Consistencia y Latecoming en Aplicaciones Colaborativas

TRABAJO DE GRADO

Director : Luis Mariano Bibbó

Facultad de Informática
Universidad Nacional de La Plata

Trabajo de grado

Consistencia y Latecoming en Aplicaciones Colaborativas

Director: Luis Mariano Bibbó

Aixa Mauriello
aixa_mauriello@hotmail.com

Mariana Mela
marianamela@hotmail.com

Marzo de 2005

Facultad de Informática
Universidad Nacional de La Plata

Agradecimientos

Queremos dar las gracias ante todo a nuestro director de tesis Luis Mariano Bibbó, por habernos brindado todo su conocimiento, y más aún, por habernos guiado en el transcurso de toda nuestra tesis. Agradecerle las horas de trabajo, la buena predisposición y sobre todo, por haber generado un clima de trabajo sumamente amigable, en el que realmente fue un gusto para nosotras compartir con él.
Queremos agradecer también a las personas que desinteresadamente contribuyeron en este trabajo de grado: Piri, Mauricio y Nicolás.
Finalmente, no queremos olvidar a quienes compartieron con nosotras las horas de trabajo haciéndolas más amenas: Andrea, Patricia, Mariana, Hugo, Gabriel, Ezequiel, Sandra, Sergio, Mariané, por su compañerismo incondicional, y a nuestros jefes: Graciela y Ricardo, que afectuosamente nos acompañaron a lo largo del desarrollo de nuestra tesis, muchas gracias por habernos cedido tantas horas de trabajo.

Aixa y Mariana

Quiero agradecer principalmente a mis padres, Salvador y Mirta, por su constante apoyo y ayuda desmedida. Gracias por haberme brindado la oportunidad de alcanzar esta meta.
A mis hermanos, compañeros incondicionales.
A mi gran amiga Guillermina, mi hermana del alma, por estar siempre.
A mis abuelos, Yaya, Pedro, José y Toti, aunque muchos ya se han ido, aún en el recuerdo no quiero dejar de nombrarlos, porque fueron un gran sostén a lo largo de toda mi carrera. A mi familia en general, porque no quiero olvidar a nadie, les agradezco simplemente por estar.
A mis amigos de la facultad, a todos, por hacer de estos siete años una época especial.
No quiero dejar de mencionar a Aixa, compañera desde mi primer día en esta carrera y amiga desde hace años, con quien compartí el desarrollo de este trabajo de grado.
Y por último, quiero agradecer muy especialmente a la persona, que con su ejemplo, me enseñó que siempre se puede ser mejor en lo que uno hace, sin perder la humildad y la sencillez. Me demostró que alcanzar cualquier meta en la vida es posible, si ponemos nuestra mente y todo nuestro corazón en ella: Piri, gracias por hacer la diferencia en mi vida.
Al lector, dejo este trabajo de grado como una muestra de ello.

Con afecto
Mariana

Agradezco principalmente a mi mamá, Catalina, por haberme brindado la posibilidad de realizar mis estudios universitarios en la Ciudad de La Plata.
A mis compañeros de facultad, y a quien fue desde la primer semana mi compañera y amiga, Mariana.

A la familia de Mariana, la cual estuvo todos estos años y me hizo sentir en cada momento para de ella.
A Mauricio quien acompañó todo el desarrollo de esta tesis y soportó mis grandes peleas con las herramientas visuales.
Y principalmente agradecer a quien estuvo todas las horas sentada a mi lado, Feli.

Con cariño, Aixa

Indice General

Indice de Figuras

Capítulo 1

Introducción

Muchos de los sistemas de software actuales, tales como procesadores de textos o herramientas para consultar una base de datos, solo contemplan la interacción entre el usuario y el sistema.

Dado que una parte significativa de las actividades que desarrollan las personas, habitualmente, se lleva a cabo en un contexto mas bien grupal que individual, surge la necesidad de proveer un mecanismo que soporte además la interacción usuario-usuario.

Sin embargo, durante años, se ha ignorado el carácter esencialmente colectivo y cooperativo del trabajo humano, lo que ha llevado a construir sistemas de software multiusuario, tales como sistemas para el manejo de información de oficinas, que proveen un soporte mínimo para la interacción usuario-usuario.

Se ha supuesto que el trabajo individual no es influenciado por otros o no debería serlo, por esto los sistemas multiusuarios, no proveen a los usuarios el tipo de ayuda que es útil en las actividades grupales. Es más, los actuales sistemas de computación, incluidos los sistemas multiusuarios, han sido hechos pensando en la interacción aislada de un usuario con el sistema, dándole a los usuarios la ilusión de que están efectivamente solos interactuando con el computador. Apoyar la interdependencia que existe entre la gente afectaría en forma directa la eficiencia de los trabajos en una organización.

Por esta razón, hace un poco más de dos décadas, nació una disciplina científica denominada CSCW (Computer Supported Cooperative Work), o lo que es lo mismo, Trabajo Cooperativo Asistido por Ordenador [Grudin94].

Esta disciplina es considerada como una vía para describir cómo la tecnología de los computadores puede ayudar a los usuarios a trabajar conjuntamente en grupos [BS93].

Hoy en día, las aplicaciones que soportan el trabajo en grupo reciben la denominación genérica de *"groupware"* o también conocidas como *"aplicaciones colaborativas"* y se enmarcan dentro del área de investigación de CSCW, disciplina que intenta guiar en el correcto análisis, diseño y desarrollo de tales sistemas.

1.1 Aplicaciones Colaborativas

En la actualidad hay diversidad de definiciones que intentan establecer un concepto claro de *Aplicación Colaborativa* o también llamada *Aplicación Groupware.*

Específicamente en [EGR91] se define a las aplicaciones groupware como *"sistemas basados en computadora que ayudan a grupos de usuarios, que se comprometen en una tarea o meta común, y que proporcionan una interfaz de ambiente compartido"*

Cabe destacar que en la definición anterior los conceptos de *tarea común* y *ambiente compartido* son esenciales. Estos conceptos excluyen a los sistemas multiusuarios, tales como sistemas de tiempo compartido, cuyos usuarios pueden no compartir una tarea común, asimismo, la definición tampoco especifica que los usuarios estén activos simultáneamente.

Las aplicaciones colaborativas o aplicaciones groupware que específicamente soportan actividades simultáneas son llamadas *aplicaciones colaborativas de tiempo real* (ver Capítulo 2).

Sin embargo, no existe una línea divisoria rígida entre los sistemas que son considerados groupware y los que no lo son, por tal motivo se enmarcan a las aplicaciones groupware en un gran espectro multidimensional. Esto llevó a la necesidad de definir una taxonomía que pueda clasificarlas de acuerdo al momento en que se realizan las colaboraciones sincrónicas y asincrónicas) y por el lugar en donde se encuentran los participantes (cara a cara y a distancia) (ver sección 1.1.2).

Así es como existen muchos dominios de aplicaciones colaborativas. Por ejemplo, hay aplicaciones que facilitan la interacción a distancia entre miembros de un equipo de trabajo; otras que apoyan la comunicación de personas reunidas físicamente; otro tipo de aplicaciones, por ejemplo, ayudan a distintas personas en la confección de un mismo documento electrónico.

A pesar de los distintos tipos de aplicaciones que hay, el objetivo de todas ellas, cualquiera sea su dominio, básicamente es, asistir a un grupo de usuarios en la comunicación, colaboración y coordinación (ver sección 1.1.1) de sus actividades en forma conjunta a través de una computadora [EGR91].
En otras palabras, cualquiera sea el dominio de las aplicaciones colaborativas, todas comparten un factor común que las caracteriza: hacer más sencillo el proceso de compartir información entre un grupo de personas que realizan una tarea común.

1.1.1 Comunicación, Colaboración y Coordinación

Los sistemas colaborativos difieren de los sistemas tradicionales, al permitir que los usuarios interactúen directamente entre ellos, ya sea editando un documento, consultando una base de datos, etc., utilizando al computador como principal herramienta de interacción.

La interacción de un grupo de trabajo puede ser de distintas maneras, lo cual define un estilo de colaboración. Tal colaboración exige que las personas compartan información. Y para compartir información es imprescindible definir un modelo de comunicación.

La efectividad de la comunicación y colaboración puede reforzarse si las actividades de un grupo son coordinadas.

La coordinación de actividades es un aspecto sumamente importante, pues determina cómo puede compartirse la información. Es la mejor manera de asignar los recursos y otros aspectos relacionados a objetos comunes. Ayuda a la organización de las actividades de los miembros del grupo.

1.1.2 Taxonomía de Aplicaciones Groupware

Las diferentes taxonomías nos permiten clasificar el amplio espectro de aplicaciones groupware según diferentes criterios.

Existen cinco taxonomías:

- Taxonomía temporal/espacial
- Taxonomía de mapa 3x3
- Taxonomía según la funcionalidad de la aplicación
- Taxonomía según el acceso permisivo o restrictivos de los usuarios
- Taxonomía según el grado de estructuración de la información

1.1.2.1 Taxonomía temporal/espacial

Esta primer taxonomía [Johansen88] toma como dimensiones determinantes el tiempo y espacio, permitiendo así clasificar las aplicaciones groupware por el momento de tiempo donde se realiza la interacción (sincrónicas y asincrónicas) y por la distribución o lugar donde se encuentran los participantes (cara a cara y a distancia).

Utilizando entonces estas dimensiones tenemos cuatro grandes áreas:

- Sistemas que permiten que sus miembros interactúen en un mismo lugar o cara a cara, y en donde estas interacciones se llevan a cabo en un mismo momento (sincrónico)

- Sistemas que permiten la interacción asincrónica de los usuarios pero siempre en un mismo lugar, estos sistemas generalmente están orientados al manejo de datos e información

- Sistemas que permiten interacción distribuida en un mismo momento de tiempo y por lo tanto facilitan la realización del trabajo de personas que se encuentran geográficamente en lugares diferentes

- Sistemas que permiten interacción distribuida asincrónica, lo cual permite a los usuarios estar distribuidos geográficamente y poder trabajar juntos sin necesidad de realizar sus tareas en un mismo momento en el tiempo (asincrónico)

Si bien la clasificación que nos proporciona esta taxonomía es bastante amplia, se considera que la misma no cubre todo el espectro de aplicaciones colaborativas,

debido a que si bien podemos llegar a clasificar los sistemas según si los miembros trabajan tanto de forma sincrónica como asincrónica, deberíamos tener en cuenta el grado de colaboración de los usuarios.

	SINCRONO (mismo tiempo)	ASÍNCRONO (distinto tiempo)
LOCAL (mismo lugar)	REUNIONES CARA A CARA Pantalla compartida para explicaciones Utilidades con respuesta de la audiencia Entornos de conversación y tormentas de ideas (posible aplicación: toma de decisiones)	ADMINISTRACIÓN / MANEJO DE DATOS Raramente utilizado, un posible ejemplo: trabajo en turnos (en el mismo ordenador)
REMOTO (en lugares distintos)	REUNIONES REMOTAS Pizarra electrónica Charla (chat) Aplicaciones compartidas Video tele conferencia	MECANISMOS DE COORDINACIÓN Transferencia de ficheros Correo electrónico Grupos de noticias (news) Foros de debate (posible aplicación: toma de decisiones) Flujo de trabajo (workflow)

Figura 1.1 – Taxonomía Tiempo/Espacio

1.1.2.2 Taxonomía de Mapa 3x3

Esta taxonomía es una ampliación de la anterior, y la misma fue planteada por Jonathan Grudin [Grudin94]. En esta clasificación la actividad colaborativa puede llevarse a cabo en "tiempo real" (mismo tiempo), en diferentes momentos de tiempo pero predecibles (distinto tiempo pero predecible), por ejemplo cuando un usuario envía un correo electrónico a otro, se espera que el mismo sea leído al poco tiempo de su recepción; en distintos momentos pero los cuales son impredecibles (distinto tiempo pero impredecible), por ejemplo la creación en grupo de documentos.

En lo referente a la dimensión espacial, la actividad colaborativa puede realizarse en un mismo lugar, en distintos lugares pero conocidos por todos los usuarios o en distintos lugares pero no todos conocidos por los usuarios.

	MISMO TIEMPO	DISTINTO TIEMPO PERO PREDECIBLE	DISTINTO TIEMPO PERO IMPREDECIBLE
MISMO LUGAR	Reuniones cara a cara	Trabajo por turnos	Habitaciones de equipo
DISTINTOS LUGARES PERO PREDECIBLES	Video tele conferencia	Correo electrónico	Escritura colaborativa
DISTINTOS LUGARES PERO IMPREDECIBLES	Seminarios interactivos	Grupos de noticias	Flujo de trabajo

Figura 1.2 – Taxonomía Mapa 3x3

1.1.2.3 Taxonomía según la funcionalidad de la aplicación

Este esquema de clasificación de aplicaciones colaborativas fue propuesto por David Coleman [Coleman97], el mismo se encuentra orientado al producto, es decir centrado en la funcionalidad de la aplicación.

Esta clasificación propone la creación de doce categorías: correo electrónico y mensajería, calendario y programación de grupo, sistemas de encuentro o reunión electrónica (pizarras o foros de debate), sistemas de conferencia de escritorio en tiempo real (los usuarios pueden trabajar simultáneamente sobro un mismo documento), sistemas de conferencia en tiempo no real (tipo tablón de anuncios), flujo de trabajo, edición documental multiusuario (manejo de documentos de grupo), utilidades de trabajo en grupo y herramientas de desarrollo de aplicaciones colaborativas, servicios colaborativos, frameworks colaborativos y finalmente aplicaciones colaborativas basadas en Internet.

1.1.2.4 Taxonomía permisiva/restrictiva

Esta clasificación diferencia entre aplicaciones colaborativas restrictivas y aplicaciones colaborativas permisivas [GK90]. Las aplicaciones restrictivas dirigen el trabajo del usuario a través de una serie de pasos de cumplimiento obligatorio, por ejemplo un sistema de flujo de trabajo. Las aplicaciones permisivas, en cambio, dan al usuario la libertad a la hora de actuar, por ejemplo sistemas de conferencia en los cuales los usuarios participan en el momento que ellos consideren necesarios, pudiendo incluso no participar.
Si bien esta taxonomía nos permite hacer una clasificación de las aplicaciones colaborativas, la misma no es de uso frecuente debido a que varias aplicaciones colaborativas pueden presentar aspectos restrictivos y permisivos al mismo tiempo, por ejemplo las pizarras compartidas con moderador, en donde el moderador determina quien participa y quien no en determinado momento, pero una vez que se le otorga el permiso el usuario tiene la libertad de participar o no.

1.1.2.5 Taxonomía según el grado de estructuración de la información

Las aplicaciones colaborativas pueden tener una estructuración de la información baja, medio o alta. Si la estructuración es baja los sistemas no requieren que la información que facilitan y distribuyen entre los usuarios este muy estructurada, por ejemplo, sistemas de correo electrónico, los cuales se utilizan para el intercambio y para compartir información en cualquier formato o con cualquier estructura. Las aplicaciones con este grado de estructuración, proporcionan básicamente la comunicación entre los usuarios.

En segundo lugar, están las aplicaciones que requieren algo más de estructura en la información manejada, por ejemplo en una pizarra electrónica entre varios usuarios, cuanto más organizada y estructurada esté la información aportada al espacio común, más fácil será su tratamiento y gestión. Estas aplicaciones aportan, mas que comunicación entre los usuarios, la interacción sobre objetos de información compartidos [Rodden93]

Finalmente, aplicaciones colaborativas basados en la coordinación, es decir aplicaciones que requieren que la información tratada esté bien estructurada, por ejemplo, los sistemas de flujo de trabajo modelan la secuencia de tareas que se suceden en un proceso de trabajo y los roles que desempeña cada uno de los participantes. Cuando una tarea es completada, el trabajo es automáticamente encaminado a la persona encargada de la siguiente tarea, todo este proceso solo es factible si la información está lo suficientemente bien estructurada.

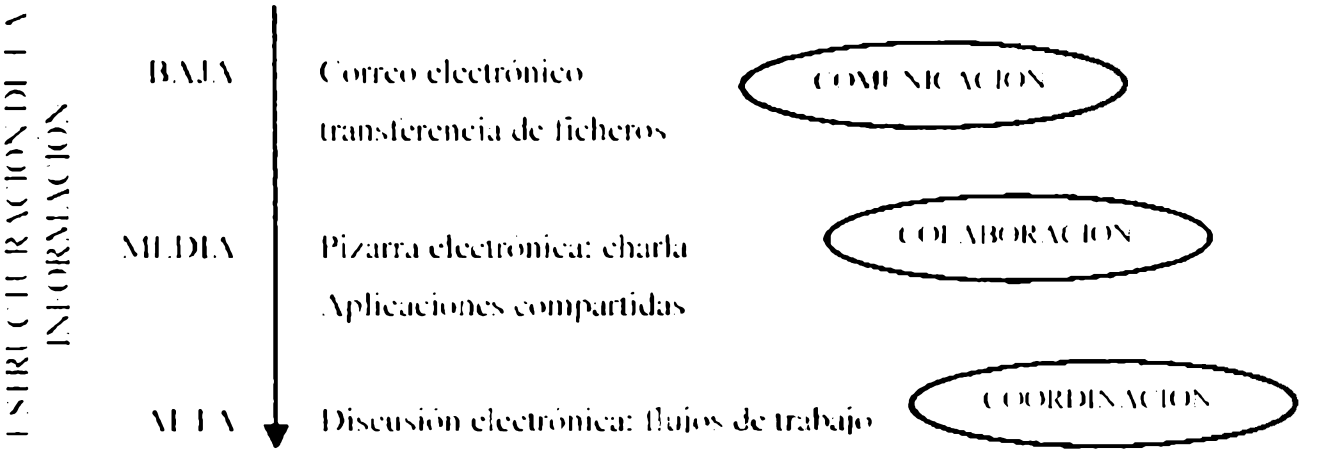


Figura 1.3 – Taxonomía según el grado de estructuración de la información

1.2 Motivación

A pesar de la existencia de una gran variedad de aplicaciones groupware, que soportan muchas tareas cooperativas, varios estudios de la CSCW encontraron que tales aplicaciones no soportan adecuadamente la variabilidad del trabajo cooperativo en las organizaciones.

Sin embargo desarrollar esta clase de aplicaciones es una tarea compleja, dado que involucran diversas áreas de trabajo, tales como sistemas distribuidos, comunicaciones, interfaces humano-computador, bases de datos, y algunas otras áreas más.

En aplicaciones groupware que desempeñan funciones en un espacio de trabajo compartido, tales como las aplicaciones colaborativas sincrónicas, es imprescindible mantener alguna clase de consistencia entre dos o más representaciones del mismo espacio de trabajo compartido, a pesar de las actividades concurrentes de los usuarios en el mismo instante de tiempo.

Así es que uno de los problemas con los que deben enfrentarse los desarrolladores de tales aplicaciones, es que muchas veces las actividades concurrentes de los usuarios conducen a acciones concurrentes sobre el mismo espacio de trabajo compartido, lo cual puede causar representaciones inconsistentes de la misma información.

La fundamental causa de inconsistencia es que las acciones se llevan a cabo en el mismo instante de tiempo. Cuando un usuario lleva a cabo alguna acción sobre un conjunto de datos, otro usuario puede estar modificando los mismos datos en el mismo instante de tiempo, pero debido al tráfico de información en la red, tales acciones se procesan en diferente orden, y es allí donde se producen las inconsistencias. Existen dos caminos para tratar este problema, los cuales serán detallados en la sección 2.4.

Otro de los grandes inconvenientes en sistemas con espacio de trabajo compartido es cómo permitir la incorporación tardía de un nuevo usuario en el sistema, ya que esta acción requiere que al mismo se le informe la secuencia ordenada de las tareas efectuadas hasta llegar al estado actual del sistema. Esto nos enfrenta al problema de cómo informar al nuevo usuario sin afectar el trabajo actualmente realizado por el resto de los usuarios en el sistema (ver sección 2.4)

Dado que hoy en día, cada vez es mayor la necesidad de contar con sistemas con espacio de trabajo compartido, donde se permita además la interacción usuario-usuario en el mismo instante de tiempo (aplicaciones colaborativas sincrónicas), es de vital importancia el análisis y estudio de estos dos problemas.

El desarrollo de esta tesis intenta ofrecer una solución a los problemas de consistencia y latecoming planteados en esta sección, sobre la base del análisis de los frameworks DYCE (IPSI) [sección 4.3] y COAST (IPSI) (ver sección 4.2) y el estudio del servidor de aplicaciones SAMETIME(Lotus) (ver sección 4.4) que ofrece una arquitectura óptima para el manejo de comunicaciones.

En lo que respecta a los frameworks mencionados antes, cada uno define un modelo de arquitectura a seguir, teniendo en cuenta la robustez del sistema, performance, claridad del modelo de arquitectura, flexibilidad y mantenimiento del mismo.

Se espera, a partir del estudio de dichos frameworks, obtener un nuevo modelo de arquitectura para el desarrollo de aplicaciones colaborativas sincrónicas, que cumpla con las propiedades antes mencionadas garantizando consistencia de información y dando soporte a la incorporación de nuevos usuarios en el sistema (latecoming), a través de un camino sencillo y flexible.

Es en este contexto que se da la realización del presente trabajo, cuya principal contribución radica en satisfacer y dar una solución a los problemas de consistencia y latecoming antes mencionados.

1.3 Objetivos

De acuerdo a lo expresado y planteado en este capítulo, los objetivos del presente trabajo son:

- Establecer una definición clara de Aplicación Groupware, también conocida como Aplicación Colaborativa

- Analizar el desarrollo de las Aplicaciones Groupware Sincrónicas, y centrar dicho análisis en dos problemas esenciales: Latecoming y manejo de consistencia en ambientes con espacio de trabajo compartido.

- Definir una arquitectura que proporcione una solución a los problemas citados anteriormente, de acuerdo al análisis de dos frameworks: Dyce, Coast, y sobre la base del estudio del servidor Sametime, que guiarán en el desarrollo de dicha solución.

Capítulo 2

Aplicaciones Colaborativas Sincrónicas

Un sistema multi-usuario implica la presencia y participación simultánea de todos los usuarios. Esta clase de sistemas es clasificada como sistemas sincrónicos los cuales difieren ampliamente de los sistemas asincrónicos, donde la colaboración se permite en cualquier momento, sin la presencia simultánea de todos los miembros del equipo.

Como se mencionó en la sección 1.1.2, los diferentes tipos de interacción pueden ser considerados en una matriz Espacio/Tiempo, la cual esta basada en dos dimensiones principales de sistemas colaborativos: por el lugar donde se encuentran los participantes (Espacio) y por el momento en que se realizan las actividades (Tiempo).

Un sistema colaborativo, basado en interacción distribuida sincrónica o interacción en tiempo real, soporta a un grupo de trabajo en el mismo momento y en diferentes lugares, por ejemplo sistemas de conferencias en tiempo real o programas de diseño compartido. Tales aplicaciones son llamadas *Aplicaciones Colaborativas Sincrónicas.*

Lo mencionado anteriormente puede resumirse según una definición dada por [LL02]:

> *"Los Sistemas Colaborativos Sincrónicos permiten a un grupo de usuarios distribuidos geográficamente colaborar en una tarea común en el mismo instante de tiempo."*

2.1 Herramientas Colaborativas

En esta sección se describen varios tipos de aplicaciones colaborativas sincrónicas existentes y se mencionan algunos sistemas representativos de cada uno de ellos.

- ***Videoconferencia:***

  Desde la aparición del video a través de la telefonía en películas de ciencia-ficción y la introducción de imágenes también por medio del teléfono por AT&T en la Feria Mundial de 1964, los diseñadores han estado explorando el uso del video para apoyar el trabajo cooperativo a distancia.

  A pesar de una falta de hallazgos en la investigación respecto de que el video refuerza muchos tipos de trabajo cooperativo (Ej. [Egid88; OCL+93;

Heer96]), los investigadores han continuado su investigación intentando encontrar aplicaciones útiles para esta tecnología. Esto ha llevado al desarrollo de prototipos e investigación de sistemas con un enfoque en varios de los aspectos de comunicación a través del video.

Un ejemplo de tales aplicaciones son los sistemas como CRUCERO [Root88] y Montage [TaIR94] (vea Figura 2.1), que permite ver qué persona está disponible para la conversación.

En general este tipo de sistemas permiten la transmisión de audio y video entre varias personas. Las comunicaciones son del tipo 1-1, 1-N o N-N.

Figura 2.1 – Montage: Herramienta de Videoconferencia

- ***Pizarras Compartidas (Shared Whiteboards):***

En muchas reuniones formales e informales, particularmente aquéllas que involucran discusiones sobre diseño y otros aspectos complejos, generalmente se encuentra a las personas dibujando bocetos (Ej, en un whiteboard o pizarra), señalando artículos particulares y relaciones.

Otras personas en la reunión pueden referirse a tales dibujos, y pueden proponer modificaciones alterando el dibujo.
Por esta razón se diseñan sistemas de pizarras compartidas para apoyar tales reuniones, particularmente cuando los participantes no están en el mismo cuarto. Los objetos dibujados en el espacio de trabajo compartido proporcionado por tal pizarra son inmediatamente visibles para todos los otros usuarios conectados.

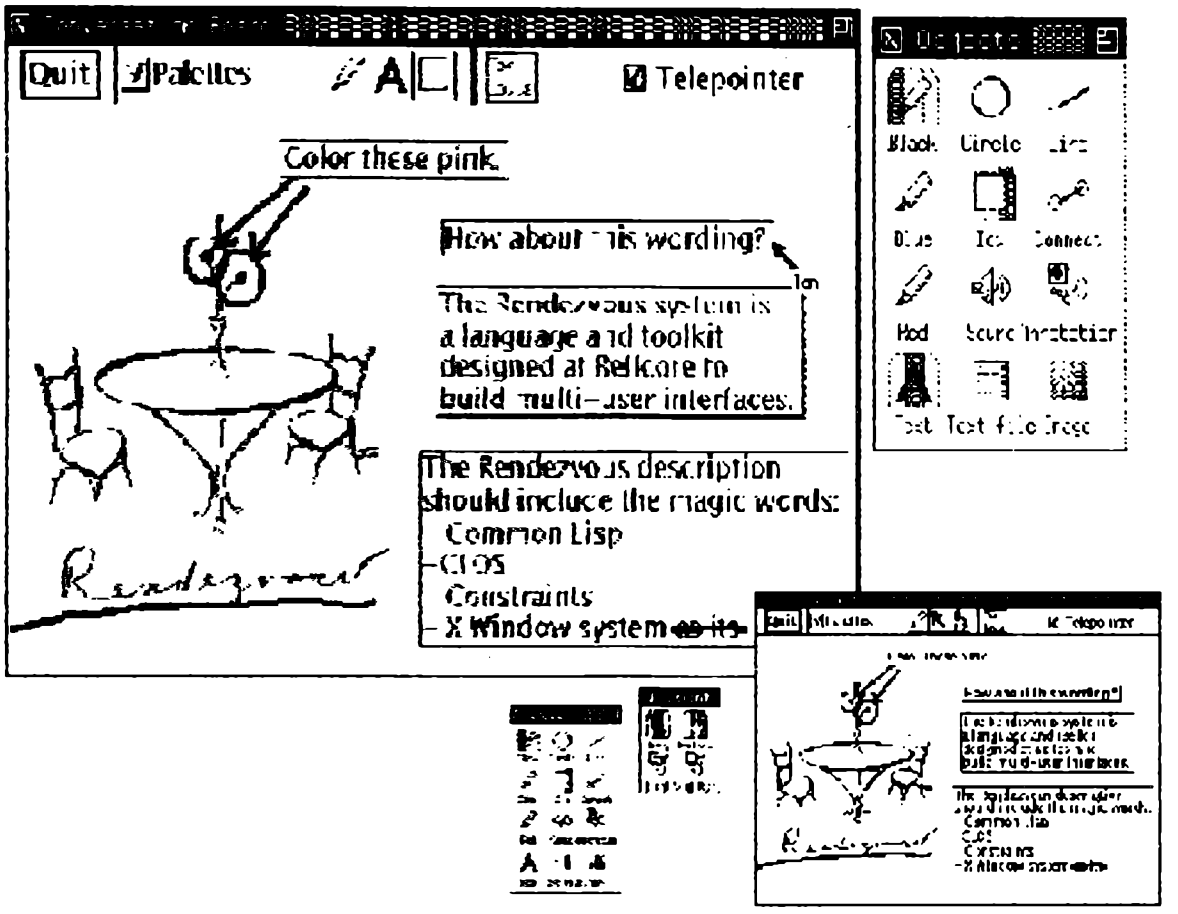


Figura 2.2 – Pizarra Compartida

- ***Sistemas de Chat:***

  Similar al E-mail y a los sistemas de conferencia de computadora, los sistemas de chat proporcionan discusiones basadas en texto mediadas a través de la computadora entre los usuarios.

  A diferencia del E-mail y los sistemas de conferencia actuales, cada párrafo o frase que se teclean, es inmediatamente visible en las pantallas de otros usuarios, que facilitan una interacción rápida en las discusiones o charlas.

  Los primeros Sistemas de chat, como el chat de UNIX, empezaron como rasgos suplementarios de sistemas operativos, facilitando la comunicación entre dos usuarios de una red.

  Más recientemente, sistemas como Internet Relay Chat (IRC) (vea Figura 2.3) proporcionan comunicación interpersonal para un número arbitrario de usuarios conectados a Internet.

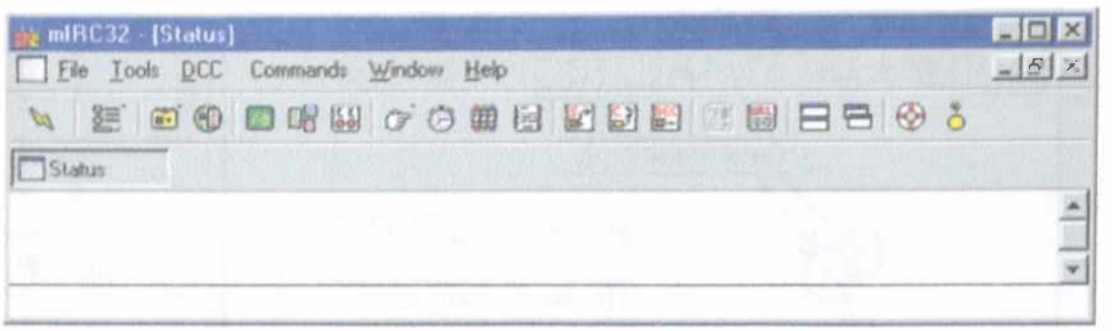


Figura 2.3 – Sistema de chat

- ***Sistemas de Edición Colaborativa:***

  Una de las tareas más frecuente que soportan las computadoras, es la creación de documentos (authoring) ofreciendo soporte a las personas para que logren llevar a cabo dicha tarea.

  Hasta hace poco, los editores convencionales y principalmente los procesadores de textos, ayudaban a los individuos a refinar los documentos para producir una copia final con un pequeño soporte para co-authoring.

  Sin embargo, muchos de los documentos son el resultado de la cooperación entre dos o más personas. Por ejemplo, en campos de la ciencia el 65% de los artículos son escritos por dos o más autores [FKL+88].

  En general, los sistemas de edición colaborativa, permiten a varios colaboradores o autores, crear un documento haciendo anotaciones sobre él simultáneamente y de forma que cada uno sea capaz de ver las anotaciones realizadas por los demás a la vez que realiza las suyas propias.

  Un ejemplo específico de estos sistemas es la herramienta SEPIA, analizada en [HW92]:

Los elementos de procesamiento de estos sistemas, requieren inferir propiedades del sistema global y mantener la consistencia del estado global, observando y manipulando parámetros locales.

Una perspectiva de desarrollo de estos sistemas, es tratar de aplicar en ellos los mejores resultados de la investigación de la teoría de los sistemas distribuidos, tales como los algoritmos para sistemas operativos distribuidos y las bases de datos distribuidas. Por ejemplo, la aplicación de algoritmos para hacer más tolerantes a fallas a los sistemas colaborativos, resolviendo problemas tales como la replicación de datos, el mantenimiento de servidores consistentes y la búsqueda eficiente de información distribuida.

Comunicación: La contribución de las comunicaciones trata con el intercambio de información entre agentes remotos, ya sean humanos o de computador.

En los sistemas colaborativos, uno de los desafíos de las comunicaciones es hacer que las interacciones distribuidas sean tan efectivas como las interacciones cara-a-cara, o incluso que sean una alternativa que las reemplace.

Esto significa proveer mayor interconectividad y ancho de banda, así como también protocolos para el intercambio de información de diferente tipo, como por ejemplo texto, imágenes, voz, video, o realidad virtual.

Interacción humano-computador: importancia de las interfaces de usuarios en los sistemas de cómputo.

La interacción humano-computador es un campo multidisciplinario, incluye diversas técnicas de diseñadores industriales y gráficos, expertos en gráficos de computadoras y también involucra conceptos de las ciencias cognitivas.

Estas interfaces son sensitivas a factores como grupos dinámicos y la estructura organizacional, factores que normalmente no son considerados relevantes en el diseño de interfaces de usuario. Sin embargo es de vital importancia que las ciencias sociales y los usuarios finales jueguen un rol relevante en el desarrollo de interfaces de grupos.

Inteligencia Artificial: La contribución de la inteligencia artificial hasta ahora no ha sido relevante en los sistemas colaborativos, sin embargo su potencial contribución es muy promisoria.

Por ejemplo, un sistema podría simular participantes con algunas características humanas (agentes inteligentes) cuando el número de participantes es menor que el requerido por la aplicación.

El agente inteligente ejecutaría acciones que serían vistas por los demás usuarios como acciones de usuarios corrientes.

El enfoque de la inteligencia artificial también puede ser apropiado para construir manejadores de conferencia suficientemente flexibles como para atender grupos muy diferentes entre sí, así como también atender comportamientos diferentes de un mismo grupo.

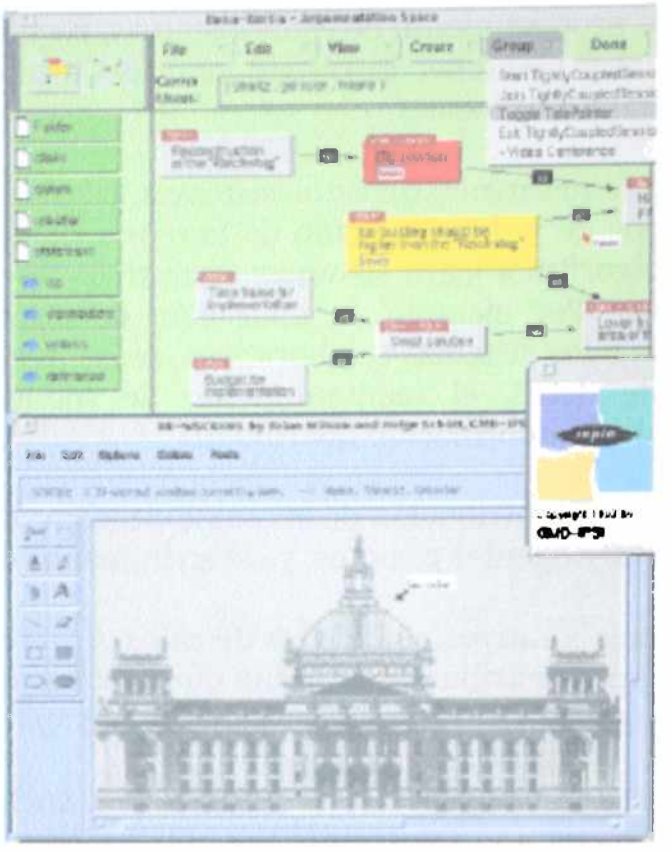


Figura 2.4 – SEPIA: Sistema de edición colaborativa

2.2 Metodología y aspectos generales para la Construcción de una Aplicación Colaborativa Sincrónica

Las aplicaciones colaborativas en general, se basan en la metodología y contribución de muchas disciplinas, en particular hay al menos cinco disciplinas o perspectivas para la construcción de una aplicación colaborativa exitosa:

- Sistemas Distribuidos
- Comunicación
- Interacción humano-computador
- Inteligencia Artificial
- Teoría Social

A continuación se detalla cada disciplina.

Sistemas Distribuidos: La contribución de los sistemas distribuidos se relaciona con la descentralización de los datos y el control. Considerando que los trabajadores de muchas actividades colectivas frecuentemente están distribuidos en el espacio y el tiempo, los sistemas para apoyar estas actividades se ajustan en forma natural al carácter de los sistemas distribuidos.

Teoría social: esta perspectiva hace hincapié en la teoría social o sociología en el diseño de sistemas colaborativos. Los sistemas diseñados desde esta perspectiva abarcan los principios y explicaciones derivadas de la investigación sociológica.

El aporte de la teoría social trata con el uso de los groupware, analizando los procesos de grupo (con o sin la tecnología de por medio), analizando sus efectos en las organizaciones humanas así como en el diseño futuro de estos sistemas.

Evidentemente, estos cinco dominios de estudio no son independientes entre sí, en los sistemas colaborativos ellos se combinan y benefician mutuamente.

Los investigadores en CSCW han identificado además varios aspectos que deberían incorporarse específicamente en un sistema colaborativo sincrónico. Sin embargo, la incorporación de ellos dependerá del sistema a desarrollar:

> ***- Grado de Cooperación (Task Coupling)***
> Nivel de colaboración de los miembros que trabajan juntos. Un sistema colaborativo debe proveer una colaboración flexible
>
> ***- What You See Is What I See (WYSIWIS)***
> Define un modo de colaboración sincrónica donde todos los participantes ven exacta o aproximadamente lo mismo.

Este aspecto es demasiado estricto. Seguir este principio de forma rigurosa será útil dependiendo en qué tipo de entorno quiere desarrollarse la aplicación.

Debido a que para muchas colaboraciones WYSIWIS no es útil se han propuesto cuatro dimensiones en las que las restricciones de WYSIWIS pueden ser más flexibles.

Dimensiones del WYSIWIS que pueden ser más flexibles:

- **Espacio**

Las localizaciones de todos los elementos son las mismas siguiendo las directivas de WYSIWIS. Sin embargo, a veces es útil permitir a los colaboradores tener diferentes vistas de los elementos compartidos

- **Tiempo**

Bajo WYSIWIS de forma estricta, los cambios en la información compartida son vistos por todos los colaboradores casi simultáneamente.
Flexibilizando esta restricción, podemos permitir que el sistema envíe modificaciones en intervalos, quizás diferentes espacios de tiempo para cada colaborador. Si se flexibiliza totalmente esta característica se obtiene un sistema asincrónico

- **Población**

Siguiendo WYSIWIS todo el grupo ve los mismos datos compartidos. A veces puede ser útil que los subgrupos se comuniquen entre ellos, sin necesidad de que el grupo entero lo vea.

- **Compatibilidad**

Bajo WYSIWIS todo el mundo ve la misma representación de los datos compartidos, haciendo más flexible esta restricción podemos proveer a cada colaborador con diferentes representaciones, quizás dependiendo del rol en la colaboración.

De lo dicho anteriormente sobre el aspecto WYSIWIS podemos concluir que su principal ventaja radica en que todos los colaboradores saben que están viendo lo mismo. Sin embargo, este aspecto es demasiado estricto. El mayor inconveniente surge en cómo hacer más flexibles las restricciones de WYSIWIS, de tal forma que los colaboradores no pierdan la noción de que están trabajando juntos. Como solución a este problema surge el concepto de Group Awareness.

2.3 Group Awareness

2.3.1 Concepto

Cuando los usuarios trabajan juntos, en el mismo momento y en el mismo lugar, saben lo que realiza cada uno de los integrantes del grupo. Sin embargo, cuando los colaboradores están distribuidos, mucha de esta información se pierde. Sin esta información los grupos tienen dificultad para coordinar el control de acceso, comunicación y trabajo.

Se define en [EGR91] Group Awareness como un mecanismo que provee un contexto grupal actualizado y notificaciones de las acciones de cada usuario cuando sea apropiado en un espacio de trabajo compartido.

2.3.2 Aspectos Fundamentales

En un entorno monousuario es importante notificar al usuario cuando determinadas restricciones se violan o cuando operaciones automáticas provocan triggers o alertas.

La notificación o awareness se hace más importante aún en entornos multiusuario, ya que los usuarios deben saber cuando otros usuarios efectúan cambios que impactan al trabajo común. Es por eso que las aplicaciones colaborativas deben hacer énfasis en el mecanismo de awareness, una forma de alertar y modificar la interface de un usuario en respuesta a acciones llevadas a cabo por otro usuario en otra interface.

Existen diferentes tipos de awareness, en [JCBZ99] se da una posible clasificación de los mismos. Los diferentes tipos tienen que ver con los usuarios de la aplicación, los objetos manipulados por ellos, las acciones que se llevan a cabo, capacidades de los distintos tipos de usuarios, etc.

A continuación se mencionan algunos aspectos importantes relacionados con el concepto de Awareness.

- Percepción de Asistencia (Presence Awareness): permite saber qué usuarios están disponibles, donde se encuentran y la actividad realizada por cada uno de los miembros del grupo

- Percepción del compromiso o interés (Engagement Awareness): es el conocimiento del nivel de compromiso de un colaborador en la actividad.

    - Incluye aspectos como el interés que muestran los otros colaboradores, la atención prestada y el estado emocional de estos.

    - Cuando se trabaja en el mismo lugar estas características son fácilmente perceptibles, no ocurre lo mismo en los sistemas distribuidos.

- Percepción de jerarquía (Structural Awareness): conocimiento de los roles de los colaboradores

- Percepción del área de trabajo (Workspace Awareness): definido como el conocimiento al instante de las interacciones y localización de otros participantes dentro del área de trabajo.

- Concurrencia: múltiples usuarios trabajan simultáneamente dentro de una aplicación colaborativa, esto genera accesos concurrentes a un recurso compartido que podrían crear conflictos.

    - Debido a esto según la aplicación se necesitarán o no diferentes mecanismos de bloqueo, mediante los cuales podemos dar privilegios de acceso a los recursos a algunos usuarios y denegarle a otros el acceso.

    - Específicamente las aplicaciones colaborativas sincrónicas deben tratar con estos conflictos, dado que tales acciones pueden generar inconsistencias en la información, problema que será desarrollado en la sección 2.4.1

- Separación de interfaz y datos: permite acceder a los colaboradores a datos desde diferentes localizaciones. Los datos de la aplicación deben ser separados de su interfaz

- Roles de los usuarios: en colaboraciones usuales, cada colaborador realiza una subtarea diferente. Idealmente una aplicación colaborativa deberá soportar cada rol.

- Llegada tarde (Latecoming): muchas aplicaciones colaborativas sincrónicas necesitan soportar situaciones donde los usuarios se incorporan tarde a la sesión, en estos casos se plantea la cuestión de cómo se llevara el estado actual al nuevo usuario, sin perturbar demasiado a los demás colaboradores, tema que será tratado en la sección 2.4.2

2.4 Problemas que afectan el desarrollo de aplicaciones colaborativas sincrónicas

Muchos trabajos en el área de los sistemas colaborativos se han centrado en la resolución de problemas puntuales. Sin embargo, si nos fijamos en cómo la gente interactúa en una categoría específica, veremos que la mayoría de las aplicaciones que caen en dicha categoría cumplen requisitos comunes. Estos requisitos son los relacionados con el estilo de colaboración y pueden ser provistos por una plataforma.

La ventaja de contar con una plataforma es que disminuye el tiempo y complejidad de desarrollo de una aplicación colaborativa.

Desafíos relevantes que se encuentran en la implementación de este tipo de sistemas dependen de algunos requisitos que determinan el tipo de arquitectura de cómputo sobre la cual se apoyan estos sistemas.

Visualizamos cuatro de estos requisitos:

- Tamaño del grupo
- Grado de acoplamiento de las interfaces
- Tiempo de respuesta corto
- Disponibilidad del sistema.

El ***tamaño máximo del grupo*** pone dificultades para proveer intercambio de datos satisfactorio entre los procesos, esto quiere decir que si todos los participantes del grupo se encuentran interactuando, el desempeño del sistema no se degrada notoriamente.

El tiempo de respuesta que se espera para cada usuario es del orden del tiempo de respuesta de un sistema multiusuario de tiempo compartido.

En el caso más relajado, los procesos deberían poder intercambiar libremente información, esto significa permitir que los nodos formen un grafo fuertemente conectado. Luego, un incremento en el número máximo de usuarios multiplica la complejidad en los distintos componentes del sistema, ya que cada usuario introduce un nivel de actividad mayor y un más alto grado de concurrencia.

El ***grado de acoplamiento de las interfaces*** está muy unido con el nivel de granularidad de la información que se necesita que compartan los usuarios, lo cual depende de las características de la aplicación colaborativa. En un sistema de archivos compartido la granularidad puede ser al nivel de archivos. En un editor de texto multiusuario la granularidad puede ser al nivel de caracteres.

El grado de acoplamiento es también un factor que incide fuertemente en el intercambio de datos. El problema más serio es el tiempo que toma notificar a todos los usuarios de los cambios en la información compartida. En efecto, un acoplamiento fuerte exige al sistema que todos los usuarios dispongan de un mismo display de información obtenido de un espacio de información común, de modo que, cuando se produzca algún cambio en algún display, ese cambio se refleje también en todos los demás displays. Esto requiere de sistemas de comunicación sincrónicos de muy alta velocidad. La complejidad crece cuanto mayor es el acoplamiento y la conciencia de grupo que se desea otorgar a los participantes.

El ***tiempo de respuesta*** viene determinado por los dos requisitos anteriores, el tamaño máximo del grupo y el grado de acoplamiento de las interfaces. Una forma de proveer tiempos de respuesta cortos es reduciendo el intercambio de mensajes por la red. Esto requiere mantener copias de ciertos estados del sistema en cada estación de usuario, para reducir el intercambio de datos por la red cada vez que un usuario interactúa con el sistema.

De acuerdo con esto, una arquitectura de cómputo colaborativa puede ser centralizada, replicada o híbrida. (Temas que serán tratados detalladamente en el Capítulo 3)

En una arquitectura centralizada, la aplicación se encuentra centralizada en un solo computador, al igual que la información compartida. La ventaja de una arquitectura centralizada es su simplicidad, ya que los datos se manejan de manera centralizada, el acceso a los datos y mantener su consistencia es simple.

En una arquitectura replicada existe una réplica de la aplicación y de los datos en cada estación de trabajo de un usuario. Esta solución obliga a enfrentar problemas de sincronización e inconsistencia de los datos, pero podría proveer tiempos de respuesta más cortos.

Ya que tanto las arquitecturas centralizadas como las replicadas ofrecen beneficios y limitaciones, una alternativa frecuente es la arquitectura híbrida entre centralizada y replicada.

En una arquitectura híbrida existe un servidor que ejecuta la aplicación y mantiene centralizadamente los datos compartidos, pero en cada estación de trabajo de un usuario hay un caché de información compartida, que ayuda a reducir el tiempo de respuesta del sistema.

Sin embargo, cualquiera sea el tipo de arquitectura en el que se base el desarrollo de nuestra aplicación colaborativa, debemos enfrentarnos al problema de cómo garantizar consistencia de datos, sin olvidarnos de proveer un tiempo de respuesta corto (sección 2.4.1).

El ***requisito de disponibilidad*** se refiere a la recuperación del sistema cuando los componentes fallan. La disponibilidad también depende de la arquitectura de cómputo elegida. Una arquitectura centralizada, por ejemplo, es fuertemente sensible a la falla del servidor central. Hacer el sistema más tolerante a las fallas requiere de una arquitectura replicada y un manejo apropiado para reaccionar a las fallas.

De lo dicho anteriormente podemos concluir que el desarrollo de toda aplicación colaborativa requiere del análisis de estos cuatro requisitos: tamaño de grupo, acoplamiento de interfaces, tiempo de respuesta corto y disponibilidad del sistema.

Sin embargo se ha sumado un gran desafío en el desarrollo de las aplicaciones colaborativas sincrónicas, debido a que se ha introducido un nuevo requisito denominado *"Latecoming"*. Como se mencionó en la sección 2.4.2 y en la sección 1.2, este concepto trata sobre cómo permitir la incorporación tardía de un usuario a la sesión de trabajo en un ambiente colaborativo. Este concepto será tratado detalladamente en la sección 2.4.2.

Nuestro análisis se centra entonces en dos requisitos puntuales que afectan directamente el desarrollo de una aplicación colaborativa sincrónica, en primer lugar cómo garantizar consistencia de información en un espacio de trabajo compartido, y en segundo lugar cómo permitir la llegada tarde de un usuario a la sesión de trabajo compartida sin afectar el trabajo de los otros usuarios en la sesión, a través de un camino sencillo y flexible.

2.4.1 Consistencia

Como se dijo en la sección 1.2 la fundamental causa de inconsistencia en ambientes colaborativos sincrónicos es que las acciones que realizan los usuarios, se llevan a cabo en el mismo instante de tiempo.
Cuando un usuario lleva a cabo alguna acción sobre un conjunto de datos, otro usuario puede estar modificando los mismos datos en el mismo instante de tiempo, pero debido al tráfico de información en la red, tales acciones se procesan en diferente orden, y es allí donde se producen las inconsistencias.

El siguiente es un ejemplo de ello [LL02], considerar dos usuarios A y B, ambos inician una sesión de edición de grupo en un editor colaborativo desde el mismo estado inicial de un documento, el cual es representado como el string "abc". Supongamos que A inserta el caracter 'x' en la posición 1 del string "abc", antes de 'a'. Y al mismo tiempo B inserta el caracter 'y' en la posición 2 entre 'a' y 'b'. De este modo estas dos operaciones, el insert de A y el insert de B son operaciones concurrentes. Como resultado de esto el estado del documento en A será "xabc" y el estado en B será "aybc". Esto muestra que el resultado de la ejecución concurrente de las operaciones generaron estados inconsistentes del mismo documento inicial "abc".

