mensajes entre los sitios A y B 100
5.8 Arquitectura de la cola de mensajes entre los sitios A y B 101
5.9 Proceso de validación y conexión de un usuario 102
5.10 Proceso de envío de mensajes entre los usuarios 104
5.11 Proceso de desconexión de los usuarios 105
5.12 Entidad de control de acceso para los editores de texto 106
5.13 Entidad de un editor de texto 107
5.14 Arquitectura utilizada para las entidades de los documentos 108
5.15 Proceso de conexión de un usuario 109
5.16 Proceso que permite subir un documento a memoria 110
5.17 Proceso que permite salvar un documento en un archivo 110
5.18 Proceso que permite realizar una copia de un documento 111
5.19 Proceso que permite eliminar de un documento 111
5.20 Proceso mediante el cual se obtiene el candado de un fragmento 112
5.21 Proceso que permite realizar una operación en un(os) fragmento(s) de una entidad 113
5.22 Arquitectura de la entidade tipo *agenda* 115
5.23 Agenda de un usuario. 115
5.24 Agenda del Jefe 116
5.25 Réplicas en el servidor de coordinación 117
5.26 Entidades de control de acceso: Users y Contactos de Victor 118
5.27 Proceso de conexión y sincronización de los datos de la agenda 119
5.28 Proceso de una operación en una entidad de acceso 120
5.29 Proceso de operaciones de citas en las entidades tipo agenda 121

6.1 Flujo de información entre los módulos funcionales 128
6.2 Procesamiento de una llamada remota via un proxy 130
6.3 Validación de un usuario utilizando listas de acceso 135
6.4 Interfaz de usuario del chat room 136
6.5 Flujo de los mensajes entre los usuarios del *chat room* 137
6.6 Instrucciones en la plataforma 138
6.7 Operaciones realizadas con la entidad tipo archivo 139
6.8 Pruebas de actualizaciones dinámicas 140

Resumen

En nuestros días, existen aplicaciones que tienen como meta hacer la colaboración entre los usuarios más fácil, cambiando la forma en que las personas se comunican. Este tipo de aplicaciones son conocidas como *groupware*. Sin embargo, la mayoría de estas aplicaciones utilizan una arquitectura para la distribución de los datos híbrida o centralizada. Esto lleva a un pobre desempeño de la aplicación, la falta de tolerancia a fallos y la pérdida o inconsistencia de los datos compartidos distribuidos.
Algunas investigaciones tratan de mejorar el desempeño de estas aplicaciones utilizando replicación de los datos, pero sólo implementan un esquema de replicación y usan sólo una política de control de concurrencia. No utilizan el conjunto de políticas que se adapte mejor a la naturaleza de la aplicación, las condiciones predominantes de la red y el ambiente de los usuarios. Por otro lado, no existe un conjunto de políticas que se ajuste completamente a ambientes poco fiables, donde las condiciones de conectividad cambian constantemente. Más aún, cada tipo de datos utilizado en las aplicaciones colaborativas deben tener un conjunto diferente de políticas (*e.g.* control de concurrencia, arquitectura de distribución, replicación, sincronización, *etc.*) de acuerdo a la naturaleza de la aplicación, la sensibilidad de los datos a la latencia de las notificaciones, las condiciones de la red y el ambiente del usuario.

Por estas razones, proponemos una plataforma flexible y extensible de entidades compartidas para aplicaciones colaborativas distribuidas. Esta plataforma fue diseñada para definir e integrar nuevas entidades distribuidas, basadas en una entidad genérica. Los desarrolladores de aplicaciones colaborativas pueden elegir entre diferentes estrategias de replicación, sincronización, control de acceso, *etc.*, haciendo la conceptualización y el desarrollo más sencillo. Esto se traduce como un mejor desempeño de las aplicaciones y permite proporcionar más failidades a los usuarios finales (e.g. los usuario pueden trabajar en modo desconectado, las aplicaciones pueden ser tolerantes a fallas, tienen un mejor tiempo de respuesta, entre otros). Esta plataforma también permite a los programadores de aplicaciones colaborativas utilizar la misma entidad compartida para diferentes aplicaciones o la misma política para diversas entidades. Más aún, se pueden crear nuevas entidades a partir de entidades ya definidas. Así, el código de software puede reusarse haciendo el desarrollo de las aplicaciones colaborativas mas sencillo y rápido.
Otra ventaja de esta plataforma es un sistema de actualizaciones dinámicas de software, el cual controla y distribuye las actualizaciones de software en tiempo de ejecución. De esta manera, es posible agregar nuevas entidades, modificar las existentes, actualizar sus instancias y las políticas asociadas dinámicamente, i.e., sin detener la plataforma.

Abstract

Nowadays, there are applications which their goal is doing collaboration between users easier, changing the way people comunicate each others. This kind of applications are known as "groupware". However, mostly of this applications use a centralized or hybrid data distribution architecture. This leads to a poor application performance, a lack of fault tolerance or disconnections and loss or inconsistencies in the distribuited sharing data.
Some researchs are trying to get better application performance using data replication, but they only use one kind of replication scheme and use only on concurrency policy. They do not use the set of policies (*i.e.* concurrency, distribution architecture, replication, synchronization, *etc*.) which adapt better to the application nature, the predominant network conditions and the users environment. On the other hand, there is not a well-fitting set of policies for unreliable environmets such as Internet or mobile/nomadic work where the users environment change constantly. Moreover, every kind of data in collaborative applications should have a different set of policies. according to the application nature, sensitivity of the data to the latency deliver, network conditions and user environment (e.g. such as Internet or mobile/nomadic work).

For this reasons, we propose a flexible extensible platform using share entities for collaborative distributed application. This platform was designed to define and integrate new distributed entities, based on a generic entity. The collaborative application programmers can choose bewteen different strategies of replication, synchronization, access control, *etc*, doing the application conceptualization and development easier. This may be traslated as a better application performance and more facilities for the final user (*e.g.* an application could support network disconnections, fault tolerance, a better response time and others). This platform also allows to the colaborative applications programmers use the same entity for different applications or use the same policy for diferent entities. Morover, it is posible to create new entities from one already define. So, the software code could be reuse doing collaborative application implementation simpler and faster.

Another interesting issue of this platform is a dynamical software updating system, which was created to control and distribute software updatings dynamically. On this way, it is posible to add new entities, modify entities already defined, update their instances and their policies on the fly, *i.e.* without stopping the platform.

Capítulo 1

Introducción

Con el desarrollo de las telecomunicaciones y de los sistemas distribuidos, así como la aparición de dispositivos inalámbricos más poderosos en los últimos años, ha sido posible desarrollar aplicaciones móviles en el ámbito del trabajo colaborativo asistido por computadora. Estos avances permitieron implementar diferentes tipos de aplicaciones orientadas a usuarios distribuidos geográficamente que desarrollan trabajos con una meta en común. Los editores cooperativos constituyen ejemplos representativos de este tipo de aplicaciones (*e.g. Cebos o Alliance*), las agendas colaborativas (*e.g. Team-Integrator, Carthago, Horde, Outlook, MAPilab*), aplicaciones de educación a distancia (*e.g. Lyceum*), juegos en línea (*e.g. Halo, World of warcraft, Quake*), mensajería instantanea (*e.g. Exodus, Kopete, aMSN, gaim, Messenger*), entre otros.
Cada tipo de aplicación tiene requerimientos diferentes de acuerdo a sus características. Por ejemplo, la latencia de las notificaciones de las acciones de los usuarios afectan de manera diferente el comportamiento global de la aplicación y la producción de los colaboradores en una aplicación de mensajería instantánea como *Messenger* y en un juego en línea como *Halo*. Las dos aplicaciones cooperativas funcionan en tiempo real, aunque los usuarios de la aplicación de mensajería, pueden seguir usando la aplicación aún cuando los mensajes tengan pequeños retardos en las notificaciones. Un juego como *Halo* se vuelve imposible de usar cuando existe una alta latencia en el reflejo de las acciones de los demás usuarios en la interfaz de todos los jugadores, debido a que desconocen la posición real de los demás.
Con respecto a los editores colaborativos, no es lo mismo usar ese tipo de herramientas en una red local, que en un ambiente geográfica y altamente distribuido (*e.g. Wikipedia* en la Internet), con millones de usuarios, por lo que se requiere una mayor disponibilidad de la información compartida. Como las fallas en la Web son posibles, se requieren mecanismos que garanticen una alta disponibilidad.
Debido a la complejidad de la implementación de las aplicaciones colaborativas, muchos desarrolladores pasan por alto los requerimientos particulares de las aplicaciones y se limitan a utilizar los mecanismos de notificación de las acciones de los demás usuarios. Ésto es más fácil de implementar, pero no satisface completamente los requerimientos de la aplicaciones cooperativas.

Aplicaciones colaborativas distribuidas son diseñadas como clientes de un servidor central para el almacenamiento de los datos y la coordinación de las acciones de los usuarios. En consecuencia, las aplicaciones no son tolerantes a fallas y su desempeño es pobre ya que no reflejan adecuadamente las acciones de los colaboradores cuando existe latencia en la red.
Los desarrolladores de aplicaciones colaborativas, al implementarlas toman muy poco en cuenta que algunos dispositivos tienen capacidades limitadas en cuanto a almacenamiento, procesamiento y capacidad de manejar ciertos servicios o formatos, por lo que no pueden manejar todos los datos de una aplicación colaborativa, ocasionando que en muchas ocasiones, sea imposible de utilizarlas.

Trabajo cooperativo en la web

Una de las aplicaciones colaborativas que tiene un gran éxito en la Web es *Wikipedia*. *Wikipedia* esta formado por los *wikis*, los cuales han probado ser una herramienta poderosa para la generación de contenido compartido y la colaboración en la Web. Los *wikis* permiten manejar páginas Web con una semántica simple, de tal forma que los usuarios pueden producir artículos en un formato libre, agregar anotaciones y comentarios. Además, algunas instituciones no gubernamentales como escuelas y centros de investigación, distribuyen instantáneas del contenido de *Wikipedia* con fines educativos.

Pero a pesar de su utilidad, el software actual de los *wikis* tiene sus deficiencias. Los datos de los *wikis* se dividen de acuerdo al lenguaje en que fueron escritos y el tema. Cada servidor almacena los datos y coordina la producción y modificaciones de los artículos. Debido a que los datos se encuentran centralizados en un servidor, este tipo de arquitectura es inútil cuando el servidor no está disponible.
Dado que la Internet es un ambiente poco fiable con múltiples desconexiones, el uso de los *wikis* se ve mermado considerablemente. Es posible tener la información de los *wikis* disponible por medio de las instantáneas pero, no es posible hacer actualizaciones al contenido, debido a que el flujo de la información es en un sólo sentido, perdiéndose muchas de las discuciones y comentarios generados durante las clases en las que éstas se utilizan.

Existen algunos intentos por mejorar el desempeño de los *wikis* como *DTWiki* [13]. *DTWiki* está basado en una política optimista de manejo de las producciones, que permite a los usuarios modificar un mismo artículo de manera simultánea. Ésta incrementa la colaboración, pero como la granularidad de la información es a nivel del *namespace* del *Wiki*, en vez de usar grano fino a nivel párrafo, ocurren múltiples inconsistencias debido a las actualizaciones concurrentes de las páginas. *DTWiki* más que garantizar la consistencia de los datos (*i.e.* produciendo una versión correcta a partir de las dos versiones), ofrecen coherencia de los mismos (*i.e.* detecta las modificaciones concurrentes), por lo que los usuarios tienen que resolver manualmente las inconsistencias de los datos debido a producciones concurrentes.

Requerimientos para el trabajo colaborativo en un ambiente distribuido

Para ofrecer un buen nivel de servicio y de confort a los usuarios, consideramos que las aplicaciones colaborativas deben:

- **ser tolerantes a fallas**. Si algún elemento de la red falla, los usuarios deben poder seguir trabajando (en modo temporalmente desconectado), evitando que se pierda el trabajo realizado o se produzcan versiones antogónicas,
- **tener un buen desempeño**, es decir deben ofrecer herramientas eficientes y un tiempo de respuesta razonable,
- **soportar el trabajo nomada y/o temporalmente desconectado**, *i.e.* las aplicaciones deben poder manejar las desconexiones, permitiendo que el usuario pueda seguir trabajando, aún cuando no está conectado de manera continua y permitir que se pueda pasar de un dispositivo de interacción a otro, sin ruptura de su sesión de trabajo,
- **ser persistentes a las fallas** y ser capaces de lidiar con las inconsistencias, temporales,
- **soportar actualizaciones dinámicas** de software incrementando la disponibilidad del servicio y así, la continuidad del proceso global de cooperación.

Para que las aplicaciones colaborativas satisfagan los cinco primeros requerimientos es necesario contar con mecanismos de replicación de los datos entre los sitios de colaboración, ya sea parcial o total, permitiendo que los usuarios: a) puedan seguir trabajando a pesar de las fallas en la red o en los servidores b) que las aplicaciones tengan un mejor desempeño y c) que el trabajo nómada sea posible a pesar de la baja capacidad de los dispositivos y las continuas desconexiones y reconexiones debidas a la movilidad de los usuarios. Pero si la distribución de réplicas incrementa la disponibilidad de la información en la red de colaboración, en contraparte, puede causar inconsistencias en la información. Así, es necesario proveer los mecanismos de notificación de los cambios que mejor se adapten al ambiente del usuario, al tipo de aplicación y las características de los dispositivo en donde se realiza el tratamiento de los datos.

Políticas de replicación/actualización de la información distribuida

Para diseñar e implementar los mecanismos de distribución, notificación y sincronización de las réplicas se pueden tomar diferentes criterios o políticas. El problema es que cada política fue creada para solucionar un caso de cooperación especial, por lo cual no funcionan adecuadamente en todos los ambientes ni con todas las aplicaciones.

Así por ejemplo, una política basada en turnos funciona bien en aplicaciones que requieren turnos, como algunos juegos (e.g. domino, cartas, damas chinas, etc.), pero son completamente ineficientes con otros tipos de aplicaciones, ya que los usuarios

tienen que esperar largos períodos de tiempo antes de poder interactuar, limitando de esta manera la colaboración. De hecho si el entorno cambia, e.g. la aplicación pasa de utilizarse de un ambiente local a la Web o si el número de usuarios se incrementa considerablemente, será necesario analizar si las políticas utilizadas inicialmente siguen siendo las más adecuadas o aplicables, ya que son factores determinantes en la selección de la estrategia de replicación a seguir.

Al seleccionar una política se deben elegir varios criterios como la granularidad de los datos, la forma de resolver inconsistencias, el tiempo entre las sincronizaciones de las entidades, el tipo de reloj a utilizar para determinar el orden de las actualizaciones, etc. Este conjunto de criterios debe ser analizado y probado para verificar el correcto funcionamiento de la aplicación, ya que son determinantes en el desempeño de la misma.

Proponemos diseñar e implementar una plataforma flexible y extensible que permita utilizar diferentes políticas. Así, usando esta plataforma, el desarrollador de aplicaciones colaborativas puede seleccionar entre un conjunto de políticas de control de concurrencia, distribución y notificación para escoger la solución que se ajusta mejor de acuerdo al escenario y a los requerimientos donde se ejecuta la aplicación. También permite adaptar una nueva solución cuando el ambiente de colaboración cambie (e.g. incremento en el número de usuarios, utilización de dispositivos con diferentes capacidades, cambio de las condiciones de red, *etc.*).

Problemática Actual

Como lo planteamos anteriormente, las aplicaciones cooperativas distribuidas tienen un mejor desempeño dependiendo de la estrategia de replicación de las copias de las entidades (datos) compartidas. Pero es complicado implementar el software para controlar las réplicas de la información utilizando políticas más sofisticadas, que satisfagan todos los requerimientos de las aplicaciones colaborativas distribuidas.
Si se quiere incrementar la disponibilidad de los datos compartidos entre los usuarios, es necesario proveer los mecanismos que permitan seguir soportando el trabajo colaborativo, aún cuando algunos elementos de la red de colaboración fallen. Evidentemente, se hace más complicada la implementación de la sincronización de las réplicas de la información entre los sitios colaborativos. Por esta razón muchos desarrolladores simplifican su trabajo y no utilizan la estrategia adecuada, sino que utilizan la más simple o la estrategia de sincronización de réplicas disponible, afectando el desempeño de las aplicaciones.

Actualmente existen varias plataformas que proveen sólo una estrategia de replicación de los datos compartidos. Estas plataformas o sistemas proveen a los usuarios el soporte de la notificación y sincronización de las acciones. Pero estas plataformas son limitadas, ya que utilizan sólo una política de replicación y soportan sólo una arquitectura de distribución. *WebDAV* [14][15][16], *COAST* [24], entre otras, constituyen ejemplos representativos de estas plataformas. Mientras *WebDAV* utiliza una política

de replicación pesimista (*i.e.* garantiza que cuando un usuario esté modificando un objeto, nadie más pueda hacerlo), COAST ofrece una replicación optimista (*i.e.* permite modificaciones concurrentes y resuelve a *posteriori* las inconsistencias).
Estas plataformas sólo cuentan con una estrategia de sincronización de las réplicas de los datos compartidos, por lo que cada una tiene sus limitantes. Así *WebDAV* no es tolerante a fallas y no soporta la movilidad de los usuarios. *COAST* realiza las actualizaciones en base al orden de los eventos y utiliza funciones "hacer" y "deshacer" por lo que en ocasiones los datos se corrompen. Dependiendo del tipo de aplicación, estas limitantes tendrán mayores o menores inconvenientes. Como consecuencia, se deben analizar las ventajas y desventajas, tomando en cuenta el tipo de aplicación y el ambiente (*e.g.* condiciones de red, número de usuarios, movilidad de los usuarios, *etc.*) en el que será implementada.

Existen plataformas que soportan diversos tipos de políticas como *DreamTeam* [22] y *GEN* [21]. *GEN* sólo ofrece el control de concurrencia flexible y soporta arquitecturas centralizadas y completamente replicadas, además fue un prototipo que no se le siguió dando mantenimiento. *DreamTeam* es una plataforma que permite arquitecturas flexibles al permitir adaptar la distribución de las notificaciones de las actualizaciones, de acuerdo al ambiente del usuario (*e.g.* un entorno nómada *vs.* un ambiente local). La distribución es flexible, pero sólo utiliza una política de control de concurrencia basada en bloqueos de la información, que no se puede usar en todos los tipos de aplicaciones.

Luckosh and Shummer [23] proponen un lenguaje basado en patrones de diseño. Éste simplifica el trabajo de los desarrolladores de las aplicaciones colaborativas, pero en ningún momento esta solución considera el tipo de aplicación, sólo la movilidad de los usuarios y las fallas en la red.

Saito y Shapiro [7] hacen un análisis muy detallado de los principios de la replicación optimista. Este análisis puede ayudar a determinar cual es la mejor estrategia de replicación a seguir, pero no toman en cuenta el tipo de aplicación, sólo factores ambientales como la distribución de los usuarios, la latencia de la red, la movilidad de los usuarios, *etc*.

Objetivo general del proyecto

El objetivo de este trabajo de investigación consiste en diseñar un sistema genérico, flexible y extensible de entidades compartidas replicadas. Este sistema de definición y de administración de entidades compartidas constituye y se integra como un componente central para la definición de una plataforma *middleware* para el soporte de aplicaciones cooperativas distribuidas.

Así, este trabajo se enfoca en modelar, diseñar y definir el prototipo de un sistema de entidades flexibles y extensibles para aplicaciones cooperativas distribuidas. Los mecanismos y servicios a definir, deben permitir al programador definir nuevos tipos de entidades replicadas a partir de modelos existentes. Eventualmente, se debe poder recuperar y modificar los tipos ya declarados, de una manera coherente y consistente,

facilitando el desarrollo y mantenimiento de aplicaciones colaborativas distribuidas. Dichas entidades están constituidas por: a) un tipo, b) un estado (estructura de datos), c) una capa funcional, d) una capa de administración de la replicación de los datos y e) una capa de actualizaciones de versiones de software de las clases y sus instancias en tiempo de ejecución en un ambiente de colaboración distribuido.

La capa de administración de versiones del software de las clases y sus instancias debe administrar la evolución de la entidad compartida (tipo y estructura de las instancias). También debe proveer los mecanismos de actualización en tiempo de ejecución (*on the fly updating*), considerando que las aplicaciones colaborativas están formadas por entidades multiprogramadas, compartidas, distribuidas y replicadas.

Objetivo particulares

1. Estudiar los diferentes modelos para la definición y la administración de entidades, componentes o módulos existentes.

2. Definir un modelo y una representación de una entidad, que permita la creación y la administración de entidades compartidas distribuidas de una manera genérica, flexible y extensible.

3. Estudiar las herramientas que provee el lenguaje Ruby para: a) darle flexibilidad a la programación de las aplicaciones, b) implementar los servicios y mecanismos siguiendo el paradigma de la programación orientada a objetos, c) conocer el estado de las clases y facilitar las actualizaciones dinámicas y d) implementar estos servicios flexibles usando patrones de diseño como: *strategy*, *template*, *adapter*, *facade*, *proxy*, *observer*, entre otros. Estos patrones deben facilitar la programación de las diferentes capas de software y dar flexibilidad a las entidades compartidas distribuidas.

4. Estudiar la programación orientada a aspectos y analizar la factibilidad y la viabilidad de su uso en el lenguaje Ruby.

5. Proponer un modelo y una representación de datos para los tipos de entidades y las entidades compartidas.

6. Proporcionar una interfaz de programación que permita a los desarrolladores de aplicaciones colaborativas comunicar a las aplicaciones colaborativas con las entidades compartidas distribuidas.

7. Proveer una meta-interfaz que permita al desarrollador de aplicaciones colaborativas crear e integrar nuevas políticas (distribución, control de concurrencia, *etc.*) y seleccionarlas de acuerdo a los requerimientos de cada tipo de entidad compartida y distribuida. De esta manera, se contará con una plataforma flexible y extensible.

8. Analizar los modelos existentes de actualización de versiones de componentes en sistemas distribuidos, que permitan transformar una entidad de tipo T en T'. En donde el tipo T define un modelo de instancias que incluye p atributos (t_i tal que i∈{1,..,p}) y q métodos (m_k tal que k∈{1,..,q}). Esta transformación es posible al aplicar una función de actualización sobre los atributos $f_{at}()$ y funciones de actualización $f_{amk}()$ sobre los métodos m_k. Así nos da como resultado el tipo T' formado por r atributos (t_i tal que i ∈{1,..,r}) y s métodos (m_k tal que k ∈{1,..s} y s>q).

9. Proveer al sistema los mecanismos necesarios para modificar los tipos de entidades y sus instancias en modo de ejecución, conservando la consistencia de las aplicaciones, considerando que las entidades son multiprogramadas, distribuidas, compartidas y replicadas.

Así, queremos definir una plataforma que soporte aplicaciones colaborativas distribuidas que provea un tipo genérico de entidad compartida que incluya las siguientes funcionalidades:

- **La capa "funcional"** de la entidad (*i.e.* el comportamiento de la entidad) y su estado (estructura de datos). Por ejemplo, en una aplicación de pizarrón colaborativo son las funciones que permiten agregar, editar o suprimir figuras y sus atributos como son tamaño, color, forma, *etc.*, almacenados en la estructura de datos. En este caso, desde el punto de vista de la entidad, son los objetos dibujados y agregados por el usuario.

- **La capa de "administración"**: se encarga de aspectos relacionados con la adminiinistración de la sincronización de los datos: arquitectura de distribución de las entidades (*e.g.* centralizada, totalmente replicada, parcialmente replicada, *etc.*), control de concurrencia (*e.g.* siguiendo políticas optimistas o pesimistas), tipo de replicación (*i.e.* la replicación de la entidad puede ser total o parcial para incrementar la disponibilidad de la información). Además es necesario proveer los mecanismos que garanticen la consistencia de las réplicas, por lo que es requerido proveer políticas de difusión y de actualización.

- Las funciones que definan **la capa de "actualizaciones dinámicas de *software*"** formada por las funciones que administren a) la versión de las clases y b) los mecanismos de actualización que permitan la producción de las nuevas versiones del software en tiempo de ejecución.

Contenido de la tesis

En el segundo capítulo se presentan algunas definiciones relacionadas con el dominio de aplicación de la plataforma. Esta parte presenta un análisis de las características de las aplicaciones colaborativas y las diferentes políticas utilizadas. También se

presenta el estado actual de las aplicaciones colaborativas en la Web, sus características de distribución, replicación y sincronización de los datos. En el tercer capítulo analiza los trabajos de flexibilidad y actualizaciones dinámicas que sirven como base al desarrollo del modelo.

Utilizando las herramientas de flexibilidad, el capítulo cuarto presenta el diseño de la plataforma y el diseño de una entidad genérica, de la cual se derivan los diferentes tipos de entidades. También describe los módulos funcionales, sus características, sus interfaces, los procesos que cada uno desarrollan y la interacción entre ellos.
Desde el punto de validación de este modelo, el capítulo V presenta y analiza diferentes aplicaciones colaborativas: un chat, un editor de documentos y una agenda.

El capítulo sexto describe la implementación del modelo y algunas herramientas propias del lenguaje de programación Ruby utilizadas para el desarrollo de la solución propuesta. Se describe la aplicación desarrollada y las pruebas hechas utilizando el modelo de actualizaciones dinámicas.

Por último el capítulo VII presenta las conclusiones y el trabajo futuro que tiene como objetivo mejorar el modelo.

Capítulo 2

Antecedentes y motivación

Actualmente se cuenta con aplicaciones colaborativas que permiten a grupos de personas realizar en un ambiente compartido, un objetivo o tarea en común, que constituye el resultado esperado de la colaboración. Las agendas colaborativas, los editores de texto, los juegos en línea, los pizarrones colaborativos, *etc*. son ejemplos típicos de este tipo de aplicaciones. Éstas son más complejas que las aplicaciones mono-usuario ya que la interfaz debe incluir los elementos de interacción necesarios que permitan: a) soportar la interacción de la colaboración y b) tener conciencia del estado del trabajo de los demás colaboradores. Además, como estas aplicaciones son distribuidas, es necesario notificar los cambios a los diferentes dispositivos donde se ejecuta la aplicación para mantener la coherencia y la consistencia de los datos, teniendo un desempeño razonable.

La primera sección tiene como función introducirnos a las aplicaciones colaborativas, así como a los elementos que permiten la replicación de los datos entre los sitios de trabajo de los colaboradores y los sitios de los repositorios (las diferentes formas que permiten mantener la coherencia y consistencia de los datos, a pesar de los cambios concurrentes efectuados sobre los mismos).

Los tipos de datos replicados entre los sitios de trabajo tienen diferentes requerimientos, por lo que en la segunda sección analizamos algunas aplicaciones colaborativas con la finalidad de ejemplificar sus diferentes comportamientos y las soluciones que se encuentran actualmente en la Web. Estas soluciones, aunque son de gran utilidad, no satisfacen completamente todos los requerimientos de los usuarios debido a que no cuentan con los mecanismos de replicación idóneos, de acuerdo a la naturaleza de los datos y al ambiente de trabajo de los usuarios.

Nuestro enfoque tiene como objetivo permitir utilizar diferentes mecanismos de replicación asociados a cada tipo de datos. Para hacer esto posible se diseñó una plataforma con una arquitectura flexible y extensible, la cual es introducida en la tercera subsección. Mediante un sistema de actualizaciones dinámicas se permite a los desarrolladores de aplicaciones colaborativas, la definición de nuevas entidades replicadas distribuidas y de sus tipos, a partir de modelos existentes, así como la integración de nuevas políticas de distribución, control de acceso, control de concurrencia, *etc*. Esta

arquitectura al ser actualizable en tiempo de ejecución permite una mayor disponibilidad de la aplicaciones colaborativas distribuidas.

2.1 Definiciones y principios de la colaboración asistida por computadoras

2.1.1 Definición de *groupware* o software de trabajo colaborativo

En nuestros días se desarrollan cada vez más aplicaciones que tienen como fin ayudar a la colaboración entre usuarios que trabajan en una meta en común. Estas aplicaciones, ayudan en mayor o menor grado, en los aspectos que conforman la colaboración humana, la cual de acuerdo con Ellis y Wainer [2] están representados por: a) el modelo ontológico, que describe la producción del trabajo colaborativo (*i.e.* datos y operaciones sobre estos datos), b) el modelo de coordinación, el cual describe las actividades de cada participante y su integración para poder completar el trabajo común y c) el modelo de interfaces de usuario, formado por las diferentes vistas de los objetos, las vistas de los participantes y las vistas del contexto.

Este tipo de aplicaciones que ayudan a la interacción humana se les conoce como groupware.

Ellis *et. al* [1] definen una aplicación/sistema *groupware* como: "*Un sistema basado en computadoras que soporta grupos de gente comprometidos en una tarea en común (o meta) y que provee una interfaz de interacción al ambiente compartido*".

Esta definición destaca el ambiente compartido, por lo que las aplicaciones en tiempo compartido como las bases de datos, los sistemas de inventarios, *etc.*, no caen dentro de esta definición ya que no proveen una interfaz de interacción al ambiente compartido.

Las características de las aplicaciones pueden variar considerablemente. La manera en que ayudan a la colaboración de los usuarios, crea diferentes expectativas por parte de los mismos. Estas aplicaciones pueden permitir la colaboración al mismo tiempo (aplicaciones síncronas) o de manera diferida (aplicaciones asíncronas), dependiendo de las características de la aplicación, *i.e.* los medios, las herramientas y las formas en que las aplicaciones permiten que los usuarios colaboren.

Una característica importante de las aplicaciones colaborativas es que los usuarios necesitan estar concientes de las acciones de los demás para poder producir en grupo de manera más eficiente. En opocisión a esta definición, en los sistemas distribuidos tradicionales, los usuarios tienen la sensación de ser los únicos trabajando aunque no sea así, al no tener conocimiento del trabajo de los demás.

A esta característica de conocer las acciones de los demás colaboradores, se le llama *conciencia de grupo*.

Dourish y Bellotti [4] definen la conciencia de grupo como: *"la comprensión de las actividades de los demás, la cual provee un contexto para la actividad propia. Este contexto es usado para asegurar que las contribuciones individuales sean relevantes para la actividad del*

grupo como un todo y evaluar las acciones individuales con respecto a las metas del grupo y su progreso."

Ellis *et. al* [1] no establecen que la colaboración deba realizarse a distancia o simultáneamente, pero estos dos factores influyen considerablemente en el tipo de aplicación y sus requerimientos. Una sesión de trabajo cooperativa en la cual los colaboradores están geográficamente distribuidos debe contar con los mecanismos necesarios que complementen la comunicación entre los colaboradores. Dichos mecanismos sustituyen la comunicación no hablada, cuando la colaboración no es cara a cara, (*e.g.* expresiones faciales, estados de ánimo, etc.). Una aplicación síncrona debe asegurar que los datos entre los usuarios se actualicen de manera rápida y consistente. Shapiro y Saito [7] definen una aplicación consistente como: aquella que al finalizar el trabajo, todos los sitios llegarán al mismo estado de los datos. Las actualizaciones de los datos en aplicaciones síncronas debe ser oportuna, *i.e.*, los usuarios deben conocer el trabajo de los demás en un tiempo que permita tomar las decisiones adecuadas. Así, para evitar conflictos entre los usuarios, es necesario que los cambios se distribuyan lo más rápidamente posible, para evitar confusiones entre los usuarios debido a que no están enterados de los cambios.

Actualmente, estas aplicaciones colaborativas están cambiando la manera de interactuar de las personas. Vemos usuarios colaborando en la producción de artículos o libros, utilizando aplicaciones colaborativas. Éstas los ayudan en la representación, coordinación e integración de los datos compartidos por medio de interfaces de usuario que proporciona una adecuada conciencia de grupo, así como diferentes vistas de los datos.

No hace muchos años, la única forma de comunicarse con las personas en otros continentes era usando la vía telefónica, con comunicaciones deficientes y caras. Hoy en día la comunicación es más eficiente y es posible hacerlo por medio de diferentes aplicaciones colaborativas como los chats, la videoconferencia, las redes sociales, etc. Pero es necesario desarrollar aplicaciones más poderosas que faciliten la colaboración entre los usuarios, soportando los diferentes dispositivos que se encuentran actualmente en el mercado. En el ambiente de trabajo colaborativo móvil y/o nómada, sin el software de *groupware*, no se puede explotar las facilidades de comunicación y procesamiento en dispositivos móviles de tercera generación (*e.g.* el *smartphone*, el *iphone etc.*). El software de *groupware* permite el uso de aplicaciones colaborativas a pesar de la baja capacidad de estos dispositivos y la movilidad de los usuarios.

Para muchos grupos de trabajo es importante estar comunicados la mayor parte del día, debido a que mejoran de manera importante el tiempo de reacción ante los retos y los problemas que se presentan en el trabajo cotidiano. Las pérdidas económicas o las pérdidas de oportunidades ocasionadas por la falta de información oportuna se pueden reducir considerablemente, por lo que las empresas de vanguardia deben proporcionar a sus trabajadores y sus clientes los medios que promuevan la colaboración continua.

En nuestros días, las personas (en su mayoría jóvenes) pasan horas jugando videojuegos en línea, por lo que la colaboración no sólo se da con fines laborales, sino

que forma parte del entretenimiento de las personas. A diferencia de otros tipos de aplicaciones colaborativas, las cuales para realizar un trabajo en común son más heterogéneas, los juegos colaborativos es su mayoría son groupware síncrono.

Si observamos la colaboración en nuestros días y la comparamos con la existente hace 20 años atrás, podemos ver que ha cambiado radicalmente y se vislumbra que seguirá cambiando, conforme la aplicaciones colaborativas evolucionen, proporcionando a los usuarios nuevas formas de interacción.

2.1.2 Control de concurrencia

Cuando los colaboradores están interactuando en un ambiente distribuido y trabajando desde varias computadoras, si ellos tienen acceso a diferentes réplicas de la información, el orden en que se reciben los datos y los eventos puede variar, dando como resultado inconsistencias en las entidades compartidas. Cuando el tiempo de transmisión de las notificaciones es corto (*e.g.* las máquinas de los sitios de colaboración se encuentren conectadas a una red local), puede que haya pocas inconsistencias, si la aplicación proporciona los mecanismos para que los colaboradores tengan una buena conciencia de grupo. Pero cuando las máquinas de los sitios de colaboración están geográficamente distribuidas, la latencia de la red es considerable, incrementando la probabilidad de los conflictos en las operaciones, por lo que es necesario establecer un control de concurrencia.

Greenberg y Marwood [3] define al control de concurrencia como: "*la actividad de coordinar las acciones que potencialmente se interfieren, de un proceso que opera en paralelo*".

Grano de la información compartida

Cuando hablamos de control de concurrencia, es importante definir la granularidad de los datos compartidos, la noción de unidad de un dato compartido, o el tamaño del fragmento de los datos que un usuario puede manipular o grano de coperación. Por ejemplo, si la información compartida y colaborativamente producida es un libro, podemos considerar como un objeto al libro, a un capítulo, a una sección, a un párrafo o a una palabra. Se dice que el grano de colaboración es de tamaño grueso cuando el fragmento es grande (*e.g.* el libro entero o un capítulo). Cuando el tamaño del fragmento es reducido, como en el ejemplo a nivel párrafo o palabra, se dice que el grano de colaboración es fino. Si el grano es grueso se limita la concurrencia; si el grano es muy fino, se incrementa el *overhead* debido al alto requerimiento de procesamiento de los datos, ocasionando un elevado número de mensajes, de tal forma que en ocasiones puede llegar a mermar el desempeño de la aplicación.

El control de la concurrencia se puede llevar a cabo ejecutando las operaciones atómicas de manera serializada (*i.e.* tomando turnos) o permitiendo que los eventos se ejecuten sin una serialización previa y los eventos que sean incosistentes se repararen posteriormente. Greenberg y Marwood [3] establecen que "*las técnicas no optimistas aseguran que los eventos se reciban sólo en orden, y así garantizar la consistencia. Las técnicas optimistas permiten que los eventos se reciban fuera de orden y en ese caso las inconsistencias*

deben ser detectadas y reparadas". Las técnicas pesimistas, garantizan la consistencia pero limitan más la concurrencia, aunque se puede incrementar disminuyendo la granularidad de los objetos.

Dentro de las teorías pesimistas se usa comúnmente el control de concurrencia basado en candados. Greenberg y Marwood [3] definen un candado como: "*un método para ganar el privilegio de acceder a un objeto por un período de tiempo*". Los usuarios solicitan el permiso, si les es otorgado realizan los cambios en el objeto y por último liberan el candado. Existen algunas variantes de candados, *e.g.* un candado optimista es aquel que permite a los usuarios manipular los datos antes de que se otorgue el permiso. Si el candado finalmente es negado, los cambios realizados son deshechos y los datos compartidos regresan a un estado anterior. Los candados optimistas funcionan en ambientes confiables (*i.e.* sin desconexiones ni retardos), ya que mejora el tiempo de respuesta de las aplicaciones. Pero si la latencia es considerable, o peor aún, si existen desconexiones, los cambios realizados pueden ser considerables. Si estos cambios deben ser deshechos, los usuarios sentirán que su trabajo fue en vano, ocasionando descontento hacia el sistema.

Saito & Shapiro [7] definen las políticas optimistas como aquellas que: "*permiten que los datos sean accedidos sin una sincronización a priori basándose en el supuesto de que los problemas rara vez ocurrirán, si no es que nunca. Las actualizaciones son propagadas ... y los conflictos ocasionales son solucionados después de que ocurren*".

Para corregir los problemas de serialización se pueden deshacer los eventos y volverlos a hacer en orden, o utilizar una función de transformación que llegue al mismo resultado, como si fueran ejecutados en orden.

Algunos tipos de datos funcionan adecuadamente con este tipo de políticas, ya que permiten una mayor colaboración, mientras que en otros casos (*e.g.* la producción de un artículo), la solución a las inconsistencias puede llegar a corromper los datos, perdiéndose el trabajo de todos los colaboradores. En otros (*e.g.* la selección del color de una figura en un pizarrón blanco), el sistema selecciona una de las producciones; ya sea aleatoriamente o en base a alguna política; perdiéndose el trabajo de los demás y generalmente la consistencia de la colaboración.

Criterios para conformar una política de replicación optimista

Cuando se utiliza una política optimista es necesario determinar varios criterios o parámetros que permiten definirla. En seguida presentaremos estos parámetros que influyen de manera determinante en el comportamiento de la aplicación.

- El **tipo de serialización**, que es la manera en que se establece el orden correcto de los eventos. La serialización sintáctica establece el orden con base a cuándo, dónde y quién, *e.g.* marcas de tiempo.

  Para ejemplificar la serialización, podemos suponer que dos usuarios están modificando un dibujo como se muestra en la figura 2.1

  Primero uno modifica el color de una casa y después el segundo modifica el color del techo. Debido a la latencia de la red, el segundo no ha recibido los cambios del

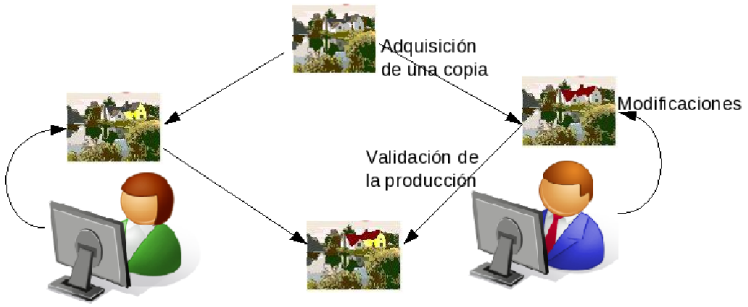

Figura 2.1: Modificaciones concurrentes

usuario número uno, por lo que cuando sean integrados, si se usa una serialización sintáctica, la aplicación deshacerá el cambio del color del techo, modificará el color de la casa y volverá a pintar el techo. Las operaciones de deshacer y rehacer pueden causar confusión en los usuarios.

La serialización semántica conmutativa de las operaciones se basa en la conmutación e idempotencia de los eventos asociados, conservando el orden en que fueron recibidos cuando estos alcanzan el mismo estado final independientemente de que difieran del orden real de las operaciones. En el ejemplo del dibujo compartido no es necesario deshacer las operaciones, ya que pintar el techo y pintar la casa son operaciones conmutativas, por lo que podemos agregar los cambios y las imágenes en los sitios llegarán al mismo estado.

La serialización semántica, basada en transformaciones operacionales, utiliza una función de transformación del objeto o fragmento que le permite alcanzar el estado final, cuando las operaciones no son conmutativas, sin tener que deshacer y rehacer las operaciones requeridas. Siguiendo el ejemplo anterior, si el primer usuario pinta la casa de amarillo y después el segundo usuario pinta la casa de verde, se puede establecer la regla de que el último color es el que prevalece, sin tener que realizar operaciones de deshacer y rehacer. También se pueden optimizar las operaciones eliminando con base en filtros por: depencia (un evento no es permitido si no se ejecuta otro antes), implicación (si ocurre un evento tiene una consecuencia), elección (puede ocurrir un evento u otro pero no ambos), *etc*.

- **La forma de detección de los conflictos** ocasionados por modificaciones concurrentes al mismo fragmento de los datos. Una detección sintáctica se basa en relaciones de ocurrencia de los eventos (*happens-before*). Una detección semántica utiliza información semántica para detectar los conflictos.

- **La forma en que se corrigen los conflictos**. Un conflicto detectado puede ser ignorado y así conservar la última modificación (*Thoma's write rule* [7]). Otra posibilidad consiste en dejar que los usuarios los resuelvan. También se pueden crear políticas que permitan resolverlos de manera automática combinando las dos versiones generadas.

- **El tipo de fragmentación** que se utilizará en las entidades. Una fragmentación fina ayuda a decrementar el número de conflictos, debido a que la probabilidad de que dos usuarios modifiquen el mismo fragmento es menor entre más pequeño sea, pero fragmentos muy pequeños incrementan el costo de administración de los mismos y requieren un ancho de banda mayor al crecer el número de notificaciones, lo cual puede afectar el desempeño de la aplicación, dependiendo de sus características (*e.g.* si es síncrona o asíncrona).

Políticas de replicación optimistas vs. pesimistas

Cuando existe una buena conciencia de grupo, y el trabajo de los demás se conoce perfectamente y en el tiempo oportuno, las políticas optimistas promueven la colaboración ya que permiten que los usuarios trabajen libremente sin restricciones del sistema. Pero estas políticas pueden llegar a causar molestias en los usuarios cuando hay inconsistencias en los datos, ocasionando la pérdida del trabajo realizado. Dependiendo de la frecuencia de las inconsistencias y la cantidad de trabajo perdido, se debe implementar una política de control de concurrencia optimista. Por ejemplo, no es lo mismo elegir la hora correcta de una cita en una agenda colaborativa, que perder las modificaciones hechas a un artículo durante 1 hora, porque al integrar los cambios, el documento quedó ilegible, o la política no validó los cambios porque aprobó los realizados por otro usuario.

Muchos investigadores consideran que las políticas de control de concurrencia optimistas son mejores que las pesimistas. Por ejemplo Saito y Shapiro [7] señalan 3 razones por las que no se puede obtener un buen desempeño basado en políticas pesimistas, principalmente candados, en ambientes como la Web:

- Un algoritmo de replicación pesimista, que intente sincronizarse con un sitio que no está disponible, podría quedar bloqueado indefinidamente.

- Algoritmos pesimistas escalan pobremente en áreas amplias, debido a la latencia de la red, ya que los usuarios requieren validar la operación que desean realizar antes de llevarse a cabo.

- Algunos tipos de datos requieren compartición optimista, para que la colaboración fluya correctamente, *e.g.* un juego síncrono colaborativo (*e.g. Halo*) no puede bloquear las acciones de los jugadores, sino que debe permitir que jueguen libremente.

Sin embargo, las políticas pesimistas son la mejor opción para ciertos tipos de datos o en ciertos tipos de ambientes. Además existen variantes pesimistas que promueven la colaboración como los candados compartidos [15] o la coordinación distribuida a nivel fragmento [9].

Así, con el objetivo de elegir una política de control de concurrencia para la administración de diferentes entidades compartidas, es necesario considerar:

- **la cantidad de inconsistencias:** si las inconsistencias potenciales son reducidas los usuarios pueden ser capaces de manejarlas eventualmente. Pero cuando son muchas la aplicación pierde usabilidad, debido a que los usuarios se tienen que dedicar a tareas de administración, más que de producción;

- **la naturaleza de los datos:** algunas estructuras y/o datos permiten la conmutación de los eventos, o existen las funciones de transformación que permiten llegar a un estado consistente, mientras que en otros tipos de datos las inconsistencias deben ser solucionadas por los usuarios y en algunos casos es necesario retrabajar los datos, ya que son propensos a corromperse;

- **el número de usuarios** que interactúan en la aplicación: al incrementarse el número de colaboradores la probabilidad de que interfieran unos con otros aumenta, incrementándose el número de inconsistencias,

- **la naturaleza de la aplicación** y la naturaleza de la colaboración: se debe analizar la forma en que se debe manejar el control de la concurrencia, cuando no es impuesto por el sistema, para determinar el comportamiento que los usuarios esperan obtener de la aplicación colaborativa distribuida.

Históricos de los datos

Algunas de las políticas de control de concurrencia optimistas requieren conocer los estados anteriores de los datos, para poder corregir las inconsistencias, deshaciendo y rehaciendo las operaciones.

También es conveniente almacenar el estado anterior de los datos como respaldo, o simplemente a solicitud del usuario. Por estas razones se debe llevar un registro de las acciones precedentes por medio de *versioning* o históricos.

Versioning es cuando se almacenan los diferentes estados o versiones por los que pasa la entidad compartida.

En el caso de los históricos, se almacena el listado de las operaciones que permiten producir las diferentes versiones de la entidad.

