Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Sistema de actualización de entidades replicadas en un ambiente colaborativo distribuido

Tesis que presenta

Miguel José Navarro Dávila

para obtener el Grado de

Maestro en Ciencias

en Computación

Directores de la Tesis

Dr. Dominique Decouchant

Dr. José Guadalupe Rodríguez García

México, D. F. **30 de marzo de 2010**

Resumen

La evolución en el desarrollo de la infraestructura de la red mundial de información (Internet), origina cambios en la manera en como creamos y damos soporte al trabajo cooperativo de las personas, organizaciones e instituciones. Dentro de este ambiente, altamente distribuido, es de gran importancia proporcionar espacios de trabajo robustos e integrados capaces de soportar el trabajo colaborativo, y ofrecer herramientas adaptables y flexibles que permitan simplificar sus tareas y los ayuden a alcanzar sus objetivos de manera eficiente.

El objetivo principal de los sistemas y aplicaciones colaborativas está enfocado en definir y ofrecer herramientas de producción cooperativa integrada dentro de un ambiente cooperativo, robusto, eficiente y consistente. Las herramientas cooperativas están diseñadas e implementadas tomando en cuenta los siguientes requerimientos: a) deben permitir a los usuarios producir eficientemente de una forma convenida, y b) deben asegurar la integridad de la información cooperativa producida.

Estos sistemas cooperativos están basados en la definición y el uso de entidades compartidas que son replicadas y regularmente actualizadas entre los sitios participantes. Cuando se diseña y desarrolla una aplicación colaborativa dentro de una red de área extensa, la correcta elección de la política de replicación, actualización y notificación requerida para administrar la información compartida, sigue siendo un gran problema.

Con el fin de solucionar este problema, hemos diseñado el prototipo de un sistema de actualización genérico, flexible y extensible, que provee soporte para controlar varios tipos de entidades colaborativas distribuidas requeridas por los distintos tipos de aplicaciones (e.g. agendas, pizarrones colaborativos, chats, producción cooperativa de sitios Web, definición colaborativa de ontologías y editores colaborativos.)

Partiendo de un modelo de plataforma flexible y extensible, proponemos un conjunto de herramientas útiles para el desarrollo de aplicaciones colaborativas distribuidas. Utilizando funciones de replicación, actualización y notificación que están directamente asociadas por naturaleza a las entidades compartidas, esta plataforma automáticamente administra su distribución y sincronización.

Estas funciones están encapsulada dentro de módulos de administración de replicación que pueden ser utilizados para controlar entidades diferentes. Todos los módulos de administración de la replicación utilizan la misma interfaz genérica. Debido a que cada módulo (que encapsula una política de replicación específica), puede ser fácilmente modificado y reemplazado, logramos proveer la extensibilidad y la flexibilidad en esta plataforma. De esta forma el administrador de replicación de entidades distribuidas permanece completamente independiente de las diferentes aplicaciones colaborativas que lo utilizan.

Más precisamente, este trabajo introduce los principios de diseño y la implementación de un sistema de actualización cuyo objetivo es sincronizar automáticamente las distintas réplicas distribuidas de entidades compartidas administradas. Los resultados y soluciones obtenidos se presentan y justifican en relación al campo de los sistemas colaborativos distribuidos.

Abstract

The evolution of the development infrastructure of the worldwide information network (Internet), generated changes in the way of how to conduct and support cooperative work of individuals, organizations, and institutions. Within such an highly distributed environment, it appears of main importance to provide integrated and powerful workspaces able to support collaborators' common work, and that offer flexible and adaptable tools whose aim is to simplify their tasks and help them to reach their objectives in a joint and efficient way. The main objectives of collaborative applications and systems are targeted to define and offer cooperative production tools that are integrated within a robust, efficient and consistent cooperative working environment. Such cooperative tools are especially designed and implemented taking into account the following requirements: a) they have to allow users to efficiently produce in a concerted way, and b) they have to ensure the integrity of the cooperatively produced information.

These cooperative systems are based on the definition and use of shared entities that are replicated and regularly updated between collaborators'sites. Thus, many propositions of policies have been defined to handle the replication, the updating and the notification of the distributed information. When designing/developing a collaborative application within a wide area network, the selection of the suited replication-updating-notification policy required for the management of a given shared information remains a major problem.

In order to provide a solution to this problem, we designed the prototype of a generic, extensible and flexible updating system that provides supports to handle various kinds of distributed shared entities required by the different collaborative distributed applications (e.g., group diaries, whiteboards, chats, cooperative production of websites, collaborative definition of ontologies and collaborative editors).

From the design of a flexible and extensible platform model, we propose a set of useful tools for developing distributed collaborative applications. Using functions of replication, updating, and notification that are directly associated with the nature of the shared entities, this platform automatically manages their distribution and synchronization.

Each entity replication, updating, and notification functions are encapsulated into replication management modules that can be used for the administration of different specific entities. All replication management modules follow the same generic common interface. Because each module (that encapsulate a specific replication policy) can be easily modified, derived, or replaced, the extensibility and the flexibility of this platform is achieved. In this way, the replication management of some distributed shared entities remains completely independent of the different collaborative applications that use them.

More precisely, this work introduces the design principles and the implementation of an updating system whose objective is to automatically synchronize the different distributed replicas of the managed shared entities. The provided solutions and results are presented and justified relatively to the field of distributed collaborative systems.

Agradecimientos

A mis asesores, por la paciencia, el conocimiento, y el apoyo brindado, mi más profundo agradecimiento y admiración...

Dr. Dominique Decouchant:
Fue agradable el trabajar con usted, lo disfruté. Me brindó la confianza y el apoyo necesarios para seguir adelante con este trabajo de tesis. Agradezco mucho sus consejos, el conocimiento transmitido, sus atenciones y su paciencia. Me permitió crecer en lo personal y en lo académico mucho mas de lo que pensé. Muchas Gracias.

A mis revisores:
Dra. Sonia G. Mendoza Chapa, Dr. José Guadalupe Rodríguez y Dr. Joaquín Sergio Zepeda Hernández por el tiempo dedicado a la revisión de esta tesis, por sus comentarios, consejos y su ayuda durante mi permanencia en el CINVESTAV

Agradezco al Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional por ofrecerme la oportunidad de cumplir una etapa más de mi carrera profesional, por los conocimientos adquiridos durante la permanencia y por la excelente formación que nos permitirá seguir avanzando y aportando beneficios en bien del país.

Sofía Reza:
Por todas sus atenciones, y su apoyo. Por resolver nuestros problemas, por escucharnos y por brindarnos la confianza de acudir a ella cuando lo necesitamos.

A mis amigos:
A toda la generación 2007 de computación del CINVESTAV, gracias por su amistad y por todos los ratos alegres que pasamos. A mis amigos de la UAM que siempre me animaron y también a aquellos que admiro mucho y nunca perdieron la fe en mi.

Gracias a mi familia por ser el pilar que me sostiene, que me apoya y da fuerzas para seguir adelante...

A mis papas: *No encontraré palabras suficientes de agradecimiento. Su apoyo y cariño me dieron siempre la fuerza para seguir este camino. Sus consejos y confianza la determinación para no rendirme. Su protección y solidaridad la energía para levantarme cuando había caído. Y su honestidad, responsabilidad y bondad desinteresada me otorgaron los valores y la motivación para dar mi máximo esfuerzo. Son todo para mí, este trabajo de tesis se los dedico a ustedes. Ha sido un esfuerzo en conjunto.*
Muchas Gracias, los quiero!.

A mis hermanos:

Pavel y Paris: gracias por soportarme, por apoyarme en todo lo que he querido hacer, por los momentos que hemos pasado y por la confianza que me tuvieron desde el principio para emprender esta etapa de mi vida. Gracias por su apoyo!.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Hoy en día vivimos las maravillas y comodidades que nos ofrece la red mundial de comunicación. Su crecimiento y aceptación ha sido muy grande en los últimos años, por consiguiente se ha vuelto una herramienta indispensable. Cotidianamente nos comunicamos con un amigo al otro lado del mundo tan solo con oprimir un botón, compartimos información y se trabaja de forma distribuida pero conjunta, colaborando a la producción de documentos, bases de datos, ontologías, etc.
La compartición y la globalización de las organizaciones obliga a transformar la forma de trabajo: nuevas formas de colaboración que permiten a las personas interactuar o colaborar desde cualquier parte del mundo, imponen la necesidad del desarrollo de herramientas de coordinación elaboradas y eficientes. La dificultad de proveer herramientas de coordinación adecuadas es más complicada de lo que parece.
Los avances tecnológicos recientes proporcionan la posibilidad de trabajar de manera conjunta con personas en lugares remotos, la eliminación gradual de los lugares de trabajo cotidianos, las diferencias de horarios de trabajo, etc., requieren aplicaciones que provean mecanismos de compartición, distribución, notificación y actualización eficientes y adaptables de la información compartida, así como un entorno colaborativo distribuido seguro, confiable y eficiente.

Siguiendo este objetivo se han realizado propuestas que pretenden resolver problemas relacionados con la colaboración de personas que requieren contribuir y compartir recursos, información en sitios que están en lugares geográficamente distantes. Actualmente existen muchas aplicaciones colaborativas con las que interactuamos a diario (e.g chats, editores colaborativos, agendas electrónicas, sitios web, Wikis, etc.). Todas estas aplicaciones proveen un ambiente enriquecido donde existe interacción entre grupos de personas, ya sea trabajando de manera conjunta, produciendo documentos, diseños o simplemente divirtiéndose o comunicándose.

El diseño e implementación de software colaborativo distribuido no es realmente un dominio de investigación nuevo, el TCAC (Trabajo Colaborativo Asistido por Computadora) o CSCW [Greif, 1988][Ellis and Gibbs, 1992][Johansen, 1988] por sus siglas en ingles, es un enfoque de diseño cuyo propósito es desarrollar ambientes colaborativos que faciliten la interacción entre las personas. Este dominio de investigación comenzó en los años 80 y

propone ubicar a los distintos tipos de aplicaciones cooperativas dentro de clasificaciones definidas en base a la ubicación y el tipo de sincronización necesaria para comunicarse. La clasificación considera los distintos tipos de colaboración posibles tomando en cuenta la ubicación y el tiempo, podemos identificar dos tipos de aplicaciones:

- Las aplicaciones que soportan interacciones directas y relaciones entre colaboradores.
  ✓Aplicaciones de videoconferencia, agendas, pizarrones colaborativos
- Las aplicaciones que proponen soporte para interacciones asíncronas.
  ✓La Web, Wikis, blogs, flujos de trabajo, editores de texto.

La implementación de ambos tipos de aplicación se realiza de manera distinta, por ello parte del problema radica en la necesidad que existe al aplicar los dos tipos interacción en una misma aplicación. Esto implica mecanismos distintos que no necesariamente son compatibles entre si. Observemos algunos ejemplos:

Wikis[Larman, 2005].
Son sitios Web de contenido editable cuya idea es recibir contribuciones a la información publicada en el sitio por parte de cualquier persona en cualquier parte del mundo. En un nivel básico las Wikis abarcan un conjunto de páginas web editables por el usuario escritas en un lenguaje simplificado de etiquetas. La ventaja principal de este esquema es la rapidez con la que se puede crear contenido nuevo, poseen un control de versiones que permite llevar un historial de cambios del contenido, de esta forma se puede regresar a versiones anteriores del documento.
Utiliza una arquitectura centralizada que consiste en mantener la información en un sólo repositorio. De esta manera se controlan las operaciones de escritura y lectura con un mínimo de complejidad. El problema es sin duda, la disponibilidad de los recursos en casos de congestión. Además como un único punto de ruptura, si el servidor queda fuera de línea deja totalmente inaccesible la información. Para que la interacción pueda realizarse es necesario realizar una conexión sincrónica con el sitio Web.

Chats.
Son aplicaciones para el envío de mensajes entre usuarios con motivos recreativos o laborales. Al igual que los Wikis funcionan de manera centralizada al momento de iniciar una sesión, pero posteriormente pueden funcionar de manera descentralizada comunicándose directamente con otros usuarios, (i.e. el envío de información se realiza de manera directa entre usuarios sin tener que depender enteramente de un servidor, por lo que la única tarea del servidor es enviar la lista de contactos y su ubicación).
No hay necesidad de implementar mecanismos de resolución de conflictos dado que la información sólo es punto a punto y la coordinación se realiza por medio de un protocolo social, esto significa que los usuarios resuelven los conflictos y establecen un orden de entrega de mensajes sin depender de un mecanismo automático.
La interacción se realiza de manera síncrona, es decir que deben coincidir en el tiempo para realizar la comunicación, pero la presencia es asincrónica, es decir no coinciden en ubicación.

Email.
El funcionamiento del correo electrónico es un caso ideal para mostrar otro tipo de interacción. Para realizar la comunicación no es necesario coincidir en el tiempo y ubicación. El protocolo utilizado para enviar y recibir correos electrónicos funciona enviando los mensajes de un sitio no directamente al sitio destino, sino a un repositorio temporal el cual será accedido por el destinatario en un tiempo posterior.
Se utiliza una arquitectura centralizada la cual es un repositorio temporal donde se almacenan todos los mensajes. El problema principal es que quedará completamente inaccesible la información para los usuarios si el servidor queda fuera de línea.

Como podemos observar varias aplicaciones necesitan un tipo de colaboración específica que no necesariamente es mutuamente compatible. Existen muchos detalles que necesitan ser resueltos para construir una aplicación colaborativa. En el caso de un editor de texto colaborativo se requiere de ciertas características útiles al usuario para hacerlo un producto atractivo, a nivel programador se necesitan de herramientas que faciliten el desarrollo de la capa de distribución, por ejemplo; envío de datos, coherencia de información etc.

Todas estas características se implementan usando algunos modelos de actualización de información, arquitecturas y protocolos que han sido publicados a lo largo del tiempo, algunos de estos modelos se basan en la idea de bloquear un recurso para evitar conflictos de concurrencia, otros envían actualizaciones de datos a todos aquellos que compartan un mismo documento y algunos otros periódicamente verifican sus contactos por modificaciones.
El caso del editor de texto sirve para ilustrar que existen diversas necesidades para compartición de información y que existen diversas técnicas de desarrollo actualmente. Además si el diseñador de aplicaciones colaborativas decide incorporar alguna otra característica al editor, (e.g. un mensajero instantáneo, una pantalla de videoconferencia o un área para dibujo compartido), entonces el diseño de esa aplicación puede volverse muy complejo, sin embargo quedan por mencionar algunos problemas derivados.

A continuación realizamos un pequeño análisis a algunas aplicaciones colaborativas consultadas. Nos ayudará a comprender los problemas que existen actualmente.

AccessGrid
AccessGrid es un ensamble de recursos y tecnologías que incluyen pantallas de gran tamaño, ambientes de presentación e interactivas, e interfaces, así como visualización de simulaciones científicas. Proporciona un ambiente de colaboración a través de múltiples proyectores, pantallas, cámaras, y da soporte a trabajo en grupos, es decir es un sistema de videconferencia avanzado.
Se apoya en gran medida en la multidifusión por IP (multicast) para el transporte de las señales. Esto también significa que los usos de AccesGrid están limitados a aquellos sitios en donde está activada la multidifusión en la red. Esto ha derivado en un uso casi mayoritario entre la comunidad de investigación científica interconectada típicamente por redes con multidifusión con usos industriales e instalaciones creciendo constantemente.
AccesGrid utiliza una arquitectura descentralizada, es decir que los sitios se comunican

entre si de manera directa sin pasar por un servidor central. Utiliza un protocolo de paso de mensajes llamado jabber (posteriormente llamado XMPP) el cual se ha utilizado como una extensión para mensajería instantánea, presencia de información y middleware orientado a mensajes. Tiene algunos problemas de escalabilidad, problemas de redundancia, sobrecarga y no tiene la capacidad para soportar archivos binarios, debido a que el protocolo XMPP esta codificado como un documento XML. Por lo tanto tiene algunas deficiencias en cuanto a flexibilidad, es decir que no puede ser utilizado por otro tipo de aplicaciones cooperativas sin realizar cambios al código, tarea que puede resultar muy complicada debido al alto acoplamiento de la distribución al código de la aplicación. Su utilidad queda limitada a servicios de teleconferencia y/o envío de información visual a otras terminales. Dado que el envío de información es mediante un multicast, no se puede utilizar para aplicaciones de tipo editor colaborativo, donde se requiere un tratamiento de conflictos mas eficiente.

Google Documents
Es un ambiente basado en Web para la edición compartida de documentos de texto cuya publicación se realiza de manera directa en el sitio Web. Permite la colaboración entre personas en tiempo real para la edición de los documentos. Poseen algunas características interesantes entre las que se mencionan:

- La capacidad de compartir documentos almacenados de manera distribuida, ya sea en los roles de propietario (owner), lector (reader), o colaborador (collaborator),
- La generación de documentos online con la ventaja de publicar los cambios de los otros usuarios en tiempo real,
- La capacidad de acceso a los documentos en un ambiente web y las facilidades que resultan de compartir los documentos en línea, soporta varios formatos de documentos de texto,
- La gestión y el almacenamiento de distintas versiones de los documentos, permitiendo el almacenamiento distribuido a través de los servidores de Google, dando la posibilidad de realizar cambios y modificaciones entre diferentes versiones,
- Algunos mecanismos de conciencia de grupo como la presencia de otros usuarios al compartir un documento.

La desventaja principal que observamos es que se depende de los servidores de Google para obtener los beneficios de este editor colaborativo, no es posible seguir trabajando si el servidor deja de funcionar o si el usuario no posee una conexión a la red. Básicamente el problema surge porque este tipo de tecnología se basa en el denominado cloud computing, que es un conjunto de sistemas interconectados por medio de la infraestructura de red.
Se han hecho comparaciones con servidores centralizados de los años 50 y 60 donde los usuarios se conectaban a través de terminales tontas a los mainframes. Esta comparación se realiza para poner en evidencia que el cloud computing no permite realizar modificaciones a los protocolos o a los algoritmos de almacenamiento ni a las aplicaciones que

se proporcionan, básicamente los usuarios dependen de lo que ofrezca el proveedor de servicios en cuanto a aplicaciones y permisos.

Bayou [Theimer et al., 1995] [Theimer et al., 1997]
Es un sistema de almacenamiento replicado cuya intención es dar soporte colaborativo asíncrono a cualquier aplicación colaborativa distribuida. Utiliza modelo de datos relacional y está dirigido a aquellos equipos con algún tipo de conexión intermitente o que se trasladan constantemente de ubicación, es decir equipos móviles; ya que soporta cambios y modificaciones en el contenido de la información que realicen en modo desconectado para luego integrarlos en la base de datos replicada, utilizando para ello una serie de servidores que almacenan los datos maestros. Utiliza algunas técnicas de replicación optimista donde no se requiere una interacción de manera directa entre las réplicas, es decir que la actualización puede realizarse en diferente tiempo y en distinto lugar. La resolución de conflictos se realiza mediante la verificación de dependencias que consiste en realizar algunas operaciones en la base de datos relacional, tal que se compruebe que no se altera el estado. En caso contrario se realizan algunos procesos de integración de información que pueden involucrar algunas transformaciones a las operaciones para ajustarse a un resultado deseado. Una buena cualidad es que implementa una bitácora de todas las escrituras que se han realizado en el tiempo, de esta forma es capaz de retroceder o volver a un estado consistente.

Algunas desventajas que observamos es que es complicado de mantener y en caso de cambios es difícil introducir un nuevo comportamiento, además para observar los cambios hay que reiniciar nuevamente la aplicación. Esto es inaceptable cuando existen múltiples usuarios colaborando en algún documento.
El problema es que utiliza una coordinación de actualizaciones basada en una política optimista y esto deja fuera a muchas aplicaciones que no necesitan de todo este mecanismo complejo para operar. En el caso de un pizarrón colaborativo no es posible esperar tanto tiempo hasta que se actualice la información.

Problemas identificados.
Las aplicaciones que presentamos constituyen ejemplos donde el problema en común es programar dentro de la aplicación el código de coordinación para la actualización de la información. Por lo tanto, es una solución que obedece a intereses particulares. Es decir, no se promueve la reutilización y mantenimiento de código y además resulta complicado y costoso. Estos son los problemas que hemos identificado:

Mantenimiento.

- Porque se hace costoso y complicado corregir, modificar y agregar funcionalidad cuando el código está tan cohesionado, es decir posee muchas dependencias con el código de la aplicación.

Flexibilidad

- Porque muchos de esos sistemas no pueden trasladarse de plataforma sin realizar cambios o adaptaciones. Tampoco se puede cambiar o agregar mecanismos o proto-

colos de manera sencilla sin afectar en forma significativa a los demás módulos. Eso sin mencionar que el sistema original se diseñó con un propósito y modificarlo para realizar otra función distinta es absurdo y costoso.

Lo que buscamos es facilitar el desarrollo de estas aplicaciones creando una arquitectura por capas donde la capa de distribución permanezca separada del resto; de esta manera creamos un modelo flexible y dinámico donde los desarrolladores de aplicaciones colaborativas distribuidas son libres de crear y cambiar la funcionalidad de la capa de distribución sin necesidad de afectar a las demás capas. El software propuesto debe adaptarse a cambios de requerimientos y proporcionar herramientas que faciliten la tarea de diseñar e implementar aplicaciones cooperativas distribuidas. De esta manera el cambio de funcionalidad se enfocará en la adaptación de los metadatos de la entidad compartida y no en el código, así se libera al programador de esta tarea tan tediosa. Así la aplicación cooperativa distribuida extiende su funcionalidad a un costo muy bajo relativo al mantenimiento.

Enfoque del trabajo.
El enfoque de este trabajo se centra en la definición de nuevos paradigmas y mecanismos para administrar la distribución y actualización de la información compartida distribuida. En el dominio de las aplicaciones o sistemas cooperativos, la difusión de la producción cooperativa, la notificación y su integración con la producción de otro usuario en sus réplicas locales de entidades compartidas deben reflejarse de forma transparente al usuario. La única actividad de cada usuario consiste en concentrar todos los esfuerzos en la producción de trabajo cooperativo con otros usuarios, por eso la actividad de colaboración y comunicación colectiva coordenada puede ser eficiente evitando gastar esfuerzos en tareas de administración.
En los sistemas distribuidos es común el envío de información entre los nodos, por ello se debe tomar en cuenta las distancias, diferencia de horarios, y latencias en la red; por ello se buscan siempre soluciones que maximicen el ancho de banda, mejorando los algoritmos de transmisión etc., todo ello con el fin de mejorar la disponibilidad de los recursos a los diversos usuarios en la red.

1.0.1. Métodos de solución

Una técnica muy usada hoy en día es la replicación de los datos compartidos. Gracias a esta técnica los navegadores Web actuales pueden incrementar su eficiencia de respuesta a lo que el usuario solicita. En realidad lo que sucede es que el navegador guarda una copia de la página visitada de manera local en su caché, si la página no sufre cambios en periodos cortos de tiempo, el usuario puede consultar la página de manera local en vez de solicitarla al sitio Web; así se reduce en gran medida el tiempo de acceso porque permite seguir trabajando incluso si la página no esta actualizada. Sin embargo un problema potencial en este modelo es que el programa cliente debe hacer verificaciones periódicamente al sitio buscando nuevas actualizaciones, o de otra manera no va a presentar información actualizada si el sitio Web modifica su contenido. Esto resulta en la eventual pérdida de consistencia porque este usuario no necesariamente posee la misma información que los

demás.
Una solución probable es utilizar un enfoque centralizado para distribuir copias del contenido de un sitio remoto. Esto permite distribuir una copia de la misma información entre varios participantes para mejorar el desempeño y la disponibilidad de ese recurso, (ver figura 1.1), sin embargo se paga un gran costo porque mantener confiable y consistente esa información después que ha sido modificada por varios usuarios, es complicada y difícil. Con la finalidad de mejorar los recursos todos los sitios poseen una copia de la información y el servidor contiene la copia maestra, la cual es distribuida a todos sitios interesados por medio de la red.

Replicación optimista.

Para difundir de manera confiable y coherente esa información se debe utilizar un modelo lógico que defina una serie de reglas específicas que deben ser válidas y equivalentes en cualquier medio distribuido donde se apliquen. Con esto nos referimos a equipos en lugares distantes, diversos anchos de banda, distintos espacios de memoria, ciclos de reloj distintos, desconexiones, fallas de transmisión, ataques intencionales, corrupción de datos, etc.
Definir un orden de integración de las producciones puede resultar en una tarea sumamente complicada. Una manera de realizarlo es mediante la llamada *transformación de operaciones* [Li et al., 2000][Sun and Ellis, 1998][Lee et al., 2002], o *replicación optimista*, la cual consiste en establecer un orden causal para la generación y ejecución de operaciones.
Así una operación de escritura definida como O_1 seguida de una operación de borrado O_2, será vista en el mismo orden O_1O_2 por el sitio S_i, definiendo una función de transformación T aplicada a toda aquella operación que necesite ser modificada para llegar a ese mismo estado.

La transformación de operaciones utiliza vectores de tiempo para llevar un control sobre el momento en el cual se realizaron las operaciones, con la finalidad de poder realizar correcciones o aplicar funciones de transformación que llevarán a un estado consistente. Esta técnica no es necesaria en algunas aplicaciones como los mensajeros instantáneos; un simple protocolo social es suficiente. La transformación de operaciones se utiliza en algunas aplicaciones [Edwards, 1997] [Theimer et al., 1995] [Ellis and Gibbs, 1990] principalmente porque es una buena técnica que mantiene consistente la información a pesar de las diferencias en la latencia de la red. Sin embargo existen algunos riesgos en el uso de esta técnica, por ejemplo el problema de las réplicas divergentes, que resulta cuando el orden de las operaciones no es el mismo en todos los sitios. Este tipo de problemas es común cuando se ejecutan operaciones concurrentes en varios sitios, el reto es crear un algoritmo que sea capaz de minimizar este impacto y proveer una manera confiable de replicación de información. En el capítulo 2 profundizaremos en este tema, hablaremos de las mejoras, los algoritmos y las desventajas que la replicación optimista tiene que ofrecer.

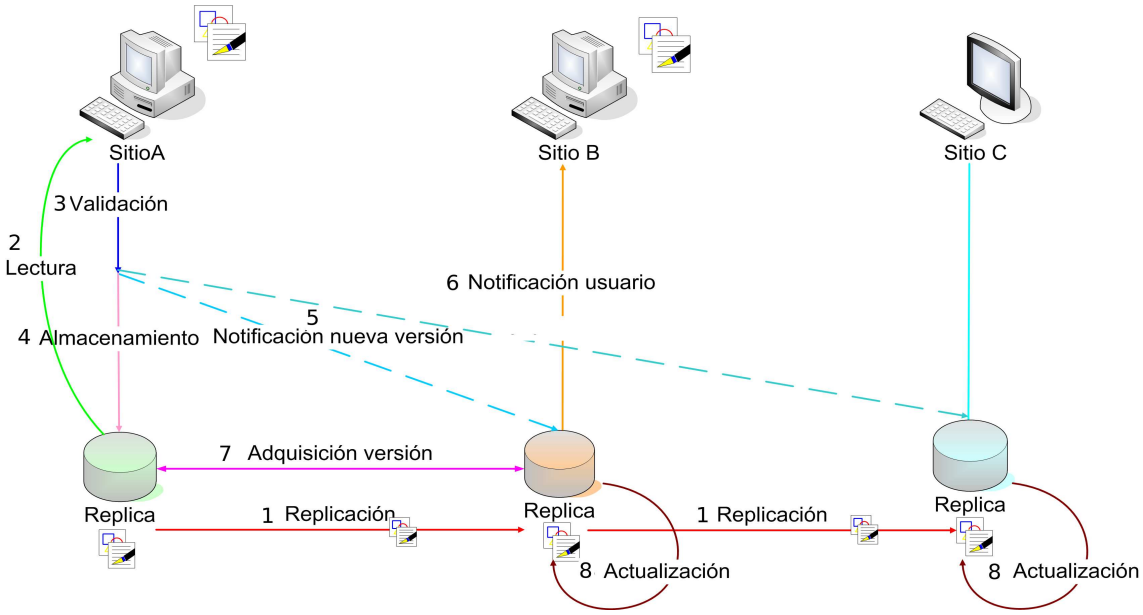

Figura 1.1: Replicación de información.

Replicación pesimista.

La replicación optimista como vimos, se basa en la idea de que eventualmente se llegará a un estado consistente, y evita usar mecanismos más estrictos para la modificación de información.

Sin embargo existe otra manera de asegurar la consistencia de la información y es a través de bloqueos para evitar la incoherencia en los datos compartidos; de esta manera, se impide a otros procesos obtener el recurso, ya que se mantiene reservado de manera exclusiva el acceso a la sección critica (semáforos en los sistemas operativos), y se mantiene el tiempo que transcurra la modificación. Esta manera de proceder pertenece a la replicación pesimista, ya que se considera de alta prioridad y fuente de una falla potencial una tarea de esta naturaleza.

También existen algunos protocolos que utilizan esta idea para asegurar la consistencia de los datos compartidos, por ejemplo los algoritmos *primary-copy* [Budhijara et al., 1993] y [Li and Li, 2005], eligen una copia de los datos y la etiquetan como la principal, de manera síncrona propagan la información a las copias secundarias. En caso de que la copia principal falle o se pierde (e.g. Falla del sistema que detiene la réplica primaria) se elige una nueva réplica primaria de las restantes secundarias.

Las administración de la consistencia de tipo pesimista presentan algunas desventajas [Saito and Shapiro, 2005], por ejemplo en el caso de una sincronización, existe el riesgo de permanecer en un bloqueo indefinido si no se realiza una administración adecuada. Es difícil de implementar la escalabilidad para redes en áreas amplias, porque al tratar de actualizar a los diferentes sitios de manera frecuente se produce un aumento de tiempo de respuesta y bajo rendimiento que estaría en proporción al número de sitios conecta-

dos. Por último, no provee del soporte necesario para el trabajo desconectado, es decir la única manera de trabajar y recibir actualizaciones es permanecer conectado a la red. Este tipo de replicación se utiliza en clientes de mensajería instantánea, agendas colaborativas, *PDA'S*, cachés.

Las aplicaciones colaborativas condicionan los tipos de replicación que necesitan. El problema es que en el estilo actual de desarrollo se crean soluciones particulares de coordinación de actualizaciones y de información para un sólo tipo de necesidad. Pero combinar varias soluciones dentro de un mismo sistema, es una herramienta poderosa que puede ser útil en el desarrollo de aplicaciones colaborativas distribuidas.

1.1. Objetivo general.

El objetivo de este trabajo es el desarrollo de un sistema de actualización de entidades replicadas, que trabaje como soporte a la colaboración en aplicaciones groupware, además de proveer la actualización automática de entidades replicadas.
Proveer los servicios necesarios para la distribución de información implementando políticas de actualización, notificación de entidades replicadas. Estas políticas que funcionan a manera de *drivers*, son diseñadas para ser intercambiables y sustituibles. Desde el punto de vista del diseño, requerimos una arquitectura flexible de módulos intercambiables que funciona como un controlador de dispositivos bajo de la capa de aplicación.

1.1.1. Objetivos particulares.

Los objetivos particulares son:

- Abstraer de la capa de aplicación la tarea de distribución de información, de esta forma se libera al programador de aplicaciones cooperativas de la innecesaria tarea de escribir código de distribución y a la aplicación de quedar atada a una sola política, la cual en muchas ocasiones se queda obsoleta, dados los constantes cambios en los requerimientos.

- Permitir un módulo flexible, que se encarga de la tarea de distribución información compartida, es decir, de las tareas de replicación, notificación y actualización de manera automática y se comunica con la capa de aplicación por medio de las interfaces diseñadas para recibir distintos parámetros, de acuerdo a las necesidades de la aplicación.

- Permitir el intercambio, adición y modificación de las políticas de replicación y notificación de las entidades en tiempo de ejecución, que permiten implementar la flexibilidad entre los módulos.

- Permitir el soporte al trabajo nómada, aislado y desconexo, por medio de una bitácora de operaciones que guarda registro de las operaciones aun permaneciendo desconectado de la red. Así mismo, se implementa como producto secundario la tolerancia a fallos.

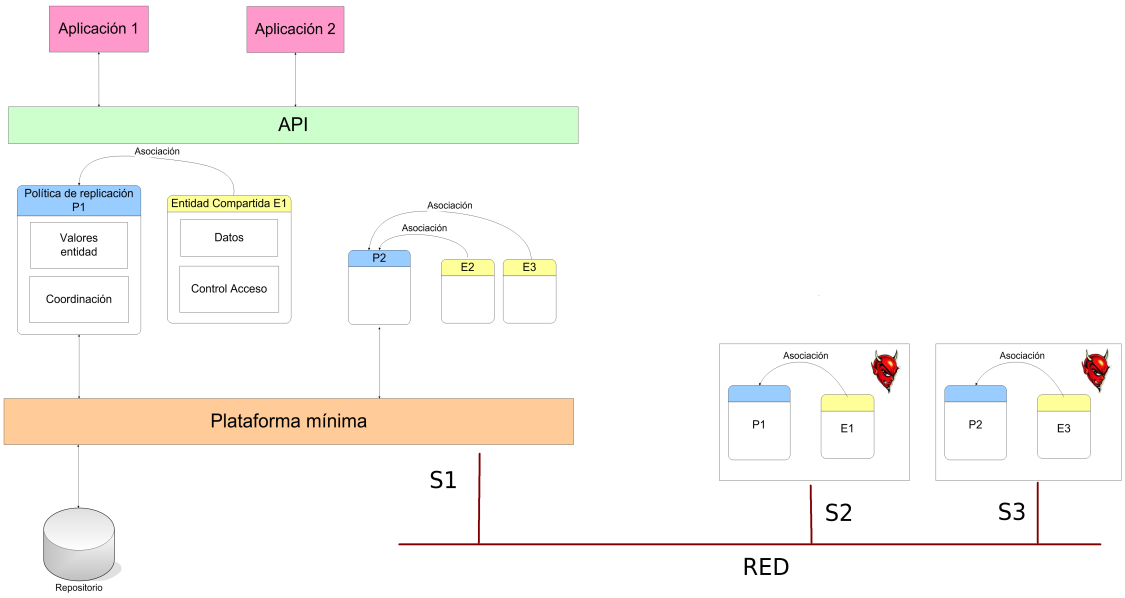

Figura 1.2: Plataforma Mínima

1.1.2. Funcionalidad básica

La idea básica es separar el código de la aplicación colaborativa de los componentes encargados de la comunicación, distribución y replicación de información. De esta forma, se crea un módulo independiente y la interacción con la aplicación se realizará a través de un único punto de contacto (interfaz).
Esto es necesario si queremos proveer un sistema flexible, extensible y escalable, porque al diseñarlo en base a componentes independientes, su funcionalidad puede ser reemplazada en cualquier momento de ser necesario. En la figura 1.2 podemos apreciar la funcionalidad que debe proveer este sistema, los componentes independientes (políticas de replicación) son asociados a entidades, funcionan de manera independiente y se comunican por medio de una interfaz.

La interfaz de interacción entre este sistema y la aplicación recibe la información necesaria para iniciar el proceso de replicación/actualización. Por medio de ella también se logra intercambiar información, por ejemplo notificaciones de actualizaciones, envíos de datos, resultados de búsquedas. La interfaz provee métodos de interacción como el siguiente en el que se muestra los datos requeridos para crear una nueva réplica:

```
createEntityReply( idEntity, user, cPolicy, ePolicy )
```

Los parámetros recibidos indican el usuario interesado en crear una réplica, el identificador, el método de control de concurrencia y el método de coordinación de actualización/replicación.
Así la aplicación puede liberarse de la tarea de programar el código de distribución y solamente debe crear un mecanismo que organice y administre el flujo de información que recibe y envía al sistema actualizador de entidades.

Los métodos de replicación/actualización y control de concurrencia elegidos pueden ser diseñados por algún programador de políticas de replicación/actualización y ser incluidos dentro del sistema de actualización. En el capítulo 3 mostraremos como podemos realizar estos anexos. Los métodos de replicación/actualización y control de concurrencia que pueden ser agregados en cualquier momento le dan diversidad de opciones a este sistema, además su diseño permite ser reemplazados incluso en tiempo de ejecución; además es posible cambiar su comportamiento por medio de archivos de configuración, los cuales evitan la necesidad de edición de código.

Todas estas características nos permiten un sistema flexible, dinámico y extensible que puede ser modificado, mejorado y configurado de manera sencilla.

1.2. Arquitectura.

La arquitectura propuesta que podemos observar en la figura 1.3 nos muestra las características anteriormente descritas. La capa superior indica la aplicación colaborativa que anteriormente realizaba la labor de distribución por si misma. Con esta propuesta se ha separado totalmente en un módulo independiente el cual realiza todas estas tareas.

Explicación de la figura: a) La aplicación delega las funciones de distribución a las capas inferiores. b) La administración de la información organiza las entidades e indica al SAE los métodos de actualización requeridos. c) El SAE es responsable por distribuir la información proporcionada por el administrador de entidades y el demonio realiza acciones automáticas correspondientes a la política de actualización.

La interacción es realizada por la interfaz hacia la capa administradora que ordena la información y posteriormente la entrega a la aplicación. En la capa del sistema de actualización podemos observar la actividad que se realiza en los procesos de replicación/actualización. Los procesos demonios realizan tareas automáticamente sin necesidad de que el sistema reciba una orden de actualización, son ellos los que permiten la actualización automática. Las capas muestran el nivel de los programadores responsables del desarrollo y administración, es decir mostramos una separación de responsabilidades.

Capa de aplicación.
Esta capa esta a cargo del desarrollador de aplicaciones colaborativas y es independiente de la capa de administración y de la capa de distribución. Por ello su desarrollo puede seguir cualquier patrón de diseño, o método de construcción. Sin embargo al no ser responsable por crear su propio código de distribución de información depende por completo de algún otro módulo que le permita realizar esta acción. Su desarrollo no es tema de este trabajo de tesis.

Capa de administración.
Se encarga de interpretar, administrar y organizar toda aquella información que es transmitida y recibida a través del sistema de actualización. Es quien da sentido al proceso de distribución, porque el sistema de actualización es en realidad un autómata que recibe

órdenes, no tiene capacidad para alterar la información. Su desarrollo no es tema de este trabajo de tesis.

Capa de distribución.
Recibe y entrega los datos que le indica la capa de administración a los sitios interesados. Su comportamiento esta determinado por las instrucciones que recibe y por la coordinación de replicación/actualización programada. No altera en ninguna forma los datos transmitidos y recibidos.

Como ayuda en el proceso de replicación existe un proceso demonio el cual permanece activo durante la actividad del sistema de actualización de entidades. Su función es ejecutar de manera automática una serie de operaciones de actualización dependiendo de la coordinación de actualización/replicación programada.