Por ello, los sistemas colaborativos, necesitan de un control de concurrencia para resolver estos conflictos entre operaciones simultáneas por parte de los usuarios en un espacio de trabajo compartido.

En [EGR91] se describen diversos métodos para el control de concurrencia en sistemas colaborativos sincrónicos. Son métodos específicos para esta clase de sistemas dado que los mismos incrementan los problemas de concurrencia debido a su naturaleza sincrónica.

A continuación se detallan algunos métodos para el control de concurrencia.

1- Bloqueo simple

Una solución al problema de concurrencia es simplemente bloquear los datos cuando éstos son efectivamente almacenados, para que ningún usuario pueda modificarlos en el momento del almacenamiento.

Sin embargo este método presenta tres problemas:

a- La sobrecarga de requerir y obtener el bloqueo, incluye tiempo de espera, si el dato ya fue bloqueado antes, lo cual causa una degradación en el tiempo de respuesta.

b- Acerca de la granularidad: por ejemplo cuando un usuario edita un texto no está claro que debe bloquearse si el usuario decide insertar un caracter en el medio de un párrafo. La pregunta es: ¿qué debe bloquearse, el párrafo entero, la palabra que se modificó o solo el caracter que se insertó?

c- Se necesita de un tiempo de bloqueo para solicitar el bloqueo y para liberarlo. Por ejemplo:¿Debe pedirse el bloqueo cuando se mueve el cursor en un editor de texto o cuando realmente se ejecutó la operación?

2- Unidad de control centralizada

Esta solución introduce una unidad de control centralizada. Esto implica que los datos son replicados sobre las estaciones de trabajo de todos los usuarios.

La unidad de control centralizada recibe los requerimientos de un usuario y hace un broadcast de estos requerimientos a todos los usuarios en el sistema.

Las mismas operaciones se realizan en el mismo orden para todos los usuarios, todas las copias de los datos son idénticas.

Esta solución introduce los problemas asociados con componentes centralizados, como la existencia de un único punto de fracaso, y el conocido problema del cuello de botella.

3- Detección de dependencia.

EL modelo de detección de dependencia es otro método para el control de concurrencia en sistemas multiusuarios. Este método utiliza operaciones timestamps para detectar acciones conflictivas, las cuales son resueltas manualmente.

La gran ventaja de este método es que no es necesaria la sincronización: las operaciones no conflictivas se ejecutan inmediatamente tan pronto sean recibidas y la respuesta es muy buena.

Sin embargo, cualquier método que exige la intervención del usuario para asegurar la integridad de los datos, es vulnerable a cualquier error del usuario.

4- Ejecución reversible

Las operaciones son ejecutadas inmediatamente, pero la información es guardada para que la operación pueda ser desecha mas tarde si es necesario.

Cuando una o más operaciones interfieren y se ejecutan concurrentemente, una o más de estas operaciones se deshace y se vuelve a ejecutar en el orden correcto.

Este método es muy responsivo. Sin embargo la necesidad de ordenar las operaciones globalmente es una desventaja.

5- Transformación de operaciones

Este método permite tener sistemas altamente responsivos.
Esta técnica es usada en el editor GROVE [EGR91]. Cada usuario tiene su propia copia del editor GROVE, y cuando se lleva a cabo una operación, ésta se ejecuta inmediatamente sobre la copia almacenada localmente. Luego, la operación se envía al resto de los usuarios en la sesión.

Existe un vector de estado donde se almacena cuántas operaciones fueron recientemente ejecutadas desde otras estaciones de trabajo.
Cada copia del editor tiene su propio vector de estado, el cual se compara con el resto de los vectores. Si tanto el vector local como el vector que se envía como broadcast son iguales entonces se ejecuta el requerimiento, de lo contrario se realizará una transformación antes de su ejecución.

Volviendo al ejemplo mencionado anteriormente sobre el documento "abc" citado en [LL02] se utiliza dicho método para garantizar consistencia de información. Después de que todas las operaciones fueron ejecutadas localmente, se propagan hacia el resto de los sitios. Si todas las operaciones remotas se ejecutaran como se mencionó antes, daría como resultado información inconsistente.
Dado que la ejecución de estas operaciones remotas puede llevarse a cabo cuando el estado inicial del documento sufrió algún cambio antes, es necesario utilizar algoritmos para transformación de operaciones antes de que dicha operación sea ejecutada.
Por ejemplo, cuando A recibe la operación remota de B ins('y',2), verifica si el contexto de esta operación ya ha sido cambiado por su operación local ins('x',1), con lo cual el estado inicial del documento cambió a "xabc", ahora la posición 2 en el que B deseaba insertar 'y', cambió por 3, así la operación de B ins('y',2) será transformada a ins('y',3). Luego el estado del documento en A después de la ejecución de la operación es "xaybc". Del mismo modo ocurre en B, cuando recibe la operación ins('x',1) de A, verifica si el documento cambió después de su operación local ins('y',2) por lo cual el estado del documento ahora es "aybc", sin embargo la posición 1 del documento no se modificó, aquí no es necesaria la transformación de la operación.

En conclusión, la transformación de operaciones es eficiente y mantiene consistencia si todos los usuarios comienzan su sesión con el mismo estado inicial del documento. Es decir, si un usuario se incorpora tarde a la sesión, el estado inicial del documento no será el mismo. En este caso, la transformación de operaciones no resuelve el problema, es decir no soporta el concepto de "latecomer" (sección 2.4.2)

Los anteriores son solo algunos de los métodos más importantes que hoy en día se utilizan para garantizar consistencia de información en ambientes colaborativos.

Sin embargo antes de determinar qué método utilizar, lo primero que se debe decidir es qué camino seguir frente a este problema, y aquí se plantean dos alternativas según [Hofte98]:

a- Permitir la presencia de inconsistencias temporales siendo tratadas posteriormente, ó
b- Prevenir la aparición de todo tipo de inconsistencia.

En primer lugar, permitir inconsistencias temporales puede ser una mejor opción que intentar prevenirlas.

Por ejemplo, permitir inconsistencias ocasionales en una pizarra compartida (Ej. La inconsistencia puede surgir en la intersección de dos líneas dibujadas simultáneamente por usuarios diferentes con colores diferentes), puede ser más deseable que prevenir las inconsistencias (Ej. Bloqueando los pixeles, o las regiones o la pizarra compartida en su totalidad), que inevitablemente deteriora el carácter responsivo del sistema.

Aún cuando la prevención de inconsistencia es la opción preferible, los diseñadores de aplicaciones groupware tienen muchas opciones. Por ejemplo, en lugar de negar o bloquear ciertas acciones (como se hace con el método de bloque simple), puede ser mejor posponer acciones de usuarios de manera que todas las acciones se ejecutan en todos los sitios en el mismo orden.

Resumiendo, el camino y el método que elijamos dependerá estrictamente del tipo de aplicación colaborativa que se quiera desarrollar. A partir de allí será tarea del diseñador elegir el camino correcto para asegurar la consistencia de información.

2.4.2 Latecoming

Todas las aplicaciones que usualmente son utilizadas por varios usuarios al mismo tiempo deben ser agrupadas en una sesión colaborativa. Si un nuevo usuario (latecomer) quiere incorporarse a la sesión de una aplicación colaborativa que actualmente esta siendo ejecutada por uno o varios usuarios, deben garantizarse dos tareas:

- El estado actual de la aplicación debe ser transferido al nuevo usuario (latecomer)

- Durante la transferencia del estado, ninguna aplicación de los otros usuarios puede cambiar su estado.

Según se define en [ITW01] un Latecomer (rezagado) es una persona que quiere asistir a una aplicación colaborativa que ya está siendo ejecutada por otros usuarios.

Entonces la primera tarea será transferir al nuevo usuario (latecomer) el estado de la aplicación al momento en que se incorpora a la sesión.

De acuerdo al patrón de diseño MVC [BMRSS96] una aplicación interactiva puede ser dividida en tres partes separadas: el modelo, la vista y el control.

Nuestro análisis estará centrado en cómo transmitir la información basada sólo en el modelo de la aplicación y el control correspondiente para llevar a cabo dicha tarea.

Para transferir el modelo de la aplicación, se debe capturar el estado actual del programa desde el momento que se inició la sesión colaborativa hasta el momento en que ingresó el nuevo usuario a la sesión (latecomer).

La segunda tarea es bloquear todas las aplicaciones colaborativas simultáneamente para evitar eventos de cambio de estado durante la transmisión. Los eventos de usuarios que suceden antes de que todas las aplicaciones tengan conocimiento de que deben ser bloqueadas, son almacenados en una cola de mensajes y deben ser enviados a todas las aplicaciones después de que finalice el proceso de latecoming.

Lo detallado anteriormente plantea un escenario ideal, sin embargo la realidad es muy distinta a esto.

Un caso específico de ejemplo es la herramienta de chat conocida como "MSN Messenger" o cualquier aplicación de mensajería instantánea. Estas herramientas permiten que dos o más personas puedan conversar simultáneamente en el mismo instante de tiempo. Una característica importante que ofrecen es la posibilidad de invitar a más de un usuario a una conversación. Y aquí se plantea el problema de latecoming. Un usuario que se incorpora tarde a la conversación solo podrá interiorizarse del tema a partir del momento en que se incorporó a la conversación, esto quiere decir, no tendrá manera de saber lo que los otros usuarios conversaron antes de su incorporación.

Según se pudo investigar [IM01], existen dos posibles direcciones para permitir que las aplicaciones colaborativas soporten la propiedad de latecoming:

- Hacer un playback (retroceso) de los eventos que ocurrieron antes de la incorporación del nuevo usuario a la sesión. Es decir, transferir la ocurrencia de eventos en el orden en el que ocurrieron hasta llegar al estado actual de la aplicación, ó

- Exportar sólo el estado final de la aplicación.

Ambos métodos tienen sus ventajas y desventajas.

El primero impone mínimos requerimientos sobre la aplicación y permite ver al nuevo usuario (latecomer) como se llegó al estado actual de la aplicación. Sin embargo esto puede requerir una cantidad excesiva de almacenamiento porque las sesiones pueden durar un largo tiempo.

La otra técnica requiere un almacenamiento mínimo y toma relativamente un tiempo breve para actualizar al usuario sobre la situación actual de la aplicación. Sin embargo el nuevo usuario no podrá ver como fue el desarrollo de los eventos hasta llegar a la situación actual del sistema. También la aplicación debe proveer un mecanismo para exportar su estructura de datos durante la sesión.

Volviendo al ejemplo anterior sobre herramientas de mensajería instantánea veamos cómo aplicar los métodos antes mencionados:
Por ejemplo, si decidiéramos implementar latecoming mediante el método de retroceso o playback deberíamos guardar cada conversación de cada usuario en el orden en que ocurrió. El nuevo usuario, antes de iniciar su conversación, recibirá todo el estado actual de la conversación que mantuvieron previamente los usuarios antes de su llegada.
Sin embargo sería complicado implementar el segundo método de exportar solo el estado final de la conversación, la cuestión es: ¿cuál será el estado final de la conversación?. No se puede hablar de un estado final en una conversación, porque en tal caso qué se informará al nuevo usuario, el último párrafo, la conversación en su totalidad, pero si fuese de ese modo, estaríamos implementando el método de retroceso.
Y aquí llegamos a la conclusión de que el método que se decida utilizar dependerá en gran medida de las características de la aplicación que se quiera desarrollar.

Además la propiedad de Latecoming debe proveer todos o algunos de los siguientes requisitos, algunos contradictorios:

- Elasticidad a los fallos individuales de los sitios.
- Transparencia a los desarrolladores y usuarios de una aplicación, comenzando por partes de propósito general de una infraestructura colaborativa.
- Alta interactividad necesaria para ofrecer a los latecomers un tiempo pequeño de actualización.
- Uso económico de recursos, particularmente de almacenamiento, y
- Flexibilidad en la inspección histórica de los eventos, desde la historia completa hasta el estado final.

En general implementar esta propiedad es una tarea difícil, y debería desarrollarse idealmente a través de una infraestructura de colaboración. Este es el método que se adopta en muchos sistemas actuales.

Los problemas planteados en las secciones 2.4.1 y 2.4.2 serán analizados en el capítulo 5 donde se intentará dar una solución a cada uno de ellos.

2.5 Conclusión

Los *Sistemas Colaborativos Sincrónicos* permiten a un grupo de usuarios distribuidos geográficamente, colaborar en una tarea común, en el mismo instante de tiempo. Se basan en la metodología y contribución de muchas disciplinas, en particular, hay al menos cinco perspectivas para la construcción de una aplicación colaborativa exitosa: Sistemas Distribuidos, Comunicación, Interacción humano-computador, Inteligencia Artificial y Teoría social. Estos cinco dominios de estudio no son independientes entre sí, ellos se combinan y benefician mutuamente.

Cada disciplina contribuye a un buen diseño y desarrollo de estos sistemas, sin embargo construir este tipo de aplicaciones no es una tarea sencilla, dado que los usuarios que colaboran en una misma sesión están distribuidos y de esta forma mucha información se pierde. Sin esta información, los grupos tienen dificultad para coordinar el control de acceso, comunicación y trabajo en general, por ello surge el concepto de Group Awareness, como un mecanismo que provee un contexto grupal actualizado y notificaciones de las acciones de cada usuario cuando sea apropiado en un espacio de trabajo compartido.

Muchos trabajos en el área de los sistemas colaborativos se han centrado en la resolución de problemas puntuales. Como resultado se concluyó que hay una serie de requisitos comunes que debe satisfacer toda aplicación colaborativa y estos requisitos son los relacionados con el estilo de colaboración y pueden ser provistos por una plataforma.

La ventaja de contar con una plataforma es que disminuye el tiempo y complejidad de desarrollo de una aplicación colaborativa. Tales requisitos determinan el tipo de arquitectura de cómputo sobre la cual se apoyan los sistemas colaborativos.

Puntualmente se analizaron cuatro de estos requisitos: (1) tamaño del grupo, (2) grado de acoplamiento de las interfaces, (3) tiempo de respuesta corto y (4) la disponibilidad del sistema.
De acuerdo con esto, una arquitectura de cómputo colaborativa puede ser centralizada, replicada o híbrida.
En una arquitectura replicada existe una réplica de los datos en cada estación de trabajo. Esta solución obliga a enfrentar problemas de sincronización e inconsistencia de datos, pero podría proveer tiempos de respuesta más cortos; en una arquitectura centralizada nos enfrentamos a los problemas de cuello de botella o un único punto de fracaso si la componente central falla.
Como vimos tanto las arquitecturas centralizadas como las replicadas ofrecen beneficios y limitaciones, una alternativa frecuente es la arquitectura híbrida, entre centralizada y replicada, combinando las ventajas de ambas.

Sin embargo, cualquiera sea el tipo de arquitectura en el que se base el desarrollo de nuestra aplicación colaborativa se debe garantizar la consistencia de los datos, sin olvidarnos de proveer un tiempo de respuesta corto.
La fundamental causa de inconsistencia en ambientes colaborativos sincrónicos es que las acciones que realizan los usuarios, se llevan a cabo en el mismo

instante de tiempo, pero debido al tráfico de información en la red, tales acciones se procesan en diferente orden, y es allí donde se producen las inconsistencias.
Frente a este problema se plantearon dos alternativas: permitir la presencia de inconsistencias temporales ó, prevenir la aparición de todo tipo de inconsistencias.
Independientemente del camino que se decida utilizar, todos los sistemas colaborativos necesitan de un control de concurrencia para resolver estos conflictos.
Se analizaron varios métodos que intentan dar solución a dicho problema. Entre ellos se encuentran el **bloqueo simple (locking), unidad de control centralizada, detección de dependencia, ejecución reversible y tranformación de operaciones.** Cada método plantea ventajas y desventajas. La utilización de cada uno dependerá en gran medida del sistema a desarrollar.

Por otra parte, también se habló de la necesidad, cada vez mayor, de permitir la incorporación tardía de un usuario *(latecomer)* a la sesión sin interferir en el trabajo de los otros usuarios en la aplicación, lo que se conoce como *"latecoming"*.
Según se pudo investigar, existen dos posibles direcciones para permitir que las aplicaciones colaborativas soporten esta propiedad: hacer un playback (retroceso) de los eventos que ocurrieron antes de la incorporación del nuevo usuario a la sesión ó, exportar sólo el estado final de la aplicación.

Cualquier método que se decida implementar tendrá sus ventajas y desventajas pero ambos deben cumplir con los siguientes requisitos: elasticidad a los fallos individuales de los sitios, transparencia a los desarrolladores y usuarios de una aplicación, alta interactividad, uso económico de recursos, y flexibilidad en la inspección histórica de los eventos.
En general implementar esta propiedad es una tarea difícil, y debería desarrollarse idealmente a través de una infraestructura de colaboración.

Resumiendo, se deben contemplar varios puntos a la hora de desarrollar un sistema colaborativo sincrónico: en primer lugar se debe decidir el tipo de arquitectura (centralizada, replicada o híbrida) que se va a utilizar, definir un método de control de concurrencia para garantizar consistencia de información y adaptar dicho método para que el sistema permita la incorporación tardía de usuarios en la sesión, o lo que es lo mismo que ofrezca soporte a la propiedad de latecoming.

Actualmente hay muchas aplicaciones que garantizan consistencia, otras que dan soporte a la propiedad de latecoming, sin embargo es difícil encontrar hoy en día una aplicación que reúna todas estas características.

Es por ello que resulta imprescindible encontrar una solución a los problemas de consistencia y latecoming, para que el desarrollo de las aplicaciones colaborativas sincrónicas no esté limitado a formas particulares de trabajo colaborativo.

Capítulo 3

Arquitecturas

La arquitectura de una aplicación de software caracteriza los componentes de la aplicación, la función llevada a cabo por cada componente, y la interacción entre estos componentes [SG96, KBAW94].

Para definir específicamente la arquitectura de una aplicación colaborativa, debemos conocer la forma de colaboración de los distintos usuarios, es decir, debemos preguntarnos por ejemplo qué componentes de la aplicación pueden ejecutar concurrentemente, qué componentes deben replicarse, cuáles componentes deben en todo momento ser consistentes entre todos los usuarios, etc. [RU00]

Es decir, primero debemos definir claramente el tipo de aplicación colaborativa a diseñar, luego describir la arquitectura de colaboración que va a llevar a cabo dicha aplicación.

Si la aplicación a diseñar es una aplicación colaborativa sincrónica, debemos recordar que el objetivo de la misma es reunir a usuarios que se encuentran distribuidos geográficamente y conectados a través de una red. Los usuarios a la hora de trabajar manipulan información compartida, y cualquier modificación realizada por uno de ellos debe ser contemplada inmediatamente por el resto de los usuarios (consistencia de información) [BAK95] .

Podemos dividir la etapa de diseño de una aplicación colaborativa en dos partes:

- Diseñar un esquema para aplicación colaborativa, en el cual se van a definir los distintos componentes que integrarán la aplicación
- Diseñar un esquema para la distribución de la aplicación colaborativa, en el cual se especificará como se van a distribuir los componentes de la aplicación a través de la red, se van a contemplar problemas relacionados con las diferentes plataformas utilizadas por los usuarios, algoritmos de sincronización, control de concurrencia de los usuarios, mejoras en el tiempo de respuesta de la aplicación, tolerancias a fallas, etc

Una vez definidos ambos esquemas, vamos a describir el modelo de arquitectura a utilizar. A continuación se mencionan todos los modelos. [Patterson94]

- Arquitectura centralizada de componentes
- Arquitectura replicada/distribuida (múltiples servers)
- Arquitectura híbrida
- Arquitectura con comunicación directa
- Arquitectura con estructura asimétrica

Cabe destacar que la mayoría de las aplicaciones colaborativas diseñadas actualmente se basan en modelos de arquitectura centralizada o replicada.

Pero independientemente del modelo seleccionado a la hora de diseñar una aplicación, no debemos dejar de tener en cuenta que una aplicación colaborativa sincrónica debe satisfacer:

- Requerimientos de los usuarios
  - ✓ Procesamiento de las acciones de los propios usuarios sin pérdida de tiempo.
  - ✓ Rápida propagación de las acciones de los otros usuarios
  - ✓ Se deben soportar sesiones dinámicas
- Punto de vista del desarrollador
  - ✓ Los problemas crecen a medida que el número de usuarios que interactúan incrementa en una aplicación colaborativa
  - ✓ Objetivo abstracto: reducir la complejidad de aplicaciones colaborativas hasta que sea comparable con la complejidad de aplicaciones single-user.

Entonces a la hora de seleccionar un modelo de arquitectura debemos analizar cómo se adapta el mismo a cada uno de los requerimientos de los usuarios y a los requerimientos del desarrollador.

3.1 Modelos de Arquitecturas Colaborativas

3.1.1 Arquitectura Centralizada

Este tipo de arquitectura se caracteriza por la existencia de un nodo central, cuya función será la de atender los requerimientos que realicen los usuarios.

Cuando definimos una aplicación colaborativa basada en este modelo de arquitectura, existirá una única instancia de la aplicación la cual residirá en el nodo central. (Figura 3.1)

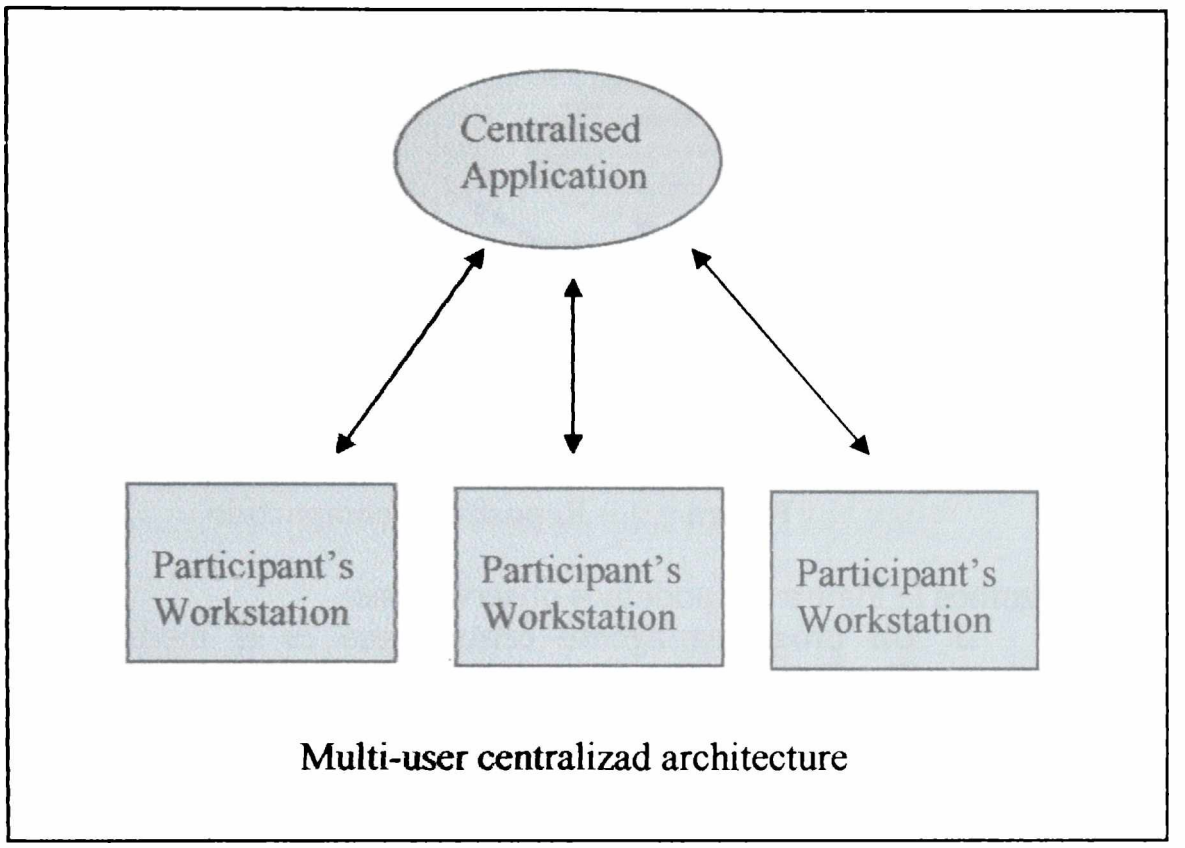


Figura 3.1 – Arquitectura centralizada

En caso de querer trasformar una aplicación single-user en una aplicación colaborativa basada en un modelo de arquitectura centralizado, la misma no necesitará gran cantidad de cambios, ya que si bien se va a trabajar con varios usuarios, solo uno va a estar accediendo a un determinado componente de la aplicación en un instante de tiempo. Para que esto suceda, los requerimientos de los usuarios deberán ser serializados y filtrados.

Podemos entonces garantizar consistencia de información, debido a que si un usuario realiza alguna modificación sobre un componente los demás usuarios que participan en la sesión de grupo observarán dicho cambio.

Una aplicación con repositorio compartido (Figura 3.2) es un ejemplo típico de este tipo arquitectura.

Tenemos varios usuarios que se encuentran interactuando con una base de datos (Repositorio Compartido) al mismo tiempo, y la información siempre se mantiene consistente ya que se encuentra centralizada en un solo lugar, cualquier cambio realizado por alguno de los usuarios en la base será contemplado por el resto de los usuarios.

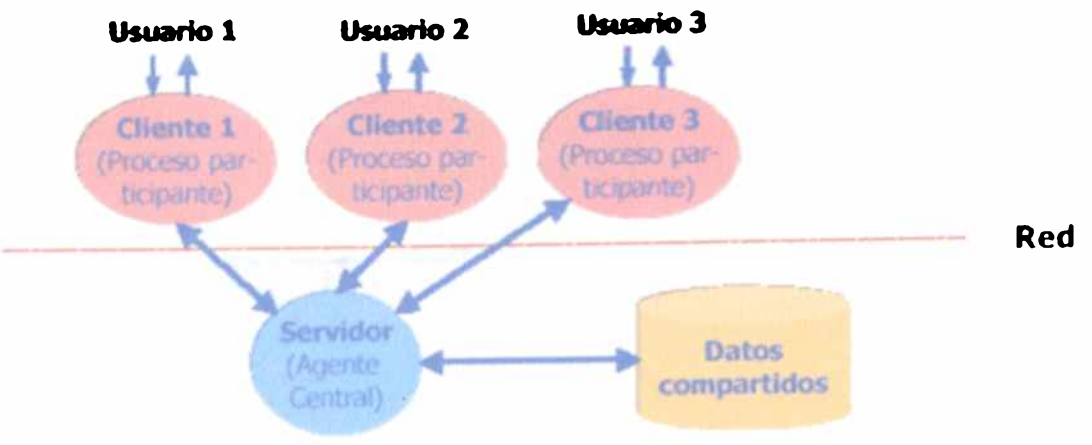


Figura 3.2 – Repositorio compartido

Si analizamos la Figura 3.2 podemos observar que:

- Un programa agente central que es el mediador del trabajo distribuido realizado por los usuarios.
- Cada usuario ejecuta un proceso participante que recoge las entradas de los usuarios y las envía al agente central.
- El agente comunica qué cambios tiene que mostrar en su pantalla.

Si bien en esta aplicación contamos con la ventaja de que todos los usuarios tienen una vista consistente de los recursos compartidos, tenemos como desventajas [GT00]:

Coordinación del uso de recursos: si un usuario intenta realizar alguna operación que modifique un recurso, solo podrá efectuarla si dicho recurso se encuentra disponible. En tal caso el usuario tendrá un uso exclusivo del recurso, el cual se encontrará bloqueado para cualquier solicitud de modificación proveniente del resto de los usuarios. Una vez finalizada la o las operaciones que el usuario desea realizar, el recurso es desbloqueado, y nuevamente estará disponible para la solicitud de cualquier usuario.

Sobrecarga de la red: todos los usuarios interactúan con la misma base de datos que se encuentra ubicada en un nodo central, cada operación que los mismos quieran realizar se traduce en una solicitud a dicho nodo central, esto genera una gran sobrecarga de la red ya que todos los requerimientos están dirigidos a un único nodo.

Falla del nodo central: toda la información compartida se encuentra en el nodo central, cada usuario a la hora de realizar una operación requiere información que se encuentra almacenada en el nodo central. En caso de que ocurra alguna falla en el nodo central ningún usuario va a poder seguir realizando sus actividades de forma normal.

3.1.2 Arquitectura Replicada

Esta arquitectura se caracteriza por la existencia de una réplica de la aplicación colaborativa en cada uno de los nodos de los usuarios, así como también una réplica de los datos en cada nodo. (Figura 3.3)

Cabe destacar que los datos que se replican a cada nodo, se encuentran almacenados en algún nodo para así garantizar que la réplica que se envía a todos al inicio de la aplicación es la misma, garantizando de tal manera una consistencia entre los datos que son replicados.

Cada nodo será encargado de mantener la integridad de los datos, así como también de comunicar al resto de los cambios que realice [LJLR90].

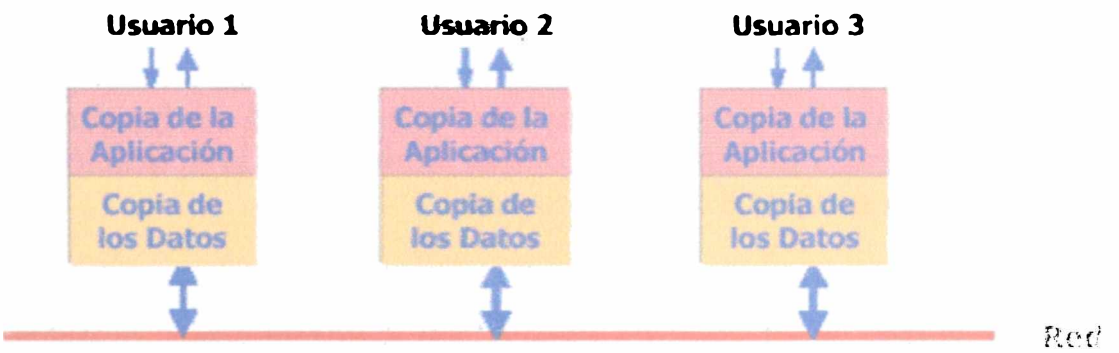


Figura 3.3 – Arquitectura replicada

Las aplicaciones replicadas mejoran los tiempos de respuestas, debido a que cada usuario trabaja en la aplicación en forma local, por lo tanto todas las operaciones son ejecutadas localmente en lugar de ser propagadas a un nodo central.

Si bien esto reduce la sobrecarga de la red, en caso de que un usuario realice una modificación sobre algún componente, debe propagarla al resto de los usuarios, contemplando que la propagación no cause una excesiva carga de la red [GT00, HBRPW94].

Como mencionamos en el párrafo anterior, este tipo de arquitectura reduce la sobrecarga de la red, pero trae aparejados problemas que en una arquitectura centralizada no debían ser contemplados, como por ejemplo [PS94]:

> **Diferentes estados iniciales para las réplicas de la aplicación:** Al inicio de una sesión de grupo, las réplicas individuales que se encuentran en los diferentes nodos empiezan a menudo con estados iniciales diferentes.
> Esto se debe a que los archivos requeridos para poder comenzar a utilizar la aplicación están disponibles en un solo nodo, por lo cual, cuando un participante quiere iniciar su sesión debe copiarse estos archivos a su nodo. En este caso pueden generarse conflictos si existe al menos un usuario que ya tiene su réplica de la aplicación y está listo para efectuar alguna

modificación, mientras que otro usuario todavía está esperando para iniciar su aplicación, en cuyo caso, la copia de los archivos que el mismo realice se va a encontrar desactualizada, respecto a la versión del o los usuarios que ya iniciaron su sesión.

Una posible solución a este problema es retardar toda operación que cualquier usuario quiera realizar hasta poder garantizar que ya todos los usuarios del grupo tienen su réplica de la aplicación en su nodo, y así garantizar que todas las réplicas inicialmente eran idénticas.

Comportamiento determinístico: si garantizamos que el estado inicial de cada réplica de la aplicación es idéntico, y contamos con igual secuencia de entrada en varios usuarios, debemos obtener un mismo estado final, lo cual algunas veces no ocurre debido a la influencia del contexto y parámetros temporales que influyen en el comportamiento de la aplicación.

Recordemos que los usuarios además de encontrarse distribuidos geográficamente, cada uno de ellos puede estar trabajando en una plataforma diferente, en cuyo caso, debe contemplarse que ninguna configuración en las distintas plataformas modifique el comportamiento de la aplicación frente a una determinada secuencia de operaciones.

Estados idénticos: lograr que todos los usuarios tengan una réplica idéntica de la aplicación suele ser muy difícil, debido a que la mayoría de las aplicaciones permiten a los usuarios modificar o adaptar la aplicación localmente según sus gustos particulares, permitiendo por ejemplo refinar sus propios menúes y definir las combinaciones necesarias para acceder a diferentes comandos. Por lo cual en realidad contamos con diversas versiones de la aplicación, y mantener una versión de la configuración de cada usuario no siempre es posible, ya que en caso de alguna falla no se puede garantizar que cada usuario vuelva a tener la aplicación con su configuración personal.

Orden de los eventos de entrada: en una aplicación colaborativa debemos mantener la consistencia después de la realización de una operación o requerimiento de un usuario, en caso de que éste haya modificado el estado de algún componente de la aplicación debemos reflejar este cambio en cada una de las réplicas para así mantener la consistencia.
Como pueden ser varios usuarios los que estén realizando cambios, las operaciones realizadas por los mismos deben aplicarse en cada réplica manteniendo el orden en el cual fueron realizadas, de lo contrario podríamos obtener inconsistencias entre las distintas réplicas.

Miembros de la sesión: El número de usuarios que se encuentran en una sesión no es fijo, durante una sesión un usuario puede ser incorporado o algunos pueden abandonar la sesión. En caso de que un usuario sea incorporado su estado inicial no va a ser igual al estado inicial del resto de los usuarios, sino que se le debe notificar de todos los cambios que se fueron realizando (latecomming), cada usuario perteneciente al grupo debe notificar al nuevo usuario qué operaciones realizó, las cuales modificaron el estado de la réplica, y el nuevo usuario deberá aplicar los cambios realizados por los distintos usuarios en el orden en que se fueron realizando, y así tendrá su réplica consistente con el estado de las réplicas de los demás usuarios.

Un ejemplo de una aplicación con arquitectura replicada, es un sistema manejador o administrador de conferencias (Figura 3.4). Los distintos nodos van a tener una réplica de la aplicación y existirá un nodo en el cual se encuentre el administrador, quien coordinará la conferencia y permitirá que al incorporarse nuevos usuarios se les notifique el estado y los cambios realizados.

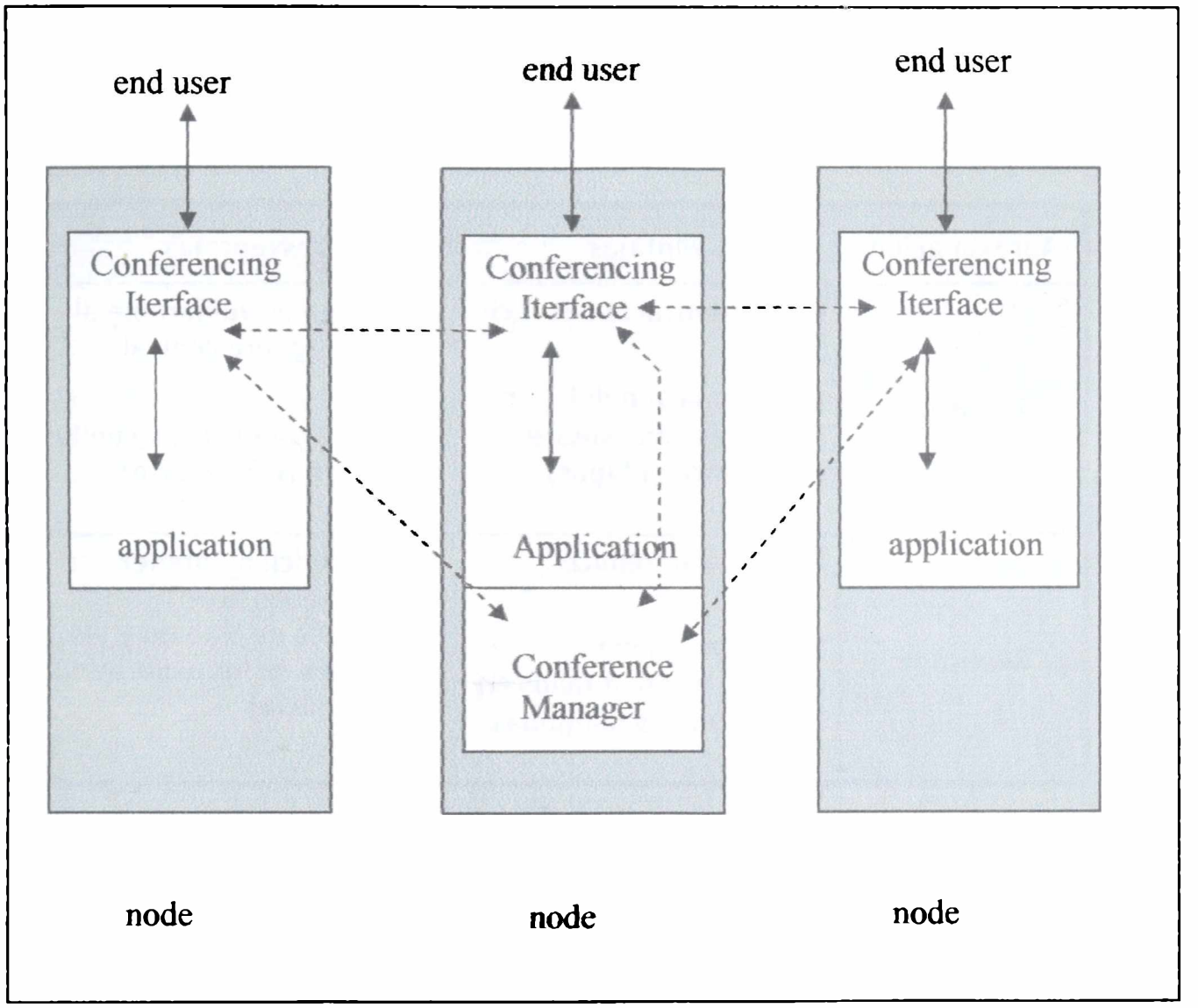


Figura 3.4 – Sistema Manejador o Administrador de conferencias

Cabe destacar que en cualquier ejemplo de aplicación que se base en un modelo de arquitectura replicada, va a existir un nodo el cual tendrá como tarea al comienzo de la utilización de la aplicación garantizar que las réplicas sean iguales, como así también será el encargado de mantener la información que va a ser replicada.

En el ejemplo planteado, los usuarios tienen una réplica local de la aplicación la cual permite realizar la conferencia, pero cada vez que quieran remitir algo a otro usuario, lo harán de manera replicada. Todas las réplicas deben ser idénticas

3.1.3. Arquitectura Híbrida/Semi-Replicada

Este tercer esquema de arquitectura, se deriva de los dos mencionados anteriormente, tratando de mantener las ventajas y superar desventajas que se presentan en cada uno de los esquemas anteriores.

Por lo mencionado en 3.1.2 la principal ventaja de la arquitectura replicada es que la interacción entre el usuario y la aplicación se realiza en forma local, disminuyendo así la carga de la red excesiva que se generaba con una arquitectura centralizada. Sin embargo una arquitectura replicada es difícil de sincronizar, debe existir una consistencia entre los estados de todas las aplicaciones que se replicaron en los diferentes nodos de la red, consistencia que garantizaba una arquitectura centralizada.

Podemos entonces sintetizar las ventajas y desventajas de los dos esquemas anteriores en el siguiente cuadro Figura 3.5:

Aproximación	Ventajas	Desventajas
Centralizada	La sincronización es fácil La información del estado es consistente (está en cada lugar)	El sistema es vulnerable al fallo del agente central Se puede producir un cuello de botella (tráfico alto)
Replicada	El tráfico se reduce El sistema es más robusto frente a fallos en la red y en las máquinas	Es más difícil mantener sincronizadas las superficies de trabajo y las peticiones de los usuarios (consistencia)

Figura 3.5 – Ventajas/Desventajas de Arquitecturas centralizada y replicada

Una arquitectura híbrida plantea entonces permitir que las aplicaciones de los participantes usen un servidor central que se va a ocupar de mantener todos los recursos compartidos (mantener consistencia). La interacción entre un usuario y un recurso se realizará en forma local, para luego replicarse los cambios al servidor central (disminuir sobre carga de red).

A modo de ejemplo supongamos que queremos trabajar con una aplicación colaborativa sincrónica basada en MVC (Modelo-Vista-Controlador ó Model-View-Controller respectivamente[1]) (Figura 3.6). Vamos a plantear cómo sería su diseño en un esquema de arquitectura centralizado (Figura 3.7), en un esquema replicado (Figura 3.8) y finalmente en uno híbrido (Figura 3.9).

[1] El concepto **Model-View-Controller** es traducido al idioma Español para facilitar la comprensión a a aquellos lectores que no estén familiriazos con el idioma Inglés. En el capítulo 5, estos conceptos serán mencionados como Model-View-Controller respectivamente dado que los mismos son comúnmente conocidos de esta manera en la terminología Informática.

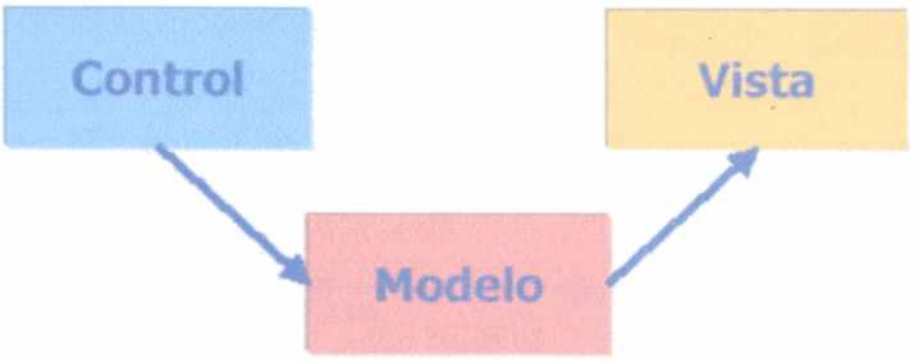


Figura 3.6 – Esquema Modelo-Vista-Controlador

Primero recordemos cuál sería la función de cada componente del MVC.

Modelo: representación de un modelo sistemático del problema
Vista: visualización del modelo mediante alguna representación visual
Controlador: permite al usuario o al entorno modificar el estado del modelo

En un modelo de arquitectura centralizada vamos a tener el Modelo centralizado en un nodo. (Figura 3.7).

Los demás nodos tendrán localmente su vista y su controlador, pero cualquier modificación que quieran realizar sobre la información será solicitada al nodo central.

La comunicación de un nodo con el nodo central se realiza siempre mediante la Vista y el Controlador, nunca a través del Modelo.

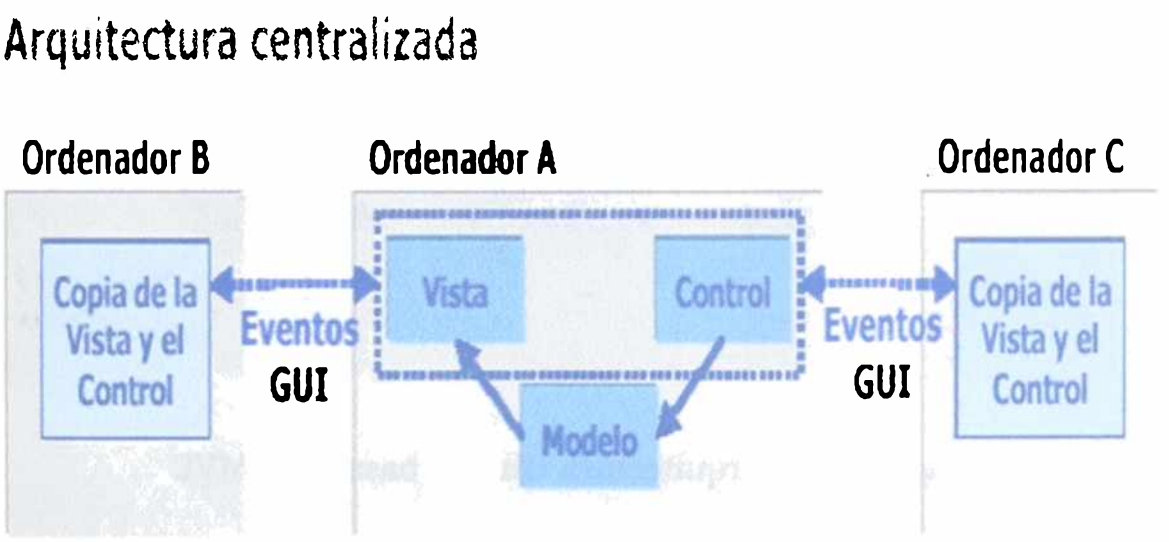


Figura 3.7 – Arquitectura Centralizada basada en MVC

En un modelo de arquitectura replicada (Figura 3.8) cada nodo de la red tendrá una réplica completa del MVC, y entre ellos se comunicarán para mantener sincronizadas las réplicas mediante los controladores luego de la realización de alguna modificación

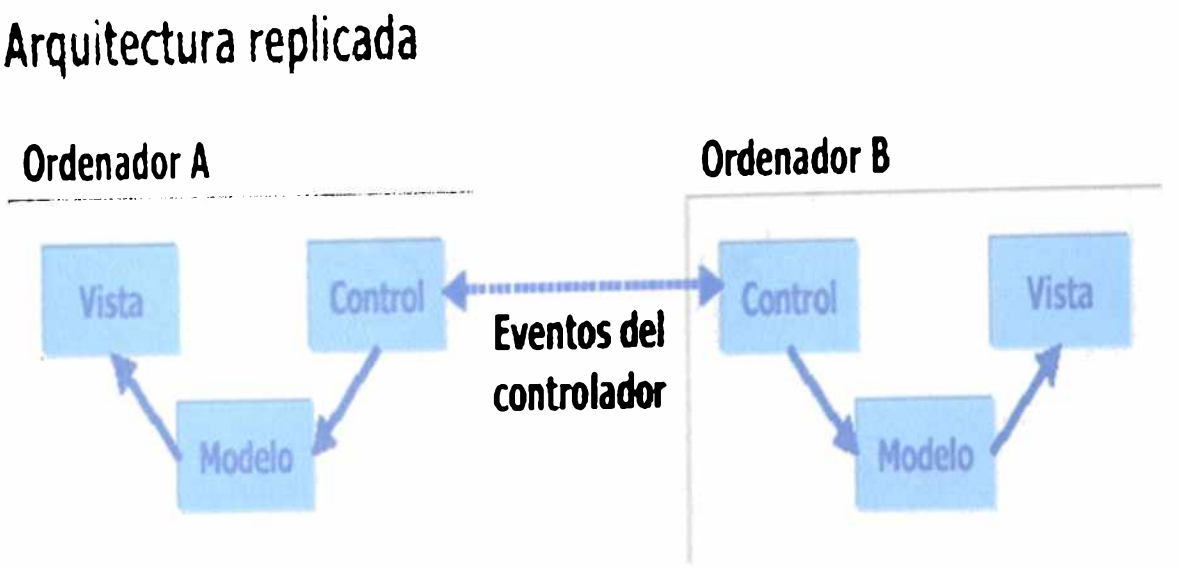


Figura 3.8 – Arquitectura Replicada basada en MVC

Finalmente en un modelo de arquitectura híbrida (Figura 3.9) tendríamos que cada nodo tiene una réplica del MVC, pero existe un nodo central en el cual se encuentra centralizado el Modelo y con el cual todos los demás nodos sincronizan para mantener así el modelo consistente entre todos los usuarios.

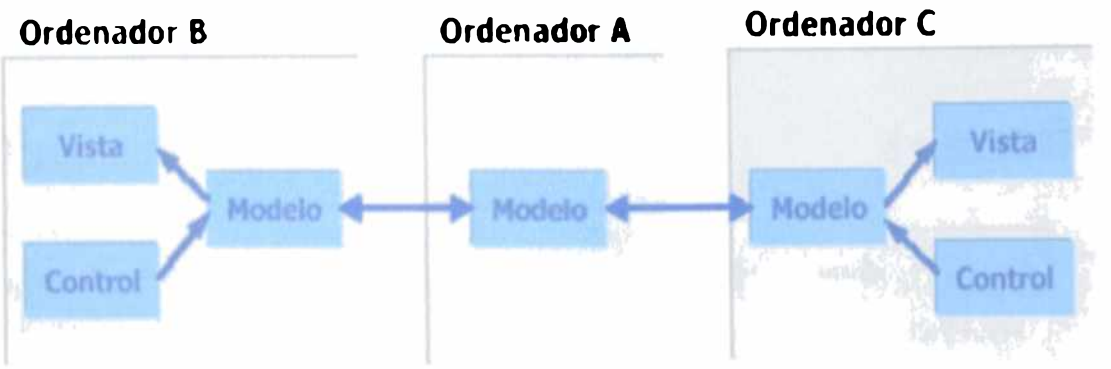


Figura 3.9 – Arquitectura Híbrida basada en MVC

3.1.4 Arquitectura con Comunicación Directa

En este modelo de arquitectura no existe un nodo central que se encargue de atender los requerimientos de los usuarios. Todos los nodos de la red se encuentran conectados y tienen localmente una copia de la aplicación, de manera

tal, que si un usuario realiza una modificación en un componente, debe notificarla al resto de los usuarios [Dewan00].

Por lo mencionado anteriormente, podríamos llegar a pensar que se trata de una arquitectura replicada, pero sin embargo, no contempla ninguna de las consideraciones necesarias para una arquitectura replicada, las cuales fueron mencionadas en 3.1.2.1, por lo cual podemos citar como ejemplo que se diferencia de una arquitectura replicada porque no permite la incorporación de nuevos usuarios, ya que la información necesaria para poder iniciar un nuevo nodo con igual estado inicial que cualquier otro nodo de la red, no se encuentra almacenada en ningún sitio.