2.1.3 Distribución/notificación de las modificaciones de datos compartidos

Los usuarios de un sistema *groupware* pueden estar distribuidos y trabajar en diferentes lugares, así como también pueden trabajar/producir en diferentes momentos. Dependiendo de la forma en que los usuarios se encuentren distribuidos en tiempo y espacio para el tratamiento de los datos, las aplicaciones tendrán características y requerimientos de notificación y distribución de los datos compartidos particulares. Por ejemplo, comparemos una aplicación de e-mail, una aplicación de chat y una aplicación de voz IP. Las tres aplicaciones comunican a los usuarios por medio de mensajes, pero el e-mail es una aplicación asíncrona, por lo que no importa si el otro usuario recibe el mensaje

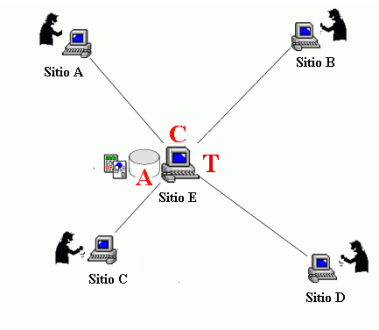

Figura 2.2: Arquitectura centralizada

inmediatamente, en una hora o en un día, lo primordial es que el mensaje debe llegar intacto. En el caso del chat, pequeños retrasos son manejables por los usuarios, pero si los retrasos son importantes la comunicación pierde continuidad. Al ser mensajes escritos, al igual que el e-mail, también deben llegar sin errores ni pérdidas. En el caso de una aplicación para comunicación por voz IP, no sólo importa que la latencia de los datos sea mínima, sino que no haya varianza en los retrasos, para que la voz pueda ser comprensible. Sin embargo dado que para digitalizar la voz se toman 8000 muestras/seg., si se pierden algunos datos (siempre y cuando sean pocas las pérdidas), los usuarios ni siquiera se percatan de su falta ya que la mente se encarga de completar los datos faltantes. Como vemos las 3 aplicaciones tienen requerimientos diferentes a pesar de que las 3 tienen como finalidad comunicar a usuarios distantes.

Los datos compartidos pueden distribuirse entre los diferentes componentes del sistema distribuido y dependiendo de donde se realice: a) el almacenamiento de los datos, b) el tratamiento de los datos para su producción y c) la coordinación y la integración de las diferentes producciones de los datos, se definen las diferentes arquitecturas de distribución.

Decouchant *et. al*. [9] señalan que: "*la arquitectura de distribución de un sistema groupware define los componentes y los mecanismos de administración asociados, que son ejecutados o usados en los diferentes sitios cooperativos, así como la localización de su ejecución o uso... la arquitectura de distribución influye directamente en las funcionalidades que provee el sistema distribuido (e.g. desempeño, eficiencia, tolerancia a fallas y escalabilidad)*".

La diferencia básica entre las diferentes arquitecturas de acuerdo a Patterson [8], es la forma en que se alcanza la consistencia y coherencia de los datos. "*Existen dos formas de alcanzar esta consistencia: manteniendo una sola copia del estado o manteniendo múltiples copias que están sincronizadas usando un protocolo de mantenimiento de consistencia*".

Arquitectura centralizada

La manera más sencilla de mantener el estado consistente de los datos es teniendo sólo una copia en un único servidor centralizado como lo muestra la figura 2.2. Los usuarios se conectan de manera remota al servidor y trabajan utilizando el protocolo

gráfico X11 windows, para ofrecer al usuario una interfaz de interacción remota, por lo que todos los procesos de almacenamiento y procesamiento de los datos se realizan en el servidor central. Como todos los procesos se realizan en la misma máquina, esta arquitectura es fácil de implementar y el control de la concurrencia se da de manera natural, de acuerdo al orden en que se registren las modificaciones en el servidor.

Pero esta arquitectura presenta varios inconvenientes:

- El tráfico que genera el protocolo X11 es elevado, debido a que toda la información de la interfaz del usuario es transmitida entre el sitio del servidor y el sitio del cliente. Se requiere un elevado ancho de banda para que la aplicación tenga un buen desempeño. Por esta razón, no es conveniente utilizar esta técnica de interacción entre los componentes distribuidos de una aplicación cooperativa en un ambiente remoto, donde el ancho de banda es un recurso limitado y poco fiable.

- El servidor centralizado se vuelve un cuello de botella, debido a que todo el procesamiento se realiza localmente. Esta arquitectura funciona bien cuando el número de usuarios es bajo, pero se vuelve inservible cuando el número de usuarios se incrementa.

- Esta arquitectura no es tolerante a fallas de la red o de las máquinas, debido a que todo el tratamiento se lleva a cabo en un único elemento (el servidor central) volviéndose un punto muy vulnerable, por lo que si falla o es inaccesible para algunos usuarios, no es posible seguir trabajando.

Arquitectura completamente distribuida o replicada

Buscando solucionar los problemas introducidos por una arquitectura centralizada, se propuso realizar una copia del estado de los datos compartidos en cada uno de los sitios de los colaboradores. Como se muestra en la figura 2.3, las modificaciones realizadas en los otros sitios se integran y validan por medio de un protocolo de difusión y actualización de copias, con el objetivo de eliminar completamente el servidor central. De esta manera, cada uno de los sitios trabajan de manera independiente sin los cuellos de botella inherentes a los servidores centralizados, haciéndolos tolerantes a los retrasos, a las fallas de la red o del servidor central o a las fallas de los sitios de los clientes. Las aplicaciones tienen un buen desempeño, ya que el almacenamiento y el tratamiento de las réplicas de los datos compartidos se realizan localmente. Inclusive es posible seguir trabajando de manera desconectada si se utiliza una política de replicación optimista, o se tiene el candado del fragmento que se está modificando. Para poder sincronizar los datos se requiere un protocolo de difusión/actualización de las copias, el cual genera un tráfico mucho menor que X11 windows.

Sin embargo una arquitectura completamente distribuida también tiene sus inconvenientes:

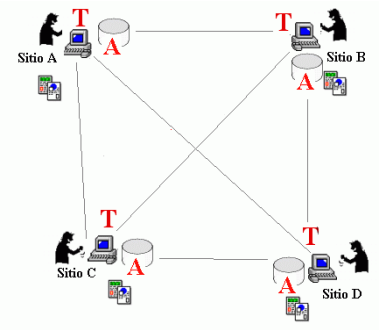

Figura 2.3: Arquitectura completamente replicada

- La implementación es más compleja, debido al protocolo de difusión y actualización de las copias, aunque éste es un problema a nivel de programación y no a nivel de colaboración.

- La existencia de varias copias de la información puede generar inconsistencias, debido a los retrasos de la red o a las fallas de los sitios, ocasionando conflictos entre los usuarios o diferencias en la percepción del ambiente compartido. Dependiendo de la naturaleza de las aplicaciones, éstas pueden ser más o menos complejas de solucionar, como se vió en la subsección 2.1.2 referente al control de concurrencia.

- En principio se requiere conocer las direcciones de los dispositivos conectados previamente, para poder conectarse directamente con los otros usuarios. Sin embargo, en un ambiente móvil con direcciones dinámicas basadas en DHCP no es posible, ya que éstas cambian constantemente.

- Los dispositivos deben ser capaces de almacenar y realizar el tratamiento de los datos. Hoy en día, los celulares o PDAs que utilizan algunos usuarios para conectarse a la Web, no cuentan con las capacidades mínimas para poder utilizarlos como un sitio completamente autónomo, que proporcione todos los servicios de almacenamiento y procesamiento.

- Algunas aplicaciones, dependiendo de sus características como el tamaño de los fragmentos y el tipo de sincronía (*i.e.* síncrono o asíncrono) escalan pobremente cuando se incrementa el número de colaboradores y el volúmen de información, debido a que cada sitio tiene que notificar a todos los demás. Por ejemplo, si se tienen 10 participantes cada sitio deberá notificar a los otros 9, por lo que el número de enlaces requeridos por la aplicación es de 90, incrementado el tráfico en la red.

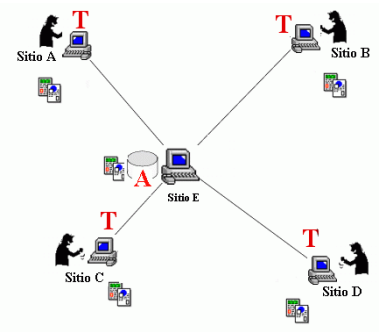

Figura 2.4: Arquitectura híbrida

Arquitectura híbrida

Existe una variante derivada de la combinación de las arquitecturas centralizada y completamente replicada. Esta arquitectura se le conoce como híbrida. Como vemos en la figura 2.4 , esta arquitectura centraliza el almacenamiento de los datos, pero el tratamiento se lleva a cabo en los sitios de los colaboradores.

A pesar de que la arquitectura híbrida mejora el desempeño de las aplicaciones, al distribuir el tratamiento de los datos y simplificar los problemas inherentes al control de concurrencia y al llevarse a cabo las actualizaciones en la copia centralizada en el servidor, esta arquitectura tiene muchos problemas:

- El servidor central, donde se lleva a cabo el almacenamiento de los datos, sigue siendo un punto vulnerable de esta arquitectura, ya que no es posible trabajar si falla o es inaccesible debido a problemas en la red.
- Los usuarios no pueden trabajar desconectados, debido a que siguen dependiendo del servidor central para el almacenamiento de los datos.
- Los dispositivos deben ser capaces de realizar el tratamiento de los datos, por lo que los dispositivos de baja capacidad no siempre son capaces de ejecutar las aplicaciones colaborativas.

En la práctica, existe muchas variantes derivadas de las arquitecturas anteriormente mencionadas, con la intención de solucionar los problemas inherentes a cada una de ellas, permitiendo mejorar el desempeño y la disponibilidad de las aplicaciones colaborativas distribuidas. Las variaciones se basan en los siguientes factores:

- El uso de **múltiples servidores de replicación y cachés de los datos compartidos** para incrementar la disponibilidad de las aplicaciones distribuidas. Se pueden tener múltiples servidores que funcionen como respaldo o distribuyendo la carga, ya que los clientes son atendidos por más de un servidor. En este caso, es necesario sincronizar las diferentes copias en los servidores.

- El uso de **servidores que sirven como punto de encuentro** en un ambiente de direcciones dinámicas.

- El uso de **servidores que apoyen a los dispositivos de bajas capacidades (*e.g.* dispositivos móviles)**. Estos dispositivos disponen de un espacio reducido para el almacenamiento de los datos, baja capacidad para el tratamiento de mismos y/o la incapacidad de manejar la distribución de las notificaciones. Normalmente se realizan réplicas de la información, ya sea parciales o totales para poder ser tratadas por éstos.

Independientemente del tipo de arquitectura que se seleccione, es necesario considerar otros aspectos relacionados con la distribución y notificación de los mensajes de actualización de los datos compartidos entre los diferentes componentes de la aplicación colaborativa distribuida:

- **El número de réplicas** que se pueden actualizar simultáneamente de una entidad compartida y distribuida. Una réplica que puede actualizarse se conoce como copia maestra. Varias copias maestras incrementan la disponibilidad y mejoran el desempeño de las aplicaciones, pero aumentan la complejidad de la implementación al tener que sincronizarlas.

- **El tipo de transferencia** de la cual existen dos variantes: transferencia del estado también conocido como actualizaciones por valory transferencia por lista de las operaciones. La primera consiste en enviar la entidad o el fragmento de entidad resultante una vez que se llevan a cabo los cambios. En la segunda se envian las operaciones que permiten llegar al estado final. Si los objetos son grandes es mejor enviar las operaciones.

  Pero si los objetos son pequeños, una mejor opción puede ser transferir el estado, aunque también se debe considerar el número de operaciones necesarias para alcanzar el estado final. Por ejemplo, después de una desconexión, el número de operaciones se incrementa por lo que es conveniente transferir el estado.

- **La frecuencia de las sincronizaciones**, se puede sincronizar la entidad cada vez que un sitio consumidor de notificaciones lo requiera (*pull*) o cada vez que un sitio productor de modificaciones genere un cambio (*push*). Si se actualiza una aplicaciÃşn sÃşlo cuando requiere los datos, se puede deteriorar la conciencia de grupo de los colaboradores, debido a que no están enterados de los cambios. Cuando se actualiza utilizando *push* se realizan muchas actualizaciones que no son requeridas por los usuarios, incrementando el procesamiento de los mensajes de notificación; además la aplicación requiere un mayor ancho de banda, debido al incremento del tráfico ocasionado por las continuas notificaciones de las actualizaciones de los datos compartidos.

- **La arquitectura de la propagación de las notificaciones**, permite distribuir las modificaciones a las entidades, diseminando dichos cambios entre todos los sitios

de colaboración. Las estrategias básicas para resolver este problema de acuerdo con Tannenbaum [12] son: organizar los nodos en un árbol, de tal manera que exista una ruta única entre cada par de nodos u organizar los nodos en una red acoplada en la que un nodo tendrá varios vecinos con múltiples rutas, garantizando que las notificaciones sean entregadas. A partir de éstas dos estrategias es posibles crear múltiples soluciones. Si sólo existe una ruta entre dos sitios, se corre el riesgo de que no se propaguen todas las notificaciones. Si existen varias rutas de propagación de los cambios entre los sitios, el tráfico se incrementa considerablemente.

Al diseñar la arquitectura de propagación de las notificaciones es necesario considerar: a) el ancho de banda disponible, b) la frecuencia de las actualizaciones y c) el número de sitios a notificar.

- **La manera en que se realiza la confirmación de las operaciones**: las operaciones realizadas en las entidades distribuidas son enviadas a todos los sitios de colaboración. Cuando se utilizan políticas optimistas, las notificaciones pueden ser rechazadas por los sitios debido a las inconsistencias (*e.g.* cambiar el color de una casa que fue borrada) o confirmadas. Una vez que todos los sitios confirman la operación, se sabe que ya no va haber conflictos con otras operaciones. La confirmación de las operaciones puede ser de manera implícita por conocimiento común, al tener conciencia de las operaciones que han recibido todos los sitios. También se puede detectar utilizando vectores de confirmación en conjunto con vectores de relojes o matrices de marcas de tiempo. Otra manera es la confirmación por concenso (ver Saito y Shapiro [7]).

Arquitecturas de coordinación

De acuerdo con Decouchant [9] el servicio de coordinación tiene como función integrar las modificaciones de las entidades compartidas, creando versiones consistentes de dichas entidades. El servicio de coordinación se encarga de resolver los problemas relacionados con las inconsistencias, ocasionadas por las modificaciones paralelas y concurrentes de las entidades . El servicio de coordinación también tiene las funciones de:

- definir los actores del proceso de colaboración en términos de personas, grupos, roles o agentes inteligentes de software;
- identificar las actividades y tareas (colaborativas o no), y más particularmente las relaciones temporales entre ellas y los actores;
- definir los actores que están a cargo de las diferentes tareas y las actividades.

Para definir los actores de un sistema cooperativo, es necesario establecer los roles de los usuarios, ya que no todos tienen las mismas responsabilidades ni deben realizar las mismas funciones, por lo que no todos deben poder realizar cualquier acción en

cualquier entidad compartida. Una de las funciones de la arquitectura de coordinación es la administración del control de acceso, la cual permite restringir u otorgar el acceso a los diferentes colaboradores. Para definir las acciones que un usuario puede realizar se utilizan las políticas de control de acceso. Edward Keith [6] define una política en groupware como: "*principio o plan general que guían las acciones que gobiernan como interactúan los usuarios y las aplicaciones*". Las políticas de control de acceso son aplicadas a los roles de los usuarios. De acuerdo con Edwad Keith [6] "*un rol es una categoría particular de usuario con un conjunto de derechos de acceso*". Las roles pueden definirse de manera estática/*a priori* (*e.g.* una lista de nombres) o dinámicos, (*i.e.* que pueda variar de acuerdo al ambiente en donde se desenvuelven los usuarios otorgando los privilegios cuando se cumpla una condición dada, por ejemplo un periodo de tiempo o una ubicación determinada).

De manera idéntica a la estructuración de las arquitecturas de replicación de los datos compartidos, la arquitectura de coordinación puede ser centralizada la cual, de acuerdo con Decouchant [9]; consiste en concentrar las funciones de coordinación en un servidor dedicado. Esta arquitectura es la más simple de implementar, pero es vulnerable a fallas (fallas del servidor y/o de la red de comunicación). Además, el sitio donde se encuentran se vuelve un cuello de botella, al tener que llevar a cabo toda la coordinación.

Otra arquitectura es la coordinación completamente distribuida entre todos los sitios. Ésta proporciona una mayor disponibilidad de las aplicaciones colaborativas distribuidas, pero los algoritmos que permiten llegar a la consistencia de los datos son más complejos. Sobre todo si se utilizan políticas optimistas basadas en la serialización como se vió en la subsección 2.1.2, en ocasiones es complejo, sino imposible, llegar a un estado consistente de los datos.

2.1.4 Protocolos de comunicación

La Web y la mayoría de las intranets actuales utilizan la *suite* de protocolos de TCP/IP. La *suite* proporciona los protocolos TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*) para la comunicación extremo a extremo (sin importar los diferentes dispositivos de comunicación intermedios por los que pasan los paquetes de datos como son *routers*, *lan-switches*, *gateways*, etc.), mediante los cuales se comunican los protocolos de comunicación de las aplicaciones. Cada uno de estos protocolos tiene ciertas características que permiten un mejor manejo en las notificaciones de los datos. Estas características se deben considerar antes de elegir el protocolo de comunicación.

TCP (*Transmission Control Protocol*)

TCP es un protocolo orientado a la conexión entre los usuarios. Debido a que IP es un protocolo que no garantiza la entrega de los paquetes, TCP verifica la entrega y controla el flujo del envío de los mismos. También establece/termina la conexión entre los

dispositivos finales (computadoras, servidores, celulares, etc.). Las características de TCP que influyen en el comportamiento de las aplicaciones colaborativas distribuidas son las siguientes:

- Cuenta con un control de flujo que limita el envío de paquetes cuando la red está lenta, aseguarando que tanto las aplicaciones como los elementos intermedios de la red, no se saturen.
- Cuenta con un control de secuencia, por lo que entrega a la aplicación los paquetes en orden.
- Confirma la entrega de los paquetes, garantizando que todo lo que es enviado sea entregado, si se pierde lo reenvía.

Estas características hacen de TCP un protocolo robusto, ideal para el envío de datos, pero debido al control de flujo y el reenvío de paquetes genera latencia en la entrega de los datos a las aplicaciones, por lo que no se recomienda en datos altamente síncronos. También requiere un mayor ancho de banda, ya que requiere incluir los campos necesarios que le permitan garantizar las características antes mencionadas.

UDP (*User Datagram Protocol*)

A diferencia de TCP, UDP es un protocolo ligero que no establece una conexión previa antes de enviar los mensajes. Fue diseñado para proveer un servicio en donde los mensajes sean intercambiables. Las características que debemos considerar antes de utilizar UDP son:

- Es un protocolo ligero, es decir sus mensajes son pequeños, por lo que requiere un menor ancho de banda.
- No cuenta con un control de flujo, las aplicaciones deben tomar por su cuenta la velocidad en que se transmiten los paquetes.
- No garantiza la entrega de paquetes, por lo que puede haber pérdidas; de esta manera las aplicaciones tienen que solicitar el reenvío o ser capaces de manejar la pérdida.
- No garantiza la entrega ordenada de los mensajes, por lo que las aplicaciones deben establecer la secuencia.

Por sus características UDP no se recomienda para el envío de archivos de datos. Normalmente se utiliza para aplicaciones ligeras como la gestión de dispositivos o aplicaciones altamente síncronas que soporten pérdidas de datos pero no retrasos. Un ejemplo de este tipo de aplicaciones son las videoconferencias, ya que si se pierde un

"cuadro", los usuarios no lo detectan debido a que por lo regular se utilizan 24 cuadros por segundo. A veces se congela la imagen momentáneamente, pero la videoconferencia puede continuar si las pérdidas no son muchas.

Encima de estos protocolos se montan los protocolos de comunicación de las aplicaciones, por ejemplo HTTP es el principal protocolo que utilizan los los navegadores para buscar información en la Web. SMTP es el protocolo utilizado entre los servidores de e-mail para enviar los mensajes. Algunas aplicaciones utilizan protocolos propietarios para la comunicación de los sitios, por ejemplo los chats.

2.2 Aplicaciones colaborativas y la colaboración en la Web

Como introduce Ellis *et. al* [1], las aplicaciones *groupware* se pueden diferenciar siguiendo una matriz tiempo-espacio, que permite clasificarlas en aplicaciones con interacciones síncronas y asíncronas, que pueden igualmente permitir una cooperación colocalizada o en ambiente distribuido.

Así, se puede constatar que existen múltiples tipos de aplicaciones *groupware*, con características muy variadas. Para que la colaboración se dé de manera adecuada, los requerimientos de los usuarios en cuanto a desempeño de la aplicación varían considerablemente dependiendo del tipo de aplicación, del número de usuarios, de la distribución geográfica, de la posibilidad o necesidad de movilidad de los usuarios, del tipo de conciencia de grupo requerido, etc.

Con la creciente difusión y disponibilidad del Internet, muchas empresas han desarrollado aplicaciones colaborativas, geográficamente distribuidas, que aprovechan el uso de la Web, permitiendo que sus empleados trabajen y colaboren de manera remota.

Pero las aplicaciones propias de la Web, que son utilizadas por un gran número de usuarios, geográficamente distribuidos, requieren un análisis detallado para que tengan un buen desempeño.

A continuación se analizan algunos tipos de aplicaciones colaborativas, representativas de este dominio de investigación.

2.2.1 Juegos cooperativos

Este tipo de aplicaciones cooperativas, permiten a un grupo de usuarios competir entre ellos al jugar en línea, aconteciendo el juego en un ambiente virtual compartido. Algunos juegos permiten crear equipos los cuales se comunican entre sí mediante voz o textos.

Los juegos en línea se pueden dividir en dos categorías, aquellos basados en turnos (*e.g.* cartas, damas chinas, *etc.*) o juegos de ambiente compartidos donde los usuarios realizan acciones concurrentes libremente. Todas son aplicaciones orientadas a conexión, por lo que jugar fuera de línea (*i.e.* desconectado) no tiene sentido.

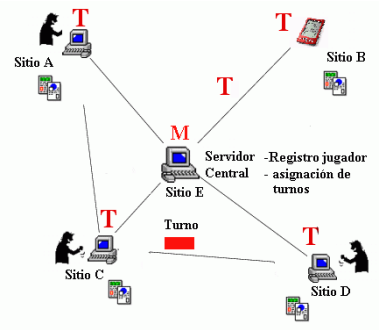

Figura 2.5: Arquitectura de juegos por turnos

Juegos cooperativos basados en turnos

La dinámica de este tipo de juegos consiste en: un grupo de jugadores se reunen y se asignan turnos para jugar. En este tipo de aplicaciones difícilmente habrá un conflicto entre los datos, debido a que están basadas en turnos y cualquier acción realizada cuando la persona no tiene el turno, debe ser rechazada. Las personas pueden aceptar pequeños retrasos debido a que tienen que esperar a que los demás jugadores realicen su jugada. Por este motivo, los usuarios no son sensibles a la latencia de la red, mientras ésta no sea significativa. El ancho de banda requerido es mínimo, y por lo regular el número de jugadores es pequeño debido a los tiempos de espera ocasionados por los turnos. Este tipo de aplicaciones cooperativas están basadas en una arquitectura P2P (ver la figura 2.5). Ésta incluye un servidor centralizado que actúa como punto de reunión y de descarga de la aplicación por parte de los jugadores. Para implementar este tipo de aplicaciones se utiliza un turno o *token* que indica el turno de los jugadores. Para iniciar el juego el servidor central, donde se encuentra la aplicación, registra a los jugadores y les asigna los turnos.

Este tipo de aplicaciones pueden ejecutarse en dispositivos móviles. El problema es que no todos estos dispositivos son capaces de manejar conexiones P2P. Como se ve en la figura 2.5 (sitio B), el servidor central debe ayudarlos a realizar las notificaciones al resto de los sitios, ya que no cuenta con una conexión directa con los demás. El sito B debe pasar el *token* al servidor central y éste a su vez, notificar las acciones de este sitio y pasar el *token* a los otros.

Como las aplicaciones de este tipo son ligeras y el número de jugadores es pequeño, la única forma en que el servidor centralizado pueda ser un cuello de botella es cuando se incrementa el número de instancias del juego (*i.e.* juegos simultáneos) y la solución para resolver este problema es: distribuir el número de instancias entre más de un servidor, lo cual incrementa también la disponibilidad de la aplicación, ya que un servidor puede funcionar como respaldo del otro si se le envían las notificaciones del estado del juego. En caso de falla del servidor central, se ejecuta una excepción en los dispositivos para que se conecten al servidor de respaldo. Para soportar rupturas de los enlaces de comunicación, los servidores deben estar en diferentes locaciones.

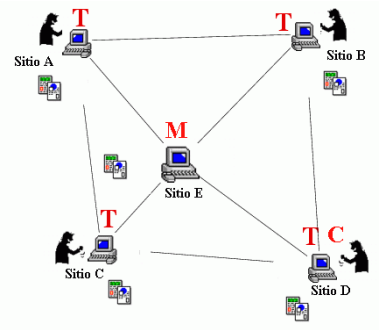

Figura 2.6: Arquitectura de juegos colaborativos en ambiente compartido

Este tipo de aplicación es sencilla de implementar si se usa la arquitectura y el control de concurrencia adecuados. Por ejemplo, si se utilizan candados, se genera más tráfico al tener que solicitarlos y removerlos. Si se utiliza una política optimista puede generar inconsistencias y empobrecer el desempeño de la aplicación al tener que deshacer las operaciones.

Para garantizar que las actualizaciones lleguen a todos los elementos de la red y detectar cuando un usuario se desconecta se recomienda utilizar como protocolo de comunicación TCP.

Juegos cooperativos en ambiente compartido

Los jugadores se encuentran inmersos en un ambiente virtual, en donde pueden moverse y actuar libremente. Los jugadores se conectan a un servidor central para obtener la aplicación. La coordinación de estos juegos es centralizada, ya sea en el servidor del punto de reunión, o en alguno de los sitios donde se encuentran los jugadores. Como se muestra en la figura 2.6, el sitio D se encarga de coordinar los eventos de los demás. Las acciones de los jugadores son enviadas al servidor central, asignándoles un orden de acuerdo como va recibiendo las notificaciones. Por su naturaleza utilizan una política de control de concurrencia optimista, basada en marcas de tiempo, *i.e.* se determina el orden de las acciones de los jugadores con base en el tiempo en que fueron realizadas.

Debido a que el sitio donde se lleva a cabo la coordinación designa el momento en que ocurrieron los eventos, la latencia de la red puede influir en los resultados, ya que modifica la conciencia de la ubicación de los demás participantes e influye para determinar el momento en que ocurrieron los eventos. Por esta razón, creemos que la coordinación debe estar distribuida entre los diferentes sitios. Para poder determinar de manera global el orden de los eventos, los sitios deben estar sincronizados.

Este tipo de aplicaciones son compleja de distribuir y escalar debido a que son muy sensibles a la calidad de la comunicación entre los participantes. Se necesita conocer la correcta ubicación de los demás jugadores y sus acciones para que el juego pueda ser coherente. Por esta razón, todas las acciones deben ir encaminadas a reducir la latencia

Figura 2.7: Juego *World of Warcraft*

de las notificaciones:

- La notificación de los eventos debe tener prioridad sobre el almacenamiento de los datos.
- Los datos deben ser clasificados en eventos trascendentes que afectan la evolución del juego, por lo que las notificaciones no deben extraviarse (*e.g.* se murió tal jugador, el jugador tomó el arma, etc.) y no trascendentes, los cuales proporcionan información adicional respecto al estado y ambiente de los jugadores, pero que las pérdidas eventuales no afectan la percepción de los jugadores.
- El protocolo de comunicación para las notificacioens debe ser UDP debido a que que el control de flujo de TCP puede introducir latencia en las actualizaciones de los eventos, pero para los eventos trascendentes se debe garantizar la entrega.
- Las capa de distribución debe implementar el orden de las notificaciones (serialización).
- Sería interesante ver el comportamiento de las aplicaciones utilizando una coordinación *peer to peer* ya que la coordinación centralizada introduce pequeños retrasos en las notificaciones.
- El desempeño de estas aplicaciones se puede mejorar si se utilizan redes basadas en protocolos con diferenciación de tráfico (*e.g. MPLS o DiffServ*), etiquetando los mensajes de notificación con una alta prioridad sobre el resto del tráfico de la red, para que sean entregados más rápidamente.

Hoy en día, se encuentran disponibles en la red, varios sitios que permiten jugar en línea, ya sea en la Web o en una red local.

Ejemplos de estos tipos de juegos son *Halo*, *Ghost*, *World of Warcraft*, *Quake*, *CounterStrike*,etc. La figura 2.7 muestra una vista del juego *World of Warcraft*, en la cual se puede ver 2 jugadores.

Un ejemplo: *Xbox Live*

Xbox Live es una aplicación colaborativa, la cual permite a jugadores distribuidos en Internet, conectarse a través de la red para jugar en línea. Comparten un ambiente virtual en donde se enfrentan o colaboran para ganar en el juego. Los jugadores pueden comunicarse entre sí por medio de mensajes escritos via de teclado, o por voz mediante una diadema telefónica.

Los jugadores se encuentran distribuidos geográficamente, principalmente en Estados Unidos, Europa y Japón. En México existe una densificación considerable de jugadores en la zona centro del país. *Xbox live* permite enfrentar a diferentes jugadores de acuerdo a su habilidad, pero no pone restricciones geográficas. Así pueden jugar alguien en México, Estados Unidos e Inglaterra simultáneamente, por lo que la distribución de los jugadores afecta la eficiencia del juego.

Con base en la investigación realizada por Lee *et. al* [17], los juegos cooperativos utilizan dos arquitecturas. Algunos como *MMORPGs*, *Quake*, o los juegos casuales basados en la Web utilizan una arquitectura cliente-servidor. El servidor central se encarga de la coordinación. Otros utilizan una arquitectura *P2P*, en donde sólo utilizan un servidor como punto de reunión y después se comunican directamente entre ellos. La coordinación del juego se lleva a cabo en uno de los dispositivos de los jugadores, por lo que utilizan una coordinación centralizada.

El problema de utilizar una coordinación centralizada es que el desempeño de la aplicación depende considerablemente de la calidad de la comunicación con la que cuentan los sitios de colaboración hacia el servidor central. Cuando el enlace es bueno, los jugadores conocen la ubicación real de los demás y sus acciones se ven reflejadas correctamente. Pero cuando existen retrasos en la red, los jugadores creen que los demás están en cierta posición, pero esa era la posición de los jugadores segundos atrás. De igual manera, cuando hay retrasos en la red, las acciones de los jugadores llegan al segundos después servidor encargado de la coordinación, por lo que sus acciones son registradas algún tiempo después de que se llevaron a cabo. Esto ocasiona inconsistencias entre lo registrado y la realidad de los eventos, haciendo imposible jugar.

Para poder solucionar estos problemas es necesario acercar la coordinación a los jugadores, ya sea mediante servidores distribuidos geográficamente, que se encarguen de registrar los eventos en el momento real de su ocurrencia, o en el caso de los juegos *P2P*, en vez de seleccionar un *host*, se manejan vectores de relojes y se sincronizan los eventos por votación.

2.2.2 Mensajería instantánea

La mensajería instantánea se ha vuelto muy popular en nuestros días, permitiendo a los usuarios mandar mensajes de texto y compartir documentos. Algunos ofrecen una noción de co-presencia: muestra la foto de los usuarios e íconos especiales que expresan su humor.

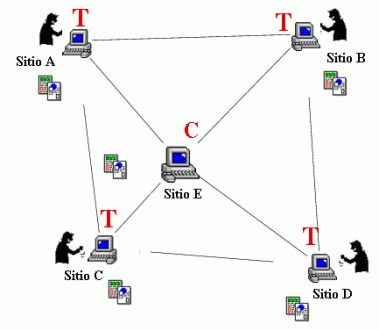

Figura 2.8: Arquitectura de chat

Aunque la mensajería instantánea es síncrona y los retardos en la información pueden resultar incómodos, no es tan sensible a éstos como las videoconferencias o los juegos cooperativos. Como los mensajes y los archivos son datos, no debe haber pérdidas de las notificaciones por lo que se recomienda un protocolo que garantice la entrega de los mensajes como TCP. La granularidad de los datos es a nivel mensaje. En este tipo de aplicaciones no existen modificaciones a los mensajes, ni siquiera se requiere almacenar los datos. Los nuevos mensajes son regularmente actualizados en las interfaces de interacción de los participantes y no requieren un orden estricto ya que si dos usuarios envían un mensaje de manera simultánea, no afecta que en la interfaz de un usuario aparezcan en un orden diferente. Además los usuarios por lo regular siguen un protocolo social, en donde vuelven a enviar un mensaje hasta que llega la respuesta del mensaje anterior. Debido a estas características no se requiere una política de control de concurrencia.

Como vemos en la figura 2.8, para mejorar la comunicación entre los particitantes se puede utilizar una arquitectura híbrida, en donde el servidor central sólo sirve como punto de reunión y se encarga de autenticar a los usuarios y notificar a los demás participantes cuando un usuario se conecta o se desconecta. Los usuarios se envían los mensajes directamente entre ellos, sin necesidad de pasar por el servidor central. Debido a que no requieren un control de concurrencia, la implementación de esta arquitectura es fácil.

Generalmente las conversaciones entre los participantes no son almacenadas, pero de requerirse, se pueden almacenar en el servidor central y establecer el orden conforme los mensajes arriben a éste.

Algunos ejemplos de mensajería instantánea contituyen: *Exodus* [42], *Kopete* [45], *aMSN* [50], *Pidgin* [46], *Messenger Live* [47], *GoogleTalk* [48], etc. La figura 2.9 muestra la interfaz de usuario de Exodus.

2.2.3 Agendas colaborativas

Las agendas colaborativas permiten administrar, organizar y visualizar las actividades de un grupo de trabajo, planificar las actividades de los participantes, compartir con-

Figura 2.9: Mensajería Instantánea

tactos y compartir notas. Los usuarios tienen un espacio privado y unos espacios compartidos con diferentes grupos, por lo que se debe tener un control de acceso diferente para cada uno de los espacios que maneja el usuario.

Este tipo de aplicaciones son asíncronas (*i.e.* soportan el trabajo de los usuarios aún cuando estén desconectados), aunque algunos cuentan con chats o foros de discusión, los cuales requieren sincronía; estas funcionalidades adicionales deben analizarse de manera independiente de la aplicación principal. El protocolo de comunicación propuesto es TCP, debido a que los datos que comparten las diferentes agendas deben ser consistentes. La granularidad de los datos puede ser la misma que utilice el calendario para reservar una cita (*e.g.* día, hora, minutos, etc).

Para este tipo de aplicaciones cooperativas se recomienda utilizar una política de control de concurrencia optimista. Una cita tiene dos fases, cuando se propone y cuando es validada por el sistema y los demás usuarios implicados. Las inconsistencias deben ser resueltas por políticas de la aplicación de acuerdo a los roles de los usuarios, políticas de transformación debido a diferencias en datos distintos (*e.g.* el sitio y la hora) o por los usuarios. Se puede usar una política optimista para soportar desconexiones y que en el momento en que surge una nueva actividad no se olvide agendarla. Además los usuarios requieren tener su agenda siempre a la mano, independientemente de que estén conectados o no. Cuando ocurra una inconsistencia el usuario debe elegir entre un dato y otro. Esto no es algo molesto debido a que sólo debe aceptar o rechazar la nueva actividad. Además debemos tener presente que las agendas son sólo recomendaciones a los usuarios de las actividades que pueden realizar; si una actividad es concurrente con otra se notifican ambas y el usuario decide que hacer. Cuando el usuario no desea citas en ciertos horarios, se puede utilizar de

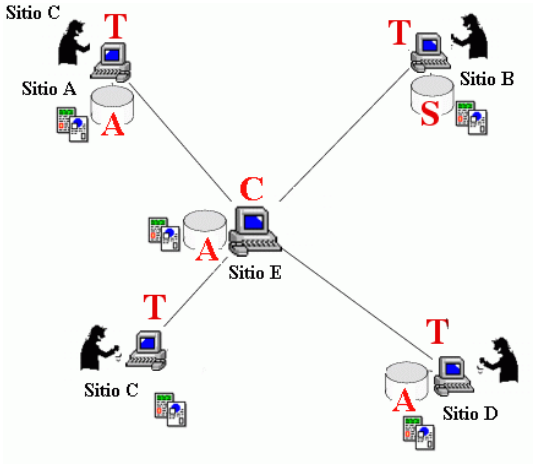

Figura 2.10: Arquitectura de distribución asociada a una agenda

manera conjunta con la política optimista, una política pesimista basada en candados para bloquear ciertas áreas de la agenda.

Como se puede ver en la figura 2.10 podemos utilizar una arquitectura completamente replicada para almacenar los datos. Para coordinar las actualizaciones se puede tener un servidor centralizado. Para soportar la movilidad de los usuarios, se debe almacenar una copia de los datos en los dispositivos de los usuarios (ya sea parcial o total dependiendo de la capacidad de los dispositivos) y cuando se tenga acceso al servidor central se deben resolver las inconsistencias. Para detectar las inconsistencias de los datos en la diferentes réplicas de la información se puede utilizar un vector de relojes lógicos [11], de tal forma que cada réplica lleve un control de los cambios realizados localmente. Se puede implementar un servidor de respaldo para incrementar la disponibilidad de la aplicación, el cual deberá tomar el control en caso de que el servidor principal falle o sea inaccesible.

Ejemplos de calendarios colaborativos son Team Integrator[43], Carthago[44], Horde [49], Outlook[51], MAPilab[40], *etc.* En figura 2.11 se presenta una vista de MAPilab en donde las actividades de diferentes usuarios se presentan con diferentes colores. Esta aplicación maneja grupos de usuarios con diferentes niveles de permisos. De acuerdo al rol, el usuario puede ver o no las actividades del grupo, modificarlas, asignar nuevas tareas, *etc.* Las actividades registradas se dividen en actividades privadas y actividades compartidas: las privadas sólo pueden ser vistas por el usuario y las compartidas por el grupo de trabajo.

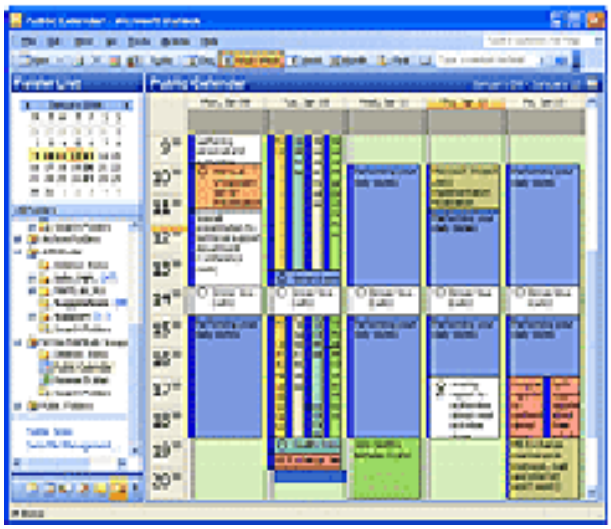

Figura 2.11: Calendario de MAPilab

2.2.4 Editores cooperativos

Los editores cooperativos permiten a un grupo de trabajo crear documentos, sincronizando las diferentes copias generadas por los usuarios. De esta manera los usuarios son liberados del trabajo de la integración de las producciones de los documentos. También proveen el acceso a los revisores del documento, reduciendo el tiempo de aprobación. Algunos de estos editores se encargan de administrar este proceso, permitiendo conocer el estatus del documento durante su creación y aprobación.

En algunos editores, aunque requieren que los documentos sean sincronizados, los usuarios pueden desconectarse y seguir trabajando en alguna sección. Por esta razón son síncronos/asíncronos, dependiendo de si los usuarios pueden seguir trabajando aún cuando se desconecten temporalmente. Para soportar mayor concurrencia los documentos pueden ser fraccionados, *i.e.* divididos en capítulos, secciones o párrafos.

Normalmente, los usuarios trabajan partes separadas de un documento, a menos que de común acuerdo decidan producir versiones concurrentes para mezclarlos posteriormente, por lo que normalmente se utilizan candados para el control de concurrencia. Para permitir que diferentes usuarios puedan concurrentemente producir una misma parte del documento, se puede implementar una variante utilizada por WebDAV [14][15][16] conocida como candados compartidos. Un usuario puede modificar el documento que está controlado por otro, pero sólo con la autorización explícita de aquél que tiene el candado. Si se implementa esta política es necesario incluir en la aplicación las funciones que permitan la fusión de las dos versiones.

Algunas plataformas utilizan políticas optimistas suponiendo que los usuarios no modificarán un fragmento del documento si son concientes de que otro lo está modificando, por lo que requirieren de una conciencia de grupo fuertemente acoplada. Pero no es posible soportar desconexiones temporales ya que afectan la conciencia de

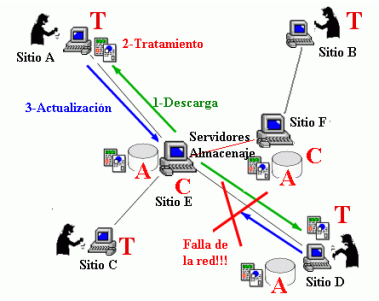

Figura 2.12: Arquitectura para editores de documentos

grupo. Por lo que siempre es mejor garantizar que no se produzcan modificaciones concurrentes debido a que no existen las funciones de transformación que permitan integrar las dos versiones de manera satisfactoria. Ésto ocasiona que los usuarios deben hacer la integración de las versiones de manera manual, lo cual implica retrabajarlas o eliminar alguna de ellas. Algunos editores como DTWiki [13] eligen de manera arbitraria una de las versiones (normalmente la que es almacenada al último), perdiendose los cambios de los otros usuarios.

Otros editores como los basados en COAST [24] utilizan operaciones de *undo* y *redo* para reordenar los eventos, pero a veces el documento se corrompe ocasionando que la versión resultante sea ilegible.

Cualquiera de estas dos soluciones resulta molesta para los usuarios si no se estableció una autorización previa por parte de los mismos, por lo que siempre que existan modificaciones concurrentes del mismo fragmento, debe haber una autorización explícita por parte de los usuarios. Cuando se utilizan candados no es relevante el orden de los eventos, ya que siempre llegan a un estado consistente de los datos. Pero si se utiliza una política optimista basada en estampas de tiempo (*e.g.* la versión correcta es la última desde el punto del orden temporal definido), importará para determinar cuál es la última versión.

Debido a que los editores manejan archivos de datos y las notificaciones deben llegar íntegras, se recomienda utilizar TCP como protocolo de comunicación para garantizar la entrega de los mensajes.

Las arquitecturas que mejor se adaptan a un control de concurrencia basado en candados son aquellas que utilizan una coordinación centralizada (*i.e.* en un servidor único), pero esta arquitectura limita la movilidad de los usuarios, por lo que se puede utilizar una coordinación centralizada a nivel fragmento. La coordinación por fragmento consiste en tener una sóla copia maestra de cada fragmento y muchas réplicas de sólo lectura. El sitio que tiene el fragmento maestro, es el único que coordina las modificaciones en éste, ya sea via un *proxy* (*i.e.* representante de las entidades compartidas que se encuentra en un sitio remoto) o localmente. Si un usuario desea seguir trabajando a pesar de las desconexiones, se realiza una réplica y se obtienen las copias maestras de los fragmentos que desea modificar como lo propone *PIÑAS* [9] y *DreamTeam* [23]. Para mejorar el desempeño de la aplicación el tratamiento de los datos

debe realizarse del lado de los usuarios. La figura 2.12 muestra una arquitectura con dos servidores para el almacenamiento y la coordinación de los documentos (sitios E y F) y los usuarios se pueden conectar mediante un *proxy*. Podemos observar que el sitio D cuenta con un enlace poco fiable. Por esta razón, se le permite crear una réplica del documento con la finalidad de poder soportar desconexiones, mientras que aquellos usuarios que trabajan en un ambiente fiable o no cuentan con la capacidad de almacenamiento utilizan un *proxy*.

Un ejemplo de un editor de documentos colaborativo es *Alliance* [10]. Éste permite crear documentos Web de manera segura y consistente. Provee los servicios de autenticación de los usuarios, el servicio de nombres para los documentos y los dispositivos, permite compartir y administrar los documentos, soporta la replicación y administra el almacenamiento consistente de los archivos. Utiliza una arquitectura híbrida cuando los usuarios cuentan con un enlace confiable, pero permite la replicación de los documentos con el objetivo de soportar desconeciones temporales.

Los sistemas Wikis de la Web

Un ejemplo actual y muy representativo de los editores de texto en la Web son los *Wikis*, los cuales han probado ser herramientas valiosas para la generación y la administración colaborativa de contenidos en la Web. Los *Wikis* están constituidos por un conjunto de páginas Web escritas en un lenguaje simplificado, permitiendo a los usuarios producir y publicar artículos como en *Wikipedia*. Éstos utilizan una semántica simple por lo que son fáciles de utilizar en áreas como la educación, la colaboración y la generación de contenido local. También es posible incorporar fácilmente anotaciones y discusiones al contenido principal. Así, muchas organizaciones no gubernamentales como centros de investigación, asociaciones y escuelas, distribuyen instantáneas del contenido de *Wikipedia* con fines educativos y/o informativos. Las razones proncipales del éxito de los *Wikis* son:

- Están provistos de una herramienta de fácil uso que permite crear y compartir documentos de contenido.
- No utilizan documentos estructurados, por lo que satisfacen las necesidades de muchas organizaciones. De esta forma los usuarios pueden compartir páginas sin ser forzados a seguir un cierto esquema o una semántica elaborada.
- Permiten incorporar anotaciones y discusiones fácilmente a través de la Web en el contenido principal.
- Muchos *Wikis* proveen un sistema de control de revisión.
- Muchas instituciones no gubernamentales esencialmente de educación, distribuyen instantáneas del contenido de la *Wikipedia* como parte de su material digital para los salones de clases.