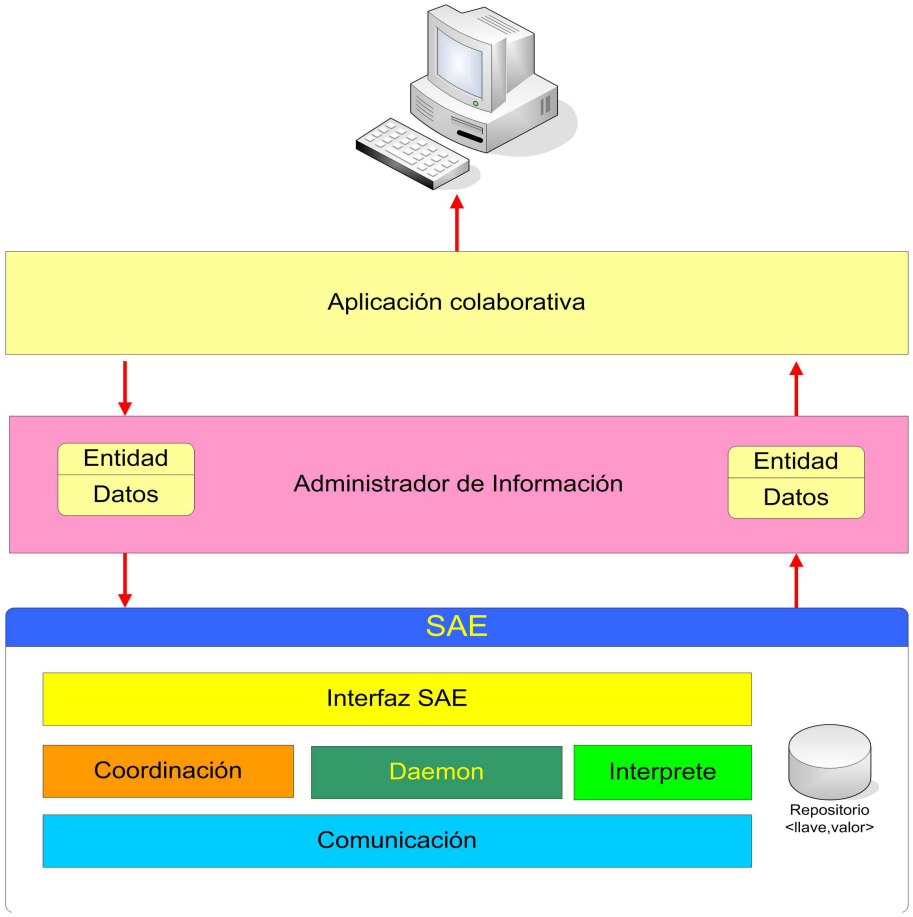

Figura 1.3: Arquitectura del sistema SAE.

Proveemos de transparencia de distribución y replicación al ocultar el funcionamiento de los módulos mediante el uso de interfaces. Así se logra una alta cohesión y bajo acoplamiento, porque los módulos del sistema permanecen independientes del resto y pueden

seguir reglas distintas de desarrollo e implementación, ya que los procesos externos sólo ven la interfaz del sistema.

1.3. Organización de la tesis.

La tesis esta organizada en cinco partes, en el capítulo dos se analizan y exploran los antecedentes y conceptos básicos sobre los cuales esta sustentado el trabajo. Se estudian las distintas arquitecturas de distribución: centralizadas, distribuidas e híbridas, mostrando algunas de sus ventajas y desventajas. Se muestran también un análisis de las políticas de replicación, con algunos enfoques pesimistas y optimistas [Patterson, 1995] [Sun and Ellis, 1998] [Li et al., 2000], políticas de notificación basadas en *push* y *pull*. El uso de marcas de tiempo y semántica para la resolución de conflictos y algunos patrones de diseño los cuales fueron utilizados para el diseño de los módulos del sistema.

En el capítulo tres, se muestra el diseño general del sistema, comenzando por justificar el diseño en base a las necesidades y los objetivos propuestos. Se describe a detalle el funcionamiento de cada uno de ellos y su interacción con los demás, así como el mecanismo de incursión de políticas de replicación, el funcionamiento del proceso demonio y el soporte al trabajo nómada.

En el capítulo cuatro, se presenta la implementación del sistema *SAE.*, las clases cliente y servidor que permiten la interacción remota, las clases de políticas de replicación y actualización, que permiten la realizar la coordinación, el repositorio de operaciones que permite el soporte al trabajo nómada. Además presentamos los mecanismos que nos permite ejecutar de manera remota las funciones necesarias para realizar el proceso de actualización. También se listan los métodos, clases, objetos, tablas etc., necesarios para el funcionamiento. En el capítulo cinco finalmente mostramos los resultados obtenidos, conclusiones y el trabajo futuro.

Capítulo 2

Marco Teórico

2.1. Antecedentes

Un sistema distribuido está conformado por componentes independientes ubicados en lugares distintos pero que trabajan juntos aparentando ser uno solo [Tanenbaum, 2007]. Esta definición nos induce a pensar que es sencillo crear un sistema distribuido, sin embargo lograr tal grado de cooperación no es evidente.

Existen muchos factores involucrados en una conversación entre dos personas tal que pueda realizarse una comunicación efectiva. Por ejemplo al estar presente en un mismo sitio a una misma hora, usar el mismo lenguaje, hablar por turnos, utilizar el mismo contexto de conversación etc. Estos factores son cruciales porque a falta de ellos la comunicación puede distorsionarse. En sistemas distribuidos ocurre lo mismo, solo que esos factores se traducen en protocolos de comunicación, sincronización, manejo de sesiones.
Implementar ese tipo de cooperación en el software tiene sus complicaciones: para establecer un enlace entre equipos remotos es necesario utilizar alguna fuente de energía: pulsos eléctricos, luz, microondas, ondas de radio etc., y un medio físico para trasmitir la información: cables de metal, aire, fibra óptica. Se puede hacer uso de múltiples tecnologías, pero todas ellas comparten un mismo problema; no son infalibles.
Si consideramos que una transmisión de datos entre dos computadoras puede terminar instantáneamente si cortamos la energía eléctrica, demostramos que trabajamos con una tecnología que es propensa a fallas y un medio de transmisión no fiable.
La tecnología siempre tiene puntos de falla y por lo tanto crear un sistema que se recupere de tales fallas es un punto importante en el desarrollo de sistemas distribuidos.
Existe otra complicación más que debemos tomar en cuenta: ¿Cómo mantener la coherencia en la información en una sesión de cooperación entre equipos?.
Actualmente se han hecho grandes esfuerzos para crear mecanismos que ayuden a preservar un estado estable y ordenado de información. Algunos de ellos proponen algoritmos para preservar el orden de las escrituras en sistemas Groupware [Schuckmann, 2005] [Ellis and Gibbs, 1989] [Ellis and Rein, 1988] [Roseman and Greenberg, 1996] utilizando bloqueos para asegurar la consistencia. Otros proponen un enfoque en el que no se necesite de una sincronización explícita para llevar a cabo una cooperación [Vahid et al., 1994] [Miller and Ferguson, 2005] [Nuno et al., 2000]. Todos estos sistemas proponen soluciones

para un tipo específico de problema, buscar una propuesta genérica que satisfaga a todos los enfoques y necesidades es un trabajo más complejo.

El proceso que estas aplicaciones utilizan para la resolución de conflictos es a menudo complejo y difícil de entender y más aun de implementar, es por esta razón que aun se involucra al usuario para resolver conflictos o simplemente que establezca el protocolo social, el cual lleve un orden a una sesión de cooperación. El problema es que involucrar al usuario en el proceso disminuye la productividad, es tedioso y no es placentero de realizar.

2.1.1. Arquitecturas

Los sistemas distribuidos trabajan como componentes autónomos dispersos por la red, los cuales se comunican unos con otros utilizando algún protocolo de comunicación para preservar un orden.

Las arquitecturas de red nos ayudan a aligerar la complejidad inherente a este tipo de sistemas mediante la separación de los componentes en diversas capas [Tanenbaum, 2007], por ejemplo:

- **Arquitectura por capas.** Aquí los componentes se estructuran en capas de tal manera que la capa superior se comunica con la capa subyacente, pero no con el resto de las capas, el control fluye de capa en capa (ver figura 2.3).
- **Arquitectura basada en objetos.** Cada componente se comunica a través de llamadas a procedimientos remotos (RPC). Esta arquitectura se utiliza para crear sistemas cliente-servidor (ver figura 2.4).
- **Arquitectura centrada en los datos.** Los componentes se comunican entre sí por medio de un repositorio compartido donde las comunicaciones se realizan a través de archivos (ver figura 2.3).
- **Arquitectura basada en eventos.** Los procesos publican eventos que sólo son accesibles vía una suscripción y solamente así son visibles por otros procesos. La ventaja es el bajo acoplamiento porque no necesitan referirse mutuamente de manera explícita (ver figura 2.4).

Estas arquitecturas presentan características de comunicación que son necesarias para satisfacer necesidades de alguna aplicación en particular. Por ejemplo, una arquitectura basada en datos es útil en una base de datos replicada. Todos estos mecanismos de comunicación buscan proveer de transparencia de distribución; es decir, ocultar los procedimientos necesarios que permiten la comunicación con un sistema remoto. Ahora veremos como son organizados los diversos componentes de un sistema distribuido, lo cual trae consigo diversos estilos de diseño de software colaborativo.

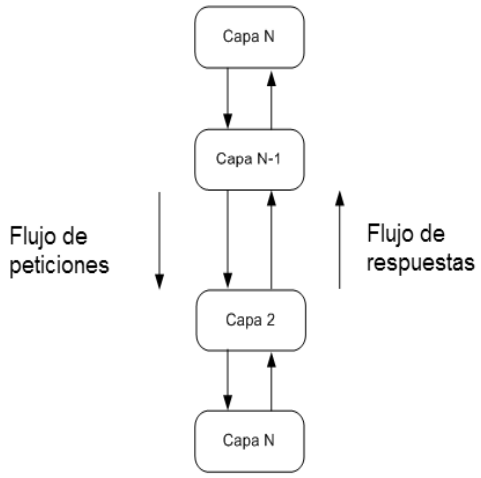

Figura 2.1: Arquitectura por capas

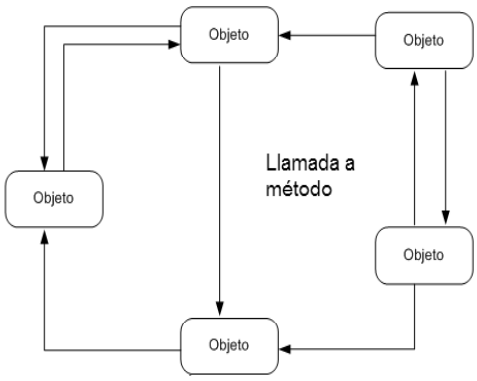

Figura 2.2: Arquitectura basada en objetos

Arquitectura centralizada

Esta arquitectura es utilizada mayormente en aplicaciones de tipo cliente/servidor, se basa en el envío y recepción de peticiones vía un enlace. El servidor es un proceso que atiende las peticiones que recibe del proceso cliente, y es el mecanismo simple y sencillo que utilizan los servidores en la Web y por lo tanto el más difundido (ver figura 2.5). El servidor centralizado ofrece varios servicios los cuales son accesibles a través de una *URL*, el cliente la usa realizar el enlace hacia el servidor y una vez establecido lo utiliza para enviar y recibir mensajes. Los clientes usan el enlace de comunicación con un enfoque síncrono, es decir que necesitan establecer una comunicación punto a punto a un mismo tiempo, de tal manera que pueden recibir peticiones y recibir respuestas en ese momento.

Este esquema sincrónico posee algunas desventajas por ejemplo, si un cliente ha realizado el enlace con algún servidor remoto, éste no es capaz de saber si el servidor realmente puede atender su petición o si permanece fuera de línea. No existe la posibilidad de conocer sobre el estado del servidor ni de los clientes, por lo que se puede crear un estado de espera indefinida. Pero incluso puede ser peor: el cliente puede interpretar esos retrasos como errores de transmisión y puede reenviar la misma operación varias veces, lo cual resulta en envíos innecesarios y saturación de la red.

En el contexto del trabajo cooperativo distribuido este tipo de arquitecturas a pesar de sus variaciones, no proveen de manera satisfactoria las necesidades que requerimos: es decir, por un lado la disponibilidad de los recursos no se garantiza, ni posee tolerancia a fallos y tampoco tiene soporte a la escalabilidad. En un servicio compartido cuya responsabilidad recae en un servidor centralizado, no solo se vería comprometida la disponibilidad de recursos, debido a las numerosas peticiones por parte de los clientes; también

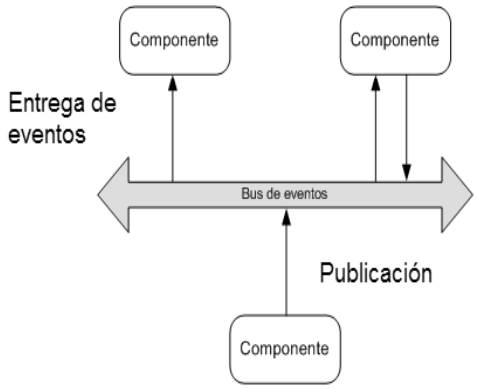

Figura 2.3: Arquitectura centrada en los datos

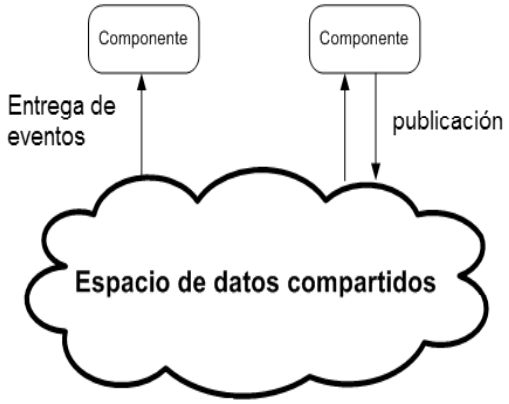

Figura 2.4: Arquitectura basada en eventos

estaría en riesgo el trabajo de todos, porque si el servidor queda fuera de línea dejaría a los participantes bloqueados. Por lo tanto no es del todo satisfactoria para proveer de manera eficiente el trabajo cooperativo distribuido.

Arquitectura Descentralizada.

Es una arquitectura completamente replicada, es decir que los componentes de tratamiento de la información se encuentran en todos los sitios, no hay servidores puesto que cada sitio se comporta igual que los demás. Entre las ventajas se encuentran que cada servidor o cliente trabaja con su porción de datos compartidos aunque como veremos más adelante, esto presenta en sí mismo algunas desventajas. Cada sitio ó peer [Lua et al., 2005] posee un comportamiento simétrico en donde los nodos están formados por procesos que se comunican entre sí por medio de un canal de comunicación, (e.g. a través de TCP), una arquitectura distribuida típicamente posee el siguiente esquema 2.6. Otra ventaja en el uso de esta arquitectura es que el tratamiento a la información se realiza de manera local en todos los sitios, con lo cual se mejora substancialmente el rendimiento, independientemente de la latencia de la red, evitando la congestión que sufre la arquitectura centralizada. Además el tratamiento de la información en cada sitio ofrece un beneficio adicional, puede permitir el trabajo fuera de línea. Las desventajas surgen cuando se considera que administrar las réplicas distribuidas requiere un mecanismo complejo y además se necesita de un servicio eficiente de sincronización. La arquitectura descentralizada ofrece dos esquemas de sincronización: redes estructuradas y no estructuradas [Aberer and Hauswirth, 2005].

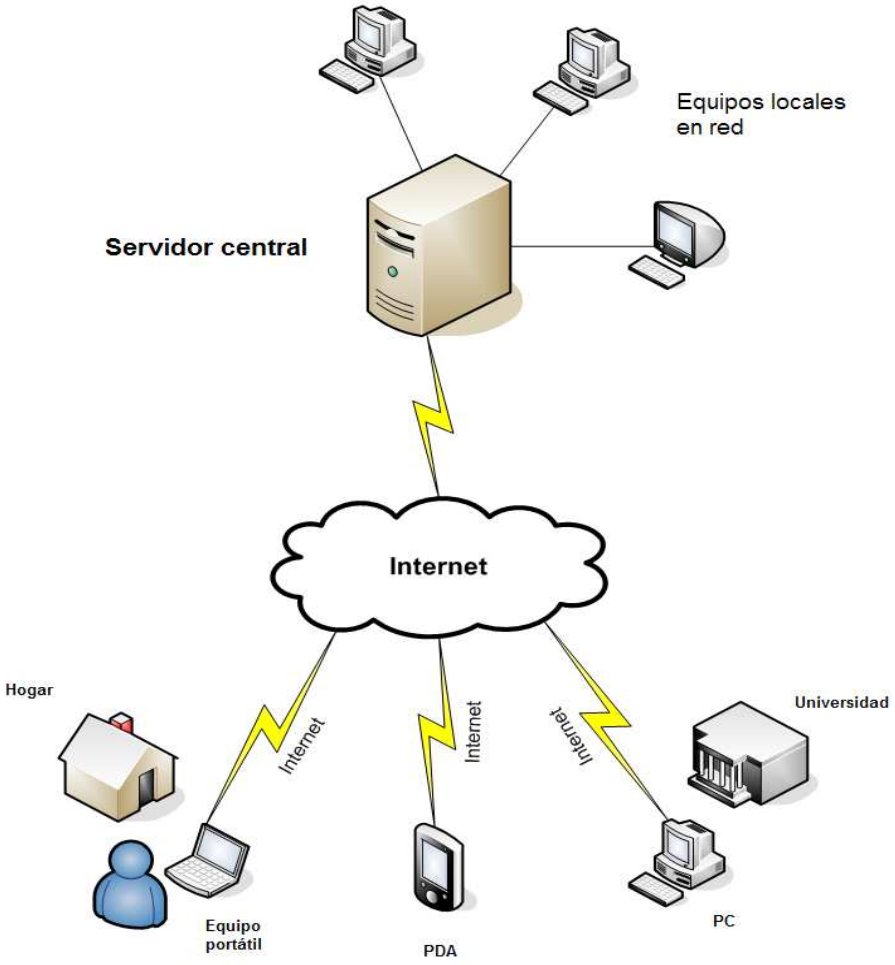

Figura 2.5: Servidor Centralizado

Arquitectura peer-to-peer estructurada

Las arquitecturas peer-to-peer estructuradas utilizan un proceso determinista denominado *DHT* o Distributed Hash Table por sus siglas en ingles, el cual asigna un identificador a cada nodo en la red para ubicarlo de manera rápida, generalmente es un identificador de 128-bits. Se realiza un mapeo de la llave de un elemento de datos al identificador de un nodo basado en una distancia métrica, entonces se regresa la dirección de red del nodo. Por ello, las búsquedas de los nodos son muy rápidas, del orden de $O(log(N))$ en algunos casos. A continuación presentamos algunos ejemplos de *P2P* estructurados extraídos del análisis de [Lua et al., 2005].

Tapestry.
Tapestry implementa una red *P2P* estructurada que asigna llaves a los datos y organiza los peers en un grafo en el que se mapea cada llave de datos hacia un peer.

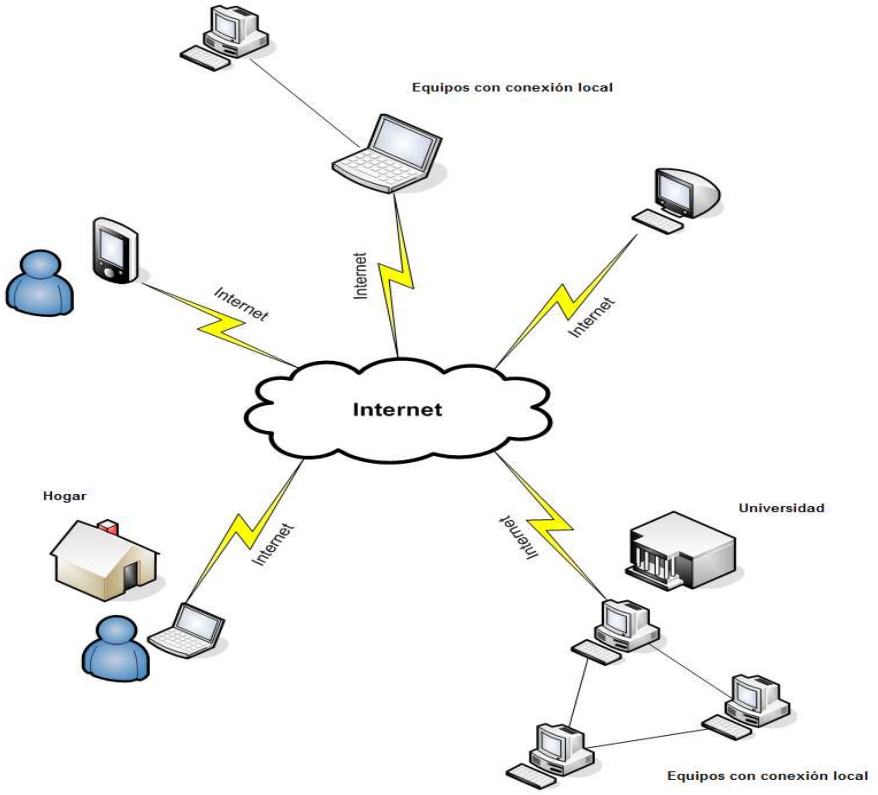

Figura 2.6: Red Descentralizada

- Emplea un descentralismo aleatorio para lograr distribución de carga y encaminamiento local. Se utiliza una técnica de búsqueda distribuida basada en un algoritmo propuesto por [Plaxton et al., 1997] con algunos mecanismos adicionales que proveen de escalabilidad, disponibilidad y adaptación en caso de falla.

- Utiliza una estructura de datos distribuida, la cual usa para la localización de objetos de datos que se conectan a un nodo raíz, también utiliza múltiples raíces para cada objeto de datos, con lo cual previene un simple punto de falla.

- Aunado a lo anterior mantiene un registro de la ubicación de todas las réplicas e incrementa la flexibilidad semántica al permitir, a nivel aplicación, elegir de un conjunto de réplicas a aquellas que correspondan a un criterio de búsqueda.

- La manera en que se maneja el problema del punto de falla es asignar múltiples raíces hacia el objeto de datos y utilizando lo que se conoce como encaminamiento sustituto, se inserta información de ubicaciones en Tapestry seleccionando peers raíces de manera incremental durante el proceso de publicación.

- Una búsqueda de un nodo se realiza de la manera siguiente: se asume que el peer X existe y se le puede localizar mediante un mensaje encaminado hacia él. La ruta hacia un identificador inexistente encontrará entradas vacías a lo largo del camino e intenta encontrar un enlace que pueda comportarse como una alternativa al enlace deseado, es decir, aquel que esté asociado con el dígito X. El encaminamiento termina cuando un mapa es localizado donde el único camino no vacío pertenece al peer actual, entonces ese peer es designado como el peer substituto.

- Un punto importante que debemos tomar en cuenta es que se mantiene un caché para recuperación de fallas, llevando un control sobre aquellos sitios que han dejado de responder y con las notificaciones periódicas para detectar enlaces y fallas en servidores o re-enrutamientos. Durante una falla cada entrada en el mapa de nodos mantiene dos vecinos de respaldo además del vecino cercano/primario como medida extra de seguridad.

- Es interesante la manera en como se asume el ruteo de los datos e incluso la semántica que se pueda dotar a las búsquedas de los nodos, aunque esto conlleva generar algunas soluciones particulares, como replantearse la estructura de la entidad compartida, aumentando ciertos parámetros se puede ganar en eficiencia de búsqueda.

Viceroy

Es otra red *P2P* descentralizada que esta diseñada para controlar la ubicación, búsqueda de datos y recursos. Se utiliza *consistent hashing* [Karger, 1997] para distribuir datos tal que estén balanceados a través del conjunto de servidores y residentes en los servidores al unirse o dejar la red.

- Utiliza *DHT* para el manejo de la distribución de datos entre un cambiante conjunto de servidores, de ésta manera los peers pueden tener contacto con cualquier servidor en la red para localizar cualquier recurso almacenado por nombre.

- Utiliza enlaces entre sucesores y predecesores formando un anillo lógico para cortas distancias por medio de niveles, los cuales albergan a los nodos de acuerdo a la llave y su proximidad con el número de nivel. Cuando N peers se encuentran en modo operacional, uno de los niveles $\log N$ es seleccionado con una probabilidad cercana.

- Usa un algoritmo para conectar a todos los peers, que va desde los niveles más bajos a los más altos del árbol, una búsqueda recursiva de un nodo toma $O(\log N)$ pasos donde N es el número de Nodos. Es bastante razonable tomando en cuenta un número de nodos grande y en crecimiento.

Desventajas con las arquitecturas *P2P*.

Aunque las redes *P2P* estructuradas poseen ventajas dado que crean un grafo delineado y cada nodo es rastreable, poseen algunas desventajas. Se requiere conocimiento global a fin de construir el recubrimiento de la red. El peer raíz es el punto de quiebre del sistema, y así se puede generar la congestión del punto de enlace. Los sistemas basados en *DHT* poseen algunos problemas en términos de latencias en búsquedas de objetos, por cada

salto los peers encaminan un mensaje hacia el siguiente intermediario, que puede estar localizado muy lejos respetando la topología física de la red *IP* subyacente. *DHT* además, asume que todos los peers participan de manera ecuánime en el hosting, publicando objetos de datos o su información de ubicación, lo que puede inducir a cuellos de botella en peers de poca capacidad.

Finalmente *DHT* garantiza encontrar una llave si es que ésta existe, pero no captura las relaciones entre objetos y su contenido. También existen problemas asociados al desempeño, por ejemplo, *timeouts* en búsquedas, el tiempo de reacción de los *peers* en una recuperación periódica y también la elección de vecinos cercanos. Estos problemas se pueden disminuir un poco reduciendo los retardos, midiendo los *RTT* en un número de peers cuando se construyen estas tablas. Con ésta solución se reduce el tiempo de encaminamiento.

En el caso de las redes *P2P* con *DHT*, estos sistemas son susceptibles de brechas de seguridad en ataques de peers maliciosos. Por ejemplo, un peer malicioso puede retornar información incorrecta en caso de una búsqueda, se puede contrarrestar este tipo de ataque utilizando algunas técnicas cartográficas para verificar la autenticidad de esa información. El hecho de que los peers no sigan de manera adecuada el protocolo de búsqueda puede contribuir a que algunos peers maliciosos intercepten información.
Algunos principios de diseño se necesitan para crear defensas contra este tipo de fallas, principios que se pueden establecer en la forma de invariantes verificables del sistema para las peticiones de búsqueda, asignación de *NodoID*, selección de Peers para encaminamiento, evitar la creación de puntos únicos de responsabilidad.
Algunas primitivas de diseño para proveer un encaminamiento seguro van desde asignación de identificadores, mantenimiento a tablas de ruteo hasta asegurar la entrega de mensajes.

Un punto importante es el hecho de que muchos modelos computacionales *P2P* de gran reputación evalúan la fiabilidad del comportamiento a través de retroalimentación y mecanismos de interacción, lo cual permite una adaptación a diversas circunstancias.

Peer to peer no estructurado.

Las arquitecturas peer-to-peer no estructuradas, en lugar de establecer un método determinístico para ubicar recursos, usan algoritmos aleatorios que construyen una topología de red. Cada participante posee una lista de vecinos con los cuales tiene contacto. Esas listas son creadas de acuerdo a ciertos criterios, algunos de ellos son simplemente por elección aleatoria.

Características.

Así mismo los elementos de datos están almacenados de manera aleatoria entre los nodos de la red. Por lo tanto la única forma de encontrar algo es inundando la red con consultas de búsqueda. Estas listas de nodos se conocen como listas parciales [Jelasity et al, 2004] y existen muchas formas de construirlas.

Aquí los nodos intercambian las entradas de dichas listas con cierta regularidad. Esas entradas son las ubicaciones de los nodos vecinos y tienen cierta duración asociada, es decir, indican la edad del enlace a ese nodo. La manera más común de crear estas vistas es intercambiar entradas entre los nodos contactados o simplemente deshacerse de las entradas viejas, de ésta forma se mantienen las listas de nodos actualizadas y se van eliminando los nodos que no responden. Al momento de abandonar la red un nodo usualmente termina su tarea y se va, cuando los nodos hagan referencia a ese nodo que no responde, ellos lo quitarán de su vista parcial y escogerán otro.

Riesgos.
Existen también riesgos al escoger este tipo de estrategias, principalmente es manera injusta de elección que con el tiempo favorece a los nodos con alto grado de entrada, dado que se eliminan a los nodos con entradas viejas.
Como punto interesante, es posible agrupar ciertos nodos en base a proximidad semántica, esto significa poder crear redes que permiten usar algoritmos de búsqueda muy eficientes en base a atributos. Una derivación de este esquema son los llamados *superpeers* [Yang and Garcia-Molina, 2003], a medida que la red crece se torna complicado encontrar recursos específicos, por lo que en una red p2p tradicional se acumulan las peticiones en cada nodo a medida que realizan búsquedas por la red.
En general las redes *P2P* ofrecen un buen método flexible para que los nodos sean agregados o borrados de dicha red.
A continuación examinamos un conjunto de redes *peer-to-peer* que ponen en evidencia las ventajas y desventajas de este modelo.

Freenet.
Es una red *P2P* "adaptante"lo cual significa que cada peer contiene las direcciones de otros peers y una referencia sobre el contenido que mantienen cada uno de ellos, de ésta manera tienen la habilidad de mantener un registro actualizado basado en el espacio máximo disponible controlado por el administrador de red, así también provee de seguridad contra peers maliciosos.
Puntos importantes.

- Las peticiones por llaves son enviadas de un peer a otro a través de una cadena de peticiones a sitios de tipo proxy, en los cuales los peers toman una decisión acerca de la próxima ubicación a la cual enviarán la siguiente petición. Esto permite compartir espacio de disco no utilizado; una especie de extensión lógica a su propio dispositivo de almacenamiento. El algoritmo utilizado se ajusta en el tiempo para hacer eficiente el desempeño ya que se utiliza la información de contenido de sus vecinos y con esto se reorganiza constantemente la distribución de los peers.

- Cada petición utiliza un *HTL(hops-to-live)*, que es un contador aplicado a la petición, el cual es decrementado en cada peer para evitar cadenas infinitas, es similar a (*TTL*) de *IP*. Cada identificador posee un identificador único con la finalidad de evitar ciclos y rechazar peticiones ya atendidas.

- No existe punto de falla, dado que todos los peers se comportan de manera idéntica:

no hay privilegios, no hay jerarquización etc. Esto hace al sistema *P2P* altamente escalable y aumenta su desempeño, dando como resultado una ruta constante de enrutado mientras los peers se unen y dejan la red sobrepuesta.

- La idea de utilizar un contador de saltos es buena estrategia en un sistema en el cual no se sabe con exactitud el número de nodos y por ello la ubicación precisa de alguno de ellos, también llevar un registro del contenido de cada nodo evitando así el pasar dos veces por un mismo lugar.

Gnutella

Este proyecto consiste principalmente en un protocolo descentralizado para búsquedas distribuidas en topología planas de peers *(servents)*. Lo interesante acerca de este trabajo es el modelo para ubicación y extracción de información. No provee control sobre la topología de la red o la ubicación de los archivos, así que los *peer* se unen con unas reglas sencillas.

Puntos importantes.

- Esta red es organizada por peers que se incorporan a la red siguiendo reglas sencillas. El resultado es una topología con ciertas propiedades pero el albergue de los datos no se basa en ningún conocimiento de la topología por lo que para encontrar un item se debe proceder preguntando a los nodos vecinos, y el método más común para hacer esto es la inundación.

- Los *peers* proveen servicios de cliente/servidor a través de una interfaz que utilizan para emitir búsquedas y recibir resultados, al mismo tiempo reciben peticiones de búsqueda de otros peers para verificar la correspondencia con sus datos locales y responder de manera adecuada. Esta característica es útil en caso de falla puesto que si un integrante falla el contenido deseado aun puede ser localizado ya que seguro se encontrará la ubicación en otro peer.

- Con el manejo de pocos mensajes puede ser capaz de realizar búsquedas. Básicamente maneja tres tipos de mensajes: membresía, búsqueda o transferencia. Con estos tipos de mensajes puede iniciar una red, emitir una búsqueda y transferir contenido, así que podemos tomar ésta idea como parte de nuestro diseño, porque cada mensaje incluye un campo TTL (saltos realizados) que evita reenvíos o re-ejecuciones.

Las redes *P2P* no estructuradas carecen de un método determinista para la localización de sus recursos, por lo tanto usan un método alterno para organizar su estructura y realizar búsquedas de miembros, contenido y recursos. Tales métodos pueden ser útiles en redes donde el número de nodos es desconocido y la topología de red dinámica, pero sobre todo son capaces de sobreponerse a fallas, lo cual es una característica muy útil.
Poseen algunas desventajas como ser propensas a ataques, pero implementando un estricto control sobre la identidad de cada individuo, mantener el envío de mensajes controlado puede minimizarse este hecho. A continuación veremos un modelo que combina las ventajas de las redes *P2P* estructuradas y las no estructuradas: las redes híbridas.

Arquitectura Híbrida
Este tipo de arquitectura se utiliza en sistemas distribuidos colaborativos principalmente por las ventajas que brinda. El sistema se inicia con un esquema cliente/servidor y posteriormente la colaboración se realiza de manera distribuida. La razón es que es relativamente sencillo ubicar recursos de ésta manera y tener un control centralizado de los cambios y accesos. Existen algunas aplicaciones que utilizan este tipo de arquitectura debido a estas características:

Bitorrent.
Bittorrent es un sistema *peer-to-peer* que utiliza una arquitectura híbrida la cual consiste en un servidor centralizado que funciona como localizador de recursos y que a su vez es localizable vía un archivo *.torrent*.

- Este archivo contiene la información necesaria para ubicar los recursos que se desean acceder. En realidad contiene la dirección del servidor que posee la lista de sitios que poseen la totalidad o parte del recurso.
- Una vez que se obtiene el archivo *.torrent*, el peer se vuelve activo y envía peticiones a los sitios de su lista de distribución para descargar porciones aleatorias del recurso solicitado, así mismo envía porciones locales del archivo hacia otros sitios solicitantes (ver figura 2.7).
- El punto débil de ésta arquitectura es el servidor centralizado porque es el único punto de acceso a las ubicaciones de los recursos. Por ello si se pierde la conexión con el servidor el sistema entero es bloqueado.

Sin embargo aun con estas inconveniencias es atractivo utilizar estas arquitecturas porque disminuyen en gran medida la implementación de mecanismos de búsquedas de recursos, de inconsistencias y al mismo tiempo poseen las ventajas de la disponibilidad de recursos de las redes descentralizadas.

2.1.2. Sistema groupware distribuido

En la actualidad la interacción entre un usuario y un sistema ya no es una tarea aislada, es decir, no se trata solamente de intercambiar información entre una pantalla y una persona, si no del intercambio de información con múltiples usuarios.

Analogía.
En un proceso creativo enfocado al diseño de un automóvil, varias personas están involucradas, estas personas se reúnen en algún sitio y generan una lluvia de ideas, la cual finaliza con un producto. El trabajo final es resultado de la cooperación de varias personas que aportaron ideas de manera individual. Como este ejemplo podemos pensar en muchas otras situaciones en las que las personas trabajan conjuntamente para realizar una labor en común, personas que no sólo trabajan en un mismo lugar, sino personas distribuidas alrededor del mundo.
Gran parte de esas personas utilizan actualmente las computadoras como herramienta

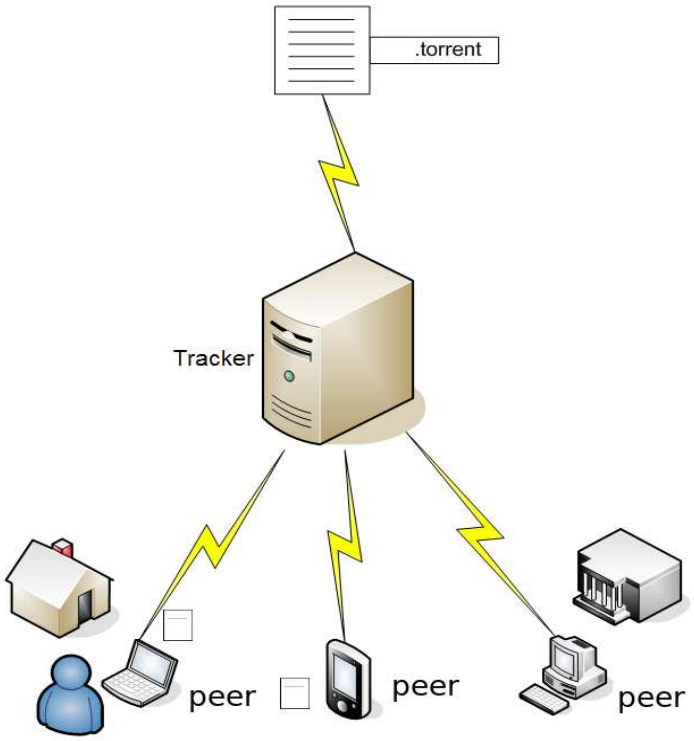

Figura 2.7: Funcionamiento de Bittorrent

creativa, que ayuda en ese proceso productivo.
Por ello surge la necesidad de proveer software que ayude en esa labor de cooperación de manera eficiente utilizando la tecnología existente o desarrollando nuevas ideas.

Características.
El software colaborativo ha sido desarrollado desde hace algunos años [Ellis and Gibbs, 1999] definieron por primera vez el término *Groupware* para referirse a este tipo de sistemas. La definición que propusieron fue: *Groupware: Sistemas de cómputo que dan soporte a grupos de personas involucradas en una tarea común y que proveen de una interfaz para un ambiente compartido*, que implica mecanismos, procedimientos y protocolos que en conjunto logran que esa idea sea una realidad.

Sin embargo dada la variedad de tipos de organización que las personas pueden establecer para trabajar en conjunto, no existe una manera única de proveer herramientas genéricas que ayuden a realizar ésta labor de manera eficiente. Por ello existe una clasificación, la cual utiliza como base el espacio y el tiempo de colaboración, para orientar de mejor manera el desarrollo de aplicaciones colaborativas, podemos observar esta clasificación en la figura 2.8.

El diseño de una aplicación *Groupware* obedece los requerimientos o necesidades del grupo de personas que lo va a utilizar, por ello realizar un buen diseño y facilitar el uso de la aplicación es bastante recomendable.