En este tipo de arquitectura la comunicación entre los distintos nodos se realiza mediante pasaje de mensajes, un nodo realiza una modificación sobre algún componente y lo comunica al resto de los nodos. Cada nodo al recibir una notificación de modificación proveniente de algún usuario, realiza localmente la misma modificación.

Si bien este tipo de arquitectura es muy simple de implementar, presenta como desventajas: sobrecarga de la red debido a las notificaciones que se realizan a todos los usuarios, difícil manejo de consistencias, ya que no existe un nodo central en el cual se encuentre agrupada toda la información, y cuando un nodo recibe varias notificaciones provenientes de diferentes usuarios se debe poder asegurar que las mismas se realizan en algún orden al que fueron enviadas, ya que de no ser así podríamos tener que la información almacenada entre los distintos nodos es inconsistente.

Podríamos ver este tipo de arquitectura como un acercamiento a una arquitectura replicada, es decir, mejora las desventajas de una arquitectura centralizada, pero no contempla todos las ventajas de una replicada, por tal motivo este tipo de arquitectura en la práctica no es utilizado para el diseño de aplicaciones colaborativas sincrónicas.

Gráficamente podemos representar este tipo de arquitectura como en la Figura 3.10, en la cual se muestra la arquitectura de una aplicación compartida con pasajes de mensajes.

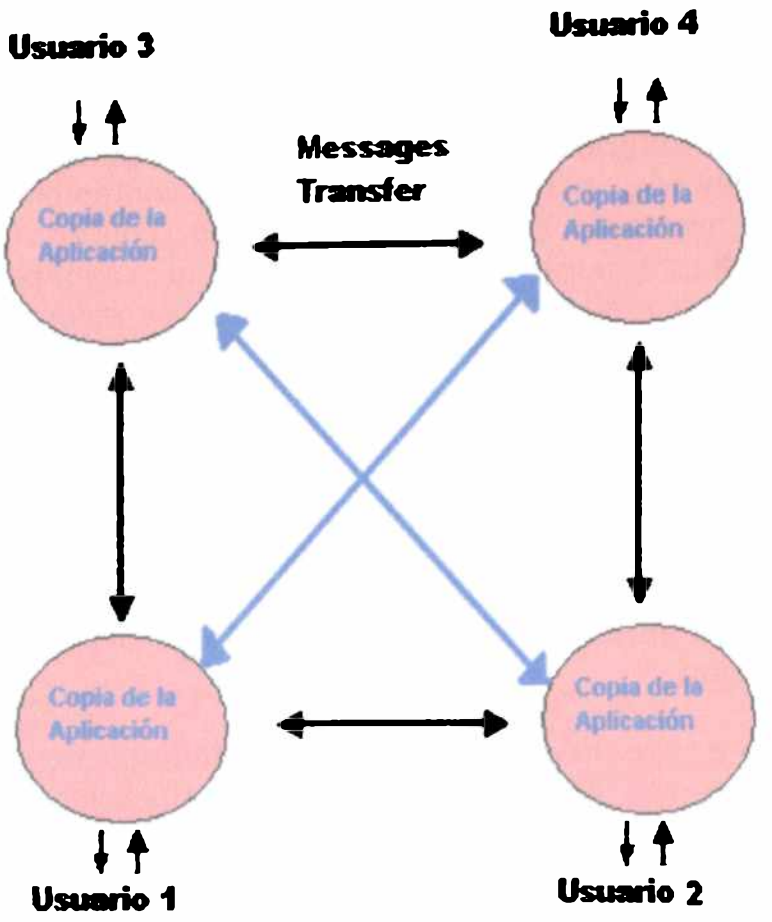


Figura 3.10 – Arquitectura con pasaje de Mensajes

3.1.5 Arquitectura con estructura asimétrica

Este modelo es similar al modelo del inciso anterior (3.1.4), salvo que no todos los nodos poseen una réplica completa de la aplicación (Figura 3.11) [Dewan00].

Existen componentes que son replicados en algunos nodos y en otros no, es decir la replicación es asimétrica, debe existir al menos un componente, el cual fue replicado en todos los nodos. En la Figura 3.11 se representa esto con las diferentes figuras que se encuentran dentro de cada nodo, notar que solo una se encuentra en todos los nodos.

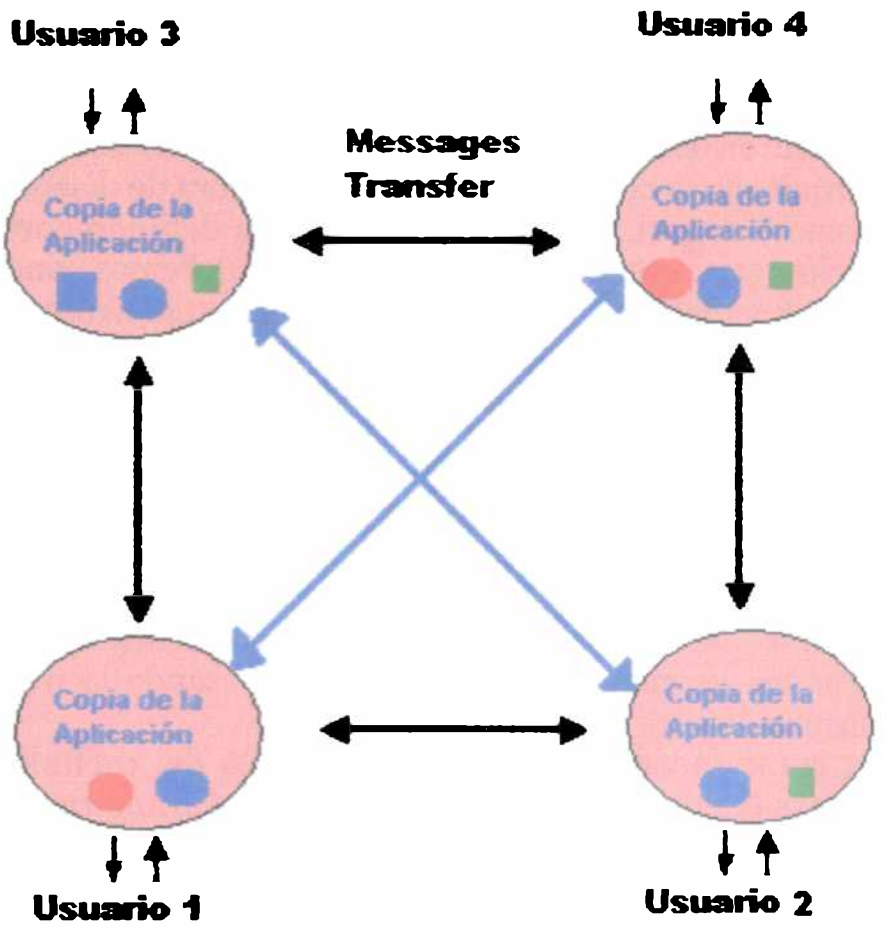


Figura 3.11 – Arquitectura con Estructura Asimétrica

En caso de que un usuario realice un cambio sobre una componente que no se encuentra replicado en todos los nodos, el mismo realizará más notificaciones que las necesarias provocando una sobrecarga de la red.

Al igual que en el modelo anterior no se contemplan las consideraciones necesarias para una arquitectura replicada (ver 3.1.2.1), y en realidad en este modelo tiene una sobrecarga mayor que en el anterior.
Además presenta como inconveniente, que si bien podría llegar a pensarse que es útil en algunos casos la réplica parcial entre los nodos, la misma debería estar acompañada con algún mecanismo que permita a los mismos saber a qué nodo deben comunicarle los cambios de los componentes modificados. Sin embargo esto no es contemplado, por lo cual la mayor desventaja de este esquema es la sobrecarga innecesaria de la red producida por la falta de conocimiento de a qué o cuáles nodos se deben notificar en caso de realizar alguna modificación sobre un componente.

Cabe destacar que tanto este esquema de arquitectura como el anterior no son utilizados actualmente en aplicaciones colaborativas debido a que presentan mas desventajas que ventajas, por tal motivo la mayoría de las aplicaciones colaborativas actuales basan sus diseños en alguno de los tres primeros modelos planteados, es decir, centralizado, replicado o híbrido.

3.2 Conclusión

Por todo lo visto en este capítulo concluimos, que no se puede afirmar que esquema de arquitectura nos conviene utilizar a la hora de diseñar una aplicación colaborativa sincrónica, si no tenemos claramente definido con anterioridad el tipo de aplicación colaborativa a diseñar, es decir, debemos conocer la forma de colaboración de los distintos usuarios, saber qué componentes de la aplicación pueden ejecutar concurrentemente, qué componentes deben replicarse, qué componentes deben en todo momento ser consistentes entre todos los usuarios.

Si bien planteamos la existencia de cinco esquemas de arquitecturas: arquitectura centralizada de componentes, arquitectura replicada/distribuida (múltiples servers), arquitectura híbrida, arquitectura con comunicación directa, arquitectura con estructura asimétrica, pudimos observar que, todas las discusiones planteadas por los diferentes autores, buscan una respuesta real sobre qué esquema es mejor pero casi siempre basando la discusión entre un esquema centralizado o un esquema replicado.
Esto se debe, como mencionamos en el desarrollo del capítulo, a que los dos últimos esquemas (comunicación directa/ estructura asimétrica) podemos verlos como acercamientos a tratar de superar las desventajas de una arquitectura centralizada, pero sin llegar a contemplar en ambos casos las ventajas que nos ofrece una arquitectura replicada.

También observamos, que si bien la mayoría afirma que lo ideal sería el uso de una arquitectura híbrida, muy pocas aplicaciones actuales se basan en este tipo de esquema dada su complejidad.

A la hora de plantear entre si conviene la elección de un esquema centralizado o un esquema replicado, la mayoría de los autores ponen el esfuerzo de diferenciación en el modo en el que el esquema facilita la sincronización, la sobrecarga de la red, los tiempos de latencias, la facilidad de incorporación de nuevos usuarios, etc. Pero rara vez consideraron los grandes problemas de inconsistencias que se pueden llegar a tener si el esquema seleccionado es el incorrecto, y éste será nuestro objetivo, es decir, diseñar un esquema de arquitectura que no solo contemple las ventajas y desventajas de los esquemas centralizado y replicado, sino centrar nuestra atención en cómo el esquema de arquitectura planteado puede solucionar los problemas de inconsistencias.

Vamos entonces a definir un esquema de arquitectura híbrido considerando, como mencionamos en este capítulo, que una arquitectura híbrida garantiza que las aplicaciones de los participantes usen un servidor central que se va a ocupar de mantener todos los recursos compartidos (mantener consistencia). La interacción entre un usuario y un recurso se realizará en forma local, para luego replicar los cambios al servidor central (disminuir sobrecarga de red).

Si bien podríamos decir que nuestro esquema a definir va a ser un esquema híbrido, veremos posteriormente en el capítulo 5, que el esquema finalmente propuesto no cumple estrictamente con lo especificado en este capítulo para una arquitectura híbrida, dado que se plantea un manejo de información en forma centralizada y replicada permitiendo tener la menor inconsistencia de datos

posible. La mayor diferencia radica en que no hay replicación total de datos sino que algunos serán tratados de manera centralizada.

Capítulo 4

Casos de Estudio: Frameworks y Arquitecturas

4.1. Introducción

En este capítulo se presentará el análisis de dos frameworks para el desarrollo de componentes colaborativas sincrónicas: ellos son DyCe (IPSI) y COAST (IPSI).
El objetivo de este análisis será discutir qué arquitectura de las estudiadas en el capítulo 3 implementa cada framework, y qué metodologías de control de concurrencia utilizan para garantizar consistencia de información.

También se detallará el resultado de las pruebas realizadas sobre algunas herramientas concretas desarrolladas con cada uno de ellos, centrando nuestro estudio específicamente en el manejo de consistencia y latecoming, lo que nos guiará para establecer nuestro modelo de arquitectura basándonos en sus ventajas y en mejorar las desventajas de cada modelo estudiado.
Además, se expondrá el análisis de un servidor de aplicaciones colaborativas, conocido como Servidor Sametime de la empresa IBM desarrollado por Lotus, con el objetivo de presentar un modelo de arquitectura óptimo para el manejo de comunicaciones en cualquier aplicación colaborativa sincrónica independientemente del modelo de arquitectura que implemente la aplicación a desplegar.

Se recomienda al lector, que no esté familiarizado con el concepto de *Framework*, remitirse al Apéndice C antes de comenzar la lectura de este capítulo.

4.2. Framework COAST

4.2.1 Introducción

Las aplicaciones colaborativas permiten, a diversos usuarios distribuidos geográficamente, poder trabajar juntos en un ambiente computarizado.
En cualquier aplicación colaborativa existen una clase de requerimientos básicos que se deben contemplar. En el caso de una aplicación colaborativa sincrónica, existen tres requerimientos abstractos que pueden ser declarados desde el punto de vista de los usuarios.

- Puesto que las interfaces de usuario de manipulación directa requieren regeneración rápida, las aplicaciones groupware sincrónicas deben asegurar

el no-retardo en el procesamiento de las acciones del usuario independiente de la conexión de la red.

- Las aplicaciones sincrónicas requieren por definición una rápida propagación de las operaciones del usuario para habilitar conocimiento de grupo óptimo bajo las condiciones de red dadas.

- Dado que las diferentes situaciones cooperativas requieren de diferentes aspectos de un espacio de trabajo compartido para poder ser acoplado a los diferentes usuarios concurrentes, las aplicaciones sincrónicas necesitan de un soporte para el comportamiento dinámico de la sesión.

Una herramienta para el desarrollo de aplicaciones colaborativas debe ofrecer las siguientes características:

- Una arquitectura general para aplicaciones colaborativas que sea adaptable a diferentes situaciones o tareas a través de un camino flexible.

- Proveer bloques básicos ya construidos, y componentes genéricas, los cuales puedan ser re-usados y refinados para diferentes situaciones concretas.

- Proveer una metodología con reglas y convenciones acerca de cómo las funcionalidades típicas de cada aplicación pueden ser desarrolladas en términos de la herramienta.

A través del uso de tales herramientas, puede esperarse un código con muchos menos errores, y puede tender a ser más entendible, re-usable y extensible.

COAST *(Coperative Application Systems Toolkit)* es una herramienta que ofrece a los desarrolladores una arquitectura para aplicaciones colaborativas y clases correspondientes que pueden ser usadas para implementar tales aplicaciones.
La implementación de esta herramienta no requiere de una plataforma específica.

Los requerimientos del usuario son reflejados en las siguientes características:

- El concepto de replicación combinado con un control de concurrencia totalmente optimista hace posible contar con un proceso inmediato de acciones del usuario independientes de la latencia de la red.

- Un mecanismo de replicación de datos que garantiza una rápida propagación de las acciones de los usuarios en el sistema.

- Un concepto basado en Sesión, que apoya el comportamiento de sesión dinámica.

Por otra parte los requisitos de desarrollo quedan contemplados a través de las siguientes características:

- Manejo de replicación que permite a los desarrolladores tratar de manera transparente objetos replicados como objetos compartidos.

- Manejo de transacciones controlando los accesos a objetos compartidos garantizando la integridad de los datos y su consistencia.

- Acoplamiento de sesión controlada, el cual permite al desarrollador de la aplicación controlar el acoplamiento y desacoplamiento de los aspectos del objeto desde un punto central, dejando el resto del código de la aplicación intacto.

- Manejo de vistas que realiza tareas de actualización de pantallas fuera de la tarea del desarrollador.

COAST es una herramienta para el desarrollo de aplicaciones colaborativas sincrónicas, que refuerza la utilidad y simplifica el desarrollo de tales aplicaciones.
Ofrece componentes básicas y genéricas para el diseño de aplicaciones colaborativas sincrónicas.

Las características básicas de esta herramienta incluyen accesos controlados en las transacciones hacia objetos replicados compartidos, manejo de replicación transparente, y un control de concurrencia totalmente optimista.
En las próximas secciones detallaremos aspectos de su arquitectura.

4.2.2 Arquitectura de COAST

COAST usa un mecanismo de arquitectura replicada, por ejemplo una aplicación colaborativa consiste de diferentes procesos de aplicación (Ej.: instancias de un programa de aplicación) corriendo sobre diferentes sitios. Cada aplicación opera exactamente sobre un documento que es compartido entre estos procesos de aplicación. Los datos del documento son totalmente replicados. Un proceso de aplicación es diseñado de acuerdo a la arquitectura general descripta en la figura 4.1.

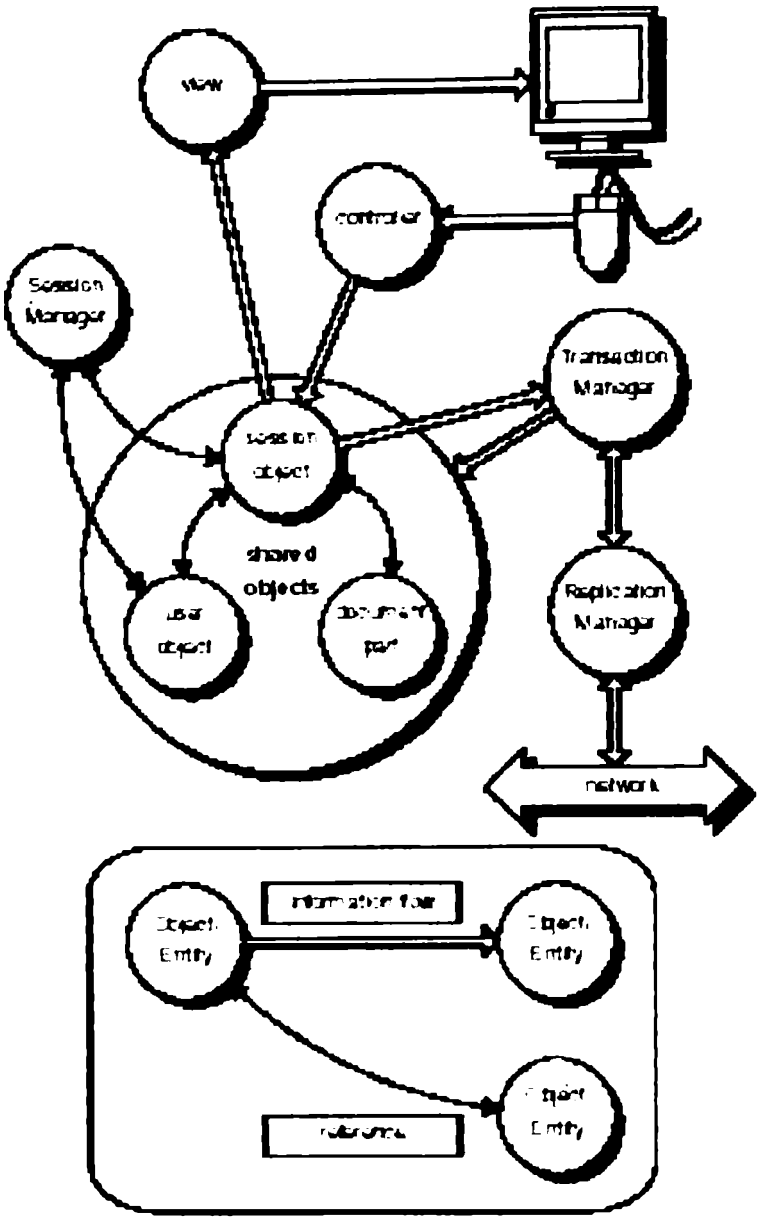


Figura 4.1 – Arquitectura de COAST

Cada aplicación colaborativa manipula un documento compartido. Un documento consiste de partes las cuales pueden emplear una estructura de objeto arbitraria. Este documento es completamente replicado para cada proceso de aplicación.

Los accesos desde los usuarios a los documentos compartidos (o sus partes) son coordinados a través de un **objeto de sesión** o también conocido como *objeto session*.
Los objetos de sesión proveen group awareness (ver capítulo 2 inciso 2.3) y un acoplamiento específico de los aspectos del documento compartido entre los usuarios concurrentes.

Los usuarios interactúan con el documento compartido a través de las **vistas** (visualización de un documento en una ventana) y los **controladores** o controllers (procesan las entradas de los usuarios desde la ventana).

Las vistas y los controladores acceden al documento usando el objeto de sesión asociado. Esto es una extensión del conocido concepto MVC (Model View Controller) [KP88]

Un **administrador de sesión** permite a los usuarios crear, unir o eliminar sesiones (de esta manera abrir y cerrar las ventanas de aplicación asociadas).

Los **objetos de usuario** representan a los usuarios concurrentes del documento. Son usados por los objetos de sesión para mantener listas de sus usuarios que están participando actualmente.
Toda la manipulación del documento compartido, los objetos de sesión, y los objetos de usuario, es encapsulada en las **transacciones**.

Un **administrador de transacciones** garantiza la integridad de los objetos compartidos.

Un **administrador de replicación** es responsable de la sincronización de los objetos replicados (Ej. Documento, objetos de sesión y objetos de usuario).
De este modo ordenando los objetos replicados, se pueden crear documentos persistentes.

4.2.2.1 Manejo de Sesión

El propósito de los objetos de sesión es modelar un grupo de usuarios que interactúan con una aplicación.

- Los objetos de sesión mantienen una lista de objetos de usuarios los cuales representan la participación de cada usuario en la sesión.
- Los objetos de sesión se refieren a una parte del documento al que se están refiriendo actualmente.
- Los objetos de sesión proveen group awareness y definen como los participantes de una sesión trabajan juntos teniendo en cuenta el grado de acoplamiento de los aspectos del documento compartido. Por lo tanto los objetos de sesión actúan como mediadores entre el view-controller y las partes del documento.

Los objetos de usuario y los objetos de sesión en COAST son persistentes y son almacenados con el documento.

La administración de las sesiones (Ej.: crear, unir, salir de una sesión) es manejada a través de las facilidades del administrador de sesión de COAST .
Automáticamente abren las ventanas de aplicación para cada participante de una sesión sobre una parte especifica del documento.
Cuando un usuario entra a una sesión, el objeto usuario asociado es agregado a la lista de usuarios de la sesión.
Eliminar una sesión significa simplemente borrar el objeto usuario de la lista de usuarios en la sesión.

4.2.2.2 Grados de acoplamiento

La replicación de objetos resulta en una copia exacta de todos los aspectos de un documento en cada estación de trabajo.
Mientras este acoplamiento estricto reúne los requisitos de grupos que trabajan en el mismo tema de una manera herméticamente acoplada, se convierte en un obstáculo para muchos otros casos.
Por ejemplo en una editor de texto compartido la posición del scroll necesita ser acoplada a todos los miembros de una sesión herméticamente acoplada, pero no necesita ser acoplada a todos los usuarios que están trabajando sobre el documento.

Como Coast es una herramienta para aplicaciones colaborativas en general, provee mecanismos para acoplar y desacoplar aspectos de los objetos compartidos dinámicamente.
Se usan los términos de **"modo de colaboración"** y **"contexto de aplicación"** para describir esta clase de acoplamiento dinámico.

El modo de colaboración define las reglas para el acoplamiento de los aspectos de los objetos compartidos entre los usuarios o grupos de usuarios.

Modos de colaboración:
- Herméticamente acoplado
- Débilmente acoplado
- Modo individual
- Modo Subgrupo

Un **modo de colaboración** asigna cada aspecto a un nivel en el que ese aspecto tiene que ser acoplado entre los usuarios.

Un **contexto de aplicación** agrupa las propiedades de la situación actual en la que los usuarios están trabajando con una aplicación.

Las propiedades de un contexto de aplicación C son:
- El grupo de usuarios G colaborando actualmente.
- El usuario local U

4.2.2.3 Manejo de transacciones

COAST introduce tres tipos de transacciones:

- **Transacciones de usuarios**: Usadas para encapsular las modificaciones de los objetos compartidos llevadas a cabo por los controladores. Esta es la única diferencia con el concepto original de Model View controller. Las transacciones de usuarios son llevadas a cabo inmediatamente sobre los objetos replicados localmente (actualización local). Luego son replicados por el administrador de replicación a todos los sitios para que la modificación sea haga en el ámbito global.

- **Transacciones de vistas**: Usadas para encapsular los accesos a las vistas de los objetos compartidos. Son inicializadas automáticamente por el sistema y no modifican el objeto compartido.

- **Transacciones de actualización**: Usadas por el administrador de replicación para ejecutar globalmente las transacciones de usuarios que se llevaron a cabo solo a nivel local desde otros sitios de trabajo y que afectaron a los objetos replicados. Las transacciones de actualización son inicializadas por el sistema.

4.2.2.4 Manejo de Replicación

El administrador del sistema COAST provee un ambiente en el cual el desarrollador puede manejar los datos de la aplicación del mismo modo que en una aplicación no colaborativa. Los detalles técnicos como replicación, notificación y control de concurrencia son ocultos al desarrollador de la aplicación.

4.2.2.5 Notificación de cambios

Las modificaciones a un documento pueden ser llevadas a cabo en cualquier momento trabajando sobre los objetos replicados usando transacciones de usuarios. Para que estos cambios puedan ser reflejados en otros sitios se debe hacer un broadcast de dichos cambios. Este proceso es llamado notificación

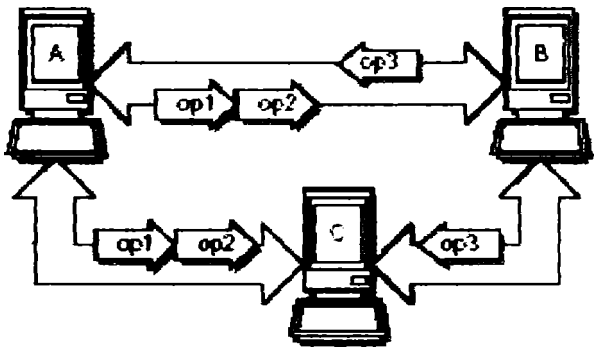


Figura 4.2 – Notificación de cambios en COAST

La entrega de las notificaciones pude retardarse dependiendo de la velocidad de la red, pero debido al control de concurrencia totalmente optimista que maneja el sistema COAST, este no tiene que esperar por tales resultados.. Esto significa que el usuario puede continuar trabajando aun cuando exista una conexión de red inestable o lenta hacia los otros sitios.

4.2.2.6 Control de concurrencia optimista distribuido

Las copias de un documento en diferentes sitios deben ser sincronizadas. Esto se lleva a cabo mediante las notificaciones. Por supuesto que pueden llevarse a cabo notificaciones conflictivas a un documento que podría dar como resultado un documento inconsistente si cada sitio simplemente ejecuta las notificaciones que recibe.

Eliminar este tipo de inconsistencias es tarea del control de concurrencia.

Un objetivo en el diseño de COAST fue garantizar el no retardo del procesamiento de las acciones del usuario. Es así que los diferentes procesos de aplicación deben trabajar independientemente uno de otro.
Se ha desarrollado un algoritmo de control de concurrencia optimista (DOCC-Distributed Optimistic Concurrency Control) el cual cumple con esta necesidad.
Esta basado en el algoritmo ORESTE [KBL93]. Intenta minimizar las transacciones de undo y redo cuando las transacciones son re-llamadas.

Las transacciones consisten de un conjunto básico de operaciones que son definidas por el administrador de transacciones. Esto tiene la ventaja de que pueden realizarse operaciones undo y redo sin involucrar al diseñador de la aplicación y esa conmutatividad de transacciones puede ser detectado por el administrador de replicación.

4.2.3 Análisis de Herramientas desarrollados con COAST

Para estudiar el comportamiento y las características del framework COAST se analizaron dos herramientas desarrolladas sobre la base de dicho framework. El objetivo de nuestro análisis esta centrado en como las herramientas manejan la consistencia de información y si dan soporte a la propiedad de latecoming de acuerdo a lo que el framework permite desarrollar.
Las dos herramientas que se analizaron fueron el UML-Editor [SSSoo] y CROCODILE.

La herramienta UML-Editor permite la creación de diagramas de clases UML en forma colaborativa, donde los participantes pueden crear, editar y borrar clases dentro de un diagrama compartido.

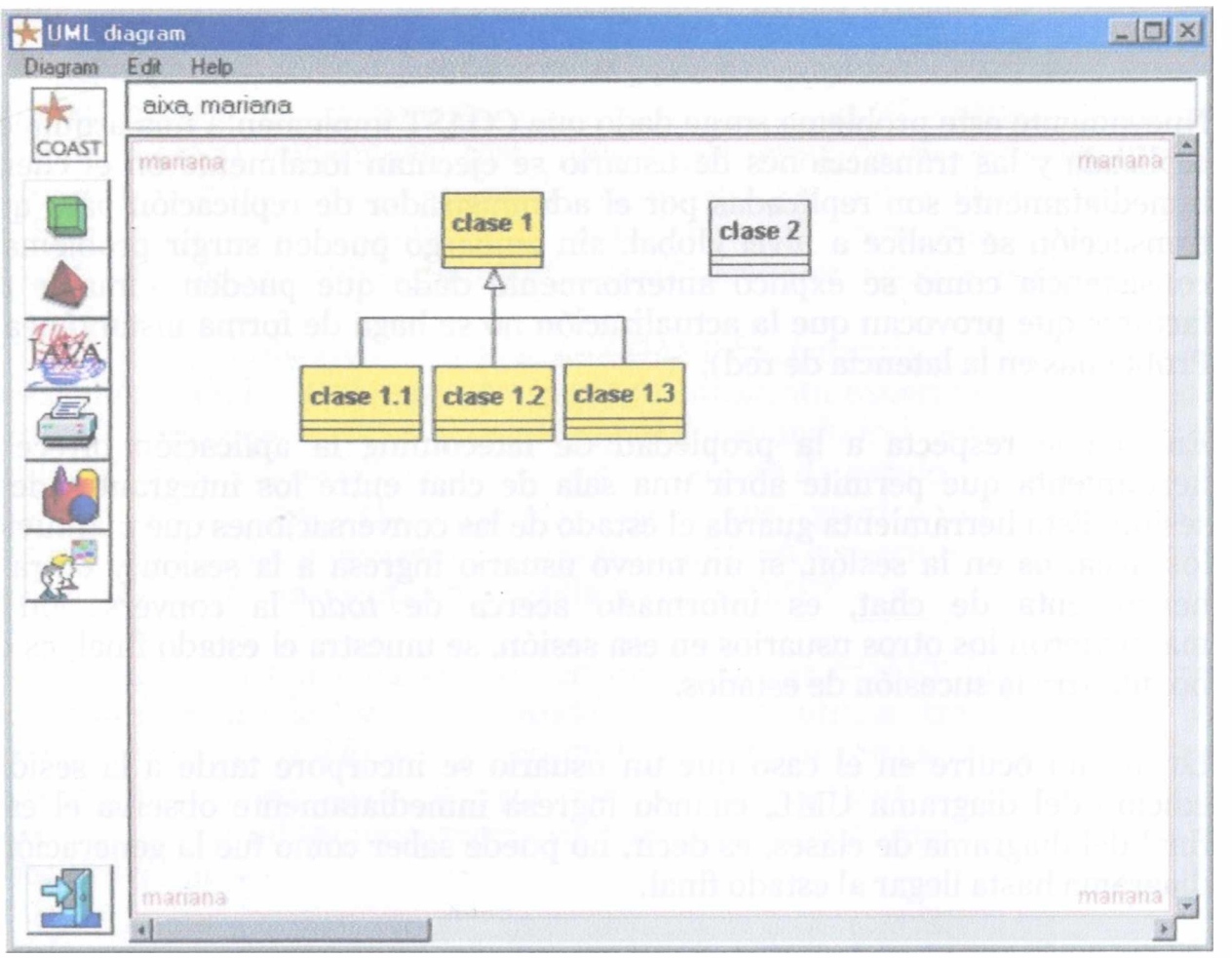


Figura 4.3 – Herramienta UML-Editor desarrollada con COAST

Se realizaron una serie de pruebas para estudiar el comportamiento de la aplicación, respecto al manejo de consistencia de información y a su incorporación de la propiedad de latecoming, como se mencionó anteriormente.

Entre dichas pruebas, la más significativa fue cuando un usuario borra una de las clases del diagrama UML al mismo instante que otro usuario esta editando sus propiedades.
El primer usuario logra borrar la clase sin problemas, y el segundo usuario aún cuando la clase ya había sido borrada podía seguir editando sus propiedades.
Una vez finalizada la edición de la clase, ésta ya no existía en el diagrama UML, esto fue porque anteriormente había sido eliminada por otro usuario dentro de la sesión.

Aquí se pudo comprobar que la herramienta implementa *transformación de operaciones* como método de control de concurrencia para garantizar consistencia en los datos, pero debido a que COAST implementa una arquitectura replicada los datos se copian localmente en el cliente, por eso el segundo usuario podía seguir editando la clase que ya había sido borrada por otro usuario dentro de la sesión. Cuando finalizó la edición de la misma desapareció del diagrama de clases ya que el cliente recibió la notificación adecuada desde el servidor de que la clase ya había sido eliminada anteriormente en la sesión.
En este caso en particular, la aplicación no responde adecuadamente, dado que no esta bien que un usuario pueda seguir editando una clase que en realidad ya fue borrada por otro usuario anteriormente, como así tampoco se debería

permitir a un usuario borrar información que esta siendo manipulada por otro usuario en la sesión.

Nuevamente este problema surge dado que COAST implementa una arquitectura replicada y las transacciones de usuario se ejecutan localmente en el cliente e inmediatamente son replicadas por el administrador de replicación para que la transacción se realice a nivel global, sin embargo pueden surgir problemas de consistencia como se explicó anteriormente dado que pueden sumarse otros factores que provocan que la actualización no se haga de forma instantánea (Ej. Problemas en la latencia de red).

En lo que respecta a la propiedad de latecoming la aplicación ofrece una herramienta que permite abrir una sala de chat entre los integrantes de una sesión. Esta herramienta guarda el estado de las conversaciones que mantuvieron los usuarios en la sesión, si un nuevo usuario ingresa a la sesión y entra a la herramienta de chat, es informado acerca de *toda* la conversación que mantuvieron los otros usuarios en esa sesión, se muestra el estado final, es decir no muestra la sucesión de estados.

Lo mismo ocurre en el caso que un usuario se incorpore tarde a la sesión de edición del diagrama UML, cuando ingresa inmediatamente observa el estado final del diagrama de clases, es decir, no puede saber como fue la generación del diagrama hasta llegar al estado final.

Al igual que con la herramienta mencionada anteriormente se realizaron varias pruebas para ver el manejo de consistencia y latecoming en CROCODILE.

La herramienta soporta el trabajo de grupos de usuarios en forma colaborativa sincrónica y asincrónica.
Cuando los usuarios están trabajando en modo sincrónico comparten componentes como textos, gráficos o cuadros que se encuentran en una biblioteca general y pueden realizar sobre ellos modificaciones en forma individual o grupal. Cuando los usuarios se encuentran trabajando en forma grupal, por ejemplo comparten una pizarra, si un usuario realiza una modificación la misma es observada por el resto inmediatamente, debido a que el framework utiliza transformación de operaciones para mantener consistentes los datos.

Mientras un usuario se encuentra en su sesión puede crear, importar, hacer anotaciones sobre diferentes documentos, y luego deposita dichos documentos en una biblioteca general la cual puede ser accedida por el resto de los usuarios.
Si el usuario abandona la sesión y crea un documento en forma asincrónica, posteriormente puede agregarlo y se notificará al resto de la existencia del nuevo documento.

En cuanto a la propiedad de latecoming, en caso de que un usuario abandone su sesión y algún documento sea modificado, cuando retorna (reingresa a la sesión) podrá observar todos los cambios realizados pero sin poder observar la sucesión de estados sino el estado final del documento.

4.2.4 Conclusión

Según se explicó en la sección anterior, de acuerdo al análisis citado en [SKSH96], COAST implementa una arquitectura replicada, es decir, los objetos son copiados localmente en cada sesión de usuario. Las transacciones de usuario son llevadas a cabo localmente e inmediatamente son replicadas por el administrador de replicación para que la modificación se realice a nivel global.

Principalmente, el objetivo de diseño de COAST fue garantizar el no retardo del procesamiento de las acciones del usuario, es decir, un usuario ejecuta acciones sobre los objetos compartidos, y puede seguir trabajando independientemente de cuando se lleven a cabo las notificaciones de cambio de estado del objeto a nivel global, y aquí surge el problema, dado que pueden ocurrir posibles inconsistencias en la información, como se explicó en la sección 4.2.3 de acuerdo al análisis de dos herramientas desarrolladas con COAST.

Hay dos puntos a tener en cuenta, en primer lugar COAST implementa una arquitectura replicada, y en segundo lugar se utiliza transformación de operaciones como método de control de concurrencia para garantizar consistencia de información, si bien este método funciona adecuadamente, se pueden generar posibles inconsistencias temporales, hasta que la notificación se realice a nivel global.

Y en lo que respecta a la propiedad de latecoming solo se informa al nuevo usuario el estado final de la sesión, esto también se debe a que el método de transformación de operaciones no funciona bien en caso de querer implementar playback de los eventos en una sesión colaborativa, con lo cual no es posible informar al usuario la secuencia ordenada en que ocurrieron los cambios en la sesión de trabajo compartida.

4.3 Framework DYCE

4.3.1. Introducción

DyCE, un framework Java para el desarrollo de componentes colaborativos llamados Groupware Components. Los componentes desarrollado sobre DyCE pueden ser distribuidos en redes, usados de manera colaborativa y combinados con otros componentes para formar ambientes colaborativos. Se presenta adicionalmente el modelo de programación central que permite el acoplamiento y reutilización de componentes.

En la actualidad, se está incrementando rápidamente la necesidad tanto de colaboración en equipos como de ambientes de trabajo flexibles.
Los ambientes colaborativos involucran a equipos de trabajo cambiantes y de diferentes tamaños, utilizando diferentes herramientas o combinaciones de ellas. A medida que la unidad de trabajo evoluciona y la complejidad de los productos crece, los requerimientos de nuevas herramientas aumentan. Otro de los factores que fomentan estos ambientes colaborativos, es la tendencia a la creación de

organizaciones virtuales, por ejemplo organizaciones distribuidas geográficamente, con la necesidad de compartir la creación de nuevos productos.

Las aplicaciones CSCW permiten que un grupo de usuarios resuelva una tarea en común de manera cooperativa sobre artefactos compartidos. Las herramientas computacionales usadas para soportar esos trabajos colaborativos se denominan *Groupware*.

Una tendencia adicional es la movilidad de los trabajadores, para lo cual se debe dar soporte a muchas plataformas. Se debe tener en cuenta que los dispositivos móviles tienen limitaciones por ejemplo de memoria, de pantallas, etc., para lo cual los sistemas deben ser adaptables a la situación de cada uno de éstos. Los miembros del equipo de trabajo colaborativo tienen que poder hacer extensivo el ambiente colaborativo con el fin de poder adaptarlo a sus necesidades de cambio. Es importante destacar la evolución del desarrollo basado en componentes que permite la reutilización y extensión de ambientes de aplicación.

DYCE se basa en tres conceptos:

- ✓ Replicación de Objetos
- ✓ Código Móvil
- ✓ Modelo de programación orientado a las tareas

4.3.2 Metas y Requerimientos

Requerimientos que originaron el diseño de DYCE:

- ❑ Combinación de herramientas y reutilización:
- ❑ Herramientas de diferentes tipos deberían poder ser integradas y compartidas dentro de un mismo artefacto de trabajo colaborativo.
- ❑ Las herramientas colaborativas deben ser utilizables en diferentes contextos.
- ❑ Extensibilidad
    - ○ Propiedad de personalización en tiempo de ejecución:
        - ▪ Beneficios para programadores: capacidad de desarrollo y mejoras en los sistemas
        - ▪ Beneficio para usuarios: capacidad de adaptar el sistema a sus necesidades
    - ○ Extensibilidad en tiempo de ejecución: capacidad de desplegar y poner a disposición de los usuarios finales nuevas funcionalidades en las aplicaciones de manera transparente para ellos.
- ❑ Independencia de la plataforma
- ❑ Soporte a varios modelos colaborativos
- ❑ Se debe dar un soporte tanto a modelos colaborativos sincrónicos como asincrónicos.

4.3.3 Diseño del Sistema

DyCE fue un Framework desarrollado para dar soporte y solución a lo planteado anteriormente, y puede actuar como la base para el desarrollo de nuevas

herramientas groupware, brindando un ambiente para el uso de dichas herramientas.

DyCE fue orientado hacia una arquitectura basada en componentes, ya que este tipo de arquitectura es la que mejor soporta los requerimientos de flexibilidad y extensibilidad descritos anteriormente.

4.3.3.1 Componentes Groupware

Los componentes groupware son el concepto fundamental y central del framework DyCE

Se puede definir estos de la siguiente manera:

- Herramientas visualmente interactivas y potencialmente complejas, dándole a los usuarios la posibilidad de manipular datos compartidos..

- Componentes mas complejos que un simple componentes de interfaces de usuario como ser un elemento de entrada de texto (Input Text). Un ejemplo de estos componentes es un editor de dibujos compartido. Estos pueden ser compartidos y combinados junto a otros componentes para formar un ambiente colaborativo.

- Son componentes que soportan los conceptos colaborativos, es decir pueden acceder a la información acerca de la situación colaborativa actual del ambiente.

4.3.3.2 Arquitectura del Sistema

Sistema cliente-servidor desarrollado en Java, realizando la comunicación entre ambos mediante RMI y Sockets asincrónicos. Lo anterior hace que DyCE sea portable hacia cualquier ambiente Java, por ejemplo soporta ambientes en donde se utilicen dispositivos móviles implementados con J2ME (Java 2 Micro Edition).

Los usuarios acceden a los componentes Groupware vía Escritorios Groupware (*Groupware Desktop*) de donde se inician sesiones colaborativas o acceder a los componentes compartidos. Las aplicaciones desarrolladas pueden tener accesos selectivos y específicos a los diferentes componentes, por ejemplo mediante portales corporativos o intranets.

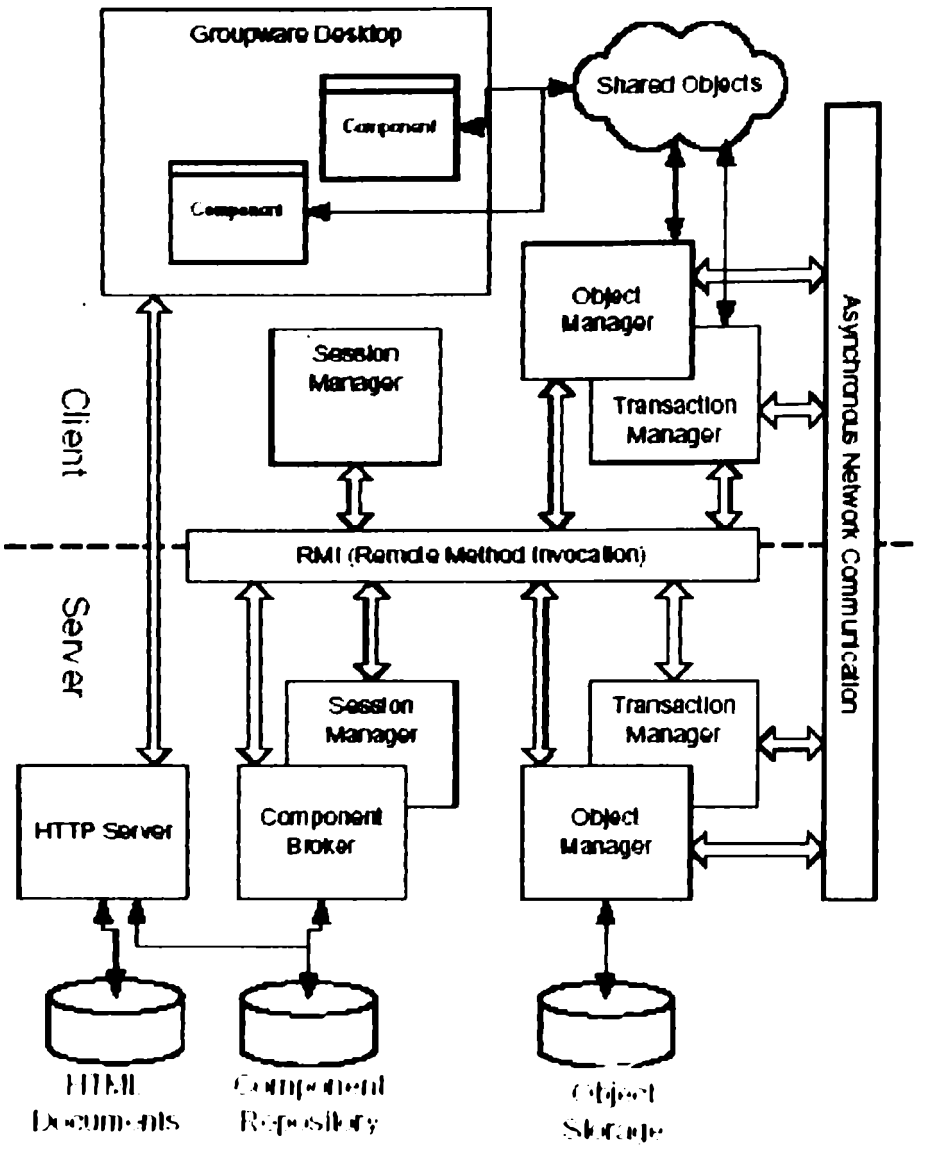


Figura 4.4 – Arquitectura de DyCE

En DyCE el Server tiene las siguientes responsabilidades:

- **Administración de Objetos:** el Server mantiene y administra todos los datos compartidos. Los datos son almacenados de manera persistente en un objeto Base de Datos para lograr la persistencia en los cambios dentro de una sesión colaborativa. Se accede a estos datos mediante interfaces RMI.

- **Administración de Transacciones:** los cambios se realizan a través de manera segura, encapsulándose los mismos dentro de transacciones. El Server procesa las transacciones óptimamente y realiza el *undo* de manera efectiva en caso de problemas en la ejecución de las mismas.

- **Administración de Componentes:** los componentes Groupware se almacenan dentro del *Component Repository*, y son administrados por el Component Broker, en donde debe registrarse cada componente que se disponga para el Sistema y los usuarios. Cuando los clientes solicitan un componente, se realiza una búsqueda dentro de los componentes registrados en el Broker, en donde se identifica los componentes, su implementación y su respectivo almacenamiento.

- **Administración de Sesiones:** el Server DyCE se encarga del mantenimiento de cada una de las sesiones colaborativas iniciadas. La información de cada una de las sesiones es brindada por el *Session Manager*

- **Servicio Http:** el Server DyCE contiene embebido un servidor http que se utiliza en el momento en que los clientes invocan a un componente al Broker, este es bajado dinámicamente hacia el cliente mediante http. De igual manera, se pueden hacer despliegues de nuevos componentes desarrollados mediando el mismo protocolo. Adicionalmente el Server http que brinda DyCE puede ser utilizado para montar aplicaciones web o intranets en donde se acceda a estos componentes compartidos y se inicien sesiones colaborativas.

4.3.3.3 Objetos Compartidos

Los objetos compartidos pueden estar acoplados de manera de dar a los grupos de usuarios una forma de interactuar con un estado compartido común a todos. Se identifican tres conceptos básicos para solventar diferentes requerimientos:

- Compartimiento de Aplicaciones: el resultado de una aplicación central es distribuido entre todos los usuarios. Las entradas de los usuarios son ingresadas a la aplicación de manera serial, como si fuesen ingresadas por un solo usuario. Estrictamente para ambientes WYSIWIS con comunicación sincrónica.

- Compartimiento de Eventos GUI: cada usuario ejecuta una instancia de la aplicación colaborativa y cada evento GUI es capturado y distribuido entre las demás instancias de manera que simule un procesamiento local. Solo es soportado para aplicaciones exactamente idénticas y es estrictamente para ambientes colaborativos WYSIWIS en donde se pueda identificar las mismas operaciones en los distintos clientes.

- Compartimiento de Datos: cada usuario ejecuta una instancia de sus aplicaciones, las cuales tienen acceso a una serie de datos compartidos comunes para todas. Estos datos pueden tener una implementación centralizada o replicada. Adicionalmente, se pueden basar las aplicaciones en una arquitectura MVC permitiendo una mayor flexibilidad en el ambiente colaborativo. Es posible usar un modelo compartido debido a que los objetos son compartidos semánticamente consistentes. También se permite diferentes modos de acoplamiento y combinaciones entre comunicación sincrónica y asincrónica brindando mayor flexibilidad.

DyCE adopta el compartimiento de datos, ya que es el concepto que brinda la mayor flexibilidad. Los datos son separados de los componentes que se utilizan para acceder a esos datos de una manera similar en la que se separa las aplicaciones de los modelos de dominios.

El estado compartido es modelado como objetos compartidos, los cuales son replicados dinámicamente en un sistema distribuido y son modificados por los componentes a través de transacciones seguras y son mantenidas en un estado consistente global y cada uno de los clientes mantiene una réplica de este estado.

El framework DyCE ofrece clases para el desarrollo de Componentes Groupware, como así también el modelo de objetos utilizados en el mismo.

Modelo de Objetos DyCE

El framework DyCE provee las clases (*MobileComponent*) para el desarrollo de nuevos componentes, como así también los objetos del modelo (*ModelObject*) utilizados por ellos. Las clases derivadas de *ModelObject* proveen una interfaz a los elementos de datos generales, que son contenidos dentro de un conjunto de instancias *RObject*. Las subclases de *ModelObject* definen los esquemas de objetos compartidos.

De esta manera, el mecanismo de replicación solo necesita interactuar con las instancias *RObject*, y las instancias *ModelObject* pueden ser recreadas localmente. Las subclases de *ModelObject* mantienen lógica de aplicaciones directamente relacionadas a objetos compartidos y no vinculados a un componente en particular, logrando una clara separación:

- Clases de Componentes: usado para visualizar e interactuar con objetos compartidos.
- Clases del Modelo del Dominio: que proveen una interfaz hacia los objetos compartidos, mediante significados semánticos.