Pero a pesar de su utilidad, el software actual no trabaja del todo bien: los *Wikis* utilizan particiones de red de acuerdo al lenguaje y al ambiente. Cada partición usa una arquitectura centralizada, que es completamente inoperante cuando el servidor central no está disponible. Dado que el Internet es un ambiente no confiable sujeto a fallas (red y/o máquinas), el uso de los *Wikis* se ve mermado. Existen soluciones que permiten redireccionar las llamadas a los *Wikis* cuando hay fallas. Por ejemplo los *web-catching proxies* realizan una copia de los documentos para acercarlos al usuario final y mejorar el desempeño de la aplicación. Otro ejemplo son las instantáneas o copias de la información. Éstas pueden servir para lectura cuando un *wiki* está desconectado, pero no es posible hacer actualizaciones debido a que el flujo de la información es en un sólo sentido, haciendo que muchas discusiones no queden registradas, por lo que sólo son soluciones parciales al problema de disponibilidad de los *Wikis*.

Hace algún tiempo se podían hacer modificaciones al contenido sin restricciones, es decir, libremente, pero debido a los conflictos ocasionados, la mayoría de los Wikis han implementado algún tipo de control de acceso, para la modificación de los documentos.

Existen algunos intentos por mejorar el desempeño de los *Wikis* como *DTWiki* y *Webdav*, pero ofrecen sólo una política de replicación, que no constituye la mejor solución en la mayoría de los casos. Además utilizan como granularidad de la información el *namespace* que es definido por *Wikipedia* [53] como: "*un conjunto de nombres en el cual cada uno es único*", *i.e.* el identificador del nombre del *Wiki* que direcciona hacia un página dentro de la Web, ocasionando múltiples inconsistencias de la información, debido a las actualizaciones concurrentes de las páginas.

Web Distributed Authoring Versioning (WebDAV)

WebDAV [14][15][16] fue propuesto por la IETF para soportar la producción cooperativa en la Web y aunque era la visión original de Tim Berners-Lee cuando creó el primer navegador, los navegadores subsecuentes como *Mosaic* [52] fueron de sólo lectura, concebidos únicamente para la consulta de la información.

Kim [16] define "*WebDAV como una suite de extensiones del protocolo HTTP/1.1 que soporta la autoría colaborativa y la administración del namespace, colocando y recuperando las propiedades de los metadatos, el control de acceso y el manejo de las versiones de los recursos en los servidores Web remotos*".

La intención de *WebDAV* de acuerdo con Whitehead [14] es de proveer una infraestructura para el soporte del trabajo colaborativo asíncrono que suministra una interfaz común con muchos tipos de repositorios.

WebDAV ofrece los mecanismos que permiten:

- La prevención de la sobreescritura mediante candados pesimistas, garantizando que sólo el dueño del candado puede escribir. Utiliza también una variante que permite al dueño de un candado compartirlo, dejando que los protocolos sociales

se encanguen del control de la concurrencia. Pero esta opción no siempre está disponible e introduce inconsistencias que deben ser resueltas posteriormente por los usuarios.

- El manejo de versiones de documentos Web.
- La administración avanzada de colecciones de recursos u otras colecciones definidas por los usuarios, permitiendo ordenarlas jerárquicamente.
- El control de acceso a los documentos compartidos.

WebDAV es un protocolo muy difundido, que es usado por diversos softwares. El servidor *HTTP Apache, Jakarta Tomcat, KDE Desktop, Microsoft Exchange, Microsoft IIS, Microsoft Windows XP, Gnome Desktop* con *Nautilus file manager, SAP NetWeaver, TikiWiki TikiDav*, entre otros, son ejemplos representativos de software que usan este protocolo.

El principio del funcionamiento de *WebDAV* está basado en una arquitectura híbrida, en donde el almacenamiento de los datos es centralizado en el servidor Web, y el tratamiento de los mismos es distribuido en cada sitio de los clientes.

Pero *WebDAV* tiene algunas limitantes importantes que restringen su uso:

- Las únicas políticas de control de concurrencia que maneja están basadas en candados pesimistas o candados compartidos, por lo que es poco flexible, limitando la colaboración.
- Este protocolo no es tolerante a fallas. Si una falla ocurre, el candado es liberado después de un tiempo (*timeout*), pero los cambios se pierden.
- WebDAV no soporta el trabajo móvil y/o nómada, debido a que no admite desconexiones.

Debido a sus limitaciones, WebDAV no es un protocolo que pueda soportar de manera adecuada la colaboración distribuida en ambientes poco fiables como el Internet, ni tampoco el ambiente de trabajo móvil y/o nómada.

DTWiki (*A Disconnection and Intermittency Tolerant Wiki*)

DTWiki [13] es un sistema Wiki que tiene como objetivo soportar el trabajo colaborativo de Wikipedia aún en ambientes con conexiones intermitentes. DTWiki utiliza una política de notificación optimista basada en *TierStore* que es un sistema de archivos distribuidos para aplicaciones tolerantes a retrasos. Se basa en el modelo de DTN (*Delay-Tolerant Networking*), el cual a) realiza una sincronización del contenido del sistema de archivos, b) ofrece replicación parcial de los datos compartidos y c) detecta conflictos de modificaciones concurrentes de archivos simples. *TierShore* garantiza la coherencia más que la consistencia, es decir sólo puede detectar los conflictos resultantes de las modificaciones concurrentes pero no los puede resolver.

Debido a la falta de semántica inherente a todos los Wikis, DTWiki utiliza una serialización sintáctica por lo que sólo detecta los conflictos y deja a los usuarios la tarea de corregirlos manualmente.

DTWiki constituye un intento interesante para mejorar el desempeño de los Wikis, pero el tamaño de los fragmentos es grueso (el *namespace*) lo que ocasiona muchos conflictos debido a las modificaciones concurrentes.

DTWiki se basa en una arquitectura híbrida, centralizada para el almacenamiento de los datos y distribuida para su tratamiento. Su mayor inconveniente consiste en que sólo permite un tipo de control de concurrencia y una sóla arquitectura, por lo que no es capaz de adaptarse a los diferentes ambientes de los usuarios. Además no ofrece soluciones a los conflictos ocasionados por las actualizaciones concurrentes, sino que el usuario debe resolverlos.

Platform for Interaction, Naming and Storage (PIÑAS)

PIÑAS [9] es una plataforma diseñada para la producción cooperativa de documentos Web. *PIÑAS* ofrece el servicio de identificación, el servicio de nombres únicos y la administración de autores, documentos, recursos, sesiones, proyectos, aplicaciones y eventos.

Maneja y ofrece mecanismos de fragmentación, control de acceso, control de concurrencia, replicación y actualización automática en un ambiente distribuido.

Para la resolución de los nombres de los servidores utilizan el sistema de designación de la Web (*i.e.* URL's) y asegura la autenticación de los usuarios, lo que permite identificarlos de manera única independientemente de sus localizaciones y/o de los dispositivos que estén utilizando.

Maneja una fragmentación dinámica (*i.e.* división en granos de cooperación) de los documentos de acuerdo a la estructura lógica de los mismos, permitiendo una mayor concurrencia sin imponer restricciones a la estructura de los documentos.

Para ofrecer servicios tolerantes a fallas en la red, PIÑAS permite replicar los documentos en cada servidor HTTP que se necesite. Esta plataforma define el control de la concurrencia por medio de candados pesimistas, en donde sólo una copia de cada fragmento puede ser modificada a la vez. Ésta se llama copia *maestra* y en todo momento es única, pero puede haber muchas copias con acceso de sólo lectura.

Los documentos son fragmentados y las copias maestras de los fragmentos de cada documento pueden estar distribuidos en diferentes sitios, permitiendo una coordinación distribuida.

A través de la definición del usuario que expresa sus sitios de trabajo y de almacenamiento. *PIÑAS* permite a los colaboradores elegir si quieren utilizar una arquitectura distribuida, completamente replicada o híbrida. Cuando un usuario elige una arquitectura completamente replicada se copian los datos tanto para almacenamiento como para tratamiento.

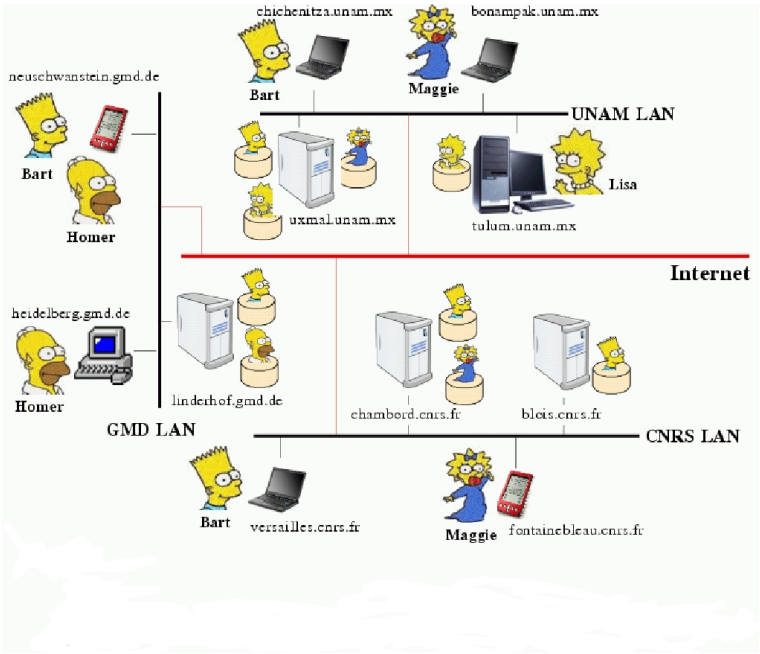

Figura 2.13: Arquitectura de PIÑAS

Esta arquitectura es ideal para los ambientes poco fiables del Internet y proporcionar soporte a usuarios móviles/nómadas. Pero al utilizar candados pesimistas, puede ocasionar que un documento quede bloqueado indefinidamente. Cuando se elige una arquitectura híbrida, los datos compartidos son accesibles desde sitios remotos, sin embargo, se pueden mantener copias locales para acelerar el tratamiento de los datos, mejorando el desempeño de las aplicaciones. La arquitectura híbrida se adapta bien para los usuarios conectados localmente mediante una LAN. Para cada ambiente LAN se utiliza un servidor HTTP que permite almacenar la información como se puede ver en la figura 2.13. Al tener los datos compartidos en diferentes sitios simultáneamente se incrementa la disponibilidad, la confiabilidad y el desempeño de las aplicaciones.

La coordinación es distribuida entre los diferentes servidores HTTP y los usuarios se pueden conectar a (y desconectarse de) los diferentes servidores de acuerdo a su localización geográfica permitiendo así producir en un ambiente de trabajo nómada.

Resumiendo, PIÑAS cuenta con una arquitectura robusta capaz de soportar el trabajo nómada y el trabajo local de manera eficiente, pero adolece de flexibilidad para manejar diferentes políticas de control de concurrencia que permitan adaptarse mejor a los diferentes ambientes de cooperación y a los diferentes tipos de aplicaciones colaborativas.

2.2.5 Aplicaciones cooperativas para la educación a distancia

Existen aplicaciones cooperativas cuyo objetivo es facilitar y soportar la educación a distancia. En una sesión de clase, un expositor o maestro presenta un tema con ayuda

de diapositivas o de un pizarrón virtual desde una sala de videoconferencia o un salón de clases. Los usuarios se conectan de manera remota para participar en la sesión de clase, de manera pasiva (escuchando) o activa (aportando comentarios).

Estas aplicaciones generalmente cuentan con un sistema de conferencia de voz o de videoconferencia. Proveen un espacio de trabajo virtual, que proporciona pizarrones, mapas, diagramas. Además, proveen funciones para compartir documentos, funciones para realizar votaciones, levantar la mano y hablar a los diferentes participantes, entre otras. Cada función maneja diferentes tipos de datos, por lo que deben ser analizados y tratados de manera independiente.

Diapositivas para soporte de la exposición

Durante una sesión de clase, generalmente, las diapositivas para soporte del expositor no sufren cambios, por lo que se puede enviar una copia y sólo notificar los eventos que el expositor realiza sobre las diapositivas (*e.g.* cambio de lámina, señalamiento de algún elemento o cambios menores al contenido).

Una arquitectura con un servidor centralizado, donde se administre el documento de las diapositivas, en un sitio cercano a la ubicación del expositor con réplicas de sólo lectura en los sitios de los participantes puede funcionar adecuadamente.

Voz y/o video usados durante una sesión de clase

La voz y el video son datos sensibles a la latencia, por lo que las acciones deben ir encaminadas a reducirla. Como se vió con los juegos concurrentes estas acciones son:

- El envío de los mensajes de voz y/o video debe realizarse antes que el almacenamiento de los datos compartidos en memoria secundaria.
- El protocolo de comunicación UDP se debe preferir debido a que el control de flujo de TCP puede introducir latencia en la actualización de la integración de los nuevos datos recibidos.
- Este tipo de aplicaciones pueden aceptar pequeñas inconsistencias (*i.e.* pérdidas de paquetes de información) y corregirse con las continuas actualizaciones, por lo que la pérdida de un número mínimo de mensajes es razonable.
- La capa de distribución debe definir y respetar el orden de las notificaciones (serialización) si se utiliza UDP.
- Si un mensaje se retrasa, puede ser ignorado para no retrasar las actualizaciones de este tipo de datos.
- El desempeño de estas aplicaciones se puede mejorar si se utilizan redes basadas en protocolos con diferenciación de tráfico (*e.g. MPLS* o *DiffServ*), etiquetándolos con una alta prioridad.

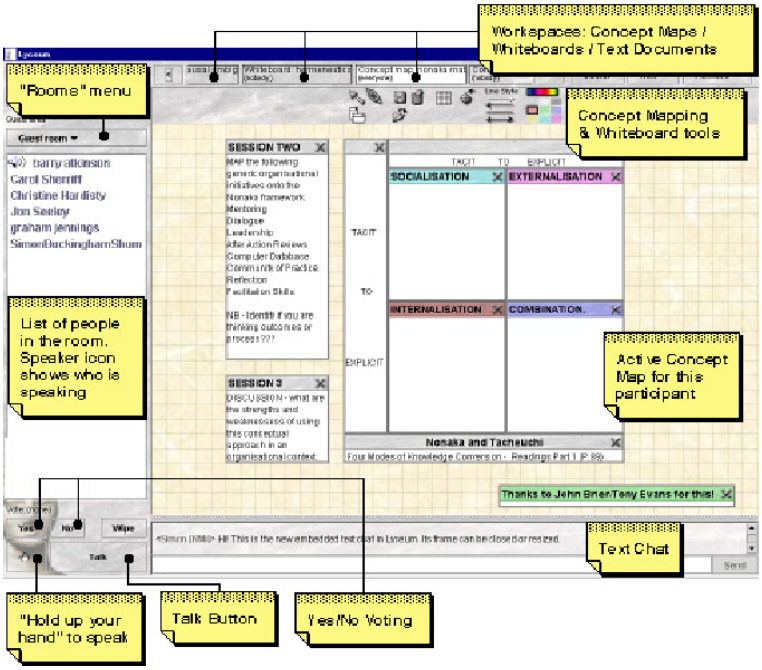

Figura 2.14: Educación a distancia: la aplicación Lyceum

Los pizarrones, la herramientas para votaciones y demás servicios que maneje la aplicación deben ser analizados de manera independiente. Cada solución de difusión y/o actualización se debe adaptar a las necesidades de cada tipo de dato. En este punto, ya se identifica la necesidad de proveer y soportar políticas de distribución/actualización que se adapten a los diferentes tipos de datos compartidos, que maneje una misma aplicación colaborativa distribuida.

Debido a que la mayor parte de los datos se generan del lado del expositor, sería deseable que la coordinación y la difusión de las notificaciones se realicen en el sitio donde se expone la clase o en un servidor cercano (*i.e.* en la misma LAN). Esto es factible porque normalmente el expositor cuenta con las herramientas y el equipo necesarios para impartir su clase al encontrarse en universidades o en instituciones dedicadas a la educación.

Normalmente el expositor lleva el control de la clase y las aplicaciones cuentan con una opción de "pedir turno" que les permite participar a los alumnos. El expositor debe otorgar el turno de manera explícita y pasar el control al sitio donde se encuentra el participante. En ese momento el flujo de la información es generado en el sitio que participa. Este sitio debe enviar las notificaciones al servidor para que realice la difusión de los datos. En muchos casos la calidad del enlace afectará el desempeño de la aplicación, pero generalmente las intervenciones de los alumnos son cortas y pocas.

Lyceum [5] constituye un ejemplo representativo de este tipo de aplicación. La figura 2.14 muestra su interfaz. Soporta conferencias de voz, se pueden compartir archivos, cuenta con un pizarrón virtual, votaciones y herramientas que permiten pedir el turno

a los alumnos para participar en la clase.

2.2.6 Síntesis de tipos de aplicaciones cooperativas

Como vemos, existe una gran cantidad de categorías de aplicaciones colaborativas. Cada una de ellas tiene características propias, por lo que el manejo de los datos compartidos y la sincronización de las réplicas entre los usuarios debe hacerse de acuerdo al tipo de aplicación.

Así, por ejemplo, una aplicación en tiempo real a distancia es más sensible a la sincronización regular de los datos. Un juego como *Halo* por ejemplo, requiere que una gran cantidad de datos sean constantemente actualizados. El tiempo entre el evento y la notificación a todos los usuarios debe ser mínima para que puedan conocer la ubicación exacta de cada jugador. Las actualizaciones con una frecuencia de un segundo hacen a la aplicación inútil, aunque pequeñas inconsistencias pueden ser imperceptibles por los usuarios ya que se corrigen con el constante movimiento de los jugadores o son compensadas por el cerebro humano. Cuando dibujamos una línea por medio de puntos, si un punto falta, la mente automáticamente rellena el espacio faltante. De igual manera cuando se juega *Halo*, si falta un *frame*, el usuario no lo percibe. En cambio, un editor de texto puede admitir razonablemente pequeños retrazos e inclusive puede aceptar desconexiones. Los usuarios pueden seguir trabajando, pero las actualizaciones deben ser consistentes: no se pueden alterar o perder los datos.

La siguiente tabla presenta un resumen de las características de algunas aplicaciones cooperativas. La latencia se refiere a los retrasos en las notificaciones de las acciones de los demás colaboradores. Una aplicación soporta desconexiones cuando los usuarios pueden a) seguir produciendo aunque se encuentren desconectados del grupo de trabajo, b) guardar su trabajo y/o c) enviarlo a los demás usuarios cuando estén conectados.

Existen otros factores, aparte de las características del tipo de aplicación cooperativa, que pueden influir en el desempeño de una aplicación distribuida como: el desempeño de la red, la distribución de los usuarios y las capacidades del dispositivo. En cuanto al dispositivo de interacción/trabajo del colaborador, la memoria, la capacidad de almacenamiento, la disponibilidad del software específico, la capacidad de manejar video o voz, la capacidad de manejar ciertos protocolos de comunicación, etc constituyen otras características a examinar. Sin embargo, la siguiente tabla presenta en términos generales, las arquitecturas y el control de concurrencia recomendado.

Almacenamiento centralizado de los datos y tratamiento distribuido.

Utiliza un servidor centralizado como punto de reunión e implementación de la aplicación, pero la comunicación y la coordinación se realiza en los dispositivos de los usuarios.

La arquitectura y la política de control de concurrencia presentan variantes que influyen directamente en el desempeño de la aplicación que pueden tanto facilitar como prohibir la coolaboración entre los participantes, por lo que es necesario hacer una anáisis más profundo de todos los parámetros por cada tipo de datos compatidos. Al-

Tipo de aplicación	**Latencia**	**Descone-xiones**	**Fragmen-tación**	**Concurrencia**	**Solución de conflictos**	**Pérdidas de datos**
Editores texto asíncronos	Si	Si	Si	No	Manual	No
Juegos por turnos	Pequeños retrasos	No	Si (jugada)	No	No existen	No
Juegos concu-rrentes (eventos trascendentes)	No	No	No	Si	Hacer/ deshacer	No
Juegos concu-rrentes (otras acciones)	No	No	No	Si	Confusión en los usuarios	Si
Mensajería instantánea	Pequeños retrasos	No	Si (mensaje)	Si	Operaciones conmutativas	No
Educación a distancia (diapositivas)	Pequeños retrasos	No	No requiere	No	No existen	No
Educación a distancia (multimedia)	No	No	No	No	Se ignoran	Si
Calendarios	Si	Si	Si	No	Manual/ reglas	No

Table 2.1: Algunas aplicaciones colaborativas y sus características

Tipo de aplicación	**Protocolo de comunicación**	**Arquitectura**	**Coordinación**	**Control de concurrencia**
Editores de texto	TCP	Centralizada*	Centralizada	Candados
Juegos por turnos	TCP	Centralizada*	Centralizada	Tokens
Juegos concurrentes	UDP	Híbrido	P2P**	Serialización
Mensajería instantánea	TCP	Híbrido	P2P**	Sin control de concurrencia
Educación a distancia (diapositivas)	TCP	Centralizada*	Centralizada	Solicitud de token
Eduacación a distancia (multimedia)	UDP	Híbridos	P2P**	Serialización
Agendas	TCP	Centralizada*	Centralizada	Optimista

Table 2.2: Soluciones propuestas para la replicación de los datos

gunos de estos parámetros fueron examinados en la subsección 2.1.2 y en la subsección 2.1.3.

2.2.7 Soluciones utilizadas actualmente

Como se vio en las secciones anteriores, cada aplicación colaborativa, tiene sus propias características y sus propias necesidades.

Pero diseñar e implementar la comunicación y las políticas de notificación, distribución y control de concurrencias es una tarea central, por lo que la aceptabilidad de la aplicación depende directamente de sus características. Para diseñar y proveer una implementación adecuada se deben tener conocimientos en arquitecturas de distribución, el manejo adecuado del control de concurrencia, conceptos de colaboración, etc. Además, al ser un área en desarrollo, muchas veces ni siquiera los expertos se ponen de acuerdo en cual es la mejor solución para la sincronización de las réplicas y es necesario probar varios escenarios, para ver cual es el que mejor se adapta a los requerimientos de las aplicaciones y de los usuarios de acuerdo al medio ambiente en el que colaboran.

Por estas razones, muchas veces los desarrolladores no hacen un análisis previo para determinar la solución que se adapte mejor a las necesidades de la colaboración planificada. Muchos desarrolladores simplemente implementan la que conocen y al no ser la solución adecuada se vuelve compleja de implementar. Como consecuencias, las aplicaciones no tienen el desempeño adecuado y surgen muchas inconsistencias en su funcionamiento. La cordinación de los datos compartidos queda entrelazada con el código de la aplicación. Así, la evolución se torna imposible haciéndo la aplicación difícil de mantener. Además la posiblidad de reutilización del código no es viable.

Para facilitar el trabajo de los desarrolladores de aplicaciones colabarativas, existen muchas plataformas orientadas a simplificar su desarrollo. Pero la mayoría sólo implementan una única solución de coordinación entre los sitios (arquitectura de distribución y control de concurrencia), por lo que sólo se adaptan a cierto tipo de aplicaciones y en la mayoría de los casos no establecen claramente el tipo de aplicaciones para las que son diseñadas.

COAST [24], *WebDAV* [14][15][16], *DTWiki* [13], entre otras constituyen ejemplos típicos de estas arquitecturas.

Desde este punto de vista *COAST* es un *toolkit* que utiliza una política de control de concurrencia optimista basado en la serialización de las modificaciones y resuelve los conflictos utilizando operaciones que permiten hacer y deshacer las modificaciones. Eventualmente estas operaciones pueden ocasionar conflictos y destruir la consistencia de los documentos. Éste implementa una arquitectura completamente replicada.

WebDAV como se vió en la subsección 2.2.4, es una extensión del protocolo HTTP V1.1 que realiza la coordinación y el almacenamiento de los datos compartidos de manera centralizada. El tratamiento de los datos es procesado en los sitios remotos, por lo que no es un protocolo de cooperación tolerante a fallas y no soporta desconexiones. El control de concurrencia está basado en candados pesimistas.

Algunas aplicaciones hacen flexibles la arquitectura de distribución, pero utilizan

sólo una política de control de concurrencia. Ejemplos de estas plataformas son *PIÑAS* [9] y *DreamTeam* [22]. Ambos soportan arquitecturas flexibles, por lo que soportan desconexiones, fallas en la red y retardos, pero proponen solamente candados pesimistas, por lo que sólo se puede aplicar satisfactoriamente en cierto tipo de aplicaciones o para la administración de algunas entidades que se adaptan a estas políticas.

Hace algunos años existió un prototipo llamado *GEN* [21] que hacia flexible el control de concurrencia y basaba su funcionamiento en una arquitectura completamente replicada, pero no se continuó dándole mantenimiento.

2.3 Una arquitectura de trabajo cooperativo flexible y extensible

Para ofrecer un buen servicio y confort a los desarrolladores de aplicaciones y más aún para la satisfacción de los usuarios de aplicaciones cooperativas distribuidas, es esencial que estas aplicaciones deban:

- **Ser tolerantes a fallas**.- si algún elemento de la red falla, otros colaboradores deben poder seguir trabajando y evidentemente se debe evitar que se pierda el trabajo realizado.
- **Tener un buen desempeño**.- es decir un tiempo de respuesta razonable, para que la colaboración no pierda continuidad.
- **Ser persistente a las fallas**.- si los sitios cooperantes fallan, deben ser capaces de lidear con las inconsistencias temporales de las réplicas de la información compartida.
- **Soportar el trabajo nómada**.- más que garantizar la corrección de la información compartida en caso de falla, deben lidear con las desconexiones, permitiendo que el usuario pueda seguir trabajando aún cuando no esté conectado de manera continua.
- **Soportar actualizaciones dinámicas**.- que permitan agregar nuevas funcionalidades sin afectaciones a las aplicaciones, incrementando así, la disponibilidad del servicio.

Una plataforma que maneje diferentes arquitecturas

Para que las aplicaciones coolaborativas satisfagan los cuatro primeros requerimientos, se requiere que todo el control de la replicación y notificación de los datos compartidos sea acorde a los requerimientos de las aplicaciones. Como vimos en la sección 2.2, a grandes rasgos dependen principalmente del tipo de aplicación, del tipo de dispositivo y de la calidad de la red. Pero, actualmente no se cuenta con una plataforma que permita manejar diferentes tipos de arquitecturas, ni diversas políticas de control

de concurrencia.

Una plataforma siempre disponible
Además debemos considerar que las aplicaciones colaborativas Web requieren una alta disponibilidad debido al gran número de usuarios que la utilizan en diferentes partes del mundo y consecuentemente en distintos usos horarios, por lo que deben estar disponibles las 24 horas del día, los 7 días de la semana.

Una plataforma actualizable dinámicamente
Pero las aplicaciones cooperativas requieren actualizaciones frecuentes para a) la correción de errores, b) la integración de nuevos requerimientos y c) el ajuste a cambios en el entorno de la aplicación. Para poder actualizar una aplicación sin afectar el servicio, requiere soportar actualizaciones dinámicas.

Una plataforma flexible y extensible
Con la intención de soportar múltiples tipos de aplicaciones en diferentes ambientes, proponemos diseñar e implementar una plataforma capaz de proveer soporte a las aplicaciones para la distribución de los datos compartidos que sea capaz de ofrecer diversas políticas de control de concurrencia y de manejar diferentes arquitecturas de distribución.

La plataforma se basa en una arquitectura formada por una plataforma mínima capaz de integrar diferentes entidades compartidas. Una entidad compartida administra un tipo de datos (*e.g.* los datos replicados propios de la aplicación). En consecuencia ésta utiliza el mismo control de concurrencia y la misma arquitectura de distribución para un mismo tipo de datos compartidos. Si una aplicación requiere más de un tipo (*e.g.* las aplicaciones para la educación a distancia ver 2.2.5); es posible integrar a la plataforma más de un tipo de entidad. De hecho la mayoría de las aplicaciones requieren al menos dos tipos de entidades: las de control de acceso y los datos propios de la aplicación.

La plataforma también debe contar con las herramientas que permiten definir las entidades, integrarlas con las aplicaciones y actualizar/adaptar sus definiciones en tiempo de ejecución.

2.3.1 Arquitectura general de la plataforma propuesta

Siguiendo los requerimientos identificados se definió una plataforma extensible con entidades replicadas distribuidas flexibles y adaptables en tiempo de ejecución. Además esta plataforma debe soportar el trabajo móvil/nómada. Esta plataforma debe también permitir al desarrollador de aplicaciones cooperativas agregar o derivar nuevos tipos de entidades (a partir de las ya existentes) y suprimir, modificar o especializar los tipos y entidades en uso. También debe soportar la definición, modificación o supresión de nuevas políticas. La figura 2.15 presenta la estructura de la plataforma.

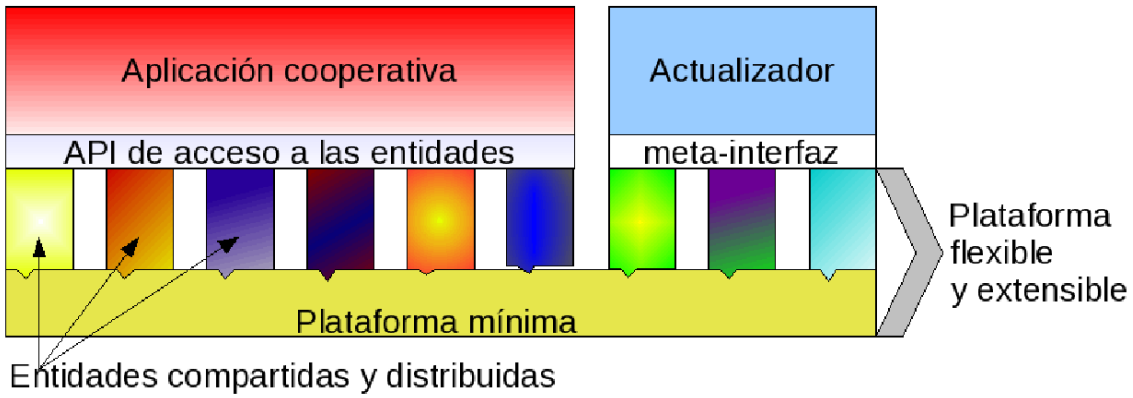

Figura 2.15: Arquitectura de la plataforma

Plataforma mínima

La plataforma mínima constituye la base de la plataforma a la cual se integran las entidades compartidas y los módulos funcionales que permiten integrar nuevas políticas de replicación, control de acceso y control de concurrencia.

La plataforma administra cuatro tipos principales de usuarios:

- **Nivel programación de la plataforma**
  - **Super-programadores**: Esta categoría integra usuarios que pueden definir nuevos tipos de entidades, agregar nuevas políticas y en términos generales, pueden modificar cualquier clase de la plataforma.
  - **Desarrolladores**: Esta clase de usuarios desarrollan las aplicaciones, al crear nuevas instancias de los tipos de entidades y políticas existentes.
- **Nivel utilización de la plataforma**
  - **Administradores**: Se encargan de administrar las aplicaciones, crear nuevas instancias de las entidades de control de acceso y definir los usuarios que tienen acceso a las aplicaciones.
  - **Usuarios**: Son los usuarios finales de las aplicaciones. Utilizan las entidades propias de las aplicaciones (*e.g.* archivos de texto, agendas, colas de mensajes, etc).

Entidades compartidas

Las entidades compartidas y distribuidas están formadas por estructuras de datos que contienen los datos manejados por la aplicación y las funciones que permiten, administrarlos, sincronizar las diferentes réplicas de la información, notificar y recibir las notificaciones de los cambios hechos a los datos de/para la aplicación colaborativa

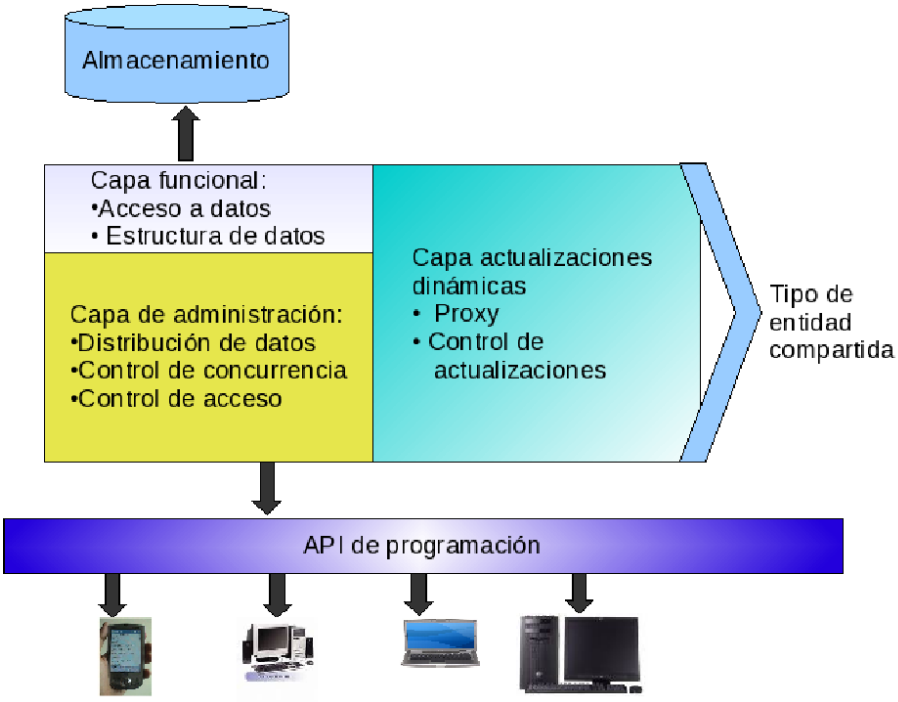

Figura 2.16: Arquitectura de la entidad

distribuida. Cada tipo de entidad debe satisfacer los requerimientos de desempeño y usabilidad de los distintos entornos de trabajo cooperativo distribuido.

Como se puede ver en la figura 2.16, la entidad compartida, replicada y distribuida está formada por las siguientes capas:

- **Capa funcional**: Incluye el estado de la entidad *i.e* estructura de datos con el valor propio de la entidad (*e.g.* en un pizarrón compartido: las figuras incluidas, su posición y su formato) y la interfaz de programación que incluye las funciones que permiten manipular su estado (*i.e.* modificar, agregar, borrar). Para permitir la fragmentación de los datos de la aplicación, cuenta con los mecanismos que permiten administrar los fragmentos.

  La capa funcional también incluye la funciones que permiten administrar y guardar la estructura de datos de manera persistente para los espacios de almacenamiento.

- **Capa de Administración**: La capa de administración está formada por las políticas de distribución, control de concurrencia, difusión y actualización que permiten integrar nuevas versiones y así asegurar la sincronización de la entidad replicada.

  También cuenta con los atributos de control de acceso que permiten validar las acciones de los usuarios en la entidad.

- **Capa de actualizaciones dinámicas**: La capa de actualizaciones dinámicas administra el tipo de entidad y la versión a la que pertenece la instancia. Cuenta con las funciones o métodos que permiten modificar los parámetros de administración de la entidad transformándola de un tipo T a T'.

API (Application Programming Interface)

Esta API (ver la figura 2.15), permite a los desarrolladores de aplicaciones colaborativas integrar las aplicaciones en la plataforma y de esta manera permitirlas acceder a las entidades compartidas distribuidas.

Cualquier desarrollador de aplicaciones con una capacitación mínima puede utilizar el API y desarrollar nuevas aplicaciones colaborativas.

Meta-Interfaz

Permite a los super-desarrolladores de la plataforma definir y agregar nuevos tipos de entidades, opcionalmente a partir de modelos ya existentes. También permite agregar o modificar políticas de replicación o control de acceso, así como cualquier clase de la plataforma. La meta-interfaz tiene como objetivo dar flexibilidad y extensibilidad a la plataforma.

Actualizador

El actualizador nos permite controlar el cambio de versión de una entidad distribuida replicada en tiempo de ejecución. Este componente se comunica con la meta-interfaz para realizar los cambios necesarios a la plataforma y/o sus elementos (*e.g.* entidades, políticas de replicación, políticas de control de acceso, *etc.*) Para que el actualizador funcione, la entidad debe contar con los elementos necesarios, *i.e.* los mecanismos que le permitan conocer su estado, controlar el acceso a las llamadas de las clases y las instancias de las clases que se encuentran en memoria.

Capítulo 3

Objetos flexibles y actualizaciones dinámicas de software

Un sistema colaborativo distribuido requiere ser actualizado constantemente para a) poder corregir errores, b) implementar nuevas funcionalidades) y c) realizar cambios en el entorno de trabajo. Para realizar las actualizaciones de un sistema colaborativo sin decrementar su disponibilidad, es necesario actualizarlo en tiempo de ejecución, *i.e.* dinámicamente.
Las actualizaciones dinámicas nos permiten que las entidades replicadas compartidas, se adapten de acuerdo al entorno, mejorando así, el desempeño de las aplicaciones en ambientes poco fiables como a) la Internet y b) el trabajo móvil o nómada.

Las aplicaciones colaborativas distribuidas utilizan diferentes tipos de entidades compartidas (*e.g.* la cola de mensajes de una chat, los documentos de un editor, las listas de control de acceso, *etc.*). Si se desea que la aplicación tenga un buen desempeño y proporcione mayor confort de programación/interacción a los programadores y/o colaboradores, (ver capítulo II), estas entidades deben ser asociados a diversas políticas de control de concurrencia, notificación, distribución, *etc*.
Con el objetivo de definir una plataforma de trabajo cooperativo distribuido flexible y extensible, se requieren técnicas que permitan intercambiar cada política asociada a la entidad en tiempo de ejecución como si fueran piezas de un lego.

Dentro de la literatura, encontramos algunas técnicas que permiten proveer características de flexibilidad a las entidades compartidas: el diseño de patrones y la programación orientada a aspectos, las cuales se estudian en la primera sección. Básicamente explicaremos algunos patrones de diseño utilizados para el modelo desarrollado en el presente trabajo, sus ventajas y sus desventajas con respecto a la programación orientada a aspectos.

Los objetos flexibles ofrecen a un sistema de software las siguientes ventajas:

- Dan versatilidad al sistema al permitir utilizar diferentes estrategias para una misma forma de realizar algo.

- Eliminan el código redundante.
- Reducen el tiempo de programación.
- Reducen errores de programación.
- Facilitan la conceptualización del sistema.

Actualmente, no existen aplicaciones colaborativas capaces de soportar actualizaciones de software en tiempo de ejecución, pero existen varias aplicaciones distribuidas con actualizaciones dinámicas de software, por lo que en la sección II presentaremos algunos enfoques que permiten realizarlas.

3.1 Técnicas que permiten hacer los objetos flexibles

3.1.1 La programación orientada a aspectos

Cuando se desarrollan las aplicaciones utilizando los lenguajes de programación tradicionales (*i.e.* la programación orientada a objetos, la programación estructural y la programación funcional), en ocasiones, algunas funcionalidades se encuentran distribuidas y así mezcladas en toda la aplicación. Ésto da como resultado la dispersión y el entrelazamiento de código, debido a que no es posible separar completamente las diferentes funcionalidades, por lo que es necesario combinarlas en un mismo objeto.

Por ejemplo, si se quiere implementar la seguridad en un sistema, es necesario implementarla en cada módulo. Si las funcionalidades de las aplicaciones se encuentran dispersas y entrelazadas, su cambio y/o su mantenimiento se vuelve más complejo y difícil, haciéndolas más propensas a errores de programación.

Normalmente un lenguaje de programación tradicional, como se puede ver en la figure 3.1, utiliza un compilador que tiene como entrada el código de la aplicación y como salida la aplicación generada en código binario.

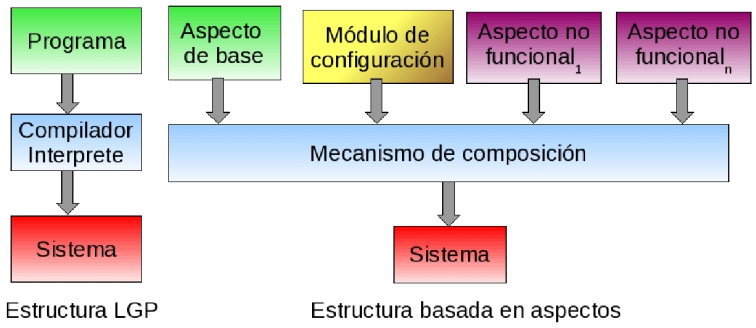

Figura 3.1: Arquitectura tradicional *vs.* POA figura obtenida de [37]

Kiczales [31] propone descomponer la aplicación en sus diferentes funcionalidades para poder separarlas, las cuales denomina aspectos. Como se ve en la figura 3.1, es

necesario introducir los siguientes elementos:

Aspecto de base

Implementa la funcionalidad básica de la aplicación principal, para lo cual fue diseñado el sistema.

Aspectos

Un aspecto encapsula la funcionalidad que deseamos entrelazar con la funcionalidad básica, agregando a la aplicación propiedades adicionales. Normalmente, los aspectos son características no funcionales como la sincronía, la seguridad, la distribución, *etc.* Cuando se utiliza la programación orientada a aspectos, a diferencia de otras técnicas, los aspectos se programan de manera independiente. Posteriormente, en un proceso de precompilación, son insertados en el aspecto base, para producir la aplicación final.

Módulo de configuración

En el módulo de configuración se establecen los puntos de unión en donde se deben insertar los aspectos al aspecto base.

Entrelazador

El entrelazador se encarga de realizar la composición de los aspectos y el aspecto base de acuerdo al módulo de configuración. Una vez compuesto se genera el sistema final.

Mediante estos elementos es posible implementar cada una de las características de la aplicación (funcionales y no funcionales) de manera independiente. Su desarrollo resulta ser más fácil y se reduce el código de la aplicación, al no tener que repetirlo cada vez que se requiere agregar la funcionalidad a algún bloque del programa.

3.1.2 Patrones de diseño

Gamma *et. al* [18] proponen el uso de los patrones de diseño, para simplificar la solución de problemas comunes de la programación orientada a objetos. Un patrón de diseño se define como una solución pre-empaquetada para resolver un problema de diseño común. Su finalidad es poder reutilizar código y así facilitar su programación. Alexander *et. al* [19] lo definen como: "*Cada patrón describe un problema el cual ocurre una y otra vez en nuestro ambiente, describe el núcleo de la solución para ese problema y la manera en que puede usarse esta solución un millón de veces, sin implementar lo mismo dos veces*". El poder utilizar la misma solución sin volverla a programar, es especialmente esencial para el reuso y el mantenimiento de la aplicación.

Además, Gamma *et. al* [18] acentúan: "*Cada diseño de patrón se enfoca en un problema o asunto de diseño en particular de la programación orientada-objetos. Éste describe cuando se aplica, donde se aplica de acuerdo a las restricciones de diseño y las consecuencias o pros y contras de su uso*." A partir de esta proposición se definieron 23 diseños de patrones que se utilizan frecuentemente en la programación orientada a objetos. Se realizó un

análisis para identificar aquellos patrones de diseño que nos permitirán definir una plataforma flexible y extensible: *observer, template, strategy, facade, adapter* y *proxy*. Estos patrones se explican con un poco más de detalle, aunque basandonos en la versión para Ruby que presenta Olsen [20]. Para una mayor comprensión el lector se puede referir a Olsen [20] o Gamma *et. al* [18].

Patrón de diseño *template*

Este patrón funciona como un esqueleto para un algoritmo. Se basa en la herencia y algunas partes del algoritmo son delegados a subclases.

Para implementar el patrón de diseño *template* (ver la figura 3.2), se crea una clase abstracta. Las operaciones son implementadas en las clases concretas las cuales heredan sus atributos y sus métodos de la clase padre. En la clase abstracta se pueden definir los métodos por omisión. Estos métodos se conocen como ganchos o *hooks*. Si la clase concreta requiere una implementación del método diferente al de omisión, éste es redefinido al implementar la clase concreta. De esta manera la clase abstracta funciona como una plantilla que permite definir los métodos de manera general, para luego ser especializados en la implementación de las subclases.

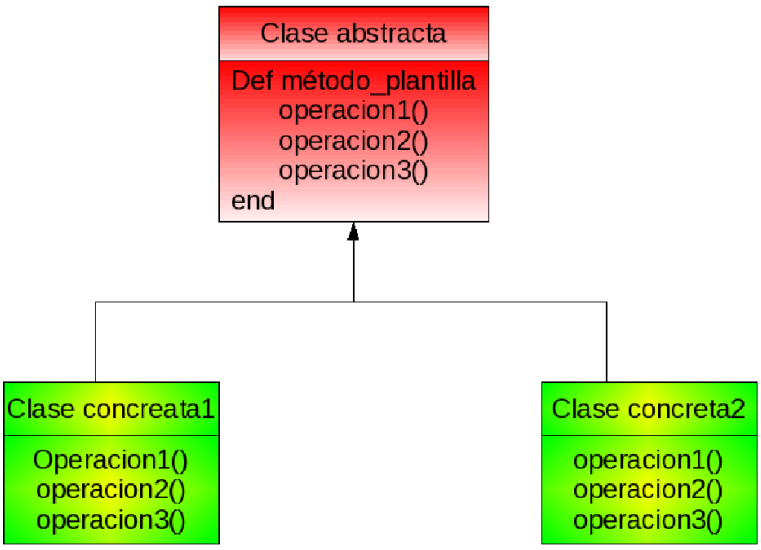

Figura 3.2: Patrón de diseño template

Patrón de diseño *strategy*

Este patrón se utiliza cuando se tiene un trabajo bien definido, pero existen varias formas de hacerlo: Se recomienda utilizarlo cuando se tienen varios algoritmos para un mismo trabajo y se escoje uno de los algoritmos de acuerdo al contexto. Por ejemplo, si se requiere realizar un reporteador y existen varios formatos (*e.g.* texto plano, html,

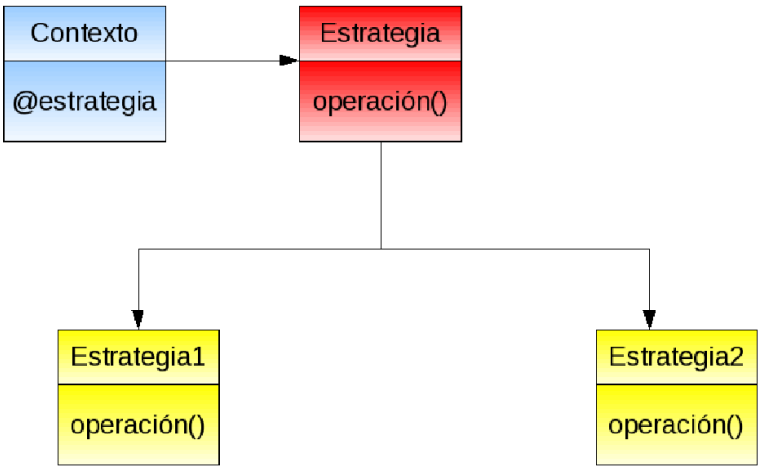

Figura 3.3: Patrón de diseño estrategia

etc.), se recomienda encerrar la parte constante en una clase base y utilizar una estrategia para tratar cada formato en particular. Este patrón a diferencia de los *templates*, se basa en la delegación.