Sin embargo no solo una aplicación groupware debe ser creada en base al usuario que lo va a utilizar, sino también las herramientas necesarias para crear aplicaciones colabo-

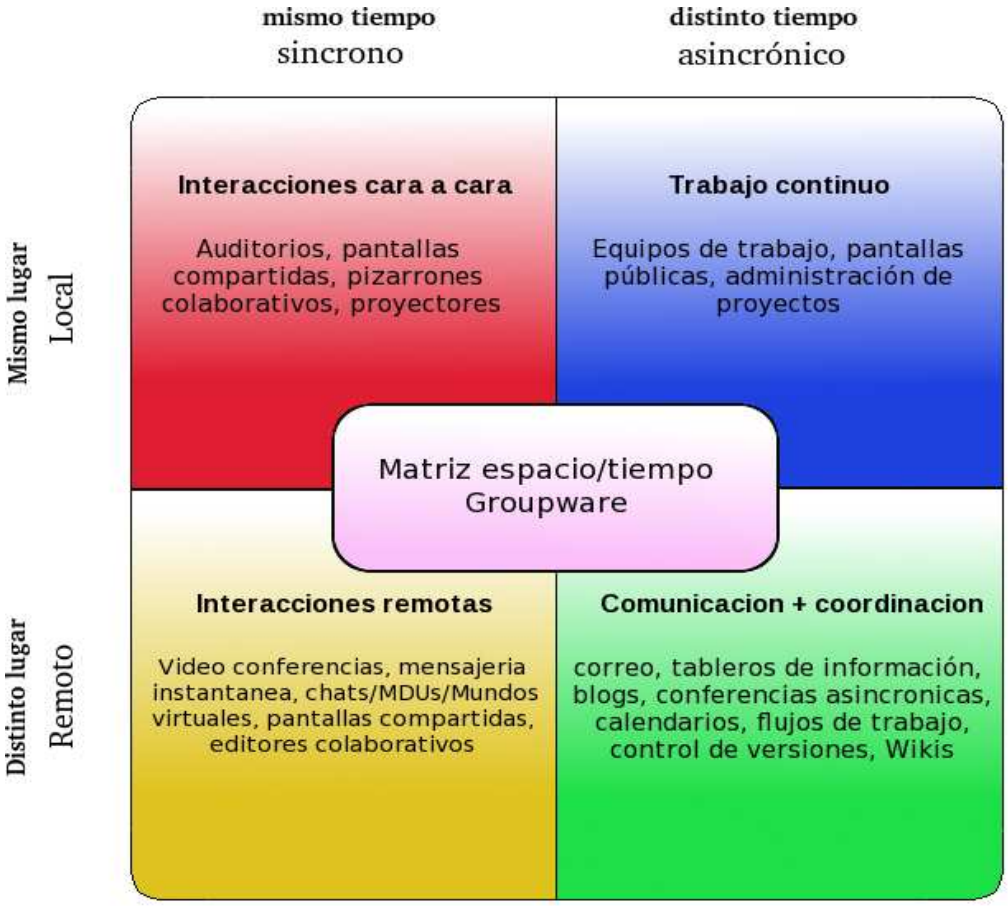

Figura 2.8: Matriz Groupware

rativas. Facilitar la labor de desarrollo de estas aplicaciones contribuye a su aceptación y fomenta su uso.

Ante todo, un sistema groupware colaborativo es un sistema distribuido, por lo tanto posee algunas características importantes: por ejemplo, mecanismos para envío y recepción de mensajes, control de concurrencia, control de acceso, protocolos de transferencia, de ubicación de recursos, de consistencia, replicación, sincronización y tolerancia a fallas.

2.1.3. Comunicación.

En cuanto al servicio de comunicación hay ciertas restricciones que se adoptan, por el simple hecho de mantener orden en las operaciones, estas reglas son establecidas ya sea por el sistema (protocolos) o por parte de los usuarios (protocolos sociales). Estas reglas permiten un acceso concurrente ordenado de los objetos compartidos, mediante los cuales se pueden realizar con seguridad múltiples accesos a recursos y con ello se asegura el correcto funcionamiento de las operaciones de lectura y escritura. Todo sistema distribuido implementa estos mecanismos conocidos como control de concurrencia, gracias

a esto se pueden realizar múltiples accesos a recursos compartidos y remotos asegurando la coherencia de la información.

Como los usuarios utilizan los recursos de manera concurrente es natural la existencia de conflictos, por ello el sistema groupware debe considerar algunos principios:

- **Un sistema de colaboración sensible a las interacciones**. Denota la capacidad de respuesta en corto tiempo y notificaciones rápidas, con ello se asegura una interacción adecuada.

- **Interface de grupo.** Implementación plena de *WYSIWIS (what you see is what i see)*, es decir, debe haber coherencia en la información, todos los participantes deben tener una imagen idéntica de los datos.

- **Distribución de red en área amplia.** Una función importante es permitir la colaboracion no solo de manera local, si no desde cualquier punto geográfico,

- **Replicación de los datos compartidos.** Al necesitar una capacidad de respuesta inmediata es necesario tener una copia local de la información en cada sitio participante, con ello se evita una potencial catástrofe si la información no es replicada y falla la comunicación.

- **Robustez contra desperfectos.** Recuperarse de cualquier anomalía inesperada, conexiones fallidas, agregar un participante nuevo en sesiones abiertas, etc.

Consistencia de datos compartidos.

En una sesión de colaboración existen múltiples actualizaciones de datos que todos los participantes reciben, el periodo en el que esto sucede depende del tipo de aplicación *Groupware* colaborativo. Estas actualizaciones de datos deben llevarse a cabo en un orden y tiempo especifico en que todos los participantes puedan obtener satisfactoriamente la misma información.

Existen varios tipos de control de concurrencia que proporcionan un servicio dedicado a las necesidades de la aplicación. Sin embargo es claro que, al ser un sistema que permite compartir recursos, existen muchas posibilidades de fallas y conflictos. Una manera de hacer frente a estos problemas es mediante un control sobre los recursos compartidos, de ésta forma los usuarios y procesos ven restringido en cierta medida el acceso y de ésta manera se asegura parcial o totalmente la coherencia en la información. Aquí examinamos algunas soluciones para asegurar la coherencia de un recurso compartido:

- **Bloqueo simple.** Se bloquea el recurso por medio de candados, a fin de que sólo un usuario a la vez pueda tener acceso a dicho recurso. Aunque ésta solución puede llevar a bloqueos infinitos o perjudicar el tiempo de acceso si es que muchos usuarios quieren acceder al mismo recurso.

- **Mecanismos transaccionales.** Estas operaciones ayudan a que el control de concurrencia sea eficiente, ya sea aplicando candados o utilizando pilas que pueden retroceder o deshacer operaciones realizadas en detrimento claro del tiempo de respuesta. Se debe tener cuidado de no sobreexplotar estos mecanismos porque se corre el riesgo de ir en contra de la simplicidad y/o desviarse del objetivo principal el cual es la cooperación.

- **Control Centralizado.** Se presupone que todos los recursos se encuentran replicados y este tipo de control de concurrencia se encuentra ubicado en un servidor centralizado el cual se encarga de coordinar todas las actualizaciones que les son solicitadas. El problema es obviamente el cuello de botella constituido en el principal punto de ruptura del sistema. Además de que se pierde capacidad de respuesta debido a que la acción solicitada ocupa un RTT (Round Trip Time) en vez de ser inmediata.

- **Ejecución Reversible.** Esta técnica es utilizada cuando se necesita deshacer operaciones con el fin de rehacerlas de manera correcta. Se auxilia de marcas de tiempo para tal efecto, cuando no coinciden en alguna operación se deshacen dichas acciones y se rehacen en forma correcta. Para que este mecanismo funcione es necesario el uso de un historial el cual se almacena de manera local.

- **Transformación de operaciones.** Se basa en la utilización del envío de vectores de estado junto con la operación realizada. El objetivo es que cada sitio que posee su propio vector de estado compara este con el vector recibido en la operación: la diferencia entre los dos vectores establece una transformación a un estado equivalente entre los vectores de estado. e.g. puede ser una inserción de caracteres, o el borrado etc.

La apropiada implementación de un protocolo de consistencia permite mantener la misma información a través de todas las copias en el sistema, por eso no solo es importante contar con un protocolo sino que es indispensable.

2.1.4. Replicación de la información compartida.

La replicación de información es una técnica utilizada por los sistemas distribuidos para mejorar el rendimiento y la disponibilidad de los datos, Tanenbaum [Tanenbaum, 2007], Ellis et al.[Ellis and Gibbs, 1989]. En lugar de ubicar los datos en un sitio centralizado, se ubican copias de la información en cada sitio que necesite obtener esa información. De ésta manera las lecturas se realizan de manera local y solo en las actualizaciones cuando se realiza un enlace a otro sitio.

Ventajas:

- Tolerancia a fallos

- Tiempos de respuesta más cortos

- Alta disponibilidad

Desventajas:

- Administración de réplicas
- Consistencia difícil de implementar

El problema de mantener N copias de información coherentes, recibiendo operaciones de escritura en todo momento es delegado a un protocolo de consistencia, el cual utiliza diversas técnicas para realizar su labor.

Podemos clasificar los modelos de consistencia dentro de alguno de los siguientes como nos lo asegura [Tanenbaum, 2007]:

- *Modelos de consistencia centrada en los datos.* Aplicaciones de lectura y escritura sobre datos compartidos accesibles mediante memoria distribuida, sistemas de archivos o bases de datos distribuidas.
- *Modelos de consistencia centrada en el cliente.* Almacenes de datos donde se carece de actualizaciones simultáneas y es posible ocultar inconsistencias de manera sencilla.

Modelos de consistencia centrada en los datos

El modelo de consistencia es un contrato que se realiza entre un almacén de datos y los procesos que obtienen esos datos. Dicho contrato se preserva sí y solo sí los procesos obedecen las reglas para obtener la información. Como es de esperarse los procesos que realizan las lecturas esperan recibir la última actualización de la información. Pero para tal efecto es necesaria la existencia de una medida de tiempo, un reloj global, que indique cual es la ultima operación de escritura. Ante la inexistencia de tal reloj se necesitan alternativas para obtener el mismo resultado.
Examinamos algunos modelos de consistencia:

Consistencia continua.
Define tres indicadores para medir inconsistencias, la desviación numérica entre réplicas, desviación en el deterioro entre réplicas y desviación con respecto al ordenamiento de operaciones de actualización. Estas desviaciones numéricas indican por ejemplo el número de actualizaciones que ha recibido una replica que no han sido vistas por otras réplicas.
Yu y Vahdat 2002 [Vahdat and Yu, 2002] definen una unidad de consistencia llamada *conit*. Una *conit* representa el valor que se quiere medir ya sea una actualización, un valor numérico, una acción etc. La desventaja de este modelo radica en elegir un tamaño de conit razonable, es decir un tamaño de granularidad ideal al problema que conduzca a un estado consistente y eso es complicado de realizar.
Implementar finalmente la consistencia continua puede hacerse atractiva si se le presenta a los desarrolladores como una librería que se pueda vincular con sus aplicaciones, así que se deben crear interfaces de programación sencillas.

Consistencia secuencial.
En este modelo para realizar una actualización se requiere el consenso general de todos los participantes, es decir acuerdan un ordenamiento global en el sistema. Para tal efecto se utiliza una notación que consiste en trazar una línea horizontal en el tiempo de izquierda a derecha donde se trazan las operaciones realizadas por un proceso. La notación que representa una operación de escritura es:

$$W_i(x)a, R_i(x)b \tag{2.1}$$

Significa que el proceso P_i ha realizado una escritura de X con valor de a, y una lectura de X devolviendo el valor de b. Esta proposición nos determina si las operaciones se llevan a cabo de manera concurrente entonces las interpolaciones entre dos operaciones podrán ser aceptables siempre y cuando sean visibles para todos los participantes.
Cada proceso que detecta ésta interpolación de operaciones debe resolverla de acuerdo al orden establecido por su programa y que eventualmente todas las operaciones en general conducirán a un estado consistente. Existe un contrato entre los procesos y los datos almacenados, cualquier resultado de una ejecución válida de cualquier combinación posible de estas operaciones, debe funcionar de manera correcta para cualquier proceso P_i. Cualquier comportamiento contrario es un error [Lamport, 1978].

Consistencia causal.
Es una consecuencia del modelo secuencial, i.e. el evento b es causado por la ocurrencia de un evento previo a y todos los demás procesos deben ser capaces de detectar la secuencia de eventos a y luego b, de otro modo es una inconsistencia. Los procesos que realizan una operación no causal relacionada son concurrentes, este tipo de orden obliga en cierta medida a preservar un estado consistente, porque requiere de acciones que no violen el orden establecido. En la figura 2.1.4 se muestra un ejemplo de una secuencia causal consistente. Se realizan las escrituras concurrentes $W_2(x)b$ y $W_1(x)c$, las cuales no son vistas por todos los procesos, y no es necesario que las vean en ese orden.

P1:	W(x)a	.	.	W(x)c	.	.
P2:	.	R(x)a	W(x)b	.	.	.
P3:	.	R(x)a	.	.	R(x)c	R(x)b
P4:	.	R(x)a	.	.	R(x)b	R(x)c

Solamente en protocolos de consistencia donde el orden de las operaciones concurrentes no afecta el resultado final es posible aplicarlo, por ejemplo, una aplicación de mensajería instantánea podría utilizar este esquema siempre y cuando la granularidad del fragmento compartido sea la adecuada.

Otra posibilidad es agrupar esas operaciones en secciones controladas por medio de mecanismos de sincronización para exclusión mutua. De ésta manera el acceso se restringe a los demás procesos, hasta que se completen las operaciones realizadas por el proceso que mantiene el modo exclusivo.

Modelos de consistencia centrada en el cliente.

Estos modelos se basan en la idea de centrar la coordinación de las actualizaciones en base a las necesidades del cliente. Por ejemplo existe el problema común cuando un usuario se conecta y desconecta de manera frecuente, y cambia de ubicación y no de percibir anomalías en absoluto en la información que ha actualizado. Por ello es importante que la actualización de la información a través del sistema sea lo más eficiente y rápida posible para evitar inconveniencias a los usuarios. Veremos algunos modelos para resolver gran parte de estos problemas. Todos se caracterizan por la falta de actualizaciones simultáneas.

Consistencia momentánea.
Este modelo se basa en la idea de que si no ocurren actualizaciones con frecuencia eventualmente todas las réplicas serán inconsistentes. Esto quiere decir que se garantiza que todas las réplicas serán idénticas en determinado momento. Pero se requiere que solo un pequeño número de procesos realicen las actualizaciones, de ésta forma la implementación no resulta ser costosa.
Ejemplos de aplicaciones que siguen este modelo hay muchos, las páginas Web, la asignación de nombres DNS, los Wikis, Bayou [Theimer et al., 1995] que es un modelo centrado en el cliente desarrollado para computación móvil. En esencia cuando un proceso quiere acceder a los datos este ubica la replica más cercana y realiza las operaciones en ese lugar. Eventualmente las actualizaciones se propagan a las demás réplicas. Sin embargo sólo un proceso está autorizado a realizar dichas modificaciones. Ayudando a preservar el estado consistente, evita implementar mecanismos de control de concurrencia engorrosos.
La notación para la consistencia basada en el cliente puede describirse como :

Sea $x_i[t]$ la versión de la actualización de datos x sobre la copia local L_i al tiempo t

$WS(x_i[t])$ denota el resultado de una serie de operaciones de escritura sobre L_i

Si $WS(x_i[t_1])$ se realizó sobre la copia local L_j en un tiempo t_2, este hecho se escribe:

$$WS(x_i[t_1]; (x_j[t_2]) \tag{2.2}$$

Esta notación nos sera útil para describir los modelos que explicaremos a continuación:

Lecturas monotónicas.
Con la finalidad de propagar operaciones de escritura hacia todas las réplicas y preservar el mismo orden en todas ellas se debe utilizar un modelo de lectura monotónica. La condición dice que cualquier operación de escritura sobre un elemento x se completa antes de cualquier otra operación sucesiva sobre el mismo elemento. Por ejemplo, este modelo de escrituras se observa en una base de datos distribuida: si una transacción es realizada sobre una tabla que contiene información financiera es importante completar esa transacción antes de realizar la segunda o puede resultar desastroso. EL siguiente ejemplo un proceso de lecturas realizadas por un solo proceso en dos copias distintas:

El proceso P realiza una operación de lectura sobre x en L_1 y devuelve el valor de x_1. El

conjunto de operaciones realizadas es $WS(x_1)$ en L_1. P realiza una operación de lectura sobre x en L_2, mostrado como $R(x_2)$. La consistencia se preserva siempre y cuando las operaciones de $WS(x_1)$ se propaguen a L_2 antes de una operación de lectura o $WS(x_1; x_2)$.

Escrituras monotónicas.
Con el fin de propagar las operaciones de escritura en el orden correcto se utiliza la consistencia de escritura monotónica. Consiste en completar una escritura sobre un elemento X antes que cualquier otra escritura sucesiva sobre ese mismo elemento realizada por el mismo proceso.

Es decir que una operación de escritura sobre una copia del elemento X se realiza si esa copia se ha actualizado previamente por cualquier otra operación de escritura.

Esta técnica resuelve algunos problemas, por ejemplo, efectos adversos cuando se actualizan documentos compartidos y se ven los efectos en adelante. También el resultado de las actualizaciones de caché Web donde no se actualiza de manera inmediata la copia de los datos a pesar de haber realizado cambios en la página.

2.1.5. Administración de réplicas.

Ya hablamos de ubicación de réplicas, de modelos de consistencia, de mecanismos de comunicación y de actualización. Ahora un punto igual de importante es saber como se van a distribuir esas réplicas, con que fin y que mecanismos utilizar para mantenerlas consistentes.

La administración se divide en la ubicación del servidor de réplicas y la ubicación de contenido. La ubicación del servidor de réplicas se refiere a encontrar los mejores lugares para ubicar un servidor que pueda contener un almacén de datos. Por otra parte la ubicación de contenido se refiere a encontrar los mejores servidores para ubicar contenido.

Ubicación del servidor de réplicas.

Existen algunos factores que afectan directamente el rendimiento sobre las réplicas y estas tiene que ver con la ubicación del servidor de réplicas, pero básicamente es un problema de elección. Por ejemplo [Szymaniak et al., 2005] utiliza un método el cual considera el espacio de réplicas por regiones, estas regiones se agrupan conjuntos de nodos que accedan al mismo contenido. Por lo tanto entre más nodos se agrupen mejor será la probabilidad de ubicar un servidor de réplicas en esa región. Otros se basan en algoritmos de búsqueda como el de la hormiga [Reeves, 1995], que encuentra rutas óptimas para cierto contenido y en base a esto se establece la ubicación del servidor de réplicas.

Ubicación de contenido

Elegir el mejor lugar para ubicar el contenido está directamente relacionado con el tipo de replica. Se conocen tres tipos: réplicas permanentes, réplicas iniciadas por un servidor y réplicas iniciadas por el cliente.

Réplicas permanentes.
Estas réplicas se alojan en servidores y pueden constituir el conjunto inicial de un almacén de datos distribuido. Puede tratarse de un sitio Web cuyos archivos de alojan en un número limitado de servidores. Si un servidor posee demasiada carga para atender una petición, puede delegar la función a un sitio alterno, de tal manera que se agilice la transferencia de información: Esto se le conoce como *mirroring*, que es una técnica se puede utilizar para replicar bases de datos, servidores de archivos, Web Services etc.

Réplicas iniciadas por servidores.
En este caso en lugar de tener alojadas réplicas permanentes en sitios conocidos, se le otorga al servidor la capacidad de crear las réplicas como mejor convenga, por ejemplo ubicarlas en un sitio con mucha demanda. La intención es reducir la carga del servidor por ejemplo, se pueden migrar ciertos archivos que tengan alta demanda a los servidores próximos a los clientes que lo solicitan funcionando como un servidor caché Rabinovich [Rabinovich et al., 1999]. Puede suceder también que estas copias de réplicas puedan ser iniciadas por un servidor de manera automática en respuesta a una gran carga de peticiones en una ubicación en especifica. Tiempo después estas copias temporales son dadas de baja y el servidor está listo para desplegar tanta copias como sean necesarias si así se requiere.
Sin embargo ésta solución es complicada de implementar debido a que se debe seguir una serie de pasos y consideraciones antes de poner en ejecución una transferencia de contenido y muchas veces esto no resulta exitoso.

Réplicas iniciadas por el cliente.
Existen soluciones más sencillas a la migración de réplicas, por ejemplo las réplicas iniciadas por el cliente (caché). Los cachés son utilizados para mejorar el rendimiento en el acceso a los datos y existen muchas maneras de realizarlo. Se usan por ejemplo en los navegadores Web para evitar constantes descargas de las páginas, de ésta manera el acceso es local.

Estos cachés no se almacenan por tiempos prolongados, además es posible compartir datos de uso común entre clientes, basándose en la idea de que la información puede resultar igualmente importante para varios usuarios. Otra solución es crear servidores caché ubicados en lugares específicos y esperar que el cliente localice el más conveniente.

Distribución de contenido.
Tiene que ver con la propagación del contenido o actualizaciones hacia algún servidor, consiste en tres tipos:

- **Notificación de actualización**. La notificación de actualización es la más sencilla y no requiere de grandes anchos de banda, ya que sólo se especifica los datos que necesitan ser actualizados o invalidados, dependiendo de la situación. El éxito de este modelo depende de la adecuada administración de las notificaciones, es decir, elegir el mejor momento para propagar una notificación a las réplicas sobre la modificación de los datos.

- **Transferencia de datos de una replica a otra.** La transferencia de datos es quizá la más costosa en consumo de ancho de banda, por lo que existe también la manera

de hacer más eficiente la transferencia mediante el uso de colas de actualización, donde se agrupan las modificaciones para posteriormente mandar todo ese paquete en una sola vez.

- **Propagación de la operación de actualización a las réplicas.** La propagación de actualizaciones depende de qué tan grande es el tamaño de las actualizaciones y qué tan frecuente se realizan. El número de operaciones de actualización no debiera sobrepasar a las de lectura, porque se realizarían envíos innecesarios. Hay situaciones en las que las actualizaciones se realizan de manera constante, por lo tanto, no es recomendable propagar las notificaciones cada vez que se modifica una sección de los datos, o quizá sea buena idea propagar la notificación sobre los datos que necesiten ser actualizados.

Otra opción es indicar a las réplicas el tipo de actualización que deben realizar, funciona manteniendo un proceso asociado a cada replica, este proceso la mantiene actualizada mediante el envío de operaciones las cuales son de bajo costo en consumo de ancho de banda. El ultimo caso tiene que ver con la suposición de que la representación de cada replica es realizada por un proceso que mantiene un estado consistente gracias a la propagación de operaciones con un costo bajo en el ancho de banda. Tales operaciones pueden ser tan complejas como se requiera con tal de mantener a las réplicas consistentes.

Modelos basados en Push y Pull.
Por ultimo examinaremos otro caso, el cual tiene como objetivo decidir si las actualizaciones se envían o se extraen, se conoce como *push* y *pull*.

- **pull**. Las actualizaciones se solicitan a un servidor por parte de un cliente, es similar a los cachés explicados anteriormente (ver imagen 2.9). Funcionan verificando si los datos locales están actualizados, de lo contrario se envía la petición para obtener los nuevos datos. En caso de una modificación, los datos son enviados al caché y posteriormente al cliente que los solicitó. Este tipo de actualización es utilizado por los navegadores Web que ya hemos analizado en párrafos anteriores. La desventaja es que si el caché de datos se pierde aumentan el tiempo de respuesta.

- **push**. Las actualizaciones se realizan de manera automática, es decir, sin que sean solicitadas (ver figura 2.10). El servidor envía a todas las réplicas las actualizaciones con el fin de mantenerlas idénticas. Se usan principalmente en los servidores de réplicas permanentes donde se necesita un que exista un alto grado de consistencia entre las réplicas ya que son utilizadas por múltiples clientes, porque conforme se propaga la información a esos servidores de réplicas muchos clientes realizaran esas lecturas, se tiene entonces una relación costo-beneficio muy alta, además los datos están disponibles de inmediato.

La tabla 2.1 presenta un análisis de las ventajas y desventajas de los enfoques presentados:

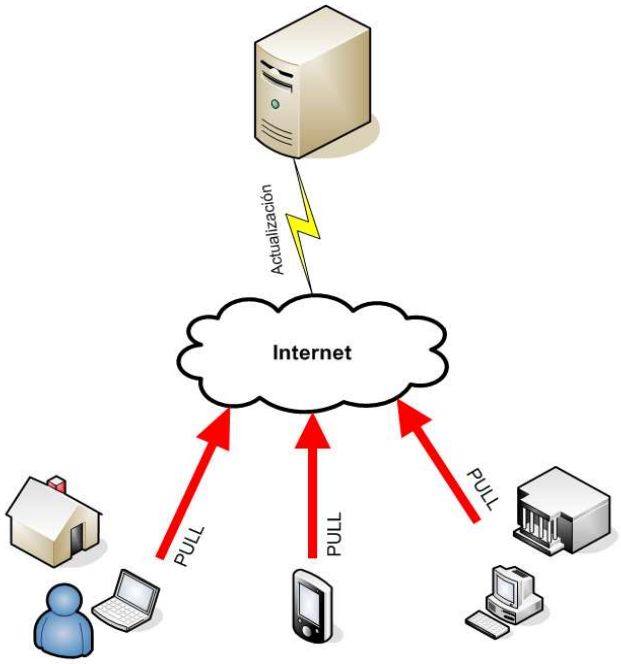

Figura 2.9: Modelo Pull

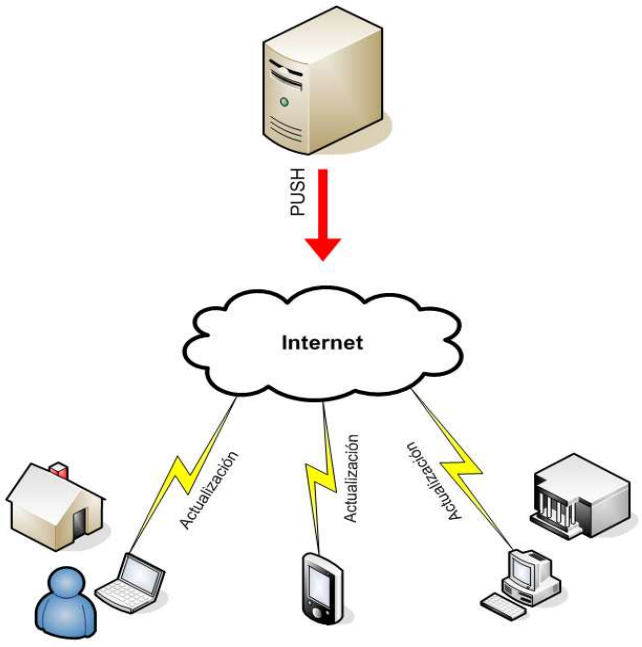

Figura 2.10: Modelo Push

Comparación entre los protocolos *push* y *pull*		
Condición	Basado en push	Basado en pull
Estado del servidor	Lista de réplicas cliente	Ninguno
Mensajes enviados	Actualiza (y posiblemente trae la actualización después)	Sondea y actualiza
Tiempo de respuesta en el cliente	Inmediato (o busca el tiempo de actualización)	Busca el tiempo de actualización

Cuadro 2.1: Protocolos *push* y *pull*

En cuanto al enfoque push es claro que el control sobre las actualizaciones pertenece al servidor de réplicas, el cual decide es el momento para propagar las actualizaciones a las demás réplicas, sin embargo existen problemas. Uno de ellos es que se tiene que llevar un control acerca del estado de las demás réplicas, esto representa una sobrecarga extra para el servidor. Por eso a menudo se opta por utilizar tanto envío de peticiones *pull* como envío de actualizaciones *push*, en forma híbrida.

En el método híbrido se establece un "contrato" en donde el servidor se compromete a entregar actualizaciones por un periodo de tiempo, cuando este periodo expira el cliente se ve forzado a pedir un nuevo contrato. Los tipos de contrato varían, dependiendo las estrategias que se requieran, por ejemplo, existen contratos que se otorgan a clientes que con frecuencia requieren actualizar sus datos y por otro lado, se otorgan contratos a elementos de datos que se sabe no serán modificados por mucho tiempo.

Existen otros tipos de modelos que tienen que ver con mantener un elemento primario para cada dato almacenado, el cual es responsable de coordinar las operaciones de escritura sobre x.

- **Protocolos de escritura remota.** En estos protocolos el objetivo es enviar operaciones de escritura a un solo servidor fijo. Por ejemplo, un proceso realiza una operación de escritura en un servidor primario x, éste realiza la actualización de su copia local de x y envía la actualización a los demás servidores de respaldo. Estos a su vez actualizan su copia, cuando todos han realizado ésta operación envían un acuse de recibo al proceso inicial (ver figura 2.11). El rendimiento es el punto malo en estos protocolos porque el proceso inicial queda bloqueado hasta que recibe las notificaciones de acuse. Con los protocolos de escritura primaria se implementa la consistencia secuencial, ya que las operaciones se realizan de manera ordenada a tiempo global único [Budhijara et al., 1993].

- **Protocolos de escritura local.** Se trata de una variación del modelo anterior, ya que las actualizaciones aun se hacen llegar a una copia primaria. Lo que cambia en este enfoque es que cuando se desea realizar alguna modificación, se hace llegar la

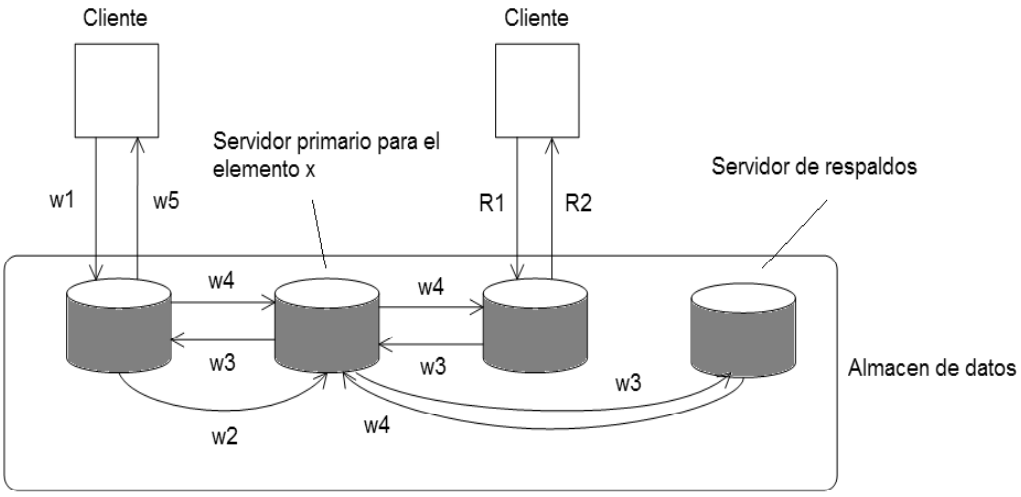

Figura 2.11: Protocolo Primario.

primaria a la ubicación donde se requiere. De ésta manera si se realizan varias modificaciones se pueden hacer de manera local, los procesos de lectura pueden realizar sus operaciones en sus copias locales, posteriormente se envían las modificaciones a las demás copias como ya vimos.

Explicación de la figura 2.11: *W1.* Escribe petición. *W2.*Remite la petición a la copia primaria. *W3.* Indica a los respaldos que se actualicen. *W4.* Acusa la actualización. *W5.* Acusa la escritura finalizada. *R1.* Lee la petición. *R2.* Respuesta a la lectura.

2.2. Tolerancia a Fallos

La tolerancia a fallos es una cualidad deseable en cualquier sistema, y es un reflejo de la calidad de su construcción. Esto quiere decir que el sistema está preparado para responder ante situaciones inesperadas como desconexiones, pérdida de paquetes, fallo en envío de datos, inconsistencia de datos, etc. Sin embargo como es de esperarse, no resulta sencillo implementar esos mecanismos para manejo de errores. Las razones por las cuales un sistema falla son diversas, es un tema delicado, hablaremos un poco de los distintos tipos de fallas y las soluciones hasta ahora propuestas en el estado del arte para resolver cada uno de ellos.
Una falla es una anomalía en la prestación del servicio que el sistema provee, ya sea por mal diseño o por factores externos. Algunos de estos factores externos son el resultado de ataques intencionales, por parte de agentes maliciosos, por lo que la seguridad en los sistemas también es un factor importante en la estabilidad de un sistema. Un factor externo son las fallas bizantinas que son anomalías aleatorias para las cuales el sistema no ésta preparado del todo.

Tipo de Falla	Descripción
Falla de congelación	Hasta hace un momento el servidor trabajaba bien, pero se detuvo
Falla de omisión	Un servidor no responde a las peticiones
Omisión de recepción	un servidor no recibe los mensajes
Omisión de envío	Un servidor no envía mensajes
Falla de tiempo	El servidor no responde dentro de un tiempo estimado
Falla de respuesta	respuesta incorrecta del servidor
Falla de valor	El valor es incorrecto
Falla de transición	Flujo de control incorrecto
Falla bizantinas	Respuestas arbitrarias en tiempos arbitrarios

Cuadro 2.2: Tipos de fallas en un sistema

Las fallas más comunes tienen que ver con el hecho de que la red es un medio no confiable de transmisión de datos, muchas cosas pueden ocurrir en el camino, pérdida de paquetes, interferencia, ataques, desconexiones, corrupción de datos, retardos, bloqueos, redireccionamientos, etc.
Si a eso le agregamos que el sistema puede dar respuestas incorrectas a eventos esperados congelamientos, flujos incorrectos, tenemos entonces un gran problema entre manos. Agrupamos a los distintos tipos de fallas en la tabla 2.2

Un método para disminuir el impacto de estas fallas es la redundancia, la cual oculta el hecho de que ocurren fallas a los demás procesos. La redundancia puede ser de varios tipos dependiendo el caso:

- *Redundancia de tiempo.* Se realizan las acciones correspondientes y si ocurre una anomalía de reenvían esas acciones nuevamente hasta ser satisfactorias.
- *Redundancia física.* Se adicionan procesos que auxilien al sistema a tolerar o a tratar esas fallas, en otras palabras es replicación de procesos.
- *Redundancia de información.* Se envían bits adicionales para recuperar mensajes mutilados.

La redundancia física es la forma en que los sistemas replicados se defienden contra las fallas, esto es sencillo de observar puesto que al existir varias copias distribuidas de la misma información aun si fallan uno o dos sitios, todo los demás podrán seguir compartiendo esa misma información. Otra manera de ocultar las fallas dentro de un sistema es el enmascaramiento de procesos, lo cual significa que tendremos copias de un proceso que ayudaran a ocultar un proceso defectuoso. Una manera de realizar esto es por medio de protocolos primarios o a través de protocolos de escritura replicados.

En el caso de los protocolos primarios existe un grupo de procesos que se organizan de manera jerárquica en la cual un protocolo primario coordina las operaciones de escritura. En caso de que este proceso se congele se acciona un mecanismo de elección para seleccionar un nuevo primario.

Por otro lado, los protocolos de escritura replicados consisten en organizar un conjunto de procesos idénticos en un grupo. También se le conoce como replicación activa, en la cual existe un proceso que realiza las operaciones de escritura aunque también es posible enviar una actualización.

En el siguiente capítulo mostraremos el diseño de nuestra propuesta de solución utilizando la teoría explorada hasta ahora.

Capítulo 3

Diseño

Las aplicaciones colaborativas utilizan diversos métodos para mantener la consistencia de la información, periódicamente actualizada de tal manera que garanticen la disponibilidad de los recursos. En este capítulo nos enfocaremos a describir y precisar la propuesta de solución que hemos adoptado en respuesta a los problemas que se presentan en los modelos actuales. Dichos problemas no permiten al desarrollador de aplicaciones colaborativas crear aplicaciones flexibles y extensibles, tampoco permiten liberar al programador de escribir código complicado para coordinar la distribución de la información, ni realizar cambios o modificaciones a las políticas de replicación/notificación. En base a estas deficiencias justificamos nuestra propuesta de diseño que soluciona estos problemas. Dotar de flexibilidad a un sistema soporte de la colaboración, para que utilice múltiples políticas de replicación, no es sencillo y es un problema que no muchos están dispuestos a afrontar.

3.1. Motivación

Para dotar de la característica de flexibilidad a un sistema distribuido es crucial que el sistema este organizado como una colección de componentes relativamente pequeños y de fácil reemplazo y/o adaptables. Muchos sistemas contemporáneos y viejos son construidos usando un enfoque monolítico en el cual los componentes son separados de manera lógica pero implementados como un enorme bloque de programa.

Problemas identificados.

- El motivo principal para modificar el funcionamiento de un sistema distribuido es causado a menudo porque algún componente ya no cumple con la tarea para la cual se diseñó originalmente. Las razones pueden ser desde cambios en los requerimientos, hasta la existencia de alguna falla. Reemplazar estos componentes o modificarlos puede resultar muy costoso y terriblemente complicado, además, no se asegura que una vez implementado e integrado realice correctamente su función, sin afectar a otros módulos de la manera que se espera.
- Actualmente se implementan los mecanismos de actualización y la propagación de operaciones en los sistemas cooperativos existentes. El problema de este paradigma

es que tenemos una estructura rígida altamente cohesionada con el código de la aplicación. Tratar de realizar un cambio es altamente complicado porque involucra cambio de módulos, funciones, métodos, clases, mecanismos etc., que comparten distintos grados de dependencias.

- En caso de una modificación, un sistema con este tipo de diseño quedara eventualmente inhabilitado para seguir prestando su función hasta que las modificaciones queden terminadas. Esto imposibilita o dificulta la inclusión de nuevas funcionalidades y por ello sistemas enteros se ven condenados a quedar obsoletos por no ser capaces de cambiar y adaptarse a los nuevos requerimientos.

Soluciones.

- Automatizar el proceso de replicación/actualización, crear un mecanismo que tome la decisión de enviar y recibir información en algún intervalo de tiempo T, de otro modo el sistema de actualizaciones seguirá funcionando de forma manual involucrando al usuario en el proceso de replicación. Para ello se crea un proceso que es ejecutado en segundo plano, un "demonio"(daemon) que es responsable de la actualización de la información.

- Identificar y modularizar las funcionalidades de la aplicación, así mantenemos control sobre cada aspecto y puede ser una manera fácil de llevar el mantenimiento de cada módulo [Sommerville, 2005].

- Aislar el módulo que controla la distribución de información del resto de la aplicación: así tendremos un mayor control sobre las funcionalidades que pueden ser modificadas o agregadas en un momento determinado. Además realizando un diseño de caja negra logramos una baja cohesión del código de este módulo con el resto de la aplicación.

Herramientas de apoyo.