Esta separación es muy útil para el desarrollo de nuevos componentes. Los objetos *RObjects* se almacenan de manera persistente en el Server. Al mantenerse estable el conjunto de datos, las implementaciones de las subclases de *ModelObject* y *MobileComponent* pueden continuar, pueden realizarse cambios y extensiones y el esquema de objetos *RObjects* continuará válido

En el diagrama de clases presentado, la línea punteada representa los límites del framework y sus extensiones. El componente *DiagramEditor* deriva de la clase *MobileComponent*, en tanto el dato del diagrama es mantenido en una clase *Diagram*, derivada de *ModelObject*. En tiempo de ejecución, cuando se cree una instancia de la clase *Diagram*, una instancia de *RObject* se creará para mantener los datos compartidos replicados.

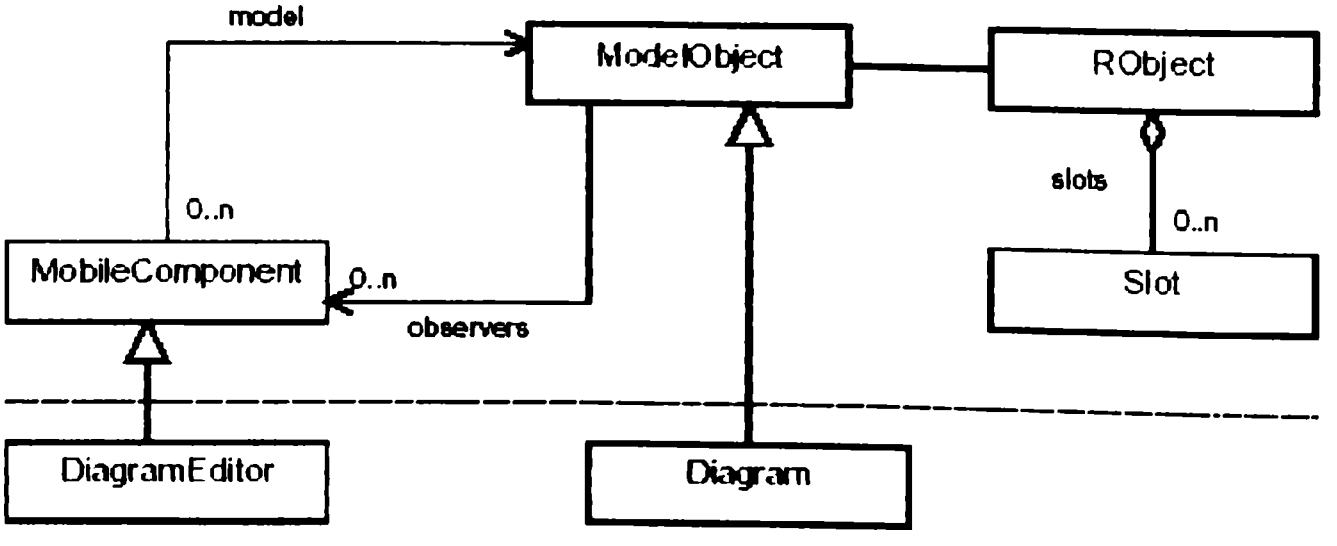


Figura 4.5 – Modelo de objetos de DyCE

4.3.3.4 Movilidad de Componentes

Uno de los requerimientos que se debe resolver es el del soporte a extensibilidad en tiempo de ejecución, es decir, la posibilidad de incorporar nuevos componentes en los sistemas que ya están siendo ejecutados y que inmediatamente queden disponibles para los usuarios.

Para esto se explota la característica de Java de poder descargar dinámicamente las implementación de las clases desde el servidor, instancias las clases y ejecutar su código (*código móvil*). Al agregar nuevos componentes se actualiza el *Component Repository* vía servicio http y éstos son registrados automáticamente en el *Component Broker*.

Esto permite a los usuarios poder tener a su disposición todos los componentes, sin necesidad de estar reiniciando sus aplicaciones. Adicionalmente, bajo esta implementación los usuarios pueden descargar solamente los componente que realmente son de su utilidad, lo cual representa una importante característica ya que ,en ambientes colaborativos son frecuentes clientes de dispositivos móviles como palms o teléfonos celulares, en donde la capacidad de almacenamiento por lo general es limitada.

Cada vez que se quiere acceder a un nuevo componente, éste se descarga en el cliente y se abre localmente. La nueva instancia creada es incorporada dentro del *Object Manager* se replica en los demás clientes que acceden al mismo componente.

4.3.4 Modelo de Programación de Componentes

Una de las metas que persigue DyCE es la de lograr una flexible reutilización y combinación de componentes, para lo cual se requiere de un modelo de programación general que provea a los componentes las posibilidad de interactuar con otros componentes, sin tener previo conocimiento de la existencia mutua.

Puntualmente se necesita identificar los componentes requeridos por el o los usuarios en una situación colaborativa determinada.

En concepto central del modelo de programación es el de ***tarea***. Es por ello que se conoce a este framework como basado en tareas (*task-based*). Una tarea es una interacción específica con un dato específico para el cual hay Componentes Groupware disponibles. El identificador único de las tareas es el nombre y es aplicado sobre algunas clases del modelo del dominio. Las tareas proveen la conexión entre los Componentes Groupware y los objetos del dominio.

Cada componente debe publicar una serie de tareas, las cuales pueden ser invocadas por otros componentes o por usuarios. Cuando se utiliza la información de estas tareas para acceder a los componentes compartidos, se dice que la tarea es ejecutada. Todo el conjunto de tareas es administrado por el Broker, en donde se busca en el conjunto de tareas el componente idóneo para

realizar alguna tarea específica. Los componentes pueden invocar servicios de otros componentes utilizando las tareas específicas.

La información de las tareas, de los usuarios y de la infraestructura de los componentes es usada por el Broker para determinar qué tarea es la requerida para ejecutar algún servicio puntual. De esta manera, diferentes componentes pueden ser asignados a distintos usuarios dependiendo de esa información, un ejemplo claro es el de usuarios con diferentes privilegios sobre los componentes.

4.3.4.1 Usar tareas para estructurar aplicaciones – Un Ejemplo

Diferentes componentes pueden ser publicados con diferentes tareas, aplicados sobre las mismas clases de objetos del dominio. Un ejemplo es un conjunto de componentes desarrollados para crear una presentación HTML compartida. Se creó un modelo de dominio central *HTMLPresentation*, el cual solo contiene una lista de URLs y un índice para la posición actual en la presentación. Un componente de edición tiene la funcionalidad de agregar nuevas páginas a la presentación, para lo cual publica la tarea de "edición", aplicado a la clase *HTMLPresentation*. Por otro lado se crea un componente con la funcionalidad de visualizar la presentación, para el cual se publica la tarea de "presentación" sobre la misma clase. Finalmente se crea un tercer componente que actúa como controlador de la presentación, es decir efectúa el avance y retroceso entre slides. Para este componente se publica la tarea de "controlar" aplicado siempre sobre la clase *HTMLPresentation*.

Cuando se realiza una presentación con estos componentes groupware, se puede crear una instancia de *HTMLPresentation* interactivamente desde el Groupware Desktop. Cuando se ejecuta la tarea de "edición" en el objeto compartido, se abre el componente de edición en una ventana en la sesión de ese usuario, permitiendo al usuario trabajar individualmente con el objeto mediante el componente asociado. Otros usuarios pueden ser invitados a la sesión colaborativa para preparar la sesión de manera conjunta agregando nuevas URLs. La sesión es replicada a los usuarios invitados, en donde se brinda accesos al objeto compartido y al componente asociado para la tarea de edición. Debido a que los componentes de los usuarios están basados en el mismo objeto compartido son automáticamente acoplados permitiendo una colaboración sincrónica en la presentación compartida.

El framework DyCE hace permanecer en un estado consistente al objeto de datos compartido. Cuando el presentador ejecuta la tarea de "presentación" en los objetos de presentación compartidos, al igual que en la tarea anterior, se invoca al componente asociado a esa tarea y se invita a los demás usuarios a entrar a la sesión.

El mismo presentador tiene la posibilidad de ejecutar la tarea de "control", de manera de guiar la presentación que está siendo visualizada por los demás usuarios que comparten la presentación.

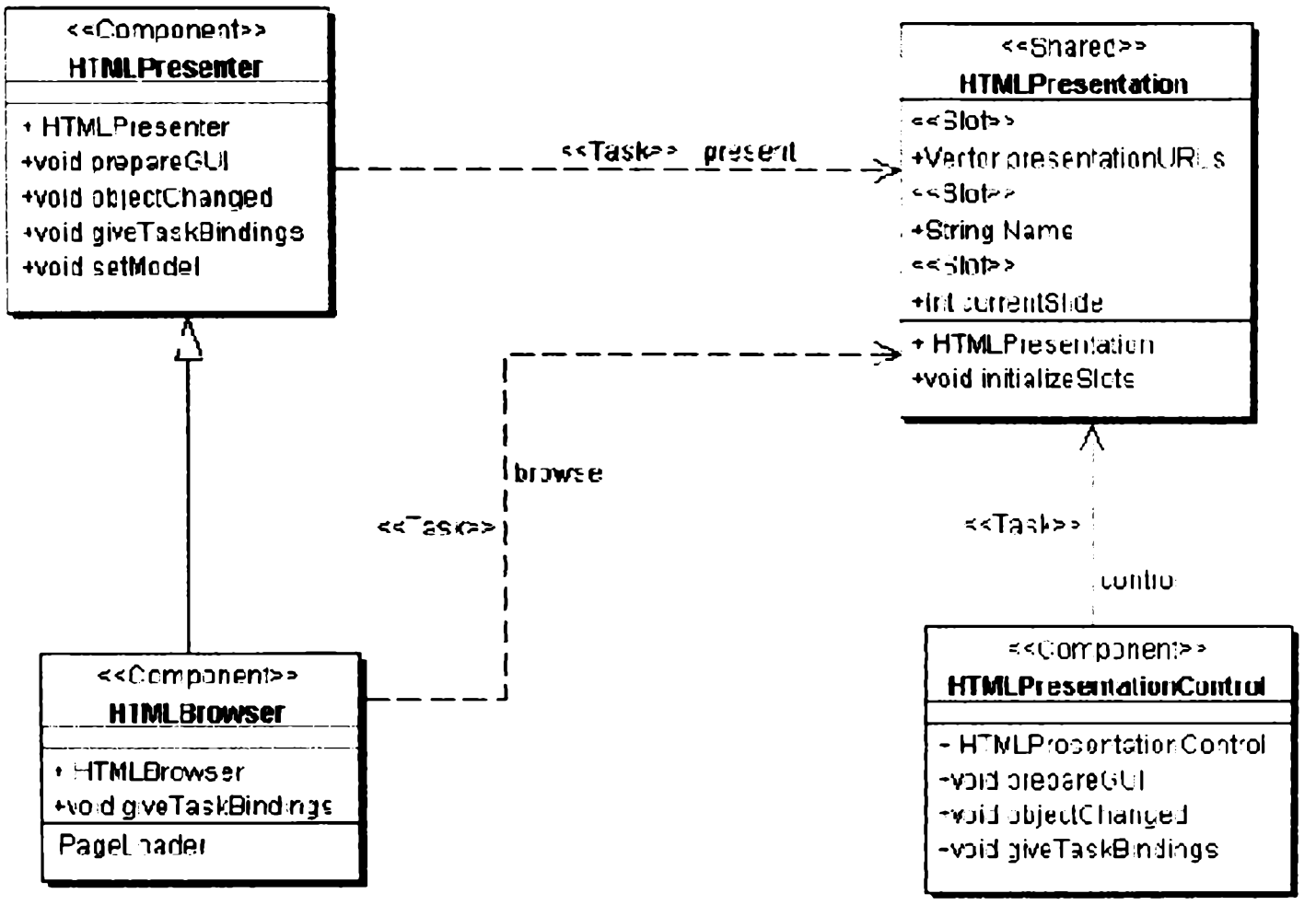


Figura 4.6 – Ejemplo HTMLPresentation

4.3.4.2 Aplicaciones y Sesiones

Las sesiones dentro de los sistemas DyCE mantienen información acerca de los usuarios involucrados en una sesión y las tareas que ejecutan. Con el fin de configurar ambientes de aplicación complejos, se puede ejecutar múltiples tareas dentro de una misma sesión. Todos los usuarios de una misma sesión comparten los mismos objetos del modelo y tareas, aunque no necesariamente los mismos componentes. Cuando se ejecuta una tarea sobre un objeto del modelo en una sesión, el componente relacionado a esa tarea es descargado en el cliente e instanciado localmente en el objeto del modelo. Esta ejecución sobre las tareas es realizada por cada uno de los clientes de la misma sesión.

Cuando un ***latecomer*** se une a la sesión, la información de la sesión es replicada en el cliente y todas las tareas incluidas son invocadas.

Las sesiones son modeladas como objetos compartidos DyCE, con lo cual son replicadas y persistentes, con lo cual estas sesiones pueden ser almacenadas y suspendidas y ser reiniciadas posteriormente, en donde al igual que con el manejo de latecoming, se invocan todas la tareas asociadas a la sesión.

4.3.4.3 Sesiones y acoplamiento

El acoplamiento de los componentes usados en una sesión son definidos por los mismos componentes. El framework asegura que el objeto compartido replicado está en un estado global consistente dentro del sistema distribuido. Debido a que los usuarios pueden estar utilizando componentes diferentes para sus tareas, no es posible aplicar estas reglas en un acoplamiento WYSIWIS. En el caso de querer compartir aspectos visuales, éstos deben ser implementados como parte del modelo compartido.

4.3.5 Composición de Componentes

El framework provee dos niveles de extensibilidad:

- Extensibilidad para programadores: brinda a los programadores Java un framework de desarrollo en el cual basarse para la creación de nuevos componentes. Al igual que provee de características propias para lograr conectividad con otros componentes de manera muy sencilla.
- Extensibilidad para usuarios finales: provee funcionalidad para poder interactuar y combinar diferentes componentes, para lo cual los usuarios son proveídos con una herramienta de composición gráfica colaborativa.

Esta herramienta provee acceso a las tareas, y a la información del modelo de objetos disponible en el servidor DyCE, y permite la creación de "configuraciones". Éstas son combinaciones de clases de objetos del modelo y tareas encargadas de resolver cual es el componente relevante. También contiene descripciones del layout de los componentes.
El DyCE Groupware Desktop provee la manera de crear configuraciones. Crea instancias de todos los elementos contenidos dentro de la configuración como modelos compartidos de componentes combinados. Luego de crear estas instancias, se invocan tareas específicas sobre esos objetos compartidos, los componentes son descargados, instanciados y alineados según las especificaciones gráficas almacenadas en la definición de la configuración. Luego los componentes pueden ser utilizados de manera colaborativa normalmente.

En el ejemplo se puede ver como dos usuarios combinan dos componentes para interactuar con una presentación HTML. Uno de los componentes deberá visualizar una URL contenida en el modelo de presentación compartida, el otro deberá poder modificar la URL. Ambos componentes serán acoplados dentro de un mismo componente (html browser). Se destaca que no se especifica cuales componentes se utilizarán, solo se menciona una serie de tareas que deberán ser ejecutadas sobre los objetos compartidos.

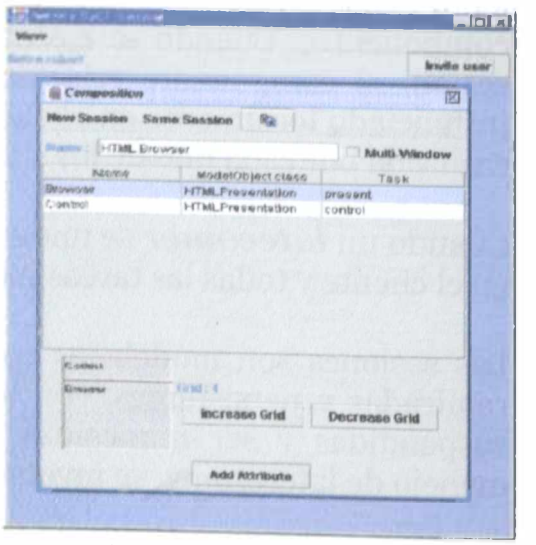


Figura 4.7 – Ejemplo HTMLPresentation
Uso de componentes

4.3.6 Ejemplos de Uso

El framework DyCE está siendo usado como plataforma en diversos proyectos de investigación y de desarrollo. En proyectos de desarrollo de un sistema de administración y modelado de procesos distribuidos para empresas virtuales, distribuidas en cinco localidades en tres países europeos.

Se desarrolló un workspace hipermedia compartido que permite la creación de gráficos de procesos anidados, de una manera gráficamente interactiva. Estas estructuras de procesos pueden ser utilizadas para modelar procesos colaborativos, por ejemplo un procesos de escritura conjunta. El workspace es modificado por una serie de componentes colaborativos. El *Workspace Editor* usado para la manipulación del mismo workspace permitiendo por ejemplo, la creación de nuevos componentes, procesos y estructuras. El *Content Editor* es usado para modelar la estructura de contenidos anidados de un artefacto compartido, por ejemplo las secciones y sub-secciones de una paper escrito colaborativamente. El *Team Editor* usado para modelar el trabajo del equipo de trabajo sobre las tareas comunes. El *Process Editor* usado para modelar las estructuras de los procesos.

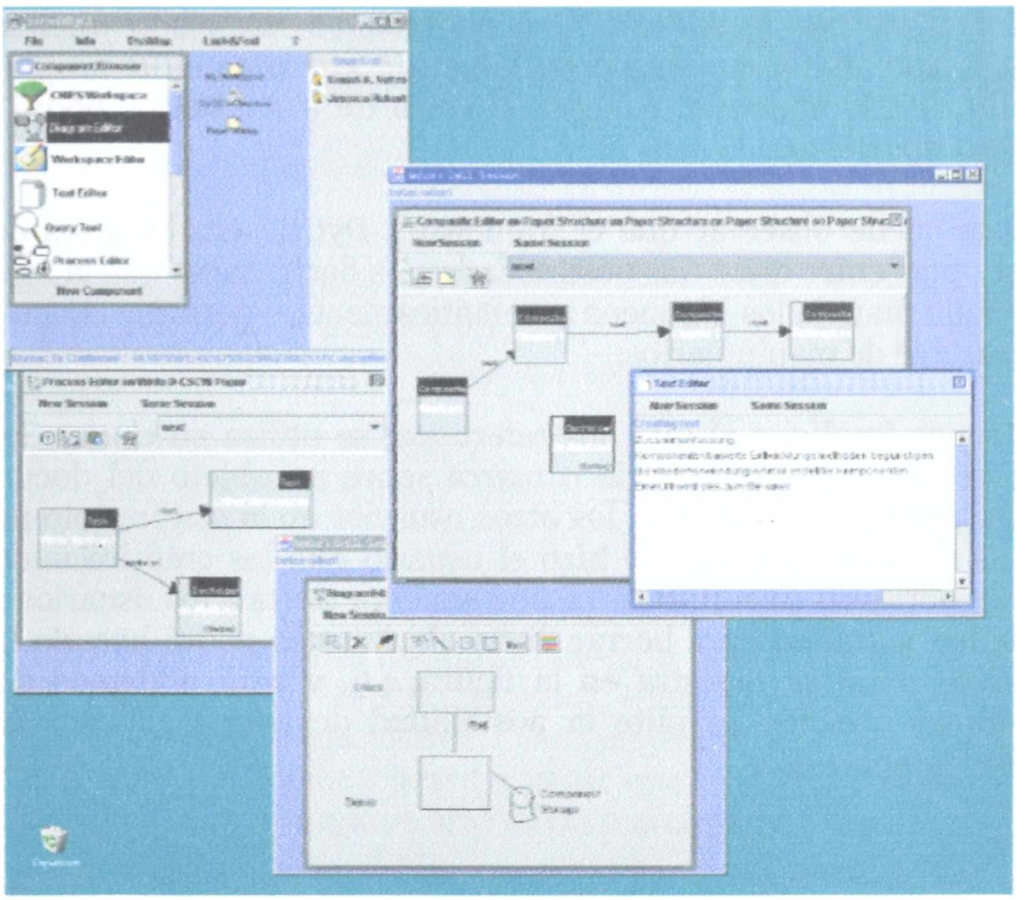


Figura 4.8 – Ejemplo: Workspace hipermedia compartido

Al estar basado en modelo de programación DyCE, los componentes se benefician con extensibilidad y reusabilidad. Es permitida la inclusión de objetos de contenido de cualquier tipo al workspace. El acceso a las tareas sobre los objetos de contenidos permite poder invocar a los componentes groupware asociados a dichos objetos. De esta manera, cuando por ejemplo se requiere hacer un dibujo en el paper colaborativo un modelo de dibujo compartido puede estar disponible

en el espacio de contenidos y las tareas publicadas sobre ese modelo de dibujo permiten la invocación del editor de dibujos compartidos.

4.3.7 SnapChat: una herramienta desarrollada con DyCE

Se estudió y analizó el comportamiento de la herramienta SnapChat desarrollada con DyCE. Nuevamente nuestro análisis estuvo centrado en cómo la herramienta maneja la consistencia de los datos y si provee soporte para la propiedad de latecoming, basando siempre nuestro análisis en ver cómo el framework permite manejar dichas propiedades.

Esta herramienta, permite la edición colaborativa de tres tipos de documentos, documentos HTML, documentos de texto y documentos gráficos. Cada usuario en la sesión puede crear nuevos documentos, editarlos y borrarlos. Para cada tipo de documento existen las herramientas correspondientes para su creación y edición.

De acuerdo a una serie de pruebas que se hicieron sobre la herramienta, comprobamos que ofrece dos tipos de mecanismos para garantizar consistencia de los datos, bloqueo simple implícito, es decir un bloqueo que es manejado internamente por la aplicación, y un bloqueo manejado directamente por el usuario, esto es, el encargado de bloquear los datos es el propio usuario, solo cuando decida liberar el recurso, el resto de los usuarios podrá modificar los datos compartidos.

Aquí se puede observar que el framework DyCE permite a los desarrolladores poder optar entre estas dos posibilidades, es decir, hacer que la aplicación que se desarrolla maneje los bloqueos automáticamente o permitir al usuario que sea el responsable de manipularlos.

El bloqueo implícito al que nos referimos, se utiliza en el caso que un usuario haya hecho algún comentario o marca sobre un objeto del documento (ya sea texto o gráfico), en ese caso, los otros usuarios en la sesión, solo pueden generar nuevas marcas sobre la que hizo el usuario que los creó inicialmente, pero no podrán borrarlas ni editarlas. La herramienta alerta a los usuarios indicando que no tienen permisos para borrar dicho objeto, solo podrá hacerlo la persona que los creó, como se muestra en la figura 4.9, y aquí podemos destacar que el framework también permite la posibilidad de asignar dueños (owners) a los objetos compartidos.

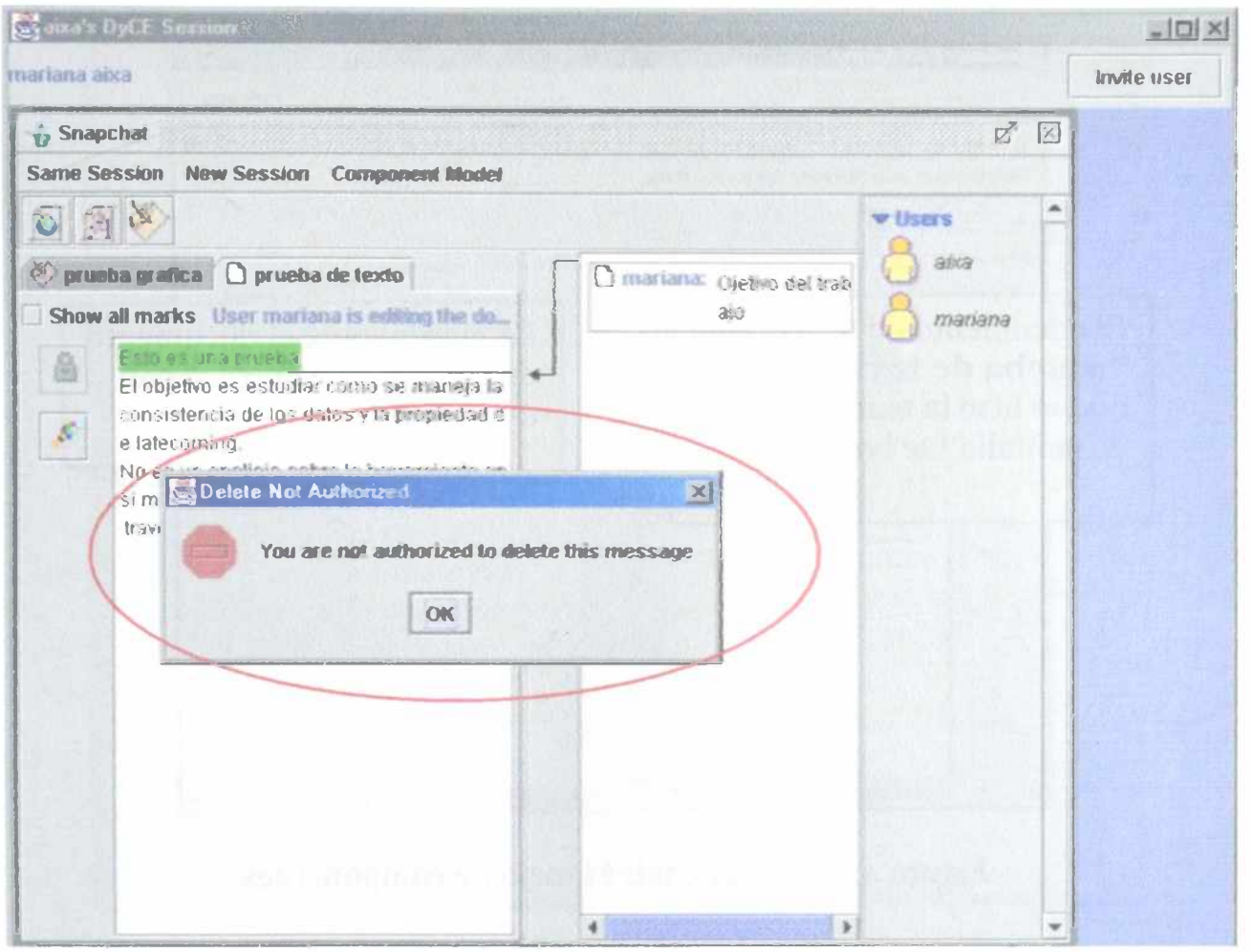


Figura 4.9 – Herramienta Snapchat: Manejo de permisos

Según lo que muestra la figura anterior el *usuario Aixa* intentó borrar una marca que había hecho previamente el *usuario Mariana*, dado que no fue quien creó la marca, la aplicación le informa que no está autorizada para borrar dicho mensaje.

Otra de las observaciones que notamos, es que en caso que se borre el objeto sobre el cual se hicieron comentarios o marcas, las mismas siguen estando, es decir se borra el objeto sin dificultad, sin embargo las marcan que se hicieron sobre dicho objeto siguen existiendo, cosa que no debería ocurrir.

Lo que muestra las Figura 4.10 es el resultado obtenido después de haber borrado un documento de texto llamado "*Prueba de texto*", sobre el cual anteriormente el *usuario mariana* había creado una marca. Como se puede observar entonces en la figura la marca aún permanece vigente a pesar que el objeto (documento de texto *"Prueba de texto"*) sobre el que se hizo dicho marca ya no exista.

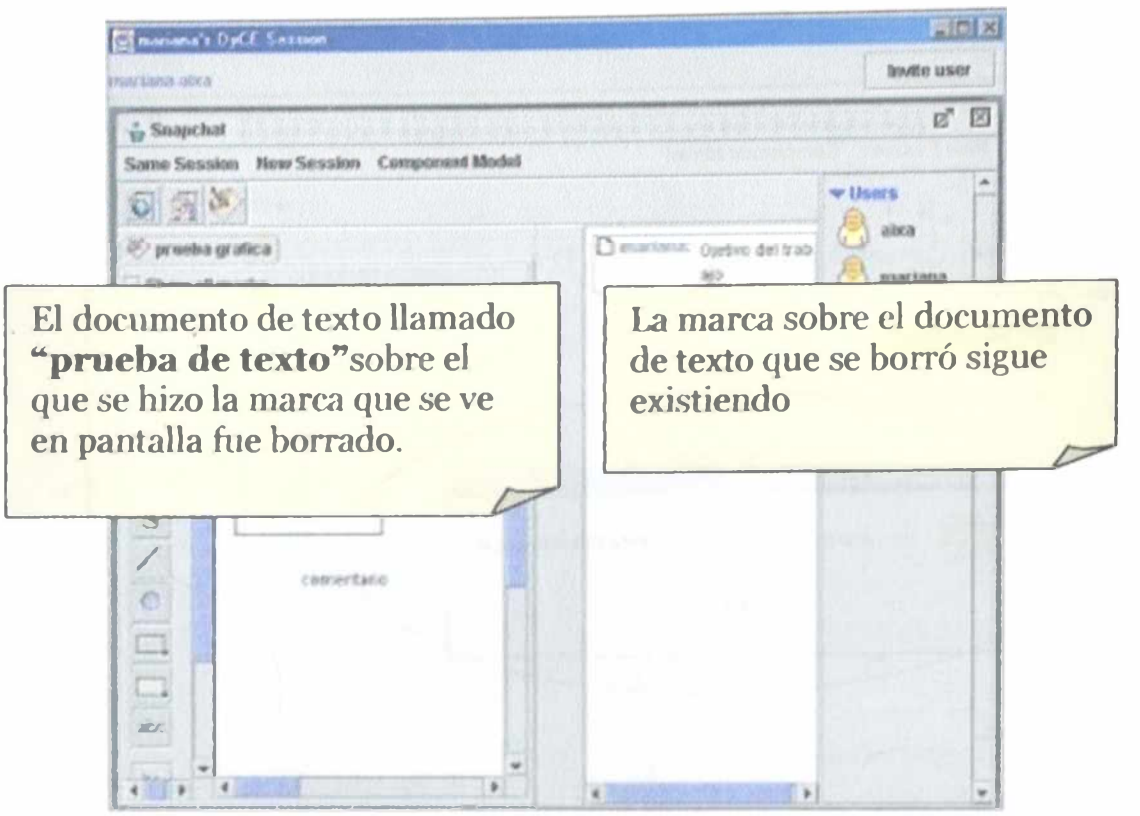


Figura 4.10 – Snapchat: **Manejo de componentes**

En este caso no se actualiza la información en su totalidad, por lo que la herramienta no maneja adecuadamente la consistencia de los datos. Esto se debe a que los componentes están claramente separados, y aquí surge el problema.

Quizás el objetivo central del framework, respecto de la separación entre los componentes, los objetos compartidos y las tareas exige un esfuerzo adicional en los desarrolladores, como se puede observar en el ejemplo anterior, es decir, si los desarrolladores no manejan cuidadosamente la consistencia de los datos, el framework deja espacios libres susceptibles a posibles errores.

Cualquier usuario puede borrar los objetos de un documento, como así también el documento mismo, aún cuando otros usuarios pueden estar editándolo. En este caso, la herramienta no bloquea los datos, según nuestro criterio esto no debería suceder.

Es importante que los datos sean bloqueados, o al menos informarle al usuario que intenta eliminar un objeto, que el mismo está siendo editado y que la operación no puede llevarse a cabo hasta que el otro usuario termine de editarlo, este conflicto surge dado que DyCE ofrece la posibilidad de replicar los datos en el cliente.

Respecto al bloqueo administrado por el usuario, esto es en el caso que se esté editando un documento de texto. La herramienta exige que los usuarios bloqueen y desbloqueen el documento, cada vez que quieran modificarlo. Esto se puede observar en la figura 4.11.

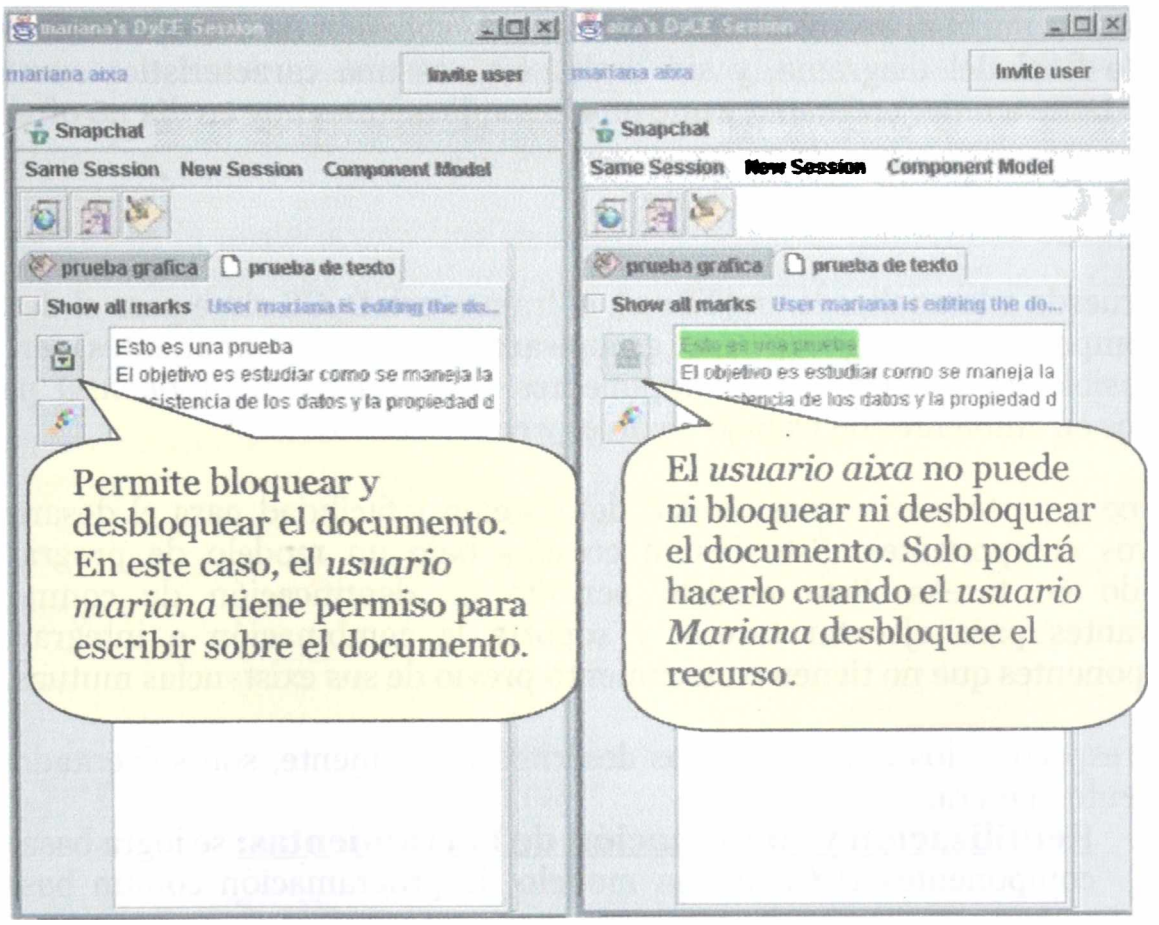


Figura 4.11 – Snapchat: Menejo de Bloqueos

En el ejemplo anterior, el *usuario Mariana* tiene control sobre el documento, cuando decida liberar el recurso deberá seleccionar la opción que lo desbloquea para que el *usuario Aixa* pueda escribir sobre el documento.

Es decir, para que un usuario pueda escribir sobre un documento éste debe estar desbloqueado, el usuario lo bloquea cuando escribe y lo debe desbloquear cuando termina de editarlo. Esto no es una buena opción, porque si el usuario olvida desbloquear el documento, no hay forma de que los otros usuarios le informen que está bloqueado porque no tienen acceso al mismo. Cualquier método que exige la intervención del usuario para asegurar la integridad de los datos es vulnerable a cualquier error del mismo.

Pero aquí solo cabe mencionar que el framework , como dijimos anteriormente, da al desarrollador la libertad de decidir cómo se manejarán los bloqueos dentro de la aplicación.

En lo que respecta a la propiedad de latecoming, la herramienta utiliza el método que muestra el estado final de la sesión.
Esto se pudo comprobar haciendo una serie de pruebas sobre un documento gráfico; inicialmente en la sesión había dos usuarios trabajando en forma colaborativa, luego de trabajar durante un tiempo considerable, un nuevo usuario se une a la sesión, e inmediatamente después de su ingreso pudo ver todo el trabajo que se hizo antes de su llegada a la sesión.

Evidentemente el framework permite guardar el estado final de una sesión colaborativa.

De esta forma el nuevo usuario no pudo ver la sucesión de eventos hasta llegar al estado final del diagrama, y sin dudas ésta es una característica sumamente importante a tener en cuenta y que muchas aplicaciones hoy en día no ofrecen.

4.3.8 Conclusión

De acuerdo a lo analizado en [TR02], el framework DyCE sirve para el desarrollo de component-based groupware (groupware basado en componentes), en donde es posible un desarrollo de componentes (*Groupware Components*) para ser usados en ambientes de trabajo flexibles y colaborativos.

Provee soporte para extensibilidad del sistema y facilidad para el desarrollo de nuevos componentes. Éstos están creados bajo un modelo de programación basado en tareas. Este modelo permite la identificación de componentes relevantes para ejecutar tareas, y soporta la combinación e integración de componentes que no tienen conocimiento previo de sus existencias mutuas.

Con respecto a los requerimientos descritos inicialmente, son solventados de la siguiente manera:

- **Reutilización y combinación de herramientas:** se logra basando los componentes DyCE en un modelo de programación común basado en tareas.
- **Extensibilidad del sistema en tiempo de ejecución:** se logra mediante el repositorio de componentes, en donde se cargan nuevos componentes y desde donde se descargan posteriormente para ser instanciados en el Groupware Desktop.
- **Independencia de la plataforma:** es lograda basando DyCE en Java como lenguaje de programación. Los clientes pueden ser distribuidos a través de Internet/Intranet embebidos dentro de páginas Web, actuando como servidor de aplicaciones el mismo Server http de DyCE. Se soporta tanto comunicación sincrónica como asincrónica. Los objetos compartidos son persistentes automáticamente y pueden ser accedidos desde almacenamiento persistente cuando hace falta, para lo cual se utiliza cualquiera de los modelos de comunicación nombrados. El trabajo de los usuarios simples, como el trabajo en grupo, así también la interacción entre ellos son soportados por el modelo de sesión, siendo la sesión la manera en que se maneja por ejemplo el **latecoming.** Las tareas asociadas a la sesión son ejecutadas en cada nuevo cliente que se asocia a dicha sesión.

Sin dudas DyCE ofrece una excelente arquitectura desde el punto de vista de la separación de los componentes, los objetos compartidos y las tareas.

Permite a los desarrolladores manipular los componentes, y el modelo de datos de cada una de ellas de manera totalmente independiente, lo que hace a un framework muy flexible y extensible.

Sin embargo, dado que su arquitectura puede ser centralizada o replicada exige de ciertos esfuerzos. Es decir, si se decide implementar una arquitectura centralizada, se deben bloquear los datos compartidos o hacer que el usuario sea el responsable de manipular dichos bloqueos, lo que hace que el sistema sea

vulnerable a los errores del usuario; en cambio, si se decide utilizar una arquitectura replicada, pueden surgir algunas inconsistencias temporales como se mencionó en la sección anterior, pero en este caso se utiliza la transformación de operaciones para garantizar consistencia de información.

Cualquiera sea la arquitectura que se decida utilizar, el framework obliga a los desarrolladores a ser muy cuidadosos en el manejo de la consistencia de datos. Ya que algún descuido, como se explicó en la sección 4.3.7, puede generar posibles inconsistencias.

No ofrece inconvenientes respecto de la propiedad de latecoming, ya que informa a los nuevos usuarios del estado final de la sesión al momento de su ingreso a la misma, lo que se debe tener en cuenta aquí es que no ofrece la posibilidad de mostrar la sucesión de eventos hasta llegar al estado final de una sesión colaborativa.

4.4 Servidor Sametime

4.4.1 Introducción

Sametime formalmente llamado IBM Lotus Instant Messaging & Web Conferencing, es un software de Lotus para la colaboración en grupo en internet.

Sametime fue diseñado para facilitar la comunicación entre colaboradores que se encuentran geográficamente distribuidos. El grupo de productos de Sametime incluye, el Servidor de Sametime, un soporte para la conexión al cliente, y herramientas para el desarrollo de aplicaciones.

El propósito de los productos que brindan colaboración en tiempo real, es aproximarse tanto como sea posible, a la experiencia que se tiene hoy en día respecto de las reuniones cara a cara.

De acuerdo a Lotus, Sametieme fue desarrollado en base a tres componentes esenciales para toda aplicación colaborativa en tiempo real exitosa: awareness, facilidades en la conversación, y la habilidad de compartir objetos.

Las aplicaciones que están disponibles con Sametime proveen capacidades de colaboración en tiempo real a través de los servicios Community (comunidad) y Meeting (reunión).

Básicamente los usuarios se loguean en el servidor de Sametime, de esta manera pueden ver quienes están on-line, comunicarse y trabajar juntos a través de las reuniones programadas o de forma instantánea.

4.4.2 Servicio Community

Los servicios community de Sametime proporcionan toda la funcionalidad relacionada con la comunidad, tal como conocimiento de la gente (awareness), espacios, conocimiento basado en espacios, mensajería instantánea, y charla (chat).

4.4.3 Arquitectura de Sametime

Sametieme define una arquitectura cliente-servidor. Los clientes en sametime son el Sametime Connect y el Meeting Room Client (MRC).
Todas las comunicaciones cliente-cliente tales como la mensajería instantánea, pasan a través del servidor de Sametime.

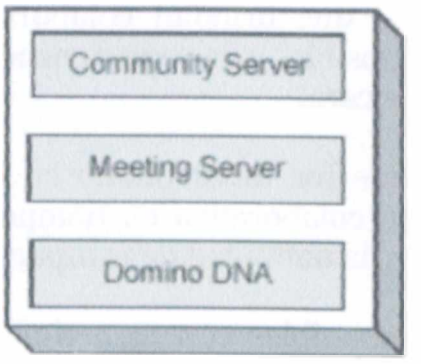
Una vez que el usuario se loguea en el servidor de sametime, puede acceder a todos los servicios y comunicarse con el resto de los usuarios logueados en el servidor.

El servidor de Sametime esta formado por tres servidores que interactúan:

El Community server – Provee los servicios tales como login y awareness.

El Meeting Server – Proporciona servicios tales como pantallas compartidas y, sonido y video a través de la IP.

El Domino™ DNA – Provee accesos a los directorios, autenticación y un servidor http.



Sametime Server

Figura 4.12 – Servidor Sametime

A continuación se describirán los siguientes temas respecto de la arquitectura del servidor de sametime

- El modelo de usuario del Sametime community
- La estructura del servidor de Sametime
- Comunicaciones
- Distribución, escalabilidad y redundancia

- Soluciones multi-server

4.4.3.1 Modelo de usuario del Sametime community

El modelo de login y de usuario soporta varios usuarios ejecutando multiples aplicaciones concurrentemente. El modelo de usuario se divide en componentes persistentes y en tiempo de ejecución.

- **Datos persistentes del usuario**

  Cada usuario tiene las siguientes propiedades básicas:

  Id de usuario – Un string que representa el identificador único y persistente de un usuario en la comunidad.

  Parametros de login – campos para registrar la identificación del usuario en la comunidad, como ser un nombre de usuario y una clave.

  Nombre de usuario – Es el nombre por el cual un usuario es identificado en la comunidad por otros usuarios. Un usuario puede tener mas de un nombre para diferentes logins.

  Descripcion – Descripción del usuario.

  Lista de privacidad – Una lista de Ids de usuarios que puede o no conocer si el usuario esta online. Esta lista se mantiene por usuario. Si un usuario se registra varias veces simultáneamente, el servidor sincroniza la lista de privacidad del usuario entre las diversas conexiones.

  El modelo antedicho permite que un usuario tenga nombres múltiples y aun así sea conocido como el mismo usuario por otros miembros de la comunidad sin importar el nombre que utilice en las distintas conexiones.

- **Estructuras runtime del usuario**

  El modelo runtime de usuario permite que cada usuario cree múltiples logins concurrentemente en la comunidad.

  Cada conexión TCP/IP de un usuario es mirada como un simple login del usuario en la comunidad. El ID del login es único en la comunidad y es asignado a cada conexión.

  Aunque todas las aplicaciones de un usuario pueden usar el mismo login, los múltiples logins son necesarios cuando se ejecutan aplicaciones no colaborativas. Actualmente las comunidades del sametime limitan los logins del usuario a una simple maquina (simple dirección IP).

- **Huespedes**

  Un huésped (guest) es un usuario no autenticado en la comunidad sametime. El administrador de la comunidad puede configurar que servicios están disponibles para los huéspedes del sistema. Un huésped no tiene un ID de usuario persistente, únicamente el ID de usuario es utilizado para identificar a un usuario huésped. Además no hay manera de distinguir los diferentes logins de un mismo usuario huésped.

- **Identificadores Inter-comunidad**

  Para permitir interacciones entre las comunidades, son necesarios un identificador único global de usuario y de logins. Por lo tanto se define un ID de comunidad para cada comunidad. La unión del ID de la comunidad al los Ids del usuario y el login asegura su unicidad entre las comunidades.

4.4.3.2 Estructura del servidor de Sametime

La comunidad de Sametime esta compuesta de las siguientes capas. Las capas son listadas de abajo hacia arriba:

Clientes – Un programa registra a un usuario como usuario de la comunidad.
Multiplexores – Mediadores entre los clientes y el servidor.
Servidor – El servidor base de la comunidad.
Aplicaciones del servidor – Servicios fuentes agregados a la comunidad.

Una comunidad es una colección de tales servidores, ya sean ambientes distribuidos (WAN) o escalables (LAN).

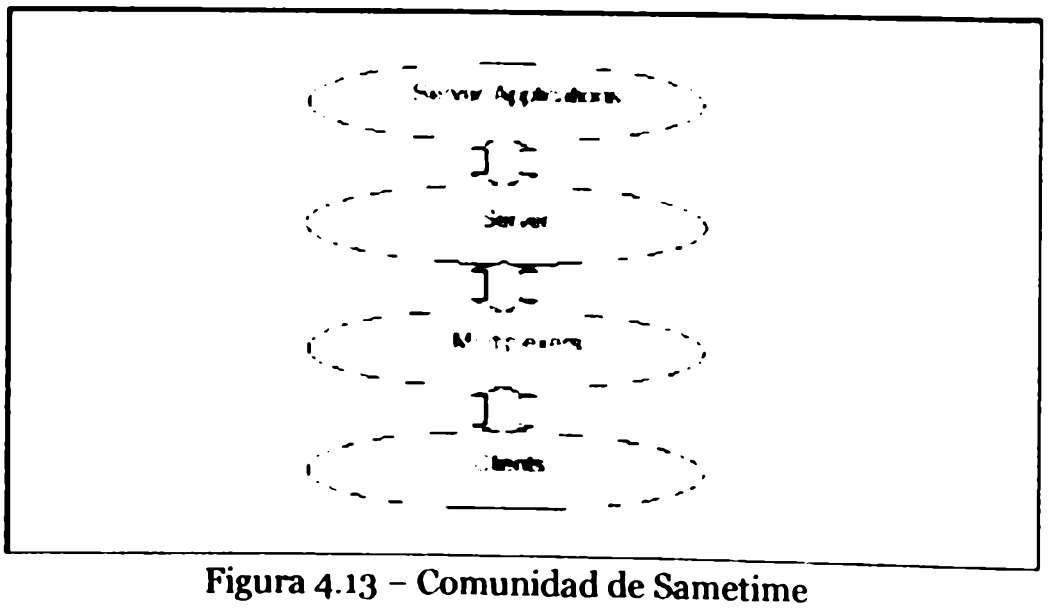


Figura 4.13 – Comunidad de Sametime

- **Capa Multiplexor**

  Los multiplexores son concentradores de I/O que permiten servidores altamente eficientes y escalables por lo siguiente:

  - ✓ **Concentracion I/O**

    Concentra las I/O de los clientes a través del servidor Sametime. Esta concentración limita las sobrecargas de los servidores puesto que tiene que enviar y recibir los datos sobre un número pequeño de conexiones. Por lo tanto la inicialización de la sobrecarga de la conexión, el pooling y la terminación son muy reducidas.

  - ✓ **Distribucion de multiples mensajes recibidos**

    El servidor puede enviar una sola copia de un mensaje a su multiplexor. El multiplexor entonces distribuye el mensaje a los recipientes indicados.

  - ✓ **Gateway**

    Los multiplexores pueden actuar como un gateway, traducen protocolos entre los clientes o servidores de terceras partes y el servidor de Sametime.

La capa multiplexor es transparente al cliente que se conecta a la comunidad a través de estos. Los multiplexores no pueden conectarse directamente al servidor pero se conectan con otro multiplexor a través de un canal de multiplexores.

- **El servidor base**

  Es responsable de lo siguiente:

  - ✓ Administrar los miembros de la comunidad (usuarios, login, aplicaciones del servidor y los multiplexores)
  - ✓ Rutear los mensajes
  - ✓ Proveer notificaciones a los miembros de la comunidad.