En el patrón de diseño de estrategias, como vemos en la figura 3.3, el contexto establece la estrategia a implementar. Por medio de un manejador de estrategias, se establecen las operaciones que son comunes a todas. Las operaciones que no son comunes se manejan en clases separadas. El manejador de estrategias delega las operaciones específicas a cada estrategia de acuerdo al contexto. Es importante resaltar que a diferencia del patrón de diseño *template*, éste se basa en la delegación.

Patrón de diseño *facade*

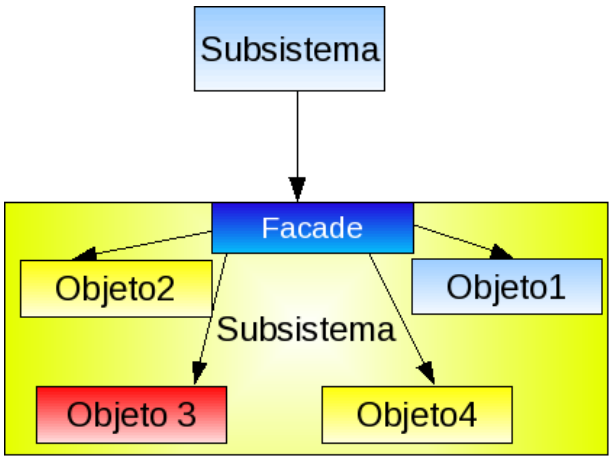

Figura 3.4: Patrón de diseño *facade*

Cuando un subsistema está formado por muchos objetos y se requiere que tenga interacción con otro subsistema, las llamadas a los métodos de las clases se vuelven tediosas y complejas, haciendo necesario conocer la composición a detalle de éstos para poder relacionarlos. Para evitar crear interfaces complejas, se puede colocar una fachada (*facede*) que direccione las llamadas a los objetos dentro del subsistema como se ve en la figura 3.4. De esta manera los subsistemas externos sólo se comunican con la fachada.

Las fachadas se deben utilizar por los siguientes motivos:

- simplifican las interfaces,
- incrementan la seguridad ya que las aplicaciones no tienen acceso directo a los módulos del sistema ni a todos los métodos de los objetos.
- encapsulan mejor los módulos que constituyen un sistema.

Patrón de diseño *observer*

Si se tiene una o más clases (observadores) que deben conocer el estado de otra (observado), para no mezclar las clases, los observadores se anotan en el control. Cuando el observado cambia, notifica al control (ver la figura 3.5) y el control notifica a cada uno de los observadores. Los observadores deben tener un método que permita manejar las notificaciones.

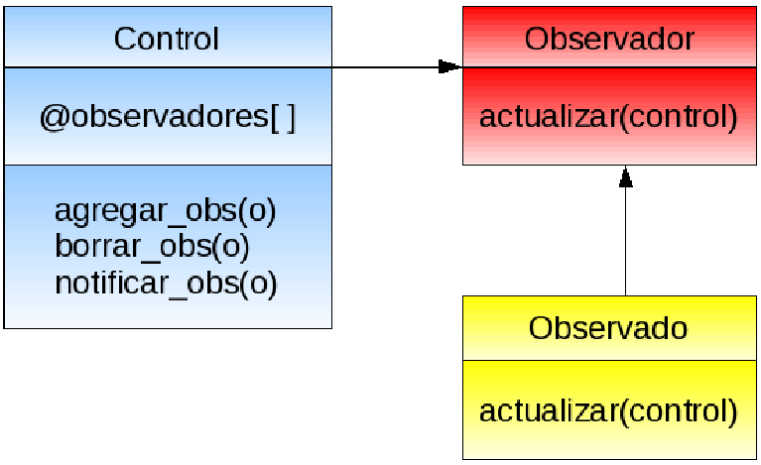

Figura 3.5: Diseño de patrón *observer*

El patrón de diseño *observer* permite separar las clases mejor y agregar nuevos observadores fácilmente a un sistema, haciendo que su crecimiento sea más sencillo.

Patrón de diseño *adapter*

El patrón de diseño *adapter* permite acoplar clases que requieren comunicarse pero sus interfaces no se adaptan.

Como se puede ver en la figura 3.6, el cliente se comunica por medio del adaptador, éste realiza la transformación de los datos y los envía ya transfomados al adaptado, comunicando a los dos objetos. La comunicación regularmente se realiza en los dos sentidos, por lo que en ocasiones el cliente se convierte en adaptado y el adaptado en cliente.

Figura 3.6: Adaptador

Patrón de diseño *proxy*

El *proxy* actua como un representante cuando no se quiere que los clientes tengan acceso directo a los objetos. Como podemos ver en la figura 3.7, el cliente hace la llamada al *proxy*, éste se encarga de realizar la petición del cliente al objeto real y regresar los resultados.

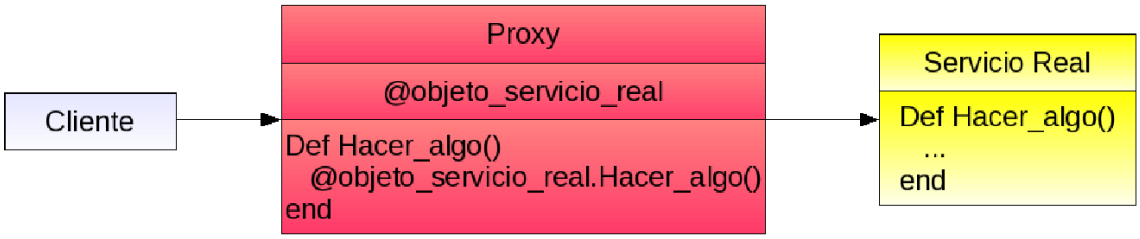

Figura 3.7: Patrón de diseño *proxy*

Los proxies tienen varias aplicaciones:

- **Direccionamiento**: El objeto que realiza la acción puede estar cambiando constantemente, *e.g.* si se tienen varios servidores para realizar balanceo de cargas, el *proxy* puede determinar cual es el servidor que va a atender la siguiente petición de acuerdo a la carga de cada servidor.

- **Control de acceso**: Los *proxies* originalmente fueron propuestos por Gamma et.al. [18] para controlar el acceso a ciertos servicios por cuestiones de seguridad. Mediante un *proxy* se puede verificar que el cliente tenga derecho a esos servicios antes de realizar la operación, incrementando así la seguridad; debido a que los clientes no conocen el objeto real que proporcionan el servicio solicitado.

- **Retraso de la operación**: Cuando se requieren realizar una operación atómica, un *proxy* puede retrasarla hasta que se cuente con todos los elementos necesarios para iniciar la operación.

3.2 Actualizaciones dinámicas de software

Los sistemas distribuidos que prestan sus servicios en la Web requieren de una alta disponibilidad por dos razones: a) debido a los diferentes usos horarios se requiere la disponibilidad de los sistemas las 24 horas del día y b) es difícil, y en algunos casos imposible, detener un sistema distribuido en la Web, ya que éste requiere mútiples entidades distribuidas.

Pero todos los sistemas deben ser actualizados debido a:

- se tienen errores de programación inevitablemente.
- se necesitan actualizaciones para poder agregar nuevas funcionalidades o nuevos requerimientos ya sea por omisión o cambios en el entorno del sistema.
- deben adaptarse dinámicamente a los cambios en el entorno ocasionados por fallas en la red, o debido a usuarios móviles y/o nómadas, en ambientes poco fiables como la Internet .

Para poder actualizar un sistema sin mermar su disponibilidad es necesario que sea flexible y actualizable en tiempo de ejecución.

El concepto de actualizaciones dinámicas ha sido aplicado en diferentes aplicaciones distribuidas; por ejemplo Loom.NET [27], PROSE [28], Proteus [30], *etc*. Hasta hoy no tenemos conocimiento de alguna plataforma colaborativa que emplee actualizaciones dinámicas.

La mayoría de las aplicaciones distribuidas con actualizaciones dinámicas utilizan una arquitectura híbrida con la coordinación y el almacenaje centralizado, *i.e.* sólo replican la información para realizar el tratamiento de los datos. Algunas aplicaciones distribuyen las funciones en diferentes servidores, pero cada funcionalidad es centralizada. Sólo Ajmani *et. al* [25] considera un método para actualizar entidades distribuidas.

En esta sección analizaremos los diferentes criterios para hacer posible actualizar una aplicación en tiempo de ejecución.

3.2.1 El problema de actualizar un sistema en tiempo de ejecución

La aplicación debe contar con los mecanismos necesarios que permitan transformar una entidad T en T', actualizando los métodos y los atributos que lo forman. Después

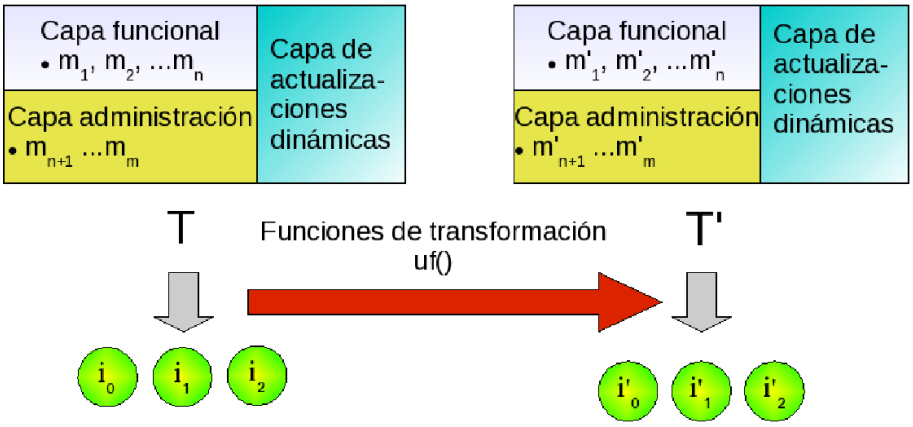

Figura 3.8: Actualización de entidades

de actualizar los tipos es necesario transformar las instancias como se muestra en la figura 3.8.

Para poder llevar a cabo las actualizaciones en la medida de lo posible, de forma transparente debemos tomar en cuenta las siguientes consideraciones:

- Los cambios se deben propagar lentamente a través de todo el sistema distribuido *i.e.* debe ser modular.
- Los cambios se deben realizar poco a poco, uno a la vez, permitiendo controlar mejor la actualización. Aunque se pueden realizar varios cambios simultáneos, no es conveniente cuando éstos afectan las mismas clases o la comunicación entre las entidades, ya que la trasformación de las instancias se vuelve mucho más compleja.
- El sistema de actualización debe soportar cualquier tipo de cambio, aún cuando las versiones no sean consistentes.
- Las modificaciones locales deben ser atómicas, *i.e.* se deben tranformar las clases y sus dependencias al mismo tiempo. Durante el proceso de actualización no debe cambiar el estado de la clase, exceptuando las llamadas a clases en objetos remotos, las cuales, debido a la modularidad de la actualización, deben tener un tratamiento diferente.
- El sistema debe ser capaz de conservar el estado persistente de las entidades y transformarlo de manera consistente con la nueva versión.
- La actualización debe ser controlada por el sistema, conociendo el estado de la actualización de cada una de las entidades distribuidas.
- El sistema debe ser capaz de lidear con al menos las dos últimas versiones transparentemente. De ser posible debe poder comunicarlas para tener un menor número de inconsistencias ocasionadas por la actualización.

- El servicio debe continuar a pesar de que los componentes tengan diferentes versiones y los usuarios no deben ser afectados por los cambios.
- Las actualizaciones dinámica no deben ocasionar dispersión de las clases y la aplicación debe contar con los elementos necesarios para reconstruir la última versión del sistema de manera coherente y consistente, facilitando su conceptualización.

En las siguientes subsecciones analizaremos los elementos que debe tener una actualización para poder cumplir con todos estos requerimientos.

3.2.2 Actualizaciones dinámicas y objetos flexibles

La forma más simple de hacer una actualización dinámica es crear objetos flexibles. Los objetos flexibles dan versatilidad a un sistema, al considerar diferentes *strategies*, *templetes* o *wrappers*, permitiendo mantener un código más comprensible y haciendo la aplicación más simple de conceptualizar. Sin embargo, los objetos flexible no consideran el estado y limitan el tipo de cambios a realizar, ya que sólo permiten elegir entre diversas opciones. Pero no permiten la corrección de errores o agregar nuevo atributos o métodos a las clases.

Existen plataformas para aplicaciones colaborativas que soportan objetos flexibles, algunas basadas en patrones de diseño como *GEN* [21] que es una plataforma que utiliza *wrappers* para generar diferentes políticas de distribución o control de concurrencia. Existen otras basadas en la programación orientada a aspectos como Lopes [32] que creó un lenguaje llamado *D* para manejar diferentes políticas de sincronización y distribución. Pero estas plataformas no permiten agregar nuevos algoritmos en tiempo de ejecución.

Realizar actualizaciones dinámicas sin utilizar objetos flexibles, es complejo y el código se vuelve complicado, quedando mezclado y revuelto, *i.e.* una parte queda en el código original de la implementación de la clase y otra en el código que transforma el objeto a la nueva clase (función de transformación de las instancias de la actualización dinámica). Algunos lenguajes permiten modificaciones dinámicas a las clases, por medio de las técnicas de reflexión. El problema es que si se realiza la modificación directa sobre la clase, no se cuenta con una última versión concentrada en un sitio, sino que la clase queda regada en el código disperso de las diferentes actualizaciones.

Por esta razón, es recomendable utilizar mecanismos que redireccionen la clase a utilizar, basados en objetos flexibles. Milazzo *et al.* [26], Rasche *et. al* [27] y Ajmani *et. al* [25] utilizan *proxies* para seleccionar la versión de la clase a utilizar.

3.2.3 Control de concurrencia de las instancias de las clases

Durante la actualización de una clase y sus instancias, puede ser que algunos métodos se encuentren en uso. Si se realiza una llamada a un método durante el proceso de actualización puede haber inconsistencias, por lo que es necesario controlar la llamadas

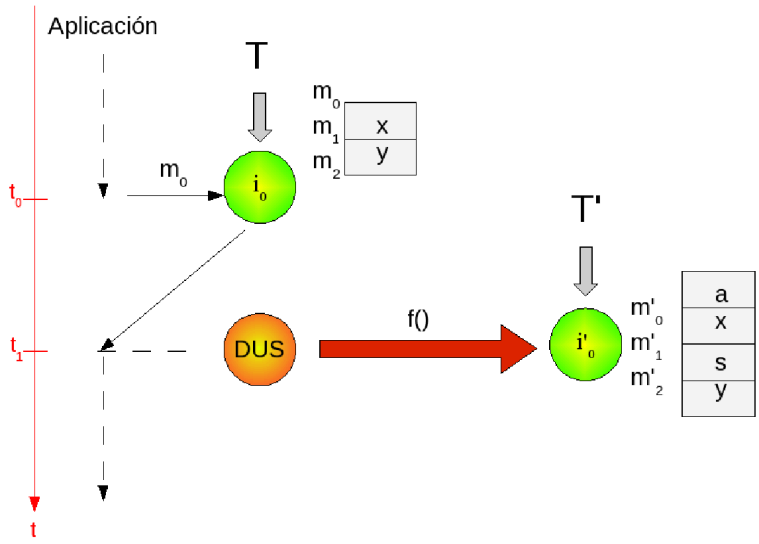

Figura 3.9: Actualización de instancias

a los métodos de los objetos durante el proceso de actualización, como se muestra en la figura figura 3.9.

El control de concurrencia de los métodos de las instancias a actualizar se puede manejar de varias formas. Se puede utilizar semáforos o monitores. Pissias and Coulson [29] utilizan sincronización por medio de semáforos y esperan hasta que no existan llamadas a los métodos para bloquear el objeto. Rashe *et. al* [27] utiliza el algoritmo de lectores y escritores.

El algoritmo de lectores y escritores se usa cuando existen procesos que requieren acceso de lectura, los cuales se pueden desarrollar de manera concurrente. Pero también existen escritores que requieren acceso exclusivo a los datos, por lo que cuando se realiza una modificación nadie más puede leerlos o cambiarlos. La solución que presenta Tanenbaum [35] utiliza un semáforo *db* para acceso a los datos y un contador de lectores *rc*. Cuando el primer lector accesa a los datos ejecuta un *down* en el semáforo *rc*, cuando un lector sale, decrementa el contador y el último lector en salir ejecuta un *up* en el semáforo, permitiendo el acceso a un escritor bloqueado, si lo hay.
El problema con esta solución es que un escritor puede estar bloqueado indefinidamente mientras haya un lector accesando los datos, de tal suerte otros lectores pueden entrar y hacer la espera del escritor interminable. Courtois [36] propone una solución a este problema colocando a los nuevos lectores en la misma cola de espera del escritor, por lo que el escritor sólo tiene que esperar a que terminen los lectores que ya estaban antes de que él llegara.

Rashe *et. al* [27] le da prioridad a los lectores, por lo que la actualización puede retrazarse indefinidamente. Para resolver este problema utiliza un temporizador y si el tiempo de actualización expira aborta la actualización.

Ajmani *et. al* [25] no utiliza semáforos, sino que utiliza *proxies* que simulan ser el

objeto durante la actualización y las llamadas existentes hechas por los diferentes hilos de ejecución son eliminadas al reinicializar (*i.e. reboot*) el objeto, dejando que el protocolo RPC resuelva aquellas llamadas que se perdieron al reiniciarlo, considerando que RPC cuenta con los mecanismos necesarios para detectar que una llamada a un método no fue concluída. Para llamadas remotas es una buena alternativa, pero en llamadas locales crea *overhead* al utilizar RPC.

3.2.4 Transformación del estado

Si se crea una instancia con una clase nueva, el estado de la clase anterior se pierde, por lo que es necesario contar con los mecanismos que nos permitan lidear con esta problemática.

Pissias and Coulson [29] esperan hasta que el objeto esté completamente inactivo y tenga su estado inicial. Este enfoque es un poco ingenuo, ya que muchas clases no regresan a un estado inicial o tardan mucho tiempo en estar completamente inactivas.

En el caso de las clases que manejan el estado de los datos de una aplicación colaborativa, difícilmente alcanzarán un estado inicial, por lo que es necesario conservar el estado persistente de los datos.

Se puede obtener el estado persistente de los objetos instanciados con la versión de la clase anterior, utilizando las técnicas de reflexión.

De acuerdo con Flanagan & Matsumoto [33] "*la reflexión también llamada instrospección, simplemente significa que un programa puede examinar su estado y su estructura*". El poder obtener el estado mediante las técnicas de reflexión varía de acuerdo al lenguaje, por lo que abordaremos las técnicas utilizadas al describir la implementación realizada en el lenguaje Ruby.

Para poder realizar las actualizaciones Cech [28] y Ajmani *et. al* [25] manejan funciones de transformación que se encargan del proceso de obtención, transformación y actualización del estado del objeto.

3.2.5 Actualizaciones dinámicas en sistemas distribuidos

La mayoría de las aplicaciones distribuidas que implementan actualizaciones dinámicas tienen los datos centralizados y aunque el tratamiento de los datos por el usuario esté distribuido, no tienen que lidiar con los problemas de sincronización de las réplicas de las entidades distribuidas. Ejemplos de estos tipos de aplicaciones son las implementaciones de Cech [27], Milazzo [26] y Rashe [27].

Ajmani *et. al* [25] proponen un modelo más general, buscando solucionar los problemas de las entidades replicadas distribuidas. Ellos utilizan *proxies* que simulan ser los objetos pasados y futuros del objeto real, con la intención de poder comunicar las entidades de diferentes versiones cuando éstas son compatibles. Cuando no lo son desactiva la comunicación pero no aborda el problema de la sincronización posterior a la actualización.

Ajmani *et. al* [25] logran manejar varias versiones simultáneamente, aunque en la práctica las funciones de actualización y los *proxies* se vuelven mucho más complejos.

3.2.6 Control de la actualización

Para que una actualización dinámica sea iniciada, requiere que alguien notifique a las entidades que existe una nueva versión y lleve el control del estado de la actualización a nivel global, *i.e.* debe saber que nodos han sido actualizados y cuales no.

Milazzo [26] utiliza un gestor centralizado de la aplicación, el cual se encarga no sólo de distribuir los cambios, sino que también recolecta la información del desempeño de la aplicación para saber en que momento requiere ser actualizada una clase o debe moverse de lugar debido a la carga del servidor donde se encuentra.

Ajmani *et. al* [25] utilizan un actualizador, el cual envía los cambios y controla el proceso de actualización, manejando las políticas de actualización que establecen el momento en que ésta se debe realizar, *i.e.* a una hora determinada o cuando se cumpla una condición. También controla la velocidad en que debe hacerse la actualización (nodo por nodo o todos simultáneamente). Maneja una base de datos centralizada que permite conocer las versiones de las clases cargadas en cada nodo. Esta base es replicada en cada sitio.

3.2.7 Actualizaciones dinámicas como un aspecto

Un sistema se debe encontrar preparado para poder hacer actualizaciones dinámicas. Para ello requiere de un conjunto de elementos que permitan administrar las versiones, recibir las nuevas clases y sus transformaciones, determinar la última versión de la clase y bloquear la clase a actualizar durante el proceso de cambio de versión. Estos elementos se pueden programar de manera separada como un aspecto como lo hace *LOOM.NET* [27], el cual posteriormente es entrelazado por el `rapier`, que es un lenguaje orientado a aspectos de dominio específico para actualizaciones dinámicas y fué desarrollado en *.NET*.

Cech [28] cuenta con un analizador de código que determina los cambios a realizar y el orden en que deben ejecutarse. Mediante el analizador de código genera las funciones de transformación que permiten realizar un cambio y cada una de éstas son programadas como un aspecto y son entrelazadas mediante un *weaver*. La propuesta es interesante pero el analizador de códigio tiene restricciones ya que no puede llevar a cabo cualquier tipo de cambio.

3.2.8 El patrones de diseño vs. la programación orientada a aspectos

Los dos métodos proporcionan flexibilidad, pero cada uno tiene sus restricciones.

- Utilizar patrones de diseño no sólo permite ser flexibles y tener un código más ordenado, sino que permite relacionar las clases de una manera más ordenada.

- La programación orientada a aspectos permite una mejor separación de las funcionalidades, reduciendo considerablemente el código.
- La programación orientada a aspectos combina las clases en tiempo de carga mientras que los patrones de diseño en tiempo de ejecución.
- Actualmente el único lenguajes para programación orientada a aspectos de propósito general es *AspectJ*. Éste sólo es soportado por el lenguaje de programación *Java*. Si se quiere programar aspectos en cualquier otro lenguage, hay que crear un lenguage de proposito específico, haciendo más lenta la implementación. Los patrones de diseño se pueden implementar en cualquier lenguaje ya que son modelos, su implementación es un poco más compleja si la comparamos con *AspectJ*, pero más simple que implementar un lenguaje.
- La implementación de las actualizaciones dinámincas de software son más fáciles de realizar en un lenguaje dinámicamente tipado. Por esta razón se descarta la posibilidad de utilizar un lenguaje estáticamente tipado como Java.
- La programación orientada a aspectos permite encapsular mejor los aspectos haciendo posible una mayor reutilización de código.

Aunque la programación orientada a aspectos permite separar mejor las funcionalidades, debido a su complejidad al utilizar un lenguaje de programación diferente de Java, utilizaremos patrones de diseño.

Aunque no descartamos la posibilidad de mezclar los dos métodos en un trabajo a futuro y crear un lenguaje de dominio específico para el manejo de las entidades replicadas, actualizables en tiempo de ejecución.

Capítulo 4

Diseño de la plataforma flexible y extensible

La implementación de aplicaciones colaborativas es un trabajo arduo y complicado, ya que deben manejar una interfaz capaz de hacer que los usuarios sean concientes del trabajo de los demás. Además tienen que distribuir y sincronizar las entidades compartidas.

Si se utiliza una plataforma que se encargue de administrar las réplicas de las entidades compartidas, ésta permite a los desarrolladores de aplicaciones colaborativas, implementarlas sin tener que hacer el difícil trabajo de diseñar y programar la distribución y la sincronización de las réplicas, dejándoles a los desarrolladores de aplicaciones colaborativas distribuidas únicamente el desarrollo del motor de producción cooperativa y de la interfaz de interacción de los colaboradores. La plataforma flexible y extensible tiene como objetivo permitir a los programadores implementar las aplicaciones colaborativas distribuidas casi como si se tratara de una aplicación local; exceptuando las funciones de: a) la producción cooperativa, y b) el manejo de la conciencia de grupo. Inclusive, la plataforma permite probar diferentes arquitecturas sin tener que modificar la aplicación.

Si se maneja esta plataforma que aisla la interfaz del usuario de las entidades compartidas, sus procesos de control de acceso, distribución y sincronización de sus réplicas y la plataforma mínima facilita el desarrollo de aplicaciones colaborativas debido a que:

- El programador puede definir entidades que satisfagan diferentes arquitecturas de distribución. Además, ofrece funciones que permite modificarlas en tiempo de ejecución soportando una gran variedad de arquitecturas de distribución más eficientes y tolerantes a fallas, sin afectar la disponibilidad de las aplicaciones.

- Las aplicaciones pueden ser más complejas de manera más fácil. Si una aplicación maneja más de un tipo de datos, es posible utilizar más de tipo de entidad compartida. Si se maneja más de un tipo de entidad, se traduce en un mayor

confort a los programadores, ya que pueden escoger la distribución y el control de concurrencia que mejor se adapten a la naturaleza de los datos.

- Las pruebas con diferentes políticas sin tener que cambiar la aplicación son fáciles de realizar. Simplemente éstas se intercambian como piezas de un lego. Estos ajustes inclusive se pueden hacer en tiempo de ejecución utilizando actualizaciones dinámicas de software.

- El trabajo es más fácil de compartir entre los diferentes desarrolladores: unos se pueden encargar de implementar la parte local, diseñando y programando la parte funcional de la aplicación casi como si fuera en modo centralizado, mientras que otros desarrollan las políticas de replicación.

- Cada funcionalidad es implementada una sóla vez para diferentes aplicaciones (pueden reutilizar y compartir el código). Una entidad de datos puede ser desarrollada para una aplicación y utilizada posteriormente en otra. Además, se puede definir un nuevo tipo de entidad a partir de pequeñas modificaciones aplicadas a los tipos de entidades ya existentes.

- Se reduce el tiempo de implementación porque cada parte es un bloque diferente de software, permitiendo que sea viable que las aplicaciones sean implementadas por más de un desarrollador. Además el tiempo disminuye debido al decremento del código redundante.

- Se reduce el número de errores al implementar cada funcionalidad sólo una vez.

- El mantenimiento es más sencillo al no tener que buscar los errores en todo el código del sistema cooperativo.

- Agregar nuevas funcionalidades es más simple, ya que se añaden de manera separada sin mezclarse con el código ya existente.

A partir de un análisis preciso del funcionamiento de un sistema cooperativo, podemos deducir que este tipo de aplicaciones siguen un proceso bien específico (ver figura 4.1). Así, cuando un colaborador localizado en un sitio A quiere compartir y colaborar en la producción de una entidad con otro colaborador en un sitio B, el proceso de trabajo cooperativo sigue las siguientes fases:

1. La entidad está replicada en los sitios de todos los colaboradores, con el fin que cada uno de ellos pueda accederla. Así, por ejemplo, se crea una replica de la entidad en el sitio B.

2. Se carga la entidad en el corazón de la aplicación en el sitio A, para que el usuario en este sitio pueda accederlos.

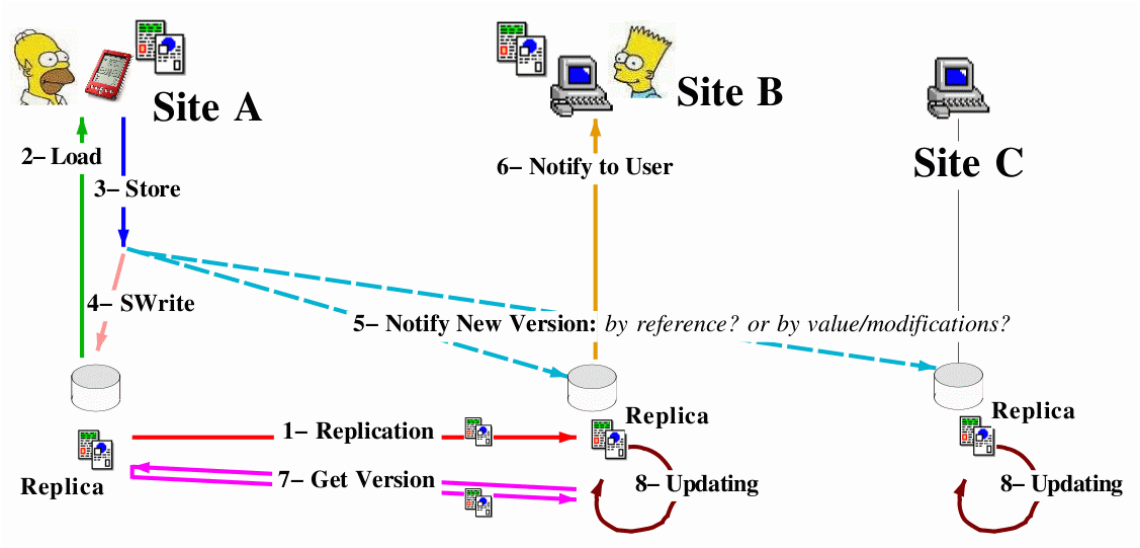

Figura 4.1: Proceso de las aplicaciones cooperativas

3. Cuando el usuario en el sitio A modifica la entidad, la aplicación debe almacenar y distribuir la notificación de los cambios a los sitios donde se encuentra una réplica. El orden en el que se debe realizar los eventos de almacenamiento, notificación y actualización de réplicas depende directamente de los requerimientos de sincronía y distribución de los tipos de datos replicados (ver Capítulo II).

4. Las modificaciones son integradas en la réplica local de la entidad compartida.

5. Las notificaciones de los cambios en la entidad compartida son transmitidas a los sitios de los colaboradores del usuario A. Así, el sitio B recibe una notificación que le indica que la entidad fue modificada en el sitio A.

6. La aplicación notifica a los usuarios en los diversos sitios de colaboración (ver sitio B de la figura 4.1), las modificaciones integradas en la entidad replicada.

7. Si las notificaciones de las modificaciones son enviadas por valor en vez de por referencia (ver 2.1.3), se obtiene directamente la nueva versión de la entidad.

8. Se actualiza la réplica en cada uno de los sitios de colaboración.

En el segundo capítulo se puso en evidencia la necesidad de una plataforma flexible y extensible que facilite el desarrollo de aplicaciones colaborativas distribuidas. En el presente capítulo se presentan sus características de diseño.

Esta plataforma genérica, tal como podemos imaginar, está compuesta de a) una plataforma mínima y b) un conjunto de entidades compartidas, sus tipos y políticas de control de acceso y replicación (ver la figura 4.2).

La plataforma mínima se encarga de integrar los diferentes elementos de la plataforma y definir los mecanismos de comunicación entre ellos (ver figura 4.2).

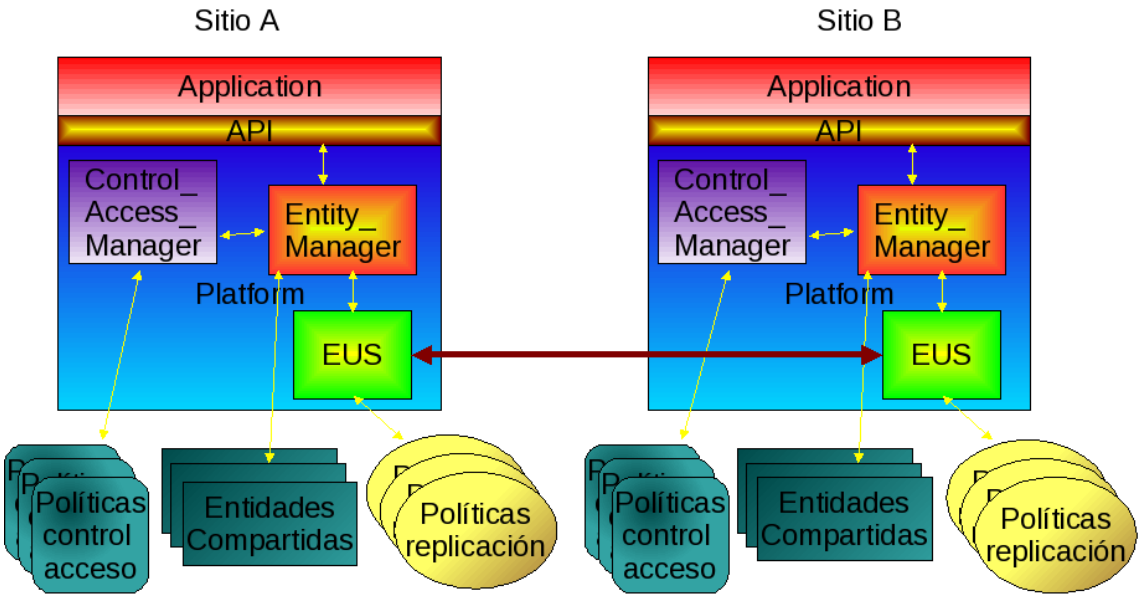

Figura 4.2: Integración de los distintos elementos de la plataforma genérica por medio de la plataforma mínima

Los tipos de entidades y las políticas de control de acceso y replicación quedan separadas de la aplicación y son asociadas mediante estrategias (ver subseccion 3.1.2).

Los módulos funcionales que constituyen la plataforma mínima son los siguientes:

- **Entity_Manager**: el administrador de entidades (Entity_Manager) crea, almacena, borra y modifica las instancias de las entidades compartidas distribuidas. También controla el acceso local a sus métodos.
- **Access_Control_Manager**: el control de acceso valida las operaciones realizadas por los usuarios en los distintos elementos de la plataforma.
- **API**: por medio del API se establece la comunicación de la aplicación local con la plataforma y viceversa.
- **EUS**: (*Entity Update System*) realiza la actualización y sincronización de las entidades replicadas entre los diferentes sitios de colaboración.
- **DAUS**: (*Dynamical Application Updating System*) lleva a cabo el control de las actualizaciones dinámicas de software.

En la primera sección se presenta el diseño de la entidad genérica replicada, sus atributos que nos permiten almacenar los datos y administrarlos de manera flexible. Se presentan los métodos que facilitan el acceso y la administración de los atributos de las entidades, así como sus interfaces. También se presenta el diseño de los *proxies* que permiten el acceso a las entidades remotas.

Cada módulo debe encargarse de validar, administrar o notificar determinados aspectos de la entidad. Por esta razón en la segunda sección se presenta el diseño de los diferentes módulos de la plataforma que permiten validar y sincronizar las operaciones realizadas en las entidades replicadas distribuidas, así como la integración de las aplicaciones colaborativas con la plataforma mediante el API de programación y la meta-interfaz.

Actualizar las entidades y la plataforma en tiempo de ejecución incrementa la disponibilidad de las aplicaciones colaborativas. El actualizador de versiones de software nos permite controlarlas y realizarlas de manera ordenada, coherente y consistente, estableciendo los cambios a realizar y las funciones de transformación que hacen posible la actualización, manteniendo el estado persistente de las clases en un ambiente altamente distribuido y poco fiable. En la tercera sección se presenta el diseño del actualizador, la manera en que se ejecutan y se controlan las actualizaciones dinámicas. Además se establece su interacción con el DAUS, permitiendo la evolución de la plataforma.

4.1 Diseño de las entidades replicadas distribuidas

Las entidades replicadas constituyen el soporte para almacenar, distribuir y sincronizar los datos compartidos entre instancias de las aplicaciones colaborativas distribuidas. Las entidades están formadas por una estructura de datos y los métodos que permiten administrar esta estructura.

4.1.1 Requerimientos de la entidad

Para poder manejar y administrar los datos de manera idónea, las entidades compartidas distribuidas deben ser flexibles, *i.e.* se debe poder modificar las políticas de replicación que administran y sincronizan los datos de manera consistente. Debido a que las entidades son representadas como estructuras de datos compartidas, se propone un modelo de entidad distribuida replicada, que satisfaga los siguientes requerimientos:

- Debe tener interfaces bien definidas y debe ser flexible, tal que permita el intercambio de cada una de sus partes, definiéndose así, nuevas entidades a partir de entidades ya existentes.

- Debe contar con una estructura capaz de representar y almacenar cualquier tipo de datos para aplicaciones colaborativas distribuidas. Además, diferentes tipos de aplicaciones cooperativas deben ser capaces de acceder a las entidades compartidas comunes de un mismo tipo.

- Debe tener asociadas funciones de control de acceso, que permitan restringir el acceso a la entidad y a los datos almacenados.

- Debe contar con los datos de distribución y control de concurrencia que permitan al EUS administrar las réplicas de la entidad.
- Debe adaptarse a cualquier tipo de arquitectura de distribución, para que las aplicaciones colaborativas distribuidas funcionen adecuadamente, independientemente del ambiente de trabajo donde se desenvuelva el usuario.
- Debe permitir la fragmentación de los datos, de acuerdo a los requerimientos propios del tipo de aplicación (ver la sección 2.1.2).

Para implementar las entidades replicadas distribuidas y sus tipos asociados se diseñó un modelo genérico, a partir del cual se derivan las diferentes entidades por medio de la herencia de clases.

A continuación, se presenta este diseño genérico con sus atributos y métodos por omisión. Si un método no satisface los requerimientos de un tipo de entidad en particular, éste se puede redefinir al crear el tipo. De esta forma, la entidad genérica funciona como un *template* o plantilla (ver 2.1.2) para todas las entidades.

4.1.2 Atributos de la entidad

La figura 4.3 muestra una entidad genérica. Posteriormente se describen cada uno de los atributos mostrados en la misma.

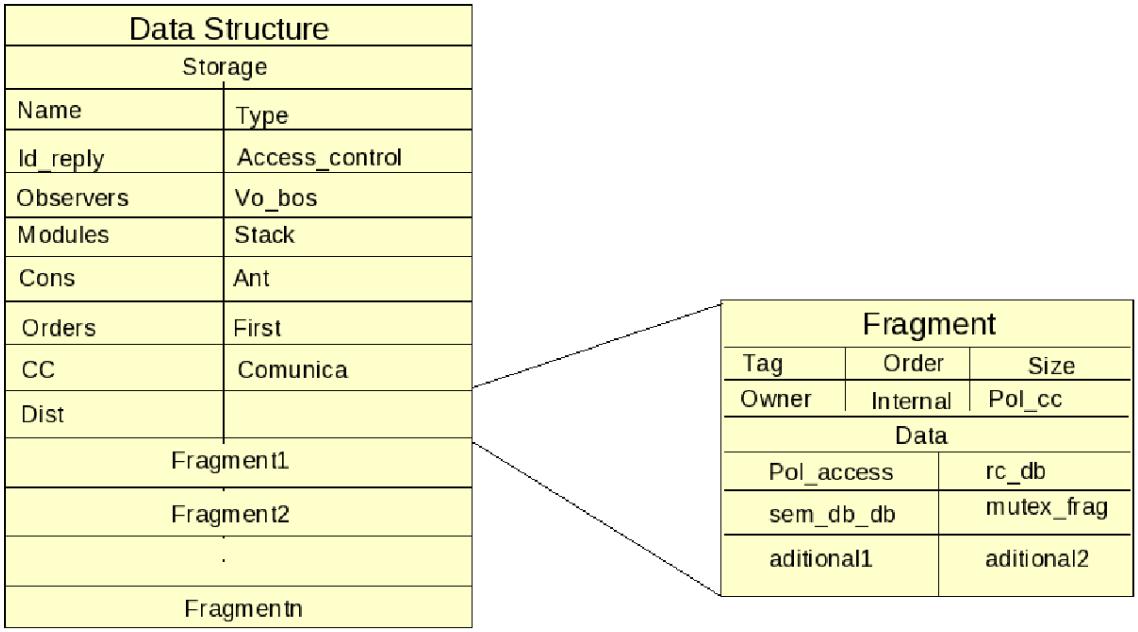

Figura 4.3: Estructura de la entidad genérica propuesta

- **Fragmento de datos:** una aplicación cooperativa define y administra la cooperación estructurando la entidad compartida en diferentes partes (fragmentos).

Se necesita información dedicada para administrar cada fragmento. Por ejemplo, en un editor de texto, se requiere administrar los párrafos del documento y su formato; en un pizarrón es necesario almacenar las figuras dibujadas por los usuarios.

Los campos propuestos para administrar cada fragmento de información son los siguientes:

- **Tag**: etiqueta que nos permite identificar el fragmento de datos como único. Con esta etiqueta sabemos que hacemos referencia a un único fragmento. Para que no existan inconsistencias en la creación de las etiquetas en una aplicación distribuida, éstas se forman a partir del identificador de la réplica y un consecutivo (*e.g.* 1_2, en donde el 1 es el número de réplica donde se creó el fragmento y el 2 el número de fragmento consecutivo creado en esa réplica).
- **Order**: en el caso de que los datos sean un conjunto ordenado, nos permite saber cual es la posición correcta del fragmento dentro del arreglo de datos. Para facilitar su manejo se utiliza una estructura ligada, *i.e.* el order es la etiqueta del siguiente fragmento.
- **Size**: es el tamaño del fragmento en bytes para saber donde inicia y donde termina cuando es almacenado. Es calculado por el sistema. Este campo sólo se requiere cuando el lenguaje no administra el espacio en memoria.
- **Internal**: indica si el fragmento contiene los datos, es una instrucción de búsqueda o la dirección del fragmento (*e.g. http://es.wikipedia.org/wiki/Trabajo_colaborativo.html*). Puede ser una bandera, pero si existe más de un formato para almacenar los datos en la entidad, puede tomar otros valores además del binario.
- **Owner**: es el identificador del usuario que creó el fragmento.
- **Data**: este atributo contiene los datos. Por ejemplo si es un editor de texto, el fragmento de texto, si es un pizarrón, la figura y sus cordernadas. Este campo puede ser un campo único, o un arreglo de campos.
- **Pol_cc**: este atributo está reservado para ser utilizado por la política de control de concurrencia del EUS. Por ejemplo puede colocar un candado, escribir las marcas de tiempo, escribir el vector de relojes lógicos, *etc*.
- **Pol_access**: restringe el acceso a nivel fragmento. Este campo puede ser un arreglo de usuarios con sus accesos, una tabla o una función que asigne accesos de manera dinámica. Valida los permisos para hacer cambios y bajas al fragmento y tiene por omisión el control de acceso de la entidad.
- **rc_db**: número de procesos que leen el fragmento de manera concurrente. Este atributo se utiliza para el control de concurrencia local en el fragmento. Este campo y los siguientes dos son utilizados para implementar el algoritmo de "lectores y escritores" (ver Tanenbaum [35]).

  - **sem_db_db**: semáforo que controla en acceso al fragmento de datos para los procesos de los lectores.
  - **mutex_frag**: semáforo de exclusión mutua para los procesos que escriben en el fragmento.

- **Type**: nombre del tipo de entidad o tipo de datos que administra esta entidad *e.g.* documentos, datos de usuarios, datos de sesiones, agendas, mensajes de texto, *etc*.
- **Name**: nombre de la instancia del tipo de entidad .
- **Id_replica**: identificador de la réplica. Este atributo permite al EUS administrar las diferentes réplicas de las instancias de las entidades en un ambiente distribuido.
- **Access_control**: política de control que restringe a los usuarios el acceso a la entidad y el nombre de la entidad con los permisos de cada usuario (listas de control de acceso *e.g.* lista de administradores, lista de usuarios, *etc*.) .
- **Observadores**: son los diferentes módulos dentro de la plataforma que requieren ser notificados cuando la entidad cambia (*e.g.* el EUS y el API).
- **VoBos**: módulos que se encargan de validar y verificar los cambios en la entidad.
- **Modulos**: conjunto de direcciones de los módulos, con los que tiene comunicación la entidad eventualmente.
- **Pila**: este atributo se utiliza sólo cuando la entidad es una pila circular, indica el número máximo de fragmentos en la pila.
- **Cons**: número consecutivo que nos indica cual es la siguiente etiqueta a crear (distribuidor). Junto con el `Id_replica` se forma el `tag` o etiqueta del fragmento.
- **Ant**: última etiqueta que se dio de alta. Básicamente este campo se utiliza para entidades ordenadas. Nos sirve para identificar más fácilmente el último fragmento que se dió de alta.
- **Orders**: campo *hash* que ayuda a hacer búsquedas rápidas en los datos cuando no se conocen las etiquetas. Por ejemplo, cuando queremos buscar el nombre de un usuario para el control de acceso, nos ayuda a encontrar la etiqueta rápidamente.
- **First**: se utiliza para entidades ordenadas. Si leemos la entidad en orden, es el primer campo a leer.
- **cc**: política de control de concurrencia que sincroniza las réplicas de la entidad. Éste es usado por el EUS para saber qué política de control de concurrencia debe utilizar. Sólo puede ser modificado por un administrador experto.