- **Diseño de componentes.** Está basado en la política de reutilización de código, que no sólo nos puede ayudar al objetivo de mantener el sistema altamente escalable y actualizado, sino también permitir un fácil mantenimiento que contribuye al mejoramiento de la calidad de sistema. Normalmente este método de diseño se utiliza para desarrollar algún algoritmo que se utiliza con frecuencia, que es repetitivo o que posee código similar, por ejemplo el efectuar el mismo procedimiento para algún calculo numérico en diversos sistemas puede ser tedioso de programar una y otra vez. Gracias a este tipo de diseño podemos se capaces de crear módulos genéricos que implementan y proveen de manera eficiente sus servicios y al mismo tiempo tener la propiedad de ser reemplazables porque la interfaz de comunicación, que es el elemento de interacción entre el componente y su exterior, no cambia.

- **Alta cohesión.** Es la característica que posee un objeto de realizar tareas sin relación aparente entre ellas, en orden secuencial y sólo una tarea a la vez.

- **Bajo acoplamiento.** Esta técnica consiste en mantener lo menos ligado los componentes entre si, de tal manera que al no tener dependencias con otros puede ser susceptible de modificaciones sin alterar el estado general de un sistema [Larman, 2005].

Las etapas propuestas para el desarrollo de componentes, descritas en [Sommerville, 2005], guían el desarrollo de los módulos que necesitamos, en la figura 3.1 se muestran las diferentes etapas involucradas en la creación de un nuevo componente, cada etapa se describe a continuación:

- **Análisis de componentes**. Especifica los requerimientos e identificar las funcionalidades.
- **Modificación de requerimientos**. Analiza la información obtenida de la primera capa para modificar los componentes.
- **Diseño del sistema con reutilización**. Define un marco de trabajo, en el que se busca reutilizar código.
- **Desarrollo e integración**. Analiza el software que no se puede adquirir, y por lo tanto se desarrolla, y finalmente se integran los componentes.

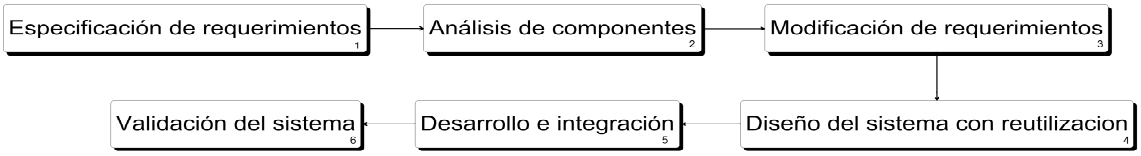

Figura 3.1: Diseño de Componentes

Gracias a ésta guía podremos realizar mecanismos flexibles que sean adaptables, intercambiables, portables y de mantenimiento sencillo, además incrementamos la vida útil de este sistema y proveemos de herramientas flexibles de administración.

3.2. Análisis de requerimientos

El proceso inicial en la creación de una herramienta, que pueda utilizarse para replicar contenido útil entre dos aplicaciones colaborativas distribuidas, consiste en un proceso de observación y análisis, en el que se pueda desarrollar un primer bosquejo del diseño general. Sabemos por los objetivos generales indicados en el Capítulo 1, que el sistema debe proveer un servicio de actualización de entidades replicadas automático. Por ello, como primer punto, debemos analizar e identificar las funcionalidades que nos permitirán cumplir con los objetivos delineados al inicio del presente trabajo de tesis. En el siguiente esquema podemos observar como funciona una replicación de datos entre dos sitios A y B, ubicados en lugares distintos y conectados a la red. Como mencionamos en el capítulo 2, existen muchas técnicas por las cuales se pueden enlazar estos equipos, pero no nos ocuparemos de eso ahora. Por lo pronto vemos que para enviar información entre dos

equipos inevitablemente debe existir una coordinación, es decir un equipo que recibe y otro que envía, en la figura 3.2 observamos este proceso que describimos a continuación:

1. Un servidor identificado como B, posee cierta información almacenada de manera local,
2. Un sitio A requiere una copia de ese contenido, por lo que realiza una petición al servidor B,
3. Una vez validada esa petición, el servidor B se prepara para enviar la información al equipo A,
4. Se establece un canal de comunicación entre los dos sitios,
5. El servidor B envía la información y después de algunas verificaciones el sitio A envía un acuse de recibo o ACK al servidor B.

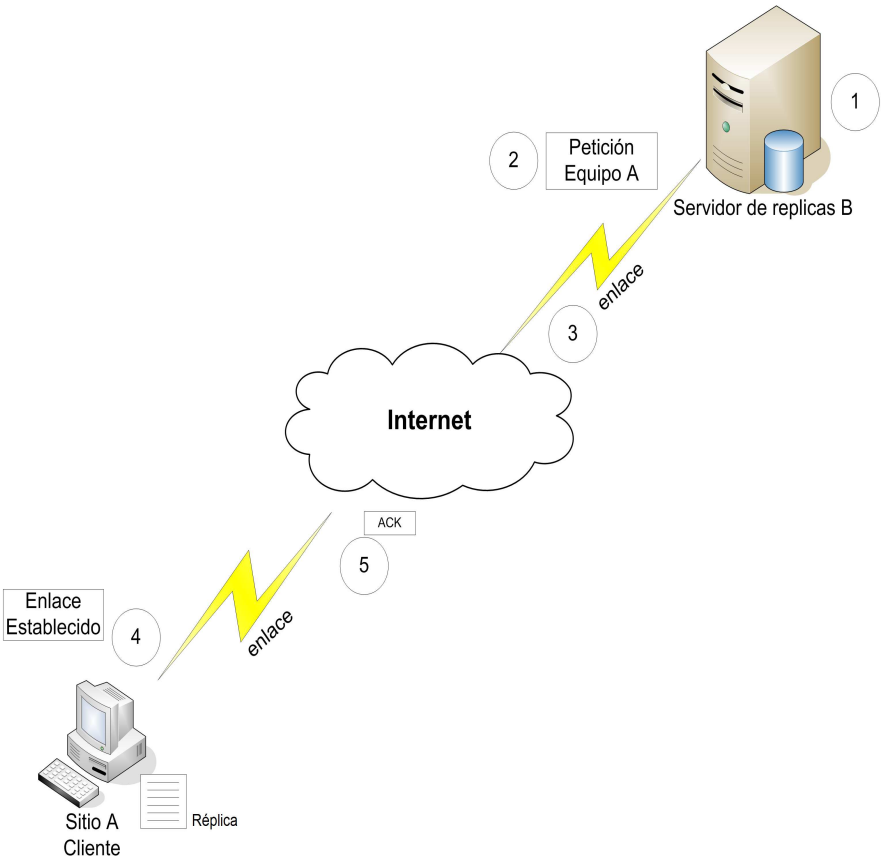

Figura 3.2: Replicación de Datos

Este modelo utiliza una arquitectura cliente-servidor para actualizar la información entre dos sitios utilizando un esquema basado en *pull*, el cual como explicamos en el

Capítulo 2, consiste de solicitudes de actualización a un servidor por parte de un cliente, como los cachés de los navegadores.
Como vimos se necesita de una coordinación entre los sitios involucrados, a fin de mantener coherente la información en los dos sitios. La existencia de diferencias entre una copia local de la información y la copia en otro sitio es una falla. Sin embargo al ser un esquema tan simple, el problema es resuelto mediante la solicitud de una actualización al servidor, puesto que el servidor siempre tiene la última versión.

Ahora bien, este escenario puede complicarse mucho más si aumenta el número de participantes, (i.e. sitios cliente), solicitando la misma información. El servidor puede ser inundado de peticiones y por lo tanto se vuelve un problema de congestión. Puede balancearse la carga al aumentar el número de servidores. Sin embargo la complejidad crece porque ahora se necesita de una coordinación entre servidores para evitar inconsistencias entre ellos.

Con una coordinación eficiente, la replicación de datos se convierte en un medio poderoso para distribuir contenido compartido a muchos clientes ubicados en lugares distantes. Tomemos por ejemplo el modelo que utiliza *Emule*, que es una aplicación para compartir archivos que utiliza una arquitectura *P2P* para su distribución. Este esquema asigna un identificador único para cada archivo compartido, por medio del cual es ubicable a través de la red.
Los clientes que desean acceder al archivo utilizan ese identificador global, el cual es obtenido a través de un servidor (ver figura 3.3), además ese servidor posee la dirección de cada sitio que contiene ese archivo. Dichos archivos son divididos en pedazos de cierta longitud, y cada porción posee su propio identificador, así es posible compartir diversos pedazos del mismo archivo por múltiples clientes.
La búsqueda por un archivo puede extenderse a todos los servidores centrales que poseen los registros de sus clientes, con la finalidad de encontrar el recurso solicitado, una vez encontrado proporcionan su ubicación para que la conexión se realice de manera directa entre peers.

En la figura 3.3 podemos observar a los servidores que albergan estos registros y a los peers que se conectan entre sí para obtener esos recursos. Un sitio servidor puede contener múltiples servidores, con el fin de balancear la carga debido a múltiples peticiones, además sirve como medida de respaldo, en caso de caídas inesperadas de servidores.

La idea al mostrar este modelo ha sido visualizar un esquema interesante de compartición de recursos en el cual, mediante unas pocas reglas, se puede obtiene una red extensa de colaboración. Si aprovechamos ésta idea y la adaptamos a nuestros requerimientos podemos obtener un sistema eficiente de compartición de recursos (entidades), que sea capaz de realizar su labor de manera automática. Pero, como primer paso, debemos describir cada una de las funcionalidades necesarias que componen este modelo.

3.2.1. Módulo de Comunicación.

Como hemos podido observar en *Emule*, se utiliza un esquema tipo peer-to-peer (par a par) para realizar la colaboración, sin embargo a pesar de que la interacción se realiza de manera directa entre dos sitios, se sigue utilizando el modelo cliente/servidor para realizar el enlace. Por lo tanto necesitamos de un módulo que nos provea de la capacidad

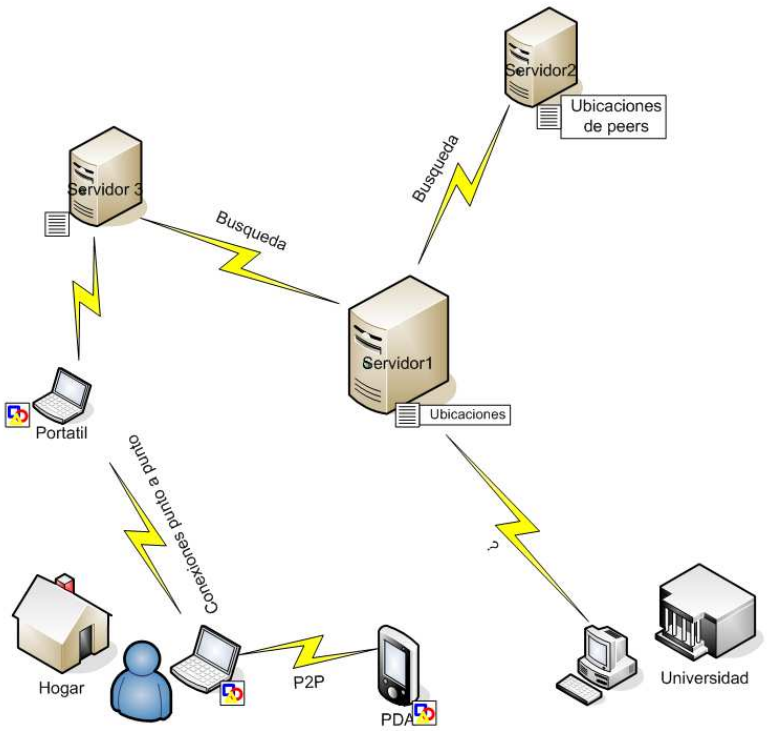

Figura 3.3: Funcionamiento de Emule

de realizar dichos enlaces tipo cliente/servidor.
Con base en la modularización, proponemos la creación de un módulo especializado con funciones tipo cliente y funciones de tipo servidor. Necesitamos que estas funciones queden independientes del resto de la implementación, es decir, ajenos a la administración del contenido y de los mensajes que se distribuyen entre equipos. Con esto logramos una alta cohesión y bajo acoplamiento entre las funciones de comunicación. Así es posible no solo extender la vida útil de las funciones, si no también permitir una implementación independiente.
Para resumir, agrupamos funciones que se utilizan para establecer enlaces entre equipos por medio de la red utilizando cualquier método de conexión, protocolo, algoritmo, etc.

Las funciones que debe proveer son:

- *Apertura de canal y enlace.* Son funciones para apertura de sockets, establecimiento de conexión a un servidor utilizando protocolos *TCP*, *UDP*, etc,
- *Sincronización.* Establecimiento de un orden entre envíos,
- *Cliente/Servidor.* Servicios que proveen este esquema de conexión,
- *Serialización.* Los paquetes deben ser preparados para ser enviados por la red.

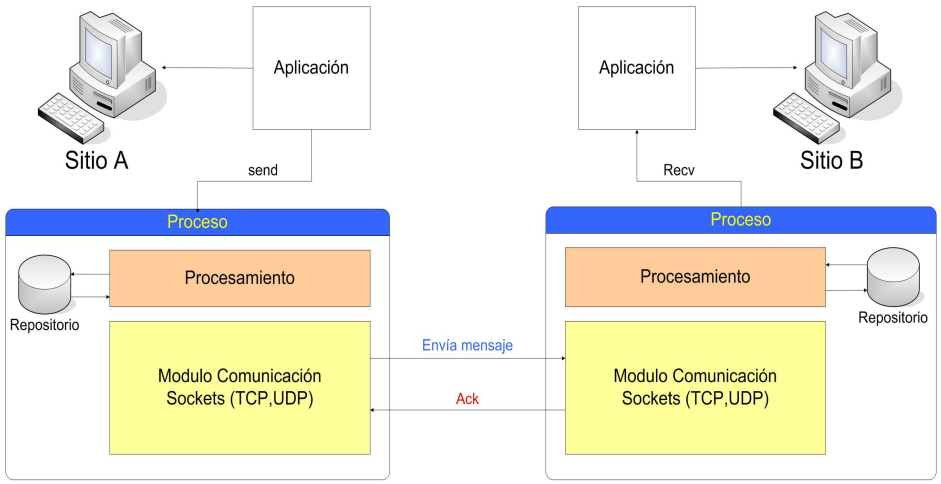

Figura 3.4: Módulo de Comunicación

Utilizando los servicios que este módulo proporciona, cualquier proceso, que lo utilice, puede establecer comunicación con algún sitio remoto evitando así escribir código redundante. El modelo que representa ésta idea se muestra en la figura 3.4, en la que indicamos la interacción utilizada en un enlace de comunicación.

Características del módulo.

Siguiendo el patrón de diseño de componentes, la interacción del módulo se realiza a través de una interfaz, la cual se utiliza como una fachada que oculta la implementación de los métodos. Por ello es posible realizar modificaciones, agregar funcionalidad y realizar extensiones o arreglos sin que los procesos externos lo noten. Con esto logramos tener un alto grado de mantenimiento.

Los servicios de cliente/servidor en cada sitio permiten la implementación de una arquitectura de tipo peer-to-peer (par a par) que proporciona la alta disponibilidad de recursos que necesitamos.

- **Servidor.** La parte servidor permanece atendiendo las peticiones de los clientes que solicitan: operación sobre los datos, obtención de recursos, búsquedas etc. Es un servicio destinado principalmente a la recepción de información, no tiene capacidad para modificar o alterar el contenido de la misma. Por ello debe entregar el contenido al proceso adecuado para su tratamiento. Se diseñó con capacidades multitarea, es decir puede crear un subproceso (hilo), por cada petición recibida. Así no permanece bloqueado y continua recibiendo peticiones.

- **Cliente.** La parte cliente posee las funciones necesarias para el establecimiento del enlace entre los equipos. Siguiendo el modelo *OSI* [Zimmermann, 1980], se diseñó para aceptar varios tipos de protocolos de comunicación con el fin de proporcionar

servicios independientemente de las características de cada sitio, (e.g. procesador, memoria, dispositivos de almacenamiento, interfaces para comunicaciones, códigos de caracteres, sistemas operativos.)

En el siguiente diagrama mostramos las funciones que proporciona el módulo de comunicación, (ver figura 3.5) el cual muestra una conexión entre sitios y la interacción cliente/servidor tipo peer-to-peer (par a par). Los equipos se encuentran conectados a la red utilizando algún protocolo e comunicación en común y todos poseen las ubicaciones de los demás, por lo que el enlace es directo.

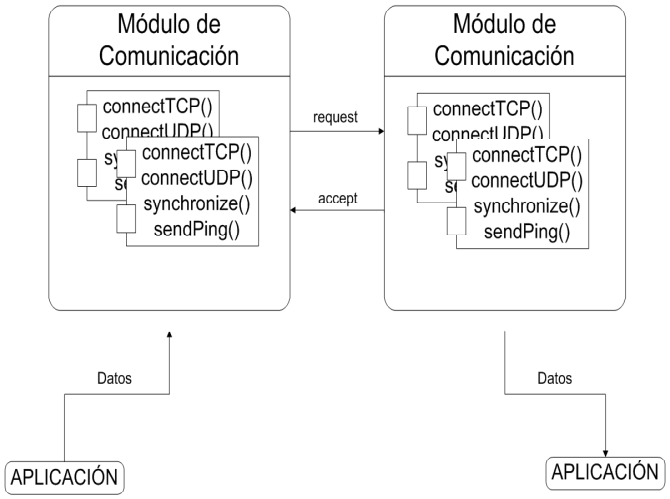

Figura 3.5: Módulo de Comunicación, Funciones del Servidor

Típicamente para enlaces en los que se requiere garantizar la integridad de los datos se utiliza el protocolo *TCP/IP*, sin embargo, no hay razón para evitar utilizar otros protocolos como *UDP* que a través de datagramas es más rápido en la entrega, pero corre el riesgo de pérdida en sus paquetes. Sin embargo es conveniente poseer la capacidad de implementar más de un protocolo con el fin de dar una diversidad y opciones de transmisión para distintas necesidades, por lo tanto este módulo tiene la capacidad para permitir la adición de varios protocolos de comunicación. Más adelante en este capítulo explicaremos como realizamos ésta tarea.

Observamos un escenario en la figura 3.6 en la que podemos apreciar la comunicación entre dos sitios los cuales poseen dos copias de la misma información. En periodos de tiempo T_i distintos se realizan modificaciones a las copias locales en ambos sitios, por lo que se requiere una actualización en ambos sitios, con el fin de mantener las copias idénticas. Aquí se utiliza un esquema de tipo *push*, el cual consiste en propagar los cambios y actualizaciones a los sitios que poseen una copia de la información, sin que éstos la soliciten. Es decir es una actualización automática. Por lo tanto cada sitio es responsable de enviar una actualización de la información modificada a los demás sitios. Existen muchos esquemas para realizar ésta acción, pero por ahora solo diremos que es necesaria

una sincronización en el tiempo, con el fin de ordenar esas actualizaciones y preservar la consistencia.

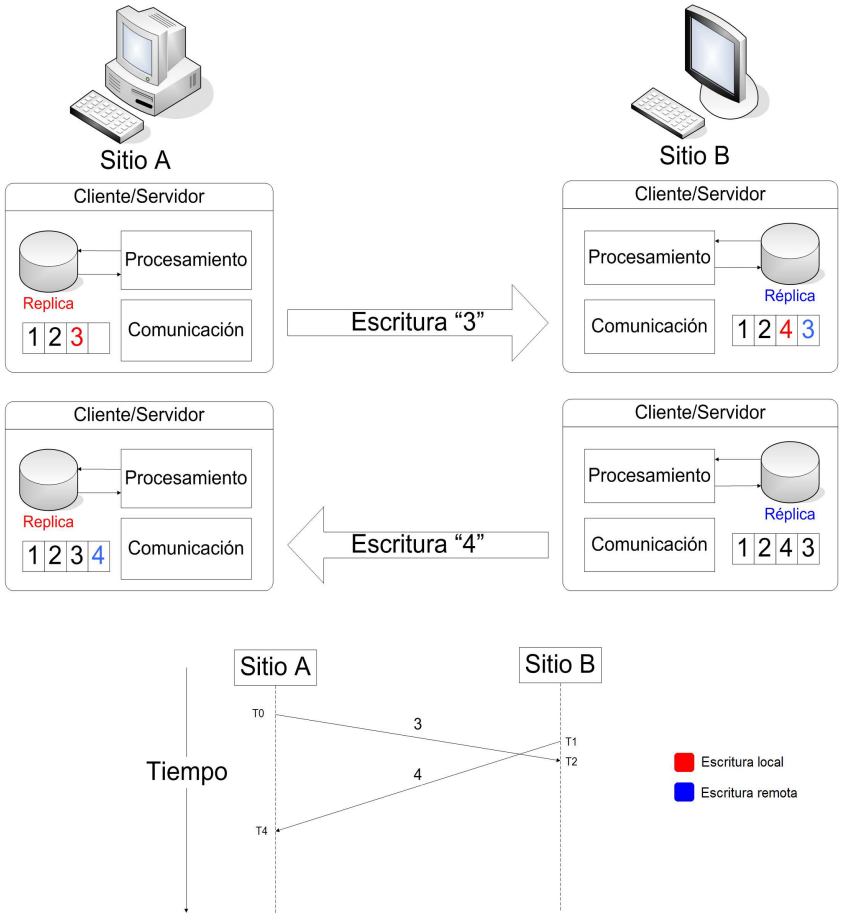

Figura 3.6: Diferencias entre Tiempos de Propagación

Explicamos brevemente los eventos ocurridos durante ésta actualización:

1. En dos sitios A y B se comparte una copia de la misma información,
2. En un tiempo T_0, el sitio A realiza modificaciones a la copia local,
3. El sitio A es responsable de difundir la modificación por lo que establece un canal de comunicación con el sitio B para enviar la actualización,
4. En un tiempo T_1, el sitio B también realiza modificaciones a su copia local,
5. El sitio B abre un canal de comunicación con el sitio A con el fin de difundir su actualización,
6. El sitio A recibe la actualización proveniente de B, al tiempo que el sitio B recibe la de A,

7. El valor en los dos sitios después de la actualización difiere, la razón fue la diferencia entre tiempos de propagación de esos valores. Ha ocurrido una inconsistencia.

Como se puede observar, con el fin de preservar un orden en las actualizaciones es necesario realizar tareas adicionales a sólo recibir y enviar paquetes. Se requiere la implementación de un mecanismo o protocolo que realice esta labor, porque si la interacción se realiza entre múltiples sitios, puede ser muy compleja la coordinación. Por ello es necesario un módulo en nuestro sistema capaz de coordinar todas estas actualizaciones, es decir, cuándo, cómo y dónde se deben realizar y bajo que condiciones. Es el módulo de replicación de información.

3.2.2. Módulo de Replicación.

Mostraremos el diseño de este módulo cuya función principal es coordinar las actualizaciones entre sitios que comparten información. La replicación de datos es un tema extenso el cual explicamos de manera breve en el capítulo dos. Explicamos además que existen muchas técnicas para mantener consistente la información, estas técnicas están en función de las necesidades de las aplicaciones.
Por lo tanto, este módulo es capaz de albergar múltiples técnicas, (i.e. protocolos de replicación/notificación) que pueden ser utilizados por una aplicación en cualquier momento. El diseño se basa también en el concepto de reutilización que utilizamos en el módulo de comunicaciones. El punto de interacción es a través de una interfaz, por medio de la cual, ingresan las instrucciones de operación.

Las funciones que éste módulo debe proveer son:

- **Permitir la reutilización de código.** Con ello se extiende la vida útil del sistema.
- **Albergar distintas políticas**. Debe ser capaz de aceptar varios tipos de algoritmos y protocolos de control de concurrencia, coordinación de actualizaciones, política de replicación/notificación etc., y ser capaz de utilizar cualquiera en cualquier momento.
- **Proveer una interfaz única.** Provee un estándar para los diseñadores de políticas de replicación/actualización. A través de esta interfaz se ingresan los datos e instrucciones necesarios para activar el motor de replicación de entidades.

Organización del módulo.

La función principal de este módulo es albergar varios tipos de políticas de replicación y actualización, y control de concurrencia que pueden permanecer activas al momento. Representamos a estas políticas dentro del módulo de replicación como componentes los cuales pueden ser removidos, reemplazados o modificados.
Esto sólo es posible si todos los componentes implementan la misma interfaz, así la entrada de datos permanece idéntica, aunque la lógica detrás sea distinta.
La comunicación entre esos componentes también es a través de una interfaz, así que depende de la lógica implementada en esos componentes, realizar una colaboración exitosa.

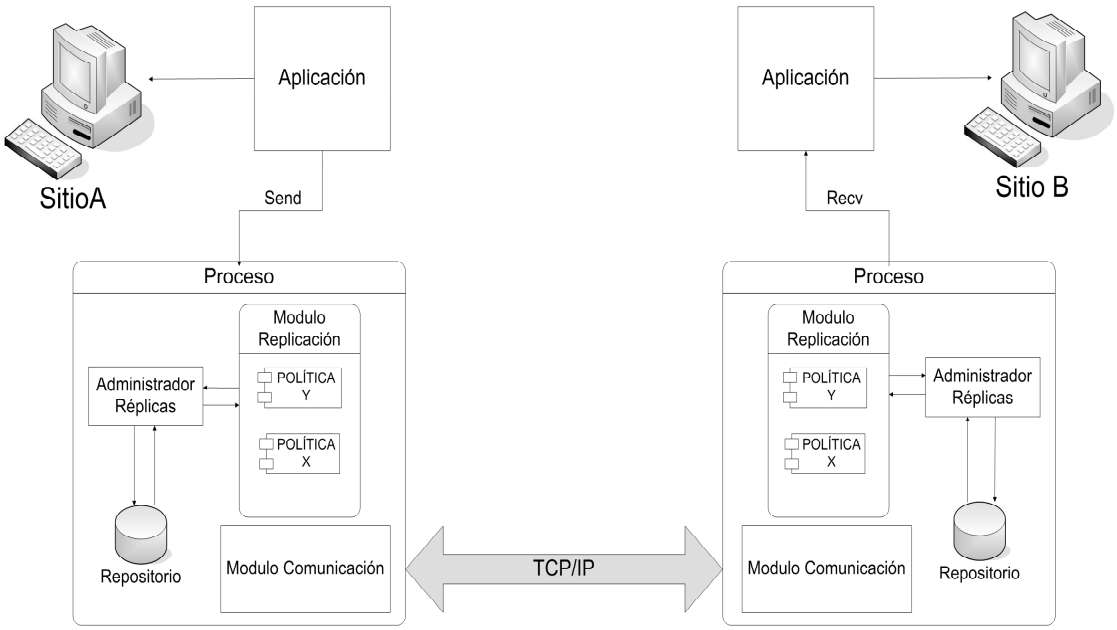

Figura 3.7: Organización del Módulo de Replicación.

En la figura 3.7 podemos observar la organización del módulo de replicación y su interacción con el módulo de comunicación. Los componentes representan la implementación de un control de concurrencia, y una política de actualización/replicación. Por medio de esta representación queremos hacer notar que estos componentes son reemplazables. El control de concurrencia interactúa con la política de actualización/replicación con el objetivo de permitir la coordinación que se requiere para mantener una o varias réplicas actualizadas.

La coordinación con otros componentes distribuidos se realiza utilizando el módulo de comunicación. Mediante esta separación de tareas logramos que la coordinación se realice de manera "local", es decir, que el diseño de estos componentes al no contemplar algoritmos de comunicación, enlaces remotos etc., se focaliza en la interacción de los componentes como si permanecieran en el mismo sitio. Proveemos de transparencia de distribución y replicación.

Estructura de datos.

El módulo de replicación es independiente de la representación de los datos que maneja. A nivel de este módulo no podemos realizar nada con esos datos. Es a nivel de la aplicación que la información cobra sentido, por ello la responsabilidad de este módulo termina al entregar esos datos a la aplicación. Sin embargo con el fin de poder realizar operaciones sobre esos datos, necesitamos un concepto, es decir, una representación de esos datos a la cual podamos asociar una actividad. Esta representación, es llamada *entidad* [Sun and Ellis, 1998][Decouchant et al., 2010] y representa los datos compartidos y requiere toda una arquitectura que administre su creación, control, y accesos para un

correcto funcionamiento.

Dicha capa se encuentra fuera de los alcances de este trabajo de tesis, debido a su complejidad y al tiempo de investigación necesario para una apropiada implementación. Por lo pronto a partir de aquí en adelante asumiremos que existe una plataforma genérica que administra y controla las entidades compartidas.

El concepto de entidad, que define esta capa, es lo que necesitamos para realizar una asociación entre los datos y las operaciones sobre estos. Para tal fin, es necesario que esta entidad posea algunos atributos:

- **Un identificador.** Nos permite ubicar a la entidad dentro del sistema.
- **Método de replicación.** Método por el cual se realiza la coordinación de actualizaciones.
- **Control de concurrencia**. Método que controla los accesos.
- **Datos**. La información objeto de todo el proceso.

Estas instrucciones establecen una asociación que condiciona las futuras operaciones sobre los datos de esta entidad. Por lo tanto cualquier operación recibida proveniente de esa entidad se efectúa utilizando un control de concurrencia y una coordinación de actualizaciones específicos.
Esta asociación no es permanente, porque el diseño de este módulo, permite cambiar el tipo de coordinación mediante una nueva asociación. Sin embargo el mecanismo que permite realizar este cambio no es sencillo. Más adelante explicaremos como funciona.

En la figura 3.8 podemos observar el funcionamiento conceptual del módulo de replicación. La entrada de las operaciones es verificada por un mecanismo que distribuye el flujo basado en el identificador de la entidad a la que pertenecen esas operaciones. Enseguida estas operaciones son ejecutadas por los mecanismos correspondientes (control de concurrencia y política de replicación), y finalmente son transmitidas a otros lugares por medio del modulo de comunicación.

El mecanismo de cargada de políticas, realiza la función de obtener, al momento que se necesita, la política de replicación/actualización y el control de concurrencia correspondiente a la entidad solicitada.

Resumiendo: la labor de este módulo es proveer de las herramientas necesarias para llevar a cabo una coordinación exitosa. Por ello el diseño se basada en el concepto de la reutilización de código que permite la adición, remoción y reemplazo de funcionalidad. El mecanismo para el reemplazo de una política de replicación se explicará con detalle en el capítulo 4.

3.2.3. Proceso autónomo

Un punto importante en el diseño de éste sistema de actualización de entidades, es poder efectuar algunas operaciones sobre los datos de manera automática.

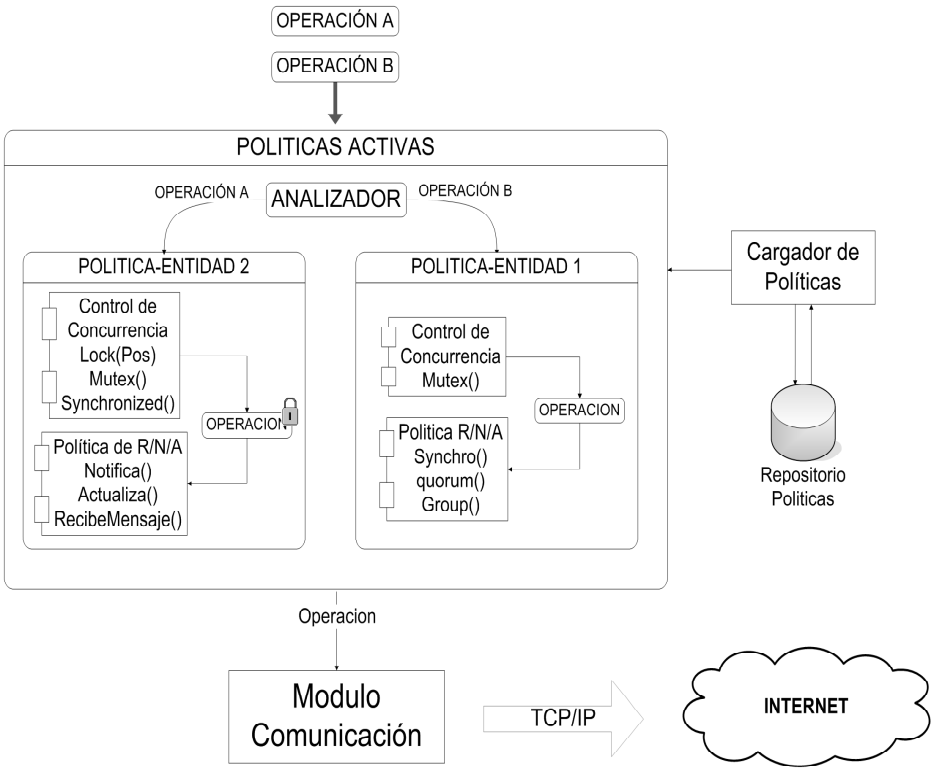

Figura 3.8: Proceso de Replicación

Esta cualidad puede ser aprovechada por alguna política de actualización/notificación, para realizar alguna tarea automáticamente. Sin embargo la implementación de estas políticas pudiera integrar su propio proceso autónomo, depende de los requerimientos o las necesidades de esas políticas.

Un proceso autónomo o (daemon), es un programa independiente que realiza labores de monitoreo u otras tareas sin intervención de factores externos. Poseer un mecanismo de estas características puede resultar benéfico porque reduce la carga a la aplicación de enviar constantes notificaciones o actualizaciones a los demás sitios. El usuario de una aplicación colaborativa también se beneficia, porque puede concentrarse en sus labores productivas.

La función de este proceso autónomo o demonio, es simplemente ejecutar las operaciones que encuentra en un repositorio, el cual funciona como una cola de mensajes. Estas operaciones son almacenadas por alguna política de replicación/actualización de acuerdo a sus requerimientos o algún evento que lo provoque. Estas operaciones también pudieran no pertenecer a ninguna política o control de concurrencia.

Gracias a esta característica es posible además dar soporte al trabajo nómada o desconectado. Por ejemplo Bayou [Edwards, 1997] utiliza una bitácora de operaciones en la que se registran todas las operaciones realizadas. Por medio de esta bitácora puede realizar transformaciones al estado global del sistema porque todas las operaciones llevan un orden. Este orden se preserva aun cuando no existe un enlace físico con otros sitios.

El funcionamiento del demonio (ver figura 3.9), inicia una vez se ha puesto en ejecución el servidor en cada sitio, por lo tanto desde el inicio buscara constantemente alguna operación que ejecutar en el repositorio. Adicionalmente pueden ser agregados mas procesos demonios para agilizar la atención de estas operaciones.

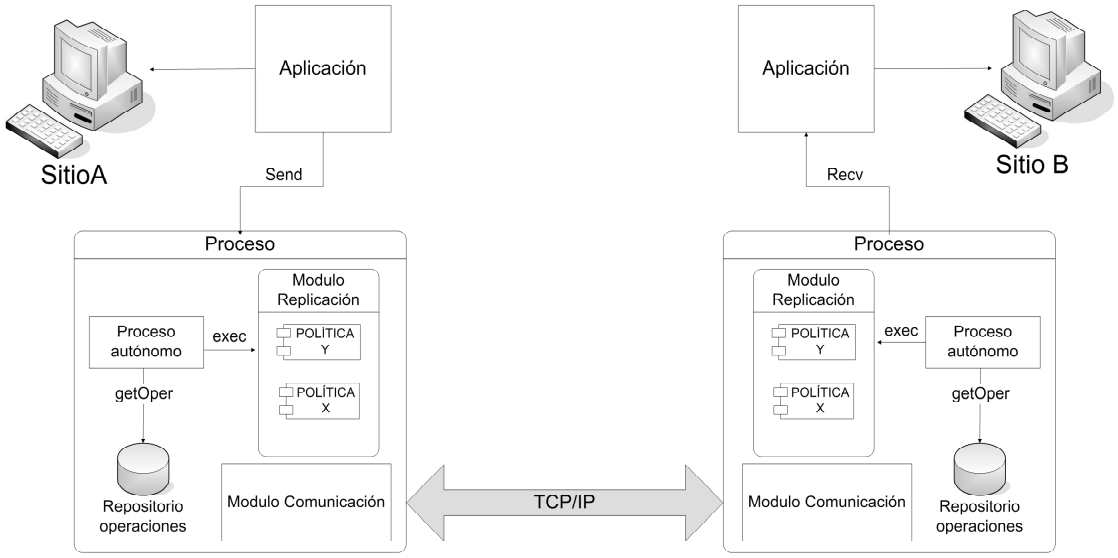

Figura 3.9: Funcionamiento del Proceso Autónomo

El diseño de este daemon (ver figura 3.10), propone un lector de operaciones, un interprete y un ejecutor. La idea detrás de su funcionamiento es similar al modelo productor-consumidor [Tanenbaum, 2003] que realizan escrituras y lecturas en una memoria compartida, en este caso el consumidor es el demonio, que lee las operaciones que el productor agrega.

Este proceso autónomo, permanece en un ciclo infinito por lo que la única manera en que puede ser detenido es al detener el servidor. Este servicio puede ser modificado únicamente al inicio mediante el uso de alguna variable, o archivo de configuración.

Mecanismos mas sofisticados para el mejoramiento de este servicio pueden ser agregados en un futuro, como por ejemplo una red neuronal que indique el mejor momento para enviar información u obtenerla de acuerdo al ancho de banda o características del sitio [Angulo, 2009], contenido de los sitios vecinos, o algún otro factor relevante.

- Este proceso autónomo permite la ejecución automática de operaciones.
- Estas operaciones están registradas en un repositorio temporal el cual es accedido por este proceso demonio.
- Estas operaciones son registradas por algún tipo de evento o política.
- La función de este proceso demonio puede ser aprovechada por las políticas de replicación/actualización implementadas.
- Es posible dar soporte al trabajo nómada mediante el uso de este proceso demonio.

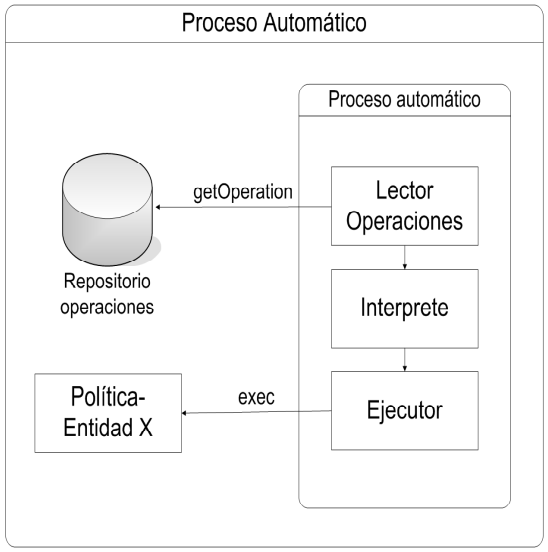

Figura 3.10: Modelo del Proceso Autónomo

3.2.4. Módulo de Datos

Se trata de un conjunto de recursos, que pueden ser archivos, tablas o variables compartidas que nos ayudan al proceso de replicación/actualización/notificación de entidades compartidas. Sin estos recursos, el proceso será aún más complejo de realizar, por ello se ha elegido agruparlos dentro de este módulo. Básicamente se trata de algunos recursos que se utilizan a manera de variables compartidas, las cuales son accedidas por el proceso demonio, las políticas de replicación/actualización, y los mecanismos involucrados en una ejecución de operaciones. Aunque son simples en su estructura son herramientas útiles para completar procesos. Podemos citar las funcionalidades más importantes de este módulo como las siguientes:

- **Carga de archivos de configuración.** Esta característica nos permite cambiar el comportamiento de una política sin necesidad de editar código,

- **Repositorio de operaciones.** Es un almacén temporal que mantiene en estado de reposo, las operaciones sin atender. Utiliza tanto una estructura en memoria como un archivo. Esto permite en caso de una desconexión o falla en el sistema de actualización, retomar el estado anterior a la desconexión.