- **Capa de aplicación del servidor**

  Los usos del servidor realzan la funcionalidad provista por el servidor. Una aplicación del servidor se conecta a un servidor y declara los servicios que este provee. El servidor encamina los requerimientos para tales servicios a la instancia adecuada de las aplicaciones del servidor

- **Esquema de la comunidad de Sametime (Sametime Community)**

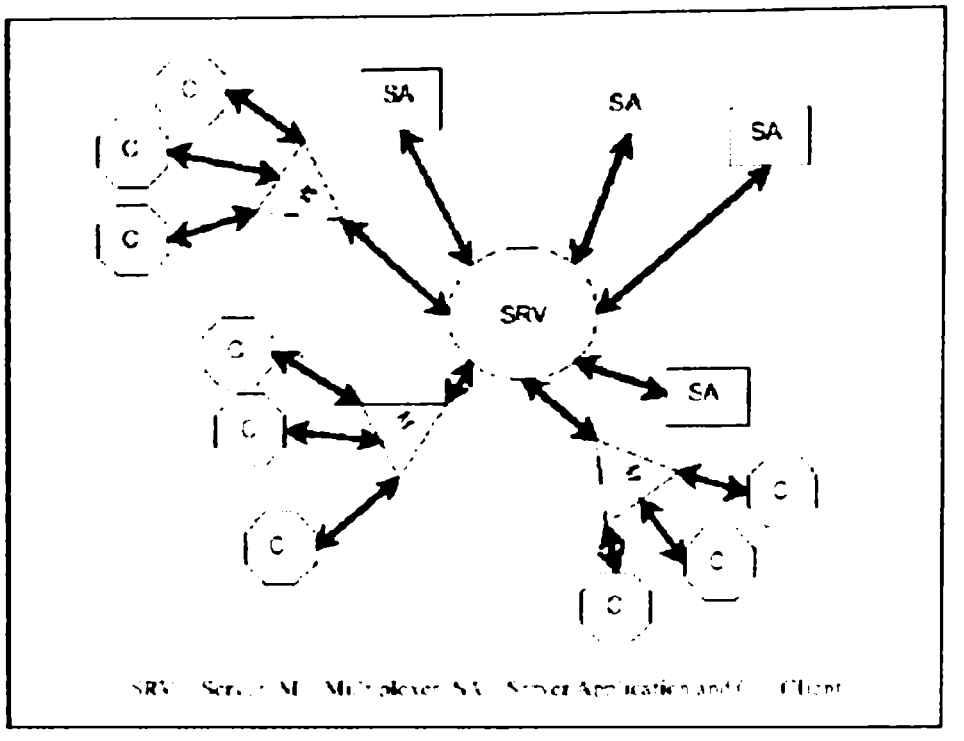


Figura 4.14 – Esquema de la comunidad de Sametime (Sametime Community)

Tanto el servidor como los multiplexores y las aplicaciones del servidor están conectados vía una conexión TCP/IP. Los clientes también se conectan a uno de los multiplexores vía una conexión TCP/IP. El servidor asigna los Ids de login de los usuarios y maneja las propiedades de cada miembro de la comunidad conectado a este.
La autenticación de los usuarios se hace vía la aplicación del server que provee el servicio de autenticación. La aplicación del servidor trabaja con la BD, mientras que el servidor en si no puede acceder directamente a ella.

4.4.3.3 Comunicacion en Sametime

- **Modos de direccionamiento**

  Hay varias maneras de dirigirse a otros miembros de la comunidad:

  - ✓ *ID de Usuario* – Mediante el id de usuario se dirige a algún usuario. Si el usuario receptor tiene mas de un login simple, uno de estos logins es seleccionado.
  - ✓ *ID de login* – Se dirige a un login en particular. En este caso no hace falta la selección de logins.
  - ✓ *Tipo de Servicio* – Se dirige a un tipo de servicio especifico que provee una aplicación del servidor. Puede haber múltiples abastecederos para el servicio solicitado. El servidor tendrá que decidir a que abastercedor servirá la petición.

- **Alcances de direccionamiento**

  El tipo mas frecuente de interacción en una comunidad es uno a uno. Los mensajes son intercambiados ente dos miembros de la comunidad.

Además de lo anterior hay tipos de interacciones de múltiples receptores que son candidatos a realizar optimizaciones.
Sametime provee los siguientes métodos para tales interacciones:

- ✓ *Uno a muchos* – El mensaje es recibido por varios receptores. El servidor distribuye una copia simple del mensaje al multiplexor implicado, mientras que el multiplexor distribuye el mensaje a todos los logins correspondientes.

- ✓ *Broadcast* – El mensaje es redireccionado a todos los miembros de la comunidad. El servidor envía una copia del mensaje redireccionado a cada multiplexor contectado a este; cada multiplexor reenvía los mensajes a sus logins.

- ❑ **Canales**

  El envió de mensajes se hace generalmente a través de conexiones virtuales llamadas "canales". Un canal puede pasar a través de varias conexiones TCP/IP. Por ejemplo cuando un cliente en un multiplexor A crea un canal sobre un multiplexor B, el canal atraviesa las siguientes conexiones TCP/IP:

  - ✓ Desde el login de usuario al multiplexor A
  - ✓ Desde el multiplexor A al servidor
  - ✓ Desde el servidor al multiplexor B
  - ✓ Desde el multiplexor B al login del otro usuario.

El canal garantiza el orden de los mensajes en una interacción y garantiza la conectividad proveyendo una notificación cuando el ruteo del canal se quebró. Ambos canales participantes reciben la notificación.

4.4.3.4 Distribución, escalabilidad y redundancia

De acuerdo a lo discutido anteriormente, la comunidad de Sametime es provista por un único servidor de base. Esta solución es adecuada para muchas comunidades. El servidor de Sametime fue diseñado para atender un gran numero de logins simultáneamente. Sin embargo, un servidor simple no puedo direccionar adecuadamente en las siguientes situaciones.

- ❑ **Distribución**

  Los usuarios de una comunidad pueden estar distribuidos en sitios geográficamente remotos. En un ambiente con un único servidor central, los clientes remotos dependen de las cualidades y condiciones de la red entre su sitio y el del servidor.
  Esto significa que si el camino de la red al servidor está caído temporalmente, los usuarios remotos no pueden conseguir servicios para actuar recíprocamente aun con usuarios cercanos. De esta manera la utilización no es optima, con lo cual se degrada su performance.

- **Escalabilidad**

  Un único servidor central tiene sus grandes limites. Estos límites son válidos para muchas organizaciones, pero no cubren las expectativas de las grandes organizaciones ni en cuanto a los servicios que brindan. Sametime ofrece una solución a este problema vía la implementación de un multi-server.

- **Redundancia**

  La dependencia que se genera cuando se trabaja con una sola maquina como servidor, es aun hoy en día un problema. Los usuarios deberían poder conectarse a la comunidad aun cuando algunos servidores están caídos o en mantenimiento.

4.4.4 Soluciones multi-servidor.

En la figura 4.9 se muestra un esquema de la comunidad de Sametime multi-servidor. Notar que el esquema solo cambia respecto del esquema de un único servidor, en que hay varios servidores conectados entre si. De esta manera los usuarios no son afectados en caso de que surja algún inconveniente en la red, dado que están conectados a un único server o a múltiples servidores.

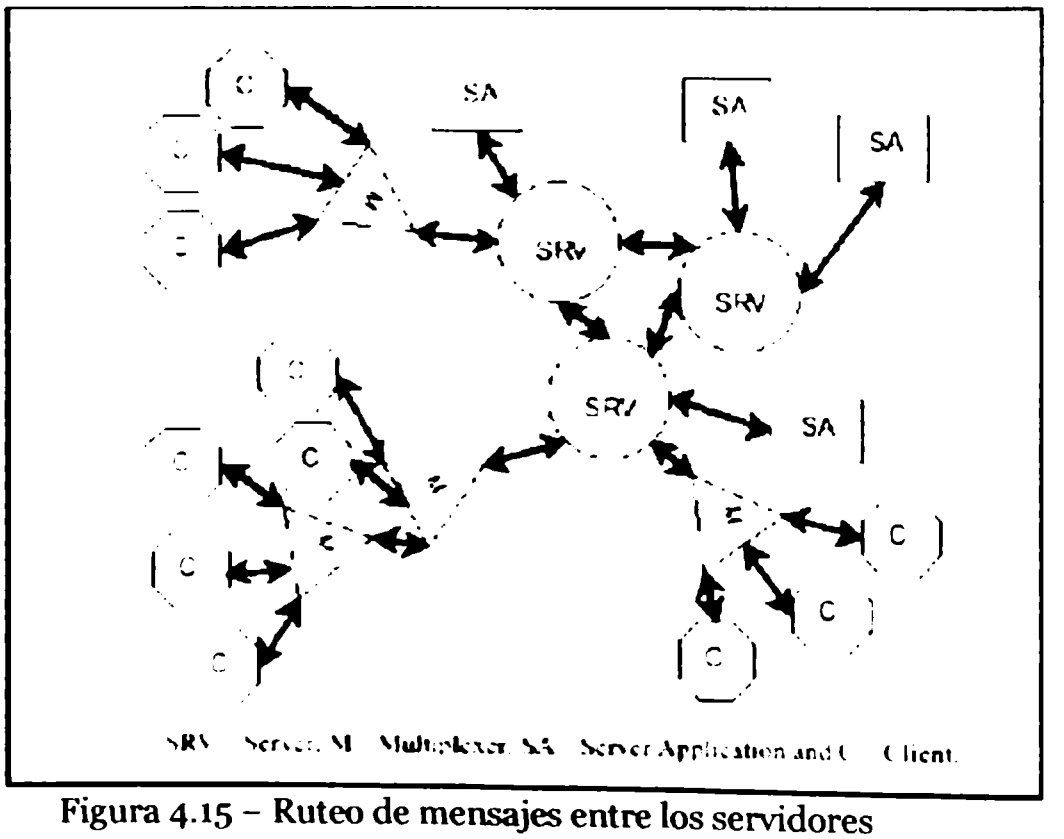


Figura 4.15 – Ruteo de mensajes entre los servidores

Sin embargo varios problemas tecnológicos deben ser tratados en una arquitectura multi-servidor, tales como el envió de mensajes, el direccionamiento de los miembros en una comunidad y mas. A continuación se describen tales problemas.

- **Routing (Ruteo)**

  Un canal o un mensaje destinado a otro miembro de la comunidad puede atravesar por varios servidores hasta llegar al destino final, por tal motivo es necesario definir una ruta.

  En las actuales versiones de Sametime evitamos este problema conectando todos los servidores entre si. Este tipo de conexión es conocida como "clique (pandilla)". La ruta entre cualquier miembro de la comunidad se hace a través de a lo sumo dos servidores: el servidor al cual el primer miembro esta conectado y el servidor en el cual el segundo miembro esta conectado.

- **Direccionamiento**

  Utilizando un único servidor, cuando un miembro es dirigido hacia un ID de usuario, o a un login de usuario o a un tipo de servicio, el servidor rutea el mensaje al miembro de la comunidad adecuado de acuerdo a sus tablas locales. En un ambiente multi-servidor esto no es posible.
  Existen diversas soluciones a este problema.
  La solución implementada en Sametime es la siguiente:

  - ✓ Las aplicaciones del servidor mantienen conocimiento global de toda la comunidad. Esta aplicación es llamada on-line directory (OLD). Cada servidor tiene una instancia del OLD. El directorio on-line reúne y mantiene el mínimo de datos necesarios para localizar las entidades on-line. (por ejemplo los usuarios) en toda la comunidad.

  - ✓ El servidor notifica a tu OLD sobre todos los servidores que hay en la comunidad. El OLD crea un canal a todos los otros OLD que existen en la comunidad, y obtiene el estado actual y las subsecuentes actualizaciones de los mismos.

  - ✓ Eventualmente todos los OLD replican sus datos a cada uno de los otros OLD. Cuando un servidor tiene que localizar a un miembro de la comunidad por su ID de usuario, simplemente consulta a su OLD.

  - ✓ Cuando un miembro es dirigido a través de su ID de login, el servicio de OLD no es necesario, dado que el ID de login contiene la dirección IP del servidor en el cual dicho login reside.

- **Almacenamiento persistente**

  Las propiedades de cada usuario son mantenidas en un almacenamiento persistente que esta asociado con cada servidor, este almacenamiento es replicado hacia todos los servidores.
  Desde un ambiente distribuido geográficamente no es posible replicar los datos en tiempo real, un usuario que cambia sus propiedades en un servidor y luego se loguea en un servidor diferente podrá tomar los datos después de que estos hayan sido registrados, es decir la actualización de los cambios no es inmediata.

Para resolver este problema, se define un servidor local a cada usuario en la comunidad. Esto se hace relacionando la dirección de un servidor a cada usuario registrado en la comunidad. Este servidor será el servidor local para el usuario que lo tenga asignado.

El cliente de un programa puede acceder al servidor local desde su almacenamiento local o conectándose al mismo servidor en la comunidad.

- **Escalabilidad y redundancia**

  Una comunidad con la propiedad de multi-servidor como describimos anteriormente puede resolver el problema de distribución.

  Sin embargo, los problemas de escalabilidad y redundancia no son resueltos aun.
  Para solucionar este problema se introduce el concepto de cluster. Un cluster es un conjunto de servidores donde todos los servidor en el cluster pueden comportarse como un servidor local de todos los usuarios del cluster. Es decir, se define un cluster local para los usuarios del cluster.

4.4.5 Kit de desarrollo de Sametime 3.0

El kit de desarrollo JAVA de Sametime 3.0 es de gran utilidad. Proporciona accesos a todos los servicios que provee Sametime, tales como awareness, mensajería instantánea e interfases compartidas. Los componentes de alto nivel AWT (Abstract Windowing Toolkit) disponibles en Sametime pueden estar embebidos dentro de cualquier contenedor AWT
El kit de desarrollo presenta una arquitectura en tres capas:

Capa de Transporte (Transport Layer) – Toda la comunicación con el servidor de Sametime pasa a través de esta capa.

Capa de servicios (Service Layer) – Provee acceso a los servicios del Sametime Community y Meeting.

Capa de Interfaces de Usuario (UI Layer) – Provee componentes UI.

La kit de herramientas para el desarrollo de aplicaciones colaborativas es modular, seguro, y extensible, proporcionando una API orientada a objeto.

El kit de desarrollo de Sametime 3.0 esta basado en componentes y los diferentes servicios que provee están divididos mediante estos componentes.

El desarrollador tiene un completo control sobre los componentes almacenados basados en los servicios Sametime requeridos por la aplicación.

El desarrollador solo escribe el código para solamente cargar los componentes que proporcionan estos servicios. Esta estructura modular da lugar a desarrollos

más pequeños y a un tiempo más corto de la transferencia directa para el usuario final.

En el kit de desarrollo provisto por Sametime los desarrolladores pueden agregar nuevas componentes que provean nuevos servicios y nuevas interfaces de usuario (UI) lo que hace que el kit de desarrollo pueda ser extendido ya sea por los desarrolladores de Sametime o cualquier otro desarrollador ajeno a Sametime.

Aquí no se describe como desarrollar tales componentes.
Lo que debe quedar claro es que el kit de desarrollo da la posibilidad de crear nuevas componentes.

Para mas información remitirse a *Working with the Sametime Community Server Toolkit* donde se explica en detalle el kit de desarrollo explicado anteriormente.

4.4.6 Conclusión

Sametime fue diseñado especialmente para facilitar la comunicación entre colaboradores, que se encuentran geográficamente distribuidos.

Básicamente, define una arquitectura a nivel comunicación. Es decir, ofrece un modelo de comunicación óptimo, para cualquier aplicación colaborativa ya diseñada.

El servidor Sametime está compuesto por tres servidores más pequeños, uno de ellos es el servidor de aplicaciones propiamente dicho, donde residirán las aplicaciones colaborativas.

Su mayor contribución, radica en mejorar los tiempos de respuestas en la interacción usuario-usuario, mediante el pasaje de mensajes.

También ofrece un buen administrador de usuarios (se acerca a los conceptos de Servidores LDAP (Lightweight Directory Access Protocol) para el repositorio de usuarios autenticados), introduciendo los conceptos de comunidad, usuario autenticado y huésped. Cada usuario autenticado tendrá un id de usuario único en el servidor y podrá disponer de varios nombres de login si así lo deseara para las diferentes aplicaciones en las que participe. La diferencia con los huéspedes, es que éstos no están autenticados en el servidor de manera persistente, con lo cual no tendrán asignados un id de usuario, solo tendrán un id de login para identificarse dentro de una comunidad de manera temporal.

Además, Sametieme, pone a disposición de los usuarios un kit de desarrollo JAVA, ofrece componentes y servicios para poder administrarlos, por lo que el desarrollador solo deberá agregar código específico para poder utilizarlos.

El análisis de Sametime 3.0 fue desarrollado sobre la base de la siguiente documentación oficial de Lotus, IBM Corporation:

- *Working with the Sametime cliente toolkits*
- *Working with the Sametime Community Server Toolkit*
- *Sametime 3.0 Java Toolkit*

Esta información esta disponible en http://www.ibm.redbooks.com

4.5 Conclusión

De acuerdo a lo expuesto en este capítulo, podemos hacer una buena distinción entre los dos frameworks analizados mediante el siguiente cuadro comparativo:

	COAST	DyCE
Arquitectura	Cliente-Servidor Replicada	Cliente-Servidor Centralizada o Replicada
Método para el control de concurrencia (manejo de Consistencia)	Transformación de operaciones	Bloqueos Simples, en algunos casos manejados por el usuario. Transformación de operaciones.
Método de Latecoming	Estado final	Estado final
Ventajas	Las operaciones se llevan a cabo localmente. No hay retardo en la ejecución de las acciones del usuario. No hay bloqueos	Buena separación entre componentes, objetos compartidos (modelo de datos) y tareas. Respuesta rápida.
Desventajas	Inconsistencias temporales	Inconsistencias temporales. Vulnerable a errores del usuario.

Hay un factor común, que se cumple en ambos frameworks respecto de la propiedad de latecoming, los dos implementan el método que muestra el estado final cuando un usuario rezagado (latecomer) se incorpora tarde a la sesión colaborativa.

Cabe destacar la importancia, de poder mostrar la sucesión de eventos que conformaron el estado final de un trabajo compartido.
Por ejemplo, supongamos el caso de la generación de diagramas UML que ofrece la herramienta UML-Editor desarrollada con COAST. Los usuarios pueden estar interesados en saber cómo fue el proceso de desarrollo hasta llegar al estado final de un diagrama de clases, cuales fueron los pasos que siguieron los usuarios, si se cambió el diseño, y cuáles fueron las razones, esto permitiría tener un análisis más exhaustivo sobre un documento compartido.
Esto es, porque todo trabajo en grupo, requiere que los participantes expongan sus diferentes puntos de vista sobre una tarea en común.
Dar la posibilidad al usuario, de observar los cambios de manera detallada es imprescindible, dado que es de suma importancia para un usuario poder tener información de los cambios realizados por parte de otros usuarios en el orden que fueron sucediendo durante su ausencia en la sesión de trabajo compartido, en caso de incorporarse tarde a la sesión de trabajo.

En lo que respecta a los métodos utilizados para mantener consistentes los datos, hay varias cosas a tener en cuenta.
Si se trata de una arquitectura totalmente replicada, pueden surgir posibles inconsistencias temporales, y cabe destacar que no es posible utilizar bloqueos para que esto no suceda, dado que hay ciertas operaciones que se ejecutan localmente en el cliente, esto es bueno, ya que no hay retardos en la ejecución de las operaciones y un buen método para lograr eso es la transformación de operaciones, pero se pueden generar inconsistencias temporales en los datos.

Si se trata de una arquitectura centralizada es imprescindible utilizar bloqueos para evitar inconsistencia en los datos, debido a que los usuarios acceden al mismo recurso en el mismo instante de tiempo. Se debe contar entonces con un mecanismo que restrinja el acceso, para que los usuarios siempre vean los datos actualizados con las últimas modificaciones. Sin embargo, esto debe tratar con los problemas ya conocidos de cuello de botella y un único punto de fracaso, lo que exige un esfuerzo extra para contemplar ese tipo de fallas; además el hecho de bloquear los datos genera retardos en los tiempos de respuesta.

La cuestión es tratar de pensar en una arquitectura que tome las ventajas de las arquitecturas replicadas y centralizadas. Como se explicó en el capítulo 3, es lo que se conoce como arquitectura híbrida.

Sin embargo, hoy en día no hay demasiadas aplicaciones colaborativas que se basen en este tipo de arquitectura, ni frameworks que la hayan puesto en práctica.
No es un camino sencillo de implementar, pero puede ser una buen punto de partida para lograr que las aplicaciones colaborativas ofrezcan vistas actualizadas de manera inmediata, evitando inconsistencias temporales. Como así también ofrecer al usuario la posibilidad de llegar tarde a una sesión de trabajo compartido, obteniendo información actualizada de lo que se hizo minutos antes de su llegada sin afectar el trabajo del resto de los usuarios en la sesión.

Capítulo 5

Modelo de Arquitectura Propuesto

Como se ha dicho en varias oportunidades durante el desarrollo de este trabajo de grado, la construcción de aplicaciones colaborativas sincrónicas no es una tarea sencilla.
Por esta razón, muchos trabajos de investigación se han centrado en resolver problemas puntuales, debido a que el desarrollo de estas aplicaciones abarcan diversas áreas de investigación.

Específicamente nuestro trabajo estuvo orientado en resolver dos problemas puntuales: en primer lugar dar solución a la inconsistencia de información que actualmente afecta a las aplicaciones colaborativas sincrónicas, lo que interfiere y dificulta el trabajo en grupo; y en segundo lugar, permitir la incorporación tardía de un nuevo usuario (latecomer) a una sesión de trabajo compartida, ofreciendo la posibilidad de informarle la secuencia detallada de los eventos hasta llegar al estado final de la sesión, lo que se conoce como propiedad de latecoming.

Para dar solución a lo detallado anteriormente, se han analizado específicamente dos frameworks, DyCE y COAST (ver capítulo 4), ambos intentan ofrecer ciertas facilidades a la hora de construir tales aplicaciones. Cada uno define una arquitectura a seguir basándose en los modelos ya conocidos, básicamente en los tres más utilizados para la construcción de este tipo de aplicaciones: como vimos ellos son, el modelo centralizado, replicado e híbrido analizados en el capítulo 3.

En el caso particular de DyCE, se define una arquitectura replicada o centralizada, donde hay una clara separación entre los componentes (pizarra colaborativa, chat, editor gráfico, etc) y los objetos, es decir, entre los componentes y el contenido que es manipulado por cada uno de ellos; por ejemplo, el componente Pizarra manipula objetos gráficos como ser círculos, cuadrados, etc. Esto es una gran ventaja, ya que cualquier cambio que se desee hacer sobre los componentes o los datos es totalmente independiente uno del otro, lo que ofrece una gran flexibilidad, sin embargo, vimos que hubo ciertos problemas de inconsistencia en la información, o sea, en los objetos que son manejados a través de los componentes. Cabe destacar también que DyCE ofrece replicación tanto de contenido como de componentes.

Por su parte el framework COAST define una arquitectura replicada, donde los objetos y los procesos de aplicación son totalmente replicados en el cliente, los cuales son sincronizados mediante el administrador de replicación. Sin embargo esa sincronización no fue suficiente para garantizar consistencia en los datos.

En lo que respecta a la propiedad de latecoming, tanto DyCe como COAST, sólo ofrecen la posibilidad de mostrar el estado final de una sesión colaborativa, tema que fue tratado en la sección 4.5, donde se explicó la necesidad de ofrecer al usuario la posibilidad de ver la suseción de eventos hasta la llegar al estado final de un trabajo compartido.

Como se dijo antes, ambos frameworks utilizan el modelo de arquitectura replicada y por su parte DyCE ofrece la posibilidad de centralizar los datos, sin embargo es difícil encontrar hoy en día aplicaciones que utilicen las ventajas tanto de la arquitectura replicada como de la centralizada (arquitectura híbrida).

Por tal motivo, nuestro modelo de arquitectura propuesto está basado en el modelo de arquitectura híbrida, tomando además las ventajas de los frameworks analizados y mejorando sus desventajas.

El modelo propuesto fue diseñado fundamentalmente sobre los conceptos básicos de DyCE, tomando como principal objetivo la separación entre componentes y contenido, principalmente porque como se dijo antes, los problemas de inconsistencias radican en los objetos que son manipulados por los componentes y no en los componentes propiamente dichos.

Como muestra la figura 5.1, la línea vertical punteada, representa los límites de la arquitectura propuesta. Es decir, el objetivo central de este trabajo de grado será definir un modelo de arquitectura para el contenido de componentes colaborativas. El modelo de objetos fue extraído de un ejemplo concreto citado en la sección 4.4 durante el análisis del framework DyCE.
Como se puede ver en el ejemplo, se define un componente editor de diagramas (clase DiagramEditor) y el contenido del componente será un diagrama (clase Diagram).

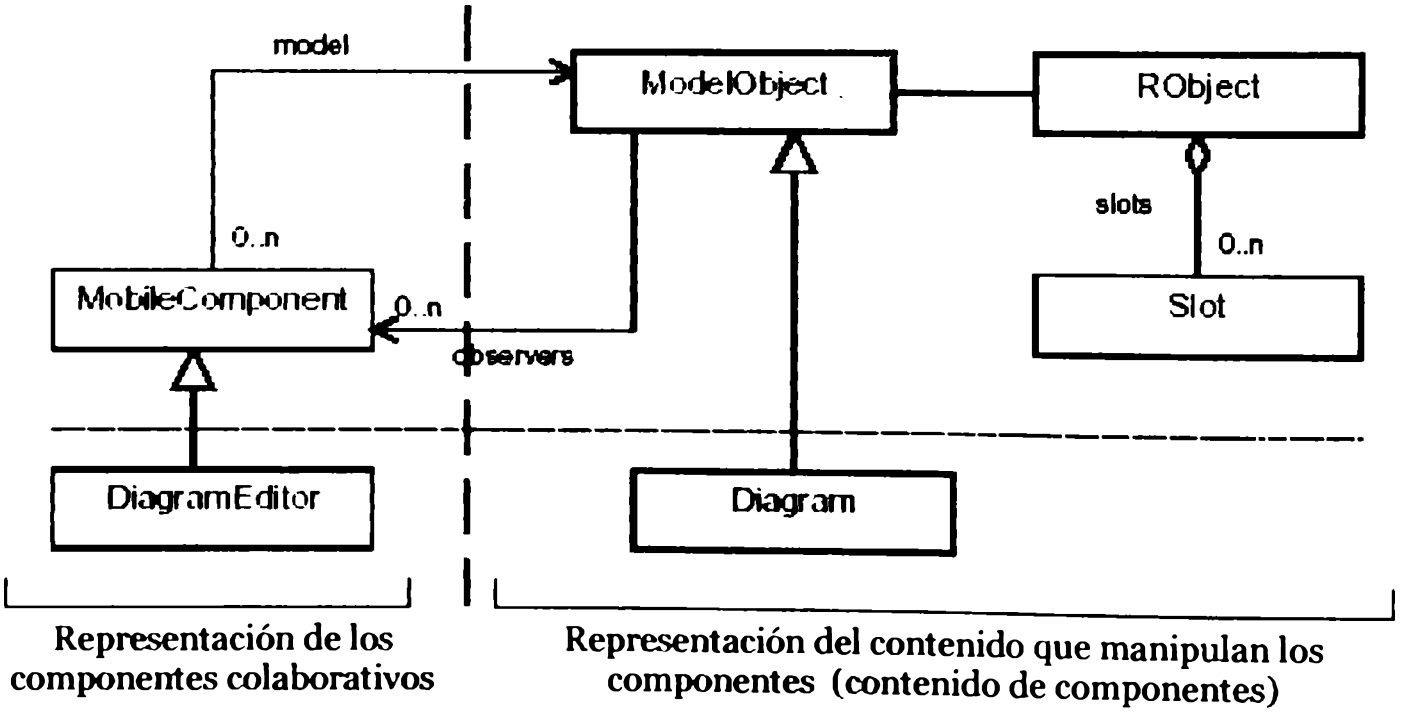


Figura 5.1 – Modelo de objetos DyCE: separación entre componentes y contenido

Básicamente centramos nuestro análisis en cómo manipular este contenido, en nuestro ejemplo, en cómo manipular el diagrama propiamente dicho, sin preocuparnos por los componentes, o sea sin preocuparnos en cómo desarrollar el Editor de Diagramas.

En las próximas secciones entonces definimos un modelo de arquitectura híbrida para el contenido de aplicaciones colaborativas sincrónicas (HA4CAC – Hybrid Architecture for Collaborative Application Content) garantizando consistencia de información además de ofrecer soporte a la propiedad de latecoming.

5.1 Modelo de Arquitectura HA4CAC

Como se explicó en el capítulo 3, existen varios modelos de arquitecturas para la construcción de aplicaciones colaborativas sincrónicas, sin embargo solo tres de ellos son los más utilizados, estos son el modelo centralizado, replicado e híbrido.

Muchas aplicaciones en la actualidad, utilizan el modelo replicado o centralizado, sin embargo hoy en día no hay ninguna aplicación que implemente una arquitectura híbrida.

Basándonos en el patrón *Model View Controller* se describe en la figura 5.2 cada modelo de arquitectura (se recomienda al lector que no esté familiarizado con este concepto remitirse al Apéndice A antes de continuar con la lectura del capítulo).

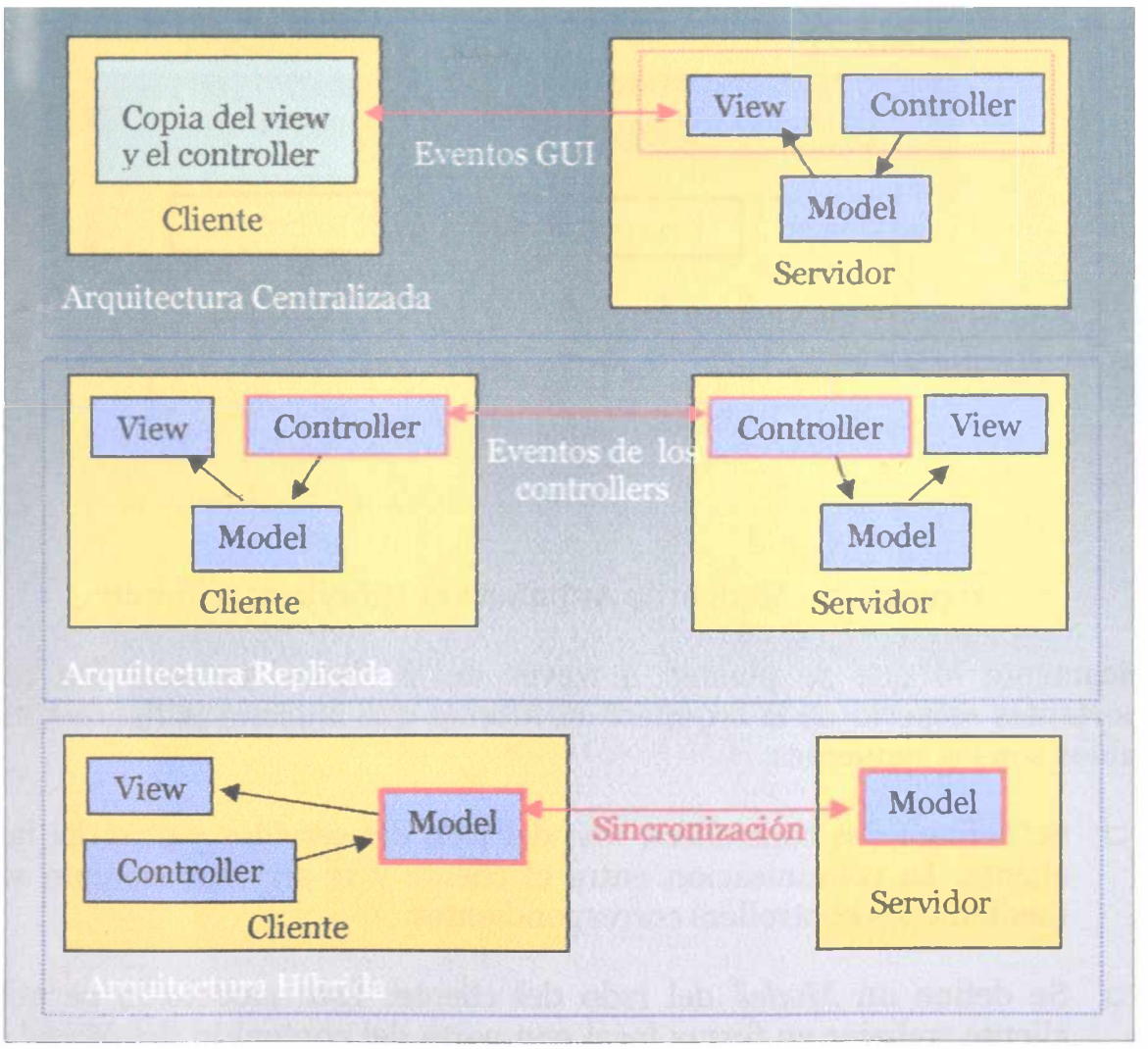


Figura 5.2 – Modelos de arquitecturas basados en el patrón MVC

Cabe destacar que hemos decidido basar la arquitectura propuesta HA4CAC en el patrón *Model View Controller*, dado que de este modo es mucho más sencillo el proceso de convertir cualquier aplicación single-user en una aplicación colaborativa, sólo modificando los controllers correspondientes.

Como se dijo en muchas oportunidades, tanto el modelo centralizado como el replicado tienen sus limitaciones (ver capítulo 3), tal vez una buena opción es usar el modelo híbrido tomando solo las ventajas de los otros dos modelos.

Como se ve en la figura 5.2, la arquitectura híbrida mantiene un modelo centralizado el cual es replicado en cada cliente; cada cambio que sufra el modelo replicado será automáticamente actualizado en el servidor central. Los modelos del cliente y del servidor se mantienen sincronizados, lo que garantiza integridad de los datos manteniendo consistente y actualizado el modelo central.
Sin embargo no estamos de acuerdo en que haya una sincronización directa entre los modelos del cliente y del servidor, ya que esto no respeta el concepto central del patrón *Model View Controller*, el cual plantea que cualquier cambio en el modelo debe realizarse mediante la vista y el controlador, nunca a través del modelo.
De esta manera decidimos hacer cambios en el esquema híbrido que se muestra en la figura 5.2, lo que llevó a definir un nuevo modelo de arquitectura híbrida como se puede ver en la figura 5.3

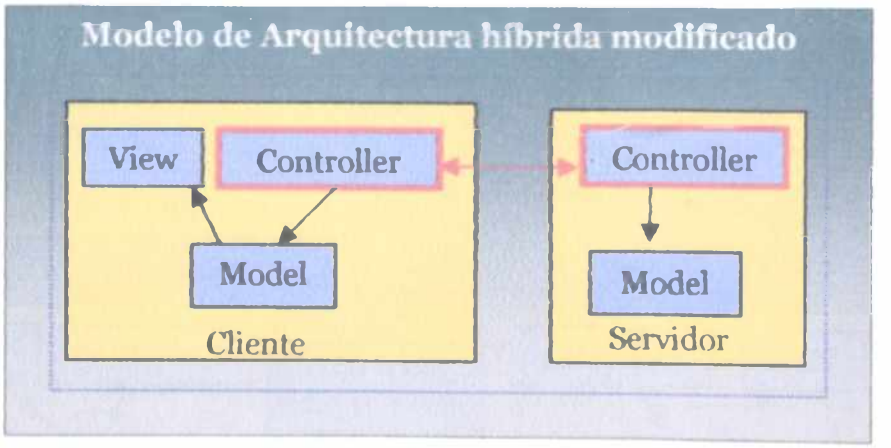


Figura 5.3 – Modelo de Arquitectura Híbrida modificado

Básicamente lo que se plantea a través de la figura 5.3 son dos cambios importantes respecto de la arquitectura híbrida que muestra la figura 5.2. Estos cambios son los siguientes:

- Se definen dos controllers, uno del lado del servidor y otro del lado del cliente. La comunicación entre el cliente y el servidor siempre se hará mediante los controllers correspondientes.

- Se define un *Model'* del lado del cliente. Este modelo le permitirá al cliente trabajar en forma local con parte del contenido del *Model* que se encuentra en el servidor. Este concepto será explicado con más nivel de detalle en las próximas secciones. Lo que debe quedar claro aquí es que el modelo que se replica en el cliente solo es una parte del modelo que se

encuentra en el servidor, la replicación se hará dinámicamente en tiempo de ejecución.

5.2 Diseño de la arquitectura HA4CAC

En la figura 5.4 se muestra el diseño general de la arquitectura híbrida para el contenido de aplicaciones colaborativas sincrónicas propuesta en este trabajo de grado (HA4CAC – Hybrid Architecture for Collaborative Application Content).

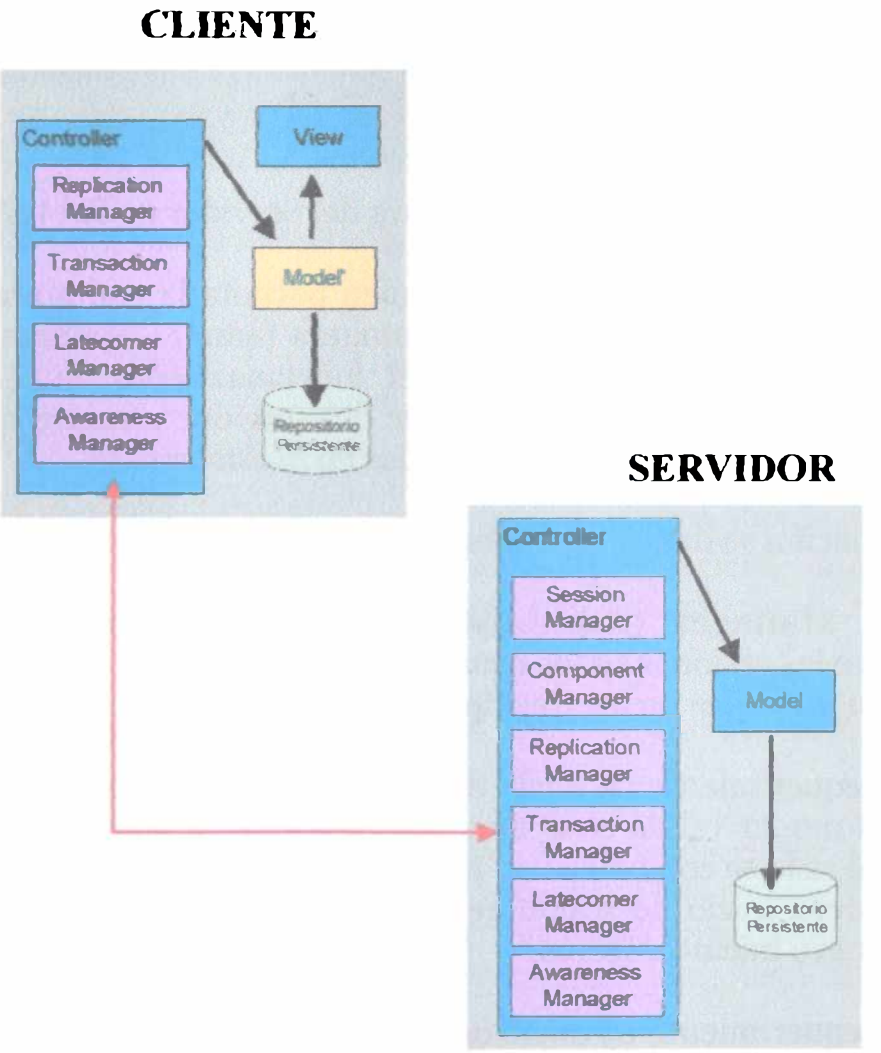


Figura 5.4 – Arquitectura híbrida para el contenido de Aplicaciones Colaborativas Sincrónicas (HA4CAC)

5.2.1 Estructura del Servidor

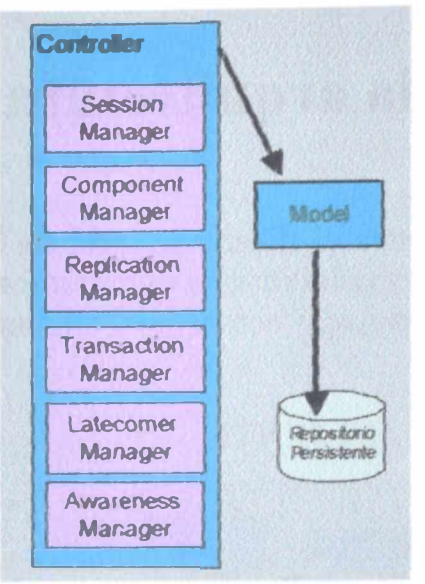


Figura 5.5 – Estructura del Servidor en HA4CAC

El *controller* del servidor va a ser quien permita la comunicación bidireccional con cada uno de los clientes. Los *managers* (administradores) que se presentan son los mínimos requeridos para el funcionamiento de HA4CAC, se podría modificar la estructura del *controller* del servidor siempre que se asegure que existan todos los manejadores indicados en la estructura.

A continuación se detallará el funcionamiento de cada *manager*:

Session Manager (Administrador de Sesión): mantiene información referente a las sesiones de los usuarios en la aplicación.
El *session manager* recibe tres tipos de requerimientos:

- Requerimiento de inicio de sesión de un usuario ya registrado: el *Session Manager* valida en el modelo que los datos ingresados sean correctos, en tal caso se comunica con el *Transaction Manager* para que éste registre el nuevo inicio de sesión generando una transacción con fecha y hora del nuevo inicio de sesión.

- Requerimiento de inicio de sesión de un nuevo usuario de la aplicación: el *Session Manager* valida que los datos del nuevo usuario no existan en el modelo, y luego se comunica con el *Transaction Manager*, el cual se va a encargar de generar las transacciones necesarias para poder reflejar en el modelo los datos del nuevo usuario, y los datos referentes a su inicio de sesión los cuales serán también almacenados en el repositorio persistente.

- Requerimiento de abandono de sesión.

En todos los casos el *Session Manager* posteriormente se comunica con el *Awareness Manager* para que el mismo notifique a los clientes del ingreso/egreso de un usuario en la sesión de trabajo.

Component Manager (Administrador de Componente): Administra la información relacionada con un componente, es decir, objetos del componente, usuarios y permisos de los usuarios sobre los objetos del componente.
Van a existir tantos *Component Manager* como componentes tenga la aplicación. Por ejemplo, si tenemos una aplicación colaborativa cuyos componentes son pizzarra, chat y editor gráfico, tendremos tres *Component Manager*.
El *Component Manager* recibe peticiones del *Session Manager* cuando un usuario ingresa al componente correspondiente como así también responde a eventos del cliente que indican que un usuario está por abandonar el componente.
El *Component Manager* se comunica con el *model* del servidor para solicitar información sobre los usuarios y objetos del componente, y con el *Latecomer Manager* cuando un usuario ha ingresado a un componente y se le deben notificar todos los cambios realizados en la misma durante su ausencia.
Encargado de mantener información acerca de cuales son los usuarios que iniciaron un proceso de latecoming y bloquear objetos compartidos simples o compuestos cuando son utilizados por un usuario en la sesión de trabajo.

Replication Manager (Administrador de Replicación): encargado de dos funciones

- Replica los objetos compartidos simples (contenido centralizado) bloqueados por el *Component Manager* para que un usuario pueda modificarlo de manera local.

- Recibe notificaciones de los *replications managers* de los clientes con sus respectivas transacciones y las deriva al *Transaction Manager* del servidor anexando el identificador del usuario que efectuó la notificación.

Las notificaciones que recibe de los distintos usuarios se encolan según el orden de llegada antes de ser enviadas al *Transaction Manager*.

Transaction Manager (Administrador de Transacciones): recepciona peticiones del *Session Manager*, *Component Manager* y *Replication Manager*.
No envía peticiones a ningún manager del servidor, solo se comunica enviando tramas al *model* quien hará efectivas las modificaciones llevadas a cabo por los usuarios.
Trabaja con diferentes tipos de tramas dependiendo la transacción a realizar, por ejemplo cuando un usuario modifica un objeto la trama contendrá un header formado por la información del usuario, la información del objeto que modifica y la transacción propiamente dicha.
Cuando un usuario modifica un objeto compartido en forma centralizada, el *Transaction Manager* es quien va almacenando los diferentes eventos que este realiza para así originar una transacción (este concepto será explicado con mayor nivel de detalle en la sección 5.2.4)

Latecomer Manager (Administrador de Latecomer): encargado de obtener, a partir del *model*, la secuencia de eventos realizados en un componente. De esta manera los usuarios (latecomers) podrán observar la sucesión cronológica de cambios efectuados sobre los objetos de un componente y qué

5.2.2 Estructura del Cliente

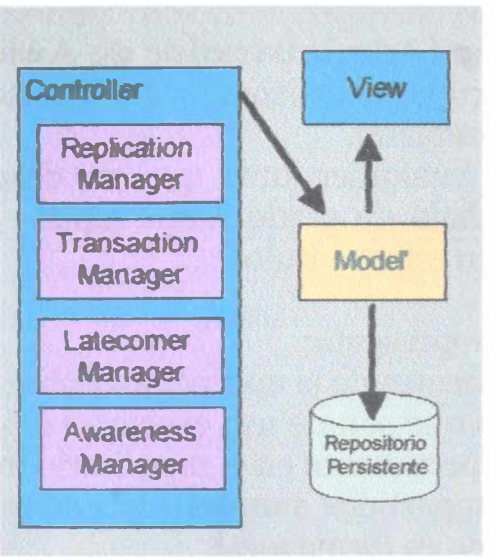


Figura 5.6 – Estructura del Cliente en HA4CAC

El *Controller* del cliente va a estar formado por los siguientes *managers* (administradores):

Replication Manager (Administrador de Replicación): Recibe las transacciones generadas por el *Transaction Manager* del cliente cada vez que el modelo local (*model*) cambia, y envía una trama al *Replication Manager* del servidor especificando la transacción realizada y el usuario que la efectuó.

Latecomer Manager (Administrador de Latecomer): recibe del *Latecomer Manager* del servidor un paquete de transacciones ordenadas en forma cronológica, tales transacciones reflejan los cambios efectuados en el contenido de un componente.
Dependiendo lo que decida ver el usuario, ya sea el estado final o la sucesión de eventos relacionados al contenido, el *Latecomer Manager* tomará las acciones específicas para cada caso.
Una vez terminado el proceso de latecoming del componente, el *Latecomer Manager* envía una notificación al *Latecomer Manager* el servidor informando que concluyó el proceso, esta notificación va a ser la que permita que los usuarios que se encuentran en el componente puedan seguir trabajando de manera centralizada.
La forma de trabajar de este manager será ampliada en el escenario 2 de la sección 5.3

Transaction Manager (Administrador de Transacciones): Cuando un usuario realiza modificaciones sobre un objeto de uso exclusivo o, sobre un objeto compartido simple el cual se encuentra replicado localmente, se genera una transacción, donde en el caso de objetos exclusivos va a ser reflejada en el *model'* del cliente siendo almacenada en el repositorio persistente local y replicada al servidor para ser almacenada en el *model*, y en el caso de objetos replicados va a ser reflejada en el *model'* e inmediatamente replicada al servidor.

usuario efectuó los mismos antes de su ingreso al componente. Como así también información sobre el ingreso y egreso de los usuarios en el componente.
Al momento de ingresar un nuevo usuario a un componente, el *Latecomer Manager* recibe una solicitud del *Component Manager* para iniciar el proceso de latecoming. En caso de que existan usuarios trabajando en forma centralizada en el componente, los mismos no se verán afectados en sus tareas, así como tampoco aquellos clientes que se encuentren trabajando de manera local.
El envió de las transacciones al *Latecomer Manager* del cliente es asincrónico, es decir, el *Latecomer Manager* del servidor envía todas las tramas al cliente y puede continuar trabajando, independientemente que haya o no finalizado el proceso de latecoming en el cliente. Cuando el Latecomer *Manager* del cliente finalice informara al *Latecomer Manager* del servidor el cual tendrá máxima prioridad para atender solicitudes de este tipo.
La forma de trabajar de este manager será ampliada en el escenario 2 de la sección 5.3

Awareness Manager (Administrador de Awareness): provee un contexto grupal actualizado y notificaciones de las acciones de cada usuario cuando sea apropiado.
Este *manager* como cualquier otro que se desee anexar a la arquitectura propuesta en este trabajo de grado, serán validos siempre que se mantenga la estructura planteada en este capítulo.

Model: representa la siguiente información:

- Componentes (pizarra, editor gráfico, etc.)
- Objetos manipulados a través de los componentes (contenido)
- Usuarios
- Permisos de acceso al contenido

Repositorio Persistente: almacena toda la información referente a la aplicación.
La implementación concreta de este repositorio es decisión exclusiva de quien utilice esta arquitectura para el diseño de una aplicación colaborativa sincrónica.
Tal repositorio puede ser implementado ya sea con xml, una base de datos, o cualquier tecnología que cumpla con dicho propósito.

La arquitectura HA4CAC permite la manipulación y control específico de los objetos, usuarios y permisos. Esto es posible dado que, los componentes y el contenido de los mismos se encuentran claramente separados, como se explicó en la introducción de este capítulo, basándonos en la arquitectura propuesta por DyCE.
Por otra parte, la manipulación de los componentes puede hacerse a través de cualquier arquitectura que pueda interactuar con la arquitectura propuesta en este trabajo de grado.

Todos los cambios realizados en el modelo de la aplicación se van a almacenar en el repositorio persistente, sin embargo el momento en el cual se almacenan efectivamente los cambios en el respositorio persistente podría ser en forma asincrónica de cuando son reflejados en el modelo. Es decir, la sincronización entre el modelo y el repositorio persistente no necesariamente debe efectuarse cada vez que se refleja una transacción en el modelo.

El *Transaction Manager* no se comunica directamente con los *managers* del servidor sino que se comunica únicamente con el *Replication Manager* del cliente el encargado de comunicarse directamente con el servidor.

Awareness Manager (Administrador de Awareness): como se dijo antes provee un contexto grupal actualizado y notificaciones de las acciones de cada usuario cuando sea apropiado.
Este manager como cualquier otro que se desea anexar a la arquitectura propuesta en este trabajo de grado serán validos siempre que se mantenga la estructura planteada en este capítulo.

Model': este modelo representa:

- Los componentes de la aplicación replicados en el cliente
- Los objetos que son de uso exclusivo del cliente (objetos almacenados de manera persistente en el repositorio local)
- Objetos compartidos simples replicados en el cliente para poder ser manipulados en forma local.