- **Comunica**: protocolo de comunicación de los datos que utiliza el *EUS* para difundir la notificaciones a las réplicas.
- **Dist**: tabla con los sitios que se deben notificar cuando se realiza un cambio. También incluye la política de distribución utilizada. Éste es utilizado por el *EUS*.

El control de acceso local de las variables generales de la entidad se controla en el *Entity_Manager*, por lo que será visto más adelante.

4.1.3 Métodos de la entidad

Los métodos públicos que permiten administrar la estructura de datos son:

- **Initialize()**: Constructor de la entidad. Inicializa los contadores. También reinicia los *hash* o arreglos (*e.g.* `vobos`, `observers`, `modules`). Se definen los parámetros por omisión que caracterizan al tipo de entidad como son el nombre, la política de control de concurrencia, la política de control de acceso, política de distribución de réplicas, el tipo de entidad y el tipo de comunicación empleada para difundir las notificaciones.
- **frag_add(datos, user, sig=nil, internal=1, pol_access=nil)**: Agrega nuevos fragmentos a la entidad. El proceso para agregar un fragmento es:
  1. Envía la validación a los módulos que verifican las operaciones en la entidad (control de acceso y EUS por el momento).
  2. Se calcula la etiqueta que identifica al fragmento.
  3. Se realiza la alta capturando los datos, el `owner` y el `internal`.
  4. Si el fragmento es externo se guarda con su respectivo formato.
  5. Se verifica el orden de los datos para entidades ordenadas.
  6. Se envía la notificación a los observadores (el API para que notifique a la aplicación y el EUS para que verifique el control de concurrencia y distribuya los cambios a las réplicas y a los *proxies*).
- **add(datos, user, tag, sig, internal, pol_access=nil)**: Permite agregar fragmentos que se dieron de alta en otras réplicas. Esta función es utilizada por el EUS y no realiza validaciones (ya se hicieron previamente) ni calcula la etiqueta. Por lo demás es igual que add_frag.
- **frag_del(tag, user)**: Elimina un fragmento de datos:
  1. Se solicita la aprobación a los módulos que validan las operaciones en la entidad (control de acceso y EUS por el momento).

2. Se eliminan los datos.
3. Se recalcula el orden de los datos de ser necesario.
4. Se envía la notificación a los observadores (el API para que notifique a la aplicación y el EUS para que verifique el control de concurrencia y distribuya los cambios a las réplicas y los *proxies*).

- **del(tag, user)**: Elimina un fragmento de datos pero sin realizar las validaciones.
- **change_frag(tag, data, user, campo="data")**: Permite modificar los datos de un fragmento.

  1. Envía la solicitud de aprobación a los módulos que requieren validar la operación (control de acceso y EUS).
  2. Realiza el cambio de los datos.
  3. Manda la notificación del cambio a los observadores (al API para que distribuya el cambio a la aplicación y al EUS para que distribuya los cambios a las réplicas y los *proxies*).

- **change(tag, data, campo="data")**: Nos permite modificar los datos de un fragmento pero sin realizar las validaciones. Este cambio fue hecho en otra réplica y es el resultado de las notificaciones, por lo que no se requiere validar.
- **frag_split(tag, posición, tag)**: Permite dividir un fragmento en dos. El nuevo fragmento es a partir del parámetro `posición`. En la entidad genérica este método no hace nada sólo es un *hook* (ver subsección 3.1.2) ya que pocas entidades soportan esta operación (*e.g.* documentos). El proceso es el siguiente:

  1. Se solicita la aprobación a los módulos que deben realizar las validaciones (el `Access_Control_Manager` valida el acceso del usuario y el EUS el control de concurrencia en las diferentes réplicas).
  2. Se crea un nuevo fragmento siguiendo el proceso de alta y se pasa el texto al nuevo fragmento a partir de la `pocisión`. Los permisos son heredados del fragmento de origen.
  3. Se notifica los cambios a los observadores (al EUS para comunicar los cambios a las réplicas y al API para notificar a la aplicación).

- **split(tag, posición, tag)**: Permite dividir un fragmento en dos sin realizar las validaciones, ya que es el resultado de la notificación de una modificación hecha en una réplica remota. Las operaciones fueron verificadas previamente en el sitio en donde se originó el cambio.
- **frag_join(tag1, tag2, user)**: Permite fusionar dos fragmentos. El proceso es:

1. Se solicita la validación del EUS y el `Control_Access_Manager`.
2. Se fusiona el segundo con el primero.
3. Se elimina el segundo.
4. Se corrige el orden.
5. Se envia la notificación al EUS para que verifique control de concurrencia y envie los cambios a las réplicas y los *proxies*.
6. Se envia la notificación al API para que notifique a la aplicación.

- **join(tag1, tag2, user)**: Permite fusionar dos fragmentos sin realizar las validaciones.
- **frag_get(tag, user)**: Regresa el fragmento de datos correspondiente al `tag`. Se utiliza para que la aplicación colaborativa lea los datos contenidos en el fragmento. Si no se especifica el fragmento regresa todos los fragmentos almacenados en la entidad.
- **vobos.add(tag, vobos)**: Agrega un módulo de validación a la entidad. Estos son definidos por el sistema.
- **vobo.del.(tag)**: Elimina un módulo de validación. Sólo lo puede modificar un usuario experto.
- **modulo.add(tag, modulo):** Agrega el enlace a cualquier módulo del sistema.
- **modulo.del(tag):** Elimina el enlace con un módulo del sistema.
- **change_ext():** Cambia el contenido de los datos de un fragmento externo (no está almacenado en la entidad sino en un archivo externo). El valor por omisión sólo es un *hook*. En el caso de los archivos, abre el archivo, lo modifica y lo vuelve a guardar.
- **add_ext():** Agrega el fragmento a un archivo externo. El default es sólo un *hook*.
- **del_ext():** Elimina el fragmento almacenado en un archivo externo.
- **join_ext():** Une dos fragmento almacenados en archivos externos.
- **split_ext():** Divide un fragmento almacenado en un archivo externo en dos.
- **ctl_order_add():** Calcula, verifica y controla el orden de la entidad al agregar un fragmento cuando los datos son un conjunto ordenado. También prepara los archivos *hash* para las búsquedas rápidas.
- **ctl_order_del():** Reordena los datos cuando un fragmento es eliminado de una entidad ordenada.

- **buscar():** Busca un dato por medio de un arreglo *hash*, regresando el fragmento donde se encuentra el dato. El valor por omisión es un *hook*. Un ejemplo es el caso de los archivos de acceso. La función `buscar` localiza el usuario por su nombre para ver si tiene los permisos necesarios para realizar una operación.

- **calc_cons():** Calcula la siguiente etiqueta. El *hook* es simplemente un incremental, pero en el caso de entidades que son colas (*e.g.* mensajes de un chat, eventos de un juego o imágenes de una videoconferencia), se reinicializa al ser igual al tamaño de la pila.

- Se deben agregar las funciones `get` y `set` que permitan modificar los atributos `access_contro, comunica, cc, id_replica, name y pila, dist.`

- Se deben agregar las funciones get para los atributos de sólo lectura `cons, ant y first`.

4.1.4 Proxies de las entidades

Como vimos en la subsección 3.1.2 del capítulo anterior, un *proxy* es un representante que se usa cuando un cliente no puede tener una réplica de un objeto. En el caso de los *proxies* de las entidades, no se puede tener acceso al objeto, debido a que se encuentra en un sitio remoto. Un *proxy* provee el medio para tener acceso a una réplica real de la entidad. El *proxy* debe memorizar la ubicación de la entidad remota, y direcciona las llamadas a una réplica real por medio del EUS. De esta manera, desde el punto de vista de la aplicación, el acceso a la entidad parece ser local y el *proxy* junto con el EUS se encargan de convertir la llamada en un mensaje hacia la entidad remota. Mediante un protocolo de comunicación el EUS (más específicamente el MU o *marshal unmarshal*) envía el mensaje al sitio remoto, donde el MU del EUS del sitio remoto se encarga de convertir este mensaje en una llamada local, recibir el resultado y mandarla de regreso al *proxy*, para que éste a su vez, le comunique a la aplicación el resultado de la llamada.

Los *proxies* son utilizados ya que en ocasiones no es recomentable o no es posible tener una réplica de una entidad. Las razones principales por las que no se recomienda tener una réplica son:

- **Dispositivos de baja capacidad**. Así, estos dispositivos (*e.g.* PDA's) son incapaces de recibir, almacenar o procesar el volúmen de información correspondiente al valor de la entidad compartida real, manejada por una aplicación colaborativa debido a la falta de ancho de banda, memoria secundaria o dificultad para procesarla.

- **Grandes volúmenes de información**. Por lo que sólo se realizan copias parciales o se acceden por medio de *proxies*.

- **Seguridad**. Algunos datos requieren un alto nivel de seguridad, por lo que no es recomendable distribuir copias en los diversos dispositivos. Ejemplos de estos,

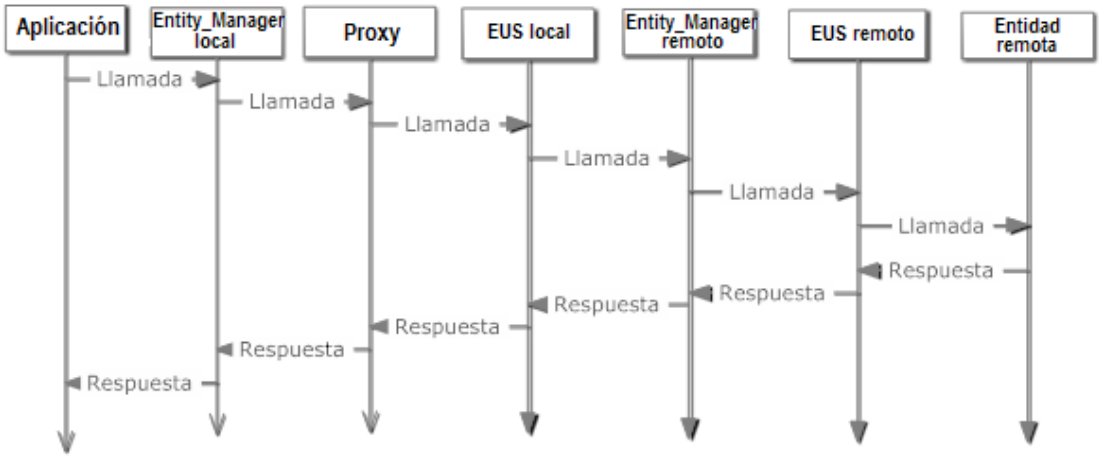

Figura 4.4: Procesamiento de una llamada remota via un proxy

son los datos de control de acceso de algunas aplicaciones. La información debe estar centralizada y sólo se valida por medio de *proxies*.

- **Simplificación de la aplicación**. Cuando se tienen copias de la información, la sincronización de las réplicas de algunas entidades puede llegar a ser muy compleja. Por esta razón se utilizan *proxies*, con una arquitectura de distribución centralizada.

Los *proxies* a pesar de empobrecer el desempeño de la aplicación y ocasionar que se pierda la autonomía de los sitios, en ocasiones son la única opción viable para poder acceder a la información que se encuentra en un sitio remoto.

Los *proxies* se crean cuando la aplicación o algún otro proceso solicita información de una entidad y no se encuentra localmente. El proceso es el siguiente:

1. Se pregunta al EUS la localización de la entidad requerida.
2. El EUS regresa la dirección de la entidad.
3. El *Entity_Manager* crea el proxy con la dirección proporcionada por el EUS.
4. El *Entity_Manager* agrega el proxy a la lista de entidades cargadas. A partir de ahora, el *Entity_Manager* no distingue si es una llamada local o remota.

Una vez que el *proxy* es creado y guardado en el listado de entidades en memoria, cualquier llamada a una entidad remota se realiza por medio de él. El proceso se puede ver en la figura 4.4:

Como es de imaginar el *proxy* tiene los mismos métodos de la entidad y cada método reenvía la llamada al objeto remoto.

Los atributos del *proxy* son el nombre y la dirección del objeto remoto al que representa.

4.2 Diseño de los módulos funcionales de la plataforma

Como se ha introducido anteriormente, la idea princial para el diseño de una plataforma flexible y extensible para el soporte al trabajo colaborativo en un ambiente distribuido, consiste en diseñarla como una plataforma mínima, estructurada en diferentes módulos que interactuan entre sí (ver figura 4.5).

Plataforma mínima

La plataforma mínima integra los módulos del sistema y permite la comunicación entre ellos. El diseño propuesto integra hasta el momento el módulo de control de acceso, el API de programación, el manejador de entidades, el EUS (sistema de actualización de entidades) y el DAUS (Dynamical Application Updates System).

La plataforma mínima constituye la base de la plataforma genérica, a la cual se integran los diferentes módulos funcionales de la plataforma genérica y sus interdependencias funcionales establecidas.

La plataforma mínima ofrece las siguientes funciones:

1. Valida al usuario conectado por medio de la aplicación local, el cual puede ser una persona, un agente u otro sistema.
2. Si no existe un usuario, la plataforma mínima no carga el API y sólo recibe instrucciones por medio del EUS.
3. Crea instancias de cada uno de los módulos funcionales.
4. Establece la comunicación entre los módulos pasándoles las direcciones en memoria donde se encuentran cada una de las instancias.

Administrador de entidades (`Entity_Manager`)

Este módulo agrupa las herramientas y funciones que permiten definir nuevas entidades compartidas, cargarlas en memoria, instanciarlas, salvarlas, borrarlas, entre otros. Este módulo es el responsable de la administración local de las entidades replicadas compartidas. Si una entidad no se encuentra almacenada localmente, carga un *proxy* que se encargue de comunicarse con la entidad remota.

También permite modificar diferentes parámetros de las entidades a partir de las instrucciones dadas por los otros módulos.

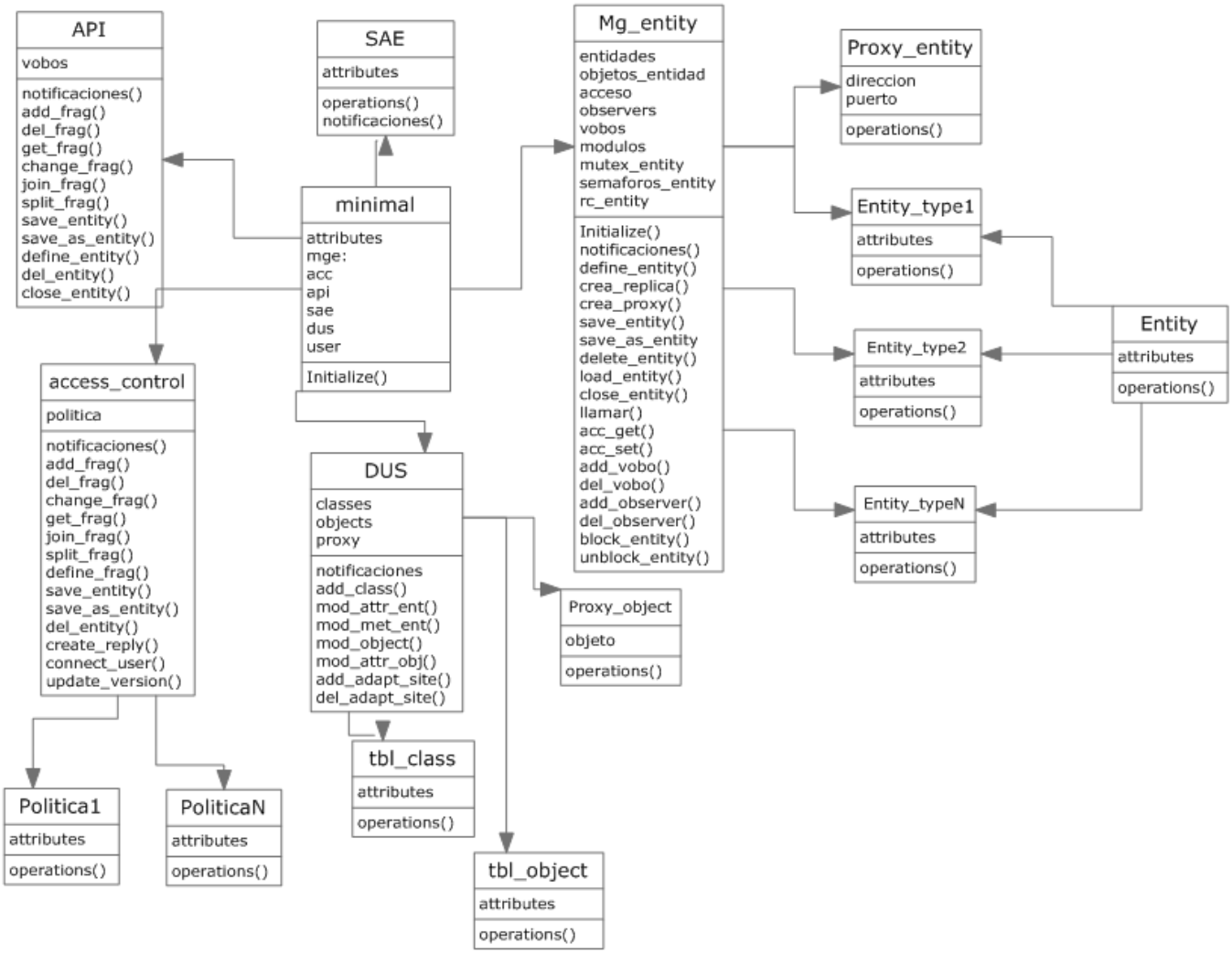

Figura 4.5: Diagrama de clases de la plataforma flexible y extensible

Control de acceso (`Access_Control_Manager`)

El módulo de `Access_Control_Manager` está encargado de la verificación y/o validación de las acciones de los usuarios en las entidades y en la plataforma en general. Para ello maneja diferentes tipos de políticas, que a su vez manejan listados de usuarios y/o condiciones de las variables de ambiente (*e.g.* la fecha).

Para que el `Access_Control_Manager` valide una operación, el módulo y/o la entidad que hacen la solicitud, deben enviar el contexto con el que se debe realizar la evaluación. Básicamente el contexto está formado por la política a utilizar, la tabla asociada (si es que la política requiere una tabla) y en el caso de las operaciones a los fragmentos de las entidades; el campo de pol_access que contienen los filtros o las condiciones de las variables de ambiente. El `Access_Control_Manager` maneja las políticas como diferentes estrategias utilizando el patrón de diseño *strategy* (ver 3.1.2). De acuerdo a la política utilizada se sigue una estrategia diferente. Cada estrategia debe contener métodos de acuerdo al diagrama de clases del la figura 4.5 que permiten validar las diferentes operaciones dentro de la plataforma.

`API` de programación

El API de programación permite usar la funcionalidad de la plataforma a las aplicaciones colaborativas. Por medio del API las aplicaciones realizan llamadas a los métodos de las entidades via el manejador de entidades. El API también recibe las notificaciones de las modificaciones hechas a las entidades para que notifique a las aplicaciones. Básicamente el API constituye una capa de servicios entre las aplicaciones y la plataforma.

El API utiliza el patrón de diseño *facade* (ver subsección 3.2.1), para simplificar las llamadas a los elementos de la plataforma, formando una fachada. El API es una interfaz de alto nivel que simplifica la comunicación entre la aplicación y los diferentes elementos de la plataforma. Como se puede ver en la figura 4.5 consta de los métodos que permiten manipular las entidades, los cuales son resueltos via el manejador de entidades.

Sistema de actualización de entidades `EUS` (*Entity Update System*)

El sistema de actualización de entidades EUS, se encarga de difundir las notificaciones y sincronizar las réplicas de las entidades compartidas distribuidas. Este sistema fue desarrollado por Miguel Navarro Dávila [38], por lo que sólo señalaremos la funcionalidad general del mismo. Para mayor información referirse a su tesis. Las funciones del EUS son:

- **Políticas de control de concurrencia**

  Para poder sincronizar las entidades compartidas es necesario aplicar una política de control de concurrencia entre las réplicas. Dependiendo de esta política, la aplicación tendrá un determinado comportamiento en cuanto a desempeño y número de inconsistencias como se explica en la sección 2.1.2. Por esta razón cada entidad tiene una política de control de concurrencia asociada, por lo que antes y después de realizarse un cambio en la entidad, es necesario notificar al EUS la política de control de concurrencia que debe aplicar y los parámetros con los que debe evaluar (*i.e.* los atributos `pol_cc` y `cc` de la entidad). El EUS autoriza la operación (si se trata de una política pesimista) o resuelve las inconsistencias (si es una política optimista).

- **Política de distribución de notificaciones**

  Cuando una entidad es modificada, es necesario distribuir las notificaciones de las modificaciones a las réplicas distribuidas. Como vimos en la sección 2.1.3, existen varios aspectos a considerar para definir los sitios a los que se deben distribuir los cambios. De acuerdo a estos aspectos se define la arquitectura de distribución de las notificaciones. La política de distribución incluye un listado local de los sitios que se deben notificar al ocurrir la validación de las modificaciones, la cual refleja la arquitectura de distribución vista desde el sitio local. Este listado también considera el tipo de transferencia (por valor o por listado de operaciones) y el método de la sincronización (*pull* o *push*). El nombre de la política de distribución y el listado de sitios a utilizar se encuentra en el campo `dist` de la entidad y se le envía al EUS junto con la notificación cuando ocurre una modificación en la réplica local de la entidad. Si se asocia cada entidad a un listado de notificaciones, permite tener una arquitectura de distribución dinámica, que se adapte a los cambios en el contexto de los usuarios.

- **Políticas de recuperación de fallas**

  Cuando un sitio se desconecta debido a las fallas en la red o si el usuario se desconecta de manera temporal (*e.g.* trabajo nómada), es necesario sincronizar las entidades nuevamente. La política de recuperación de fallas se encarga de la tarea de sincronizar los datos de manera asíncrona.

- **Comunicación entre los sitios de colaboración**

  Para realizar la comunicación entre EUS's remotos como lo muestra la figura 4.2, se utiliza una clase que se encarga de hacer el *marshal* y *unmarshal* de los objetos (MU).

  El MU convierte las llamadas de los objetos remotos en mensajes y los envía al módulo de comunicación. El módulo de comunicación envía el mensaje al sitio remoto utilizando el protocolo de comunicación especificado en la entidad (ver sección 2.1.4). De esta manera las entidades pueden utilizar diferentes protocolos

de comunicación, dependiendo de los requerimientos dados por la naturaleza de los datos. Algunos datos son sensibles a las pérdidas (*e.g.* archivos binarios), mientras que otros pueden ser sensibles a los retrasos (*e.g.* voz y video). Por esta razón las entidades deben utilizar un determinado tipo de comunicación.

4.2.1 Sistema de actualizaciones dinámicas de aplicaciones DAUS (*Dinamical Application Updating System*)

El sistema de actualizaciones dinámicas nos permite recibir las actualizaciones de software y cargarlas de manera coherente y consistente sin bajar la aplicación.

Para que las actualizaciones a la plataforma se realicen sin afectaciones a los usuarios y pueda realizar cualquier tipo de modificación, el *DAUS* debe satisfacer los siguientes requerimientos:

- debe permitir cambiar las políticas asociadas a una entidad (*e.g.* control de acceso, comunicación, control de concurrencia, *etc.*),
- debe permitir redefinir cualquier método de la entidad creando un nuevo tipo de entidad,
- debe permitir sustituir un tipo de entidad por otra y modificar todas las instancias con ese tipo,
- debe permitir sustituir una política (*e.g.* control de concurrencia, control de acceso, control de distribución, *etc*) por otra política similar,
- debe permitir modificar cualquier clase de la plataforma por una nueva versión,
- debe adaptar la comunicación entre los sitios de tal forma que el flujo de los datos sea coherente y consistente, aún durante el proceso de cambio de versión de software.

Actualización de los diferentes elementos de la plataforma

Aunque todos los elementos de la plataforma son clases, para simplificar el proceso de actualización debemos considerar algunas características particulares de sus elementos, principalmente el carácter flexible de la entidad. A continuación se presenta la actualización general de las clases y los elementos que la hacen posible. Después se diseña la actualización de las entidades y las partes flexibles de la plataforma (*i.e.* políticas de control de acceso, de replicación, *etc.*).

Actualización de las clases

La mayor parte de las modificaciones dentro de la plataforma implican el cambio de las entidades y sus tipos. Sin embargo, a veces es necesario modificar las clases que son

parte de los módulos. Para poder sustituir las versiones de software de la plataforma el *DAUS* cuenta con los siguientes elementos:

- **Tabla_Clases:** Estructura de datos que contiene el nombre único de todas las clases, el nombre o número de su última versión y su localización en memoria secundaria.
- **Tabla_Objetos:** estructura que contiene todos los nombres de las instancias de los objetos, la clase a la que pertenecen con su versión de software. Esta tabla tiene como finalidad conocer las instancias de una clase, para así, transformarlas en objetos con la nueva versión de la clase.
- **Adapter_sitio:** Adaptador que sirve como traductor entre entidades de diferentes versiones en sitios remotos. El adaptador se encarga de traducir los mensajes entre una y otra versión. Por ejemplo, supongamos que se tiene un pizarrón colaborativo y las medidas de las imágenes están en pulgadas. El cambio de versión implica que las medidas estén en centímetros. Cuando un mensaje de notificación llega al sitio con una versión diferente, el adaptador hace la transformación de las unidades en el mensaje a la versión de la entidad en el sitio. El adaptador se coloca entre el módulo de comunicación del EUS y el MU (*marshal unmarshal*).

  En caso de que las dos versiones no sean consistentes, desactiva/activa la comunicación entre los nodos conforme se realiza la actualización. Una vez actualizadas las versiones de software, el EUS sincroniza los nodos, comparando cada fragmento, detectando los conflictos y resolviéndolos de acuerdo a su política de recuperación de fallas.
- **Proxy_objeto:** Este *proxy* manda llamar la última versión de la clase del objeto. También se encarga de bloquear las llamadas al objeto durante la actualización de la clase y transferir el estado persistente de la versión anterior a la nueva, utilizando las funciones de transformación que modifican el estado de una versión a otra de manera coherente y consistente. Estas funciones son enviadas por el actualizador.

El proceso de actualización de clases es el siguiente:

1. El EUS recibe la nueva clase, la almacena y envía al DAUS la función de transformación que tiene las instrucciones para realizar la actualización.
2. El DAUS solicita al control de acceso que verifique si el usuario tiene el permiso de realizar las actualizaciones dinámicas.
3. El control de acceso valida al usuario.
4. El DAUS verifica en la tabla `objetos` las instancias de esa clase.

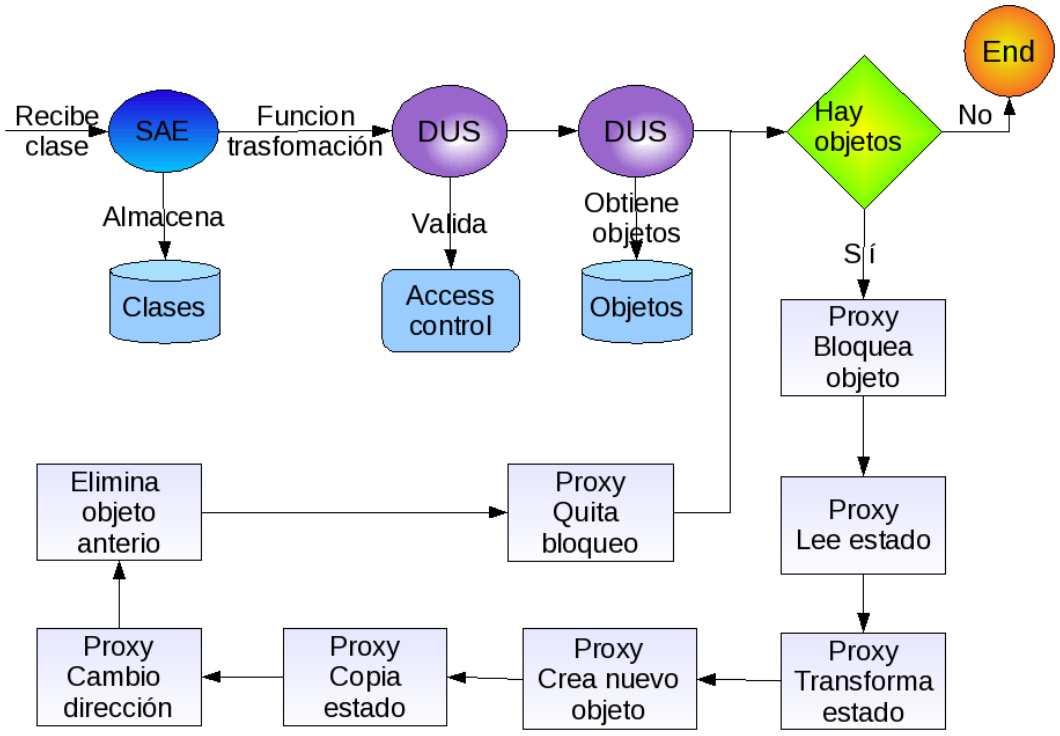

Figura 4.6: Actualización dinámica de las clases

5. El DAUS le solicita al *proxy* que cree un objeto con la nueva versión.

6. El *proxy* bloquea la instancia anterior deteniendo las llamadas a sus métodos.

7. El *proxy* lee el estado anterior del objeto, lo transforma y lo copia a la instancia con la nueva versión de acuerdo a la función de transformación.

8. El *proxy* elimina la instancia anterior.

9. El *proxy* cambia su direccionamiento a la nueva instancia. A partir de este momento, la nueva instancia recibe las llamadas a los métodos.

10. El *proxy* desbloquee las llamadas pendientes.

El proceso de actualización de clases también es presentado en detalle en la figura 4.6. Este proceso se basa en las herramientas presentadas en la sección 3.2.

Modificación de los tipos de entidades

Los tipos de entidades están basados en los patrones de diseño, su flexibilidad permite que su actualización sea menos compleja. Básicamente podemos modificar atributos para que direccionen a una nueva política. También podemos actualizar los métodos, dándonos como resultado un nuevo tipo de entidad. Pero es necesario cambiar las

instancias creadas con el tipo anterior al nuevo tipo, analizando los efectos que los cambios tienen en los datos, para transformarlos. El proceso de transformación es el mismo utilizado para transformar las instancias de las clases. Sin embargo, el manejador de entidades es un *proxy* de las instancias de los tipos de entidades. Éste se encarga de administrarlas, crearlas, controlar el acceso a sus datos, *etc.* Así, también puede funcionar como un *proxy* para llevar a cabo la actualización, por lo que no es necesario utilizar otro *proxy*, como en el caso de las actualizaciones de las clases. El manejador de entidades utiliza una tabla con las instancias de cada tipo de objeto, por lo que no es necesario utilizar la tabla general de las instancias. El proceso de transformación de las instancias de los tipos de entidades es el siguiente:

1. El DAUS solicita al `Entity_Manager` que cree un objeto con la nueva versión.
2. El `Entity_Manager` bloquea la instancia anterior deteniendo las llamadas a sus métodos.
3. El `Entity_Manager` lee el estado anterior del objeto, lo transforma y lo copia a la instancia con la nueva versión de acuerdo a la función de transformación.
4. El `Entity_Manager` elimina la instancia anterior.
5. El `Entity_Manager` cambia su direccionamiento a la nueva instancia. A partir de este momento, la nueva instancia recibe las llamadas a los métodos.
6. El `Entity_Manager` desbloquee las llamadas pendientes.

Modificación de los atributos de las entidades

Los atributos de las entidades que se diseñaron son entre otros: la política de control de acceso, la política de distribución, la política de control de concurrencia, el tipo de comunicación, el tamaño de la pila, *etc.* Estos parámetros son definidos en el constructor del tipo de entidad. Al crear un nuevo tipo que puede ser una variante del anterior (transformamos T en T'), las entidades con la versión previa requieren una redefinición de sus atributos, por lo que es necesario modificarlos en cada instancia existente del tipo de entidad correspondiente y cambiar el atributo `type` para evitar futuras confunsiones. El proceso se puede ver en la figura 4.7.

Métodos de los tipos de entidades

Modicar un método de un tipo de entidad es equivalente a implementar un nuevo tipo de entidad, pero las instancias creadas con el tipo anterior deben sustituirse con instancias del nuevo tipo y pasar el estado persistente de los datos a las instancias creadas con la nueva versión. El proceso para modificar los métodos es el mismo que el utilizado para modificar los atributos.

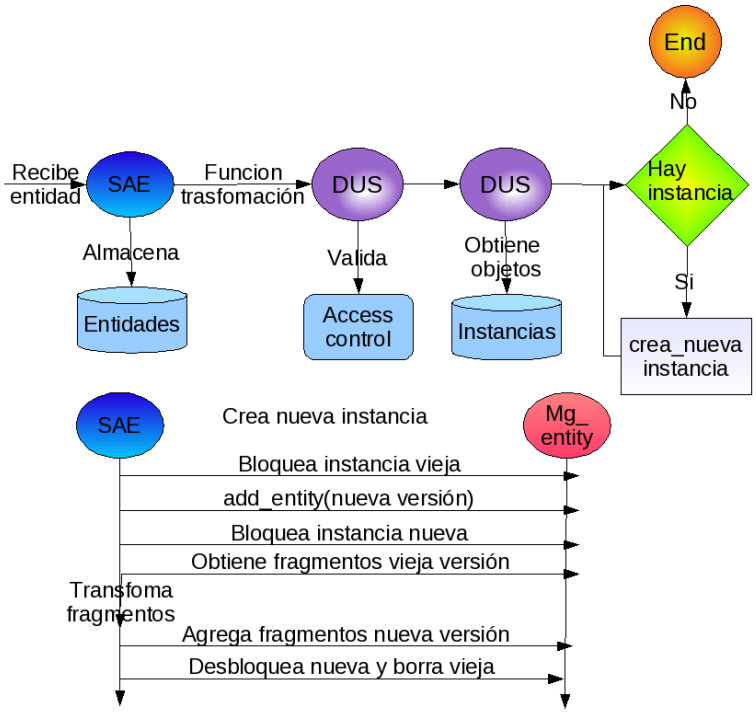

Figura 4.7: Actualizaciones dinámicas de entidades

Actualización de las partes flexibles de la plataforma

Las políticas de control de acceso, de distribución y de control de concurrencia son atributos dentro de las entidades que pueden intercambiarse. Por tal motivo se deben agregar como nuevas estrategias (recordemos que las políticas utilizan el patrón de diseño *strategy*) y modificar todos los atributos de las instancias de las entidades implicadas. El actualizador debe incluir la nueva estrategia como una nueva clase y realizar la actualización de los tipos de entidades utilizando la nueva estrategia.

Meta-Interfaz

La meta-interfaz está constituida por los métodos que permiten al DAUS realizar las actualizaciones dinámicas los cuales se pueden ver en la figura 4.5

4.3 Actualizador de versiones de software

Esta sección presenta el diseño del actualizador de versiones de software para entidades replicadas distribuidas. El actualizador está encargado de enviar las nuevas versiones de software a los sitios. También se encarga de administrar y controlar el

proceso de actualización, reduciendo al mínimo las afectaciones a los usuarios ocasionadas por las inconsistencias entre las diferentes versiones de software.

El actualizador está formado por dos partes principales: a) una agenda que permite establecer la estrategia de actualización y b) el controlador de la actualización, el cual se encarga de enviar las nuevas versiones de software y las funciones de transformación que permiten realizar los cambios de manera coherente y consistente.

4.3.1 Características del actualizador

El actualizador permite hacer cambios de versión de software en las clases de la plataforma en tiempo de ejecución.

El actualizador tiene como función agendar las actualizaciones. En él se plasma el orden en el que debe realizarse los cambios en los nodos de colaboración, *e.g.* en una arquitectura híbrida (ver subsección 2.2), primero deben actualizarse los servidores y después los clientes.

Es recomendable que los cambios se realicen progresivamente, *e.i.* un cambio a la vez, junto con todas sus dependencias. Por ejemplo, si modificamos la firma de un método debemos modificar de manera atómica todas las clases en donde se hace una llamada a ese método. El siguiente cambio no puede realizarse hasta que termine el anterior a menos que los cambios sean disjuntos, es decir no afecten a las mismas clases.

El actualizador debe satisfacer los siguientes requerimientos:

- debe tener una estrategia de actualización, es decir, debe establecer el orden en que se llevará a cabo la actualización de los nodos,
- debe poder posponer la actualización hasta una hora determinada o hasta que se cumpla una condición *e.g.* hora y fecha,
- debe permitir ordenar los cambios de acuerdo a la estrategia de actualización,
- Debe permitir definir una función de transformación que establezca la evolución del estado anterior de un objeto al estado actualizado. Este proceso debe indicarle al DAUS como:
  - obtener el estado del objeto anterior,
  - hacer la transformacion de cada atributo para que se adapte a la nueva versión del objeto,
  - crear la instancia del nuevo objeto,
  - realizar la transferencia del estado al nuevo objeto.
- Se puede agregar un adaptador que permita a) la comunicación entre dos nodos de diferentes versiones o b) rechazar la comunicación entre los nodos durante el proceso de actualización.

4.3.2 Diseño del actualizador

Las partes que hacen posible una actualización dinámica son:

- **Agenda:** permite planear el proceso de actualización y está formado por:
  - Las políticas de actualización, es decir, los criterios u horarios en que se deben actualizar los nodos.
  - Tabla con los nodos a actualizar con los siguientes datos:
    - Dirección: dirección IP del nodo a actualizar.
    - Puerto: puerto por donde el EUS recibe la actualización.
    - Orden: número que indica el orden en que se debe de actualizar el nodo.
- **Tabla de actualizaciones:** contiene la información de los cambios a realizar e incluye las direcciones de los archivos a enviar con las nuevas clases. Igualmente incluye las funciones de transformación que permiten realizar la actualización transparentemente. También contiene la dirección y el nombre de las clases que se deben sustituir.
- **Distribución:** se encarga de enviar los cambios al DAUS que se encuentra en cada nodo para que se lleven a cabo las actualizaciones.

4.3.3 Proceso de actualización

Una vez que se cumple la condición que dispara la actualización, el proceso de cambio de versión se realiza de acuerdo a la figura 4.8 y se describe a continuación:

1. El actualizador envía la notificación a los nodos que requieren ser actualizados primero junto con el adaptador de sitos, si se requiere que los mensajes de notificación entre los sitios, sean transformados para que puedan coexistir las dos versiones de software en el sistema colaborativo distribuido.
2. Si los mensajes de actualización son afectados por el cambio de versión, el DAUS coloca el adaptador.
3. El actualizador envía el archivo que contiene las nuevas clases y las funciones de transformación que hacen posible el cambio de versión, manteniendo el estado de las clases de manera persistente.
4. El EUS recibe los cambios, salva las nuevas clases y entrega las funciones de transformación al DAUS.
5. El DAUS solicita, administra y controla el proceso de actualización con los elementos de la plataforma correspondientes.

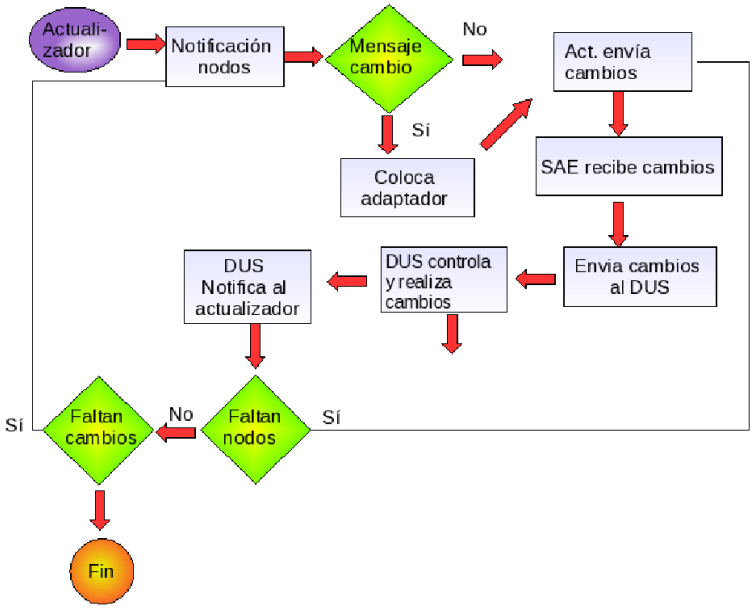

Figura 4.8: Proceso de actualización

6. El DAUS notifica al actualizador el resultado del cambio de versión.

7. Si hay más nodos por actualizar, el actualizador continúa con los faltantes, regresando al punto 3.

8. Cuando ya no hay más nodos por actualizar, el actualizador solicita al DAUS que remueva el adaptador.

9. Si hay más modificaciones, el actualizador continúa con el siguiente cambio regresando al punto 1.

En este capítulo se presentó el diseño de la plataforma flexible y extensible, así como el diseño de la entidad genérica, a partir de la cual se derivan los diversos tipos de entidades, que permiten a la plataforma adaptarse a diferentes políticas de control de concurrencia, control de acceso y distribución. En el siguiente capítulo se prueba el diseño analizando los procesos de algunas aplicaciones colaborativas y su comportamiento utilizando nuestro enfoque.

Capítulo 5

Caso de estudio: diferentes aplicaciones colaborativas distribuidas

En este capítulo se realiza el diseño de diferentes aplicaciones colaborativas distribuidas utilizando el modelo de entidad genérica, el cual se presentó en el capítulo anterior. Se verifica el diseño de la plataforma y se valida la interacción entre los diferentes módulos que la conforman.

5.1 Las aplicaciones colaborativos de tipo chat

Se empezará analizando una aplicación colaborativa tipo chat, debido a que son de las aplicaciones más sencillas de conceptualizar. Se van a estudiar primero con la finalidad de retomar algunos de los conceptos para las aplicaciones colaborativas que se verán posteriormente.

En la subsección 2.2.2 se vió el comportamiento de los chats. Así como algunos ejemplos en el mercado y su comportamiento (*e.g. Exodus* [42], *Messenger Live* [47], Google Talk [48], entre otros). La arquitectura recomendada es una arquitectura híbrida con un servidor central que sirva como punto de reunión y que maneje las sesiones de los usuarios. Los mensajes entre los usuarios pueden ser replicados en los sitios de los participantes.

Básicamente una aplicación chat requiere 3 tipos de entidades: Las entidades de control de acceso, las entidades con las listas de los contactos de los usuarios y las colas de mensajes que almacenan los datos de la comunicación entre los usuarios.

5.1.1 Entidades de control de acceso

Para administrar el acceso de los usuarios a este tipo de aplicaciones colaborativas, se requieren al menos dos instancias de las entidades de control de acceso: `usuarios` y `administradores`. La tabla `usuarios` sólo puede ser consultada por los usuarios de la aplicación y modificada por los administradores, aunque los usuarios pueden

modificar su propia información como lo podemos observar en el campo de pol_access de las figura 5.1 y la figura 5.2. La entidad de `administradores` sólo puede ser modificada por los mismos administradores. Cabe señalar que en algunos *chats* se crea la ilusión de que los usuarios crean sus propias cuentas, pero en realidad es un agente administrador del sistema el que se encarga de autenticar y realizar el alta del usuario.

Data Structure							
Name: Users				Id_replica: 1			
Type: Control de acceso				Control_Access: Admin			
Distribution: Site E				Comunication: TCP			
Cons: 1							
Concurrency_control: Candados optimistas con coord. Por fragmento							
Tag	Usuario	Lectura	Escritura	Dirección	Pol_cc	Pol_access	Internal
0_0	Admin	1	1		0	Admin	1
0_1	User A	1	1	Site A	0	Admin, A	0
0_2	User B	1	1	nil ->Site B	0	Admin, B	0
1_0	User C	1	1	Site C	1	Admin, C	0

Figura 5.1: Entidad de control de acceso de usuarios en el sitio D

Data Structure							
Name: Users				Id_replica: 0			
Type: Control de acceso				Control_Access: Admin			
Distribution: Site D				Comunication: TCP			
Cons: 3							
Tag	Usuario	Lectura	Escritura	Dirección	Pol_cc	Pol_access	Internal
0_0	Admin	1	1		1	Admin	0
0_1	User A	1	1	Site A	1	Admin, A	1
0_2	User B	1	1	nil ->Site B	2	Admin, B	1
1_0	User C	1	1	Site C	0	Admin, C	1

Figura 5.2: Entidad de control acceso de usuarios en el sitio E

Por cuestiones de seguridad es recomendable que los usuarios accesen a estas entidades mediante *proxies* (ver subsección 4.2.4).

Para incrementar la disponibilidad del chat, se pueden tener varios servidores en los cuales están replicadas estas entidades. Debido a que los cambios de estas enti-

dades son raros, la información a modificar es de tamaño reducido y las modificaciones concurrentes *quasi* inexistentes. Pero las modificaciones a este tipo de entidades no cuentan con una conciencia de grupo, sino que son bases de datos distribuidas. Por esta razón se propone el uso de candados optimistas, *i.e.*, cuando un usuario desea modificar la información, solicita el candado, pero para que la aplicación tenga un mejor desempeño, el usuario puede empezar a modificar los datos en lo que obtiene el candado, si el candado es rechazado se deshacen los cambios.

Para incrementar la disponibilidad, consideramos una arquitectura utilizando servidores con balanceo de cargas. La coordinación es a nivel fragmento y se lleva a cabo en estos servidores. Esta variante es propuesta por Decouchant *et. al.* [9]. Un fragmento en un sitio puede tener 3 estados: no se tiene la coordinación y por lo tanto no se puede modificar localmente, se tiene la coordinación sin bloqueo es decir está disponible (se puede modificar localmente o por medio de un *proxy*) y se tiene la coordinación pero está bloqueado (está siendo modificado localmente o mediante un *proxy*).