- **Registro de referencias.** La función de este registro es llevar control sobre las asociaciones entre entidades y políticas. En caso de ser necesario, se obtienen las clases de políticas asociadas a la entidad por medio de su identificador.

- **Serialización.** Es responsable por empaquetar los datos, traducir las invocaciones a métodos en un archivo de texto plano, el cual será sera transmitido por algún método a otros sitios.

La estructura del módulo de datos consiste en varios componentes los cuales son independientes entres si (ver figura 3.11). La estructura del módulo de serialización posee un módulo de creación y lectura de archivos *XML*. Se utiliza para crear el mensaje transmitido entre servidores.

El repositorio de operaciones es una estructura de datos que asigna un identificador único a cada dato que se ingresa. Por medio de ese identificador podremos recuperar esos datos. Los datos que ingresan siguen un esquema de tipo *FIFO*, es decir, se atienden como van llegando. Los datos registrados en el repositorio poseen una marca de tiempo única que es asignada al momento de ingresar a ese repositorio. Esto es útil a aquellas políticas de actualización/replicación que necesiten utilizar este valor, por ejemplo como factor determinante en la resolución de conflictos.

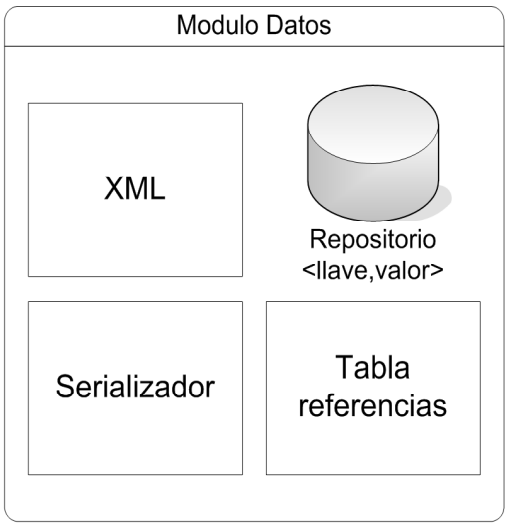

Figura 3.11: Módulo de Datos

3.3. Arquitectura

Anteriormente identificamos las funcionalidades básicas que nuestro sistema debe proveer con el fin de lograr la replicación automática de entidades flexible, extensible y automática. Ahora describiremos a detalle cada una de ellas.

Resumiendo en el apartado anterior identificamos estas funcionalidades:

- Módulo de Comunicaciones
- Módulo de Replicación
- Proceso autónomo
- Módulo de Datos

Las cuales se encargan de un aspecto en particular en el proceso de replicación de entidades. Sin embargo a pesar de ser independientes colaboran unos con otros para realizar su labor. Podemos apreciar la arquitectura modular del sistema la cual mostramos como una arquitectura por capas (ver figura 3.12). Así por ejemplo la capa superior es la interfaz que recibe las instrucciones necesarias para habilitar un comportamiento particular en el módulo de replicación, el demonio lee las instrucciones provenientes de la interfaz y ejecuta las operaciones vía el módulo de replicación, a su vez el módulo de replicación utiliza el módulo de comunicación para realizar el enlace final.

Del lado del servidor ocurre un proceso similar, la operación es recibida por medio del servidor, analizada y ejecutada. Sin embargo a diferencia del proceso de envío, en el de recepción la información es entregada al administrador de entidades.

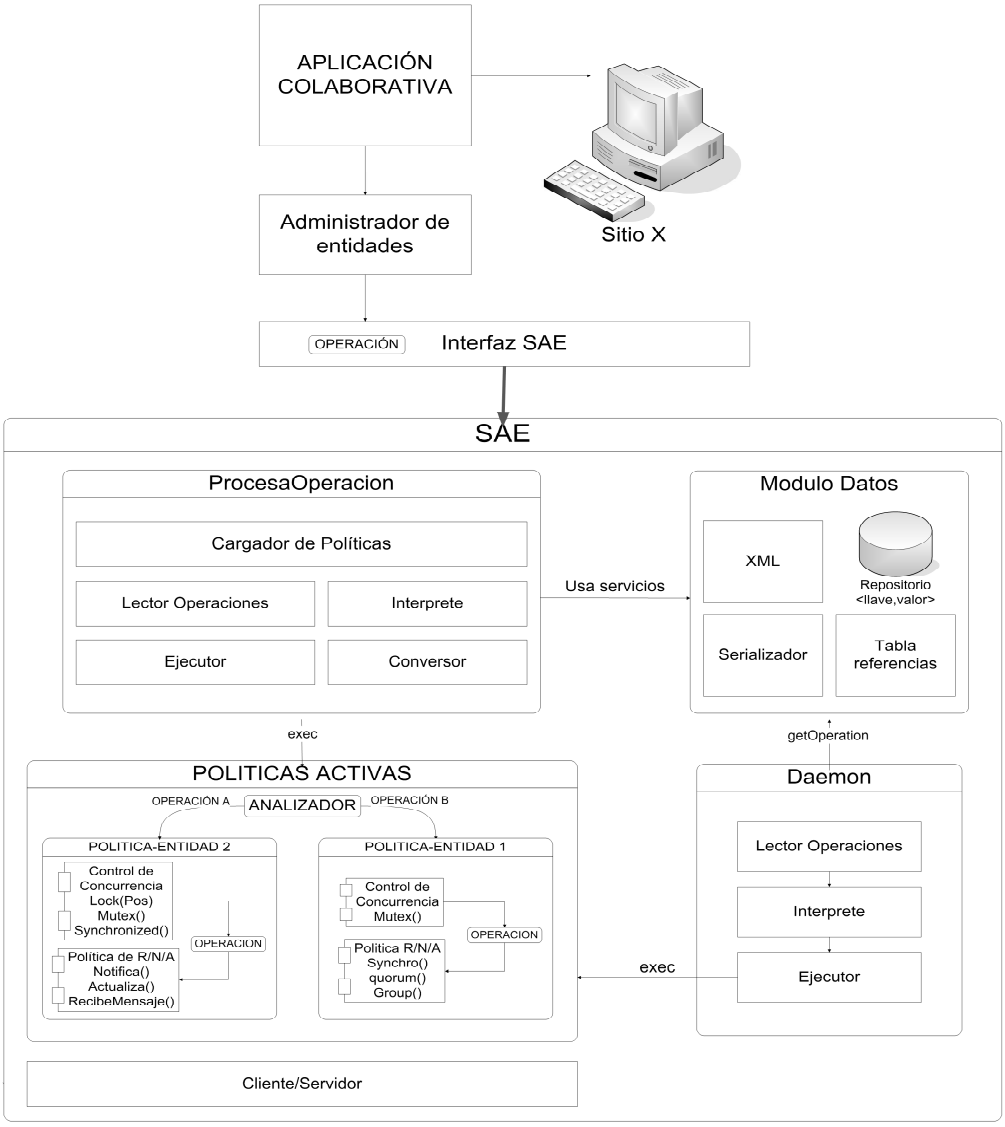

Figura 3.12: Arquitectura del Sistema *SAE*

Los módulos de replicación y automatización se encargan de la parte lógica del sistema,

mientras que el módulo de comunicación es responsable por el enlace entre equipos.

3.3.1. Cliente/Servidor *SAE*

Este módulo utiliza un esquema cliente-servidor para el envío y recepción de mensajes, lo que posibilita dar servicio de tipo peer-to-peer (par a par), aunque no es totalmente necesario que sea así. A nivel de ésta capa los datos necesarios para ubicar los recursos son proporcionados por capas superiores, lo cual permite manipular o crear arquitecturas de distribución diversas. Son las políticas de replicación/actualización/notificación quienes utilizan estas funciones para darles un contexto y un propósito, la capa de administración de entidades define la arquitectura de distribución.

La estructura del servidor *SAE* (Sistema de Actualización de Entidades), es sencilla. Se trata de un clásico servidor multihilos, el cual permanece en un ciclo infinito, a menos que ocurra algo. Permanentemente está en espera de las peticiones de algún cliente, que requiera sus servicios. Utilizamos sockets para establecer los enlaces con los clientes y habilita un puerto que es conocido en el sistema. En determinada situación es posible habilitar un segundo servidor con lo cual puede informarse a los participantes la existencia de este nuevo servidor con la finalidad de distribuir la carga.
El servidor atiende la petición delegando la tarea a un hilo quien atiende al cliente y recibe su mensaje. El mensaje utilizado entre el servidor y sus clientes es un archivo *XML* (Extensible Markup Language), cuyas características para representar información y recuperarla es muy sencilla, además es muy utilizado en la Web. Similar al HTML, solo que podemos definir nuestras propias etiquetas.

La estructura del mensaje podemos observarla a continuación, explicamos los campos, atributos y su función.

```
##-- VER EXPLICACION DE LOS NUMEROS ABAJO --##
<Msg type="APR", entidad="ID333", user="Andy">  #1,2,3
   <Method name="function", numpar="2"/>        #4,5,6
   <Param  name="ID" type="INTEGER">            #7,8,9
         5                                      #10
   </Param>
   <Param  name="ID" type="STRING">
         Hola
   </Param>
</Msg>
```

1. **Tipo de mensaje**. Tratamiento que recibirá en el servidor.

2. **Identificador de la entidad de origen**. Importante para cuestiones de verificación.

3. **Id Usuario**. Validación de un usuario.

4. **Método**. Objetivo del mensaje.

5. **IdMetodo**. Necesario para reconstruir la operación.
6. **Numpar**. Indicador para el constructor de operaciones.
7. **Parámetro.** Indica el nombre y tipo de parámetro.
8. **Nombre.** Opcional para identificar.
9. **Tipo.** Útil en la construcción de la operación.
10. **Valor.** El valor del parámetro. Por lo pronto solamente acepta cadenas y números.

Al utilizar un archivo *XML* para estructurar nuestro mensaje podemos agregar muchas instrucciones las cuales son interpretadas por el módulo de replicación/actualización. Además también podemos definir ciertos atributos que cambian el tratamiento de este mensaje en particular. Por ejemplo podemos indicar si el mensaje debe ser atendido de inmediato o si se debe almacenar en la bandeja de entrada, incluso podemos incluir algunos valores de parámetros que necesitamos para realizar procesos. Esto nos permite un gran número de configuraciones con lo cual evitamos programar constantemente este tipo de comportamiento, únicamente debemos modificar un archivo *XML*, alguna etiqueta, valor y tendremos un gran poder sobre el sistema.

El cliente del sistema *SAE*, además debe realizar la serialización de los mensajes para prepararlos para el envío a través de la red. Es el proceso en el cual se convierte cualquier estructura de datos u objeto en una secuencia de bits que puede ser transmitido por las capas inferiores del modelo *OSI*. Posteriormente una ver que alcanzo su destino se realiza la operación contraria y así recuperar la estructura de datos. Esto se conoce como *marshalling* (aplanado) y *unmarshalling* (desaplanado). La forma en que se establecen estos canales de comunicación es a través de sockets, que es un *endpoint* bidireccional a través de una red que utiliza el protocolo *IP*. Cada *socket* es administrado por el sistema operativo hacia alguna aplicación proceso o hilo. La forma en que se establece un enlace es por medio de una dirección *IP* y un puerto. Cada *socket* posee una única dirección la cual depende del protocolo utilizado, *TCP*, *UDP* y una dirección *IP*. Y dado que el servidor *SAE* puede atender a múltiples clientes concurrentemente es necesario utilizar un protocolo *TCP* que nos permite ésta característica.

Proceso de transmisión.
El proceso de envío de una operación ocurre de la siguiente manera: El módulo de comunicación recibe el paquete y realiza la serialización del mismo, verifica en la lista de distribución proporcionada a los destinatarios del paquete y abre un canal de comunicación para enviar los paquetes a cada destino, una vez recibido el acuse de recibo el cliente cierra la conexión.

Proceso de recepción.
El servidor habiendo establecido un enlace con un cliente recibe el paquete y manda un acuse de recibo al cliente. El servidor realiza el *unmarshalling* para recuperar la estructura de datos y continuar con el proceso. El mensaje es entregado al módulo que interpreta los

datos contenidos en el mensaje. Este proceso puede ser apreciado en la figura 3.13, donde podemos observar que la interpretación de los mensajes es responsabilidad del resto de los módulos, por lo que una vez recibido el mensaje el servidor termina su participación. El contenido de estos mensajes no es alterado durante este proceso.
A continuación explicaremos el tratamiento y la interpretación de esos mensajes realizado por el proceso de replicación.

Figura 3.13: Trasmisión y Recepción de Mensajes Vía *SAE*

3.3.2. Arquitectura de replicación

El módulo de replicación es responsable de mantener actualizadas las entidades, así como de proveer mecanismos para resolución de conflictos. El servicio que proporciona es permitir el uso de múltiples políticas de replicación, actualización, control de concurrencia para múltiples entidades. Esto implica poder realizar cambios, adiciones o reemplazos a estas políticas en tiempo de ejecución.

Estas políticas pueden ser la implementación de algún modelo de replicación: *primary-backup* [Bernstein et al., 1987] [Herlihy and Wing, 1990], *transformación de operaciones*

[Li et al., 2000] [Lee et al., 2002], *maestro-esclavo* [Saito and Shapiro, 2005] etc., o utilizar algún enfoque pesimista u optimista para difundir las actualizaciones.
Esta implementación es responsable de la coordinación de la actualización, replicación, notificación del contenido de las entidades distribuidas, preserva la coherencia de la información (i.e. resuelve conflictos), y transfiere operaciones entre colaboradores. La implementación de estas políticas es responsabilidad del desarrollador de la aplicación por lo cual debe ser implementado con cuidado.
Nuestra arquitectura propone una interfaz genérica la cual deben implementar todas las políticas que sean agregadas a este sistema. Esta interfaz define la implementación de las funciones, método de entrada, datos de salida y parámetros. Los métodos que implementan esta interfaz están sujetos a respetar la definición de las funciones.

Algunas definiciones de funciones que la interfaz provee pueden apreciarse a continuación:

- *add_frg(entidad, datos, posicion).* Agrega una región para su replicación
- *notify(mensaje, from, to).* Notifica sobre algún suceso
- *find_ entity(identidad, usuario).* Busca una entidad
- *del_frg(idEntidad, posicion).* Elimina una región
- *sendAct(IdEntdad, usuario, datos).* Envía una actualización
- *recvAct(datos).* Recibe una actualización
- *add_ entity(IdEntidad, usuario).* Agrega una entidad
- *del_ entity(IdEntidad, usuario).* Elimina una entidad

Pueden ajustarse estas definiciones en un futuro si se requiere.

Esta arquitectura también es responsable de proveer el mecanismo necesario para agregar nuevas políticas de replicación, actualización, control de concurrencia etc. Este mecanismo posee un registro de todas las políticas agregadas al sistema, más adelante en este capítulo explicaremos cómo registrar una política. El mecanismo elige una política del registro, de acuerdo a la solicitud recibida, obtiene la clase y la vincula con una entidad, de ésta forma inicia su funcionamiento.

La asociación o vínculo de una política de control de concurrencia, replicación o actualización permite establecer una especie de "contrato"entre el sistema y la entidad. El sistema se compromete a prestar los servicios de coordinación de actualizaciones con el método solicitado y la entidad a utilizar sólo estos servicios. Para este fin utilizamos el identificador de la entidad proporcionado por la capa de administración y el identificador del usuario propietario de esa entidad. Con ello creamos un identificador único que utilizamos para localizar a las réplicas en todo el sistema.
Por lo tanto al momento de recibir una orden de la capa de administración de entidades

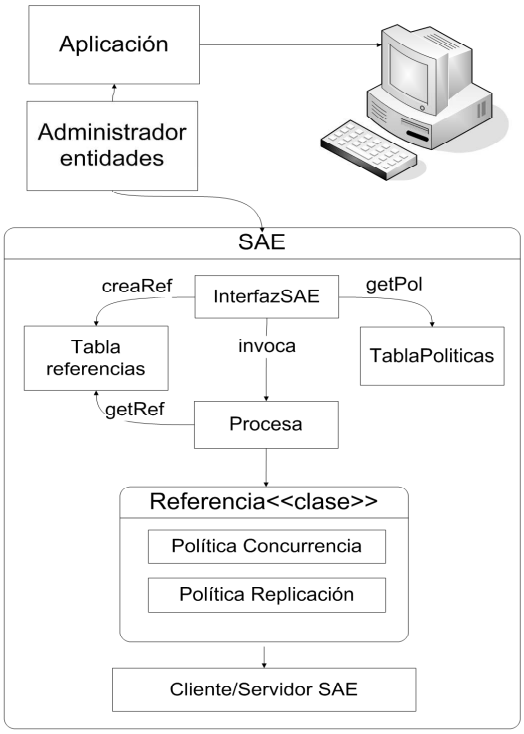

Figura 3.14: Arquitectura de Replicación

solicitando la ejecución de alguna función de replicación, actualización, o control de concurrencia, debemos recuperar la asociación registrada para esa entidad. Con ello obtenemos el método por el cual se debe realizar la replicación, en la figura 3.14 mostramos este hecho. Podemos apreciar que al momento de invocar funciones de alguna política de replicación obtenemos la asociación, y con ello la clase asociada a la entidad. Cabe mencionar que de ésta manera se logra la flexibilidad en el uso de múltiples políticas, porque para cambiar de método de replicación basta con realizar la asociación de la misma entidad con otra política.

El mensaje que utilizamos para transmitir información entre réplicas, contiene un campo el cual indica el tipo de tratamiento que recibirá al momento de ser recibido, el algoritmo 1 muestra la verificación de este campo.

Algoritmo 1 Servidor SAE

```
1: receptorMensajes( tipoMensaje, mensaje )
2: si tipoMensaje='REP' entonces
3:    ejecutaMensaje( mensaje ) **Ejecución inmediata
4: otro si  tipoMensaje='POL' entonces
5:    almacenaMensaje( mensaje ) ** Almacenamiento
6: End
```

Con ello controlamos el flujo que seguirá un mensaje de acuerdo a la política imple-

mentada. Pueden agregarse más casos en un futuro.

Asociación de entidades con políticas.
Una asociación es una vinculación de una entidad con alguna política de replicación, notificación. Esta asociación permite un tratamiento distintivo a los mensajes provenientes y dirigidos hacia la plataforma de administración de entidades. Durará mientras la entidad exista o se indique la terminación de esa asociación.

La creación de la asociación entre políticas y entidades se realiza de la siguiente manera:

- Se invoca la creación de una replica de una entidad con identificador ID_X
- Se proporcionan los datos de la política requerida para su replicación/actualización
- Se realiza la obtención de una clase de políticas, se inicia con los datos anteriores
- Se registra la dirección de referencia de esa clase en una estructura *Hash* en memoria
- El proceso demonio ahora puede atender las peticiones de ésta clase
- Finaliza el proceso de vinculación.

Inicio de la replicación.
Una vez que la entidad ha sido asociada con una política $C/R/A/N$ (Control de concurrencia, actualización, replicación, notificación), cualquier operación entrante que utilice este servicio, deberá proporcionar la identificación de la entidad correspondiente a esas políticas.
Según sea la implementación de estas políticas, la coordinación puede realizarse de muchas maneras. Una política optimista pudiera utilizar un repositorio de operaciones con el fin de realizar actualizaciones de manera asincrónica, mientras que una política pesimista utilizaría bloqueos y una sincronización para la actualización de la información.
La secuencia de eventos de una replicación de datos en la que participan varias clases es mostrada en la figura 3.15. Describiremos los sucesos en las siguientes viñetas:

1. Como primer evento se recibe una invocación a una operación X dirigida a una política Y en la interfaz *SAE*.
2. Se verifica la existencia de la clase de política Y en el repositorio de clases (tabla Hash).
3. Depende de la implementación en este punto la operación puede ser almacenada o ejecutada inmediatamente. Supongamos que es almacenada.
4. La invocación se interpreta de tal manera que pueda ser almacenada en el repositorio de operaciones.
5. Una vez almacenada queda en espera.

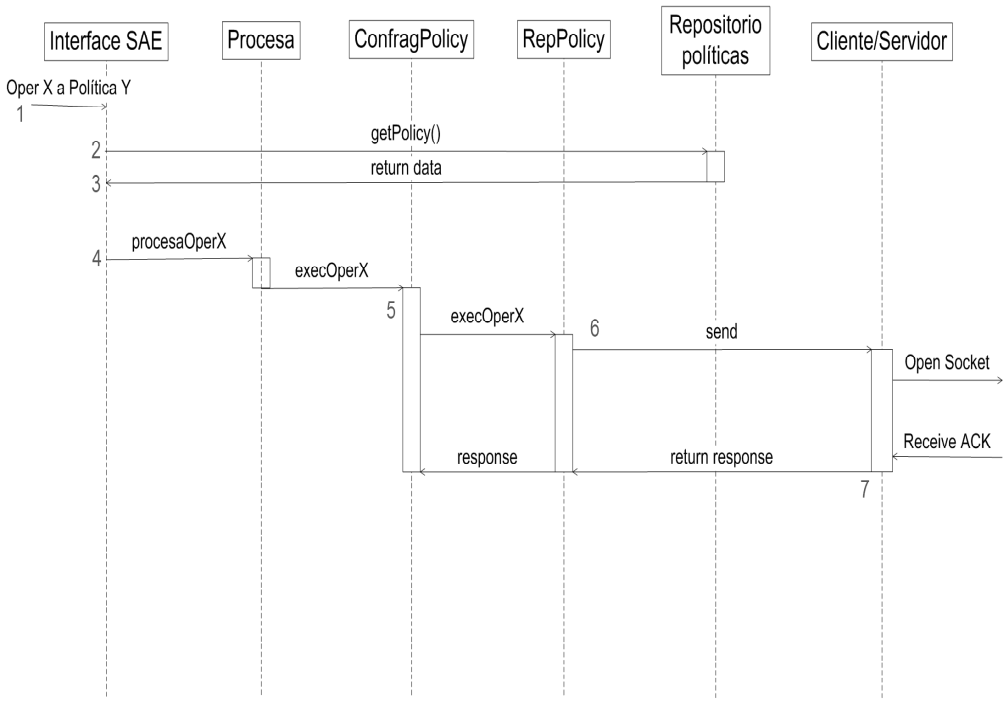

Figura 3.15: Diagrama de Secuencia, Replicación de Datos lado Cliente

6. El demonio verifica el repositorio en busca de operaciones, y obtiene la operación recién registrada.

7. El demonio traduce la operación, obtiene la referencia a la clase y ejecuta la operación. La operación en cuestión es removida del registro una vez finalizado el proceso.

La secuencia de eventos del lado del servidor se muestra en la figura 3.16 donde se aprecia el procedimiento al recibir el mensaje, describimos los sucesos a continuación:

1. EL servidor recibe la operación y la entrega a la interfaz SAE, responde al cliente.

2. Se verifica el tipo de mensaje y se comienza el tratamiento.

3. En este caso el mensaje recibido es interpretado como una operación para una politica pesimista.

4. Se obtiene la asociación entidad-política.

5. Se ejecuta la operación.

Esta secuencia de eventos se repite para toda operación de replicación, notificación, actualización. Difieren los tratamientos que se realizan a esos mensajes por parte de las políticas implementadas.

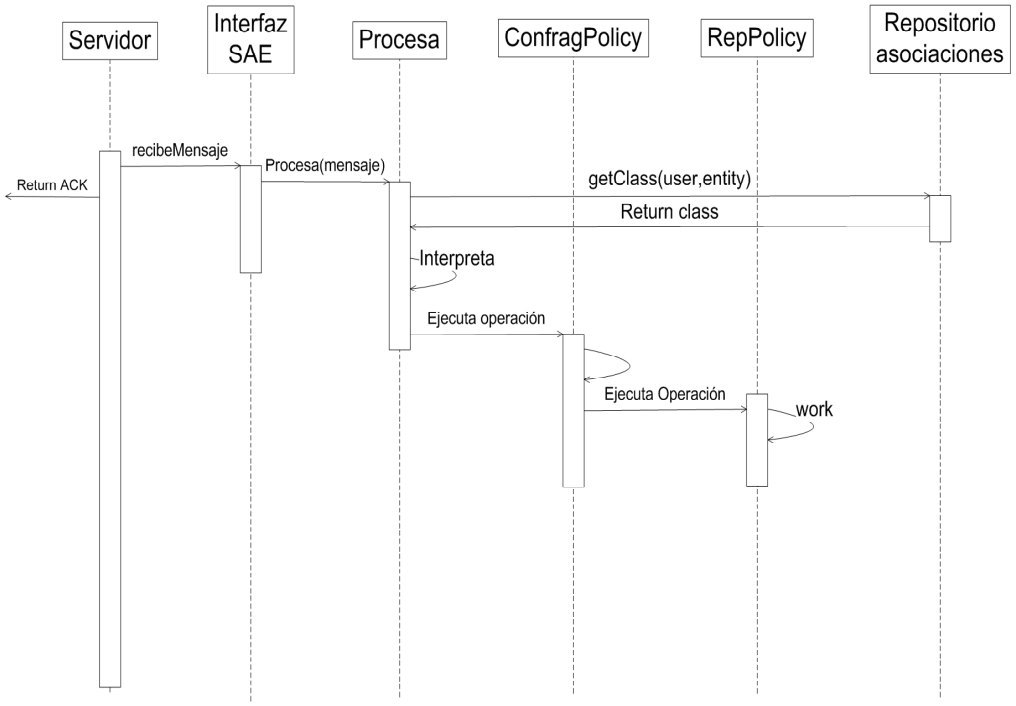

Figura 3.16: Diagrama de Secuencia, Replicación de Datos lado Servidor

3.3.3. Interfaz principal del sistema *SAE*

Es la entrada de los datos, invocaciones, operaciones provenientes del "exterior", con esto nos referimos a fuera del sistema *SAE*, por ejemplo de algún sitio remoto, de la plataforma de administración de entidades, de algún procedimiento automatizado etc.

Su diseño se basa en el patrón de diseño definido en la ingeniería de software como *Facade* [Larman, 2005]. Este patrón *Facade* o Fachada, como su nombre lo indica, es un frente, interfaz si se quiere ver así, que oculta la implementación de los servicios. Su función se limita a la atención a clientes mediante una sola entrada de los datos. Al momento de recibir la operación solicitada delega la responsabilidad de atención al método adecuado, podemos apreciar este proceso en la imagen 3.17, las operaciones provenientes del administrador de entidades son recibidas por esta interfaz cuya tarea es ocultar los mecanismos de niveles inferiores. Se apoya en los repositorios de operaciones y referencias para realizar su labor.

Las interfaces de este módulo están relacionadas al inicio de eventos, y recepción de operaciones.

- *add_frg(entidad, datos, posicion),*
- *notify(mensaje, from, to),*
- *find_ entity(identidad, usuario),*
- *del_frg(idEntidad, posicion),*

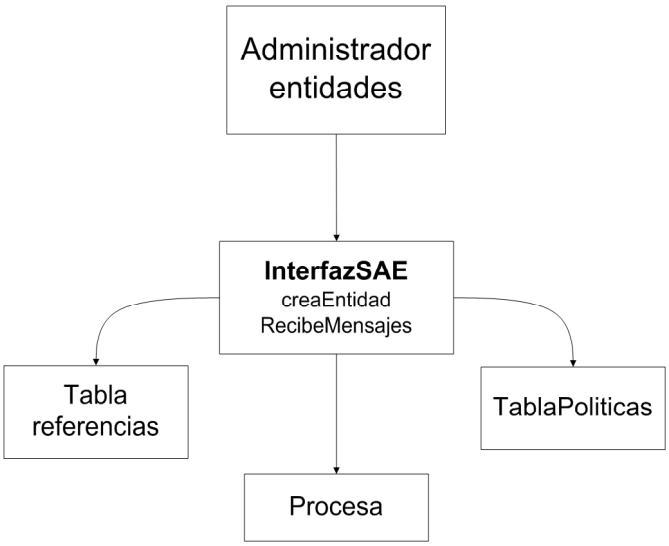

Figura 3.17: Actividad de la Interfaz

- *sendAct(IdEntdad, usuario, datos),*
- *recvAct(datos),*
- *add_ entity(IdEntidad, usuario),*
- *del_ entity(IdEntidad, usuario),*
- *recvMsg(mensaje, fuente)*

Utilizar interfaces permite ocultar el funcionamiento de los módulos, gracias a lo cual son libres de implementación.

Tareas de la interfaz principal.
Delega las operaciones recibidas al método correcto y de ésta manera no se realiza ningún tratamiento a este nivel, por lo que pueden ser modificados los métodos de tratamiento de la información en un momento posterior y la interfaz seguirá ofreciendo los mismos servicios. El tratamiento de las invocaciones y operaciones es delegado a una segunda clase. Esta clase se ocupa de varias tareas: transformar una operación en instrucciones para se enviadas por la red, construir de nuevo la operación en base a las instrucciones del mensaje obtenido, obtener las referencias de las clases, cargar el archivo de configuración, cargar de manera dinámica las clases, crear el enlace de la entidad con la política de replicación.

3.3.4. Proceso demonio (daemon)

Un proceso ejecutado en *background* (segundo plano), es útil cuando se deben realizar tareas sin la intervención de usuarios, otros procesos, o factores externos, de manera pe-

riódica, frecuente y permanente. podemos citar al proceso daemon HTTP que contienen los servidores Web, que permanece en un ciclo infinito en espera continua de las peticiones de los clientes.

En base a este mismo principio funciona nuestro proceso daemon. Su objetivo es proveer, si se requiere, una ejecución de tareas de manera automática. Esta propiedad puede ser aprovechada tanto por el sistema *SAE*, como por las políticas *C/A/R/N*. Sin embargo hacemos notar que estas políticas no están sujetas a su uso forzoso. La implementación de estas políticas pudiera incorporar su propio proceso demonio.

El proceso demonio es útil al sistema *SAE* porque permite, en caso de una desconexion, atender todas las operaciones que se registraron en el repositorio como producto de esta desconexión.
También puede utilizarse para efectuar mecanismos de coordinación entre servidores, verificar el estado general, etc. Casi cualquier cosa que necesitemos porque, como veremos más adelante, posee un mecanismo de interpretación de operaciones que le permite ejecutar cualquier función asignada.

Funcionamiento.

Las operaciones que son registradas en el repositorio, definen un juego de instrucciones, indican por ejemplo la función que debe ejecutarse y los parámetros necesarios para tal efecto. El demonio realiza un proceso de traducción de esas instrucciones para construir la función que finalmente ejecutará.
Estas instrucciones pueden construir cualquier tipo de función, por lo tanto se debe tener cuidado al elegir las funciones y los parámetros a ejecutar. Típicamente las funciones que el demonio ejecutará tienen que ver con actualizaciones en una replicación de información o notificaciones. Por ejemplo en una aplicación colaborativa que trabaja de manera asíncrona, puede llegar a ser molesto tener que apretar un botón constantemente para enviar los cambios que hemos realizado a los demás. Es mejor dejar ese trabajo a un proceso que de manera automática envía las actualizaciones de manera automática. Claro que este proceso debe permitir una configuración de tal manera que funcione como mejor nos convenga, de otro modo solo trasladamos el problema de lugar.

Diseño del demonio

Ya habíamos hablado anteriormente de los componentes de este módulo, pero nuevamente los resumimos a continuación y el esquema se puede observar en la figura 3.18:
El proceso daemon posee:

1. **Lector de operaciones.** Obtiene las operaciones del repositorio de operaciones.
2. **Traductor de instrucciones.** Las cuales obtiene del repositorio de operaciones.
3. **Cargador de clases.** Obtiene la referencia de la clase en memoria.

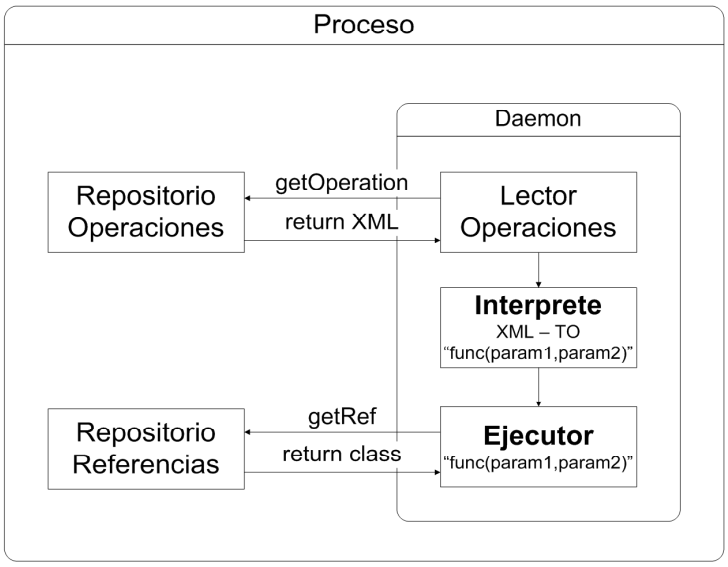

Figura 3.18: Arquitectura del Proceso Demonio.

4. **Ejecutor de operaciones.** Ejecuta la operación.

Estas funcionalidades también están presentes en el sistema *SAE*, pero el demonio también las requiere para realizar su labor y ser independiente de los procesos que se lleven a cabo en el sistema *SAE*, podemos apreciar la secuencia de eventos en la construcción de una operación en la figura 3.19, el demonio ejecutará una y otra vez este proceso de manera indefinida, hasta que se le indique lo contrario.

Hablamos de un archivo de configuración el cual le indica entre otras cosas pero ejemplo el periodo de espera entre ejecuciones, la ubicación del repositorio, etc.

La estructura de este archivo puede cambiar en la implementación, sin embargo aquí proponemos una sugerencia:

```
<daemon timestamp="false" logs="true" demon="true">
  <class rep="..\datos\" file="TO.rb"/>
  <var initS="false" hearthbeat="true" time="50" priority="true"/>
</daemon>
```

Basta con cambiar valores en este archivo para cambiar por ejemplo el periodo entre lecturas de operaciones, iniciar de manera inmediata o esperar por algún evento etc. Las posibilidades son variadas.

Pueden existir varios procesos daemon atendiendo las operaciones, esto puede funcionar así porque el repositorio organiza su contenido por medio de un identificador. Por lo tanto un proceso daemon no interfiere con las operaciones de lectura de otro daemon, ya que estos atienen un registro identificado, por vez. Sin embargo el acceso al repositorio se realiza de manera concurrente, por tal razón y para evitar que dos procesos daemon concurrentemente lean la misma operación se utilizan semáforos lo cual bloquea de manera

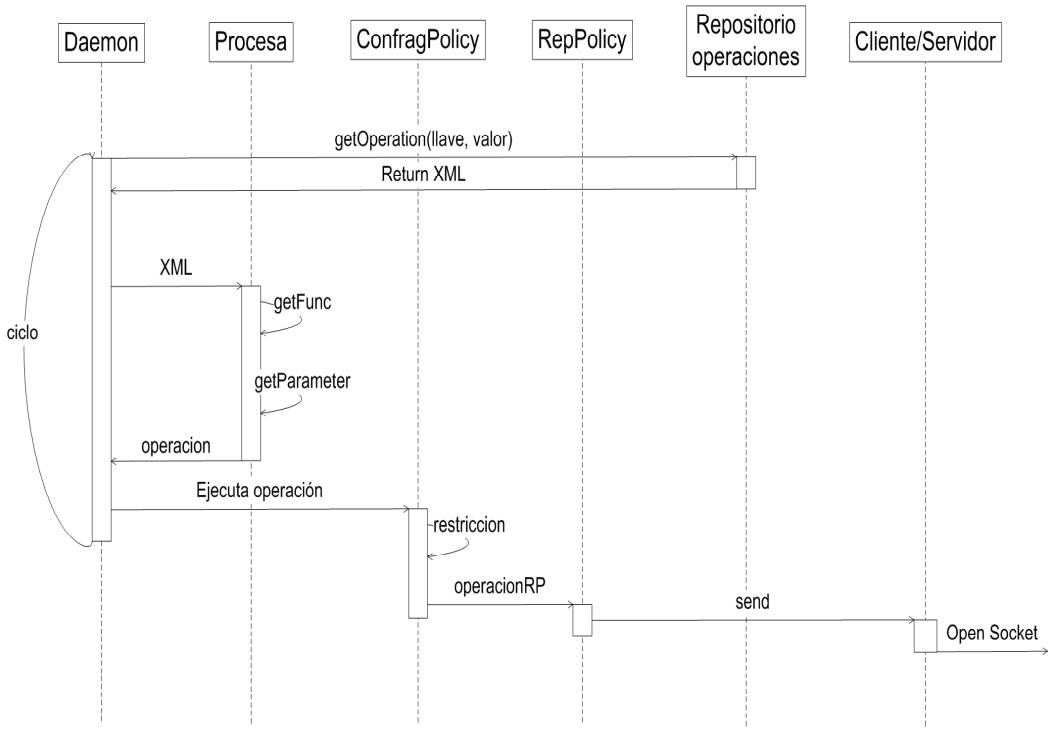

Figura 3.19: Eventos en la Construcción y Recuperación de un Mensaje.

momentánea las lecturas y escrituras y así puede obtenerse o registrarse las operaciones en el repositorio.

El esquema de este proceso de lecturas y escrituras se muestra en la figura 3.20 en el que observa el funcionamiento de varios daemons recuperando y ejecutando operaciones del repositorio. Estos procesos daemon recuperan no solo la operación que deben ejecutar, si no también la clase necesaria para realizar esa ejecución. Esto es posible porque mediante el identificador de la entidad se recuperan todas las clases y operaciones asociadas.

3.3.5. Repositorio de operaciones

El principio básico que nos ha conducido a diseñar un repositorio de operaciones es preservar un "estado"de todas las operaciones, clases y variables utilizadas en el sistema *SAE* en todo momento.
En caso de fallas, podemos recuperar valores, clases y operaciones que permanecieron almacenadas de manera segura en un medio de almacenamiento persistente.