Este concepto será ampliado en la sección 5.2.3.

Repositorio persistente: almacenar los objetos de uso exclusivo del cliente, y todas los componentes de la aplicación permitiendo así al cliente trabajar de manera local. Este concepto será ampliado en la sección 5.2.3.

5.2.3 Conceptos de Model, Model' y Objetos en HA4CAC

Ya hemos visto anteriormente una breve explicación respecto de estos tres conceptos fundamentales en el diseño del modelo de arquitectura propuesto en este trabajo de grado.

Estos tres conceptos definen cómo será el manejo de los datos a través de la arquitectura HA4CAC.

A continuación se describen brevemente cada uno de ellos siendo debidamente ampliado cada concepto en las próximas secciones.

Model: representa la siguiente información:

- Componentes (pizarra, editor gráfico, etc.)
- Objetos manipulados a través de los componentes (contenido)
- Usuarios
- Permisos de acceso al contenido

Model': representa:

- Los componentes de la aplicación replicados en el cliente
- Los objetos que son de uso exclusivo del cliente (objetos almacenados de manera persistente en el repositorio local)
- Objetos compartidos simples replicados dinámicamente en el cliente para poder ser manipulados en forma local.

Objetos HA4CAC: representan la estructura del contenido que será manipulado por los componentes. Recordar que los componentes pueden ser diseñados y desarrollados independientemente al contenido que manipulen. Por tal motivo ponemos especial atención al contenido propiamente dicho.

5.2.3.1 Model

Representa los datos asociados con una aplicación colaborativa y su comportamiento. Estos datos como mencionamos antes son los componentes que forman parte de la aplicación (como ser pizarras, editores de texto, chat, etc.), los objetos HA4CAC que representan el contenido manipulado por cada componente, los usuarios quienes harán uso de la aplicación colaborativa, y los permisos que determinan si un usuario tiene acceso a determinados objetos.

Si bien el modelo representa toda esta información, este trabajo de grado pone especial atención y tiene como principal objetivo el manejo exclusivo de una parte del mismo: objetos HA4CAC, usuarios y permisos. Definimos cómo será su estructura y funcionamiento dentro de la arquitectura, ya que ellos constituyen el contenido manipulado por los componentes.
Este modelo se mantendrá siempre actualizado mediante los controladores adecuados, explicados con anterioridad.

5.2.3.2 Model'

Este concepto surge con el fin de poder diferenciar que los datos que son replicados en el cliente y almacenados de manera persistente en su repositorio local, solo representan una parte del modelo que se encuentra almacenado en el servidor de forma centralizado, de ahí surge el concepto de " ' ".
Básicamente este modelo representa los objetos HA4CAC que son replicados en el cliente, ya sea por ser objetos de uso exclusivo del usuario o por ser objetos compartidos simples que se replican dinámicamente al cliente para mayor disponibilidad de trabajo (conceptos que serán ampliados en la sección 5.2.3.3).

Lo que debe quedar claro aquí es que el *model'* representa una réplica parcial en el cliente del modelo centralizado, formado por todos los componentes de la aplicación colaborativa más los objetos de uso exclusivo del cliente. Por lo tanto el *model'* será diferente en cada cliente dependiendo de los objetos exclusivos de cada uno de ellos.
Para entender mejor este concepto veamos la figura 5.7 que se muestra a continuación:

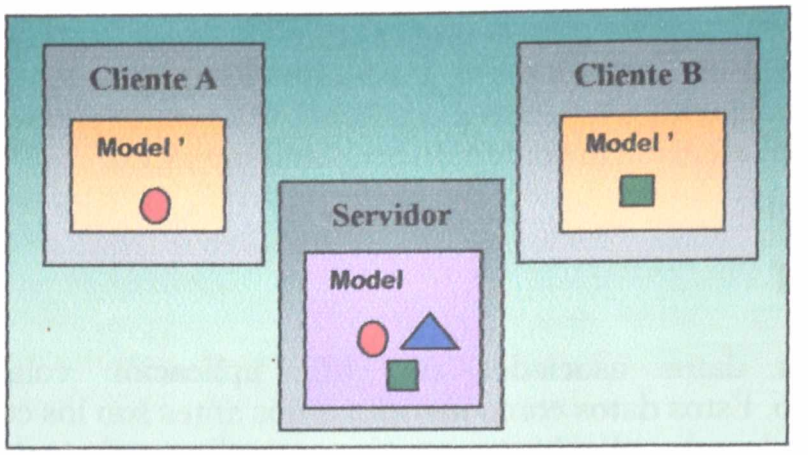


Figura 5.7 – Concepto de Model'

Como se ve en la figura 5.7 los *model'* del cliente A y del cliente B son distintos. Esto es porque el cliente A tiene acceso exclusivo sobre el objeto círculo, y el cliente B tiene acceso exclusivo sobre el objeto cuadrado, sin embargo tanto el círculo como el cuadrado se encuentran también en el *model* del servidor.
Es importante mencionar que tanto el *model* del Servidor como los *models'* de los clientes contienen los componentes, solo que en el ejemplo anterior éstos no aparecen a modo de no confundir al lector.
Los conceptos relacionados con objetos y permisos serán detallados en la próxima sección.

5.2.3.3 Objetos HA4CAC

Los objetos HA4CAC definen la estructura del contenido que será manipulado por los componentes.

Se definen dos clases de objetos HA4CAC:

- Objetos simples: representan la mínima unidad de información representable. Se puede ver a este concepto como si fuera una acción atómica indivisible.
- Objetos compuestos: están formados por una o varias composiciones de objetos HA4CAC simples.

Para entender mejor el concepto a continuación se muestra un ejemplo.

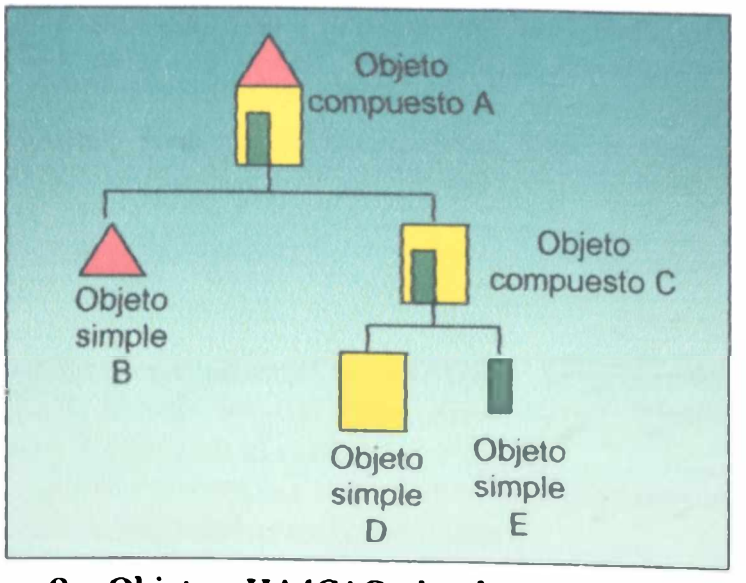


Figura 5.8 – Objetos HA4CAC simples y compuestos

Como muestra la figura 5.8, los objetos son considerados objetos simples como en el caso de los objetos B, D, E u, objetos compuestos como en el caso de los objetos A y C. Sin embargo, es importante aclarar de manera de no confundir al lector que el objeto A estará formado básicamente por tres objetos simples, estos son B, D y E organizados de manera tal que formen la figura correspondiente A como se ve en el ejemplo.
El nivel de detalle de los objetos se deja a libertad del diseñador de la aplicación. Esto quiere decir que el nivel de granularidad y desacoplamiento de los objetos compuestos en objetos simples es responsabilidad del desarrollador, aquí únicamente se define que se van a utilizar dos tipos de objetos, simples y compuestos, y solo se plantea como debe ser el manejo de los mismos.

Para entender mejor esto veamos la figura 5.9

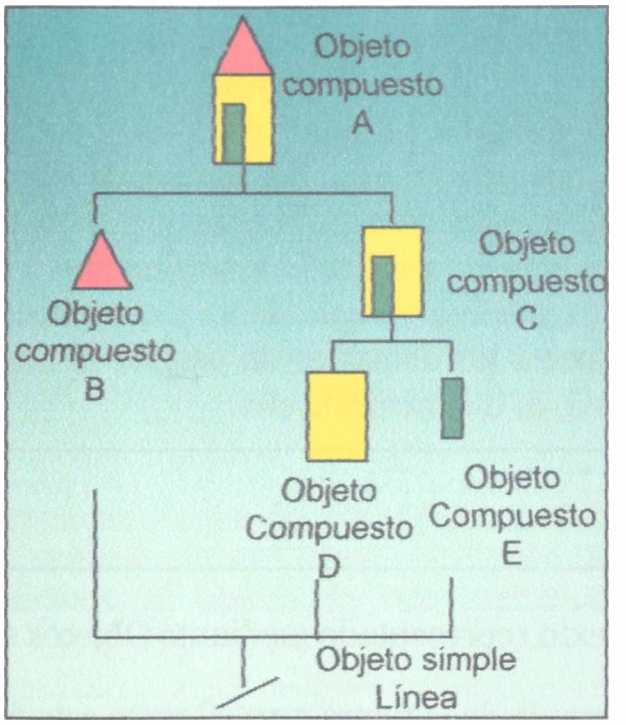


Figura 5.9 – Mayor nivel de descomposición de los objetos HA4CAC

La figura 5.9 muestra que podemos tener un mayor nivel de descomposición de los objetos compuestos que el planteado en la figura 5.8, en este ejemplo se adicionó un nivel más de descomposición, agregando como objeto simple una línea, de esta forma los objetos C, D y E de la figura 5.8 pasan a ser objetos compuestos.
Así quedo demostrado que decidir el nivel de descomposición de objetos para una aplicación será únicamente responsabilidad del desarrollador siempre que se respete la estructura antes mencionada.

Los ejemplos mostrados hasta aquí funcionarían bien si estamos trabajando con un editor gráfico o una pizarra compartida, sin embargo nos podemos preguntar como llevar los conceptos de objetos simples y compuestos a un editor de texto. Básicamente este componente manipula solo texto. La cuestión aquí será decidir cual es la representación de un objeto simple.

Veamos el siguiente ejemplo:

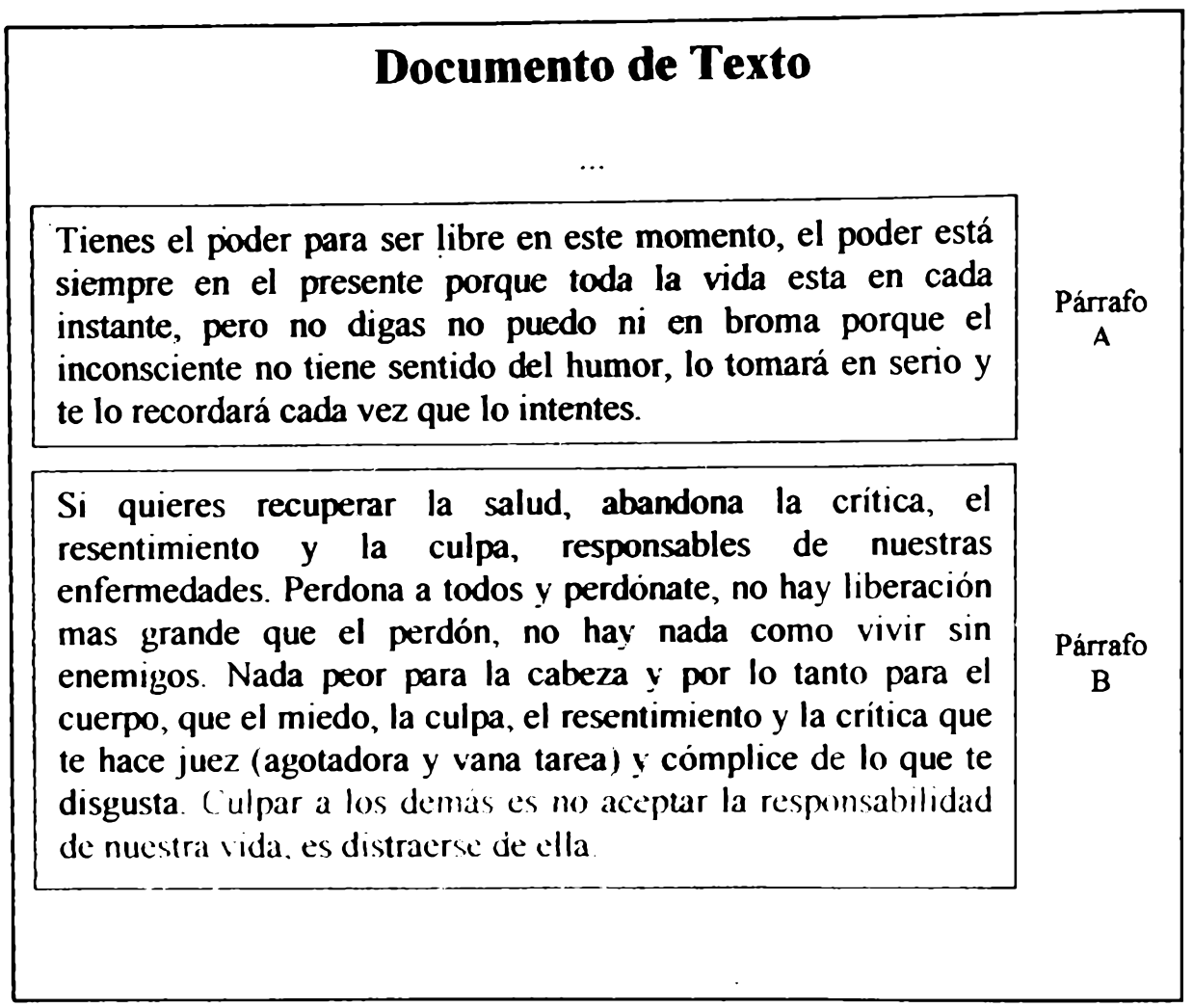


Figura 5.10 – Texto representado mediante Objetos simples y compuestos

Como se ve en el ejemplo de la figura 5.10, el texto esta formado por dos párrafos, A y B. El párrafo A tiene una sola oración y el párrafo B tiene 3 oraciones. Cada oración consta de un conjunto de palabras y cada palabra esta formada por letras. Este sería el mayor nivel de detalle para describir objetos simples y objetos compuestos. En este ejemplo, los objetos simples podrían ser las letras o las palabras y los objetos compuestos las oraciones y los párrafos, eso dependerá del grado de descomposición que queramos tener.

Simplemente con este ejemplo se quiso mostrar que, independientemente del componente que vayamos a utilizar, el modelo de objetos (contenido del componente colaborativo) siempre puede ser representado mediante objetos simples y compuestos.

Permisos de uso

Cuando un usuario realiza un requerimiento al servidor para trabajar con un objeto HA4CAC, el mismo puede ser para utilizar un objeto de uso exclusivo de otro usuario o un objeto compartido (cabe destacar que un usuario nunca hará una petición al servidor para utilizar un objeto de uso exclusivo de él mismo).
El *Component Manager* del servidor será quien recepcione este requerimiento, en el primer caso deberá indicar que no se puede utilizar el objeto por ser

exclusivo de otro usuario, y en el segundo caso deberá verificar si existen los permisos adecuados. En caso de que el usuario tenga permiso para poder trabajar con el objeto, dependiendo del tipo de requerimiento (objeto simple o compuesto) podrá trabajar en forma local (replicada) o en forma centralizada (en el servidor)

Básicamente se manejarán dos permisos generales para acceder a los objetos

Acceso total al objeto (objetos simples o compuestos)
Acceso parcial al objeto (para objetos compuestos)

Cuando un usuario crea un objeto inmediatamente se le otorga permiso de acceso total al mismo, decimos entonces que el objeto es de uso exclusivo para dicho usuario. Este objeto, ya sea simple o compuesto, se almacenará en el *model'* del cliente que lo creo y será replicado al modelo del servidor.
Que el objeto se encuentre almacenado en forma local (en el *model'*) permite que su dueño (owner) tenga mayor disponibilidad de trabajo sobre el objeto exclusivo, pudiendo trabajar de manera local cuando lo desee. Es importante destacar que el objeto también reside en el servidor y cualquier cambio de estado será notificado inmediatamente mediante el *controller* correspondiente, como se explicó en las secciones anteriores. De esta manera el *model* central siempre estará actualizado y cualquier usuario que entre a la sesión tendrá la información consistente sin sobrecargar al servidor consultando a cada cliente por sus objetos exclusivos, por el contrario la información será obtenida directamente consultando el modelo centralizado del servidor.

Si un usuario decide compartir alguno de sus objetos exclusivos para que pueda ser modificado por otro usuario, deberá otorgar los permisos correspondientes, convirtiendo de esta manera al objeto de uso exclusivo en objeto de uso compartido.

Los objetos compartidos van a residir siempre en el *model* del servidor, por lo cual el objeto que antes era de uso exclusivo ya no residirá en el *model'* del cliente como así tampoco en el repositorio persistente del usuario.

Si un usuario desea utilizar un objeto compartido debe efectuar la solicitud adecuada. Si el objeto está libre podrá utilizarlo y el *Component Manager* del servidor será el encargado de bloquear dicho objeto para que ningún otro usuario pueda modificarlo simultáneamente. Esto garantiza consistencia de información y evita problemas como los que se mencionaron durante el desarrollo del capítulo 4.
Si el objeto compartido que se desea utilizar es un objeto simple o un objeto simple que forma parte de un objeto compuesto, éste será replicado al cliente dinámicamente y bloqueado en el servidor, lo que permite al cliente tener mayor disponibilidad sobre el objeto compartido, una vez que finalice su edición será nuevamente actualizado en el modelo central y se eliminará su copia en el cliente. Por el contrario, si el objeto compartido con el que se quiere trabajar es un objeto compuesto deberá trabajarse de manera centralizada siendo bloqueado previamente por el *Component Manager* del servidor.

En la sección 5.3 se ejemplifican diferentes escenarios que permitirán comprender con un mayor nivel de detalle el funcionamiento de la arquitectura propuesta.

5.2.4 Modelo de la arquitectura HA4CAC (modelo conceptual – diagrama UML)

A continuación presentamos un diseño de alto nivel en UML que representa el modelo de arquitectura propuesto por este trabajo de grado (Figura 5.11).
Aquellos lectores que no estén familiarizados con el concepto de diseño UML remitirse al Apéndice B.

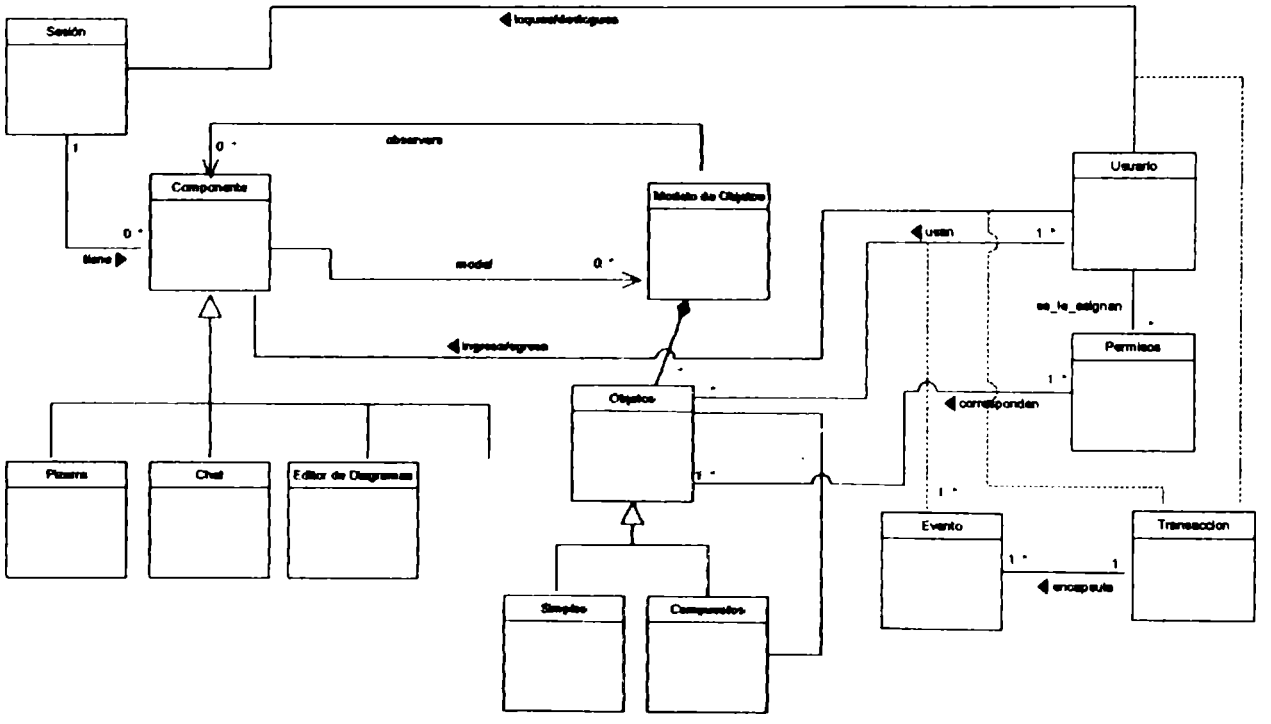


Figura 5.11 – Diseño alto nivel UML – Arquitectura HA4CAC

Elementos de la arquitectura

- Usuario: Cada uno de los usuarios se identifica con un nombre, además se le pedirán datos tales como un nick para trabajar en las diferentes sesiones, un mail, y cualquier otro dato que sea relevante para la aplicación a desarrollar.

- Sesión: Una sesión esta compuesta por un conjunto de componentes con los cuales los usuarios podrán trabajar. Cuando un usuario entra a una sesión de trabajo podrá ver un listado de todos los usuarios conectados en el mismo momento y en qué componente se encuentra trabajando cada uno de ellos.

- Componente: un componente es una herramienta que permite el trabajo colaborativo entre usuarios, como por ejemplo un editor gráfico. Un componente pertenece a una única sesión, especifica restricciones propias del componente tales como número máximo de usuarios que pueden interactuar en el componente.

- Modelo de objetos: formado por un conjunto de objetos los cuales son manipulados dentro de los componentes por los diferentes usuarios de la aplicación colaborativa. El objetivo de este modelo es permitirnos manipular en forma independiente los componentes de los objetos propiamente dichos, es decir lograr separar los componentes del contenido que ellos manipulan.

- Objetos: representan el contenido manipulado por los componentes colaborativos. Los usuarios son quienes crean estos objetos desde algún componente específico. Cada objeto tendrá un *owner* (dueño de objeto) que será el usuario que lo creó, sin embargo el owner podrá otorgar permisos al resto de los usuarios para que puedan modificarlo. El usuario dueño podrá establecer si quiere que el objeto sea accedido por todos o algunos usuarios, y en cada caso establecerá qué tipo de acceso o permiso quiere conceder a los otros usuarios. Como mencionamos en la sección 5.2.3.3 existen objetos simples y objetos compuestos. Los objetos simples representan la mínima unidad de información representable. Se puede ver a este concepto como si fuera una acción atómica indivisible.
  Los objetos compuestos están formados por una o varias composiciones de objetos simples.

- Permisos: nos permiten saber qué operaciones pueden realizar los distintos usuarios sobre los objetos que forman parte del contenido de los componentes. Si más de un usuario tiene permiso para modificar un objeto, el mismo se considerara un objeto compartido por lo cual el acceso al mismo será mediante bloqueos (*Component Manager*). Si solo un usuario puede modificar un objeto, es decir se trata de un objeto exclusivo, cuando el mismo realice una modificación se notificará al resto de los usuarios (*Replication Manager*). Recordemos que el concepto de permisos fue ampliado en la sección 5.2.3.3

- Eventos: operación realizada en un determinado momento (timestamp) por un usuario sobre algún objeto que se encuentra en un componente. Cada vez que un usuario realice alguna modificación sobre un objeto se generará un evento, el cual posteriormente formará parte de una transacción que será almacenada en el repositorio persistente del servidor, y permitirá en caso de un proceso de latecoming reconstruir cada uno de los cambios efectuados sobre los objetos de un componente. Los eventos siempre están asociados a un objeto y un determinado usuario.

- Transacción: similares a las utilizadas por las bases de datos para actualización, nos permiten mantener un orden entre los distintos eventos que realiza un usuario sobre un objeto. Cuando un usuario crea o realiza modificaciones sobre un objeto puede generar uno o varios eventos, los cuales van a formar parte de una transacción, la cual se

considerará como una única operación atómica y será almacenada en el repositorio persistente del servidor.
Además de las transacciones generadas por modificaciones en los objetos, también se manejan transacciones para registrar el ingreso/egreso de un usuario a un componente o a la sesión de la aplicación colaborativa.
Podemos entonces generalizar los tipos de transacciones en:

- ✓ Transacciones sobre objetos: formadas por un conjunto de eventos donde cada evento tendrá información sobre el objeto que se modifica, usuario que realiza la modificación, fecha/ hora y operación realizada

- ✓ Transacciones de ingreso/egreso de la sesión: contendrá información sobre el usuario que ingresa a la sesión y fecha/hora del ingreso/egreso

- ✓ Transacción de ingreso/egreso al componente: contendrá información sobre el usuario, el componente a la que se desea ingresar y fecha/hora del ingreso/egreso al componente.

A continuación mostraremos tres diagramas de interacción diseñados en UML a modo de esquematizar formalmente tres procesos básicos dentro de la arquitectura propuesta. Estos mismos procesos serán ejemplificados con un mayor nivel de detalle en la sección 5.3, el diagrama de la figura 5.12 se corresponde con el escenario 1, en tanto los diagramas de las figuras 5.13 y 5.14 corresponden con el escenario 2.

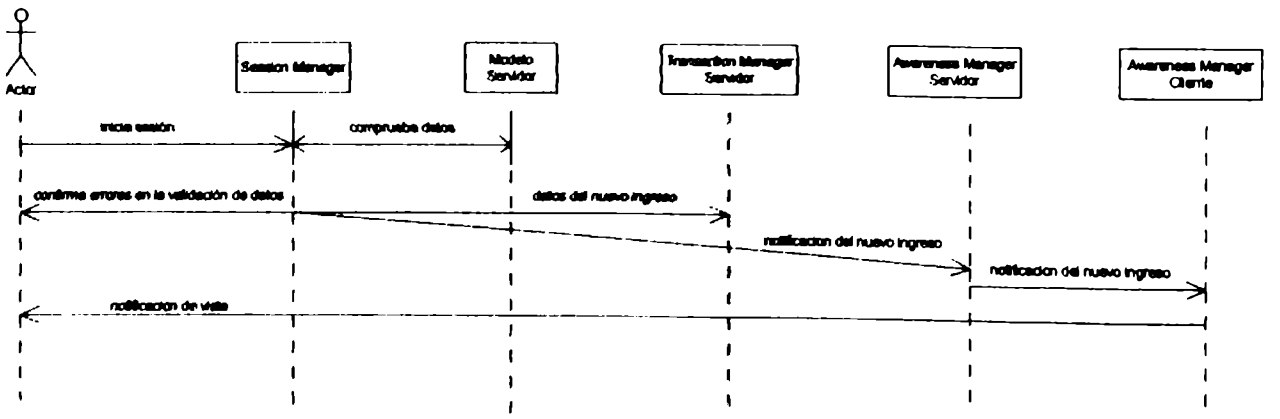


Figura 5.12 – Inicio de sesión – Diagrama de Secuencia UML

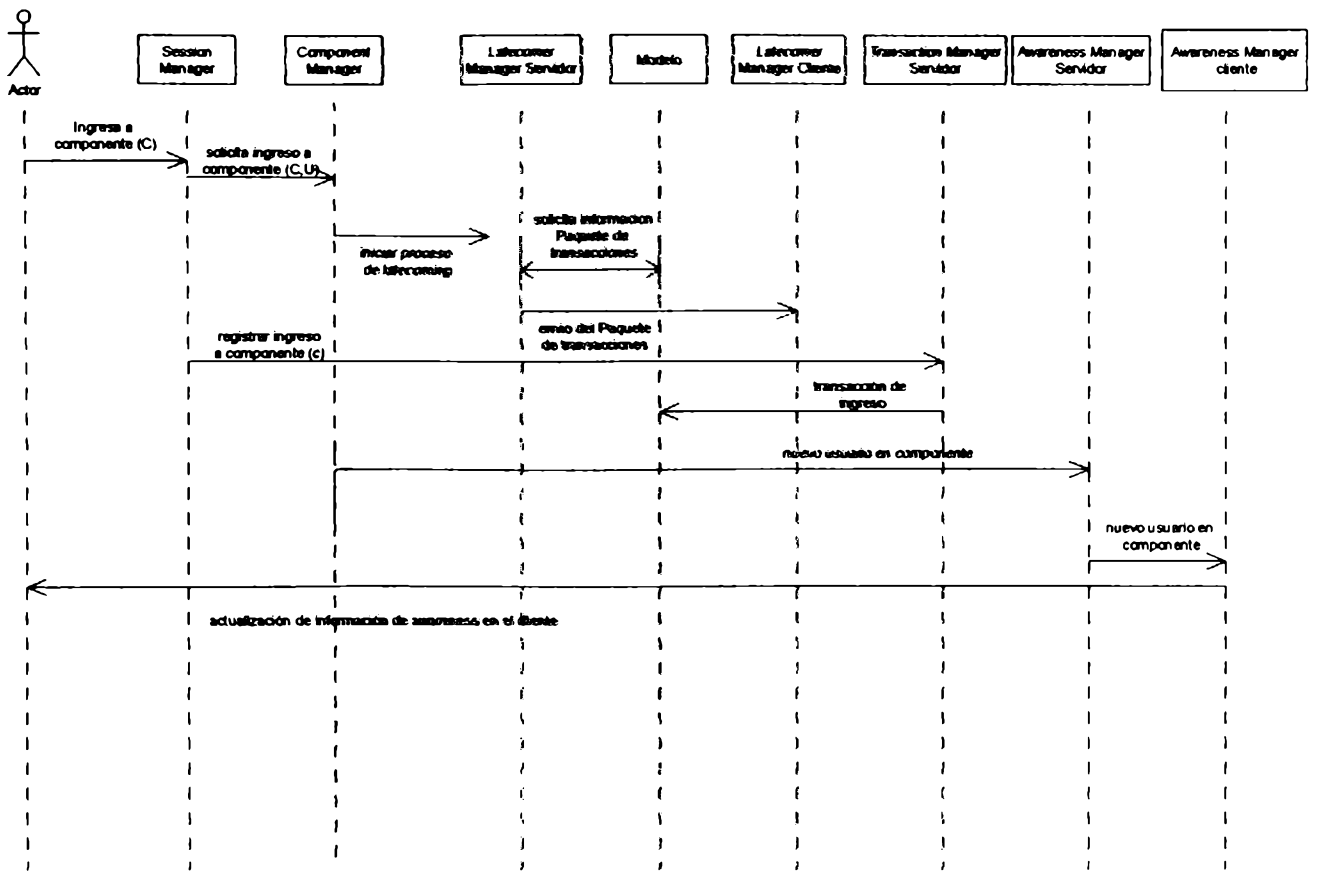


Figura 5.13 – Ingreso a un componente – Diagrama de Secuencia UML

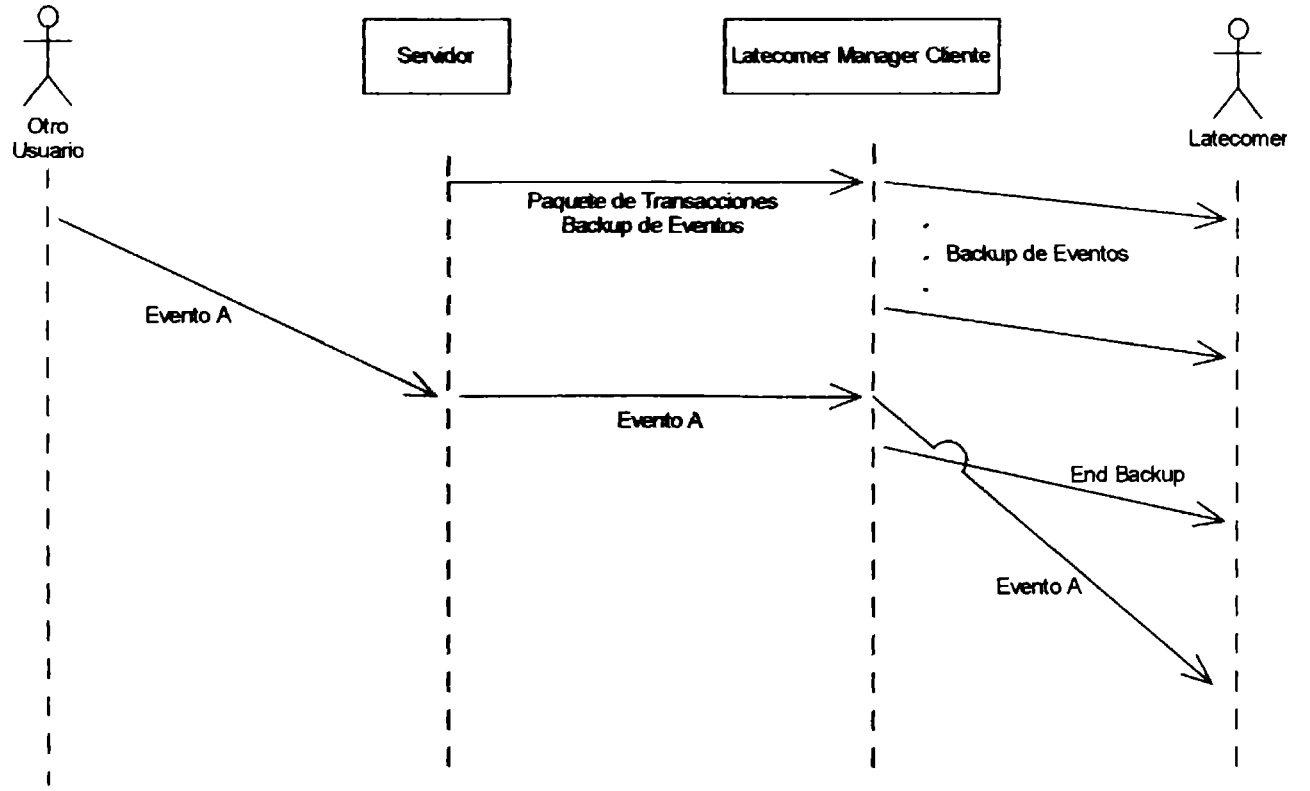


Figura 5.14 – Proceso de latecoming – Diagrama de Secuencia UML

Como muestra la figura 5.14 cualquier evento que ocurra durante el proceso de latecoming, será encolado en el cliente hasta que termine el mismo. Una vez finalizado se ejecutarán todos los eventos que ocurrieron durante el proceso. De

esta manera el cliente tendrá una visión actualizada del componente colaborativo al que ingresó.
Es importante destacar, que el ingreso de un nuevo usuario a un componente, no afecta el trabajo de ningún otro usuario que se encuentre trabajando en el componente, es decir, no se bloquea el componente, no hay bloqueos de datos como así tampoco se pierden eventos durante el proceso.

5.3 Escenarios

Como mencionamos anteriormente la comunicación entre los clientes y el servidor es a través de los *controllers*, enviando y recibiendo peticiones entre los distintos *managers* (administradores).
Para poder entender mejor cómo se realiza esta comunicación, a continuación vamos a plantear posibles escenarios y mostrar en cada uno de ellos qué *managers*, tanto del cliente como del servidor se comunican, y qué información es necesaria en tal comunicación.
Antes de comenzar con los diferentes casos, vamos a suponer que tenemos una aplicación colaborativa sincrónica la cual está formada por dos componentes:

Editor Gráfico

Editor de Texto

Los usuarios solo van a poder crear o manipular objetos dentro de estos componentes siempre que tengan los permisos correspondientes para hacerlo.
Vamos a suponer que ya existen objetos creados a la hora de comenzar a trabajar, los cuales fueron creados en sesiones anteriores, y que contamos con dos usuarios, *Aixa* y *Mariana*, que ya han iniciado sesión al menos una vez en la aplicación; y además contamos con otro usuario *Mauricio* el cual no inició nunca su sesión en la aplicación.
En las sesiones anteriores tanto *Aixa* como *Mariana* crearon objetos dentro del editor gráfico, pero no ingresaron nunca al editor de texto, por lo cual el ingreso al editor de texto no va a generar ninguna sucesión de operaciones para el proceso de latecoming pero si la va a generar el ingreso al editor gráfico.
Los objetos que se encuentran en el editor gráfico son:

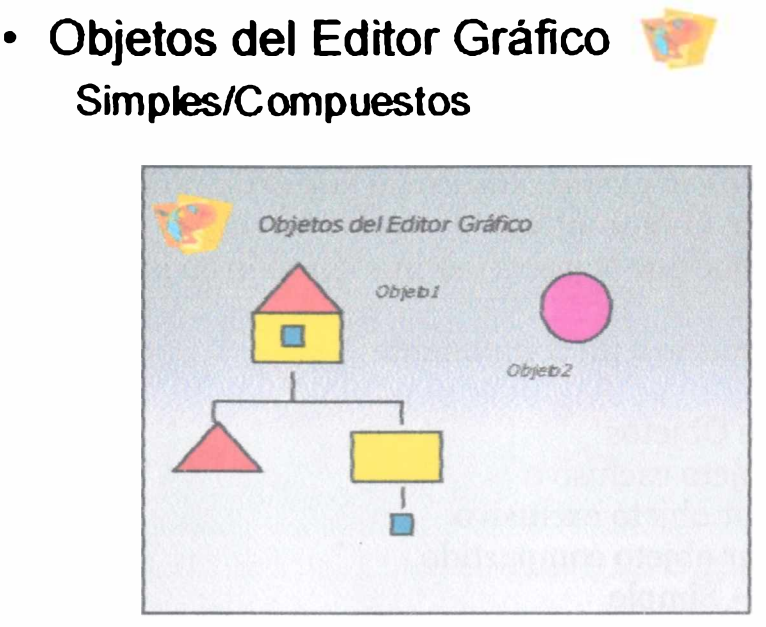


Figura 5.15 – Objetos simples y compuestos del Editor Gráfico

El Objeto2 que se muestra en la Figura 5.15 representa a un objeto simple, y el Objeto1 representa a un objeto compuesto formado por un objeto simple (triángulo y un objeto compuesto (rectángulo, cuadrado).

Para poder mostrar los distintos casos para la manipulación de objetos vamos a suponer que existen los siguientes permisos:

Objeto1
Aixa: Acceso Total
Mariana: Cuadrado acceso Total
Rectángulo acceso Total
Triángulo sólo visualiza
Objeto2
Aixa: solo visualiza
Mariana: acceso Total.

Recordemos que cuando un usuario ingresa a un componente tendrá que especificar el tipo de proceso de latecoming que quiere que se realice (estado final o playback de los eventos – ver sección 2.4.2), y que solo va a poder manipular aquellos objetos que sean exclusivos de él o para los cuales tenga permiso. Para nuestro ejemplo el Objeto2 es de uso exclusivo para el usuario *Mariana*, y el Objeto1 es un objeto compartido, por lo tanto el Objeto2 se encuentra tanto en el *model* del servidor como en el *model'* de *Mariana*, y el Objeto1 se encuentra almacenada en el servidor.
En caso de que *Mariana* quiera trabajar con el Objeto1 (objeto compuesto) solo podrá hacerlo sobre aquellos objetos para los cuales tiene permiso, en caso de que solicite un objeto para el cual tiene permiso y el mismo es un objeto simple, éste será replicado a su *model'* siempre que no esté en uso ni bloqueado, para poder trabajarlo en forma local, para nuestro ejemplo esto sería una petición para poder trabajar con el triángulo. En caso de solicitar un objeto compuesto, deberá trabajar en forma centralizada para evitar tener problemas de inconsistencias

como los mencionados en el capítulo 4, para nuestro ejemplo esto correspondería a la solicitud de trabajar con el rectángulo.

Escenarios Posibles

1) Inicio de sesión
   a- Usuario con al menos una sesión iniciada anteriormente (Usuario de la Aplicación Colaborativa)
   b- Usuario que por primera vez inicia sesión en la Aplicación

2) Ingreso de un usuario a un componente

3) Manipulación de Objetos
   a)- Crear objeto exclusivo
   b)- Modificar objeto exclusivo
   c)- Modificar objeto compartido
      c.1) – Simple
      c.2) – Compuesto

4) Convertir un objeto de uso exclusivo en uso compartido

5) Usuario que abandona un componente

6) Usuario que abandona la sesión

5.3.1 Escenario 1 - Inicio de sesión

Es este caso vamos a mostrar la sucesión de pasos cuando un usuario inicia una sesión en la aplicación colaborativa, vamos a contemplar dos casos diferentes ya que la secuencia de pasos es diferente si es la primera vez que un usuario inicia sesión ya que deberá registrarse y sus datos deben ser almacenados en el repositorio persistente del servidor.

a - Usuario con al menos una sesión iniciada anteriormente (Usuario de la Aplicación Colaborativa)

Supongamos que *Mariana* va a iniciar su sesión, recordemos que *Mariana* ya inicio sesiones anteriores por lo cual ya se encuentran almacenados en el repositorio persistente del servidor todos los datos necesarios para su registración tales como nick, contraseña, mail, etc.
También supongamos que *Aixa* ya inicio su sesión a la aplicación y se encuentra trabajando en el editor gráfico.
Tenemos entonces que registrar el inicio de sesión de *Mariana* y notificar a *Aixa* que *Mariana* ha ingresado a la aplicación. Cabe destacar que en este escenario no tenemos proceso de latecoming ya que el mismo se inicia cuando un usuario ingresa a una componente y no cuando ingresa a la aplicación.
Hasta antes de que *Mariana* inicie sesión la Figura 5.16 muestra que *Aixa* se encuentra trabajando en el editor grafico:

Figura 5.16 – View de *Aixa* dentro del Editor Gráfico

Cuando *Mariana* inicia su sesión su *controller* se va a comunicar con el *controller* del Servidor, para así establecer la comunicación Cliente/Servidor, recordemos que esta es la única forma posible de comunicación entre un cliente y el servidor propuesta por nuestra arquitectura. (Figura 5.17)

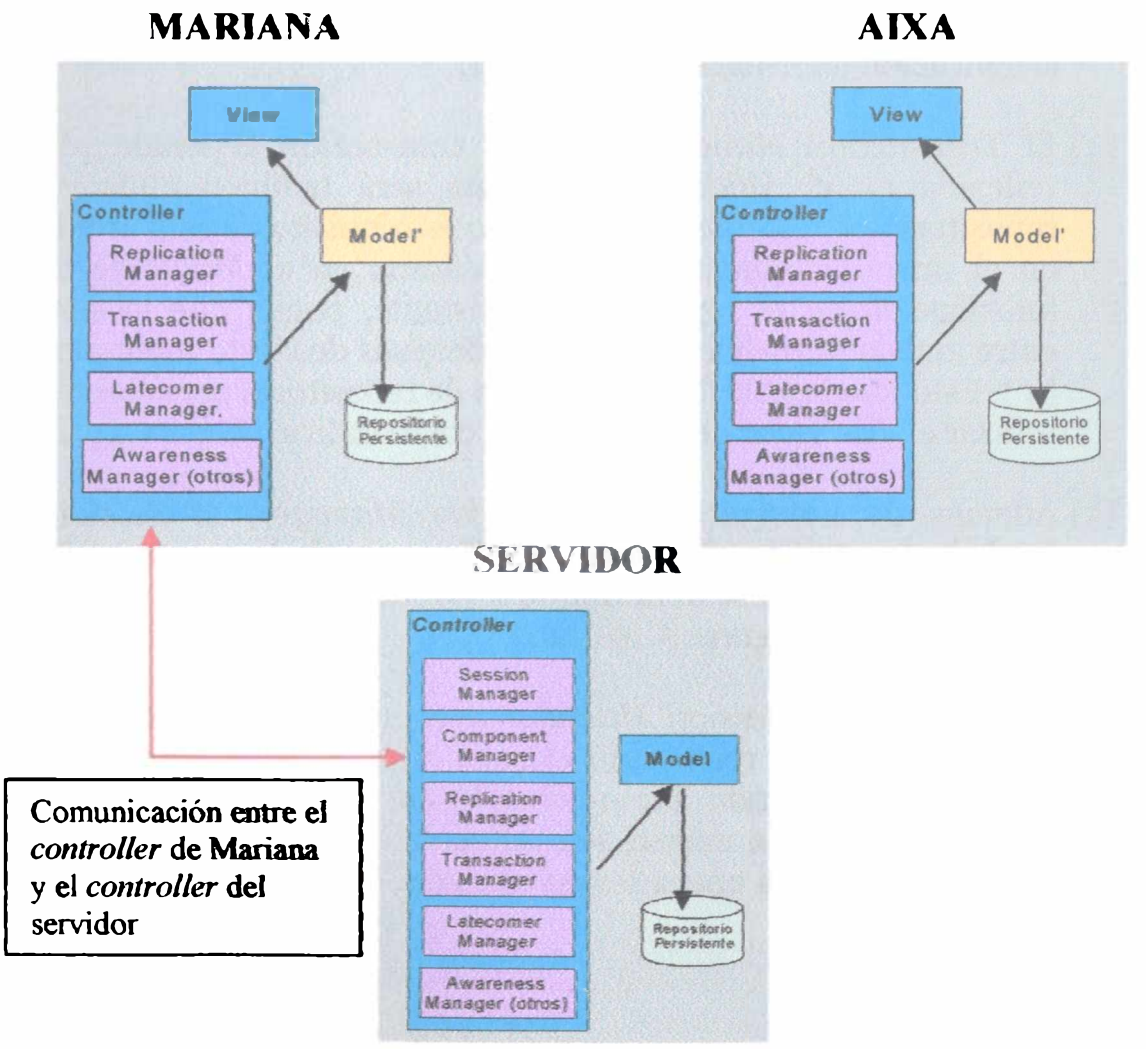


Figura 5.17 – Comunicación entre Cliente/Servidor

A continuación detallamos la secuencia de pasos que se suceden una vez establecida la comunicación entre el *controller* de *Mariana* y el *controller* del Servidor:

(1) Se va a enviar un requerimiento de inicio de sesión al *Session Manager*. En este requerimiento se pasa información tal como nombre de usuario y contraseña la cual luego va a ser verificada para comprobar que los datos ingresados son de un usuario valido de la aplicación colaborativa. (Figura 5.18)

(2) El *Session Manager* va a verificar la correctitud de la información que recibe comparando con la información que se encuentra en el Modelo del Servidor, en este paso se establece una comunicación bidireccional entre el *Session Manager* y el Modelo.
Si los datos enviados al *Session Manager* resultan incorrectos, el mismo va a notifica al cliente en este caso *Mariana* (2') tal situación. (Figuroa5.19)

(3) En caso de que los datos de inicio de sesión sea correctos, el *Session Manager* se comunica con el *Transaction Manager* el cual va a generar una nueva transacción la cual especificará fecha y hora (Timestamp) del nuevo inicio de sesión por parte del usuario.
Notar que en este caso la transacción generada no esta formada por un conjunto de eventos sino que como se explica en la sección 5.2.4 es un tipo de transacción especial que nos permite saber en que momento se logea a la aplicación un usuario. (Figura 5.19)

(4) El *Transaction Manager* envía la transacción generada para que sea reflejada en el *Model*, y la misma será también almacenada en el repositorio persistente. Notar que no es requisito de esta arquitectura que en el mismo momento que se almacena en el *Model* se almacene la información en el repositorio persistente, puede que la sincronización entre ambos se realice por ejemplo después de cierto intervalo de tiempo. (4'). Esto queda librado a la elección de repositorio persistente que decida utilizar el diseñador de la aplicación colaborativa. (Figura 5.19)

(5) Además de notificar al *Transaction Manager*, el *Session Manager* también le notifica al *Awareness Manager* de la nueva sesión iniciada, para que el mismo se lo notifique a los *Awareness Managers* del resto de los clientes (6) (Figuras 5.19- 520)

Notar que luego que el *Session Manager* le comunica al *Transacción Manager* (Paso 3), se puede estar realizando el paso 5, es decir *el Session Manager* se comunica con el *Transaction Manager* en forma asincrónica, no espera que este genere la transacción y la misma sea reflejada en el *Model*, para notificarle al *Awareness Manager* de la nueva sesión iniciada.