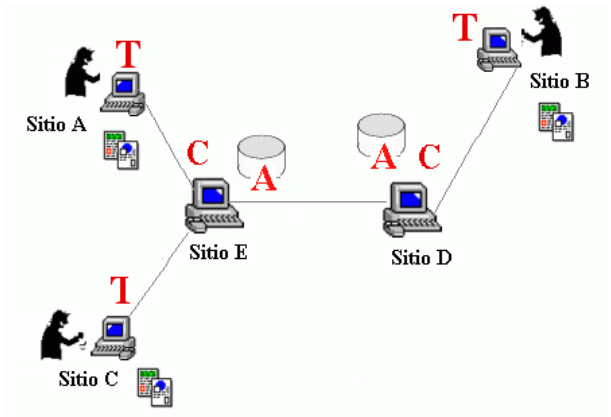

Figura 5.3: Arquitectura manejada para el control de acceso

La figure 5.3 muestra la arquitectura utilizada para el control de acceso. La información es almacenada en los sitios E y D. Los diferentes usuarios se conectan a alguno de los serviedores centrales, son validados y obtienen el estado de sus contactos.

Si se observa el campo de `id_replica` de la figure 5.1 la entidad fue creada en el sitio D (`id_replica=0`) y la figure 5.2 muestra la réplica en el sitio E (`id_replica=1`). Si se se presta atención al campo `pol_cc` de ambas figuras se puede ver que la coordinación de los fragmentos `0_0` `0_1` y `1_0` se encuentra en D y el fragmento `0_2` se encuentra en E. Cuando el usuario B se conecta en E, el fragmento correspondiente a su estado en la entidad de usuarios se bloquea. Si la coordinación del fragmento no se encuentra en el sitio donde un usuario se conecta, el servidor deberá obtenerla, solicitándola al otro servidor.

La entidad fue creada en E y replicada en D de acuerdo al número de réplica.

Cuando un usuario se conecta, la aplicación local solicita el cambio en la tabla de `usuarios`, modificando la dirección en donde se encuentra.

Todos los datos son internos, excepto el fragmento 0_0. Este fragmento direcciona a la entidad de `Admin`. De tal forma que si un administrador quiere usar la aplicación, el control de acceso primero lo buscará en la tabla de `users`. Al no encontrarlo lo buscará en los fragmentos externos, en este caso en la tabla de `Admin` la cual se puede ver en la figura 5.4

Data Structure							
Name: Admin				Id_replica: 0			
Type: Control de acceso				Control_Access: Admin			
Distribution: Site D				Comunication: TCP			
Cons: 3							
Concurrency_control: Candados optimistas con coord. Por fragmento							
Tag	Usuario	Lectura	Escritura	Dirección	Pol_cc	Pol_access	Internal
0_1	Admin1	1	1	nil	1	Admin	1
0_2	Admin2	1	1	nil	1	Admin	1
1_0	Agent1	1	1	Site D	0	Admin	1

Figura 5.4: Entidad de control de acceso de los administradores

Otro campo que se requiere para administrar a los usuarios es el campo `cons`, que es simplemente un consecutivo que se incrementa cuando un usuario nuevo es agregado a la tabla. Las etiquetas se forman a partir del número de réplica donde fue creado el registro y el `cons`. Utilizamos el número de réplica para diferenciar 2 registros que fueron dados de alta simultáneamente en sitios diferentes.

Las operaciones que podemos hacer a la entidad de acceso son:

- *Add_user()*: Alta de un usuario.
- *Del_user()*: Elimina un usuario.
- *Modify_user()*: Modifica los datos de un usuario.
- *Get_user()*: Obtiene la información de un registro específico.
- *Seek_user()*: Busca los permisos de los usuarios mediante el nombre del usuario.

5.1.2 Entidades de contactos de los usuarios

Esta entidad tiene como finalidad implementar una función de conciencia de grupo, *i.e.*, "¿Quién está presente para chatear?" (ver la figure 5.5). Ésta contiene los usuarios que requieren ser notificados cuando un usuario se conecta, los cuales podrían estar interesados en establecer una comunicación y por lo tanto requieren saber de su ubicación.

Data Structure				
Name: Cont_A		Id_replica: 0		
Type: Contactos		Control_Access: A, Admin		
Distribution: Site A, Site D		Cons: 2		
Comunication: TCP				
Concurrency_control: Candados optimistas con coord. Por fragmento				
Tag	Usuario	Dirección	Pol_cc	Internal
1_0	User G	nil	1	1
1_1	User B	nil ->Site B	2	1
1_2	User C	Site C	1	1

Figura 5.5: Entidad de contactos de los usuarios

Estas notificaciones deben ser llevadas a cabo por la aplicación que administra el chat, ubicada en los sitios D y E de la figura 5.3. Cuando un usuario inicia una sesión, la aplicación debe recorrer los contactos del usuario y actualizar en la tabla de cada contacto el estado del usuario que se conectó.

Estas entidades pueden ser replicadas localmente en el sitio del usuario, con la finalidad de soportar fallas en el servidor central, por lo que al conectarse un usuario, la aplicación debe solicitar la creación de una réplica para consulta de la información. Como vemos en la figura 5.6, la arquitectura utilizada para este tipo de entidad es replicada, aunque si algún dispositivo no tiene capacidad para almacenar o procesar la entidad, puede acceder por medio de un *proxy*, como es el caso del sitio C de la figura 5.6.

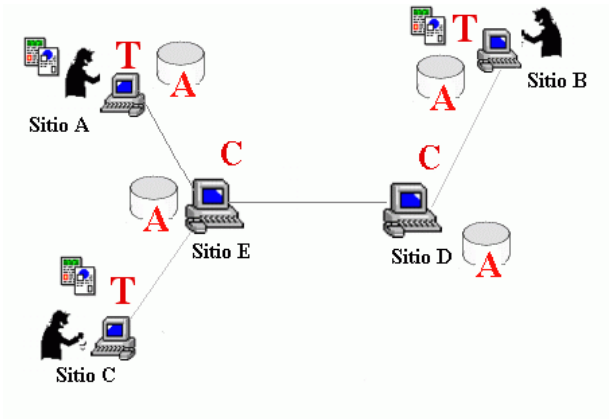

Figura 5.6: Arquitectura de contactos de los usuarios

Los usuarios pueden agregar, modificar, eliminar o consultar contactos. Estas actualizaciones se deben ver reflejadas en la copia de los servidores donde se almacenan las entidades (sitios D y E). De igual manera cuando el estado de un usuario se modifica, se debe notificar y realizar el cambio en las copias de los usuarios. Este proceso de sincronización es llevado a cabo por el EUS, utilizando una política de control de acceso basada en candados con coordinación a nivel fragmento, la cual se explicó en la sección que presentó las entidades de control de acceso.

5.1.3 Entidades de colas de mensajes

Cuando un usuario quiere establecer comunicación con otro(s), se debe crear una entidad que administre la cola de los mensajes. Ésta se puede crear en el servidor donde se almacenan las entidades de control de acceso y contactos. Pero crear la cola en alguno de los servidores tiene varios inconvenientes:

- La comunicación entre los usuarios se vuelve más lenta porque todos los mensajes entre los usuarios son desviados hacia uno de los servidores.
- El servidor se vuelve un cuello de botella, ya que se incrementa de manera considerable el número de procesos que tiene que atender.
- Al incrementarse el número de elementos de red (sitios e interconexiones) que intervienen en la comunicación entre los usuarios, aumenta la probabilidad de falla.

Si se elimina el servidor central para la coordinación de la cola de mensajes, entonces tenemos que lidear con las inconsistencias generadas debido al orden de los mensajes como vemos en la figura figura 5.7.

Data Structure	
Name: chat	Id_replica: 0
Type: Chat	Control_Access: A B
Distribution: Site B	Stack: 20
Cons: 2	Ant: 1_1
Comunication: TCP	First: 0_0
Concurrency_control: Ninguno	

Tag	Owner	Mensaje	Order
0_0	A	Hola	0_1
0_1	A	¿Cómo estas?	1_0
1_0	B	¿Qué cuentas?	1_1
1_1	B	bien ¿y tú?	

Data Structure	
Name: chat	Id_replica: 1
Type: Chat	Control_Access: A B
Distribution: Site A	Stack: 20
Cons: 2	Ant: 1_1
Comunication: TCP	First: 0_0
Concurrency_control: Ninguno	

Tag	Owner	Mensaje	Order
0_0	A	Hola	1_0
1_0	B	¿Qué cuentas?	0_1
0_1	A	¿Cómo estas?	1_1
1_1	B	bien ¿y tú?	

Figura 5.7: Cola de mensajes entre los sitios A y B

Si queremos que ambos sitios muestren los mensajes en el mismo orden, se puede introducir una política utilizando relojes, para determinar el orden de los eventos. Sin embargo si se evalua el efecto que ocasiona en los usuarios el visualizar los mensajes en diferente orden, éste es mínimo: cada sitio conserva el orden de acuerdo al momento en el que el usuario recibio el mensaje, por lo que realmente no se requiere mantener un orden estricto.

Por estas razones la arquitectura más adecuada parece ser una arquitectura P2P con control de concurrencia optimista sin corrección de inconsistencias, como vemos en la figura 5.8. La cola de mensajes generalmente no requiere ser almacenada, pero de ser necesario se pueden enviar los mensajes al servidor central y ser almacenados en el orden en que se reciban.

Figura 5.8: Arquitectura de la cola de mensajes entre los sitios A y B

Como se puede observar en la figura 5.7, la estructura define una cola circular ordenada. Los campos que permiten controlar el orden de la cola circular son *ant*: que es el último registro que se dió de alta en la entidad, *first* que es el primero en la cola, *cons* que es consecutivo para crear las etiquetas y *stack* que es el tamaño de la cola circular.

El control de acceso está limitado a los participantes de la conversación y la única operación que pueden realizar consiste en agregar un mensaje.

Otro campo que vemos es el *owner* que identifica el usuario que agregó el mensaje.

5.1.4 Proceso en los chats

Aunque existen otras operaciones características de cada entidad, en el chat existen 3 operaciones principales. A continuación veremos el proceso que se lleva a cabo con cada una de éstas.

Conexión de un usuario
Cuando un usuario se conecta a la aplicación es necesario:

1. **Validar el usuario**. El `control_access_manager` requiere verificar que sea un usuario de la aplicación. Para hacerlo es necesario acceder a la entidad de `users` en el sitio `E` o `D`. El acceso a la entidad remota se realiza por medio de un *proxy* y el EUS (ver figura 5.9).

Figura 5.9: Proceso de validación y conexión de un usuario

2. **Actualizar la ubicación del usuario en las entidades de sus contactos**. Se requiere notificar a la aplicación encargada de la coordinación que el usuario se encuentra conectado. Si se tiene una réplica local, se modifica la réplica y el EUS distribuye la notificación a los sitios `D` y `E`. Al modificar el estado en el sitio de `D`, se notifica via el API del cambio. Si localmente se cuenta con un proxy, éste sirve como representante y via el EUS se modifica la entidad en el sitio remoto `D`. Al cambiar la entidad en el sitio `D` se notifica a la aplicación que se encarga de la coordinación. Esta aplicación solicita el cambio del estado del usuario (*i.e.* disponible para chatear) en las entidades `contactos` de sus colaboradores (ver en la figura 5.9).

Envio de mensajes entre los usuarios

Es necesario crear las entidades que permitan establecer la comunicación a los usuarios con uno de sus contactos activos. El proceso se puede ver en la figura 5.10 y consiste en:

1. **Se agrega el mensaje en la entidad local.** Si es el primer mensaje que se agrega, crea una nueva cola de mensajes. Sino, simplemente se agrega el mensaje a la entidad local. La entidad envía la notificiación tanto al EUS como a la aplicación local.

2. **Se notifica en los sitios participantes.** El EUS envía la notificación a los sitios de los colaboradores para que agreguen el mensaje a la réplica (sitio B). La réplica notifica a la aplicación para que el mensaje llegue al usuario B.

Desconexión de un usuario Una desconexión se puede realizar por solicitud de la aplicación del sitio del usuario o la aplicación del servidor central (sitio *D* o *E* según sea el caso). El servidor central puede estar enviando un *heart-beat* a cada sitio conectado, después de *n heart beat* sin respuesta, puede solicitar la desconexión del usuario. La figura 5.11 muestra el caso en el que el usuario solicita la desconexión. La aplicación solicita modificar el estado del usuario ya sea en la réplica local o por medio del *proxy*, en el sitio de coordinación `E` o `D`. Cuando se modifica el estado del usuario en la entidad `Contactos`, se notifica a la aplicación de coordinación y ésta solicita el cambio de estado del usuario en las entidades `Contactos` de los colaboradores.

Ésta es una de las formas en que se puede implementar un chat utilizando el modelo propuesto, aunque no es la única, pero consideramos que es la arquitectura que provee un mejor confort a los usuarios, de acuerdo al desempeño de la aplicación y la seguridad, considerando que las inconsistencias en este tipo de aplicaciones no afectan a la colaboración.

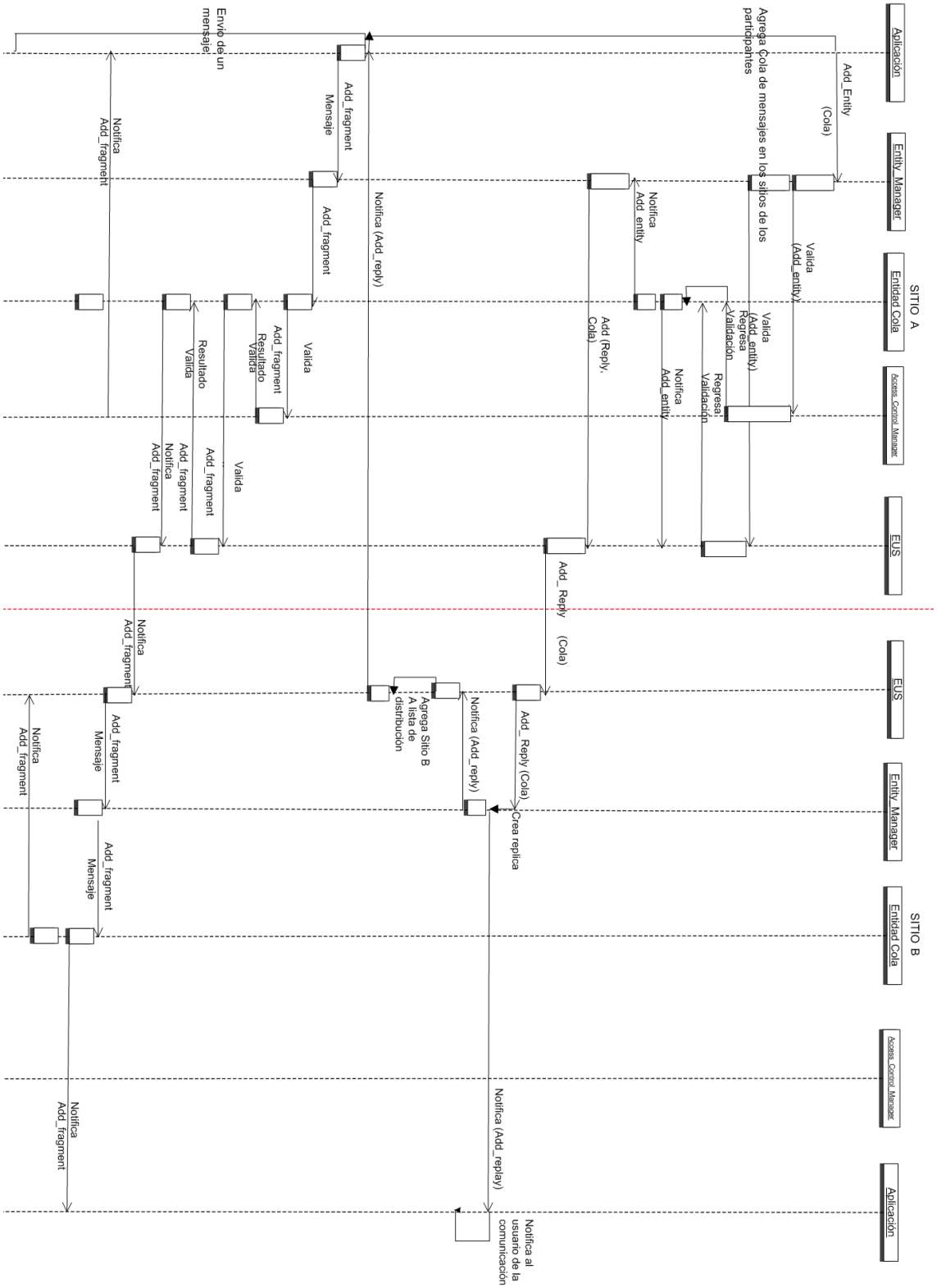

Figura 5.10: Proceso de envío de mensajes entre los usuarios

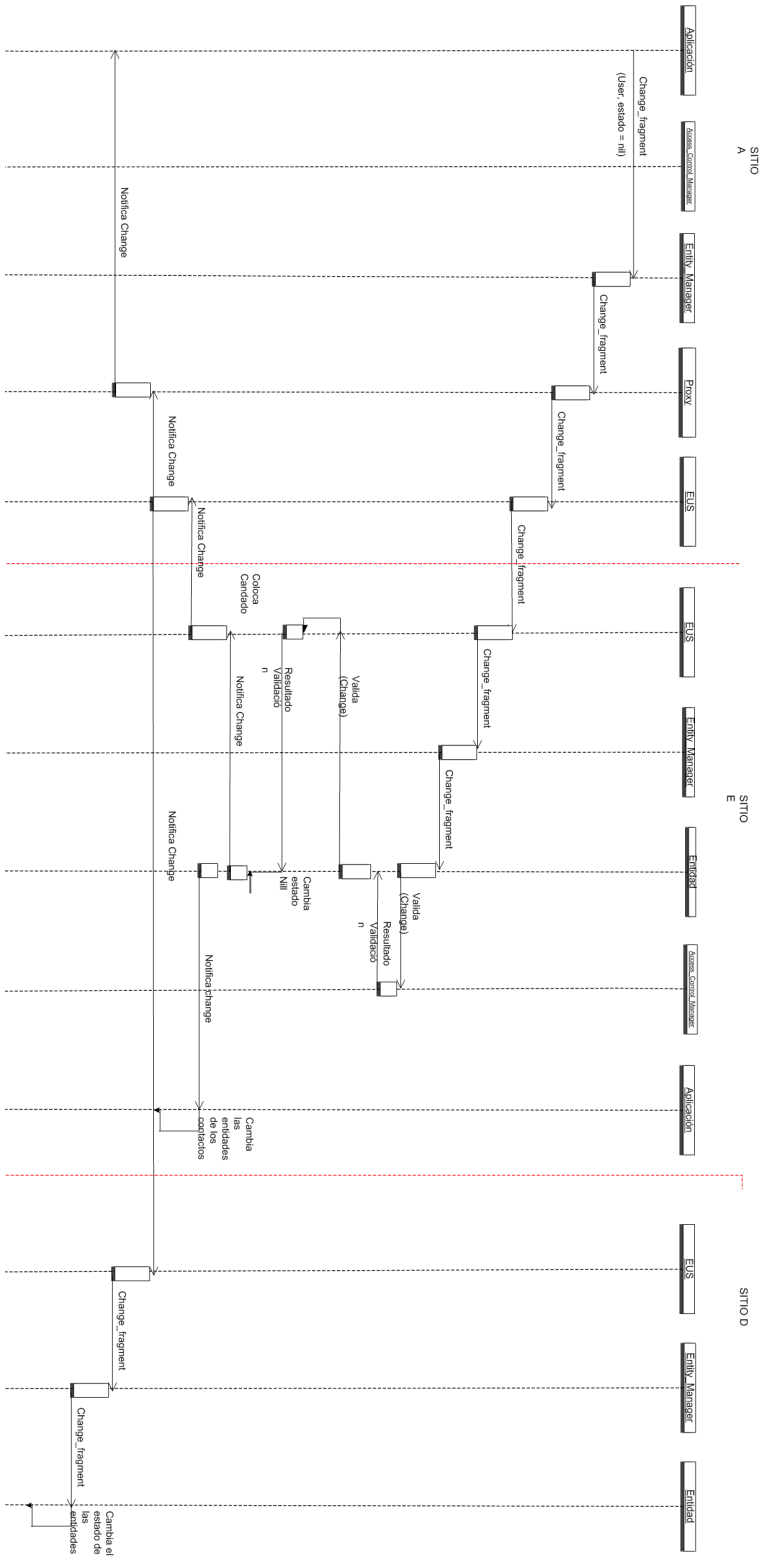

Figura 5.11: Proceso de desconexión de los usuarios

5.2 Editores cooperativos asíncronos de documentos

Los editores cooperativos son aplicaciones que requieren un conjunto más elaborado de operaciones sobre las entidades que los chats. Para que este tipo de aplicaciones funcionen adecuadamente, se requieren dos tipos de entidades: las entidades de control de acceso y los documentos.

5.2.1 Entidades de control de acceso

Los usuario pueden tener diferentes roles (*e.g.* autor, administrador, crítico, revisor, *etc.*), que definen diferentes permisos de acceso sobre las distintas partes del documento. Los usuarios pueden tener diferentes permisos (*e.g.* lectura y/o escritura) como podemos ver en la figure 5.12.

[H]

Data Structure						
Name: Users				Id_replica: 0		
Type: Control de acceso				Control_Access: Admin		
Distribution: Site D				Comunication: TCP		
Cons: 3						
Tag	Usuario	Lectura	Escritura	Dirección	Pol_cc	Internal
0_0	Admin	1	1		1	0
0_1	User A	1	1	Site A	1	1
0_2	User B	1	1	nil ->Site B	2	1
1_0	User C	1	0	Site C	0	1

Figura 5.12: Entidad de control de acceso para los editores de texto

Se pueden crear grupos de usuarios y otorgar los permisos de acuerdo al grupo al que pertenezcan. Éstos se crean y se administran de la misma manera que los administradores en la tabla de usuarios descrita en la aplicación de los *chats*. Así, para simplificar el análisis sólo creamos el grupo de administradores y el de usuarios. Los usuarios pueden tener acceso total para crear, modificar y borrar las entidades tipo documento o pueden tener acceso de sólo lectura como el *user C* de la figura 5.12. A diferencia de las entidades de los *chats*, los usuarios pueden tener acceso a toda la entidad o a ningún fragmento, por lo que los accesos se establecen a nivel entidad.

Utilizaremos la misma arquitectura para las entidades de control de acceso que se utilizó en las entidades de control de acceso de los *chats*.

5.2.2 Entidades de documentos

Las entidades de los documentos son un poco más complejas que las entidades vistas anteriormente. Es una entidad de fragmentos ordenados, como muestra la figura 5.13. Por esta razón requerimos de los campos que nos permiten establecer el orden. El campo *Order* contiene la identificación del fragmento que sigue, el *first* cual es el primero y el *last* el último. Estos campos facilitan las altas de los nuevos fragmentos, asi como el eliminarlos.

Data Structure				
Name: Text1		Id_replica: 0		
Type: Archivo		Control_Access: Users		
Distribution: Site D		Comunication: TCP		
Cons: 3		First: 0_0		
Ant: 1_1				
Concurrency_control: Candados con coord. Por fragmento				
Tag	Datos	Pol_cc	Order	Internal
0_0	\begin {document} \begin {courier} Erase una vez....}\end {courier} \end {document}	1	1_0	1
1_0	En un lugar muy lejano, habia ...	0	0_1	0
0_1	http://www.cinvestav.setup.html	1	0_2	0
0_2	Archivo.dat	2	0_1	0
1_1	select * where reg=1	0		0

Figura 5.13: Entidad de un editor de texto

En el caso de la entidades de archivos los framentos puede definirse de muchas maneras: el campo de *internal* indica si los datos son almacenados en la entidad o hace referencia a algún archivo externo. La figura 5.13 ejemplifica algunos de ellos:

- El fragmento 0_0 es interno, contiene el texto y algunos comandos de formato.
- El fragmento 1_0 también es interno, pero es texto plano (*i.e.* sin formato).
- El fragmento 0_1 direcciona hacia una página Web; al solicitar el fragmento se debe enviarla.
- El fragmento 0_2 direcciona hacia un archivo de texto plano. Así, para procesar un acceso se debe abrir el archivo, leerlo y enviar el contenido.
- El fragmento 1_1 direcciona hacia una base de datos, por lo que se debe conectar a la base de datos y hacer una consulta.

El resto de los campos son similares a los de las entidades explicadas anteriormente.

Como se vió en la subsección 2.2.4 la arquitectura propuesta para las entidades de los documentos de un editor cooperativo de documentos se controlan a nivel fragmeto. Los usuarios pueden accesar a los documentos mediante un *proxy* o crear una réplica de ellos, dependiendo de la capacidad del dispositivo con el que accesan y la confiabilidad de la red, como se ve en la figura 5.14. Para incrementar la disponibilidad de los documentos o permitir un mejor desempeño de la aplicación, los usuarios que están trabajando desde sitios remotos, se puede conectar con más de un servidor, aunque la coordinación de cada fragmento se encuentra centralizada en sólo una de las réplicas.

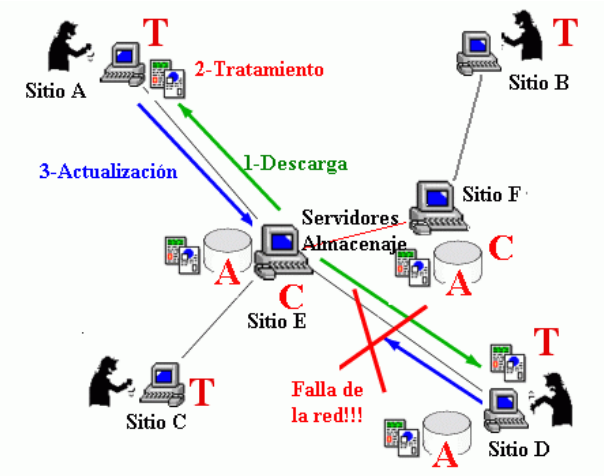

Figura 5.14: Arquitectura utilizada para las entidades de los documentos

5.2.3 Operaciones de edición y procesos en la plataforma

A continuación se describen las operaciones que pueden realizarse en una aplicación de editores de documentos y la forma que interactúan dentro de la plataforma.

Conectar usuario
Valida que un usuario tenga derecho de usar la plataforma, *i.e.* pertenezca a la entidad de `users`. En la figura 5.15 observamos el proceso de conexión suponiendo que las entidades de control de acceso se encuentran en un sitio remoto y la conexión se realiza mediante un *proxy*.

Cargar documento
Permite cargar una entidad almacenda. Si no se encuentra localmente, crea un *proxy* y solicita que se cargue remotamente, que es el caso que muestra la figura 5.16. Este proceso se manda llamar cuando se requiere acceder a una entidad que no se encuentra en memoria.

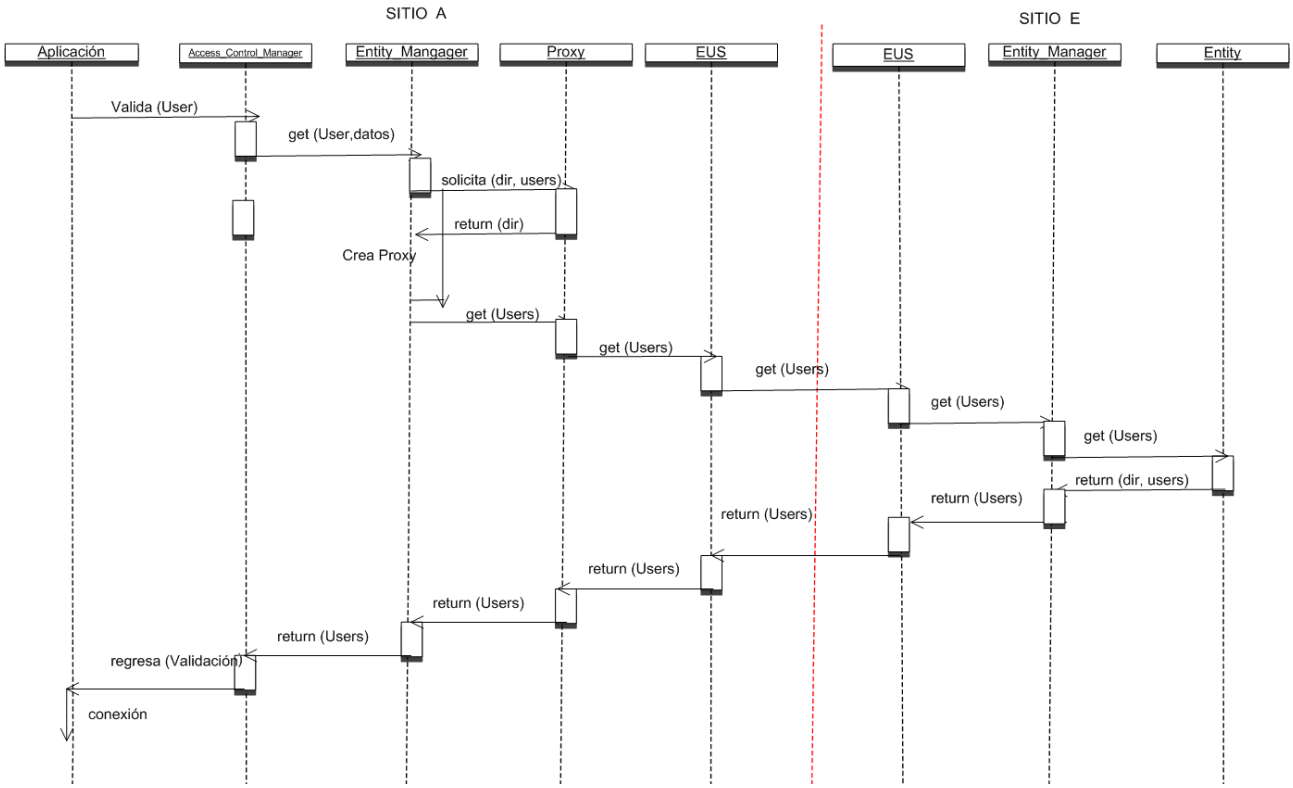

Figura 5.15: Proceso de conexión de un usuario

Salvar documento
Este proceso almacena una entidad en un archivo. La aplicación solicita al `Entity_Manager` via el API que lo guarde. El `Entity_Manager` notifica al EUS para que ejecute el proceso en los sitios remotos. El proceso se puede ver en la figura 5.17.

Salvar como un documento
Crea un nuevo documento, a partir de un documento existente, generando una copia. El proceso se puede observar en la figura 5.18. La operación es solicitada al `Entity_Manager`. El `Control_Access_Manager` y el EUS validan que el usuario tenga permisos de generar una copia y que el nombre de la entidad sea único.

Eliminar documento
Este proceso elimina una entidad existente y se puede ver en la figura 5.19. La aplicación solicita la operación al `Entity_Manager`. El `Control_Access_Manager` y el EUS validan que el usuario tenga permisos de eliminar el documento y que tenga el candado del documento para poder eliminarlo. Una vez eliminada la copia local, el EUS solicita a cada uno de los sitios que se eliminen las réplicas.

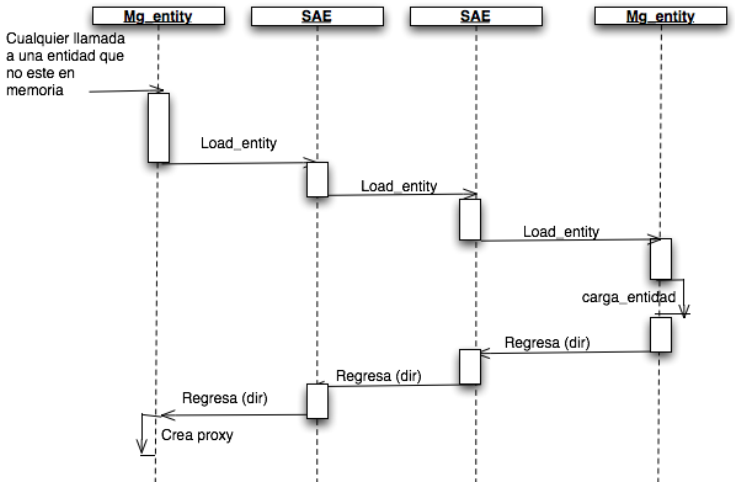

Figura 5.16: Proceso que permite subir un documento a memoria

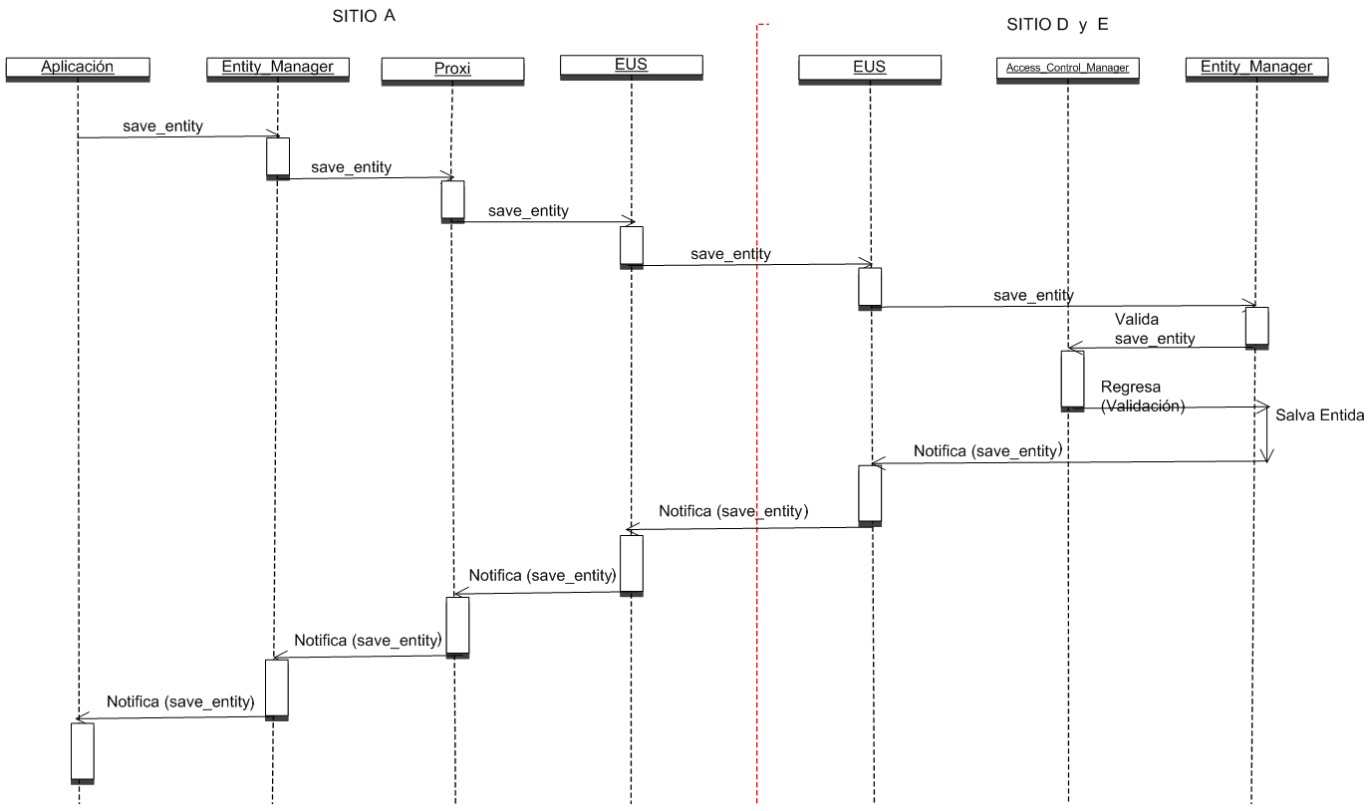

Figura 5.17: Proceso que permite salvar un documento en un archivo

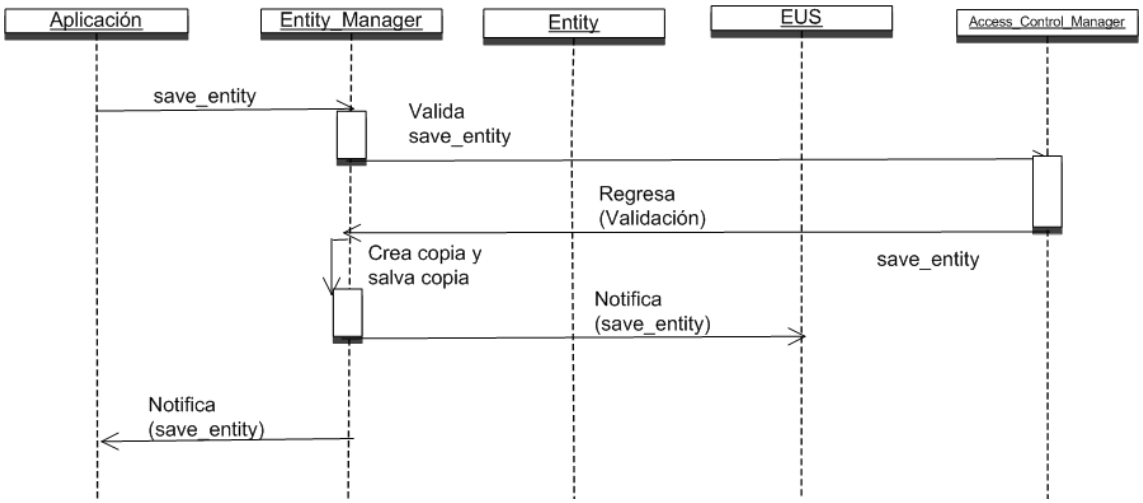

Figura 5.18: Proceso que permite realizar una copia de un documento

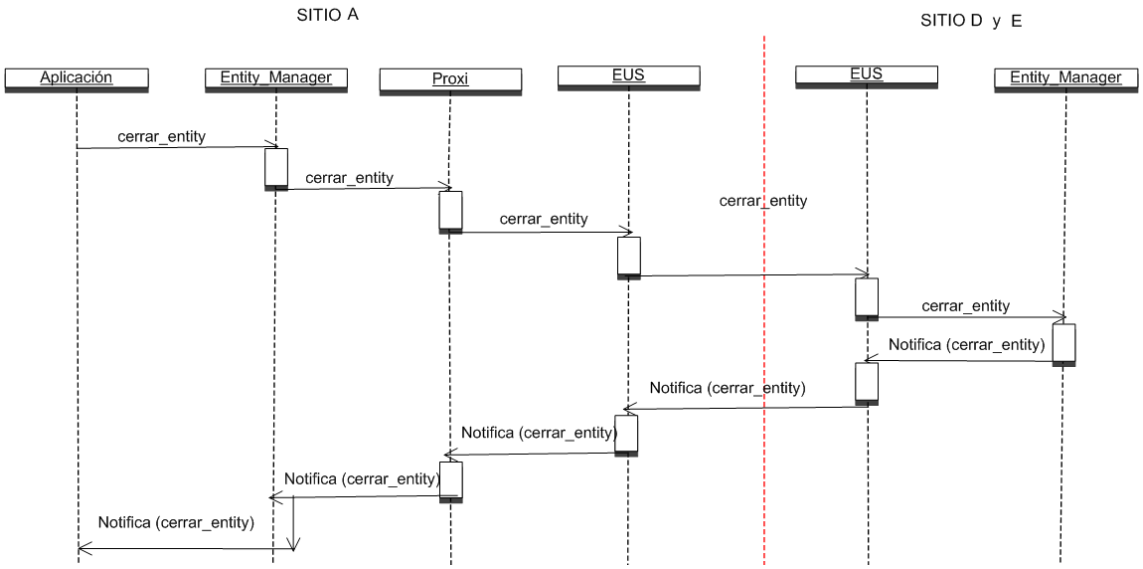

Figura 5.19: Proceso que permite eliminar de un documento

Solicitar candado

Antes de realizar una operación sobre uno o más fragmentos, se debe bloquear la operación para que nadie más modifique el/los fragmento(s) que se van a manipular. La aplicación solicita modificar el campo del `pol_cc` a "bloqueado". El `Control_Access_Manager` valida que el usuario tenga permisos de lectura en los fragmentos. El EUS valida que se pueda bloquear el fragmento. (que tenga la coordinación localmente y este disponible). Si la coordinación se encuentra en una réplica remota, el EUS solicita el fragmento remoto, junto con su coordinacón. Una vez bloqueado el fragmento

la aplicación puede solicitar operaciones en éste y el EUS notifica a los otros sitios donde hay réplicas que la coordinación del fragmento cambio de lugar. Este proceso se puede ver en la figura 5.20.

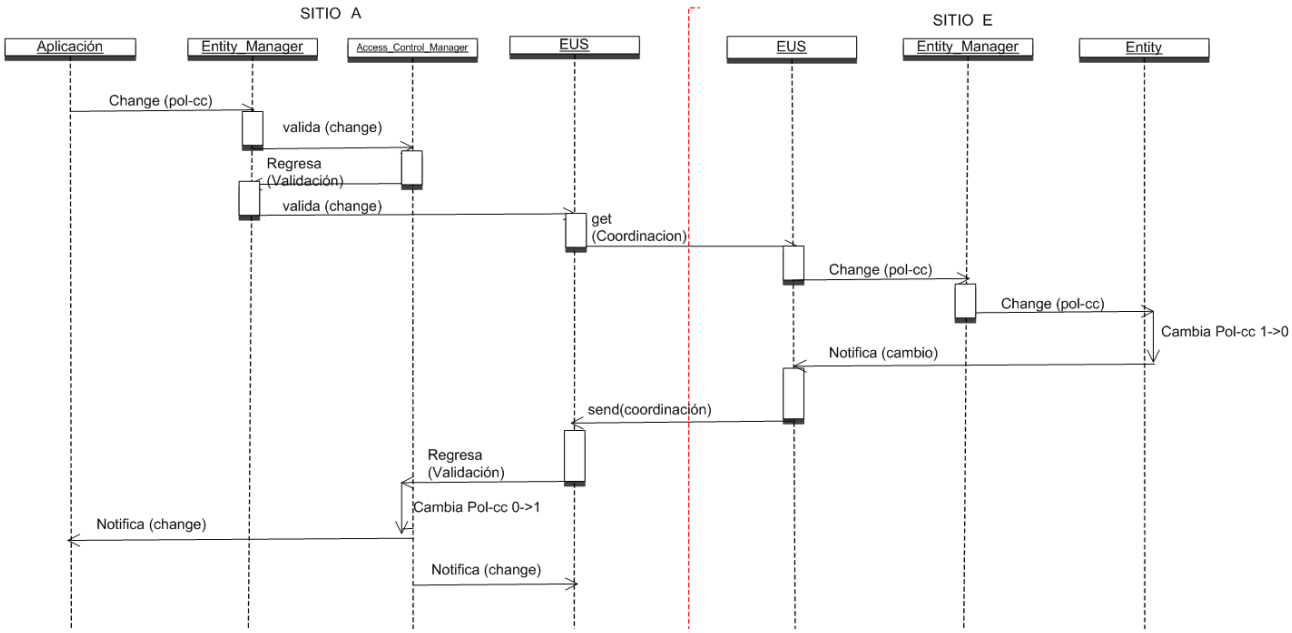

Figura 5.20: Proceso mediante el cual se obtiene el candado de un fragmento

Operaciones en los fragmentos

Las operaciones en los fragmentos de la entidad, siguen la misma secuencia, la cual se puede ver en la figura y consiste de los siguientes pasos: 5.21.

1. La operación es solicitada al `Entity_Manager`.

2. El `Control_Access_Manager` valida que el usuario tenga el permiso de realizar la operación.

3. El EUS valida que el usuario tenga el bloqueo del fragmento(s) a modificar.

4. La operación se realiza en la entidad y se verifica el orden de los fragmentos en la entidad.

5. Se notifica al EUS para que notifique a los sitios donde existen réplica y a la aplicación local, que la operación fue exitosa.

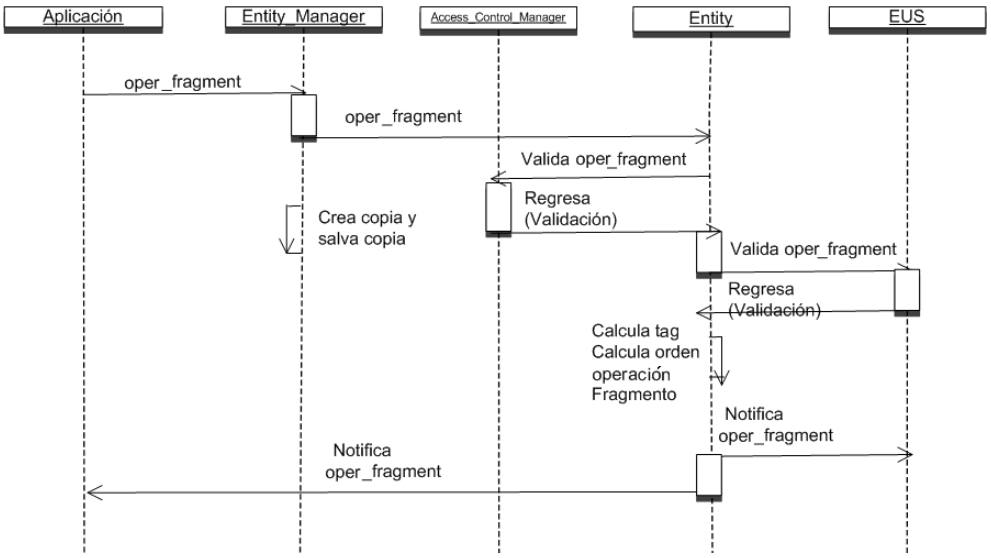

Figura 5.21: Proceso que permite realizar una operación en un(os) fragmento(s) de una entidad

Las operaciones que se pueden realizar a los fragmentos de una entidad tipo documento, son las siguientes:

- **Agregar un fragmento**. Permite a la aplicación agregar un nuevo fragmento a una entidad ya sea al final o entre dos fragmentos.
- **Modificar un fragmento** Crea una nueva versión de un fragmento. Esta operación no requiere que se verifique el orden de los fragmentos.
- **Borrar un fragmento**. Elimina un fragmento de una entidad y verifica el orden de los fragmentos después de que el fragmento fue eliminado.
- **Unir dos fragmentos**. Esta operación une dos framentos. Una vez unidos verifica el orden de los fragmentos en la entidad.
- **Dividir un fragmento en dos**. Divide un fragmento en dos a partir de una pocisión establecida por el usuario. Una vez unidos verifica el orden de los fragmentos en la entidad.