Esta característica puede ser aprovechada para proveer tolerancia a fallas en el sistema, dar soporte al trabajo nómada o llevar un histórico de operaciones [Edwards, 1997] [Theimer et al., 1997], si alguna política de replicación/actualización lo requiere.

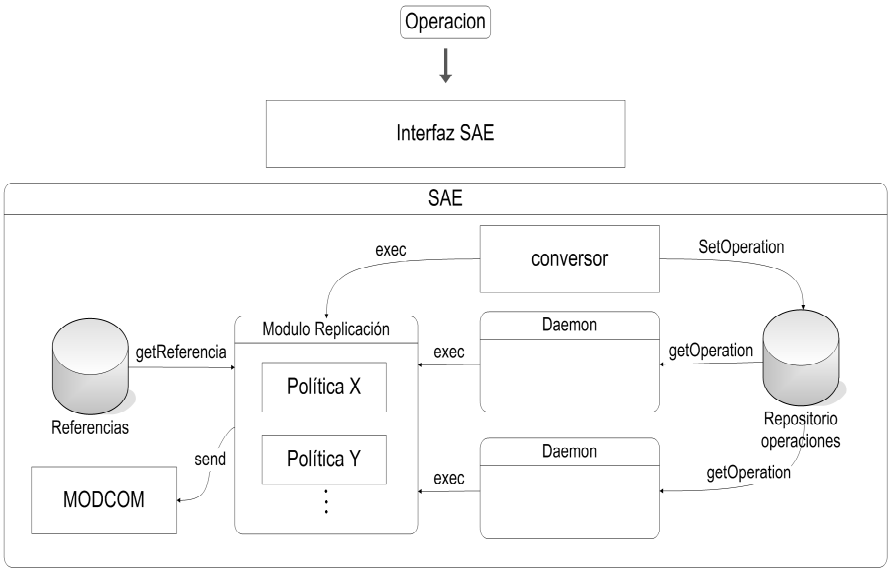

Figura 3.20: Automatización del Proceso de Replicación

Diseño del repositorio.

Es una estructura de datos en la cual por medio de una llave se accede al contendido apuntado por esa llave. Existen muchas maneras de realizar esto, por ejemplo utilizando una tabla Hash, un arreglo o un archivo.

La implementación depende de la política, pero debe cumplir con al menos un requisito: el recurso almacenado debe ser obtenido por medio de una llave. Sin embargo dependiendo de la implementación ésta búsqueda puede ser variar en velocidad de acceso, por ejemplo si se utiliza un árbol estamos hablando que el orden de una búsqueda es de $O(log_2(n))$, mientras que en una tabla Hash puede llegar a ser de $O(1)$ y en un arreglo $O(N)$.

La operación registrada debe contener una llave y un registro del tiempo en que se realizó la operación (marca de tiempo) [Lamport, 1978]. Esto es importante porque a medida que ocurren estas operaciones en todos los sitios, la diferencias de tiempo en la transferencia a otros equipos (latencia), puede diferir en el resultado final de estas operaciones. En la figura 3.21 se muestra lo que ocurre en una transferencia de operaciones entre dos sitios. En algún momento y por varias causas la propagación de una operación tardo mucho tiempo, por lo que el resultado final no es el mismo en los dos sitios.

Por ello es útil registrar el tiempo en que se originó la operación para evitar este tipo de problemas. Existen algunas técnicas para sincronizar operaciones, por ejemplo, usando servidores *NTP*, relojes de Lamport, vectores de tiempo, etc.
La estructura del repositorio es mostrado en la figura 3.22, la cual muestra la estructura de la operación y los accesos a ella por parte del proceso daemon.

El uso de este repositorio se extiende también a preservar el estado de las clases y sus valores. Esto tiene dos objetivos:

- Mantener un estado de todas las clases y los valores de sus variables. En caso de

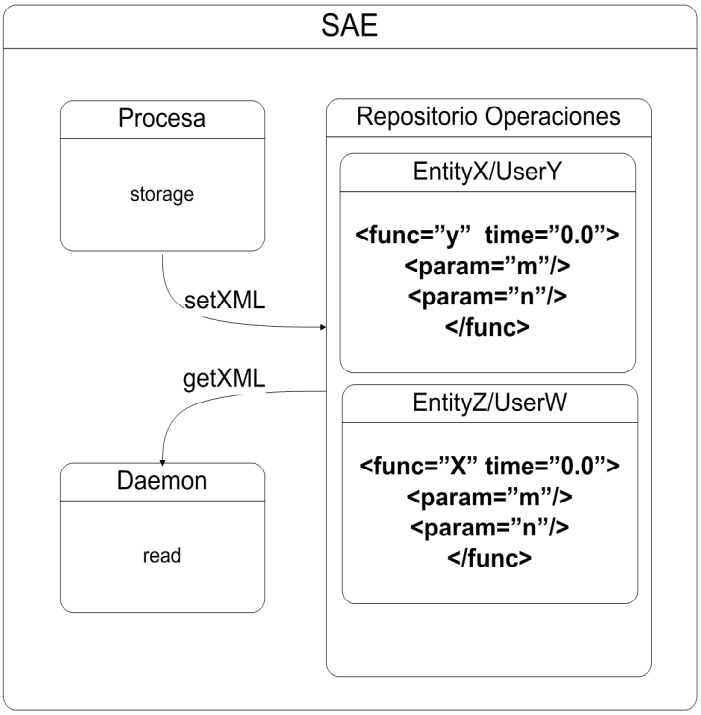

Figura 3.21: Arquitectura Repositorio Operaciones

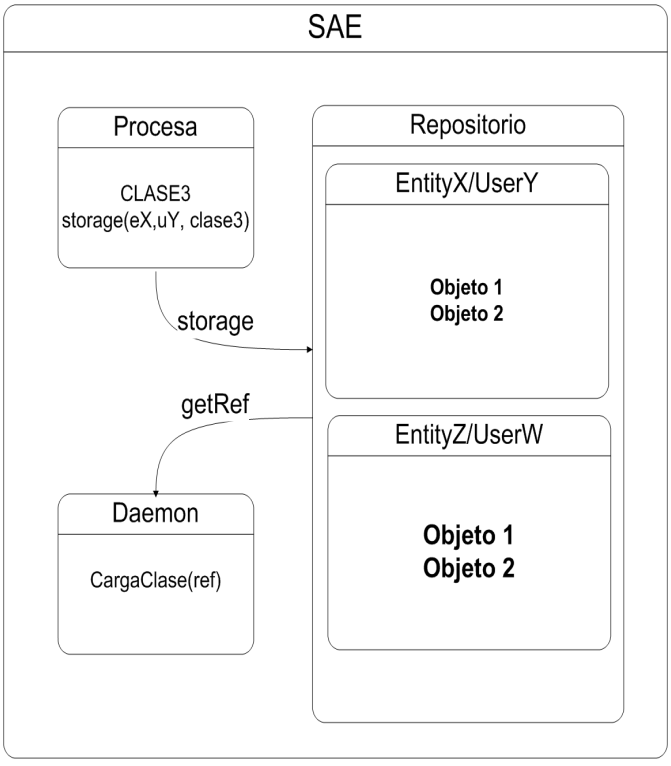

Figura 3.22: Arquitectura Repositorio Referencias

una falla en el sistema, se recuperan todas estas clases y las operaciones momentos antes de ocurrido el incidente.

- Evitar el uso excesivo de memoria. Como todas estas clases ocupan recursos, es posible mantener la clase almacenada y recuperarla solo en el momento que se requiere. Así reducimos la carga en memoria en caso de existir muchas clases.

Por ello en vez de invocar una y otra vez, registramos la referencia a esa clase la cual no es destruida y conserva los valores que se modifican con el tiempo. Es una manera eficiente de recuperar valores en una clase sin tener que recurrir a una invocación constante, además una vez finalizado el flujo de eventos que requirieron esa referencia en primer lugar, ésta no es eliminada sino que permanece en memoria, es visible y puede ser recuperada por cualquier otro proceso.

En la figura 3.23 puede observarse este proceso el cual comentaremos a continuación:

1. El servidor *SAE* recibe un nuevo mensaje
2. El mensaje es entregado a un proceso que lo analiza.
3. Una vez analizado se determina que es una notificación dirigida a una entidad X asociada con una política Y, indicando que existe una actualización en el sitio A.
4. Por medio de la llave formada por el identificador de la entidad se recupera del repositorio la referencia de la clase correspondiente con lo cual se puede obtener la función que se necesita.
5. Se ejecuta la operación.

Soporte a la desconexión o trabajo nómada

El soporte a trabajo nómada permite la producción de trabajo cooperativo en modo "*off-line*"(i.e. sin tener comunicación con otros equipos), y con la posibilidad de difundir las actualizaciones a los demás sitios participantes una vez iniciado el enlace. Estas operaciones se almacenan en un medio de almacenamiento persistente (e.g. archivo, base de datos [Edwards, 1997] [Saito and Shapiro, 2005]), mediante una marca de tiempo universal, es un mecanismo relativamente sencillo para crear un orden en las operaciones. Por eso es posible difundir, tiempo después estas operaciones y aun conservar un orden universal.

Con este fin la política de control de concurrencia utilizada, deberá implementar también un mecanismo de trasformación de operaciones [Li et al., 2000] [Sun and Ellis, 1998], que consiste en aplicar una función de transformación a todas las operaciones requeridas para preservar el estado general del sistema consistente.

Estas funciones de transformación pueden implicar, por ejemplo, tener que retroceder operaciones y transformar las operaciones subsecuentes para adaptarse al cambio. Por ejemplo una operación de escritura a del sitio X y una operación de escritura de b del sitio Y pueden tener resultados distintos cuando se propagan hacia cada uno de estos sitios.

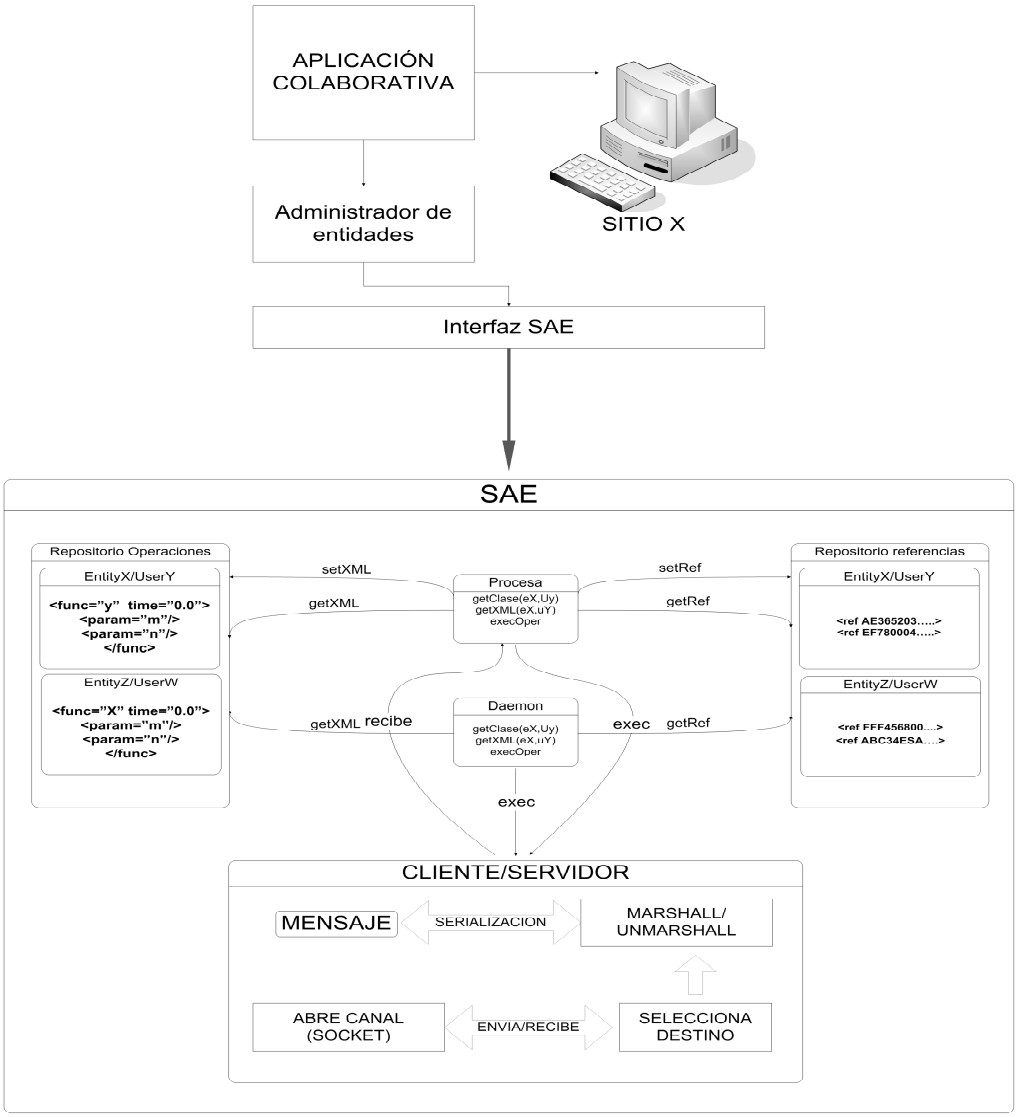

Figura 3.23: Recuperación y Ejecución de un Mensaje

- El sitio X posee ab como resultado de esta propagación,
- el sitio Y posee ba como resultado de esta propagación

Con el fin de arreglar esta inconsistencia el sitio Y puede utilizar el valor de la marca de tiempo para decidir el orden de las operaciones. Si existen discrepancias aplica en este caso una función de transformación. En primer lugar es retirada la operación desordenada aplicando la operación inversa, posteriormente se agrega de nuevo en el orden correcto. Otra técnica es mediante el uso de políticas de control de concurrencia pesimistas, las cuales pueden bloquear partes independientes de la información de manera temporal, evitando así su modificación hasta retirar el bloqueo. Estos bloqueos son emitidos a todas las replicas en todos los sitios que las compartan con el fin de evitar escrituras en esas secciones aun si el sitio origen del bloqueo permanece desconectado.

Este tipo de solución depende mas de la implementación de la política de replicación, notificación y de la coordinación de sus escrituras y lecturas.

Flujo de eventos

Como utilizamos la misma estructura de datos tanto para el registro de las operaciones coo para el registro de las clases, el diseño es idéntico, ver sección *Repositorio de operaciones*. Por lo tanto sólo describiremos el flujo de eventos que se desencadena en una desconexión en los sitios que comparten la misma entidad de información.

Se puede observar en la figura 3.24 a dos sitios los cuales realizan operaciones de actualización. En un momento ocurre la desconexión por lo que comienzan a ocurrir eventos similares en los dos sitios, explicaremos un sitio por vez.

Equipo A
Ocurre una desconexión voluntaria, es decir a propósito. Se detiene el servidor *SAE* y se le indica al daemon que cese su actividad por medio de una variable compartida. Esta variable que es leída constantemente por el daemon, le indica que debe detenerse, de ésta manera finaliza. Sin embargo aun es posible registrar operaciones en el repositorio puesto que estas no dependen del estado de la conexión. Cuando se reinicia la actividad el servidor es activado nuevamente y con ello el daemon que lee nuevamente las operaciones existentes en el repositorio. Este proceso es idéntico en caso de una falla eléctrica porque las operaciones también son registradas en un medio persistente de almacenamiento, un archivo. Por lo que al momento de iniciar el sistema se carga ese archivo y se obtienen las operaciones nuevamente.

Equipo B
El cliente/servidor *SAE* no puede hacer contacto con el equipo *A*, después de dos intentos se origina un evento de error, con lo cual se actualiza la tabla de distribución y se suspenden los envíos a esa replica. Cuando el servidor desconectado haga nuevamente contacto con este la tabla de distribución de actualizara de nuevo permitiendo la actualización nuevamente.

Para darle sentido y orden a las operaciones todos los sitios deben funcionar bajo un mismo sistema de tiempo porque debe de existir una sincronización con el fin de poder implementar orden en las operaciones realizadas, es decir una hora universal en la cual estén basada todas las operaciones, (i.e. @beats, *UTC*, *NTP*).

Por lo tanto una vez tomando en cuenta que se posee un horario universal, podemos efectuar operaciones en sitios distribuidos geográficamente con la certeza que todas operaciones realizadas llevan un registro ordenado, incluso si algún sitio permanece desconectado. Lógica de la resolución de conflictos es responsabilidad del programador de políticas de replicación, ya que es ahí donde debe programarse el tratamiento de las inconsistencias, para fines demostrativos y por falta de tiempo sólo hemos incluido un mecanismo básico de resolución de conflictos.

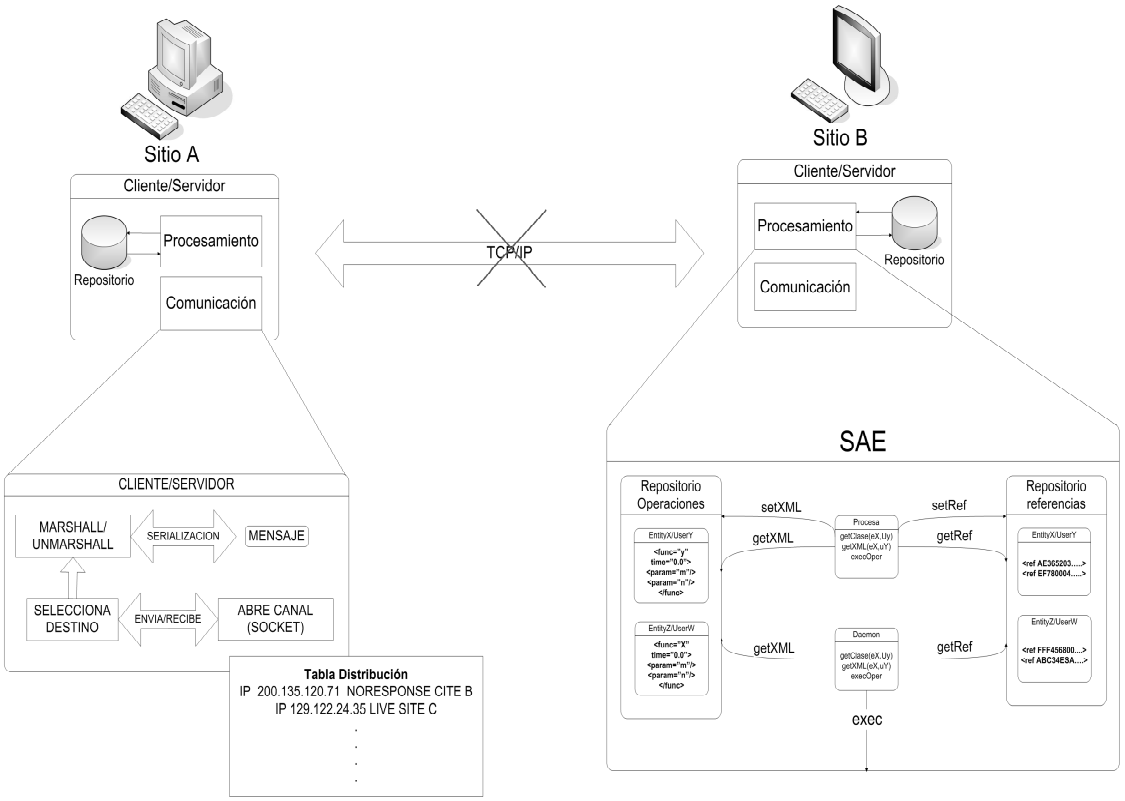

Figura 3.24: Soporte a la Desconexión

3.4. Políticas de replicación/notificación y control de concurrencia

La política de replicación es la implementación de un protocolo de consistencia de datos que permite que todas las replicas distribuidas permanezcan iguales aun si se realizan modificaciones en varios sitios a la vez. Ya vimos algunas de las técnicas utilizadas para preservar la consistencia.

Enfoque Pesimista.
El control de concurrencia nos permite establecer bloqueos, parciales o totales, a regiones de información. Existen algunas técnicas que consisten en utilizar candados, bloqueos o restricciones. Por ejemplo para preservar el orden lo más común es proteger un área de trabajo contra escrituras no deseadas, esto se logra con el uso de restricciones que pueden ser aplicadas a nivel de usuario o recurso.
Con un control de acceso adecuado se puede evitar escrituras simultáneas a la misma sección de un documento. El problema es que sin una implementación adecuada se puede bloquear esa área por siempre.

Enfoque Optimista.
El uso de marcas de tiempo ayuda en la resolución de conflictos debido a que casa opera-

ción realizada se registra en un tiempo determinado, por lo tanto a nivel global todas las replicas que comparten la misma hora universal saben con certeza el momento en que se realiza cada operación y de esa manera realizar ajustes para presentar el orden de cada operación, aun si no llegan en una secuencia similar.

3.4.1. Política de replicación/notificación implementada.

El algoritmo implementado es una variación del algoritmo *anti-entropy* utilizado en Bayou [Theimer et al., 1997] [Theimer et al., 1995] [Edwards, 1997], la cual es una política optimista que utiliza bitácoras ordenadas y repositorios para establecer un estado consistente entre servidores.

Utilizamos un esquema *push* para transmitir las actualizaciones realizadas hacia las réplicas registradas en la tabla de distribución. Las copias que ya poseen esa actualización rechazan la notificación proveniente de algún otro nodo por lo que no la transmiten de nuevo.

Este tipo de propagación utilizando un modelo epidémico tarda $log_2(n) + ln(n)$ para esparcir todas las actualizaciones.

La política posee funciones de envío y recepción que la hacen comportar como un *P2P*.

- **Funciones de envío.** Las funciones de envío tienen por objetivo transmitir información a las réplicas con las que tiene contacto. Se utilizan para enviar notificaciones, actualizaciones, datos, eventos, etc.

- **funciones de recepción.** Estas funciones reciben los mensajes, las actualizaciones, los datos, las notificaciones etc., provenientes de las otras réplicas. Las funciones de recepción también tienen por objetivo, realizar los mecanismos de resolución de conflictos.

El siguiente algoritmo muestra el diseño de la función de notificación que es utilizada para establecer comunicación con otras réplicas. Esta función es ajenas a la representación de los datos:

```
notificacion( target, message )
    direccion = Busca_en_TablaDistribucion( target )
    ack = @comunicacion.notifyto( objetivo, "#{message}/#{direccion}" )
    si ack == falso
        actualiza TablaDistribucion( "sin respuesta" )
    otro
        retorna resultado
fin_notificacion
```

Para completar la comunicación utiliza el módulo de comunicación quien finalmente realiza el empaquetado de la información para ser transmitido entre sitios. Dependiendo de la implementación de la política el proceso de envío y recepción puede cambiar.

La recepción de mensajes posee el siguiente esquema:

```
recibeMensaje( mensaje )
  caso 'ADE' %agrega entidad
      autentifica                  #AGREGA ENTIDAD
      tablaDist.agrega( entidad )
  caso 'FRG'                       #ESTABLECE REGION
       agregaPosicion(posicion,lock)
  caso 'HRB'                       #NOTIFICACION DE ESTADO
       tablaDist.actualiza(entidad, source)
end_recibe
```

Aquí se interpretan los mensajes que desean notificar algún evento a la política. Sin embargo gracias al diseño del interprete de mensajes, puede también realizar contacto con la política mediante la invocación de algún método en particular. Supongamos que se requiere ejecutar de manera remota algún método en la política de replicación. Entonces en la construcción del mensaje del sitio local se indica el método destino junto con los parámetros necesarios para ejecutar el método. Posteriormente al ser recibido en el sitio remoto se traducen estas instrucciones en una operación, enseguida se crea la invocación con lo cual se completa la tarea como explicamos en la sección *Cliente/Servidor SAE*.

La diferencia que existe en el destino de esos mensajes depende de la implementación de la política de replicación/notificación. La interpretación depende por ejemplo si se requiere una notificación de estado, el establecimiento de alguna restricción, etc.

Política de control de concurrencia.

El diseño de esta política de control concurrencia sigue un esquema de tipo optimista que utiliza marcas de tiempo como factor determinante en la resolución de conflictos.

La política posee funciones de envío y recepción que la hacen comportar como un *P2P*.

- **Funciones de envío.** Las funciones de envío tienen por objetivo transmitir las restricciones y los bloqueos a las réplicas con las que tiene contacto. También posee métodos de búsqueda de entidades, bloqueos etc.

- **funciones de recepción.** Estas funciones reciben las notificaciones de bloqueos y restricciones provenientes de otra réplica. Además los datos recibidos deben ser verificados con el fin de prevenir inconsistencias. En caso contrario se aplica un mecanismo de resolucion de conflictos.

Esta política se basa en el establecimiento de bloqueos, con lo cual se protegen los datos de las escrituras de otras réplicas. El bloqueo de una región implica definir la posición y el estado del bloqueo. La posición puede representar desde coordenadas, como en el caso de un pizarrón colaborativo, hasta el día actual, en el caso de agendas colaborativas. El establecimiento de una nueva región implica verificar que la región no se encuentre protegida previamente. Una vez establecido este bloqueo se propaga la notificación a todas las réplicas para que establezcan también el mismo bloqueo en sus copias locales.

Mostramos el diseño de una función de control de concurrencia que establece una nueva región contra operaciones de escrituras provenientes de otros sitios (ver algoritmo 2).

Algoritmo 2 Pseudocódigo para establecer una región

```
1: begin
2: verificaRegion( posicion, usuario, entidad )**verifica la disponibilidad de la región
3: si region = disponible  entonces
4:    estableceCandado( posicion ) **establece el candado
5:    notificaTodos( 'estableceCandado', entidad, datos, posicion )**notifica a las replicas
6: end
```

Posteriormente del lado del servidor ésta notificación es recibida y el fragmento establecido (ver algoritmo. 3)

Algoritmo 3 Pseudocódigo para establecer un candado

```
1: begin
2: candado = verificaCandado( posicion,entidad,usuario ) **verifica en los datos
3: si candado = false entonces
4:    estableceCandado( posicion, entidad, usuario ) **establece el candado y agrega los
      datos
5: end
```

Con el fin de proporcionar mayor control sobre los datos contenidos dentro de esta política creamos una representación utilizando *XML*. Esto nos permite ingresar varios tipos de datos con diferentes propiedades y permite representar distintas regiones y establecer atributos particulares para cada región. El diseño propuesto para esta estructura es la siguiente:

```
<Coord>
  <Fragmento id="", posI=0, posF=0, lock=false>
    valor
  <\Fragmento>
  <Fragmento id="", posI=0, posF=0 lock=true>
    valor
  <\Fragmento>
<\Coord>
```

Es más sencillo llevar un control sobre los datos contenidos dentro de esta estructura, y en caso de necesitar algún otro parámetro o atributo puede modificarse de manera sencilla.

3.4.2. Cambio de política

El cambio de política de replicación implica una serie de procesos que involucran a todas las replicas de una entidad, es un cambio coordinado cuyo objetivo es reemplazar la lógica del protocolo de replicación por alguna otra. En este breve apartado explicaremos el proceso de cambio de política de replicación, el cual característica que soporta este diseño.
Anteriormente habíamos mencionado que el módulo de replicación permite el cambio de

políticas puesto que el diseño se basa en la idea de componentes. Estos componentes son intercambiables, por tanto pueden prestar diferentes funcionalidades para alguna replica en distintos momentos. En este apartado explicaremos brevemente el flujo de eventos que ocurren en el cambio de una política de replicación la imagen se muestra en la figura 3.25 y la explicación viene a continuación:.

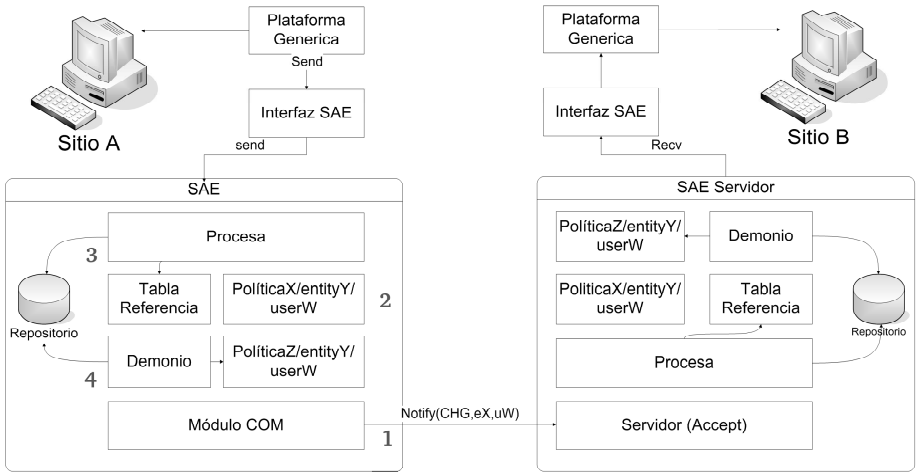

Figura 3.25: Cambio de una Política de Replicación

1. Para que ocurra un cambio en la política de replicación deben ocurrir primero un acuerdo entre todas las replicas de la misma entidad que utilizan la misma política. Por ello se emite una notificación indicando este hecho a todas las replicas, es similar al envío de actualizaciones; eventualmente todas las recibirán. Todos los sitios que reciben esas notificaciones envían sus sitios registrados esa misma notificación, excepto al origen de la notificación. Cuando todos los sitios han sido notificados se realiza el cambio.

2. Se crea una nueva asociación con la entidad, se obtiene la clase y se inician las variables en cada sitio.

3. Se registran las operaciones para la política nueva, excepto para la política anterior.

4. Todas las replicas envían una notificación indicando que el cambio se completo y una vez que todas reciben esa notificación comienza la replicación con la nueva política.

La aplicación cooperativa puede notar alguna lentitud en la actualización de su información puesto que durante ese tiempo no se entrega ninguna actualización, así que el tiempo que dure el cambio depende también de las condiciones de la red.
Puede modificarse el esquema del cambio de políticas si por ejemplo el cambio se realiza por partes es decir, cada notificación de cambio de política que llegue a su destino puede disparar este evento, realizar el cambio e inmediatamente comenzar el funcionamiento de la nueva política es decir todas las operaciones se registran bajo ésta nueva política. Al no recibir nuevas operaciones la política anterior finalmente se quedara vacía por lo que

sera eliminada.
Este proceso en todos los sitios se completara en todos los sitios aunque esto aun no ha sido probado.
En este capítulo se mostró el diseño propuesto para crear un sistema de actualización de entidades replicadas el cual está basado en la idea de la reutilización, en proporcionar características flexibles y y extensibles que pueda aceptar nuevos adiciones a su funcionalidad básica, aceptar nuevas configuraciones y nuevas políticas de replicación.
El modelo propuesto es original puesto que no tenemos conocimiento aun, de que existe otro similar. Se han tomado ideas de varios trabajos y se han fusionado aquí para crear algo innovador.

En el siguiente capítulo mostramos la implementación del sistema, los métodos, procedimientos, interfaces y lenguaje de programación utilizado que nos provee de algunas ventajas con respecto a otras plataformas. Finalmente aterrizaremos este concepto en algo tangible y mostraremos la gran flexibilidad que nos provee mediante las pruebas de funcionamiento. Se justificara además el uso de la herramienta de desarrollo y algunas técnicas utilizadas.

Capítulo 4

Implementación del sistema de actualización de entidades SAE

En el capítulo anterior mostramos las funcionalidades de los módulos y sus interacciones, las características del sistema de replicación/actualización *SAE*, estudiamos las características de flexibilidad para agregar políticas de replicación/actualización/notificación, así como el proceso que realiza la automatización de las actualizaciones. En este capítulo explicamos la implementación de ese diseño, las soluciones a las que llegamos y las ventajas de la solución propuesta. El uso del ambiente de desarrollo elegido fue un punto importante ya que nos facilitó la implementación de módulos flexibles, confiables, portables y de mantenimiento sencillo.

Este capítulo está estructurado en cinco partes. Se describe en primer lugar el lenguaje de programación utilizado sección 1.1.1, enseguida se describe el procedimiento para la adquisición de información, configuración de archivos y la representación de los datos en el repositorio de referencias. En la sección 1.3 mostraremos la implementación de los componentes clave del sistema *SAE*, procesador de mensajes, políticas de replicación y el proceso demonio. Explicaremos también con un ejemplo el principio y fin de una notificación entre sitios que comparten una entidad.

4.1. Lenguaje de programación utilizado

Con el fin de lograr todos los puntos que nos propusimos, necesitamos utilizar un lenguaje que nos proporcione capacidades flexibles y extensibles, es decir, que nos permita la manipulación de los objetos para modificar su comportamiento y además de manera sencilla, con el fin de fomentar la reutilización y extender la vida útil del código [Sommerville, 2005]. La presente tesis de maestría usa el paradigma orientado a objetos [Cook, 1990] el cual permite producir código de calidad y fomentar la reutilización. Cada objeto es considerado como una entidad que puede: interactuar con las demás mediante paso de mensajes, actuar con diversos roles y contener sus propios procesos y estructuras de datos.

4.1.1. Ruby

Este lenguaje nos proporciona todas esas características que necesitamos: es flexible, todos los tipos de datos son un objeto incluidas las clases y tipos que otros lenguajes definen como primitivas. Toda función es un método, las variables siempre son referencias a objetos.

Ruby nos permite programar de manera procedural, orientado a objetos y funcional. Soporta introspección, que es la habilidad de examinar el estado, la estructura de los objetos y alterar sus comportamientos. Es un lenguaje interpretado lo que significa que el tipo de datos se establece en tiempo de ejecución y puede ser modificado. Ruby no requiere sobrecarga de funciones, los parámetros pasados a un mismo método pueden ser de distinta clase en cada llamada a dicho método.

Para resumir las características más importantes de Ruby son:

- Es un lenguaje orientado a objetos,
- Admite sobrecarga de operadores,
- Permite el uso de hilos de ejecución simultáneos en todas las plataformas,
- Ofrece la posibilidad de redefinir los operadores,
- Acepta la carga dinámica de DLL/bibliotecas compartidas,
- Es portable, y
- Soporta alteración de objetos en tiempo de ejecución.

Ruby provee facilidad para implementar de manera sencilla procedimientos para establecer conexiones entre servidores y clientes, serializacion de objetos y ofrece capacidades de manipulación de objetos en tiempo de ejecución. Con ello logramos reunir las características que necesitamos para el cumplimiento exitoso de los requerimientos del sistema que sustenta la presente tesis de maestría.

4.1.2. Metaprogramación.

En este apartado describimos esta característica distintiva que Ruby ofrece. La metaprogramación consiste en la habilidad de modificar el comportamiento de un programa en tiempo de ejecución. Esta característica permite la sobrecarga de operadores, redefinición de métodos, variables, etc., que son interpretadas y utilizadas en tiempo de ejecución. Gracias a esta característica podemos ser capaces de incorporar nuevas clases que introducen nuevo comportamiento en cualquier momento.

Como ejemplo podemos ver el siguiente código 1 en el cual podemos apreciar la sobrecarga de operadores. La clase prueba posee un método llamado *foo* el cual solo despliega una cadena de texto. Adicionalmente el procedimiento *define_method* permite la sobrecarga de la clase permitiendo definir un nuevo método en tiempo de ejecución. Así podemos observar que es posible incorporar nuevos métodos en una clase de Ruby sin necesidad de recompilar el código.

Código 1 Sobrecarga de operadores

```
# Demostracion de la sobrecarga de
# operadores en Ruby
Class Prueba
  #M\'etodo de la clase
  def foo
   "foo"                  #despliega una cadena
  end
end

Salida en consola:
p = Prueba.new    # instancia de la clase
Prueba.send :define_method, :bar do    #definicion dinamica
end

puts p.foo        # invocacion del metodo "foo"
puts p.bar        # invocacion del metodo "bar"
>> foo
>> bar
```

Este mecanismo también conocido como *reflection* [Flanagan and Matsumoto, 2008] incluye propiedades de auto-examinación, auto-modificación y auto-replicación. Estas características posibilitan el cambio de la estructura de un programa en tiempo de ejecución lo cual influye en el comportamiento del mismo. A través de la meta-información se mantiene un registro de la estructura de un programa almacena, por ejemplo, el nombre de los métodos, de la clase, de las dependencias y el comportamiento. Con esta información, mientras es ejecutado un objeto puede ser inspeccionado con el fin de descubrir las operaciones que soporta y posiblemente incorporar nuevas operaciones.

4.1.3. Herramientas utilizadas

El desarrollo del sistema *SAE* se realizó utilizando la versión de Ruby 1.8.6, la cual incorpora librerías estándar para la creación de los *sockets*, *threads*, y elementos de sincronización de acceso a variables compartidas (*mutex*).

Sin embargo se incorporaron algunas herramientas cuyo uso nos facilitó en gran medida el desarrollo de algunas funciones.

PSTORE

Es un mecanismo de almacenado persistente, basado en archivos que tiene la propiedad de almacenar jerarquías de código en archivos de flujos de datos por medio de una llave.

Gracias a esta característica podemos almacenar objetos en un estado transitorio, es decir, podemos recuperarlos como si se trataran de objetos en memoria. Utilizamos esta

herramienta como soporte para el repositorio de operaciones, que en caso de falla tenemos un respaldo del estado de las operaciones realizadas y las instancias de los objetos (ver código 2).

Código 2 Uso de Pstore

```
store = PStore.new("../datos/BES.dat") %Abre el archivo
store.transaction do   %operaciones a base de transacciones
  cadena = "Hola"
  store['entrada'] = cadena  %almacena el objeto en el archivo
end
```

La explicación línea por línea:

1. Creamos un nuevo objeto de tipo *pstore*, iniciamos la clase con la ruta del archivo. Si el archivo no existe es creado.
2. El método transacción establece las operaciones de escritura y lectura a el archivo. Como las operaciones se realizan en base a las transacciones, no hay estados transitorios, es decir que la información de escritura y lectura se realiza inmediatamente.
3. Creamos un nuevo objeto.
4. Como se utiliza una tabla *Hash* para registrar los objetos usamos la llave *entrada*, de esta forma se recupera también.

XML Mapping

Librería extensible para la creación de objetos *XML*. La definición de estos archivos XML se realiza por medio de clases que son interpretadas para dar forma a estos archivos *XML*.

Esto nos facilitó la creación de una estructura de datos flexible y serializable en la que pudimos transmitir información de fácil acceso entre sitios. Con ello el mensaje transmitido entre sistemas *SAE* resulta mas completo, seguro y efectivo porque se puede detallar por medio de etiquetas valores de objetos que pueden ser recuperados en el sitio destino.

Podemos definir la estructura de un árbol *XML* por medio de una clase esquema, la cual define los nodos de este árbol y sus atributos. Por ejemplo tenemos la siguiente clase que define el árbol *XML* que utilizamos para representar el mensaje utilizado en el *SAE* (código 3).