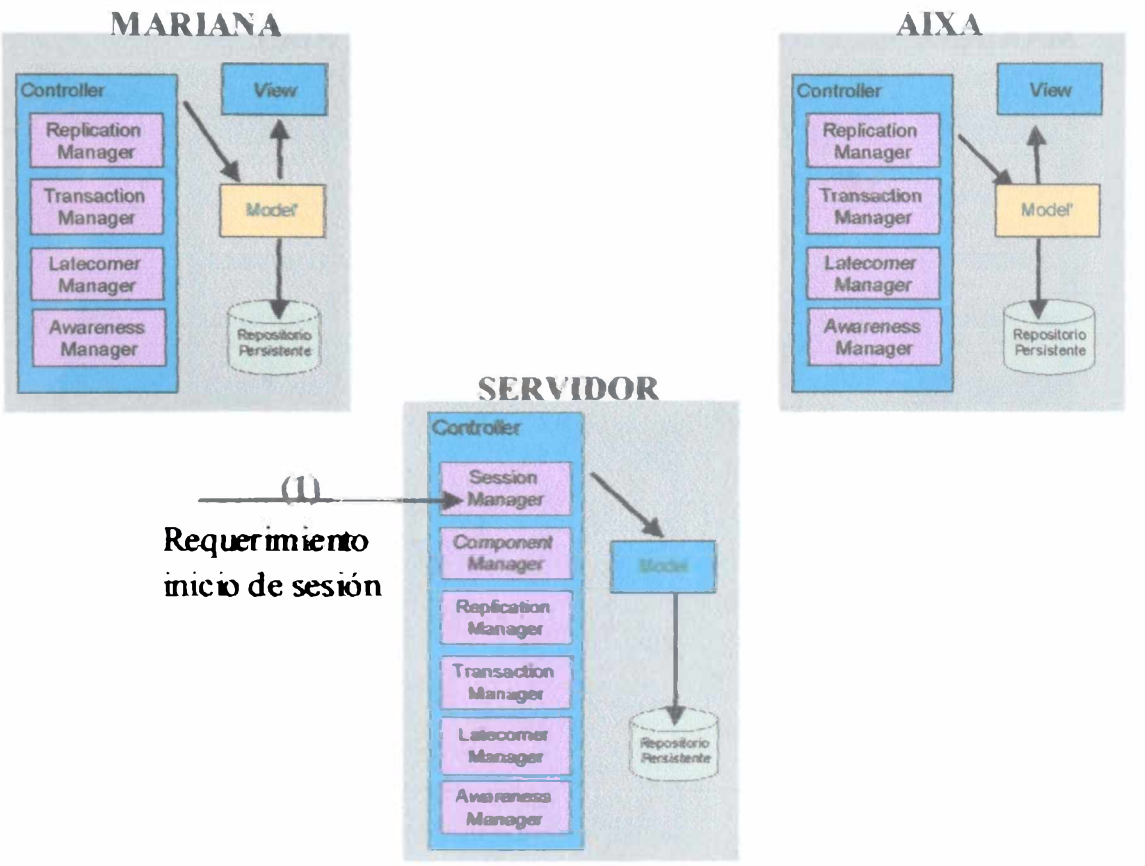


Figura 5.18 – Escenario 1 a: Paso (1)

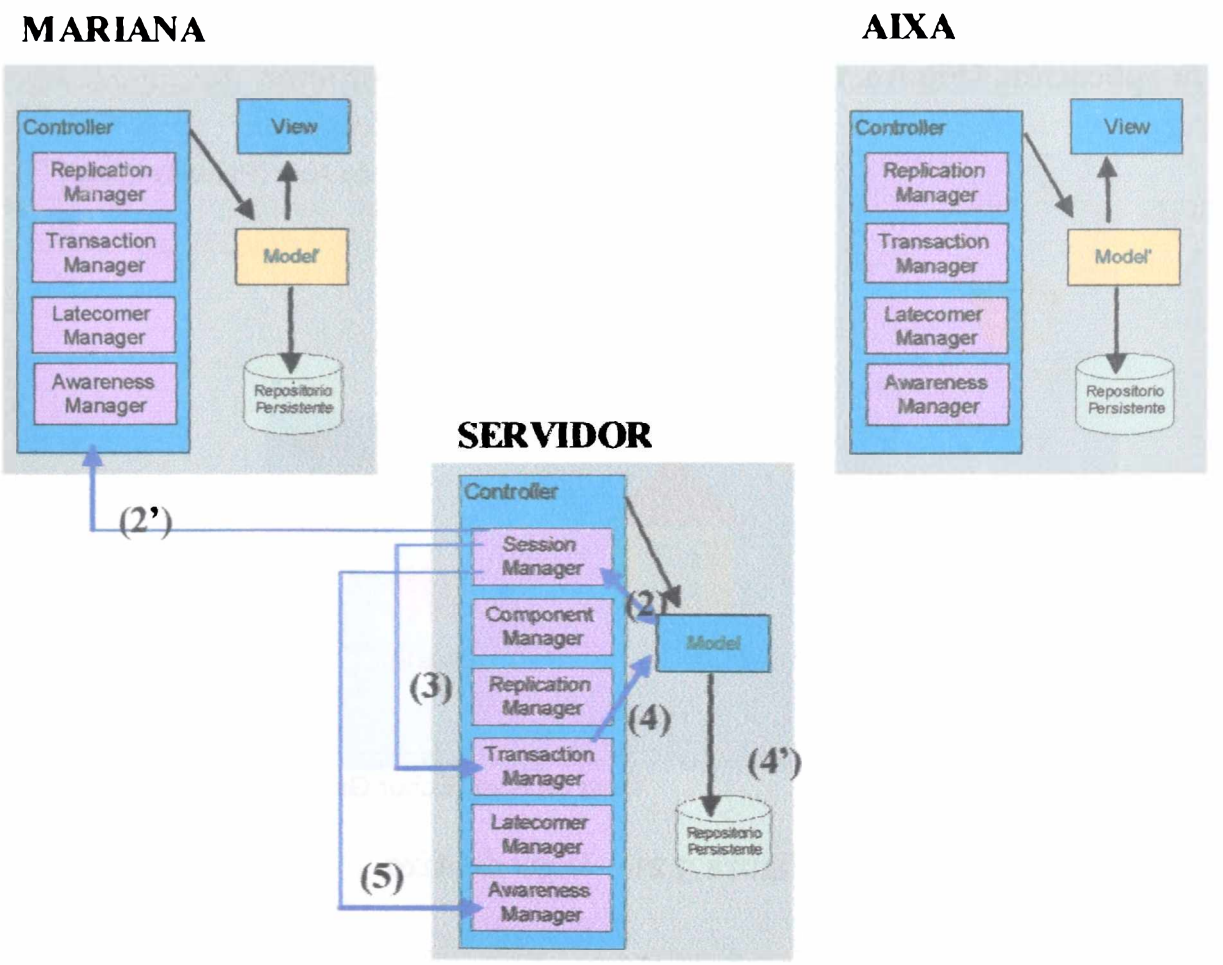


Figura 5.19 – Escenario 1 a: Pasos del 2 al 5

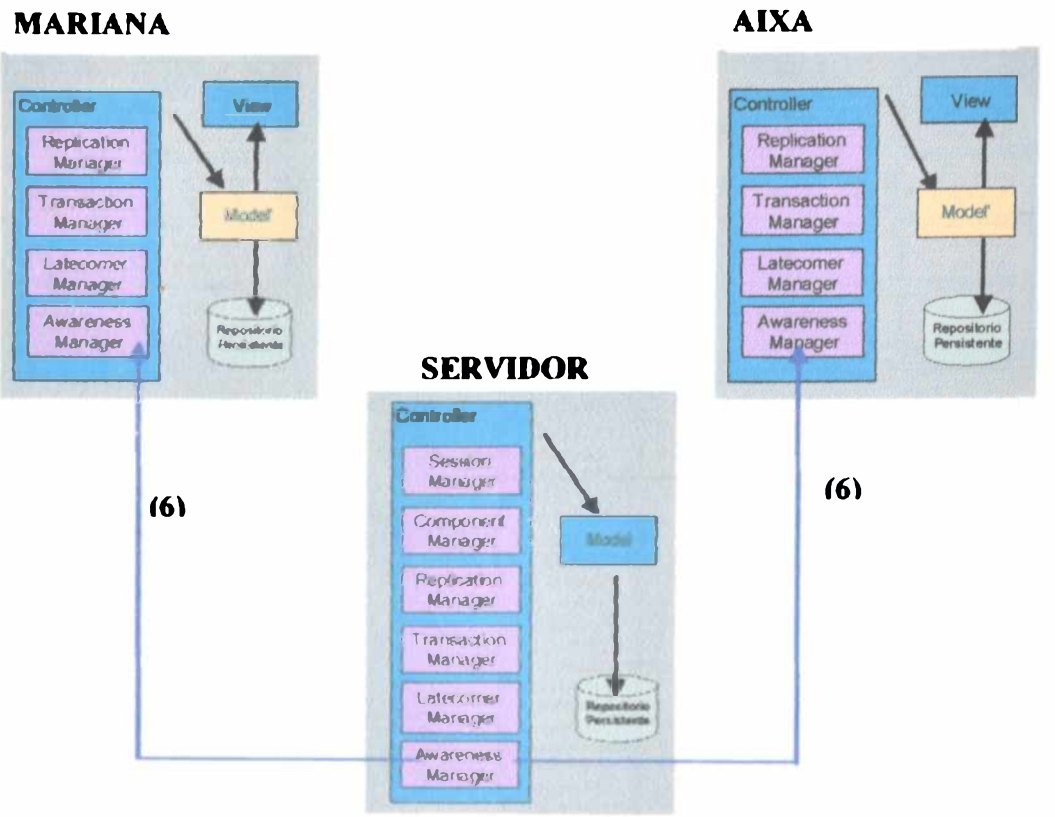


Figura 5.20 – Escenario 1 a: Paso 6

Una vez finalizado el inicio de sesión, tanto *Mariana* como *Aixa* deberían visualizar que ambas están utilizando la aplicación, para nuestro ejemplo se visualizara que *Aixa* se encuentra en el editor grafico y que *Mariana* se encuentra en la aplicación. (Figura 5.21 – 5.22)

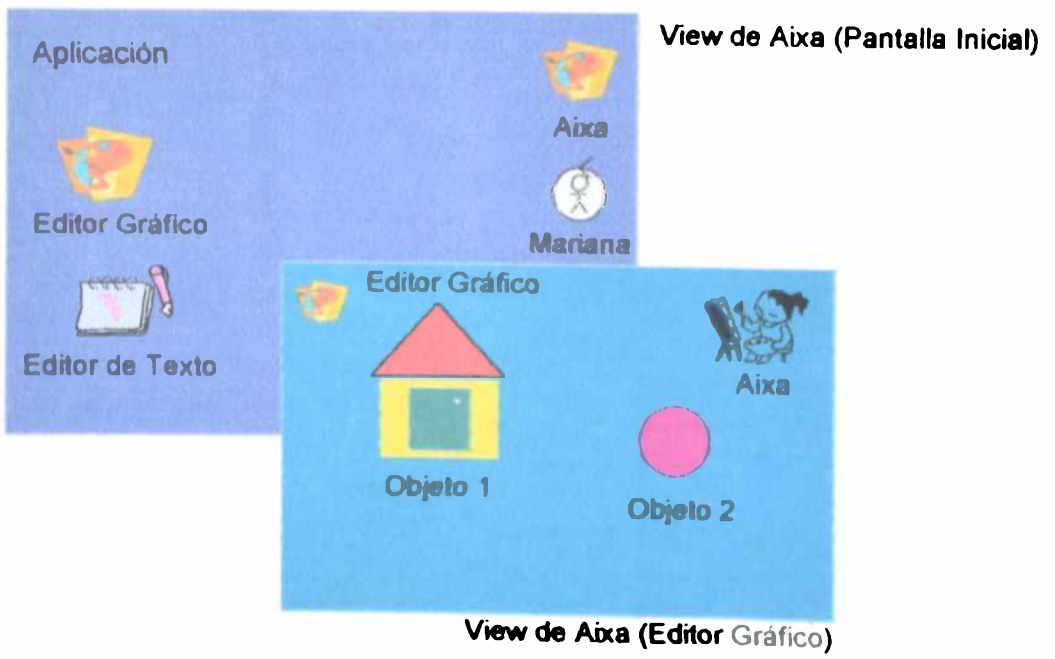


Figura 5.21 – View de *Aixa*

View de Mariana (Pantalla Inicial)
Aún no utiliza ningún componente

Figura 5.22 – View de *Mariana*

Escenario 1 b - Usuario que por primera vez inicia sesión en la Aplicación

Para poder ejemplificar este escenario vamos a suponer que sucede la siguiente situación: *Mauricio* ingresa por primera vez a la aplicación, a diferencia del escenario anterior en este caso se va a tener que almacenar la información relacionada al inicio de sesión tal como nick, contraseña, mail, etc. en el repositorio persistente del servidor.

Recordemos que *Mariana* y *Aixa* ya se encuentran utilizando la aplicación colaborativa, y en especial *Aixa* se encuentra dentro del editor grafico trabajando.
Además como en todos los casos la comunicación se realizara mediante entre el *controller* del cliente y el *controller* del servidor (Figura 5.23)

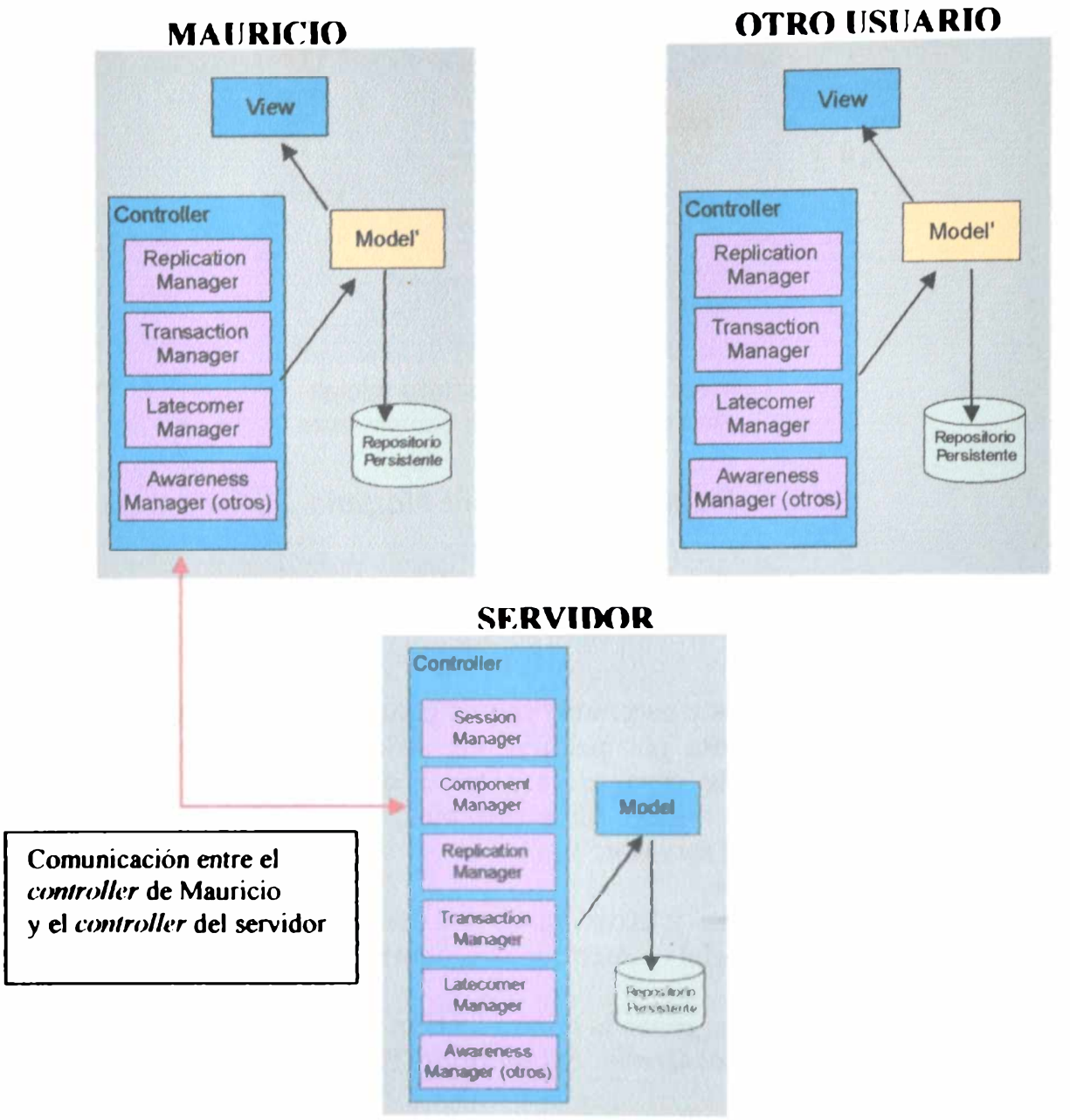


Figura 5.23 – Comunicación entre Cliente/Servidor

Secuencia de pasos una vez establecida la comunicación entre el nuevo cliente y el servidor

(1) Se va a enviar un requerimiento de inicio de sesión al *Session Manager*, con información tal como nombre de usuario, nick, contraseña, mail, etc. (Figura 5.24)

(2) Se establece una comunicación bidireccional entre el *Session Manager* y el Modelo para verificar que por ejemplo no existe otro usuario de la aplicación con el mismo nick. (Figura 5.24)

(3) Si la información ingresada es correcta, el *Session Manager* envía la información ingresada al *Transaction Manager*, para que este genere una nueva transacción, la cual va a contener además de los datos ingresados, fecha y hora del inicio de sesión. En caso de que ya exista por ejemplo un

usuario con el nick ingresado el *Session Manager* va a comunicar tal situación al cliente (3') en este caso *Mauricio*. (Figura 5.24)

(4) El *Transaction Manager* envía la transacción generada para que sea reflejada en el *Model*, y la misma será también almacenada en el repositorio persistente. Como mencionamos en este escenario inciso a) no necesariamente en el mismo momento que se refleja en el modelo se debe almacenar en el repositorio persistente, puede que la sincronización entre ambos se realice por ejemplo después de cierto intervalo de tiempo. (4') (Figura 5.24)

(5) Al igual que en el escenario anterior el *Session Manager* le notifica al *Awareness Manager* de la nueva sesión iniciada, para que el mismo se encargue de que todos los usuarios sean notificados de tal situación(6) (Figura2 5.25 – 5.26)

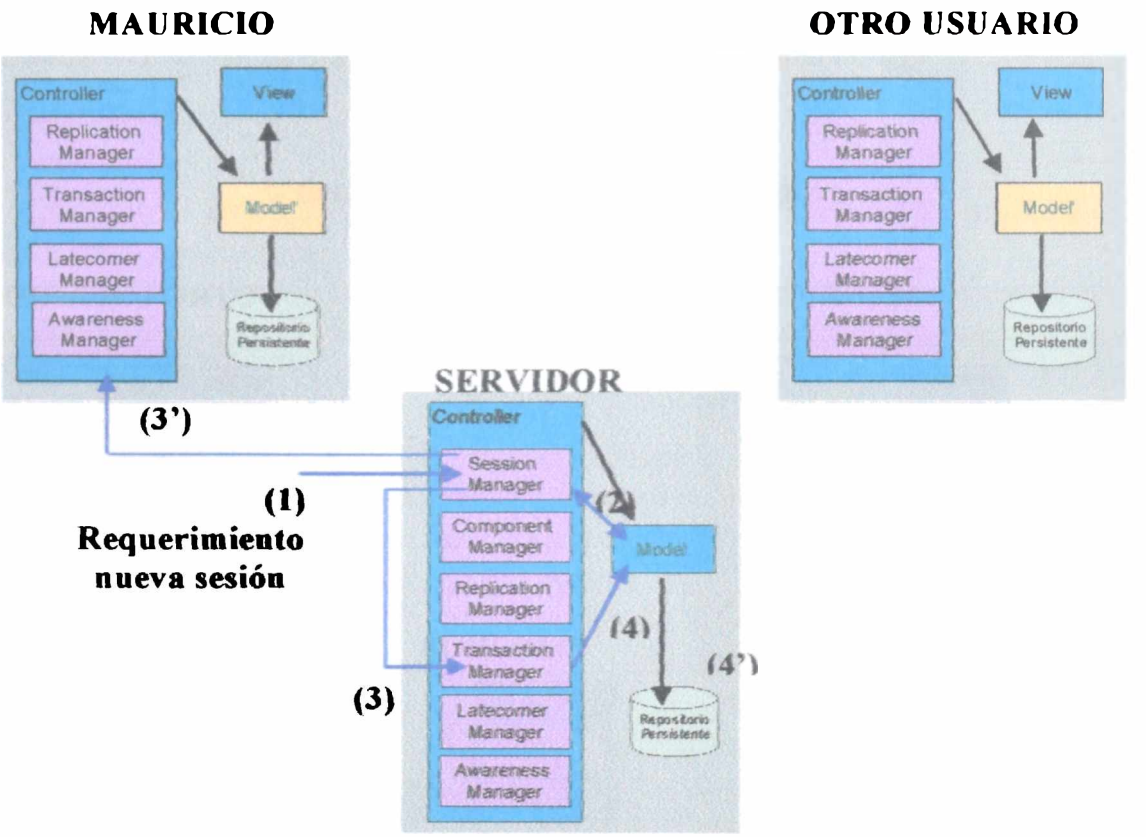


Figura 5.24 – Escenario 1 b: Pasos del 1 al 4'

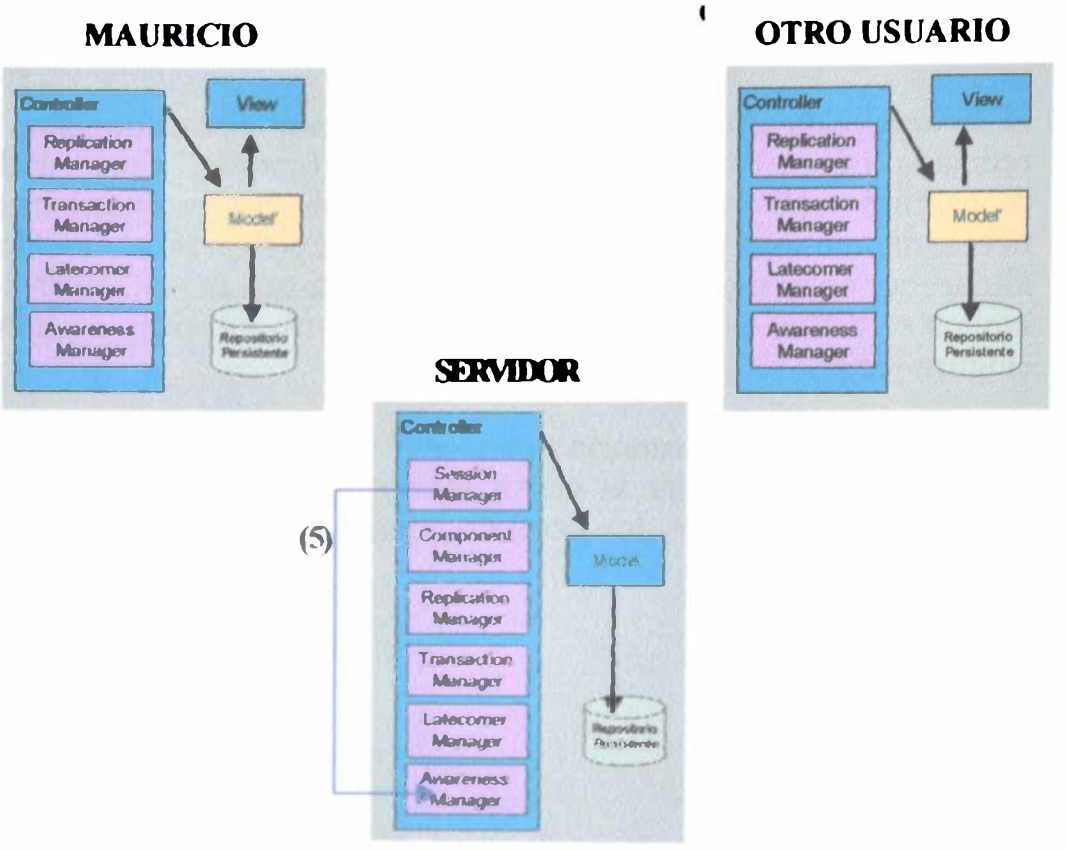


Figura 5.25 – Escenario 1 b: Paso 5

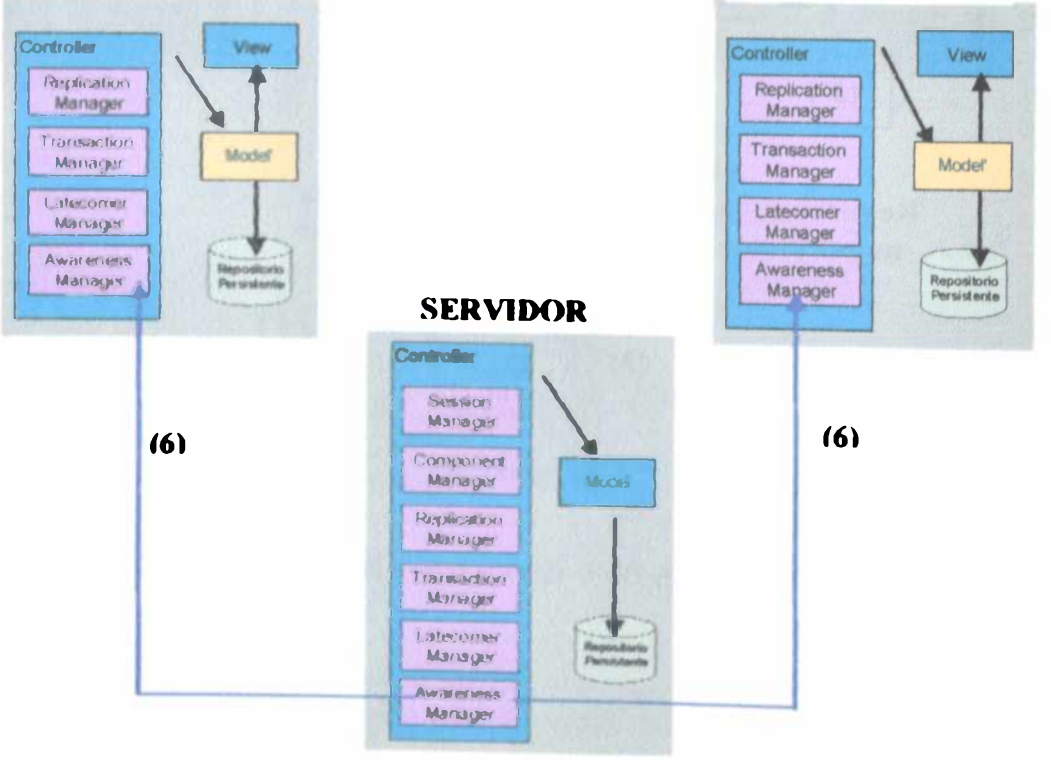


Figura 5.26 – Escenario 1 b: Paso 6

Las vistas de los tres usuarios se encontraran actualizadas permitiendo a cada uno de ellos saber en que lugar de la aplicación se encuentra el resto de los usuarios (Figura 5.27)

Figura 5.27 – View actualizada de todos los usuarios

5.3.2 Escenario 2 - Ingreso de un usuario a un componente

En este escenario se efectúa el proceso de latecoming debido al ingreso de un usuario a un componente.
El usuario al momento en que ingresa al componente debe especificar si requiere una notificación de estado final o sucesión de eventos, en ambos casos el *Latecomer Manager* del cliente va a ser quien se encargue de completar el proceso.
El proceso de latecoming mostrara como fue variando el estado de los objetos que se encuentran dentro del componente así como también la secuencia de ingresos y egresos de los diferentes usuarios al componente.
Tenemos que tener en cuenta que cuando un usuario inicia un proceso de latecoming, el tendrá para efectuar sucesión de eventos todos los eventos que se generaron antes de la fecha/hora que se inicia el latecoming, por ejemplo si el usuario inicia el latecoming el 3 de diciembre del 2004 a las 15:30 horas, la sucesión de eventos contendrá todas las modificaciones realizadas antes de esa fecha y hora. Pero mientras el usuario se encuentra en proceso de latecoming el resto de los usuarios siguen efectuando operaciones sobre objetos o ingresando/egresando del componente involucrada en este proceso, todos estos eventos serán notificados al latecomer y deberán ser encolados por el *Latecomer Manager* del cliente en un buffer para ser ejecutados tras la ejecución del ultimo evento enviado por el *Latecomer Manager* del servidor tras la solicitud de inicio de proceso de latecoming. Estos eventos se ejecutaran en el orden en que fueron encolados y una vez que el buffer se encuentre vacío recién en ese momento se considerara que ha finalizado el proceso de latecoming.

Recordemos que si algún usuario se encuentra trabajando de manera local dentro del componente su trabajo no se verá afectado por el ingreso de un nuevo usuario, así como tampoco se afecta a los usuarios que se encuentran trabajando en forma centralizada.

Para nuestro ejemplo vamos a suponer que *Mauricio* ingresa al editor grafico donde se encuentra *Aixa* trabajando.

Pasos en el proceso de latecoming:

(1) Cuando un usuario ingresa a un componente se inicia la comunicación entre su *controller* y el *controller* del servidor mediante un requerimiento al *Session Manager* de ingreso a un componente. (Figura 5.28)

(2) El *Session Manager* se comunica con el *Component Manager* del componente (Editor Gráfico) y le notifica que un usuario intenta ingresar. El *Component Manager* agrega al usuario al componente, y lo deberá marcar de alguna forma en especial para saber que dicho usuario se encuentra en un proceso de latecoming. **Es importante tener en cuenta que no se considera que el usuario ingreso al componente hasta finalizado el proceso de latecoming,** es decir el resto de los usuarios que están trabajando en el componente (Editor Gráfico) no serán notificados de que ingreso un nuevo usuario hasta que finalice el proceso de latecoming. (Figura 5.28)

(3) El *Component Manager* le envía una solicitud al *Latecomer Manager* para que este pueda realizar el latecoming. En esta solicitud se indica la fecha y hora del requerimiento de latecoming (1), ademas de indicar que usuario esta realizando tal requerimiento. (Figura 5.28)

(4) El *Latecomer Manager* solicita al Modelo todas las transacciones realizadas sobre el componente, anteriores a la fecha y hora que notifico el *Component Manager*. Este listado de transacciones detallara todos los cambios realizados sobre los objetos del componente así como también los ingresos y egresos de los usuarios, y esta ordenado en forma cronológica. La comunicación que se establece en este caso es bidireccional. (Figura 5.28)

(5) El *Latecomer Manager* del servidor envía el listado de transacciones al *Latecomer Manager* del cliente. (Figura 5.29)

(6) El *Latecomer Manager* del cliente recibirá el listado de transacciones y las ira aplicando hasta finalizar, notar que si no hubo ninguna transacciones el *Latecomer Manager* del cliente igual recibirá la notificación de inicio del proceso pero el listado de transacciones estará vacío.
Todas las transacciones que se generen mientras el cliente esta realizando un latecoming deberán ser también notificadas para ser encoladas y aplicadas antes de terminar el proceso de latecoming. Notar que se considera que el proceso de latecoming finaliza cuando se aplicaron todas las transacciones que envió el *Latecoming Manager* del servidor ante el pedido de latecoming así como también todas las que se fueron realizando por el resto de los usuarios. (Figura 5.29)

(7) Una vez finalizado en el cliente el latecoming, el *Latecomer Manager* del cliente le envía al *Latecomer Manager* del servidor una notificación de finalización del proceso del latecoming. (Figura 5.29)

(8) El *Latecomer Manager* del servidor notifica al *Component Manager* que el latecoming finalizo, por lo cual el *Component Manager* "desmarca" al usuario que se encuentraba realizando el latecoming, y recién a partir de este momento se considera que el cliente ingreso al componente. (Figura 5.29)

(9) El *Component Manager* notifica al *Session Manager* que un nuevo usuario ha ingresado. (Figura 5.29)

(10) El *Component Manager* informa al *Awareness Manager* del servidor que un nuevo usuario ingreso al editor gráfico y el *Awareness Manager* del servidor notificara a todos los clientes de este cambio a través de sus *Awareness Managers* (11). (Figura 5.29)

(12) El *Component Manager* notifica al *Transaction Manager* para que este genere una nueva transacción con los datos del ingreso del nuevo usuario al componente, la transacción se generara además de con el identificador del usuario, con la fecha y hora que indico el *Component Manager* al *Session Manager*. (Figura 5.29)

(13) El *Transaction Manager* genera la transacción y la envía al *Model* y posteriormente será almacenada en el repositorio persistente (13') (Figura 5.29)

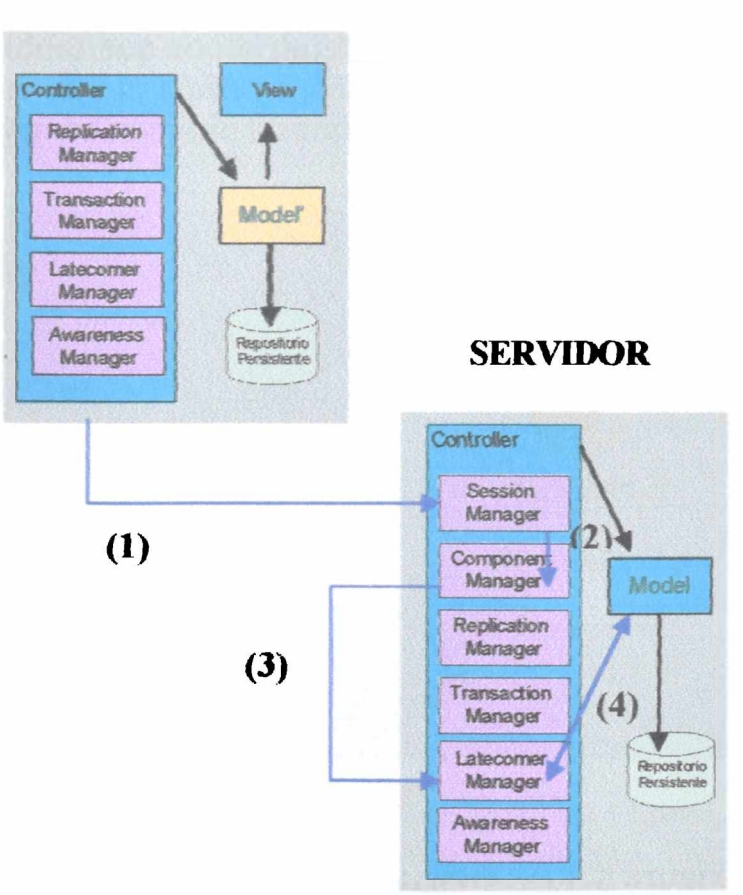


Figura 5.28 – Escenario 2: Pasos del 1 al 4

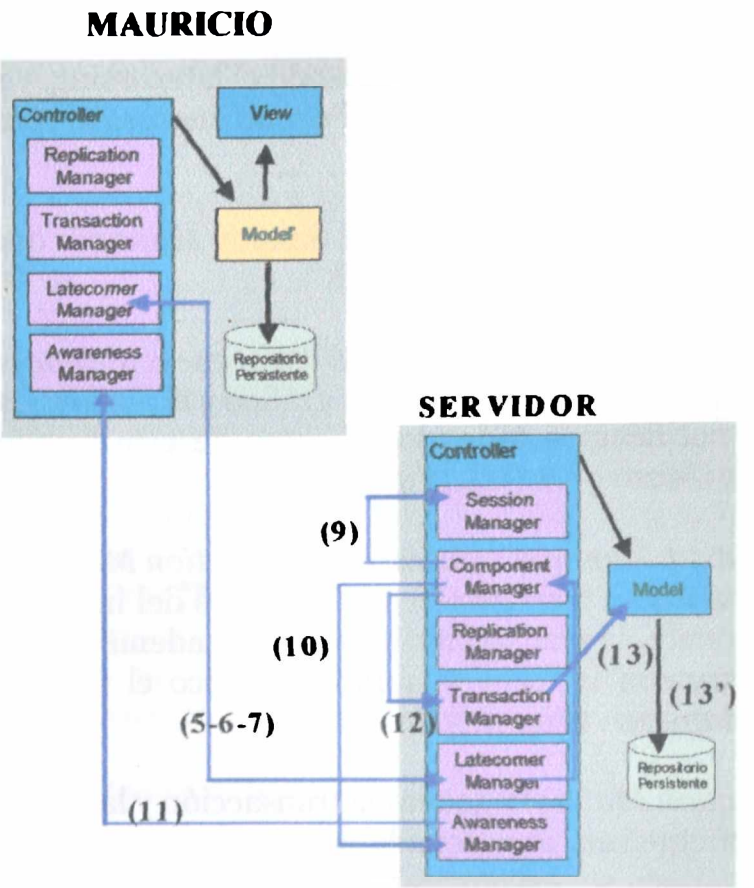


Figura 5.29 – Escenario 2: Pasos del 5 al 13'

Finalizado el proceso del latecoming tendríamos que tanto *Aixa* como *Mauricio* se encuentran dentro del editor grafico, y *Mariana* se encuentra utilizando la aplicación pero actualmente no se utilizando ninguna componente. (Figura 5.30)

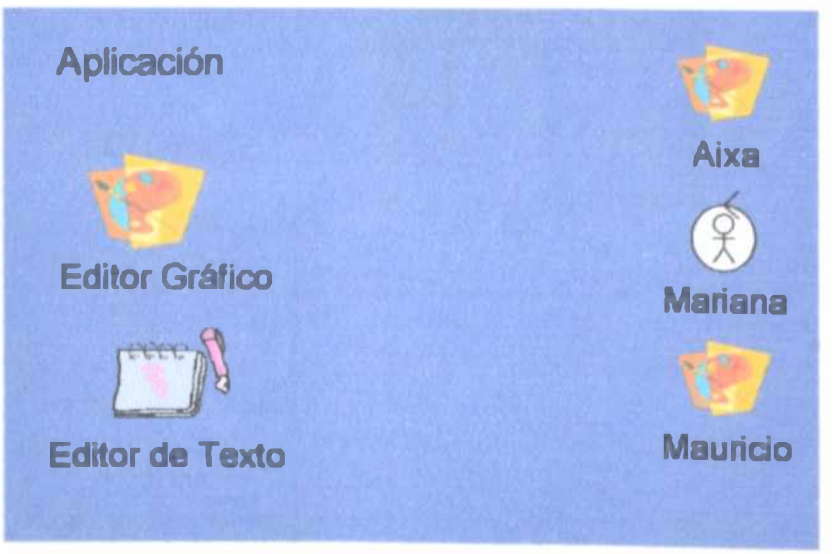


Figura 5.30 – View actualizada de todos los usuarios

5.3.3 Escenario 3 - Manipulación de Objetos

a- Crear objeto exclusivo
b- Modificar objeto exclusivo
c- Modificar objeto compartido
c.1 Simple
c.2 Compuesto

Como mencionamos en la sección 5.2.3.3 existen objetos de uso exclusivo y objetos compartidos.
Los objetos de uso exclusivo se van a encontrar almacenados tanto en el *model'* del usuario que creo los objetos así como también en el *model* del servidor, y los objetos compartidos solo se van a almacenar en el *model* del servidor.
No se permite la creación de objetos compartidos, es decir los usuarios siempre crean objetos de uso exclusivo a los cuales posteriormente le pueden conceder los permisos necesarios para otros usuarios, y de esta manera transformar un objeto de uso exclusivo en un objeto compartido (concepto ampliado en el escenario 4)
Cuando un usuario modifica un objeto de uso exclusivo tal modificación se va a realizar de manera local e inmediatamente va a ser replicada al servidor para así mantener el contenido almacenado de forma consistente.
Cuando un usuario desea modificar un objeto compartido primero debe solicitar dicho objeto al *Component Manager* del servidor correspondiente. El *Component Manager* va a verificar que el usuario tenga los permisos necesarios y que el objeto no este siendo utilizado por otro usuario.
El requerimiento que recibe entonces el *Component Manager* puede ser generar debido a que:

- El usuario desea trabajar con un objeto compartido simple: en cuyo caso el mismo es bloqueado en el servidor para evitar que algún otro usuario quiera modificarlo y se replica al *model'* del usuario para que este pueda trabajar de manera local.
- El usuario solicita trabajar con un objeto compartido compuesto, en cuyo caso deberá trabajar de manera centralizada, y el objeto será bloqueado para evitar que otros usuarios intenten modificarlo, pero el trabajo será centralizado para evitar sobrecargar a la red tanto con la replicación hacia el cliente como hacia el servidor cuando el usuario modifica el objeto, así como también las inconsistencias mencionadas en el capítulo 4.

A continuación damos un mayor nivel de detalle para cada uno de los casos planteados en este escenario

a- Crear objeto exclusivo

El usuario se encuentra trabajando de manera local en alguna componente y quiere crear un objeto, el cual puede ser simple o compuesto. Realiza entonces la creación del objeto, durante la creación del objeto cada uno de los eventos que el usuario realiza son almacenados por el *Transaction Manager* del cliente para de tal manera ir generando la transacción correspondiente que deberá replicarse al *controller* del servidor. Mientras el usuario se encuentra creando el objeto exclusivo el resto de los usuarios no observan los cambios realizados sino que

serán notificados inmediatamente apenas comience la comunicación con el *controller* del servidor, podría considerarse como que el usuario esta creando un objeto en forma asincrónica y en un determinado momento desea publicarlo para que el resto de los usuarios puedan visualizarlo, este momento seria cuando se inicia la comunicación con el *controller* del servidor.
Una vez que el usuario finaliza el proceso de creación, la transacción generada es almacenada en el *model'* local y es enviada al *Replication Manager* del cliente para que este inicie la comunicación con el servidor, enviando un requerimiento al *Replication Manager* del servidor

(1) Pasos necesarios para la creación del objeto y los cuales son registrados por el *Transaction Manager* del cliente para generar una transacción. (Figura 5.31)

(2) Una vez generada la transacción la misma es enviada al *Replication Manager* del cliente para que pueda ser replicada al servidor, y es reflejada por el *Transaction Manager* del cliente en forma local en el *model'* y almacenada en el repositorio persistente del cliente (2') (Figura 5.32).

(3) El *Replication Manager* del cliente se comunica con el *Replication Manager* del servidor y envía un requerimiento indicando información del usuario, del objeto creado y del componte al cual pertenece el objeto, así como también toda la transacción generada en (2) (Figura 5.32)

(4) El *Replication Manager* del servidor se comunica con el *Component Manager* del componente al cual pertenece el nuevo objeto para así poder registrar la existencia de un nuevo objeto como también que usuario es su dueño (Figura 5.32)

(5) El *Replication Manager* envía la información recibida en (3) al *Transaction Manager* del servidor para que este refleje los cambios efectuados en el *model.* (Figura 5.32)

(6) El *Transaction Manager* refleja la transacción en el *model* la cual en el mismo momento o posteriormente es almacenada en el repositorio persistente del servidor (6') (Figura 5.32)

(7) Recordemos que las notificaciones entre los *managers* se efectúa de manera asincrónica, es decir una vez enviada la notificación el manager que la envía puede seguir realizando en forma normal su trabajo sin esperar ningún tipo de contestación, por lo cual seguido al paso (4) el *Component Manager* notifica al *Awareness Manager* del servidor para que el mismo notifique al resto de los usuarios que se encuentran utilizando el componente los cambios realizados mediante los *Awarennes Managers* de cada uno de ellos (8) (Figura 5.33- 5.34)

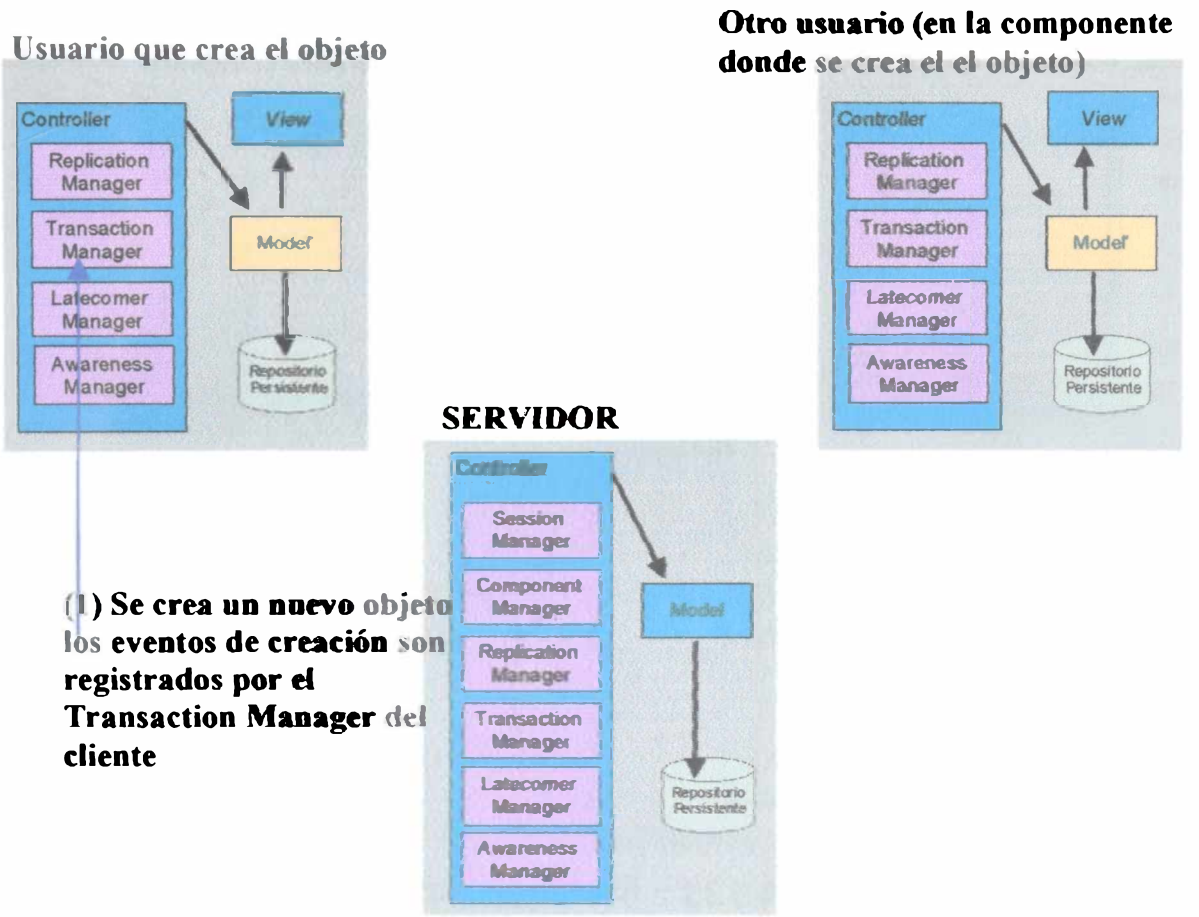


Figura 5.31 – Escenario 3 a: Paso 1

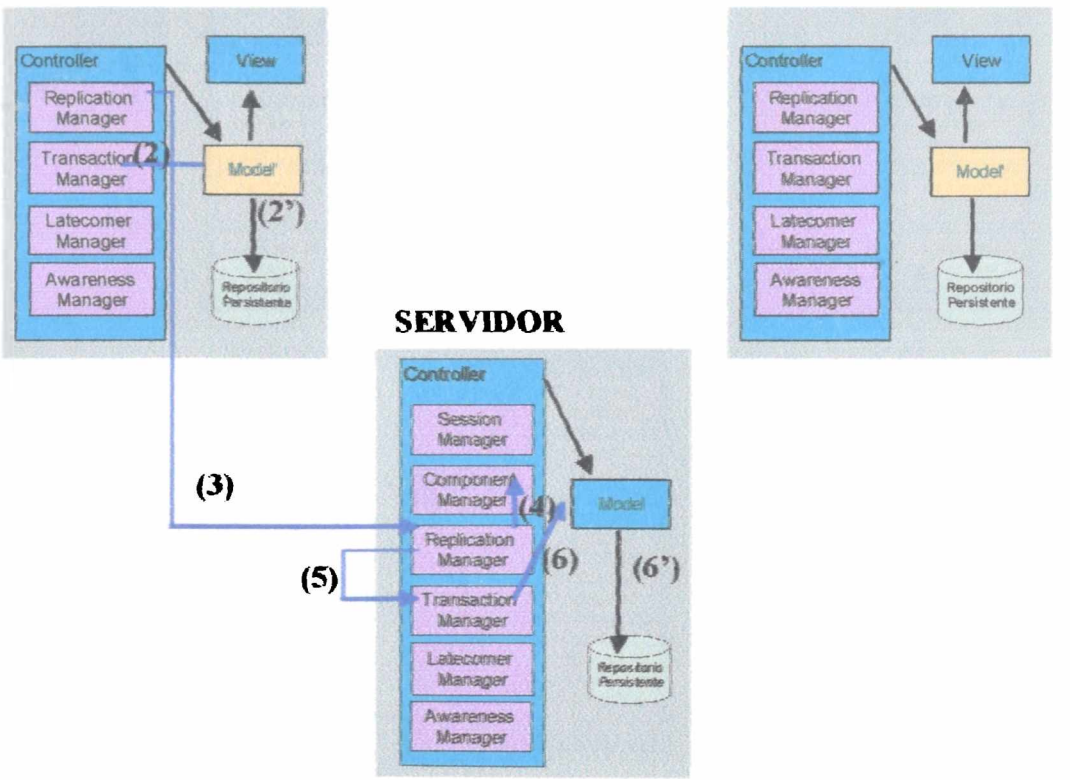


Figura 5.32 – Escenario 3 a: Pasos 2 al 6'

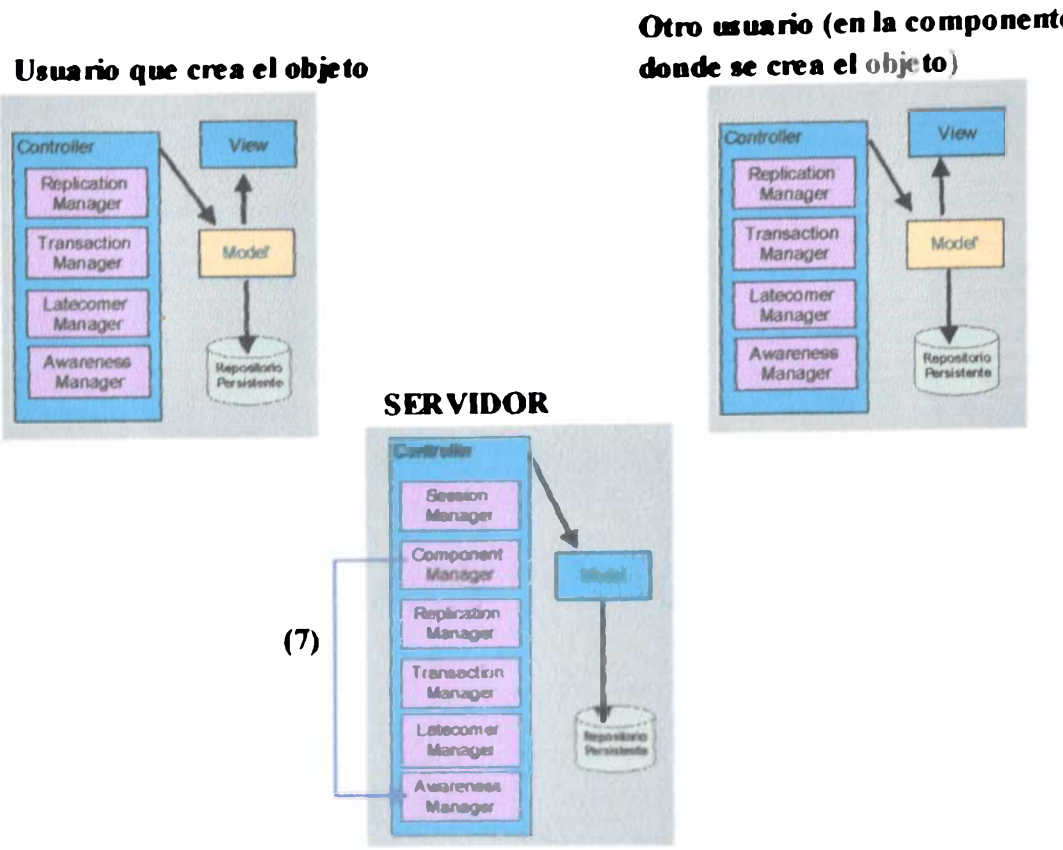


Figura 5.33 – Escenario 3 a: Paso 7

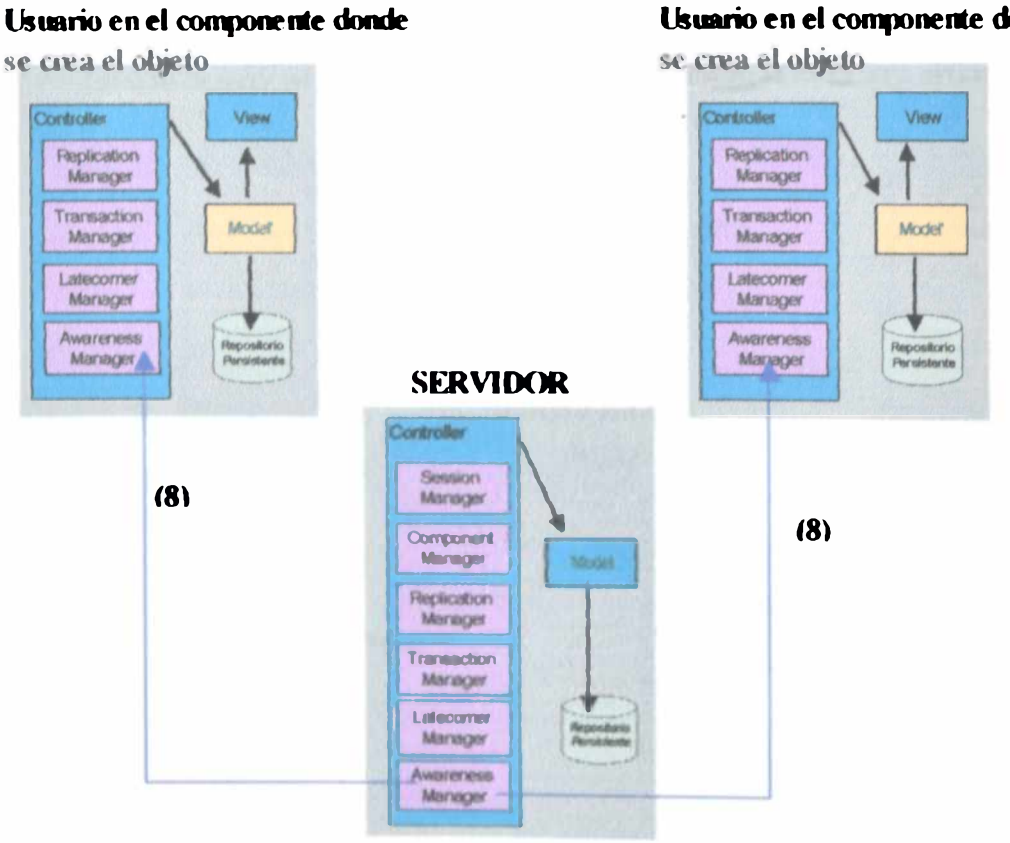


Figura 5.34 – Escenario 3 a: Paso 8

b- Modificar objeto exclusivo

La secuencia de pasos va a ser la misma que la del inciso a) con la diferencia que la transacción generada no va a ser una transacción de creación sino de modificación de un objeto. Y como el *Component Manager* del componente que contiene al objeto exclusivo ya tiene registrado que el objeto es de un determinado usuario el paso (4) va a modificarse por lo tanto la secuencia de pasos quedaría así:

(1) Pasos necesarios para la modificación creación del objeto y los cuales son registrados por el *Transaction Manager* del cliente para generar una transacción (Figura 5.35).