Obtener fragmento
Regresa el fragmento para lectura. La aplicación solicita al `Entity_Manager` el contenido del fragmento. El `Control_Access_Manager` valida que el usuario tenga permiso de lectura y el fragmento es enviado a la aplicación via el API.

5.3 Agendas colaborativas

Como se vió en la subsección 2.2.3, las agendas colaborativas permiten a los usuarios planificar y administrar sus actividades. Una agenda cooperativa, a diferencia de las agendas tradicionales, permite a los usuarios establecer tareas en común mediante citas. Debido a que el proceso de citas se da de forma asíncrona, un usuario establece la cita y debe ser confirmada por los participantes. Una vez aceptada por todos o la mayoría de los involucrados, la cita queda concertada.
Los usuarios de este tipo de aplicaciones pueden tener actividades o ser miembros de diferentes grupos de trabajo. Las actividades de un usuario pueden ser cooperativas o individuales, dependiendo de su naturaleza, de tal forma que habrá actividades que no son compartidas como las citas personales y habrá otras que se comparten con grupos de trabajo o con colaboradores. Usualmente el espacio de trabajo de cada usuario está dividido en dos espacios de citas bien separados: el privado y el compartido con el grupo de trabajo. Además cualquier usuario puede realizar una invitación, *i.e.*puede dar de alta una cita; la cual será validada por los implicados. Algunos usuarios pueden visualizar parcialmente la agenda de otros.
Ejemplos de agendas colaborativas en el mercado son: son Team Integrator [43], Carthago [44], Horde [49], Outlook [51], MAPilab [40], *etc*.
Básicamente las agendas colaborativas deben contar con dos tipos de entidades compartidas: entidades de control de acceso y agendas. Empezaremos analizando las entidades de las agendas, para que sea más sencillo la explicación de las entidades de control de acceso.

5.3.1 Entidades de las Agendas

La arquitectura propuesta para las agendas es replicada, como vemos en figura 5.22, ya que los usuarios requieren tener a la mano su agenda todo el tiempo. Se propuso utilizar un servidor central que permite coordinar las citas con otros usuarios y guardar las actualizaciones de los usuarios, mientras están desconectados, para que otros las consulten. En ocasiones el usuario desea ver las actividades de otros y se conecta mediante un *proxy* como es el caso del sitio *C* de la figura 5.22. También se puede conectar mediante un *proxy* debido a que no es un sitio donde se conecte usualmente o debido a que el dispositivo no cuenta con la capacidad de almacenamiento necesaria para realizar una réplica de la aplicación. El sitio de coordinación es el sitio *E*, pero se puede tener un respaldo en *D* por si éste falla.

Cuando se realiza un cambio en una entidad en alguno de los sitios de los usuarios, éste es distribuido al servidor central y al servidor de respaldo, como se ve en la figura 5.23 en el campo de `distribution`. También podemos ver que el control de acceso se lleva a cabo a nivel registro. Hay fragmentos que sólo pueden ser vistos por el usuario Victor como el fragmento 0_0. Otros fragmentos como el 1_0 pueden ser visualizados por los contactos del usuario.

El fragmento 1_0 de la figura 5.24 es una cita propuesta por el jefe. El jefe da de alta

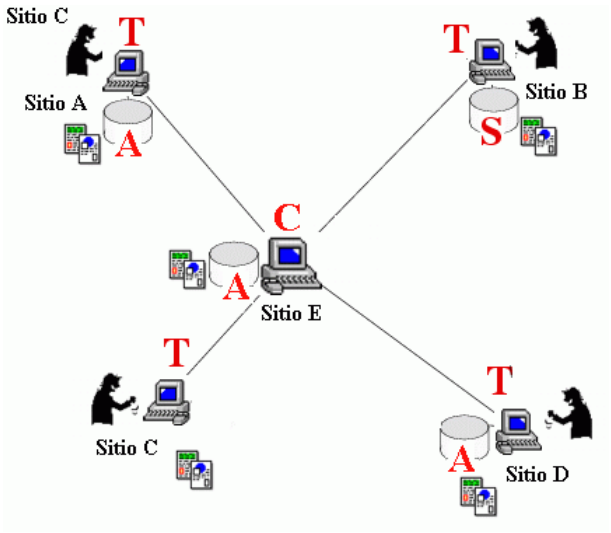

Figura 5.22: Arquitectura de la entidade tipo *agenda*

Data Structure								
Name: Agenda_Jefe				Id_replica: 1				
Type: Agenda				Control_Access: Users				
Distribution: Site D, Site E				Comunication: TCP				
Cons: 3								
Concurrency_control: Vector de relojes lógicos								
Tag	Fecha/hrs	Lugar	Descripción	Participantes	Confirmada	Owner	Pol_access	Pol_cc
1_0	01/06/09 10:00	Sala B	Junta consejo	Jefe √ Victor √	√	Jefe	C_Jefe	0_0, 1_1
0_0	01/06/09 09:00	Deportivo	Clase Natación	Jefe √	√	Jefe	C_Jefe	0_0

Figura 5.23: Agenda de un usuario.

la cita en su agenda. Ésta es replicada en el servidor en el sitio E, donde se encuentran las réplicas de las agendas de todos los usuarios. El servidor central se encarga de la coordinación entre los diferentes usuarios. Cuando una cita es agregada la aplicación que se encuentra en el sitio E hace la cita en todas las agendas de los involucrados. La cita es replicada a los sitios donde se encuentra cada usuario. El usuario tiene que aceptar y validar la cita. Cuando la cita es validada por la mayoría de los participantes, la cita es concertada. La figure 5.25 muestra las réplicas en el servidor.

Como se vió en subsección 2.2.3 proponemos una política de control de concurrencia optimista basada en vectores de relojes lógicos. Un vector de relojes lógico nos permite identificar cuando ha habido 2 cambios sobre un mismo fragmento en sitios diferentes. Supongamos que el jefe decide cambiar la junta para las 11:00. Supongamos que no se lleva a cabo la actualización en el servidor y la secretaria ve que el lugar esta ocupado y decide asignar otra sala. Al realizar la modificación directamente en el servidor central

Data Structure

Name: Agenda_Victor	Id_replica: 1
Type: Agenda	Control_Access: Users
Distribution: Site D, Site E	Comunication: TCP
Cons: 3	
Concurrency_control: Vector de relojes lógicos	

Tag	Fecha/hrs	Lugar	Descripción	Participantes	Confirmada	Owner	Pol_access	Pol_cc
1_0	01/06/09 10:00	Sala A	Junta consejo	Jefe √, Victor √	√	Jefe	C_Jefe, P_Victor	1_0
0_0	01/06/09 10:00	Vips	Desayuno Susy	Victor √, Susy X	X	Victor	P_Victor	0_0

Figura 5.24: Agenda del Jefe

mediante un *proxy*, la política de control de concurrencia cambia de 1_0 a 0_0 (debido a que es el primer cambio hecho directamente en la réplica 0). Al sincronizar ambas réplicas una cambio de 1_0 a 1_1 y la otra de 1_0 a 0_0, por lo que hay una inconsistencia. Para resolver esta inconsistencia se puede recurrir a los históricos y determinar que las modificaciones fueron hechas en diferentes partes del fragmento. Para corregirla se puede aplicar una función de trasformación que nos permita integrar ambos cambios. Si la modificación es hecha en la misma parte del fragmento (*e.g.* ambos hubieran cambiado la hora), la aplicación debe notificar la incosistencia al usuario final para que éste determine cual fragmento es el correcto.

Los demás campos son similares a las entidades vistas anteriormente.

5.3.2 Entidades de control de acceso

Como vimos en la subsección anterior las agendas colaborativas deben contar con una lista de control de acceso de todos los usuarios, para que puedan proponer actividades a los demás. Como se puede ver en la entidad de *Users* que muestra la figura 5.26, los usuarios pueden realizar invitaciones, pero no las pueden visualizar, modificar o eliminar.

Un usuario, (*e.g.* Victor), debe tener uno o más espacios compartidos con otros. Como se puede ver en la figura 5.26 los usuarios con los que comparte su espacio pueden leer, modificar y/o borrar los fragmentos pertenecientes al espacio que Victor les autorizó. Un usuario (*i.e.* Victor) también debe contar con un espacio privado al cual sólo él tiene acceso.

Debido a que la aplicación es completamente replicada, los dispositivos de los usuarios deben tener una copia de las entidades de acceso para que les permita trabajar adecuadamente. Se puede utilizar las mismas entidades de control de acceso utilizadas para los editores de texto. Las entidades de tipo `contactos` pueden utilizar una política optimista, basada en vector de relojes lógicos, el cual permita a los usuarios modificarla desde sus diferentes dispositivos, aún cuando no se tenga conexión

Data Structure

Name: Agenda_Victor	Id_replica: 0
Type: Agenda	Control_Access: Users
Distribution: Site E, Site A	Comunication: TCP
Cons: 3	
Concurrency_control: Vector de relojes lógicos	

Tag	Fecha/hrs	Lugar	Descripción	Participantes	Confirmada	Owner	Pol_access	Pol_cc
1_0	01/06/09 10:00	Sala A	Junta consejo	Jefe √, Victor √	√	Jefe	C_Jefe, P_Victor	1_0
0_0	01/06/09 10:00	Vips	Desayuno Susy	Victor √, Susy X	X	Victor	P_Victor	0_0

Data Structure

Name: Agenda_Jefe	Id_replica: 0
Type: Agenda	Control_Access: Users
Distribution: Site E, Site B	Comunication: TCP
Cons: 3	
Concurrency_control: Vector de relojes lógicos	

Tag	Fecha/hrs	Lugar	Descripción	Participantes	Confirmada	Owner	Pol_access	Pol_cc
1_0	01/06/09 10:00	Sala B	Junta consejo	Jefe √ Victor √	√	Jefe	C_Jefe	0_0, 1_1
0_0	01/06/09 09:00	Deportivo	Clase Natación	Jefe √	√	Jefe	C_Jefe	0_0

Figura 5.25: Réplicas en el servidor de coordinación

a la red. Se puede utilizar una política optimista debido a que las entidades tipo `contactos` rara vez son actualizadas de manera concurrente ya que sólo las modifica el dueño de la agenda. Consecuentemente, las réplicas sólo están en los servidores y los diferentes dispositivos del usuario. La probabilidad de una inconsistencia es mínima y de ocurrir, sólo debe ser detectada por el sistema para que el usuario decida cual es la versión correcta, si es que no se puede resolver el conflicto por medio de funciones de transformación que incluyan ambos cambios.

5.3.3 Operaciones y procesos en las agendas

Conección de un usuario y sincronización

Cuando un usuario se conecta al servidor es necesario validar al usuario y sincronizar las entidades que se encuentran en el dispositivo del usuario, con las entidades del servidor (ver figura 5.27). Si existen inconsistencia se trata de resolver por medio de una función de transformación que integre ambos cambios. De no ser posible se notifica al usuario para que determine cual es la versión correcta.

Operaciones a los espacios de los usuarios

El usuario puede realizar varias operaciones en los espacios. Las operaciones a rea-

Data Structure	
Name: Users	Id_replica: 1
Type: Control de acceso	Control_Access: Admin
Distribution: Site E,A,B,D	Communication: TCP
Cons: 1	
Concurrency_control: Candados optimistas con coord por fragmento	

Tag	Usuario	Lec	Mod	Alta	Baja	Pol_cc	Internal
0_0	Admin	1	1	1	1	1	1
0_1	Victor	0	0	1	0	1	0
0_2	Jefe	0	0	1	0	1	0
1_0	Secre	0	0	1	0	1	0

Data Structure	
Name: C_Victor	Id_replica: 1
Type: Control de acceso	Control_Access: Admin
Distribution: Site E, D	Communication: TCP
Cons: 1	
Concurrency_control: Candados optimistas con coord por fragmento	

Tag	Usuario	Lec	Mod	Alta	Baja	Pol_cc	Internal
0_1	Victor	1	1	1	1	0	0
0_2	Jefe	1	1	1	1	0	0
1_0	Secre	1	1	1	1	1	0

Figura 5.26: Entidades de control de acceso: Users y Contactos de Victor

lizar son:

- **Crear un espacio**. El usuario crea una entidad tipo `contacto`.
- **Borrar espacio**. El usuario elimina una entidad tipo `contacto`.
- **Modificar espacio**. El usuario modifica los permisos de acceso a ciertas citas otorgados a otros usuarios. Las operaciones que puede realizar son:
  - *Alta de usuario*. El usuario autoriza que otro usuario tenga acceso a ciertas citas de su agenda. Puede otorgar permisos de lectura y/o escritura.
  - *Baja de usuario* El usuario elimina los permisos de acceso a sus citas otorgados a otro usuario.
  - *Modificación a los permisos otorgados a un usuario*. El usuario cambia los permisos de acceso a sus citas, otorgados a otro usuario.

Las operaciones en las entidades tipo `contacto` primero se realizan y validan en el sitio local por medio del `Control_Access_Manager` y el EUS. Una vez validadas se realiza la operación y se difunde la notificación a los sitios D y E para que sean actualizadas las réplicas. También se notifica a la aplicación que la operación fue exitosa.

Operaciones en las agendas

Cuando un usuario crea una cita se debe notificar a todos los participantes. Para ello, primero se realiza la operación en su dispositivo local. Posteriormente la operación debe ser difundida en todas las réplicas. Cuando se notifica al servidor donde se encuentra la coordinación, éste notifica a la aplicación que realiza la coordinación para que ingrese la cita en las entidades de todos los usuarios implicados. Del servidor de

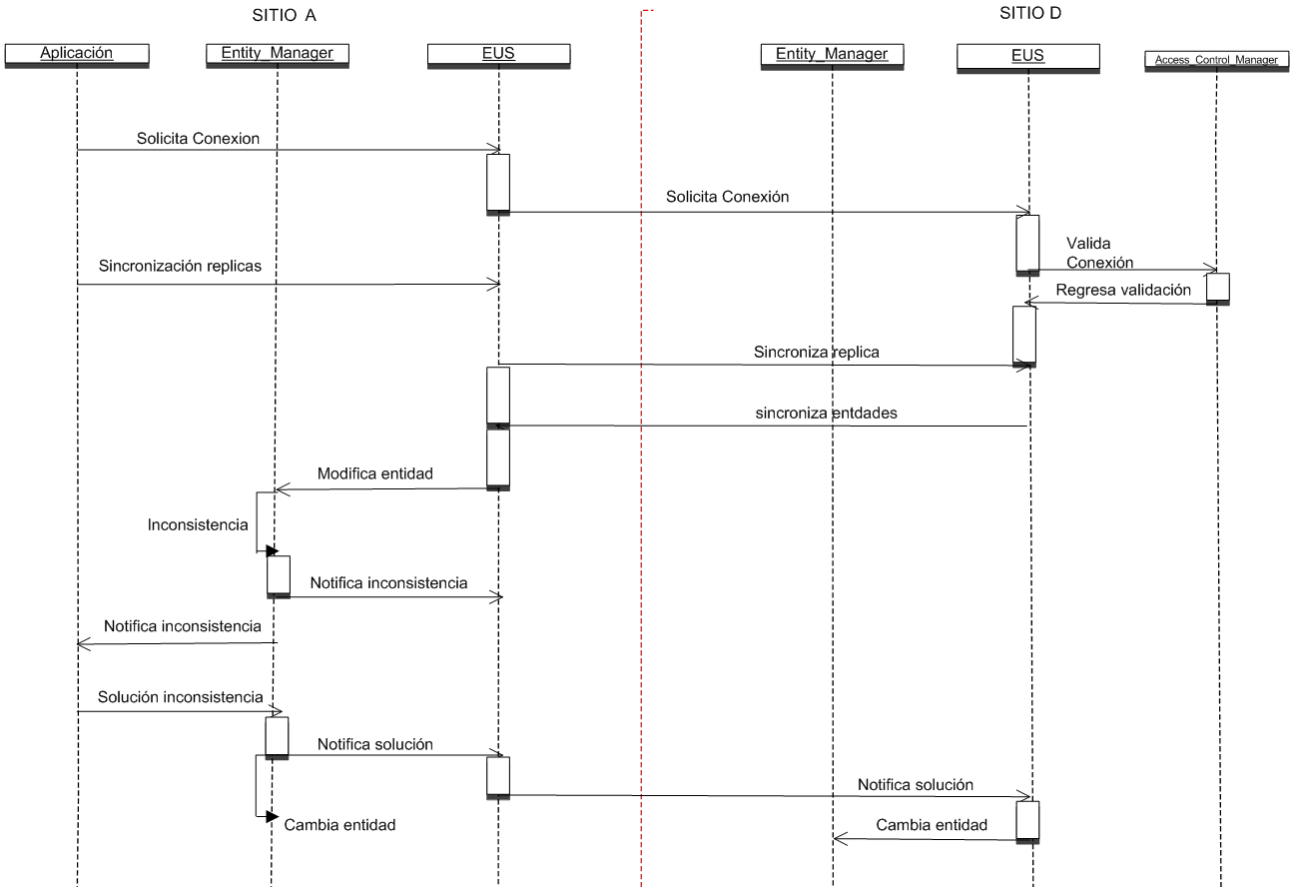

Figura 5.27: Proceso de conexión y sincronización de los datos de la agenda

colaboración las notificaciones son enviadas a las réplicas en los sitios de los participantes. Los participantes deben aceptar o no la cita y regresar la notificación al servidor de coordinación. Una vez que todos o la mayoría de los participantes aceptaron la cita, el servidor notifica a las réplicas de los participantes que la cita fue establecida. Los procesos de modificación y cancelación de cita, tienen un flujo similar. Este proceso se puede ver en la figura 5.29. El proceso está simplificado, debido a que la operación debe pasar por el API el `Entity_Manager` y el `Control_Access_Manager`.

Cabe mencionar que los procesos que se realizan por medio del EUS pueden quedar pendientes a causa de que en ese momento no se cuente con el acceso a algún sitio debido a las desconexiones de la red o la movilidad de los usuarios. Cuando esto sucede el proceso se detiene y se reanuda cuando se vuelve a tener acceso al sitio correspondiente. Las aplicaciones pueden requerir aún más procesos, la intención era mostrar el funcionamiento de la plataforma con diversas aplicaciones colaborativas.

En este capítulo se probó el modelo con 3 aplicaciones colaborativas distribuidas:

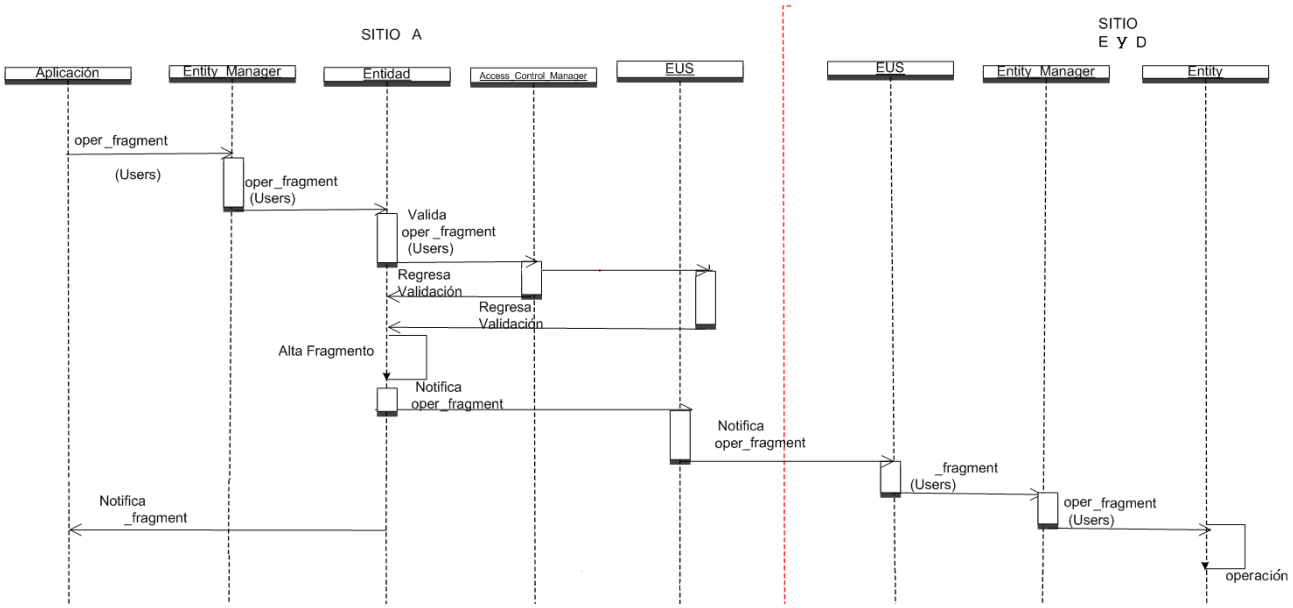

Figura 5.28: Proceso de una operación en una entidad de acceso

chat, editor de documentos y agenda colaborativa. Aunque se requiere validar con más tipos de aplicaciones, se pudo constatar que funciona adecuadamente para aplicaciones colaborativas distribuidas.
Las entidades replicadas compartidas tienen un comportamiento similar, lo que permite generalizar el proceso con pequeñas variantes, facilitando su diseño e implementación. También permite reutilizar código, ya que en ocasiones las políticas de distribución, control de acceso y control de concurrencia, son muy similares, sino las mismas.

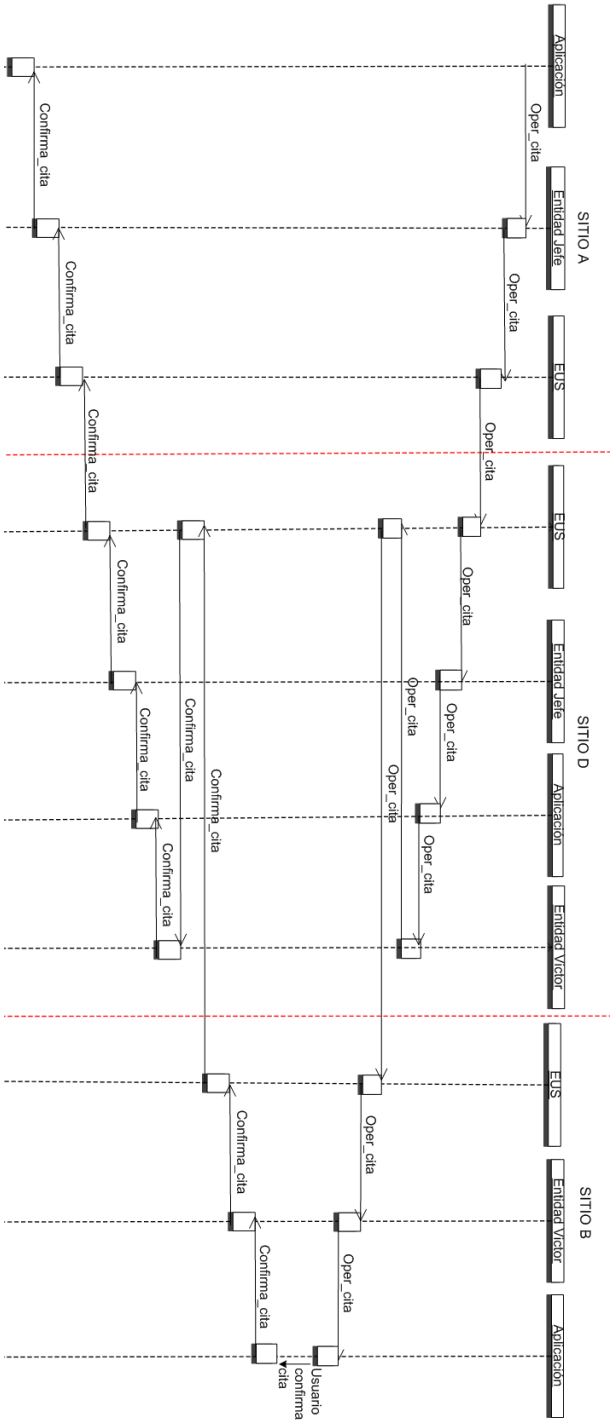

Figura 5.29: Proceso de operaciones de citas en las entidades tipo agenda

Capítulo 6

Implementación y pruebas del modelo

Con la finalidad de probar el diseño propuesto en el capítulo cuarto y su adecuación al objetivo definido, se implementó un prototipo, usando el lenguaje de programación Ruby.

En este capítulo describimos las características principales de su implementación y las pruebas realizadas. Se introducen las funcionalidades que se utilizaron, para la implementación del modelo, así como algunas funciones del *API* de reflexión del lenguaje.

Para probar relativamente, la plataforma propuesta, se desarrolló un *chat room* y una aplicación que prueben los diferentes métodos de las entidades y la interacción entre los módulos funcionales de la plataforma. Se presentan los distintos tipos de entidades replicadas definidas y sus instancias. También se introduce brevemente las políticas de control de acceso y control de concurrencia utilizadas para probar la comunicación entre los elementos de la plataforma, la flexibilidad al utilizar diferentes políticas de control de acceso y la interfaz con el *EUS* (*Entity Update System*).

Para verificar el actualizador de versiones de software se probaron dos casos especiales: 1) se agregó un nuevo tipo de entidad y 2) se modificó una instancia con este nuevo tipo.

6.1 Beneficios de Ruby y su *API* de reflexión

Aunque la plataforma puede desarrollarse en cualquier lenguaje de programación, fue desarrollada utilizando el lenguaje de programación orientado a objetos *Ruby* versión 1.8.6. Éste nos ofrece varias ventajas, las cuales facilitan la programación de la plataforma:

- *Ruby* es un lenguaje de programación orientado a objetos, haciendo la implementación de las entidades más manejable.
- *Ruby* es un lenguaje de programación dinámicamente tipado, dando flexibilidad y facilitando las actualizaciones dinámicas, al permitir modificar fácilmente la

clase a la que pertenece un objeto. Así, al ser un lenguaje interpretado no es necesario modificar *bycotes*, binarios o recompilar, aunque es un poco más lento que lenguaje compilado.

- Al ser un lenguaje dinámicamente tipado, no es necesario declarar el tipo de los objetos (*e.g. string, char, float, etc.*), adaptándose en tiempo de ejecución,
- las clases en *Ruby* son objetos por lo que son fáciles de modificar en tiempo de ejecución. Se pueden clonar, agregar, modificar y descargar las clases de ser necesario. Esta característica Facilita la implementación de las actualizaciones dinámicas de software,
- la implementación de los patrones de diseño son más sencillos que en otros lenguajes debido a la flexibilidad de *Ruby*,
- *Ruby* cuenta con un *API* de reflexión poderosa, permitiendo examinar el estado de los objetos. Ésta facilita la tranformación de los objetos para el cambio de versión de las clases por medio de las actualizaciones dinámicas de software.

A continuación se presentan algunos constructores que facilitaron la implementación de la plataforma.

attr_accessor y attr_reader

Mediante estos comandos, *Ruby* genera los métodos para acceder a los atributos de la clase para lectura o escritura (equivalentes al `set` y al `get`). A continuación ejemplificamos como Ruby proporciona la funcionalidad del `get` y la manera de llamarlo. Supongamos que tenemos la siguiente clase:

```
class Class1
    attr_accessor :atributo1
end
```

Ruby agrega el siguiente método equivalente a una operación `get`:

```
def atributo1=(atributo1)
     \@atributo1=atributo1 \#se asigna el valor al atributo
end
```

La expresión para utilizar esta funcionalidad de acceso a la variable `atributo1` de la instancia `obj1` de la clase `Class1` es de la forma:

```
obj1.atributo1=valor
```

De esta manera todos los `get` y `set` definidos en el diseño, los cambiamos por las instrucciones `attr_accessor` y `attr_reader` dependiendo de si se requieren las operaciones `get` y `set`, o solamente el `set`.

`eval`

Nos permite ejecutar una cadena de caracteres como una parte de código. Así, podemos enviar los métodos o atributos como parámetros de tipo *string*, formar la instrucción y finalmente evaluarla.

Por ejemplo, el módulo de control de acceso tiene como parámetros la política de acceso a las entidades, la tabla con los usuarios válidos y una condición sobre las variables de ambiente. Cuando recibe una evaluación, recibe estos parámetros y manda llamar a la política correspondiente formando la instrucción y evaluándola:

```
respuesta=eval("#{politica}.(#{tabla}, #{usuario},
#{condicion})")
```

En tiempo de ejecución, el módulo de control de acceso evalua la instrucción con los siguientes valores:

```
respuesta=eval(Acceso.(“users”, “Ernesto”, nil)
```

Esta instrucción manda llamar a la política de control de acceso `Acceso` y valida que el usuario “Ernesto” se encuentre de alta en la tabla `users`, sin ninguna condición adicional.

bindings

Esta función permite redefinir el alcance de los atributos de un objeto (es decir se rompe el encapsulamiento del objeto y es posible acceder a las variables privadas de un objeto). Si se modifica el alcance y se accesa a las variables privadas de los datos, se puede obtener el estado de los objetos. Así, al llevar a cabo un cambio de versión de una clase, es necesario actualizar los objetos instanciados con la antigua versión. Para ello, requerimos obtener su estado, transformarlo y adaptarlo siguiendo la estructura de la nueva versión de la clase.

Por ejemplo, supongamos que tenemos un tipo de entidad para transmitir video. Originalmente se estaba almancenando toda la información por lo que no se utilizaba una pila, sino que se almacenaba todo lo transmitido. Por cuestiones de espacio ya no se va a almacenar el video, sino que sólo se dará el servicio de retransmisión, por lo que ahora la entidad que almacena el video será una pila circular. El nuevo tipo de entidad en vez de calcular el `cons`; que controla el número del siguiente registro a dar de alta (ver los atributos de la entidad genérica, subsección 4.2), con un máximo infinito, ahora se reinicializará al ser igual al atributo *Pila*, que es el tamaño máximo de la cola circular. Las nuevas entidades utilizarán el nuevo tipo sin ningún problema, pero es necesario actualizar las entidades instanciadas con el tipo anterior. Entre otras cosas, es necesario ver el estado del atributo *cons* (consecutivo) de la entidad, transformarlo y pasarlo al nuevo objeto. Supongamos que tenemos el objeto *video1* que es una entidad de tipo *Video* y queremos transformarlo al tipo *Video_v2*. El código que permite transformar el *cons* es el siguiente:

```
Class Object
def bindings
      binding #modificamos el alcance
end
cons_anterior= eval("@cons", video1) #obtenemos el estado de
cons, valor del atributo privado
if cons_anterior<Pila
    cons_transformado=cons_anterior
else
     cons_transformado=0
end

#si es menor lo conservamos, sino lo reinicializamos

video1=Video_v2.new
video1.cons=cons_transformado
```

Cabe mencionar que de manera similar hay que transformar y pasar cada una de las variables de las instancias del tipo de entidad.

method_missing

Cuando se llama un método de un objeto que no se encuentra en una clase, ruby manda un mensaje de error llamando al método `method_missing` de la clase `Objeto`. Es posible redireccionar las llamadas hechas a un objeto hacia otro redefiniéndolo. Esto nos permite que los *proxies* pueden manejar cualquier tipo de clase, independientemente de los métodos que tenga el objeto. Para redireccionar las llamadas por medio del `method_missing` se define en los proxies:

```
def method_missing(name, *args)
@objeto.send(name, *args) #se lo envia al objeto que
representa el proxy
end
```

El método `method_missing` de la clase `Objto` tiene como argumentos el nombre del método (`name`) y los arguentos (`*args`). Se redefine este método para que cuando no exista el método en el proxy, redireccione la llamada al objeto que representa.

marshal

La función `marshal` hace el aplanado de un objeto para poder guardarlo en un archivo binario.

```
File.open(archivo, "w") do |file|
       Marshal.dump(objeto, file)
end
```

Para leer un objeto almacenado en un archivo:

```
File.open(archivo) do |file|
       objeto=Marshal.load(file)
end
```

6.2 Implementación de la plataforma

Básicamente la plataforma se implementó de acuerdo al diseño, con pequeñas variantes que a continuación se describen, debido a las ventajas que proporciona el lenguaje de programación Ruby.
Como se vió en el capítulo cuarto (diseño de una plataforma flexible y extensible), los módulos que conforman la plataforma propuesta son:

- **Entity_Manager**: crea, almacena, borra y modifica las instancias de las entidades compartidas distribuidas. También controla el acceso local a sus métodos.
- **Access_Control_Manager**: valida las operaciones realizadas por los usuarios en los distintos elementos de la plataforma de acuerdo su rol de usuario.
- **api**: permite la comunicación de la aplicación colaborativa local con la plataforma.
- **EUS (*Entity Update System*)** actualiza y sincroniza las entidades replicadas entre los diferentes sitios de colaboración.
- **DAUS (*Dynamical Application Updating System*)** ejecuta las actualizaciones dinámicas de software.

Las operaciones solicitadas por la aplicación local de manera genérica, siguen el diagrama de flujo de la figura 6.1.

1. La aplicación solicita al api una operación en una entidad.
2. El API solicita al Entity_Manager que ejecute la operación.
3. El Entity_Manager manda llamar el método en la entidad correspondiente.
4. La entidad solicita al EUS y al Control_Access_Manager que validen la operación.
5. Se realiza la operación en la entidad.
6. Se notifica al EUS para que sincronice las réplicas en los sitios de colaboración y al API para que notifique a la aplicación el resultado de la operación.

Para establecer la comunicación se implementaron las interfaces de cada módulo definidas en el capítulo cuarto.

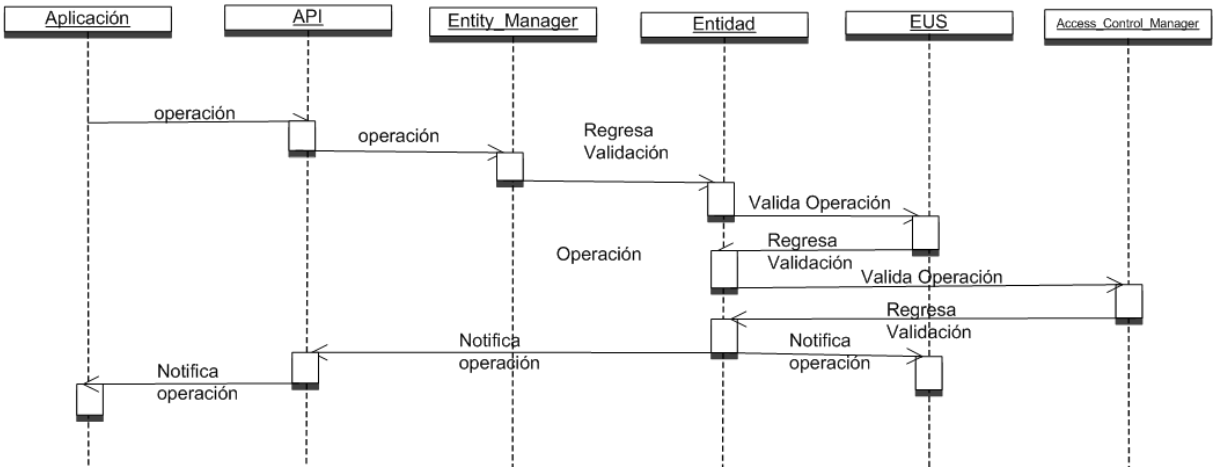

Figura 6.1: Flujo de información entre los módulos funcionales

6.2.1 Plataforma mínima

La plataforma mínima integra los diferentes elementos. Para facilitar la comunicación entre sus elementos, la plataforma mínima integró algunas partes de otros módulos y así:

- Implementa parte del API, para facilitar las llamadas entre las aplicaciones y la plataforma, al no tener que especificar la dirección del API.
- Integra las tablas de clases y objetos del DAUS (*Dynamical Application Update System*), como variables globales de la plataforma, con la intención de ser utilizadas desde cualquier módulo.

6.2.2 Control de acceso

El control de acceso recibe la instrucción a validar, junto con los parámetros con los que se realiza la evaluación. Carga la política (si no fue cargada anteriormente) y le pasa los datos necesarios para que apruebe o no la operación. Utiliza estrategias para direccionar hacia diferentes políticas de control de acceso. La interfaz del módulo de control de acceso es la siguiente:

```
respuesta=Control_Access_Manager(politica, operacion, tabla,
          usuario, condicion)
```

Los parámetros de la interfaz son los siguiente:

- `política`: es la política de control de acceso con el que se evalua la operación.
- `operación`: es la operación que se requiere validar *e.g.* alta de fragmento, baja de fragmento, salvar entidad, *etc*.

- `tabla`: en caso de que la política utilice una lista de usuarios, es la entidad que contiene los nombres de los usuarios validos para realizar las operaciones.
- `usuario`: nombre del usuario que desea realizar la operación y por lo tanto se requiere evaluar que tenga el permiso de realizar la operación solicitada.
- `condición`: se utiliza cuando se requiere evaluar la operación de acuerdo a alguna variable de ambiente (*e.g.* durante un periodo de tiempo).

La política es la politica de control de acceso con el que se evalua la operación.

Para no cargar las políticas varias veces, los objetos son almacenados en una tabla *hash*.

6.2.3 Administrador de entidades

El administrador de entidades, al igual que los otros módulos, utiliza las instrucciones `attr_accessor` y `attr_reader` en vez de los métodos `get` y `set`.

Para guardar las entidades utiliza la función `marshal` de *Ruby*, presentada anteriormente.

El administrador de entidades implementa el módulo *observadores* en donde almacenan las direcciones en memoria de los otros componentes de la plataforma que realizan las validaciones y requieren conocer las notificaciones cuando se ejecuta una operación en alguna entidad.

Las operaciones del administrador de entidades son validadas por el control de acceso. Para verificar su funcionamiento se utilizó una política de control de acceso estatica (*i.e.* lista de usuarios) y utiliza la entidad *users* (la cual se explica posteriormente). Las operaciones realizadas son las presentadas en el diagrama de clases de la plataforma flexible y extensible (figura 4.5).

6.2.4 *Proxy* de entidades remotas

Los *proxies* son instanciados por el administrador de entidades cuando se realiza una llamada a una entidad que no se encuentra localmente. Cuando se realiza una operación mediante un proxy, se sigue el proceso que muestra la figura 6.2.

- El EUS obtiene la dirección de la entidad a ser accesada via el *proxy*.
- El *proxy* lleva el mismo nombre de la entidad replicada en el sitio remoto, por lo que se mantiene en memoria y opera como la entidad real.
- Una vez creado, para todos los objetos locales es una entidad más, pero en vez de resolver las llamadas, las direcciona hacia el EUS.
- El EUS regresa el resultado de la operación al *proxy*

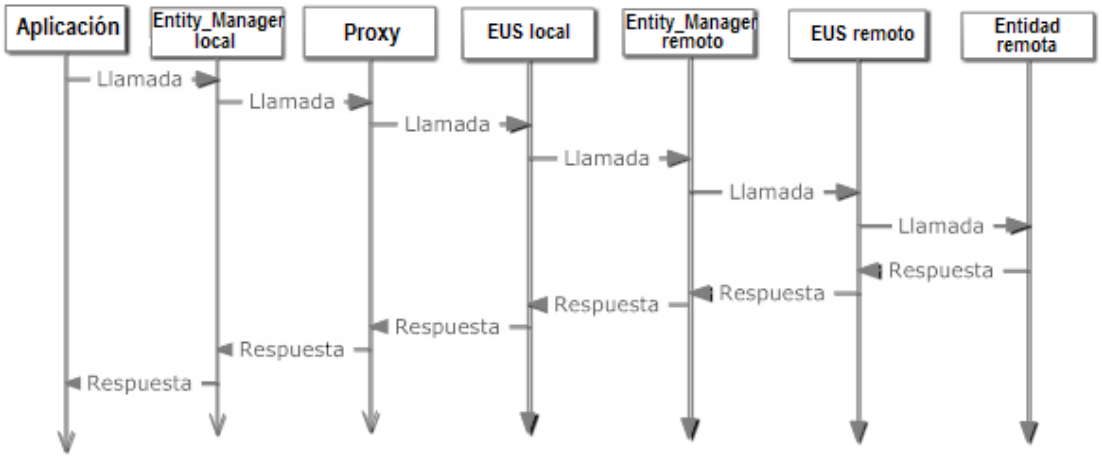

Figura 6.2: Procesamiento de una llamada remota via un proxy

- El *proxy* le entrega el resultado al cliente que hizó la llamada, como lo haría una entidad real.

A diferencia del diseño, en vez de los métodos `set` y `get`, se utilizó `attr_accessor`.

6.2.5 API (*Application Program Interface*)

El API recibe las solicitudes de modificaciones a las entidades que realizan las aplicaciones. Esta parte fue implementada en la plataforma mínima, con la intención de simplificar las llamadas.

También recibe las notificiones de las modificaciones a las entidades y las envía a las aplicaciones. Para recibirlas utiliza el módulo `observadores`. Las operaciones que las aplicaciones pueden realizar en las entidades por medio del API son las presentadas en el diagrama de clases de la plataforma flexible y extensible (figura 4.5).

6.2.6 DAUS (*Dinamical Application Updating System*)

El DAUS permite realizar actualizaciones dinámicas. En general se programó en base al diseño aunque las tablas de objetos y clases fueron implementadas en la plataforma mínima como variables globales. Las operaciones que se implementaron para poder realizar actualizaciones dinámicas de software se pueden ver en el diagrama de clases de la plataforma flexible y extensible (ver figura 4.5).

6.2.7 EUS (*Entity Update System*)

El EUS se realiza la validación de las operaciones en las entidades de acuerdo a la política de control de concurrencia asociada a la entidad. También recibe las notificaciones de la operaciones hechas en las entidades, con la finalidad de sincronizar las réplicas.
Como se comentó anteriormente, el EUS fue desarrollado por el Ing. Miguel Navarro Dávila. Sin embargo para porder llevar a cabo algunas pruebas, se implementaron algunas de sus partes.

Comunicación

Se desarrolló parte del módulo de comunicación utilizando TCP como protocolo de transporte y un protocolo propietario a nivel aplicación para comunicar entidades remotas.

MU (Marshal/Unmarshal)

Se creó la parte de conversión de los mensajes en llamadas a métodos de los diferentes elementos de la plataforma (administrador de entidades, *DAUS* y control de acceso), con la finalidad de ver las notificaciones que permiten al EUS sincronizar las entidades réplicadas compartidas.

Interfaz entre el Entity_Manager y el EUS

Se desarrollo la interfaz con la intención de verificar el paso adecuado de los mensajes entre los demás elementos de la plataforma y el *EUS*. De esta manera se simula una validación por parte del *EUS* y las notificaciones que recibe para sincronizar las entidades réplicadas. Cabe mencionar que estas funcionalidades del EUS no se realizarán hasta la integración del EUS a la plataforma flexible y extensible. La interfaz para la validación de las operaciones en las entidades replicadas es la siguiente:

```
respuesta = EUS.valida(operacion, politica, pol_cc)
```

Los parámetros de la interfaz son los siguientes:

- `operación`: es la operación en la entidad compartida que se desea validar (*e.g.* modificar fragmento, eliminar entidad, actualizar una entidad, *etc.*).
- `política`: es la política de control de concurrencia con la cual se evalua la operación (*e.g.* candados, optimista, vector de relojes lógicos, etc.)
- `pol_cc`: Parámetro que permite evaluar la política de control de concurrencia. Por ejemplo, si la política de control de concurrencia está basada en candados, indica si la entidad tiene la coordinación del fragmento en la réplica local, si el candado está disponible o está bloqueado.

La interfaz para que el EUS reciba las notificaciones es:

```
EUS.notifica(operacion(*args), dist)
```

- `operación`: es la operación en la entidad compartida que el EUS tiene que sincronizar de acuerdo a la notificación (*e.g.* eliminar fragmento, crear entidad, unir fragmentos, *etc.*).
- `*args`: son los parámetros de la operación a realizar para poder sincronizar las réplicas de la entidad (*e.g.* nombre de la instancia del tipo de entidad, usuario, fragmento, etc.)
- `dist`: direcciones de los sitios en donde se encuentran las réplicas que hay que sincronizar.

6.3 Entidad genérica

La implementación de la entidad genérica se apegó al diseño excepto por algunas modificaciones menores debido a las herramientas que proporciona *Ruby*.

- No se implementó el campo de `size`, debido a que *Ruby* administra la memoria, por lo que no es necesario calcular donde empieza y donde termina el fragmento.
- En vez de implementar los métodos `get` y `set` para accesar a los atributos utilizamos `attr_accessor` y `attr_reader` explicados anteriormente.
- Para implementar los observadores y validadores se utilizó el módulo de observadores implementado como parte de la plataforma mínima.

Las entidades son almacenadas en archivos binarios utilizando la función `marshal` de *Ruby*. Para su almacenamiento conservan el nombre de la instancia del tipo de entidad con extensión `.ent` (*e.g.* `texto2.ent, chat.ent, admin.ent`, *etc*).

6.4 Entidades y políticas implementadas

Para probar la plataforma se crearon diferentes entidades y políticas, que permitieron verificar su correcto funcionamiento.

6.4.1 Tipos de entidades

Para implementar las aplicaciones propuestas, se implementaron tres tipos de entidades: `Archivo, Chat, Acceso`.