El arbol que genera puede observarse en el codigo 4.

Utilizamos la versión 0.8.1 de *xml-mapping* disponible en:

- *http://xml-mapping.rubyforge.org/*

Código 3 Esquema del mensaje

```
require 'xml/mapping'
class Metodo; end
class Parametro; end
class Info; end

class Msg
  include XML::Mapping

  #-- NODOS ELEMENTOS DEL XML --#
  text_node :reference, "@reference"       #ATRIBUTO
  text_node :tipo, "@tipo"                 #ATRIBUTO
  object_node :metodo, "Metodo", :class=>Metodo #NODO--REFERENCIA A CLASE
  #--DEFINIMOS UN ARREGLO DE NODOS DE TIPO PARAMETRO--#
  array_node :parametros, "Parametros", "Parametro", :class=>Parametro

end

#--DEFINIMOS EL NODO METODO--#
class Metodo
    include XML::Mapping
    text_node :name, "name"
    numeric_node :npar, "@npar"
end

#--DEFINIMOS EL NODO PARAMETRO--#
class Parametro
    include XML::Mapping

    text_node :tipo, "@tipo"
    text_node :txtPar, "PTxt"
    numeric_node :numPar, "PNum"

end
```

Código 4 Estructura del mensaje

```
<Msg>
  <Metodo name="" npar=0>
  </Metodo>
  <Parametros>
    <Parametro tipo="">
      <Ptxt></Ptxt>
      <Pnum></Pnum>
    </Parametro>
  </Parametros>
</Msg>
```

Ruby es multiplataforma, y está disponible en Windows, Linux y OS X, utilizamos la versión 1.8.6, que puede ser descargado desde la página del autor

- *http://www.ruby-lang.org/es/downloads/*

Pstore esta incluido en las librerías de Ruby 1.8.6

4.2. Diagrama de clases general

El diagrama de clases general (ver figura 4.1) muestra las clases que componen al sistema *SAE* y las clases que corresponden a la interfaz de las políticas de concurrencia y replicación/actualización.

Esto es así porque la implementación de esas políticas no forma parte del diseño del sistema *SAE*. La implementación de estas políticas es libre y a gusto del programador. Sin embargo incluimos dos políticas de ejemplo para fines demostrativos.

La interacción con la aplicación, administrador de entidades u otro proceso se realiza por medio de la interfaz del sistema *SAE*.

4.3. Implementación de las clases

Describiremos cada uno de los módulos implementados, propuestos en el capítulo 3, explicaremos su funcionamiento y características.

La implementación del sistema *SAE* agrupa doce clases cada una responsable de una tarea específica pero que cooperan entre sí para lograr el proceso de replicación/actualización de información.

Empezaremos por describir las clases del cliente-servidor *SAE* las cuales son el punto de entrada y salida del sistema, enseguida explicaremos la implementación del repositorio de operaciones, y la tabla de referencias. Después describiremos la implementación de la clase que realiza la interpretación de los mensajes transmitidos entre sistemas *SAE*. Por ultimo mostraremos el funcionamiento del demonio y la función que provee.

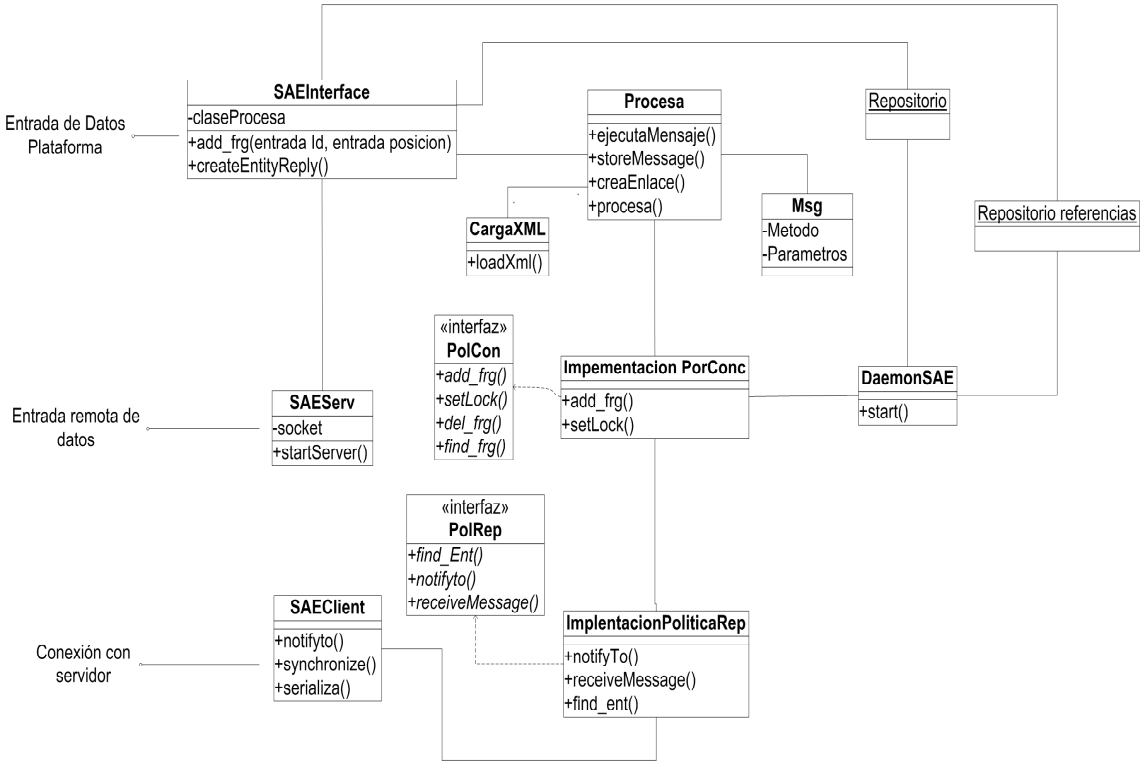

Figura 4.1: Diagrama General de Clases

4.3.1. Implementación cliente/servidor SAE

En el capítulo 3 describimos la arquitectura de diseño que nos permite transmitir y recibir mensajes entre sitios a través de la red. El servidor *SAE* se encarga de recibir los mensajes provenientes de los clientes *SAE*, sin alterar la información que poseen. Las clases: *SAEServ.rb* y *SAEclient.rb* respectivamente implementan los componentes servidor y cliente. Por lo tanto esas clases implementan métodos que establecen canales de comunicación entre los sitios por medio de enlaces de tipo *TCP* y *UDP*.

El servidor posee capacidades multihilo, con lo cual es capaz de atender múltiples clientes al mismo tiempo. Así mismo el cliente también puede establecer enlace con múltiples servidores ya que también establece un hilo por cada enlace. De esta forma el cliente no queda bloqueado esperando a terminar un enlace previo. Este proceso es bidireccional, es decir que cada sitio realiza las funciones de cliente y servidor. El servidor en cada sitio utiliza el puerto 2000 que es el valor predeterminado, aunque puede ser modificado. En un principio, y dado el estado actual de desarrollo del sistema *SAE*, la dirección *IP* de al menos un servidor debe ser conocida. Este servidor puede realizar el servicio que se requiera, posiblemente sólo sea un servidor que provea las ubicaciones de los recursos compartidos, tal y como funciona *Emule* y *bitTorrent* explicados en el capítulo 2 sección.

El algoritmo 4 muestra el funcionamiento de la clase *SAEServ.rb*. Se ubica en cada una de los sitios activos, atendiendo las peticiones de los clientes. Un único servidor puede atender peticiones de múltiples clientes tal y como lo hace un servidor Web.
Cada mensaje recibido es enviado a la interfaz *SAE* para procesarlo, más adelante deta-

llaremos este proceso.

Algoritmo 4 Servidor SAE

```
1: iniciaServidor
2: mientras true comienza
3:    abreSocket
4:    si recibe_conexion entonces
5:       cliente = servidor.accept
6:       input = socket.gets
7: end_while
8: interface.recibeMensaje( message )
9: si input=='QUIT' entonces
10:   @sockets.delete(socket)
11:   socket.close
12: End
```

Del lado del cliente Ruby nos permite de manera sencilla abrir canales de comunicación utilizando el método *TCPSocket*, que nos permite realizar la conexión con el servidor. Una vez se ha conectado con el servidor, el cliente recibe una notificación indicando el enlace exitoso (ver código 5).

Código 5 Código cliente

```
s = TCPSocket.open( host, port )       #apertura del canal
local, peer = s.addr, s.peeraddr       #datos del destino
STDOUT.print "Connected to #{peer[2]} :#{peer[1]}"
STDOUT.puts " usando puerto local #{local[1]}"
```

Después de transmitir los datos el servidor responde con un acuse de recibo ACK, el cliente por su parte envía la instrucción $QUIT$, con lo cual se cierra la conexión (ver código 6).

El esquema de la interacción cliente/servidor se muestra en la figura 4.2. El cliente del sitio A interactua con el servidor del sitio B y viceversa, los mensajes entre sitios se representan por las líneas rojas y los mensajes de respuesta en color azul. El cliente envía los datos y el servidor contesta con un ACK, el cliente contesta con un mensaje $QUIT$ lo cual finaliza la interacción.

Los mensajes, que son enviados desde los clientes hacia los servidores, contienen detalles sobre el sitio de origen, los datos del destinatario e información relacionada a las políticas de replicación, las cuales son las únicas que pueden darle una interpretación. El cliente y el servidor son totalmente ajenos a esta información.

Código 6 Código cliente

```
loop do
 case
   when local[0,8] == 'Accepted' #SE ACEPTA LA CONEXION
   s.puts( message )             #EL MENSAJE ES ENVIADO POR EL CANAL
   s.flush
   when local[0,3] == 'ACK'      #CUANDO RECIBE EL ACUSE....
   s.puts( "quit" )              #TRANSMITE UNA SENAL AL SERVIDOR
   s.flush
 end
end
```

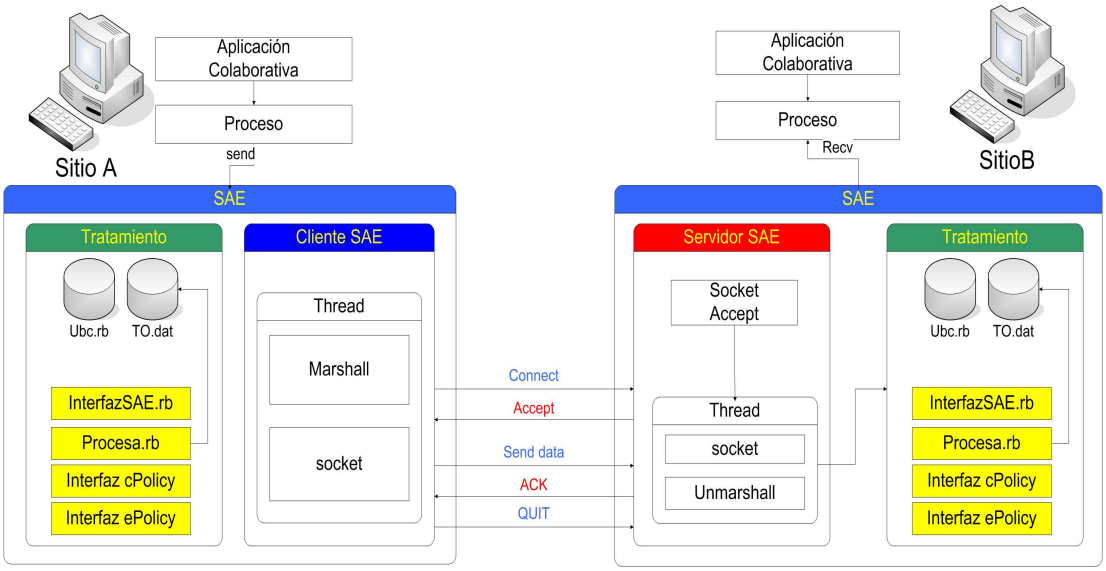

Figura 4.2: Interacción entre un Cliente y un Servidor *SAE*.

Establecimiento de un enlace

Si requerimos realizar una conexión con un servidor *SAE*, utilizamos el método: `notifyTo(ip, port, message)` donde:

- **ip.** Es la dirección del servidor *SAE*.
- **port.** Es el puerto del servidor *SAE*.
- **message.** El mensaje que interpreta el servidor *SAE*.

Por ejemplo: `SAEClient.notifyTo( 192.168.1.73, 2000, data )` envía un objeto *data* a la dirección **IP** 192.168.1.73:2000.
Si el enlace no puede ser completado genera una excepción, en la cual se actualiza el estatus del sitio apuntado por 192.168.1.73 a *Ausente* en la tabla de distribución.
Se manejan tres estados:

1. **activo.** Sitio en actividad normal.
2. **ausente.** Sitio no responde, se realizaran reintentos.
3. **desconectado.** Sitio inalcanzable, no habrá reintentos.

Estos estados indican la actividad de los peers, con lo cual evitamos conexiones con sitios inalcanzables.

En caso de necesitar sincronizar el reloj del sistema con el tiempo global *UTC*, implementamos un método que utiliza *NTP Network Time Protocol*. Este protocolo funciona para sincronizar sistemas en la red, utiliza *UDP* como capa de transporte y la sincronización es por conexión a un servidor *NTP* (*e.g. ntp2.belbone.be*). Este servicio es útil para realizar marcas de tiempo exactas y poder utilizarlas como factor relevante en la resolución de conflictos.
La forma de uso es `SAEClient.synchronizeTime()`, en *UNIX* se debe contar con privilegios de root, (ver código 7).

Código 7 Sincronización de tiempo

```
def synchronizeTime()
  data = UDPSocket.open do |sd|  #EL ENLACE ES DE TIPO UDP
    sd.send(SNTP_MSG, 0, "ntp.hiway.com.br", SNTP_PORT)  #SERVIDOR NTP
    sd.recvfrom(64)[0]
  end
end
```

Para establecer enlaces de tipo *UDP* se utiliza *UDPSocket*, pero así mismo necesita un servidor de tipo *UDP*, el cual esta disponible en el puerto 2001 en el mismo host. La invocación es idéntica al cliente *TCP*, `sendData(ip, puerto, data)`.

4.3.2. Interfaz del sistema SAE

La clase *SAEInterfaz.rb* tiene por objetivo recibir la información proveniente de la plataforma de administración de entidades, o de todo proceso interesado en replicar o actualizar su información.

Los métodos que ofrece para interactuar con el sistema *SAE* son:

- `createEntityReply(idEntity, user, cPolicy, ePolicy, tablaDistribucion)`
- `notify`(from, to, message).
- `receiveMessage( message, source )`
- `add_frg( idEntity, user, position )`

El método `createEntityReply( idEntity, user, cPolicy, ePolicy )` crea la asociación $(user/Entity) = (PolicieC, PolicieE)$ entre la entidad y las políticas de replicación/actualización.

Explicación de los parámetros:

- **idEntity.** Es el objetivo de la replicación, aunque solo es representado en el sistema por medio de un identificador (e.g. ID2020), este identificador lo provee el proceso que solicitó el servicio. Este sistema no almacena información respecto a la entidad que se replica/actualiza, únicamente provee el servicio al proceso que lo solicita. Así el sistema *SAE* es ajeno a la información de esas entidades, por lo que pueden ser administradas de la forma que le convenga a sus desarrolladores.
- **User.** Este valor sólo es utilizado para crear la asociación. A menos que se implemente alguna política que necesite este valor.
- **cPoliciy.** Se refiere al identificador de la política de concurrencia que se aplica. $cPolicy = nil$ indica que no se requiere control de concurrencia.
- **ePoliciy.** Es el identificador de la política de replicación/actualización. Estos identificadores son utilizados para localizar la clase de política correspondiente en el archivo *XML políticas.xml*, este parámetro no puede ser nulo.
- **tablaDistribucion.** Indica las ubicaciones de los sitios que comparten una copia de las replicas.

Un usuario puede tener vinculadas N entidades asociadas con $P * Q$ políticas donde N es el número de entidades, P es el número de políticas de concurrencia y Q es el número de políticas de replicación/actualización. Esto puede resultar útil en caso de que un usuario necesite actualizar la información de más de una replica a la vez. Cada política es identificada por medio de un *ID* que se configura en el archivo *políticas.xml*, el cual

describimos en el capítulo 3 sección 4.

El método `notify(from, to, message)`. Se utiliza para hacer llegar un mensaje a un sitio remoto que alberga un servidor *SAE* (ver código 8) .

Explicación de los parámetros:

- *from.* Es el origen del mensaje, está formado por el par *entidad/usuario*, que es el identificador de esta entidad en todo el sistema. El uso de este parámetro depende de la política de replicación/actualización implementada, por lo tanto puede tener múltiples usos.
- *to.* Es la replica destino del mensaje. Este valor es utilizado por la política de replicación/actualización, para encontrar el objetivo de la notificación en la tabla de distribución.
- *message.* Los datos enviados entre una replica y otra, la interpretación de los datos es responsabilidad de la implementación de la política de replicación correspondiente. Por lo tanto este mensaje puede ser desde una cadena de texto, un archivo XML, o un objeto.

Código 8 Método notify

```
#DEFINICION DEL METODO
def notify( from, to, message )
  #SE CREA LA PLATILLA DEL MENSAJE
  xml = @claseProcesa.createXML( 'notify', from, to, message )
  #SE OBTIENE LA CLASE QUE SE VA A EJECUTAR
  clase = @claseProcesa.getObject( from )
  clase.notify( to, xml )       #EL METODO SE EJECUTA
end
```

El método `receiveMessage(message, source)` es responsable de recibir los mensajes provenientes de sitios remotos. Todos los mensajes que se hacen llegar al *SAE* vía la red son recibidos aquí, aunque el tratamiento no se realiza en esta clase.

Explicación de los parámetros:

- *message.* el mensaje recibido, que como vimos puede ser de muy distintas representaciones.
- *source.* Es la dirección *IP* y el puerto del origen del mensaje cuyo valor es relevante para la política de replicación/actualización que lo requiera.

El método `add_frg(idEntity, user, position)`. Establece un área que será objeto de actualizaciones y notificaciones.
Explicación de los parámetros:

- *idEntity.* La entidad que solicita la región. Dicha región está representada en la política de concurrencia por medio de valores numéricos. Esta región puede protegerse por medio de alguna restricción indicada en la política de concurrencia.
- *user.* El usuario propietario de la entidad. Este valor solo tiene utilidad para obtener la política de replicación/actualización asociada a la entidad.
- *position.* Las coordenadas de la región. Su formato corresponde enteramente a la implementación de las políticas de control de concurrencia. Por lo tanto puede representar valores de coordenadas, índices, renglones, columnas.. etc.

4.3.3. Carga dinámica de clases

El punto fuerte de esta clase es la carga dinámica, lo que nos da la capacidad de crear código flexible y adaptable.
El método `SAEInterface.creaEntityReply()` utiliza esta característica para la creación de la asociación entre replicas y políticas de concurrencia, actualización/notificación.
La carga dinámica de clases nos permite realizar instancias de clases diferentes a tiempo de ejecución sin realizar ninguna edición de código o compilación. Esto es particularmente útil si necesitamos usar distintas clases de políticas de replicación al momento que se requieren, sin saber *a priori* su nombre, ubicación o tipo de datos.
Con ello logramos que el sistema *SAE* pueda expandir sus capacidades incluyendo políticas de replicación/actualización sin tener que editar el código de la interfaz o cualquier otro código mas que el de la propia clase de políticas de replicación/actualización.

El algoritmo 5 muestra el procedimiento para la carga dinámica de las clases y el código 9 muestra la implementación.

Algoritmo 5 Pseudocodigo carga dinámica de clases

```
1: si politicaC y politicaE != null entonces
2:    valorPolC = obtenValorRepositorio(politicaC) **clase obtenida del repositorio
3:    valorPolE = obtenValorRepositorio(politicaE) **clase obtenida del repositorio
4: clase1 = cargaDinamica('valorPolC'.inicializa(tabla, usuario/entidad))
5: clase2 = cargaDinamica('valorPolE'.inicializa(usuario/entidad, clase1))
```

Procedimiento para incluir una nueva política

Supongamos que se requiere incluir una nueva política de replicación/actualización llamada `VirtualSync.rb`. El primer paso es editar el archivo *políticas.xml* (código 10) para incluir los datos de esta política:

Se deben ingresar en los atributos de ubicación la ruta del archivo de origen, (e.g. *../policies/VirtualSync.rb*), el nombre del archivo y si se requiere utilizar los servicios de un proceso demonio.

Código 9 Carga de políticas

```
#OBTENEMOS LA INFORMACION DE LA POLITICA DEL ARCHIVO XML
if @tablePoliticas.getValue("#{cPolicy}") != nil
   archivoPoliticaC = @tablePoliticas.getValue("#{cPolicy}")[:file ]
   archivoPoliticaE = @tablePoliticas.getValue("#{ePolicy}")[:file ]
else raise Exception
end

#OBTENEMOS LAS LIBRERIAS NECESARIAS PARA LA CARGA DE LA CLASE
#TAMBIEN DEL DEL ARCHIVO XML
baseDirC = "#{dirUNX}#{archivoPoliticaC}"
baseDirE = "#{dirUNX}#{archivoPoliticaE}"
#INCLUIMOS ESTAS LIBRERIAS
require baseDirC
require baseDirE

#EL METODO EVAL INTERPRETA CADENAS COMO CODIGO. REALIZAMOS LA CARGA
clase1=eval("#{archivoPoliticaE}.new(tabla,'user/idEntity')")
clase2=eval("#{archivoPoliticaC}.new('user/idEntity',type',clase1)")
#CREAMOS EL ENLACE
@claseProcesa.creaEnlace("user/idEntity",clase2,"baseDirC*baseDirE")
```

Código 10 Archivo de configuración de políticas

```
<policies>
  <policy name="PRIMARY" timestamp="false" id="P1" type="OPT">
     <source basedirUNX="../Policies/"  file="Policy1"/>
     <misc logs="true" demon="true" >
     </misc>
     <demon initS="false" hearthbeat="true" time="50" priority="true">
     </demon>
  </policy>
  <policy name="BACKUP" timestamp="true" id="C1" type="OPT">
     <source basedirUNX="../Policies/" file="ConFragPolicy"/>
     <demon initS="false" hearthbeat="true" time="5">
     </demon>
  </policy>
</policies>
```

```
<policy name="Virtual" timestamp="true" id="V1" type="OPT">
     <source basedirUNX="../Policies/" file="VirtualSync"/>
     <demon initS="false" hearthbeat="false" time="5">
     </demon>
  </policy>
```

Una vez completada la edición, quedará disponible la clase para ser utilizada por cualquier proceso. Si no existe el archivo, la ruta es incorrecta o existen errores de sintaxis, se genera una excepción.

4.3.4. Implementación del procesamiento de mensajes

Esta clase (ver figura 4.3) se encarga de crear los mensajes que serán enviados a los diversos servidores *SAE* en una sesión de cooperación. También es responsable de recibir y comenzar el tratamiento de la información contenida en ellos.

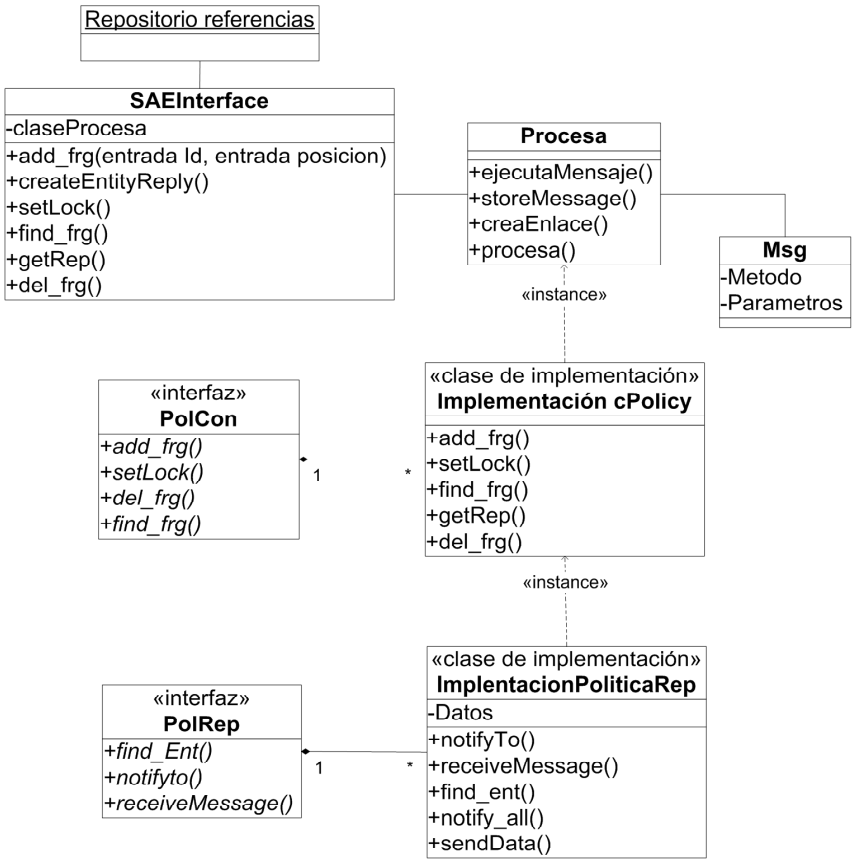

Figura 4.3: Interpretación de los Mensajes.

Esta clase posee numerosos métodos para la creación y lectura de archivos *XML*, los cuales ayudan a navegar por el árbol generado por esta estructura de datos.

Solo citaremos los métodos más importantes:

- `storeMessage(message)`.
- `execMessage(message, source)`.
- `creaEnlace(llave, clase, librerias)`.
- `procesa(input, source)`.

El método `storeMessage(xml)`. Registra un mensaje en el repositorio de operaciones, la implementación de este repositorio se explica en la siguiente sección.
Explicación de los parámetros:

- *xml.* Contiene los detalles sobre la operación que debe ejecutarse en la política de concurrencia, replicación/notificación correspondiente. Este valor puede ser accedido por cualquier proceso que tenga una instancia del repositorio, como por ejemplo el proceso demonio. El documento de tipo *XML* contiene valores, relevantes de interés para la ejecución de algún método, como pueden ser algunos valores de parámetros, la entidad que originó la operación, el tipo de mensaje etc.

El método `execMessage(llave)` ejecuta el primer mensaje apuntado por *llave* en el repositorio de operaciones.

Por medio de la llave se obtiene un arreglo de tipo *FIFO*, con operaciones asociadas a esa llave. Estas operaciones están almacenadas en una estructura de datos de tipo *XML*. De este arreglo obtenemos el primer elemento de la cola, el cual es eliminado una vez de ha completado la operación.

El método `creaEnlace(llave, clase, librerías)`, crea la asociación entre la entidad y las políticas de replicación/notificación.
Explicación de los parámetros:

- *llave.* Este parámetro es la clave con la que se registra el enlace.
- *clase.* El objeto a ser registrado.
- *librerías.* Este parámetro incluye todas aquellas dependencias necesarias para realizar una recuperación correcta de la clase. De lo contrario, al momento de recuperar la clase se genera una excepción.

El registro utiliza un medio de almacenamiento persistente (archivo), para guardar objetos. *Pstore* nos permite almacenar objetos en archivos para ser recuperados en algún otro momento simplemente abriendo el archivo de nuevo ya que el objeto recuperado es un mapa *Hash* (ver código 11). Con esta herramienta podemos proveer tolerancia a fallas en caso de una anomalía con el servidor o corte de energía.

El método `procesa(xml)`, comienza el proceso de interpretación del mensaje recibido. El primer paso es obtener el tipo de mensaje, ya que esto depende el flujo de eventos a seguir. El *XML* contiene esta información en el atributo *type*. Por ejemplo si un mensaje indica que debe ejecutar su operación inmediatamente el tipo de mensaje será *REP*.

Código 11 Registro de mensajes

```
#Formato del par\'ametro librer\'ia
libreria = librerias.split("*")  #LAS LIBRERIAS
_array = Array.new
for i in 0 .. libreria.length-1 do
 _array << libreria[i]   #Colocamos las librerias en un arreglo
end
_array << "#{llave}/clase" #llave para recuperar la clase
@storeUbc.transaction do
 @storeUbc["#{llave}"] = _array #Registro librerias
end
@storeObj.transaction do
 @storeObj["#{llave}/clase"] = clase #EL ENLACE ES CREADO
end
```

4.3.5. Repositorio de operaciones

La implementación de este repositorio consiste en el registro de objetos a un archivo usando la librería *Pstore.rb.*
Utilizamos esta idea con el fin de prestar un servicio auxiliar a cualquier política que requiera almacenar, de manera temporal, las operaciones provenientes de algún otro sitio. La idea, detrás en la implementación de este repositorio, es dar soporte al trabajo nómada que, como pudimos observar en el capítulo 3, requiere de mecanismos asíncronos para la entrega de operaciones entre políticas de replicación/notificación.
Su uso no es imperativo, incluso la implementación posterior de otras políticas de replicación/notificación pueden incorporar su propio sistema de soporte al trabajo nómada o sistema de resolución de conflictos basado en una bitácora de operaciones [Edwards, 1997].

Una variación de esta idea del repositorio nos permite llevar un registro de los objetos que se utilizan por medio de las referencias almacenadas en un archivo.
Todas las operaciones de escrituras y lecturas a este archivo se realizan a través de transacciones, es decir, cada operación de lectura y escritura es completada y almacenada al momento, evitando así estados intermedios. Por lo tanto la información se mantiene en un estado consistente si es implementado además un estricto control de concurrencia para el acceso a este archivo.
Una manera de lograr este control es por medio de semáforos, con lo cual creamos un orden en los accesos a este archivo (ver figura 12).

4.3.6. Implementación de la clase de políticas

Para llevar a cabo el proceso de replicación de información es necesario definir e implementar las políticas de control de concurrencia y políticas de replicación/actualización. El diseño de estas políticas corresponde enteramente a un análisis en el área, ya que es extenso y existen muchas propuestas para ello como lo mostramos en el capítulo 2.

Código 12 Bloqueo de sección crítica

```
#ESTABLECEMOS UNA RESTRICCION
   mutex.synchronize {
      #REALIZAMOS LA TRANSACCION
      @store.transaction do
          band = @store['entrada']   #obtenemos el objeto
          @store['entrada'] = band   #escribimos el valor
      end
   } #finaliza restricci\'on de acceso
```

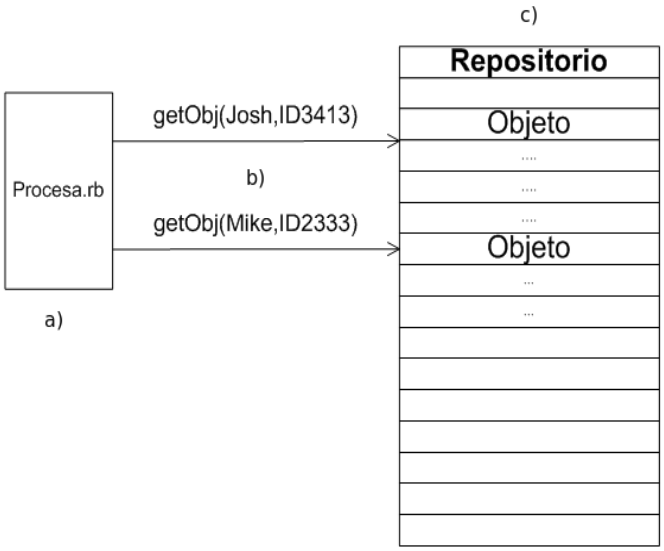

Figura 4.4: Repositorio, Recuperación de Objetos.

Por lo tanto describiremos el mecanismo del sistema *SAE* para incorporar políticas distintas y mostramos una política de ejemplo para demostración.
Cualquier política que se desee incorporar debe tener una correspondencia de métodos con la interfaz, es decir que debe implementar los métodos definidos en la interfaz *SAE*. Cuando la aplicación, un usuario o algún proceso inicie la creación de una nueva entidad, debe indicar el identificador de la política que registramos con el fin de poder ser utilizada como lo vimos en la sección 3.2 de este capítulo.

En ese momento la instancia queda registrada en el repositorio (figura 4.4) junto con el valor de sus variables las cuales quedan disponibles cuando se necesitan. La explicación de la figura: a) La clase Procesa.rb recupera la instancia de la clase almacenada b) la llave esta formada por el usuario y la entidad. c) Todos los objetos almacenados están vinculados a una entidad y un usuario

Cuando se requiere una invocación a alguno de los métodos de la clase, por ejemplo agregar un fragmento (i.e. región de escritura), se utiliza la interfaz *SAE*:

- `SAEInterfaz.add_frg(idEntity, user, position).`

La interfaz delega el método a la clase *Procesa.rb*, quien obtiene el objeto del repositorio e invoca el método correspondiente a la política de control de concurrencia. El algoritmo 6 muestra el procedimiento para una invocación.

Algoritmo 6 Pseudocódigo invocación dinámica de métodos

```
1: begin
2: metodo = obtenerValorMetodo( xml )
3: arregloParametros = obtenerParametros( xml )
4: cadenaMetodo = metodo+arregloParametros
5: clase = obtenClaseRepositorio(getUser(xml), getIdEnt(xml))
6: evalua( clase.cadenaMetodo )
7: end
```

Cuando finaliza la invocación, la clase de política de control de concurrencia informa a la política de actualización/replicación el cambio en el valor. Por su parte la política de actualización/replicación, dependiendo de su implementación, puede difundir la información de inmediato a todos los sitios registrados en su lista de distribución de manera síncrona o difundir en un tiempo posterior de manera asíncrona.
Además los procesos internos realizados junto con esta difusión depende por completo de la política diseñada. Este proceso se completa cuando la política de actualización utiliza, si así lo requiere, el cliente *SAE* para difundir sus datos a otros sistemas *SAE*.

El cambio de una política por otra es realizado cuando asociamos nuevamente una entidad con una política, como lo explicamos en la sección 3.3 de este capítulo. Los datos de la política anterior se traspasan a la nueva política al momento de crear la instancia. Esto es sencillo de realizar porque esos datos están empaquetados en un archivo *XML*, de esta forma podemos enviar como argumento los datos a la nueva política con lo cual también obtenemos flexibilidad de implementación de mecanismos para interpretar esas variables.

4.3.7. Creación e interpretación de los mensajes

Los mensajes, notificaciones y actualizaciones provenientes de otros sistemas *SAE* realizan exactamente el mismo proceso a nivel de políticas de control de concurrencia y replicación/notificación. El cliente/servidor *SAE* se encarga de la transmisión, empaquetado y desempaquetado, las políticas son ajenas a este proceso.

Creación del mensaje

El proceso inicia al momento de recibir una invocación de un método en la interfaz *SAE* destinado a alguna política replicación. Se recupera el objeto en el repositorio por medio del identificador de la entidad y el usuario, se crea una platilla vacía del mensaje:

```
xml = @claseProcesa.createXML('notify', from, to, message)
```

Enseguida se invoca al método correspondiente y comienza a incluir datos clave para la interpretación de este mensaje en otro servidor *SAE*, (i.e. el nombre del método, parámetros, tipo de mensaje), (ver código 13).

Código 13 Datos del mensaje

```
<Msg>
   <Metodo name="receiveMessage"/>
   <Parametros npar=2/>
     <Par name='from' type='STRING'>
       value
     <\Par>
     <Par name='message' type='STRING'>
       REP
     <\Par>
   </Msg>
```

Tras haber pasado por las políticas el mensaje es "serializado"(marshalling) para poder ser enviado a través de la red. El algoritmo 7 describe los eventos en la creación de este mensaje.

Algoritmo 7 Creacion del mensaje

1: begin
2: invocacion a metodo 'notify'
3: xml = creaPlantilla('notify', Parametros) **plantilla vacía
4: clase = obtenerObjeto(usuario, entidad) **obtiene el objeto
5: clase.notify(to, xml) **envía el mensaje a la clase
6: end

Una vez que el servidor *SAE* lo recibe comienza el proceso inverso, primero al mensaje se le aplica un (unmarshalling) para recuperar el mensaje. Enseguida se identifica el tipo de mensaje y dependiendo del tipo de mensaje se atiende de inmediato o se deja en una cola de espera. Cualquiera que sea el caso el paso siguiente es interpretar el árbol *XML* (mensaje). Después se recupera del mensaje el identificador de la entidad a la que va dirigida y el usuario; con todo esto se forma la llave para obtener el objeto de la política correspondiente, enseguida se recupera del mensaje el método y los parámetros que se quiere ejecutar, para realizar la invocación. En los algoritmos 8 y 9 se muestra el proceso de interpretación, al momento de llegar al servidor y posteriormente al ser interpretado por la política, la implementación se muestra en el código 14.

Este proceso es idéntico, no importa que política se implemente, siempre que se implementen los métodos de la interfaz *SAE*.

Algoritmo 8 Recepció del mensaje en el servidor

```
1: begin
2: Servidor_recibe dato ** El servidor recibe el dato del cliente
3: xml = Unmarshalling(dato) ** realiza en desempaquetado
4: Interface.recibeMensaje( xml ) ** el objeto es enviado a la interfaz
5: end
```

Algoritmo 9 Interpretación del mensaje en política

```
1: begin ** este codigo pertenece a la clase procesa.rb
2: metodo = obtenMetodo( xml ) **el objeto xml es interpretado
3: parametros = obtenParametros( xml )
4: clase = obtenerClaseRep( otenerUsuario(xml), obtenerIDEnt(xml))
5: evalua( case.metodo+parametros ) ** ejecuta la instrucción
6: end
```

4.4. Caso de muestra

Con el fin de probar el modelo se implementaron dos políticas. Una política de actualización/notificación de tipo *PULL* optimista que utiliza un histórico de operaciones como mecanismo para resolución de conflictos [Theimer et al., 1995]. Además una política de control de concurrencia, que establece bloqueos para la escritura en regiones especificadas. Explicaremos los métodos mas importantes de esta clase:

El método `add_frg(dirA, position)`, establece una nueva región protegida dada por la posición. Se verifica que la región no esté habilitada en otro sitio.
Explicación de los parámetros.

- *Id*. Identifica al fragmento para esta entidad en todo el sistema, este valor es suministrado por el proceso que lo requiere.
- *pInit*. La posición de inicio, que puede ser desde una coordenada, fecha, día, o cualquier valor relevante para el proceso que lo requiere.
- *pFin*. La posición final, este valor junto con *pInit* forman la región protegida. Se difunde a todas las replicas el establecimiento de la restricción.

El método `split_frg(id, pos)`, divide un fragmento (región) dado por la posición *pos*. Método `setLock(id)`. Establece una protección para la región indicada por ID. Se informa a todas las replicas el establecimiento de la restricción. Estas, a su vez, agregan el candado.

Método `getFrg(id)`. Obtiene el fragmento identificado por *Id*, si no lo encuentra de manera local, busca en los sitios contenidos en la lista de distribución. De esta manera la aplicación o proceso obtiene las regiones protegidas definidas en otras replicas.

Código 14 Intérprete de mensajes

```
parametros = ""
#OBTENEMOS LOS PARAMETROS DEL ARCHIVO XML
#DAMOS FORMATO A LA INVOCACION
for i in 1 .. _secondArray.length - 1 do
parametros << "\"#" << "{_secondArray" << "[" << i.to_s << "]}\""
separador = (i==_secondArray.length-1 ? "" : ",")
parametros << separador
end
#OBTENEMOS EL NOMBRE DEL  METODO
_metodo  = _secondArray[0]

#OBTENEMOS LA ASOCIACION ENTIDAD/POLITICA, I.E. LAS CLASES DE pol\'iticas
store0.transaction do
_array = store0[ "#{@_user}/#{@_entity}" ]
@_clase = _array[1]
end
#SI LA CLASE EXISTE
if @_clase != nil
eval("@_clase.#{_metodo}(#{parametros})") #EJECUTAMOS
else raise Exception
end
```

Estructura de los fragmentos

Para llevar un control sobre las regiones definidas utilizamos un archivo *XML* cuya estructura se muestra en el código 15.

Esta estructura corresponde también a la política de control de concurrencia implementada, otras políticas pudieran establecer su propio sistema de registro.

4.4.1. Política de actualización/notificación.

Esta política envía las actualizaciones de manera periódica a los sitios contenidos en su tabla de distribución. Implementamos una política de tipo *pull*, que utiliza un proceso demonio para enviar las notificaciones o actualizaciones cada cierto tiempo, este proceso demonio es suministrado por el *SAE*. Más adelante en esta sección describiremos la implementación del demonio.
Del lado del servidor existe otro proceso demonio que atiende las peticiones de manera asíncrona y cumple la tarea de proveer soporte al trabajo nómada o desconectado.
Se muestran a continuación los métodos más relevantes de esta clase.

Método `notifyTo(to, message)`. Envía un mensaje a una replica utilizando el cliente

Código 15 Estructura de datos compartidos

```
<Coord>
  <Fragmento id="", posI=0, posF=0, lock=false>
    valor
  <\Fragmento>
  <Fragmento id="", posI=0, posF=0 lock=true>
    valor
  <\Fragmento>
<\Coord>
```

SAE, el significado de los parámetros es el siguiente:

- *to*. Identificador de la replica destino del mensaje, está formado por el par entidad/usuario.
- *Message*. Representación de los datos que se va a transmitir, su formato depende de la implementación de la política de replicación/notificación.

Método `notify_all(message)`. Notifica a todas las replicas identificadas en la tabla de distribución.
Método `sendData(data, target)` Envía un objeto a alguna replica.
Descripción de parámetros:

- *data*. Es el objeto que se quiere transmitir, puede ser un archivo o un objeto Ruby. Sin embargo hay restricciones en el caso de objetos, algunos no pueden ser serializados para su transmisión.
- *Target*. La dirección del sitio *SAE* a la que sera enviado el objeto.
- *Port*. El puerto destino del servidor *SAE* que recibe este objeto.

Método `receiveMessage(mensaje, source)`. Recibe un mensaje proveniente de otra replica. Interpreta varios tipos de mensaje:

- ACT (actualización)
- *ADD* (agrega sitio)
- *HRB*(actualización de estado)
- *FND*(Busca réplica)
- *EXC*(ejecuta método)

Explicación de los parámetros.

- *mensaje*. El mensaje recibido por el servidor *SAE*, se interpreta de acuerdo al tipo de mensaje; contiene detalles sobre su origen, tipo, destino, operación solicitada y valores relevantes.

Método `send_heartbeat`. Envía una señal de vida a todas las réplicas indicando su estado, al ser recibido se actualiza la lista de distribución.

Método `setTablaDist(tablaDist)`. Establece la tabla de distribución, este dato lo puede proporcionar la aplicación, el administrador de entidades, el usuario, o algún otro proceso.

4.4.2. Implementación del proceso demonio

Esta clase tiene por objetivo realizar los eventos de actualización/replicación de la política implementada, los detalles y procedimientos se explicaron en el capítulo 3.

Funcionamiento

Lado Servidor.

Los mensajes provenientes del cliente, que no han sido procesados se almacenan en un registro o repositorio de operaciones. Este repositorio es una estructura de datos que registra objetos en archivos, funciona bajo la política *FIFO*. El demonio accede a este repositorio y obtiene la primera operación registrada en la cola, interpreta el mensaje y ejecuta la operación. El algoritmo 11 muestra el funcionamiento del proceso demonio.

Algoritmo 10 Eventos seguidos por el demonio (servidor)

```
1: begin
2: mientras continua=true comienza
3:    xml = obtenOperRepositorio( ) ** obtiene el primer registro del repositorio
4:    metodo = obtenMetodo( xml ) **el objeto xml es interpretado
5:    parametros = obtenParametros( xml )
6:    clase = obtenClaseRepositorio( otenerUsuario(xml), obtenerIDEnt(xml))
7:    evalua( case.metodo+parametros ) ** ejecuta la instrucción
8: end
```

Lado Cliente.

Las operaciones que deben ser atendidas de manera automática se cargan al momento de iniciar la asociación entidad-política. Esta información se configura en el archivo *políticas.xml* y la clase *Procesa.rb* carga esta información en el repositorio de referencias.

Estas operaciones no caducan, es decir permanecen siempre registradas, por eso el demonio del lado del cliente puede ejecutarlas una y otra vez.

El demonio del lado del cliente es idéntico al del lado servidor, pero al inicio se proporciona el repositorio asignado que va a acceder, de esta forma fomentamos su reutilización.

El procedimiento para detener el proceso daemon es agregar una operación en el repositorio de referencias de tipo "*QUIT* ". El demonio interpreta este código y finaliza el ciclo

Algoritmo 11 Eventos seguidos por el demonio (cliente)

```
1: begin
2: mientras continua=true comienza
3:    xml = obtenOperXMLPol( cPolicy ) **obtiene las operaciones de la polítca
4:    metodo = obtenMetodo( xml ) **el objeto xml es interpretado
5:    parametros = obtenParametros( xml )
6:    clase = obtenClaseRepositorio( otenerUsuario(xml), obtenerIDEnt(xml))
7:    si metodo != 'QUIT' entonces
8:       evalua( case.metodo+parametros ) ** ejecuta la instrucción
9:    otro
10:      continua = false
11: end
```

4.5. Pruebas y resultados

En las pruebas se ha verificado el cumplimiento de las características propuestas en el diseño del sistema, es decir, que todos los módulos provean la funcionalidad para la que fueron diseñados. La integración de todos los módulos también forma parte del conjunto de pruebas así como pruebas de actualización de entidades, repositorio, y tolerancia a fallos en desconexiones.

4.5.1. Condiciones de la prueba

Las condiciones previas de la prueba permiten establecer un escenario en el cual se pueda describir con detalle todo lo acontecido, así como también registrar los resultados obtenidos. Las pruebas se realizaron en 2 equipos, cuyas características se muestran a continuación:

Equipo 1

- Equipo PC escritorio
- Procesador Intel/D 1.8Ghz
- 1 GB RAM
- plataforma linux-x86, distribución Fedora 7
- conexión de red inalámbrica.

Equipo 2

- Equipo Portátil MacBook Pro
- Procesador Intel Core 2 Duo 2.2 Ghz
- 2GB RAM
- versión del sistema Mac OS X 10.5.8 (9L31a)

Figura 4.5: Servidor *SAE*. Recepción de Mensaje

- versión del kernel Darwin 9.8.0
- conexión de red inalámbrica.

Prueba 1.

- **Objetivo.** Verificar el funcionamiento del servidor *SAE* en un envío y recepción de mensajes dirigidos a una entidad. Cada equipo utiliza un esquema *P2P*, por lo que en cada uno existe un proceso servidor y un proceso cliente.
  El $Equipo_1$ envía una notificación a la entidad A del sitio remoto $Equipo_2$ la cual es una actualización de estado, por lo que el $Equipo_2$ deberá recibir la notificación, encontrar la entidad A_1 correspondiente y actualizar el estado en la lista de distribución del $Equipo_1$.
  Todo este proceso incluye el funcionamiento correcto del servidor *SAE*, la tabla de objetos en memoria, repositorio de operaciones, la interfaz de entrada.
- **Resultados**. La figura 4.5 muestra los resultados obtenidos en el servidor *SAE*. El servidor recibió una operación de actualización de estado (*Heartbeat*) del cliente destinado a la entidad *Jake/ID345*, el archivo XML es creado antes de la invocación del cliente. En el servidor este mensaje se interpretó y se realizó la ejecución. El mensaje utilizado se muestra en el código 16:

Prueba 2.

- **Objetivo.** Comprobar el soporte a la desconexión en una sesión de trabajo. Se simulará una desconexión voluntaria la cual deberá disparar un evento para almacenar las operaciones no atendidas en un archivo. Posteriormente se volverá a reconectar los servidores, cargar de nuevo las operaciones no atendidas y reiniciar el proceso demonio que atenderá nuevamente las operaciones previamente registradas.

Código 16 Mensaje enviado

```
<msg reference='FOOBAR-1234' atach='false' tipo='REP'>
  <Metodo npar='1'>
    <name>recv_heartbeat</name>  #METODO A EJECUTAR
  </Metodo>
  <Info>
    <userO>Mike</userO>
    <userD>Jake</userD>         #USUARIO DESTINO
    <entityO>ID222</entityO>
    <entityD>ID345</entityD>     #ENTIDAD DESTINO
    <msg>DEM</msg>
    <ipO>127.0.0.1</ipO>
    <portO>2010</portO>
  </Info>
  <Parametros>
    <Parametro tipo='20'>        #PARAMETRO TIPO TEXTO
      <PTxt>Mike/ID354</PTxt>    #VALOR DEL PARAMETRO
      <PNum>30</PNum>
    </Parametro>
  </Parametros>
</msg>
```

- **Resultados**. El sitio A es desconectado, las operaciones pendientes en el repositorio se suspenden y las que llegan al sistema se almacenan como van llegando en el repositorio, (ver figura 4.6). Al finalizar esta parte de la prueba prosigue el evento contrario que es volver iniciar el proceso demonio que atiente todas estas operaciones almacenadas. En la figura 4.7 podemos apreciar la consola de salida donde se van imprimiendo los sucesos como van ocurriendo.

Estos procedimientos son el corazón de todas las operaciones dentro del *SAE*. Esto es así porque todas las operaciones que salen y entran al sistema pasan por esos procedimientos: la interfaz, la clase de interpretación de los mensajes, el demonio, el repositorio de operaciones, el cliente y servidor. Las políticas de replicación/actualización son responsabilidad del programador que necesite de su funcionamiento. Las que presentamos aquí son para fines demostrativos del modelo. Con el fin de presentar un análisis y comparaciones con otros modelos necesitamos crear políticas mas robustas y adaptar varias aplicaciones colaborativas representativas para usar este sistema. En el próximo capítulo presentamos las conclusiones sobre este trabajo axial como algunas mejoras que podemos realizar al modelo.

Figura 4.6: Almacenado de Operaciones.

Figura 4.7: Ejecución del Proceso Demonio, Recuperado de Operaciones.

Capítulo 5

Conclusiones y Trabajo a Futuro

El objetivo de este trabajo de tesis ha sido el desarrollo de un sistema de actualización de entidades replicadas que trabaja como soporte a la colaboración en aplicaciones *Groupware*. Además, proveer los servicios necesarios para la distribución de información, implementando políticas de actualización, notificación de entidades replicadas.
En resumen, los objetivos logrados son:

- Las políticas funcionan a manera de *drivers* y son diseñadas para ser intercambiables y sustituibles gracias a que implementan una interfaz en común.
- El mecanismo que permite este intercambio está inspirado en los componentes de software, los cuales son módulos que pueden ser reemplazados y/o modificados.
- El mecanismo del repositorio de operaciones nos permite también dar soporte al trabajo nómada. Al mantener un registro de las operaciones, preservamos una "imagen"del estado del sistema que puede ser recuperado en caso de una falla.
- El mecanismo de paso de mensajes, nos permite encapsular datos por medio de un archivo XML, el cual se transmite entre sitios *SAE*. El mecanismo de interpretación de mensajes nos permite realizar ejecuciones a métodos de manera remota, similar a un *RPC*.
- El proceso demonio ejecuta de manera autónoma las operaciones contenidas en el repositorio de operaciones. Esta funcionalidad puede ser aprovechada por cualquier implementación de políticas de replicación/actualización. Gracias al uso de archivos de configuración (XML) podemos ser capaces de cambiar las condiciones de inicio del proceso demonio, de las políticas de replicación/actualización y el servidor SAE sin depender de la edición forzosa de código.

La arquitectura desarrollada, basada en patrones de diseño, es lo suficientemente flexible y extensible como para permitir esas adaptaciones.
Gracias al lenguaje de programación elegido pudimos cumplir con la tarea de proporcionar una plataforma con estas características. La carga dinámica de clases y la meta programación fueron un factor determinante para el cumplimiento de los objetivos propuestos al inicio de este trabajo de tesis.

La estructura de este capítulo se organiza en tres secciones. Primero analizamos los puntos logrados con respecto a la propuesta de solución que planteamos en el capítulo 1. Segundo, exponemos las contribuciones que este trabajo de investigación aporta al área de sistemas colaborativos distribuidos. Tercero, evidenciamos algunas carencias que posee el modelo en su estado actual, mostramos las extensiones que mejorarían el funcionamiento del sistema *SAE* y lo harían más robusto. También proponemos un análisis utilizando este modelo en varias aplicaciones colaborativas distribuidas con lo cual probaremos su desempeño y podremos mejorar el diseño.

5.1. Resumen de la problemática

La problemática de este trabajo de tesis se centra en la creación de un modelo genérico de replicación/actualización de entidades replicadas y en la definición de una plataforma distribuida para el soporte eficiente de aplicaciones cooperativas. El objetivo principal es facilitar la tarea del programador de aplicaciones colaborativas distribuidas, porque le permite concentrarse en la implementación de la aplicación (corazón funcional, interfaz de interacción, principalmente), en lugar de programar un sistema distribuido especifico. Con el fin de mejorar la disponibilidad de recursos, los sistemas distribuidos utilizan la técnica de replicación de información que consiste en mantener copias de la información en varios sitios. Esto resulta en una solución de lo más natural debido a que existen copias (diferentes) de respaldo en caso de fallas.
Sin embargo mantener la consistencia de la información distribuida en todas esas copias no resulta evidente.

Las técnicas de replicación de información varían de una aplicación a otra. Crear la coordinación para actualizar información entre replicas depende en gran medida del tipo de arquitectura de distribución y de las necesidades de la aplicación colaborativa distribuida.

Por ejemplo, el tipo de arquitectura de distribución adoptada, influye fuertemente en la disponibilidad y la eficiencia de acceso a la información compartida:

- **Arquitecturas centralizadas.** Por un lado las aplicaciones que adoptan una arquitectura centralizada utilizan mecanismos de caché para mantener actualizados a sus clientes. Esta técnica resulta sencilla porque la única copia permanece en un servidor. Por lo tanto no hay riesgo de alteraciones en la información, no es necesario aplicar una coordinación para ordenar las modificaciones. Pero los principales puntos de fallas siguen siendo la congestión del servidor y una disponibilidad muy limitada en caso de falla de la red o del servidor.

- **Arquitecturas replicadas.** Aquí no existen los problemas de congestión, pero el problema principal reside en el diseño de mecanismos para coordinar las modificaciones, ya que todos poseen una copia de la información. Esta arquitectura permite ganar en la disponibilidad de recursos pero pierde en eficiencia, porque se requiere una cantidad importante de transmisión de datos entre replicas y algoritmos de sincronización elaborados.

Igualmente, el tipo de aplicación colaborativa es de primera importancia:

- **Agendas Colaborativas.** Estas aplicaciones cooperativas requieren un algoritmo de coordinación que utiliza bloqueos y una sincronización para actualizar la información compartida,

- **Editores de Texto.** Pueden utilizar bloqueos parciales y una coordinación asíncrona para difundir las modificaciones a la información.

- **Pizarrones colaborativos.** No utilizan restricciones para escrituras, pero necesitan una coordinación síncrona elaborada.

- **Juegos de Video.** Pueden requerir bloqueos parciales, pero una coordinación sincrónica para difundir e integrar modificaciones a la información.

Estas necesidades también obligan a los programadores a escribir el código que realiza la coordinación de las actualizaciones como parte del código mismo de la aplicación. Esto no fomenta la reutilización de código ni extiende la vida útil de la aplicación y el mantenimiento de ese sistema es muy complicado.

La creación de un modelo genérico de replicación de entidades, que contemple todos estos aspectos es compleja, porque requiere mecanismos flexibles, extensibles y adaptables que sean capaces de incluir una variedad de opciones configurables. El objetivo es modificar el comportamiento del modelo para adaptarse a múltiples condiciones de operación. Todas estas cualidades incluyen ocultar por completo los mecanismos de distribución de información (*transparencia*) al nivel de la programación de las aplicaciones.

No es sorprendente que pocos se hayan atrevido a realizar esta empresa, pero creemos firmemente que nuestra propuesta constituye un avance notable para la administración de las entidades compartidas así como el diseño e implementación de las aplicaciones cooperativas.

5.2. Avances logrados

El desarrollo de este modelo en su estado actual (prototipo), provee avances en el área de los sistemas colaborativos distribuidos. Posee características que lo hacen capaz de extender su funcionalidad, modificar su comportamiento y adaptarse a nuevos requerimientos.

Este sistema puede soportar una variedad de políticas de replicación/notificación que provee respuestas a las necesidades de las aplicaciones sin necesidad de requerir, cambiar o introducir código a nivel de la capa de aplicación.

El análisis y diseño del sistema de actualización de entidades *SAE*, se sustenta en primer lugar del conocimiento adquirido, analizando los fundamentos teóricos en los primeros capítulos del presente trabajo de tesis. El desarrollo práctico se realizó aplicando los mecanismos propuestos en el capítulo de diseño logrando obtener un modelo (prototipo) que cumpliera con los objetivos generales y particulares trazados al inicio del trabajo de tesis.

Avances logrados desde el punto de vista teórico.

- **Estudio y caracterización de las diferentes arquitecturas de distribución.** Analizamos las arquitecturas centralizadas, híbridas y distribuidas, con el fin de obtener las características particulares de cada una de ellas. Siguiendo este estudio logramos visualizar una arquitectura que pudiera lograr/cumplir los objetivos que nos propusimos.

- **Estudio de protocolos de consistencia y replicación.** El estudio de estas técnicas de coordinación nos enseño la variedad de propuestas que existen en el área, los problemas, los retos, las ventajas y desventajas de utilizarlas. También logramos comprender lo que podemos y lo que aun no podemos realizar con este modelo.

Avances logrados desde el punto de vista practico.

- **Mecanismo de intercambio de políticas de replicación/actualización.** Permite el intercambio de políticas asociadas a una entidad por medio de la carga dinámica de clases, características que nos provee el lenguaje de desarrollo utilizado (*meta-programación*). Facilita la creación de nuevas políticas, porque no existe dependencia de estas clases con las clases del *SAE*. Estas políticas pueden ser configurables mediante un archivo de texto *XML*, con lo cual reducimos la necesidad de editar líneas de código.

- **Mensaje configurable.** Esta estructura de datos, utilizada como paso de mensajes entre cliente/servidor *SAE*, permite la inclusión de datos dentro de una estructura navegable y de fácil acceso. *XML* propone un estándar para el intercambio de información estructurada, es multiplataforma y su uso se extiende a muchas aplicaciones comunicadas vía Web. Podemos modificar su estructura para adaptarla a nuevos requerimientos.

- **Mecanismo construcción e interpretación de mensajes.** Este mecanismo nos permite interpretar los datos contenidos en un mensaje de tipo XML destinado a un servidor SAE, construir una operación relacionada a una política de replicación/actualización particular y ejecutarla. Esto nos da el poder de la flexibilidad: así ofrecimos la posibilidad de construir políticas de replicación/actualización distintas ocultando los mecanismos de distribución.

- **Mecanismo de replicación automática.** Por medio este mecanismo proveemos la capacidad de ejecutar periódicamente y automáticamente operaciones relacionadas a las políticas de replicación/actualización. Todo esto es configurado y administrable vía un archivo *XML*.

- **Mecanismo de almacenado persistente de objetos.** Gracias a este mecanismo podemos preservar datos en la memoria persistente con el fin de tolerar algunos tipos de fallas del *SAE*. También utilizamos esta característica como un repositorio temporal de clases con el fin de dar soporte al trabajo en modo desconectado en caso de desconexiones.

5.3. Trabajo a Futuro

En este apartado mostramos las mejoras al modelo que pueden desarrollarse, así como la inclusión de nuevas funcionalidades que mejorarían el desempeño del servidor *SAE*.

- Incluir un mecanismo para realizar transmisiones de datos seguras, actualmente no poseemos ningún mecanismo de encriptación de información, por lo que es susceptible a ataques o robo de información.
  Para mejorar la seguridad existen especificaciones que proporciona normas sintácticas y de procesamiento para crear y representar firmas digitales en archivos de tipo *XML* [URL: W3C Signature] [URL: XML Encryption].

  - *XML signature* especifica los métodos para asociar las claves con la información a la que hacen referencia. Utiliza reglas de procesamiento para representar firmas digitales las cuales se pueden aplicar a cualquier objeto.
  - *XML Encryption* Permite describir el modo de utilizar *XML* para representar recursos de forma digital y codificada. La especificación de *XML* Encryption esta pensada para utilizarla junto con la especificación de las Firmas digitales *XML*, así podemos firmar y cifrar el contenido de todas las mismo tiempo.

- Podemos mejorar el funcionamiento del proceso demonio de actualizaciones automáticas incluyendo un mecanismo capaz de adquirir conocimiento sobre las condiciones del entorno, y de esta manera realizar una decisión sobre el mejor momento para realizar cada actualización.
  Esto involucra adquisición de datos de los equipos (e.g. RAM, espacio en disco, condiciones de la red, ancho de banda) y de las condiciones del equipo local. (e.g. monitoreo de variables, CPU, preferencias de usuario).
  Utilizar redes neuronales artificiales, agentes o algún otro mecanismo capaz de adquirir conocimientos del entorno proporciona las condiciones necesarias para que un proceso pueda tomar decisiones sobre transmitir/aplicar una actualización o no.

- En el estado actual, el sistema sólo puede realizar conexiones de datos de tipo *TCP* y *UDP*. Una mejora consiste en añadir soporte para utilizar el protocolo *HTTP* y proporcionar servicios Web. Por lo tanto la interacción con un servidor SAE se realizaría por medio de algún sitio Web, esto podría extender su uso a otro tipo de aplicaciones no sólo aplicaciones colaborativas si no como sistemas de almacenamiento, bases de datos, albergue de aplicaciones. Necesitamos por lo tanto de un servidor de aplicaciones (e.g. Apache) y añadir mejoras al modulo de comunicación para dar soporte al protocolo *HTTP*.

- Con el fin de probar las capacidades del *SAE*, necesitamos realizar un análisis de aplicaciones colaborativas distribuidas representativas, utilizando nuestro sistema como mecanismo para la replicación/distribución de información. Esto implica llevar métricas, estadísticas de desempeño (performance), realizar comparaciones, crear tests de pruebas, etc., con la finalidad de afinar el modelo y llevarlo a un estado robusto y confiable. El desarrollo o adaptación de estas aplicaciones puede ser un

proceso largo y complicado. Por ello no contemplamos esta tarea dentro de los alcances de esta tesis de maestría.

- Debemos incluir mas políticas de control de concurrencia, varios tipos de políticas de replicación/actualización, optimistas y pesimistas basadas en técnicas: *primary-backup* [Budhijara et al., 1993], *maestro-esclavo*[Saito and Shapiro, 2005], *transferencia de operaciónes* [Li et al., 2000] [Sun and Ellis, 1998]. Con ello esperamos contar con un espectro mas amplio en técnicas soportadas y lograremos además mejorar algunos aspectos del *SAE*.

5.4. Apéndice

Groupware

Se refiere a todo aquel software colaborativo en que se tiene la noción de trabajar con un conjunto de personas, i.e. conciencia de grupo, se asocia al termino CSCW *computer supported cooperative work*, por sus siglas en ingles, definiciones y motivaciones para este término se describen en [Johansen, 1988] y [Ellis and Gibbs, 1999]. Algunos ejemplos de aplicaciones groupware: los editores colaborativos, los chats, vídeoconferencias.

Replicación

La replicación es una técnica de escalamiento y rendimiento que se utiliza para reducción de tiempos de acceso a recursos. Mantener la consistencia es un problema a la par de la replicación.

Transparencia

La transparencia hace referencia a aquella cualidad que posee un proceso o mecanismo de ocultar su funcionamiento ante los demás procesos, por lo que sólo es posible conocer las entradas y salidas de ese proceso. Algunos tipos de transparencia se muestran en la siguiente tabla:

transparencia	Descripción
Acceso	Oculta diferencias en la representación de datos.
Ubicación	Oculta donde esta localizado un recurso
Migración	Oculta si un recurso puede cambiar de ubicación
Reubicacion	Oculta si un recurso puede cambiar de ubicación mientras esta en uso
replicación	Oculta si un recurso es replicado
Concurrencia	Oculta si un recurso puede se compartido por usuarios competitivos
Falla	Oculta la falla y la recuperación de un recurso

Cuadro 5.1: Tipos de transparencia

Patrón de diseño componente

Es un enfoque que se basa en la idea de reutilización de código, se componen de una gran base de componentes de software reutilizables y de algunos marcos de trabajo de integración. Estos componentes pueden conformar sistemas por si mismos, se pueden utilizar para proveer de una funcionalidad específica. Existe toda una metodología para la creación de componentes o reutilización de código:

- Análisis de componentes
- Modificación de requerimientos
- Diseño del sistema con reutilizacion

- Desarrollo e integración

Todo esto tiene la finalidad de reducir tiempos y costos en el desarrollo de sofware [Sommerville, 2005]

IDL lnterfaz languaje definition

Si queremos realizar comunicación entre componentes del sistema necesitamos una manera de realizar la interacción, esto se logra a través de un punto de enlace o encuentro, es decir, una manera de hacer saber que servicios se ofrecen de un proceso a otro. Esta interacción se lleva a cabo por medio de las interfaces, IDL es un lenguaje común de definición de interfaces el cual nos permite crear interfaces neutrales que permiten a los diseñadores implementar funcionalidad adicional sin modificar la parte común, esto quiere decir que se preserva la portabilidad e interoperabilidad de los componentes y se permite incluso cambiar o reemplazar componentes sin afectar a los demás existentes, así se puede extender la vida útil de una aplicación.

RPC, Remote Procedure Call

Es un proceso para permitir llamadas a procedimientos remotos, por ejemplo que un equipo A pueda llamar a un procedimiento en un equipo B, mientras el proceso en A se suspende y la ejecución del procedimiento llamado ocurre en B. Se envía información a esos procedimientos por medio de los parámetros, ningún mensaje es visible al programador.

Estampa de tiempo

Una estampa de tiempo es un identificador simple que sirve para identificar cada transacción de manera única. Otra propiedad de las estampas de tiempo es la monoticidad, esto es, dos estampas de tiempo generadas por el mismo administrador de transacciones deben ser monotonicamente crecientes. Así, las estampas de tiempo son valores derivados de un dominio totalmente ordenado.

Bibliografía

[Aberer and Hauswirth, 2005] Aberer, K. and Hauswirth, M., *An Overview on Peer-to-Peer Information Systems*, Swiss Federal Institute of Technology (EPFL), Switzerland, pp. 1-14, 2005.

[Angulo, 2009] Angulo, J. S., *Arquitectura de distribución adaptable para sistemas colaborativos implantados en la Web*, Tesis de Maestría, CINVESTAV-IPN Zacatenco, Enero, 2009.

[Begole and Shaffer, 1999] J. Begole and C. Shaffer, *Flexible Collaboration Transparency: Supporting Worker Indendence in Replicated Application-Sharing Systems*, ACM,Virginia Polytechnic Institute, 1999.

[Begole et al., 2001] Begole J. Smith R., Struble A., and Shaffer A., *Resource sharing for replicated synchronous groupware*, John Wiley & Sons, pp. 833-843, 2001.

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[Budhijara et al., 1993] Budhijara N., Marzullo K., Schneider F., and Toueg S., *The primary-Backup Approach*, En Mullender, S. (ed.), Distributed Systems, pp. 199-216, 1993.

[Cook, 1990] Cook, W. R., *Object-Oriented Programming Versus Abstract Data Types*, Proceeding of the REXWorkshop/School on the Foundations of Object-Oriented Languages (FOOL), 489, Springer Lecture Notes in Computer Science, pp. 151-178, 1990.

[Decouchant et al., 2010] 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.

[Edwards, 1997] Edwards K.W., *Designing and implementing Asynchronous Collaborative Applications with Bayou*, ACM, Computing Surveys, pp. 119-128, ACM Press, 1997.

[Ellis and Gibbs, 1990] Ellis C.A. and Gibbs S.J., *Design and use of a group editor.*, In Engineering for Human-Computer Interaction., G. Cockton,Ed., Nort-Holland, Amsterdam, pp. 13-25, ACM Press, 1990.

[Ellis and Gibbs, 1999] C. Ellis and A. Gibbs, *Groupware: Some Issues and Experiences*, Morgan Kaufmann Publishers, 1993

[Ellis and Gibbs, 1992] Ellis C.A. and Gibbs S.J., *Awareness and Coordination in Shared Workspaces.*, In Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW'92), Toronto, pp. 107-114, ACM Press, 1992.

[Ellis and Gibbs, 1989] Ellis C.A. and Gibbs S.J., *Concurrency Control in Groupware Systems*, ACM, Computing Surveys, pp. 399-407, ACM Press, 1989.

[Ellis and Rein, 1988] Ellis CA., Gibbs S.J., and Rein G., *Design and Use of a Group Editor*, MCC Software Technology Program, 1988.

[Flanagan and Matsumoto, 2008] D. Flanagan and Y. Matsumoto, *The Ruby Programming Languaje*, O'REILLY, ACM, Sebastopol CA. (USA), 2008.

[Greif, 1988] I. Greif, *Computer-Supported Cooperative Work: A book of readings*, Morgan Kauffman, 1988.

[Herlihy and Wing, 1990] Herlihy, M. P. and Wing, J. M., *Linearizability: A correctness condition for concurrent objects.*, ACM Trans, Vol. 12, pp. 463-492, ACM Press, 1990.

[Jelasity et al, 2004] Jelasity, M., Guerraoui, R., Kermarrec, A. M., and Van Steen, M., *The peer sampling service: experimental evaluation of unstructured gossip-based implementations*,Proc. Middleware 2004. Berlin: Springer-Verlag, pp. 79-98, 2004.

[Johansen, 1988] Y. Saito and M. Shapiro, *Groupware: Computer Support for Business Teams*, The Free Press, N. Y., pp. 199-216, 1988.

[Johanson, 1998] N. Preguica, and H. Domingos, *Groupware: Computer Support for Business Teams*,The Free Press, 1988.

[Karger, 1997] D. Karger, *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*, proc. 29th. Annual ACM Symp., pp. 654-663, 1997.

[Lamport, 1978] Lamport,L., *Time, Clocks, and the Ordering of Events in a Distributed System*, ACM, Computing Surveys, Vol. 7, No. 2, pp. 558-565, 1978.

[Larman, 2005] Larman, C., *UML y patrones, Introducción al análisis y diseño orientado a objetos*, Prentice Hall, 2005.

[Laurillau and Nigay, 2002] Laurillau Y. and Nigay L, *Clover Architecture for Groupware*, CSCW'02, pp. 236-245, 2002.

[Lee et al., 2002] Lee Y.W., Leung K.S., and Satyanarayanan M., *Operation shipping for mobile file systems*, IEEE Transactions., Vol. 51, pp. 1410-1422, 2002

[Lua et al., 2005] Lua E. K., Crowcroft J., and Pias M., *A survey and comparison of Peer-to-Peer overlay networks schemes*, IEEE Communications Surveys, Vol. 7, No. 2, pp. 72-93, 2005.

[Li and Li, 2005] D. Li and R. Li, *An Approach to Ensuring Consistency in Peer-to-Peer Real-Time Group Editors*, Springer, Department of Computer Science, Computer Supported Cooperative Work, 2006.

[Li et al., 2000] Li D., Zhou L., Muntz R., and Sun C., *Operation Propagation in Real-Time Group Editors*, IEEE Multimedia, Vol. 7, No. 4, pp. 55-61, Oct.-Dec., 2000.

[Marsic, 2001] Ivan Marsic, *An Architecture for Heterogeneous Groupware Applications*, IEEE, 23rd International Conference on Software Engineering (ICSE'01), pp.475-484, 2001

[Mendoza, 2006] Mendoza, C.G., *Gestion d'entités partagées dans un environnement de production coopérative*, Laboratoire LSR-IMAG, Doctorado del Institut National Polytechnique de Grenoble, September 2006.

[Miller and Ferguson, 2005] Miller, J. and Ferguson, J.D., *Groupware Support for Asynchronous Document Review*, University of Strathclyde Department of Computer Science, 2005.

[Nuno et al., 2000] Nuno, P. J. Legatheaux, M., and Duarte, S., *Data Management Support for asynchronous groupware*, Computer Supported Cooperative Work Conference CSCW'00, pp. 69-78, 2000.

[Oravec, 1996] J. Oravec, *Virtual Individuals, Virtual Groups*, Cambridge University Press, Cambridge, 1996.

[Patterson, 1995] John F. Patterson, *A Taxonomy of Architectures for Synchronous Groupware Applications*, ACM, Special issue: workshop write-ups and positions papers from CSCW'94, Vol. 15, No. 3, pp. 27-29, 1995.

[Plaxton et al., 1997] C. Plaxton, R. Rajaraman, and A. Richa, *Accessing Nearby Copies of Replicated Objects in a Distributed Environment*, ACM, Computing Surveys, pp. 72-93, ACM Press, 1997.

[Preguica and Domingos, 2000] N. Preguica, and H. Domingos, *Data Management Support for Asynchronous Groupware*, CSCW Computing Surveys, Philadelphia PA, 2005.

[Rabinovich et al., 1999] Rabinovich M., Rabinovich I., Rajaraman R., and Aggarwal A., *A dinamic object replication and migration protocol for an internet hosting service*, Proc. 19th int'l conf. on a distributed computing systems, IEEE, pp. 101-113, 1999.

[Reeves, 1995] Reeves, Colin R., *Modern Heuristic Techniques for Combinatorial Problems*, McGrawHill Uk., 1995.

[Roseman and Greenberg, 1999] M. Roseman and S. Greenberg, *Groupware Toolkits for Synchronous Work*, In M. Beaudouin-Lafon (Ed.) Computer-Supported Cooperative Work: Trends in Software 7 ACM, 1999.

[Roseman and Greenberg, 1996] Roseman M. and Greenberg S., *Building real-time groupware with groupkit, a groupware toolkit*, ACM, Computing Surveys, pp. 66-106, ACM Press, 1996.

[Roxanne and Murray, 1978] S. Roxanne and T Murray, *The Network Nation: Human Communication via Computer.*, Addison-Wesley Publishing, 1978.

[Saito and Shapiro, 2005] Y. Saito and M. Shapiro, *Optimistic Replication*, ACM, Computing Surveys, Vol. 37, No. 1, pp. 42-81, ACM Press, 2005.

[Schuckmann, 2005] C. Schuckmann, *Designing object-oriented synchronous groupware with COAST*, Computer supported Cooperative Work, Cambridge MA USA ACM, Vol. 7, No. 2, pp. 30-38, 1996.

[Sommerville, 2005] Sommerville, I., *Ingeniería del Software*, Addison Wesley, 2005.

[Sun and Ellis, 1998] C. Sun and C. Ellis, *Operational Transformation in Real-Time Group Editors:Issues, Algorithms, and Achievements*, CSCW'98, 1998.

[Szymaniak et al., 2005] Szymaniak M., Perre G., and Van Steen M., *Latency-driven replica placement*, IPSJ Digital Courier, pp. 297, 2006.

[Sthepen and Olav, 2002] Sthepen S. and Olav M., *Patrones de diseño*, Prentice Hall, pp. 34-38, 2002.

[Sun et al., 2005] L. Kong, D. J. Manohar, A. Subbiah, M. Sun, M. Ahamad, and D.M. Blough, *Agile store: Experience with quorum-based data replication techniques for adaptative byzantine fault tolerance.*, In Proceedings of the International Symposium on Reliable Distributed Systems, 2005.

[Tanenbaum, 2007] Andrew S. Tanenbaum, *Sistemas Distribuidos Principios y Paradigmas*, Prentice Hall, No. 1, 2007.

[Tanenbaum, 2003] Andrew S. Tanenbaum, *Sistemas Operativos Modernos*, Prentice Hall, Vol. 2, 2003.

[Theimer et al., 1995] Douglas B., Terry and Marvin M., *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, ACM, Computing Surveys, pp. 172-183, ACM Press, 1995.

[Theimer et al., 1997] Douglas B., Terry and Marvin M., *Flexible Update Propagation for Weakly Consistent Replication*, ACM, Computing Surveys, pp. 288-301, 1997.

[Vahdat and Yu, 2002] Yu,H. and Vahdat,A., *Design and Evaluation of a Conit-Based Continuous Consistency Model por replicated Services*, ACM Transactions., pp. 277-279, 2002.

[Vahid et al., 1994] Vahid M., Chris F., and Jonh R., *Collaborative Asynchronous Inspection of Software*, ACM, Computing Surveys, 1994.

[Yang and Garcia-Molina, 2003] Yang, B. and Garcia-Molina, H., *Designing a super-peer network*, proc. 19th int'l conf. data engineering (Bangalore, India), IEEE Computer Society Press, pp. 40-60, 2003.

[Yu and Vahdat, 2002] Yu, H. and Vahdat, A., *Minimal replication cost for availability*, In 21st Symposium on Principles of Distributed Computing (PODC), 2002.

[Zimmermann, 1980] H. Zimmermann, *OSI Reference Model-The IS0 Model of Architecture for Open Systems Interconnection*, IEEE TRANSACTIONS, Vol. 37, No. 1, pp. 425-432, 1980.

Referencias Web

[URL: W3C Signature] *W3C Signature*, XML Signature WG, http://www.w3.org/Signature/.

[URL: XML Encryption] *Enabling XML security*, An introduction to XML encryption and XML signature, http://www.ibm.com/developerworks/xml/library/s-xmlsec.html/index.html