Archivo

Las entidades tipo archivo nos permiten definir un documento de texto compartido. Como cualquier tipo de entidad hereda sus atributos y métodos a partir de la entidad genérica. Debido a que es un archivo ordenado, es necesario controlar el orden de

los fragmentos, por lo que se tuvieron que redefinir los métodos de `ctl_order_add` (ajusta el orden cuando se agrega un fragmento) y `ctl_order_del` (ajusta el orden cuando un fragmento es eliminado), que permiten manejar un archivo ordenado. Para accesar más fácilmente a las entidades, el tipo de entidad `Archivo` se creó como una estructura doblemente ligada, *i.e.* en el campo de `orden` del fragmento guardamos la etiqueta del que sigue y en el *hash* `ordenes` guardamos la etiqueta del anterior para agilizar las búsquedas.

El atributo de `Access_Control_Manager` se definió con una política basada en listas de acceso, la cual es descrita más adelante.

La política de control de concurrencia utilizada es el candado, pero no se implementó dicha política, sólo se validó la entrega correcta de los mensajes al *EUS*.

El protocolo de comunicación utilizado es TCP, que es el que se propuso en la sección 2.2 para este tipo de aplicaciones y se puede ver en la tabla 2.2 Soluciones propuesta para la replicación de los datos.

Para probar la entidad se crearon, salvaron y borraron múltiples archivos de texto, a los cuales se les agregaron, borraron y modificaron fragmentos.

Chat

Los mensajes en los chats pueden almacenarse en colas circulares por lo que es necesario redefinir el cálculo del consecutivo, reinicializándose al alcanzar el tamaño de la pila, definido en 50.

En una cola de mensajes, normalmente, los fragmentos no se pueden modificar y se borran conforme se incrementan los mensajes, por lo que se redefinieron los métodos `del_frag`, `del`, `get_frag`, `change_frag`, `change`, `split_frag`, `split`, `join_frag` y `join` inhibiendo su funcionamiento.

El protocolo de comunicación utilizado es TCP, que es el propuesto en la sección 2.2 y se puede ver en la tabla 2.2 Soluciones propuestas para la replicación de los datos.

La política de control de acceso para agregar nuevos fragmentos es de acceso total (*i.e.* autoriza cualquier operación válida dentro de la entidad a los usuarios que se encuentran conectados a la aplicación), debido a que sólo es necesario validar que sea un usuario de la aplicación, lo cual se hace en el momento en que se conecta y se crea la entidad y la única operación válida es agregar mensajes a la entidad del tipo "cola de mensajes".

La política de control de concurrencia utilizada es una política optimista sin corrección de inconsistencias, es decir, autoriza una ejecución concurrente distribuida de las operaciones sin requerir aplicar ninguna corrección.

Esta entidad fué implementada para poder probar una aplicación cooperativa distribuida: un *chat room*, el cual se explica en la sección 6.5.

Acceso

Las entidades de acceso establecen asociaciones entre usuarios (o grupos de usuarios) y los derechos de acceso a la entidad considerada. No son listas ordenadas, pero en el hash mantenemos un orden alfabético de los usuarios para encontrarlos más rápi-

damente. Por esta razón redefinimos las funciones ctl_order_add (mantiene el orden al agregar un registro), y ctl_order_del (mantiene el orden después de borrar un registro) que nos permiten mantener un listado de los usuarios ordenado alfabéticamente. Los registros de la entidad (*i.e.* fragmentos) son por usuario o grupo de usuarios. En el caso de los grupos de usuarios, el registro direcciona hacia otra entidad, por lo que el atributo de *internal* es 0.

También redefinimos la función buscar (en el caso de la entidad genérica no hacia nada). Cuando se requiere saber si un usuario tiene permisos o no de ejecutar una acción dada, el algoritmo primero busca en la entidad, si no lo encuentra busca en los grupos de usuarios, por lo que debe solicitar al manejador de entidades buscar en la entidad a la que hace referencia.

Utiliza una política de control de acceso estática (ver subsección 2.1.3 en el apartado de arquitecturas de coordinación) *i.e.* una lista de nombres. Pero para poder modificar cualquier entidad de tipo *Acceso*, el usuario debe de estar dado de alta y tener el permiso de realizar la operación en la tabla de *admin*, la cual es una instancia de la entidad de tipo *Accesso*. También se crearon otras instancias de tipo *accesso* como son *users* (usuarios que tienen permiso de conectarse, crear y modificar entidades tipo *Archivo* y tipo *Chat*) y *superadmin* (usuarios con permisos de hacer actualizaciones dinámicas).

La política de control de concurrencia definida por default para este tipo de entidad es optimista, debido a que en este momento sólo se utiliza una arquitectura centralizada ya que no se cuenta con el EUS, con acceso por medio de proxies y solo puede accesar uno a la vez debido al control de concurrencia entre procesos que garantizan que dos procesos no pueden escribir simultáneamente en un mismo bloque de memoria. El tipo de comunicación es TCP que es el propuesto en la sección 2.2.

El tratamiento por defecto de los de fragmentos externos no hace nada. Cuando se define una entidad tipo `archivo` estos métodos se redefinen y se crea el archivo donde se almacenan los datos. Cuando se solicita el fragmento, en vez de regresar los datos de la ubicación, la entidad abre el archivo y envía los datos. Cuando el fragmento es modificado, la entidad abre el archivo y guarda los nuevos datos. Al eliminar el fragmento, la entidad también elimina el archivo.

6.4.2 Políticas de control de acceso

Para poder probar la flexibilidad del módulo de control de acceso se implementaron dos políticas de acceso diferentes: acceso total y acceso estático *i.e.* utilizando listas nombres

Acceso Total (access1.rb)

Esta política de control de acceso autoriza cualquier operación, *i.e.* siempre regresa *validado*. Esta política sólo es utilizada para agregar mensajes a los chats. Se supone que el usuario fue validado en el momento en que se conectó, por lo que es un usuario válido y todos los usuarios válidos pueden agregar nuevos fragmentos al *chat room*.

Acceso por listas de acceso (`accesso.rb`)

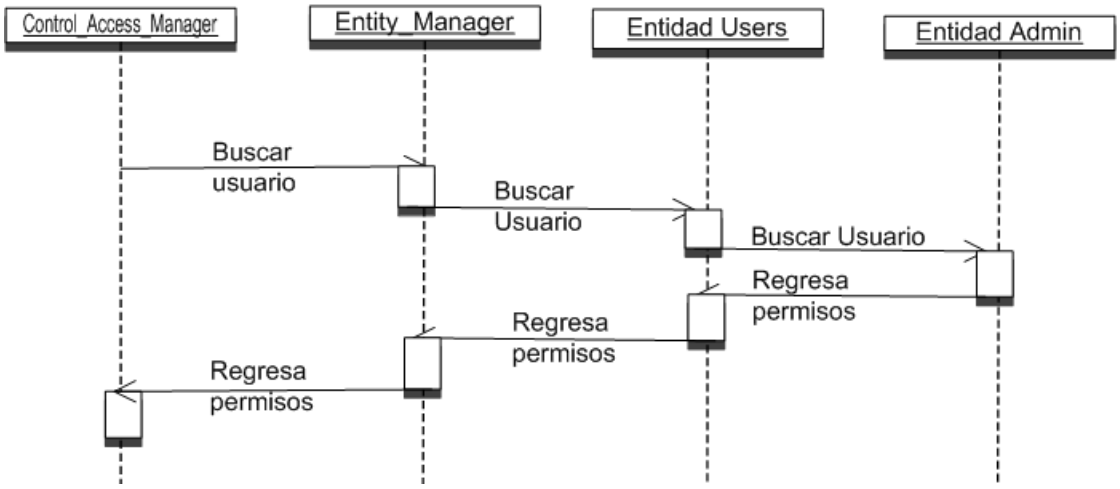

Figura 6.3: Validación de un usuario utilizando listas de acceso

Esta política utiliza listas de acceso para autorizar y validar una operación. Las listas de acceso son entidades distribuidas replicadas del tipo acceso (explicadas anteriormente). El flujo para evaluar una operación se puede ver en la figura 6.3. El `Control_Access_Manager` recibe el usuario y la instancia (tabla) con la que se va a evaluar la operación. La política solicita al `Entity_Manager` que busque en la entidad al usuario. El manejador de entidades realiza la búsqueda, primero en los registros internos de la entidad y luego en los registros externos y regresa la información encontrada del usuario. Si no la encuentra regresa `nil`. El control de acceso verifica que tenga el permiso para la operación solicitada y regresa el resultado al módulo que se lo solicitó.

Esta política es la que se usa actualmente para casi todas la operaciones dentro de la plataforma (exceptuando agregar mensajes a la cola de los chats que utiliza una política de control de acceso total).

De igual menera que se implementaron estas dos políticas, se pueden implementar otras haciendo la plataforma flexible y extensible.

6.4.3 Políticas de control de concurrencia del EUS

Como ya se dijo, falta integrar el *EUS*. Se implementó la interfaz y se verificó el envió de la validación del control de concurrencia y las notificaciones de las operaciones realizadas en las entidades compartidas para que el EUS sincronice estas entidades. Como no se implementó ninguna política de control de concurrencia, ésto equivale a una política de control de concurrencia optimista sin corrección de inconsistencias.

No se implementaron políticas de recuperación de fallas.

Como arquitectura de distribución, la única arquitectura implementada fue una arquitectura centralizada con acceso remoto utilizando proxies.

Las pruebas con el EUS fueron pobres debido a que no es tema de esta tesis, y sólo

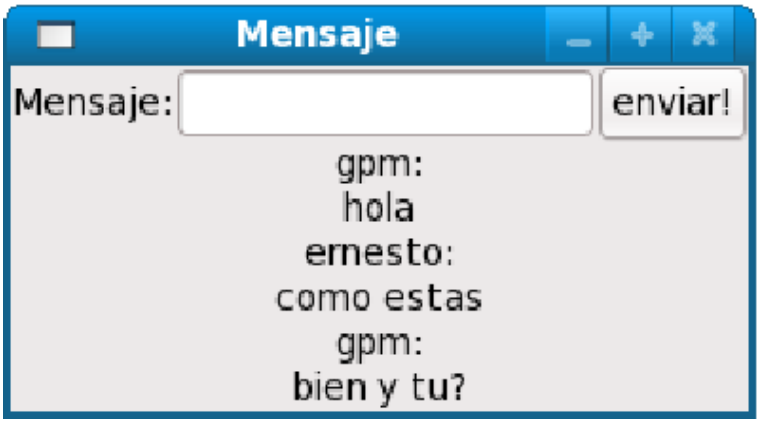

Figura 6.4: Interfaz de usuario del chat room

se implementaron pequeñas partes para probar sus relaciones.

6.5 Desarrollo de una aplicación colaborativa usando la plataforma

6.5.1 *Chat room*

Para probar la integración de una aplicación a la plataforma mínima y la interacción entre los módulos de la plataforma, se implementó una aplicación colaborativa de tipo *chat room*.

Se utilizó el tipo de entidad `Acceso` con las instancias de `users` y `admin`. También se utilizó el tipo de entidad de `Chat` con la instancia `chat`.

Para la implementación de una interfaz de usuario mínima (ver figura 6.4), se utilizó la libreria de *Ruby* `GTK2`.

Para poder probar las notificaciones, se implemento un proxy remoto, el cual aplica modificaciones en la entidad chat y son enviadas a la aplicación. El flujo de los mensajes entre los usuarios se puede ver en la figura 6.5.

El usuario local es "gpm" y el usuario remoto interactua mediante el *proxy* es "ernesto". El usuario "gpm" se conecta y sus operaciones son validadas. Una vez realizada la operación, la plataforma envía la notificación a la aplicación a través del API y ésta es reflejada en la interfaz como vemos en la figura 6.4. La interacción entre los módulos se puede ver en la figura 6.6. En las primeras 6 lineas (1) la plataforma valida la conección del usuario "gpm", primero en la entidad de `users` y después en la entidad de `admin`. Finalmente el control de acceso autoriza la conección (2) . Luego la aplicación solicita agregar un framento con el mensaje "hola" (3). La operación es validada por el control de acceso utilizando la política de acceso total (4) y después valida el EUS (5). Una vez realizada la operación se notifica al EUS y al API (6). El segundo mensaje

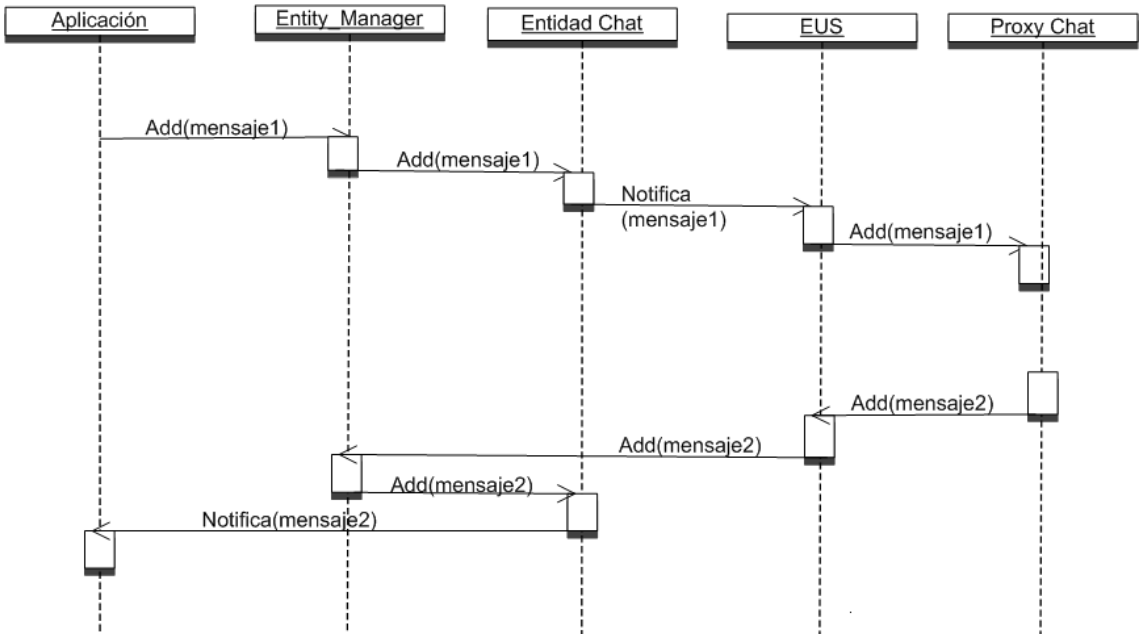

Figura 6.5: Flujo de los mensajes entre los usuarios del *chat room*

llega por medio del MU (7) y sigue el mismo proceso que el mensaje anterior. Hay un tercer mensaje, el cual tiene el mismo tratamiento que el primero.

A pesar de no tener la implementación completa del EUS se logró crear un *chat room*, pero con ciertas limitaciones. Sin embargo el objetivo de establecer y validar la interacción entre los diferentes módulos se logró.

6.5.2 Pruebas de la entidad de tipo de "archivo"

Se creó una segunda aplicación para probar las entidades tipo `Archivo`, pero no se creó una interfaz, simplemente se mandaban las instrucciones por medio del *API*, se validaba la operación y se notificaba al *EUS* y a la aplicación por medio del *API*. Podemos ver en la figura 6.7 Las operaciones realizadas.

Las diferentes operaciones realizadas con sus correspondientes validaciones y notificaciones son las siguientes:

- se validó la conexión del usuario "gpm" (1),
- se creó la entidad `texto2.ent`, de tipo `Archivo`, validada por el control de acceso utilizando la política basada en listas (2),
- Se agregó el fragmento "prueba 1 de texto2" a la entidad `texto2` (3),
- Se agregó el fragmento "prueba 2 de texto2" a la entidad `texto2` (4),
- Se agregó el fragmento "prueba 3 de texto2" a la entidad `texto2` (5),

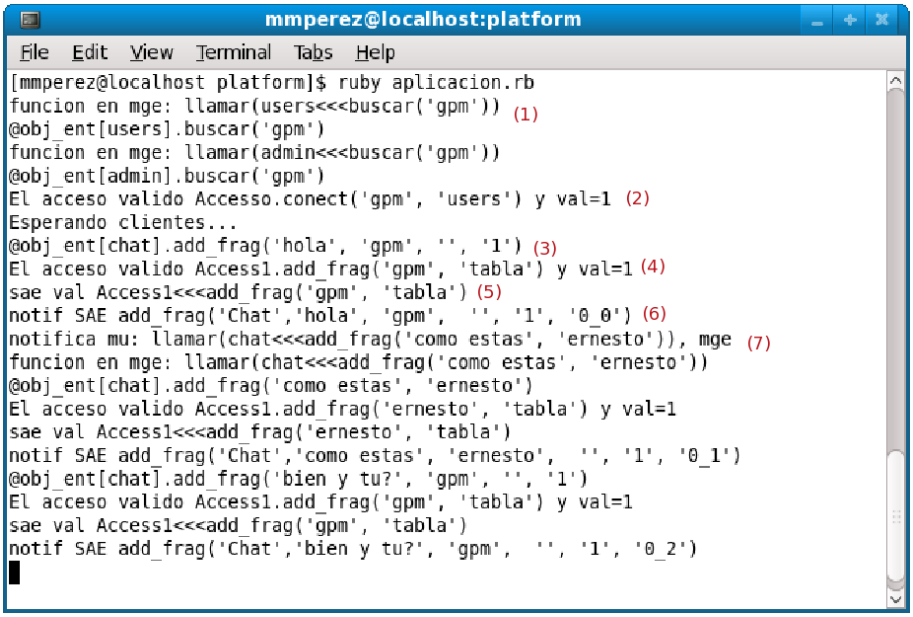

Figura 6.6: Instrucciones en la plataforma

- Se salvo la entidad `texto2` (6) y se guardo en el archivo por omisión `texto2.ent`,
- Se obtuvo el fragmento “0_1” de la entidad `texto2`, teniendo como resultado “prueba de texto2” (7),
- Se eliminó el fragmento “0_1” correspondiente a “prueba de texto2” (8),
- Se modificó el fragmento “0_2” el cual era “prueba 3 de texto2” por “prueba de cambio’.

6.6 Implementación del actualizador de versiones de software

Para implementar el actualizador se utilizó una política de inicio basada en un horario, *i.e.* se establece la hora en que se iniciará la actualización. En cuanto a la manera de actualizar los nodos, se creó una política basada en una tabla con un orden preestablecido, pero sólo se actualizó un nodo, debido a que se utilizó una arquitectura de distribución centralizada.

Las actualizaciones dinámicas realizadas consistieron en: agregar un tipo de entidad y modificar un atributo a una instancia del tipo agregado.

Figura 6.7: Operaciones realizadas con la entidad tipo archivo

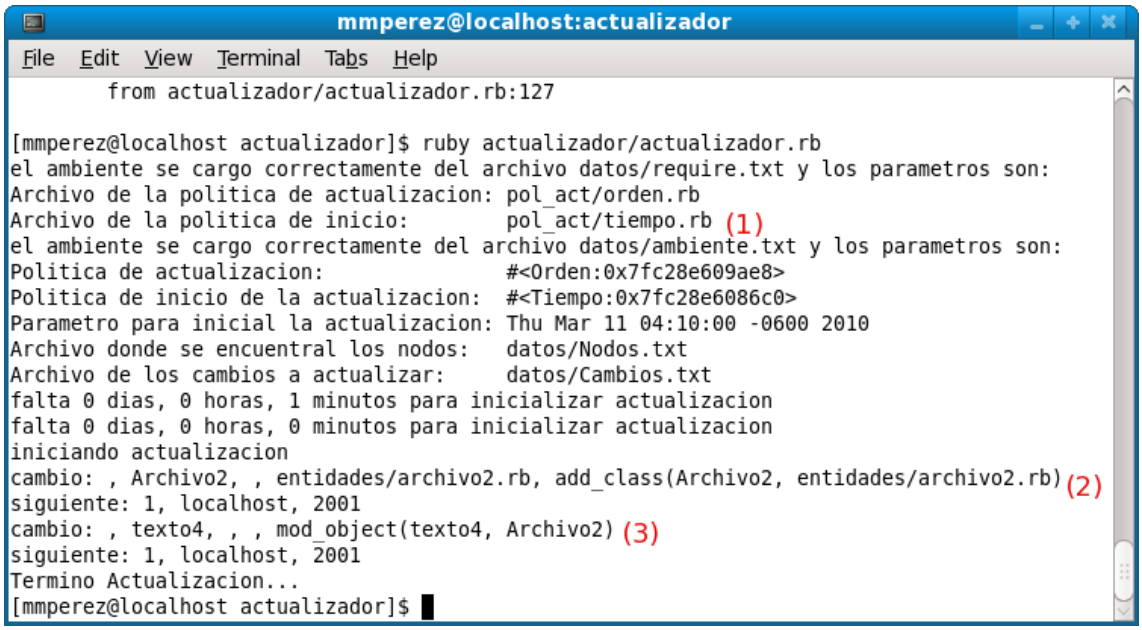

Figura 6.8: Pruebas de actualizaciones dinámicas

6.7 Pruebas de flexibilidad y actualizaciones dinámicas

Se hicieron un par de pruebas, pero es necesario crear escenarios más complejos que nos permitan evaluar a fondo su funcionalidad, sin embargo, para llevar a cabo estas pruebas es necesario a) la integración del *EUS*, y b) probarlo con arquitecturas más complejas y así estudiar el comportamiento y la recuperación de la aplicación en su globalidad. También es necesario analizar la sensación de afectación en los usuarios que se encuentren utilizando aplicaciones montadas sobre la plataforma.

A pesar de las restricciones del prototipo desarrollado, a continuación se presenta las pruebas que fueron realizadas, las cuales se pueden ver en la figura 6.8.

6.7.1 Agregación de un nuevo tipo de entidad

Durante la ejecución de la aplicación del *chat room*, se agregó un nuevo tipo de entidad. Esta entidad es una variante del tipo `Archivo`, el cual en vez de utilizar una política de control de acceso basada en listas, utiliza una política de acceso total explicada anteriormente. El tipo de entidad es llamado `Archivo2` y almacenado en `archivo2.rb`.

El actualizador sólo actualiza un nodo y envia al DAUS la instrucción de agregar la entidad tipo `Archivo2` (2).

Para verificar su funcionamiento, se utilizó un *proxy* que creó una entidad del tipo `Archivo2`.

6.7.2 Modificación de la política de control de acceso de una entidad

Después de agregar un nuevo tipo de entidad, se solicita el cambio de tipo de entidad de la instancia `texto4`, para que sea una entidad del tipo `Archivo2` (3). Se agrega un fragmento para verificar la política de control de acceso, utilizando el usuario "ernesto" constatándose que utilizaba la política de control de acceso total (*i.e.* `Access1`). Se cambió el atributo `Access_Control_Manager` de `["Access1", "tabla"]` a `["Accesso", "users"]` y se verificó su funcionamiento agregando un nuevo fragmento y verificando que utilizara la política de control de acceso basada en listas (*Accesso*).

Debido a que no se cuenta con el EUS y sólo se implementó un prototipo de la plataforma flexible y extensible, no fue posible realizar la implementación de una aplicación colaborativa real. Tampoco fue posible crear diferentes escenarios de distribución con políticas de sincronización complejas. Sin embargo se cumplieron los siguientes objetivos:

- Se verificó la flexibilidad del módulo de Control_Access_Manager al utilizar diferentes políticas de control de acceso.
- Se probó el modelo para generar diferentes tipos de entidades a partir de el modelo de entidad genérica.
- Se verificó el flujo de información entre los diferentes módulos funcionales (Control_Access_Manager, Entity_Manager, EUS, DAUS y API) de la plataforma flexible y extensible.
- Se probó la funcionalidad del Entity_Manager.
- Se verificó la modificación de un tipo de entidad en tiempo de ejecución.
- Se probó que la plataforma fuera extensible al agregar una política de control de acceso y un tipo de entidad en tiempo de ejecución.
- se verificó la comunicación de una aplicación colaborativa con la plataforma flexible y extensible por medio del API.

Capítulo 7

Conclusiones y trabajo a futuro

El objetivo de este trabajo de investigación consistió en diseñar un sistema genérico, flexible y extensible de entidades compartidas replicadas. Así, este trabajo se enfocó en modelar, diseñar y definir el prototipo de un sistema extensible de entidades flexibles y para aplicaciones cooperativas distribuidas. La finalidad de este sistema es permitir al programador definir nuevos tipos de entidades replicadas a partir de modelos existentes y eventualmente recuperar y modificar los tipos ya declarados, de una manera coherente y consistente, facilitando el desarrollo y mantenimiento de aplicaciones colaborativas distribuidas. Dichas entidades estarán constituidas por: a)un tipo, b) un estado (estructura de datos), c)una capa funcional, d) una capa de administración de la replicación de los datos y e) una capa de actualizaciones de versiones de software de las clases y su instancias en tiempo de ejecución en un ambiente de colaboración distribuido.

La capa de administración de versiones de software de las clases y sus instancias debe administrar la evolución de la entidad compartida (tipo y estructura de las instancias), y proveer los mecanismos de actualización en tiempo de ejecución (*on the fly updating*), considerando que las aplicaciones colaborativas están formadas por entidades multiprogramadas, compartidas, distribuidas y replicadas.

En este capítulo presentamos las conclusiones de este trabajo de tesis, Así como los beneficios y limitaciones de la plataforma propuesta.

Dado que el desarrollo de una plataforma que cubra todos los aspectos de las aplicaciones colaborativas distribuidas es complejo y árduo, el objetivo de este trabajo no fue proporcionar soluciones completas, sino estudiar e investigar nuevas técnivas y principios que permitan soportar de manera eficiente el trabajo cooperativo en un ambiente distribuido. Así proponemos soluciones que pueden mejorar el diseño de una plataforma, pero que por razones de tiempo con el que se cuenta para el diseño e implementación del modelo no fue posible desarrollar actualmente, por lo que se deja como trabajo a futuro.

Este trabajo también presenta los principios y las funciones desarrollados en la plataforma, que ayuden a la toma de decisiones para el diseño de las entidades replicadas distribuidas, para la elección de las políticas de control de concurrencia y distribu-

ción que mejor se adapten a la naturaleza de los datos y el entorno de cooperación del usuario.

7.1 Conclusiones

Mediante el análisis de las diferentes aplicaciones colaborativas, se definió un modelo de entidad genérica para representar y administrar datos distribuidos, compartidos y replicados. A partir de esta entidad genérica, es posible derivar entidades específicas, facilitando su reutilización ya que sólo es necesario redefinir las partes de la entidad que cambian debido a las características propias de su tipo.

Se desarrolló una plataforma mínima que permite integrar diferentes módulos:

- **Entity_Manager**: crea, almacena, borra y modifica las instancias de las entidades compartidas distribuidas. También controla el acceso local a sus métodos.
- **Access_Control_Manager**: valida las operaciones realizadas por los usuarios en los distintos elementos de la plataforma de acuerdo al rol del usuario.
- **API**: permite la comunicación de la aplicación colaborativa local con la plataforma.
- **EUS (*Entity Update System*)** actualiza y sincroniza las entidades replicadas entre los diferentes sitios de colaboración.
- **DAUS (*Dynamical Application Updating System*)** ejecuta las actualizaciones dinámicas de software.

Por su diseño, cada parte de la plataforma tiene funciones bien definidas, facilitando su conceptualización. De esta manera es fácil integrar nuevos módulos que participen en el proceso.

La plataforma especialmente diseñada para ser flexible y extensible, permite agregar nuevos tipos de entidades y nuevas políticas de control de concurrencia, de replicación/actualización, de acceso, *etc*. El agregar nuevas políticas permite al desarrollador de aplicaciones cooperativas definir aplicaciones más complejas, que soporten diferentes arquitecturas de distribución, que permitan un buen desempeño de las aplicaciones en ambientes poco fiables como el Internet y en ambientes dinámicos como el trabajo nómada colaborativo.

La capa de actualizaciones dinámicas no sólo ayuda a hacer flexible la plataforma, al permitir agregar nuevos tipos de entidades y nuevas políticas, sino que permite actualizarlos en tiempo de ejecución, proveyendo a las aplicaciones de una alta disponibilidad.

Se diseñaron los principios y la estructura del actualizador, responsable de notificar los cambios, de agendar y de controlar el proceso de actualización de la plataforma.

Aunque el *chat room* que se implementó es una aplicación relativamente sencilla de desarrollar, se pudo probar la simplificación al no tener que preocuparnos por desarrollar la comunicación, la replicación y la sincronización de la información compartida distribuida, así como el uso de tipos de entidades diferentes, con diferentes políticas de control de acceso, control de concurrencia y arquitectura de distribución.

7.1.1 Ventajas del modelo propuestos

Las ventajas principales del modelo son las siguientes:

- **Arquitectura dinámica**: la plataforma puede soportar cualquier tipo de arquitectura de cooperación distribuida. Esta arquitectura inclusive se puede adaptar en tiempo de ejecución.

- **Diferentes estrategias de replicación**: la plataforma puede soportar distintas políticas de control de acceso, control de concurrencia, recuperación de fallas, *etc.*, requeridas para la administración de los datos compartidos distribuidos.

- **Flexibilidad en el tratamiento de los datos**: las aplicaciones pueden utilizar más de un tipo de datos, y cada tipo de datos de acuerdo a su naturaleza, asocia diferentes políticas de control de concurrencia y distribución, . Esta separación entre los datos compartidos replicados y sus políticas de administración, se traduce en un mejor desempeño de la aplicación, ofreciendo un mayor confort a los desarrolladores y por transición a los usuarios.

- **Verificación de diferentes modelos de distribución**: la plataforma permite probar diferentes políticas para un mismo tipo de datos, generando tipos de entidades con pequeñas variaciones, con la intención de probar cual es la que mejor se adapta a la naturaleza de la aplicación y al entorno del usuario.

- **Reutilización de código**: es posible utilizar un mismo tipo de entidad para la implementación de varias aplicaciones, permitiendo compartir y reutilizar el código y haciendo que el desarrollo de las aplicaciones colaborativas distribuidas sea más sencillo. Además, facilita la definición y la administración de un conjunto de aplicaciones que usan las mismas entidades compartidas.

- **Un driver de replicación** La implementación de las aplicaciones colaborativas distribuidas, se simplificará de manera significativa, una vez que se tenga un buen conjunto de entidades implementadas, ya que los desarrolladores no tendrán que implementar la comunicación y sincronización de las réplicas entre los sitios de los colaboradores.

- **Una plataforma extensible**: el modelo permite agregar fácilmente nuevos módulos funcionales, nuevos actores, nuevas políticas de control de acceso, nuevas

políticas de distribución y control de concurrencia de una manera rápida, sencilla, sin mezclar código y en tiempo de ejecución, haciendo la plataforma extensible.

- **Reducción del tiempo de implementación**: el tiempo de implementación se reduce ya que los tipos de entidades y las diferentes políticas asociadas son partes que se pueden reutilizar para el diseño y la implementación de varias aplicaciones.

- **Simplificación del proceso**: la conceptualización del proceso de replicación de las entidades es más facil de entender y controlar, debido a que cada actor dentro del modelo tiene funciones específicas.

- **Alta disponibilidad**: la disponibilidad de las aplicaciones se incrementa, aún cuando se requieran modificaciones a las entidades y nuevas funcionalidades, debido a que la plataforma soporta actualizaciones dinámicas.

7.1.2 Deficiencias y limitaciones de la solución propuesta

El estado actual de la plataforma es un prototipo, al cual le faltan algunas de sus partes o fueron simuladas, por lo que queda mucho trabajo pendiente para que realmente pueda ser operacional.

Por la misma razón, no se cuentan con todos los módulos por lo que su funcionalidad está muy limitada.

Las políticas de control de acceso, distribución y control de concurrencia implementadas y experimentadas son muy pocas, por lo que es necesario desarrollar otros tipos de políticas para poder comprobar el soporte de cualquier aplicación colaborativa distribuida.

Igualmente, puede ser difícil decidir cuales son las políticas de control de concurrencia y de distribución que mejor se adaptan a una aplicación específica o inclusive, a un tipo de datos en particular. Desde este punto de vista, nos falta experimentar mucho más para adquirir un conocimiento amplio y preciso del comportamiento de estas políticas de administración de entidades compartidas distribuidas.

Para poder llevar a cabo una actualización dinámica es necesario conocer a fondo la implementación de la plataforma, ya que no cuenta con las herramientas que faciliten el trabajo.

7.2 Trabajo a futuro

Actualmente contamos con un prototipo, el cual no tiene todas las funcionalidades necesarias para implementar cualquier tipo de aplicación cooperativa distribuida, por lo que queda mucho trabajo a futuro. En esta sección presentamos algunos aspectos en los que se requiere desarrollar la plataforma.

7.2.1 Plataforma

Como primer punto, es necesario integrar una versión operacional del EUS (*Entity Update System*), verificar el uso de diferentes políticas de control de concurrencia, diferentes arquitecturas de distribución, diferentes políticas de recuperación de fallas y distintos protocolos de comunicación. Principalmente, es necesario manejar HTTP para poder implementar aplicaciones que se ejecuten en el Internet. Sin la integración del EUS la plataforma no es realmente funcional.

Para poder soportar políticas optimistas basadas en acciones "hacer" y "deshacer", es necesario diseñar, implementar e integrar el módulo de manejo de versiones, el cual debe permitir regresar a una entidad a un estado anterior.

Otro módulo que se requiere implementar para soportar el trabajo nómada, es el modulo de dispositivos móviles. Este módulo tiene por objetivo ayudar a los dispositivos de baja capacidad a filtrar la información que no son capaces de manejar, a degradar la calidad de la voz y el video para que puedan soportar la aplicación, a transmitir las notificaciones cuando no tienen la capacidad de enviarlas a todos los sitios, *etc*.

7.2.2 Entidades distribuidas y políticas de administración

Actualmente sólo se crearon algunos tipos de entidad con la intención de probar la flexibilidad del modelo. Pero es necesario verificarlo implementando diferentes tipos de datos que se adapten a un mayor número de aplicaciones y que nos permitan hacerlas cada vez más funcionales, ofreciendo un mayor confort para los programadores de aplicaciones cooperativas.

También es necesario implementar un mayor número de políticas de control de acceso, distribución, control de concurrencia, *etc.*, haciendo la plataforma capaz de proveer servicios para varios tipos de aplicaciones cooperativas.

7.2.3 Aplicaciones

Es necesario estudiar más a fondo las aplicaciones existentes, crear diseños que otorguen un mayor confort a los usuarios, al utilizar la política que mejor se adapte al tipo de datos y al ambiente del usuario. Es recomendable probar diferentes modelos y ver el comportamiento de la aplicaciones utilizando diferentes políticas de distribución y control de concurrencia, en diferentes ambientes, *i.e.* probarlas tanto en ambientes cara a cara, locales, disbribuidos, nómadas, *etc.*, con la finalidad de definir las variables que influyen en el comportamiento de las aplicaciones y crear patrones que permitan crear aplicaciones con un mejor desempeño y menor número de inconsistencias. Inclusive, al crear un modelo de comportamiento los sistemas podrían adaptarse dinámicamente al entorno del usuario.

Un trabajo interesante y muy útil, sería el desarrollo de aplicaciones cooperativas distribuidas que sean realistas, útiles, de gran tamaño que:

- usen diferentes políticas (replicación, actualización, *etc.*),
- se ejecuten adecuadamente en dispositivos heterogeneos (del punto de vista recursos y potencialidades),
- trabajen en un ambiente de trabajo móvil y/o nómada.

7.2.4 Actualizaciones dinámicas

El modelo propuesto requiere más pruebas, para verificar su correcto funcionamiento. Es necesario probar el modelo utilizando arquitecturas de distribución más complejas con la finalidad de verificar el comportamiento de las aplicaciones colaborativas distribuidas durante el proceso de actualización de versiones de software.

Es deseable diseñar un sistema que sea capaz de seleccionar la mejor entidad, de acuerdo al tipo de dispositivo y las condiciones de la red. Este sistema podría ser capaz de rediseñar la arquitectura sin afectar las notificaciones de los cambios a cada uno de los nodos.

Se puede desarrollar un lenguaje de programación de dominio específico basado en programación orientada a aspectos para separar mejor cada funcionalidad de la entidad. Principalmente, es deseable crear un aspecto de actualizaciones dinámicas, el cual permita agregar la funcionalidad de actualizaciones dinámicas a cualquier aplicación al entrelazar cada elemento de esta capa con la aplicación.

Se requiere estudiar a detalle las funciones de transformación para los diferentes cambios y sus soluciones, sobre todo las funciones que permitan la comunicación cuando se realizan cambios a las políticas de distribución y notificación.

Es deseable crear un analizador de código que pueda generar las funciones de transformación de manera automática.

Bibliografía

[1] C.S. Ellis, S.J. Gibbs, and G.L. Rein, Groupware: Some Issues and Experiences, *Communications of the ACM,* 34(1):38-58, January, 1991

[2] Clarence Ellis, and Jacques Wainer, A conceptual model of groupware, *CSCW '94 proceeding of 1994 ACM conference on Computer supported cooperative work ,* ACM press NY USA, 79-88 pp., ACM press, North Carolina, USA, October, 1994

[3] Saul Greenberg, and David Marwood, Real Time as a Distributed System: Concurrency Control and its Effect on the Interface, *ACM CSCW Conference on Computer Supported Cooperative Work,* ACM Pres, 207-217 pp., North Carolina, October, 1994

[4] Paul Dourish, and Victoria Bellotti, Awareness and Coordination in Shared Workspaces, *Computer Supported Cooperative Work,* ACM press, 107-114 pp., Toronto, Canada, December 1992

[5] Shum Buckingham, S. Marshall, S. Brier and J. Evans, Lyceum: Internet Voice Groupware for Distance Learning, *Euro-CSCL 2001: 1st European Conference on Computer - Supported Collaborative Learning*, 22-24 pp., Maastricht Netherlands, March, 2001

[6] W. Keith Edwards, Policies and Roles in Collaborative Applications, *1996 ACM conference on Computer Supported Cooperative Work*, ACM press, 11-20 pp., Boston, USA, November, 1996

[7] Yasushi Saito, and Marc Shapiro, Optimistic Replication, *ACM Computing Surveys*, 37(1):42-81, March, 2005

[8] J.F. Patterson, A Taxonomy of Architectures for Synchronous Groupware Applications, *ACM SIGOIS Bulletin Special Issue: Papers of the CSCW'94 Workshops,* 15(3):27-29, April, 1995

[9] D. Decouchant, S. Mendoza, J. Rodriguez, *Chapter: "Suited Support for Distributed Web Intelligence Cooperative Work"*, of the book edited by: R. Chbeir, Aboul-Ella Hassanien, Ajith Abraham, Youakim Badr *"Emergent Web Intelligence", "Advanced Information and Knowledge Processing"* series, Springer Verlag Publisher, 48 pages, to appear March 2010.

[10] Dominique Decouchant, Esther Martínez Gonzalez, and Ana María Enriquez, Alliance Web: Cooperative Authoring on the WWW, *String Processing and Information Retrieval Symposium & International Workshop on Groupware SPIRE/CRIWG,* IEEE Computer Society, 286 pp., Cancún México, September, 1999

[11] Coulouris George, Dollimore Jean, and Kindberg Tim, *Distributed Systems Concepts and Design*, Fourth ed., Addison Wesley, USA, 2004

[12] Andrew S. Tanenbaum, and Maarten Van Steen, *Sistemas distribuidos. Principios y paradigmas,* 2nd ed., Pearson Pretince Hall, Estado de México, México, 2008

[13] Bowei Du, and Eric A. Brewer, DTWiki: A Disconnection and Intermittency Tolerant Wiki, *In International Word Web Conference Comitte,* 945-951 pp., ACM press, Beijing, China, April, 2008

[14] E. James Whitehead Jr., and M. Wiggings, WEBDAV: IETF Standard for Collaborative Authoring on the Web, *IEEE Internet Computing*, 2(5): 34-40, September-October, 1998

[15] E.J. Whitehead, and Y. Y. Goland, WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web, *In ECSCW'99 the 6th European Conference on Computer Supported Cooperative Work*, ed. Kluwer, 291-310 pp., Copenhague Danemark, September, 1999

[16] Sunghun Kim, and E. James Whitehead Jr., WebDAV-based Hypertext Annotation and Trail System, *Proceedings of the 15th ACM Conference on Hypertext and Hypermedia,* ACM press, 87-88 pp., Santa Cruz California, USA, August, 2004

[17] Youngki Lee, Sharad Agarwal, Chiris Butcher, and Jitu Padhye, Measurement and Estimation of Network QoS Among Peer Xbox 360 Game Players, *PAM 2008,* M. Claypool and S. Uhlig, 41-50 pp., Springer-Verlag Berlin, Heidelberg Germany, 2008

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* 4a ed., Addison Wesley Professional Computing Series, Boston USA, 1995

[19] Chistopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel, *A Pattern Language,* 1st. ed., Oxford University Press, New York, USA., 1977

[20] Russ Olsen, *Design Patterns in Ruby,* 1st. edition, Addison-Wesley Professional Ruby Series, Boston USA, 2007

[21] Theodore O'grady, *Flexible Data Sharing in a Groupware Toolkit*, Master Thesis, Calgary University, November, 1996

[22] Stephen Lukosh, Adaptive and Transparent Data Distribution Support for Synchronous Groupware, *8th International Workshop, CRIWG 2002*, Springer Berlin, 431-499 pp., Springer Berlin, La Serena Chile, September, 2002

[23] Stephan Lukosch, and Till Shummer, Communicating Design with Groupware Technology Patterns, *CRIWG 2004*, Springer Verlag, 223-237 pp., Springer Verlag, Heidelberg Alemania, 2004

[24] Cristian Schuckmann, Lutz Kirchner, Jan Schümmer, and Jörg M. Haake, Designing Object-Oriented Synchronous Groupware with COAST, *1996 ACM Conference on Computer Supported Cooperative Work,* ACM New York, 30-38 pp., ACM press, Boston USA, November, 1996

[25] Sameer Ajmani, Barbara Liskov, and Liuba Shrira, Modular Software Upgrades for Distributed Systems, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings,* Dave Thomas, 452-476 pp., Springer, Francia, Julio, 2006

[26] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana, and Giuseppe Ursino, Handling Runtime Updates in Distributed Applications, *Applied Computation Simposium,* 1375-1380 pp., ACM press, New Mexico USA, March, 2005

[27] Andreas Rasche, Wolfgang Schult, and Andreas Polze, Self-Adaptive Multithreaded Applications - A case for Dinamic Aspect Weaving, *ACM International Conference Proceeding Series 4o Workshop on Reflective and Adaptive Middleware Systems,* ACM press, 1-6 pp., New York USA, November, 2005

[28] Susanne Cech, and Thomas Gross, Dynamic Updates: Another Middleware Service, *1st Workshop on Middleware Application Interaction (MAI),* 49-54 pp. ACM press, Lisboa, Portugal, March, 2007

[29] P. Pissias, and G. Coulson, Framework for Quiescence Managment in Support of Reconfigurable Multi-thread Component-Base System, *IET-Software,* 2(4):348-361, August, 2008

[30] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu, Mutatis Mutandis: Safe and Predictable Dynamic Software Updating, *ACM Transactions on Programming Languages and System,* 29(4):183-194, January, 2005

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, Aspect - Oriented Programming. Technical report, Xerox Palo Alto Research Center, Palo Alto CA, USA, June, 1997

[32] Cristina Vidiera Lopes, and Gregor Kiczales, A Language Framework for Dsitributed Programming. Technical report, SPL97-010, Xerox Palo Alto Research Center, USA, February, 1997

[33] David Flanagan, and Yukihiro Matsumoto, *The Ruby Programming Language,* First Ed., O'Reilly, San Francisco Ca. USA, January, 2008

[34] Hal Fulton, *The ruby way,* 2o ed., Addison-Wesley Professional Ruby Series, Boston USA, 2007

[35] Andrew S. Tanenbaum, *Sistemas Operativos Modernos,* 2a ed., Pearson Education, Mexico, 2003

[36] P.J. Courtois, F. Heymans, and D.L.Parnas, Concurrent Control with "Readers" and "Writers", *Communications of ACM,* 14(10):667-668, October, 1971

[37] Daniel Cruz García, *Diseño e implementació de una agenda cooperativa flexible en un entorno no confiable,* Master thesis, Centro de Investigación y de estudios avanzados del Instituto Politécnico Nacional (Cinvestav), febrero, 2009

[38] Miguel Navarro Dávila, *Sistema de actualización de entidades replicadas en un ambiente colaborativo distribuido*, Master thesis, Centro de Investigación y de estudios avanzados del Instituto Politécnico Nacional (Cinvestav), marzo, 2010

[39] Javier Solís Angulo, *Arquitectura de un sistema adaptable para sistemas colaborativos implantados en la Web.* Master Thesis Centro de Investigación de estudios avanzados del Instituto Politécnico Nacional (Cinvestav), enero, 2009

Otras Referencias

[40] Mapilab, www.mapilab.com

[41] Cebos,www.editor.cebos.com

[42] Exodus,www.exodus.jabberstudio.orb

[43] TeamIntegrator,JCP, www.alliancing.co.uk

[44] Carthago, www.carthagosoft.net

[45] Kopete,kopete.kde.org

[46] Pidgin, www.pidgin.im

[47] Live-Messenger, Windows Live Messenger, windows-live-messenger.softonic.com

[48] Goggle Talk, www.google.com/talk

[49] Horde, www.horde.org

[50] A.M.S.N., aMSN Messenger, www.amsn-project.net

[51] Microsoft Office Outlook, Microsoft Office Outlook, office.microsoft.com/es-es/outlook/HA101743593082.aspx

[52] Wikipedia Mosaic, es.wikipedia.org/wiki/Mosaic

[53] Wikipedia Espacio de nombres, es.wikipedia.org/wiki/Espacio_de_nombres

[54] Ruby, www.ruby-lang.org/es/

[55] Python Programming Language - Official Webside, www.python.org