(2) Una vez generada la transacción la misma es enviada al *Replication Manager* del cliente para que pueda ser replicada al servidor, y es reflejada por el *Transaction Manager* del cliente en forma local en el *model'* y almacenada en el repositorio persistente del cliente (2') (Figura 5.36)

(3) El *Replication Manager* del cliente se comunica con el *Replication Manager* del Servidor y envía un requerimiento indicando información del usuario, del objeto modificado y del componte al cual pertenece el objeto, así como también toda la transacción generada en (2) (Figura 5.36)

(4) El *Replication Manager* del Servidor se comunica con el *Component Manager* del componente ya que el mismo es quien sabe que otros usuarios se encuentran utilizando el componente y a los cuales se les debe notificar los cambios realizados a través del *Awarennes Manager* del servidor (Figura 5.36)

(5) El *Replication Manager* envía la información recibida en (3) al *Transaction Manager* del servidor para que este replique los cambios efectuados en el modelo. (Figura 5.36)

(6) El *Transaction Manager* refleja la transacción en el *model* y en el mismo momento o posteriormente es almacenada en el repositorio persistente del servidor (6') (Figura 5.36)

(7) Seguido al paso (4) el *Component Manager* notifica al *Awareness Manager* del servidor para que el mismo notifique al resto de los usuarios que se encuentran utilizando el componente los cambios realizados mediante los *Awarennes Managers* de cada uno de ellos (8) (Figura 5.37-5.38)

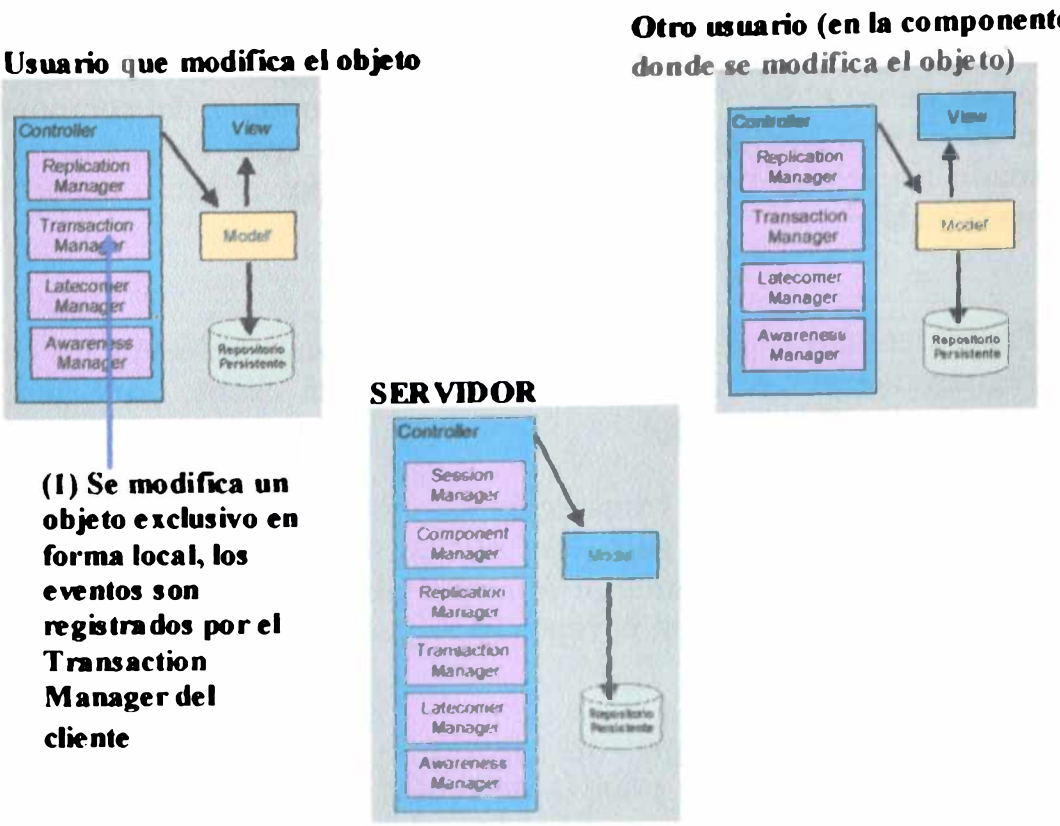


Figura 5.35 – Escenario 3 b: Paso 1

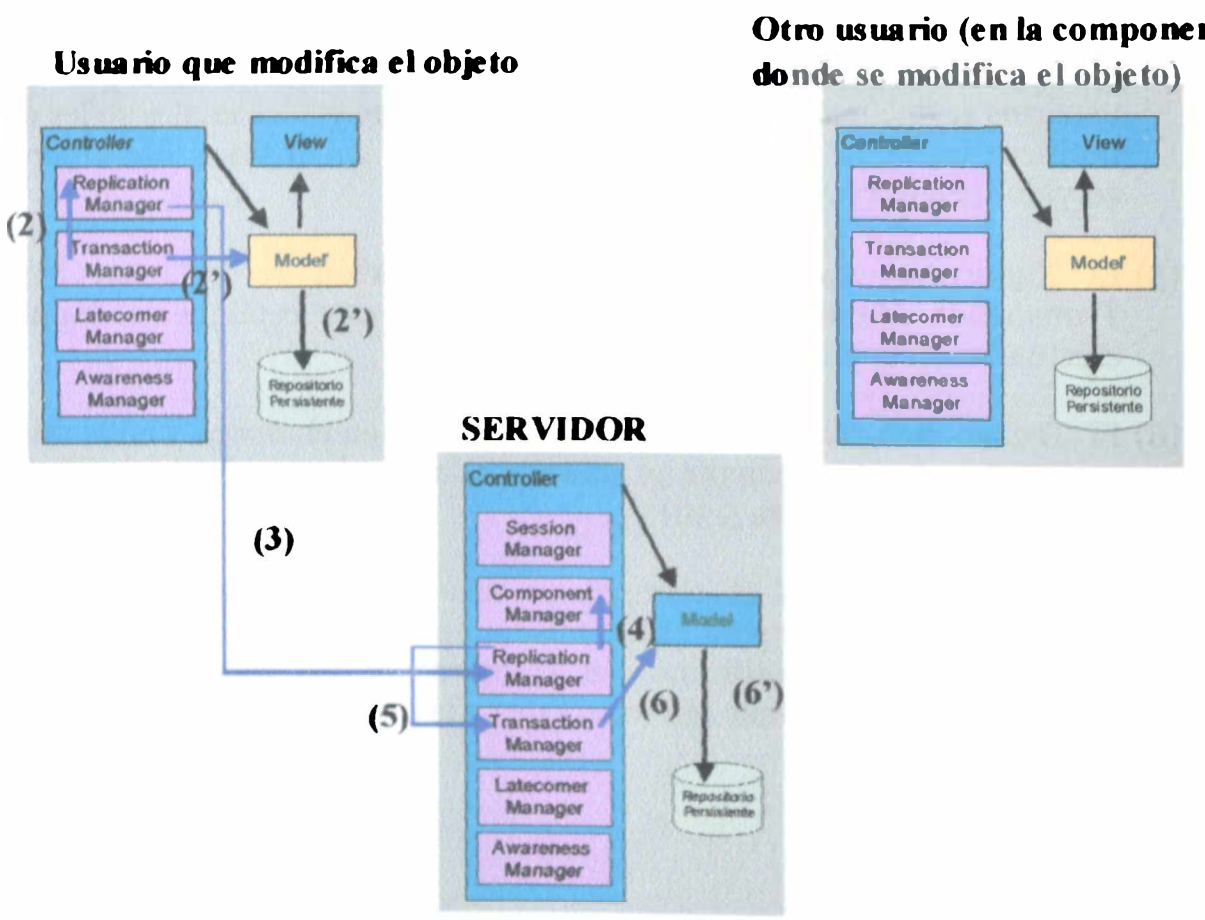


Figura 5.36 – Escenario 3 b: Pasos 2 al 6'

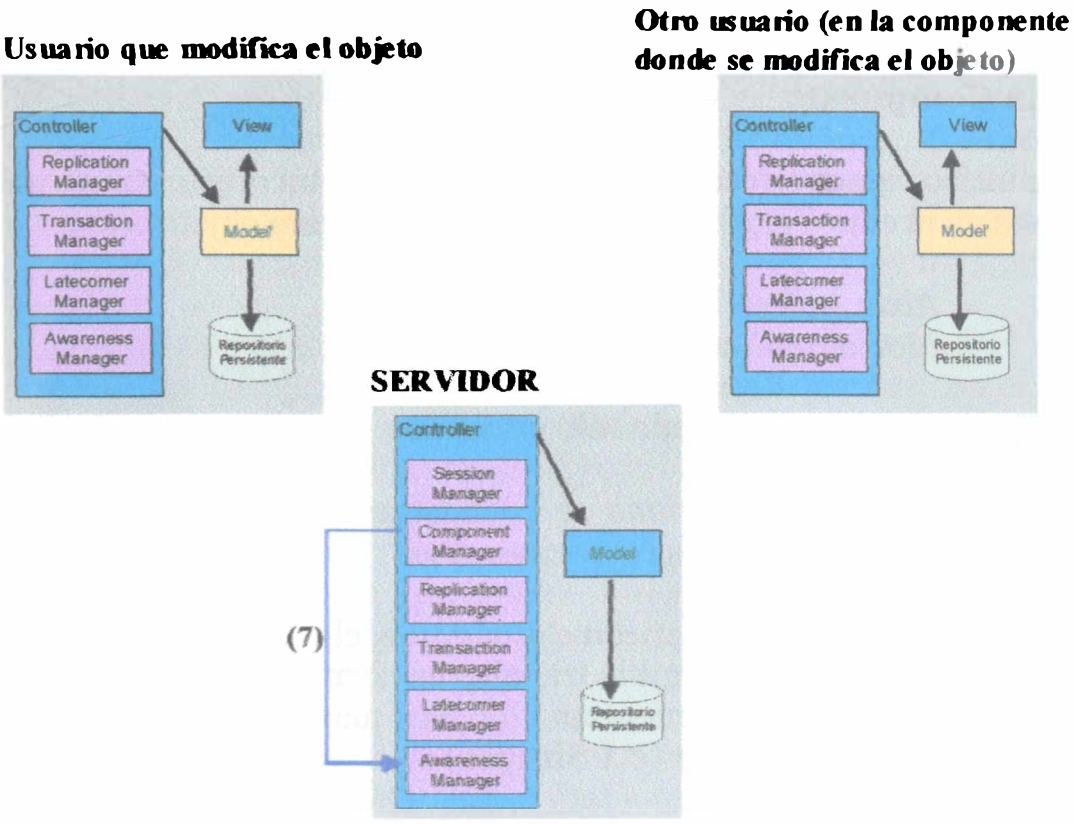


Figura 5.37 – Escenario 3 b: Paso 7

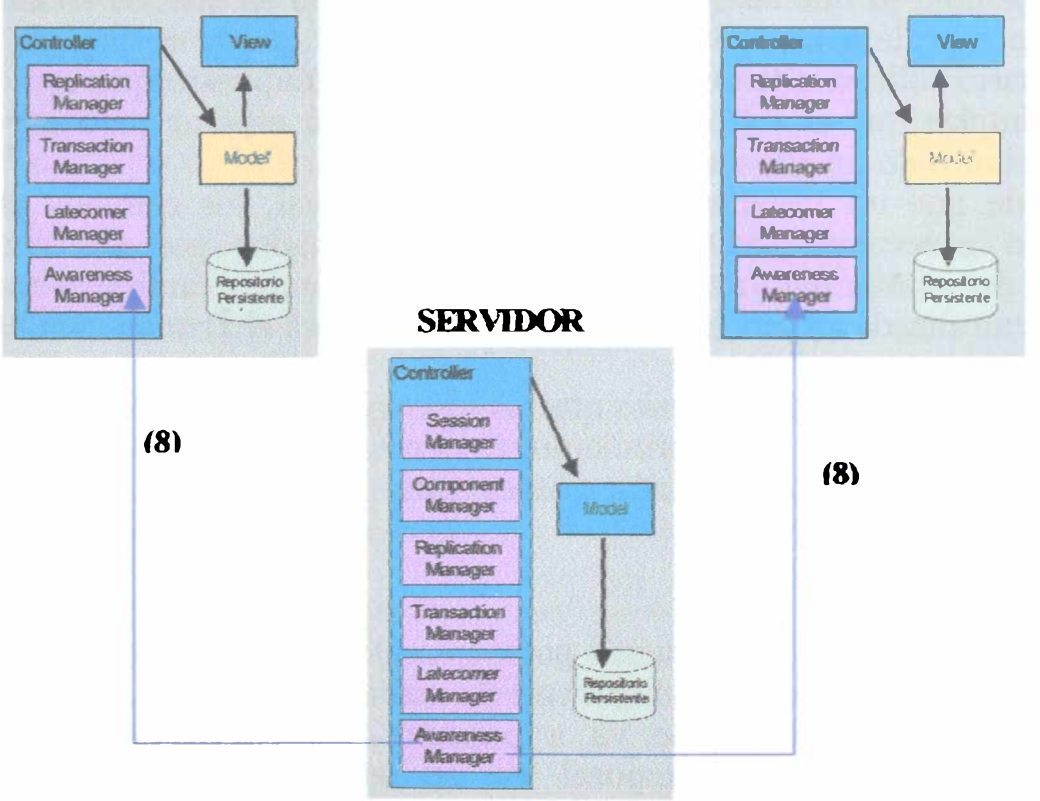


Figura 5.38 – Escenario 3 b: Paso 8

c- Modificar objeto compartido
c.1 Simple
c.2 Compuesto

Para este inciso vamos a recordar que el Objeto 1 era un objeto compartido que se encontraba en el editor grafico, y los permisos sobre este objeto eran:

Aixa: Acceso Total
Mariana: Cuadrado acceso Total
Rectángulo acceso Total
Triángulo solo visualiza

Supongamos que *Aixa* y *Mariana* se encuentran trabajando dentro del editor gráfico, y realizan los siguientes requerimientos:

- *Mariana*: quiere trabajar con el Cuadrado, el cual es un objeto simple, por lo tanto se le va a permitir trabajar en forma local, y el mismo se va a mantener bloqueado mientras ella esta manipulándolo para evitar que *Aixa* lo pueda modificar (**c.1 Simple)**

- *Aixa*: solicita trabajar con el Rectángulo, el cual es un objeto compuesto, por lo cual su trabajo se va a realizar de manera centralizada, pero se podrá atender este requerimiento una vez que *Mariana* haya finalizado la modificación del Cuadrado ya que el mismo es parte del objeto Rectángulo. (**c.2 Compuesto)**

En este ejemplo lo que se quiere mostrar claramente es que los usuarios solo podrán trabajar de manera local con aquellos objetos que sean simples, y que si algún usuario solicita estos objetos o algún objeto compuesto que contenga los objetos simples que están en uso, el mismo deberá esperar para así de esta manera evitar todo tipo de inconsistencias.
En caso de que un usuario trabaje de manera local, los cambios que vaya efectuando generaran eventos que serán almacenados por el *Transaction Manager* del cliente para así generar una transacción, en cambio si trabaja de manera centralizada, quien generara la transacción será el *Transaction Manager* del servidor.
Para nuestro ejemplos supongamos que los objetos están desbloqueados a la hora de que *Mariana* inicia el requerimiento, es decir no están siendo utilizados por ningún otro usuario de la aplicación colaborativa.

c.1 Simple

Como mencionamos anteriormente para ejemplificar este escenario vamos a suponer que *Mariana* quiere trabajar con el cuadrado, el cual es un objeto simple compartido, por lo cual se le va a permitir que trabaje en forma local, almacenándolo en forma temporal en su *model* para así agilizar las modificaciones que la misma quiere realizar, y se va a tener que indicar de alguna manera que ella esta utilizando este objeto compartido así si otro usuario lo quiere modificar, no puede hacerlo hasta que *Mariana* termine con los cambios necesarios.

Secuencia de pasos:

(1) Se establece la comunicación entre el *controller* del cliente (*Mariana*) y el *controller* del servidor mediante un requerimiento de utilización de un objeto compartido en este caso el Objeto2. Este requerimiento es recibido por el *Component Manager* del componente que contiene tal objeto, para nosotros el *Component Manager* del Editor Gráfico (Figura 5.39)

(2) El *Component Manager* debe verificar dos cosas: primero que el usuario que solicita el objeto tiene permisos para poder trabajar con el, segundo que el objeto no se encuentra bloqueado porque otro usuario esta trabajando con el, o esta trabajando con un objeto compuesto compartido que lo contiene (Figura 5.39).

(3) En ambos casos si no se cumplen las condiciones necesarias, es decir tener los permisos y que el objeto no este en uso, tal situación se notificara al cliente (Figura 5.39).

(4) En caso de que el usuario tenga los permisos necesarios para utilizar el objeto y el mismo se encuentre desbloqueado, el *Component Manager* envía al *Transaction Manager* el requerimiento para que sea reflejado en el *model* del servidor (Figura 5.39).

(5) El *Transaction Manager* genera una nueva transacción la cual indica fecha y hora, objeto y usuario que va a modificar el objeto y la misma va a ser reflejada en el *model* y en el mismo momento o posteriormente almacenada en el repositorio persistente del servidor (5') (Figura 5.39).

(6) El *Component Manager* bloquea el objeto para que otros usuarios no lo puedan modificar, y notifica al *Replication Manager* del servidor que objeto debe replicar y a que usuario (Figura 5.40).

(7) El *Replication Manager* del servidor replica el objeto al *Replication Manager* del cliente, el cual a partir de este momento podrá trabajar en forma local con el objeto solicitado (Figura 5.40).

(8) El *Replication Manager* del cliente es el encargado de que el objeto se replique temporalmente en el *model'* del cliente para así poder ser manipulado posteriormente. Vale aclarar que este objeto no se almacena de manera persistente en el repositorio local (Figura 5.40).

(9) El *Component Manager* le comunica a todos los usuarios del componente que el objeto se encuentra en uso a través del *Awareness Manager* del servidor que se va a comunicar con los *Awareness Managers* de los clientes (10) (Figura 5.40)

(11) Cuando el usuario finaliza de utilizar el objeto tal situación debe ser notificada al *Component Manager* para que el mismo desbloquea el objeto y se comunica y se deben replicar los cambio realizados, por lo cual a partir de acá los pasos son los mismos que si se tratase de la modificación de un objeto exclusivo (pasos explicados en el escenario 3 inciso b) (Figura 5.40)

Como ya mencionamos anteriormente el trabajo de los *managers* es asincrónico, ninguno se queda esperando contestaciones de otro manager, por lo cual el orden de la secuencia de pasos puede ser modificado siempre y cuando se respecte la secuencia de ejecución de acciones que realiza cada *manager*, es decir por ejemplo el paso 9 no se puede realizar antes del paso 6 ya que los mismos son realizados por el *Component Manager* y antes de avisar que un objeto se encuentra en uso se debe bloquear.

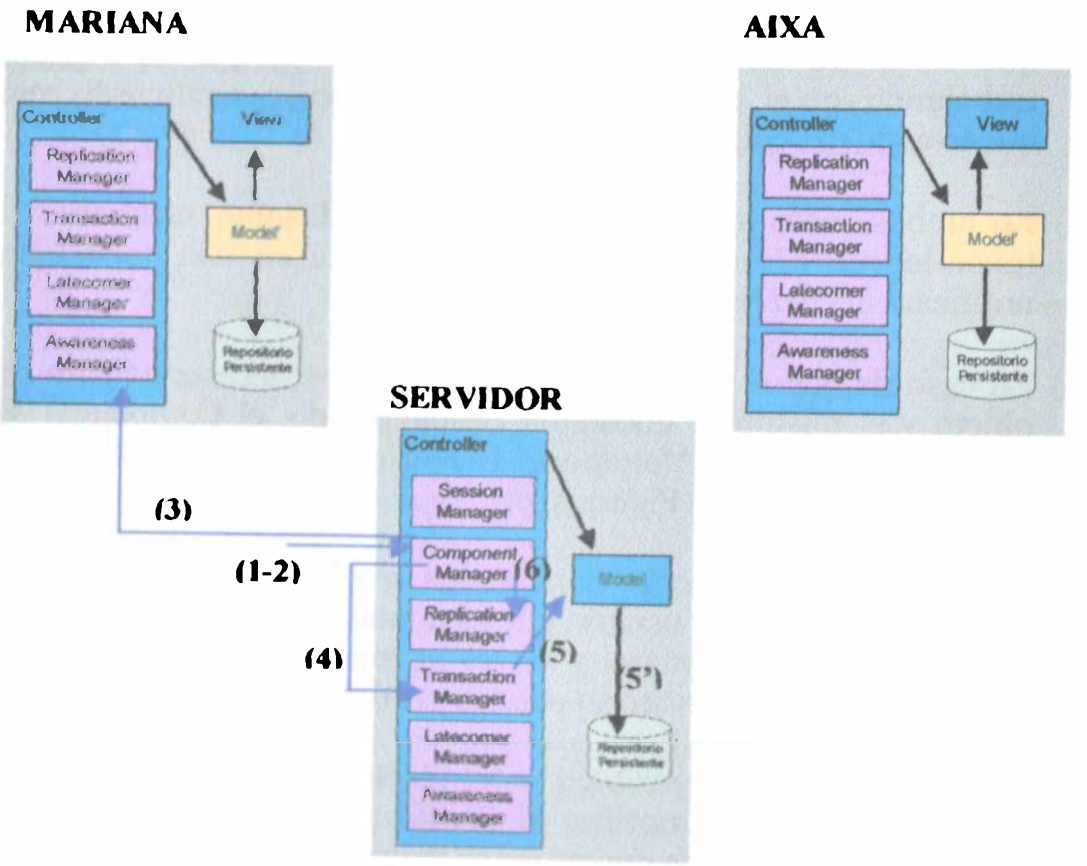


Figura 5.39 – Escenario 3 c.1 : Pasos 1 al 6

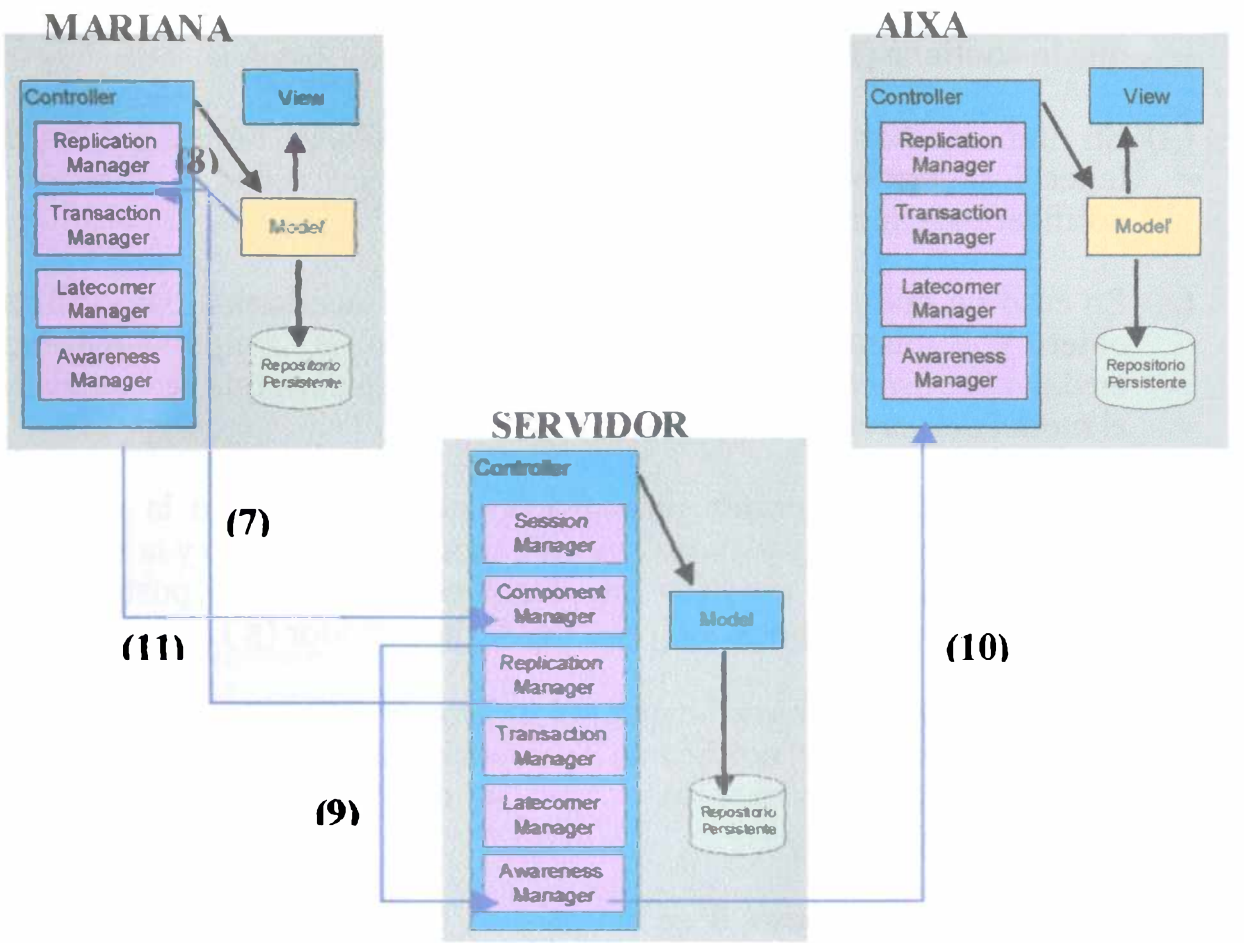


Figura 5.40 – Escenario 3 c.1 : Pasos 7 al 11

c.2 Compuesto

Supongamos ahora que *Aixa* solicita trabajar con el Rectángulo, el cual es un objeto compuesto formado por dos objetos simples (Rectángulo y Cuadrado), como mencionamos anteriormente todo el trabajo se va a realizar en forma centralizada, por lo cual los *managers* que van a intervenir en esta situación son los del Servidor.
Como en el caso **c.1** el trabajo de *Aixa* se va a encontrar demorado si el Rectángulo o directamente el Objetos2 se encuentra siendo modificado por otro usuario. Cabe aclarar que si un usuario solicito utilizar el Objeto 2 como el mismo es un objeto compuesto se bloquea todo el objeto completo, es decir cada uno de los objetos simples y compuestos que lo componen.

Secuencia de Pasos:

(1) Se establece la comunicación entre el *controller* del cliente (*Aixa*) y el *controller* del servidor mediante un requerimiento de utilización de un objeto compartido en este caso el Objeto2. Este requerimiento es recibido por el *Component Manager* del componente que contiene tal objeto, para nosotros el *Component Manager* del Editor Grafico (Figura 5.41).

(2) El *Component Manager* debe verificar dos cosas: primero que el usuario que solicita el objeto tiene permisos para poder trabajar con el, segundo que el objeto no se encuentra bloqueado porque otro usuario esta

trabajando con el, o esta trabajando con un objeto compuesto compartido que lo contiene (Figura 5.41).

(3) En ambos casos si no se cumplen las condiciones necesarias, es decir tener los permisos y que el objeto no este en uso, tal situación se notificará al cliente (Figura 5.41).

(4) En caso de que el usuario tenga los permisos necesarios para utilizar el objeto y el mismo se encuentre desbloqueado, el *Component Manager* envía al *Transaction Manager* el requerimiento para que sea reflejado en el *model* del servidor (Figura 5.41).

(5) El *Transaction Manager* genera una nueva transacción la cual indica fecha y hora, objeto y usuario que va a modificar el objeto y la misma va a ser reflejada en el *model* y en el mismo momento o posteriormente almacenada en el repositorio persistente del servidor (5') (Figura 5.41).

(6) El *Component Manager* bloquea el objeto para que otros usuarios no lo puedan modificar, y notifica al *Transaction Manager* que usuario va a empezar a generar eventos sobre ese objeto, que el mismo debera almacenar (Figura 5.41).

(7) A partir de este paso 6 se empieza a trabajar de forma distinta al escenario **3.c1.** Todos los eventos que surjan de modificaciones que realiza el usuario (*Aixa*), van a ser almacenadas por el *Transaction Manager* del servidor para generar posteriormente una transacción que será reflejada en el *model* del servidor (Figura 5.41).

(8) Una vez que el usuario finaliza de utilizar el objeto la transacción generada es reflejada en el *model* del servidor y en el mismo momento o posteriormente es almacenada en el repositorio persistente del servidor (8') (Figura 5.41).

(9) En el momento en el cual el objeto fue bloqueado por el *Component Manager* este le comunica a todos los usuarios del componente que el objeto se encuentra en uso a través del *Awareness Manager* del servidor que se va a comunicar con los *Awareness Managers* de los clientes (10) (Figura 5.42)

(11) Cuando el usuario finaliza la utilización del objeto compartido compuesto envía una notificación al *Component Manager* para que este desbloquee el objeto y comunique tal situación al Awarenes Manager del servidor (12) para que notifique a los clientes como en los paso 10 (13) (Figura 5.42)

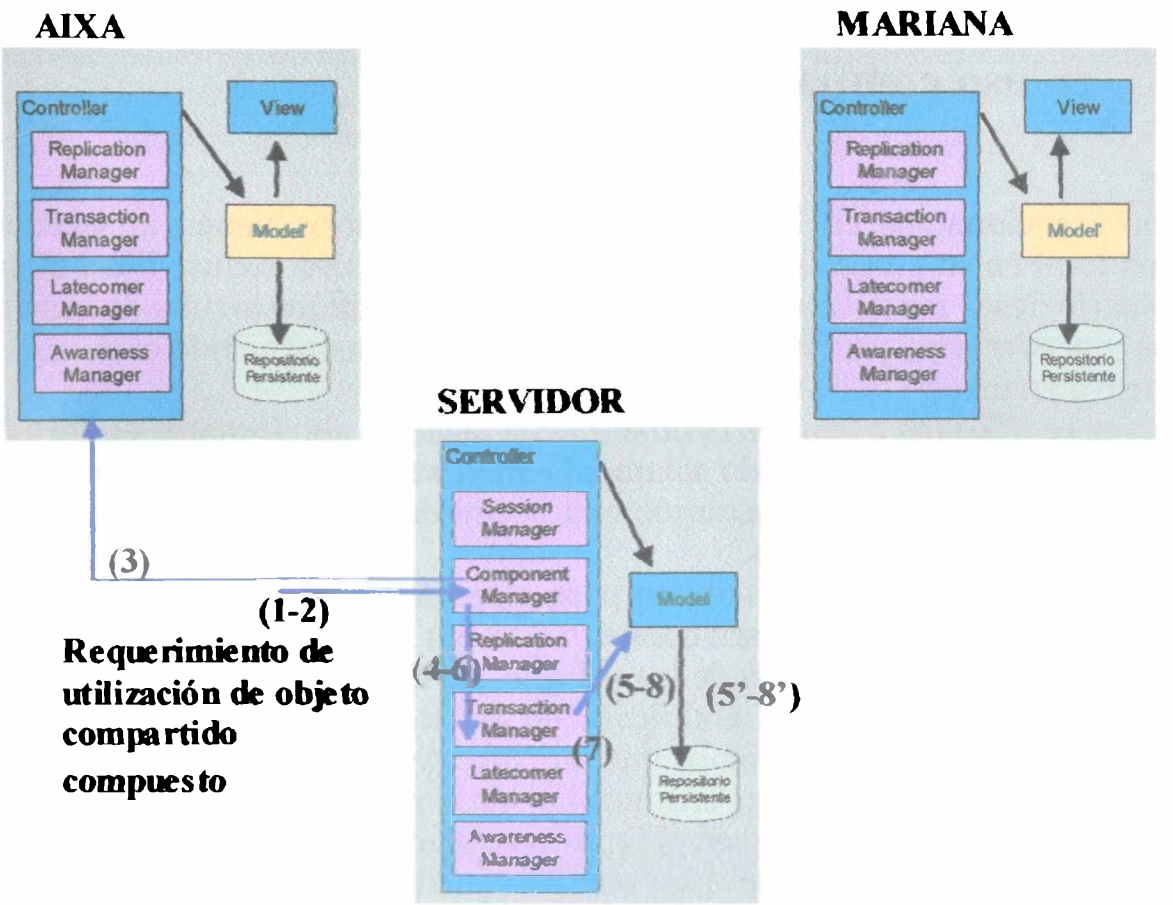


Figura 5.41 – Escenario 3 c.2 : Pasos 1 al 8'

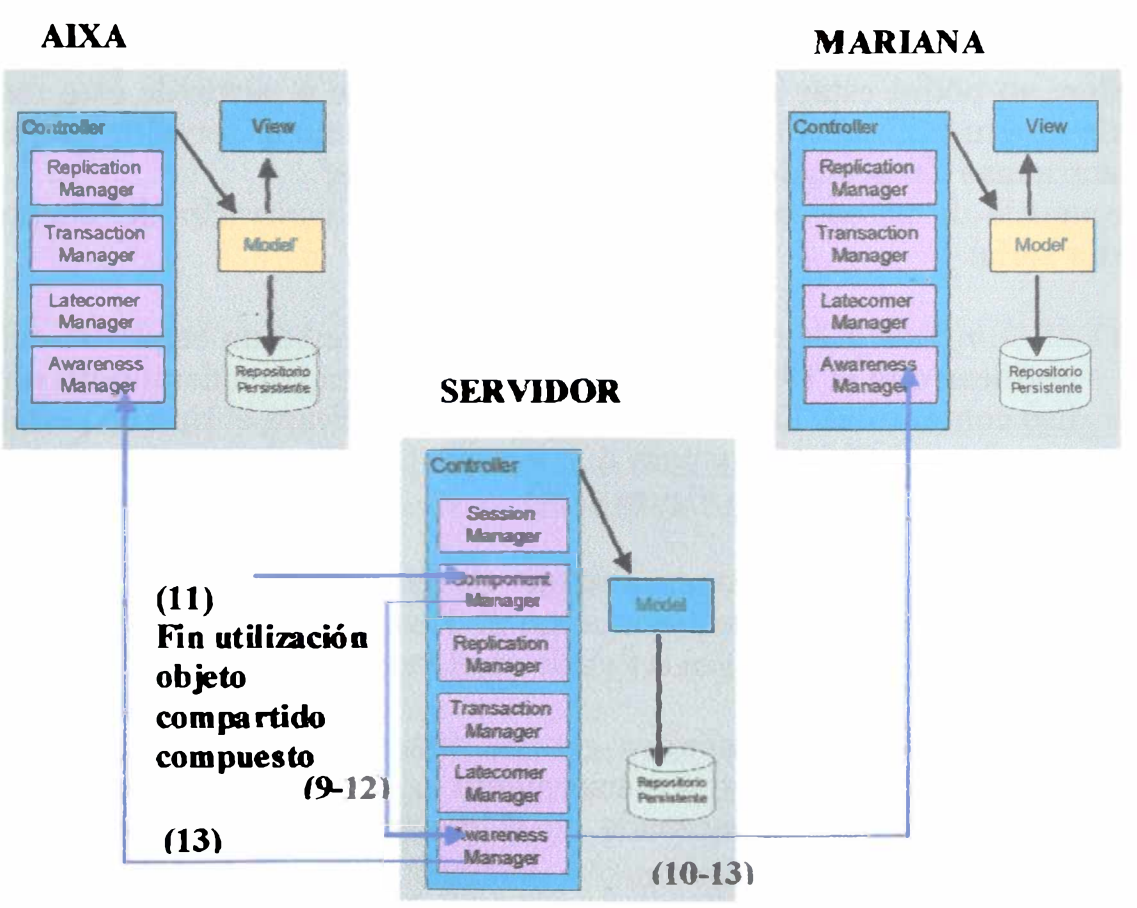


Figura 5.42 – Escenario 3 c.2 : Pasos 9 al 13

5.3.4 Escenario 4 - Convertir un objeto de uso exclusivo en uso compartido

Como mencionamos en la sección 5.2.3.3 los usuarios tienen almacenado en forma local en su *model'* aquellos objetos que son de uso exclusivo del mismo. Si el usuario desea permitir que algún otro usuario modifique alguno de sus objetos, debe concederle los permisos adecuados, y estará transformando el objeto de uso exclusivo en un objeto de uso compartido.
Una vez cedidos los permisos el objeto debe dejar de almacenarse en forma local para pasar a estar almacenado solamente en el servidor para así poder controlar que cuando alguno de los usuarios con permisos este modificando el objeto el resto tenga bloqueado su uso.
Cabe destacar que si bien el objeto pasa a residir solamente en el servidor por estar compartido no necesariamente todos los usuarios tienen permiso para modificarlo, y cuando alguno de los usuarios desee modificarlo dependiendo del tipo de requerimiento podrá modificarlo en forma centralizada o de manera local.
Para nuestro ejemplo recordemos que *Mariana* tenia acceso total sobre el Objeto 2 por lo cual el mismo se encontraba almacenado tanto en el repositorio persistente de *Mariana* como en el repositorio persistente del servidor.
Mariana quiere permitir que *Aixa* también pueda realizar modificaciones sobre el Objeto 2 por lo cual le otorgara los permisos adecuados.
La modificación de permisos se realiza de forma local, y una vez que fue finalizada recién se establece la comunicación entre el *controller* del cliente y el *controller* del servidor, mediante un requerimiento al *Component Manager* del componente que contiene al Objeto 2 (*Component Manager* del Editor Gráfico).
Una vez que *Mariana* le concede permisos a *Aixa*, y envía el requerimiento al servidor, no podrá estar utilizando el objeto, ya que a partir de este momento cuando quiera utilizar el objeto debe solicitar el mismo al servidor ya que no se encuentra mas almacenado en forma local en su *model'*.
La secuencia de pasos en la comunicación entre el *controller* del cliente y del servidor sería:

(1) El *Component Manager* del editor gráfico recibe un requerimiento para notificar que el Objeto 2 a partir de ahora sea considerado un objeto de uso compartido. En este requerimiento se indicara el tipo de permiso que se asigno al Objeto 2 y para que usuario (para nuestro ejemplo permisos asignados para *Aixa*) (Figura 5.43)

(2) El *Component Manager* atiende el requerimiento, registra el nuevo permiso, y envía el requerimiento al *Transaction Manager* del servidor para que este cambio sea reflejado en el *model* (Figura 5.43).

(3) El *Transaction Manager* genera una nueva transacción la cual es enviada al Modelo y posteriormente o en el mismo momento dicha transacción será almacenada en el repositorio persistente del Servidor (Figura 5.43).

Antes o durante la realización de estos pasos el Objeto2 debe ser eliminado del *model'* perteneciente a *Mariana*.

Este escenario no requiere ningún tipo de modificación en la vista de los usuarios. Finalizados estos pasos cualquier usuario que quiera utilizar el Objeto2 deberá solicitarlo al servidor ya que el mismo paso a estar almacenado solamente en forma centralizada, es decir dejo de ser un objeto de uso exclusivo de *Mariana*.

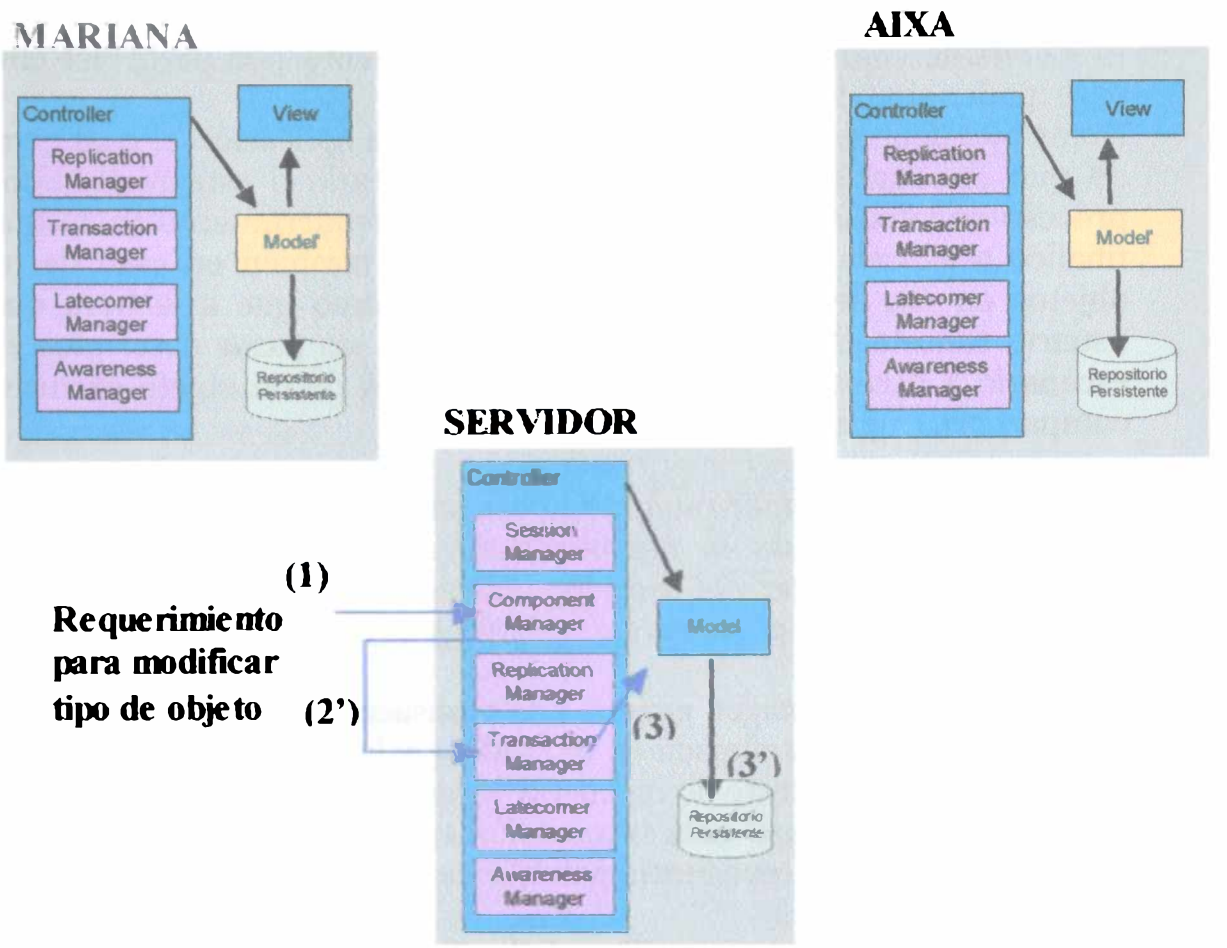


Figura 5.43 – Escenario 4 : Pasos 1 al 3'

5.3.5 Escenario 5 - Usuario que abandona un componente

Cuando un usuario abandona un componente este cambio se debe reflejar en el *model* del servidor para posteriormente cuando se inicie un proceso de latecoming se tenga los datos correspondiente al ingreso/egreso de los usuarios al componente.
El registro del abandono de un componente se realizara de forma tal de posteriormente permitir saber a que fecha/hora, que usuario abandono determinado componente. Es decir vamos a generar una transacción de ingreso/egreso de un componente (ver 5.2.4)
Cuando un cliente quiere abandonar una componente, el requerimiento correspondiente es recibido por el *Component Manager* del componente en cuestión el cual atenderá tal requerimiento.
Al momento de querer abandonar un componente el usuario no debe tener ningún objeto en uso ni debe estar efectuando un proceso de latecoming, en ambos casos deberá primero finalizar o cancelar las tareas que se encuentra realizando antes de proceder al abandono del componente.

Para nuestro ejemplo supongamos que *Mauricio* decide abandonar el Editor Gráfico, y el mismo no se encontraba trabajando con ningún objeto, ni estaba en realizando un proceso de latecoming.

(1) Como en todos los escenarios se establece una comunicación entre el *controller* del Cliente (*Mauricio*) y el *controller* del servidor.
El *Component Manager* recibe un requerimiento por parte del cliente para poder abandonar el componente.
Notar que debe existir algún mecanismo de la aplicación el cual en caso de que el usuario tenga algún objeto bloqueado o este realizando un proceso de latecoming no permita enviar tal requerimiento hasta que se finalice o cancele con las tareas pendientes, también en caso de tener objetos en uso se podría proveer un mecanismo que a la hora que el usuario cancele el uso efectué implícitamente los pasos descriptos en el escenario 3 correspondiente a la modificación de un objeto exclusivo o compartido. (Figura 5.44)

(2) El *Component Manager* remueve al usuario de la lista de usuarios que se encuentran utilizando la componente y notifica el requerimiento al *Transaction Manager* del servidor indicando fecha y hora en la cual el usuario abandona el componente. (Figura 5.44)

(3) El *Transaction Manager* genera una transacción de ingreso/egreso de componente que es reflejada en el Modelo, y la cual será posteriormente utilizada cuando se trate de regenerar la historia del componente para un proceso de latecoming. Esta transacción en forma inmediata o posterior es almacenada en el repositorio persistente (3') (Figura 5.44)

(4) El *Component Manager* notifica al *Session Manager* para que también registre el abandono del componente por parte del usuario (Figura 5.45)

(5) El *Session Manager* notifica al *Awareness Manager* del servidor el cambio realizado para que este notifique a los *Awareness Managers* de los clientes (6) (Figura 5.45)

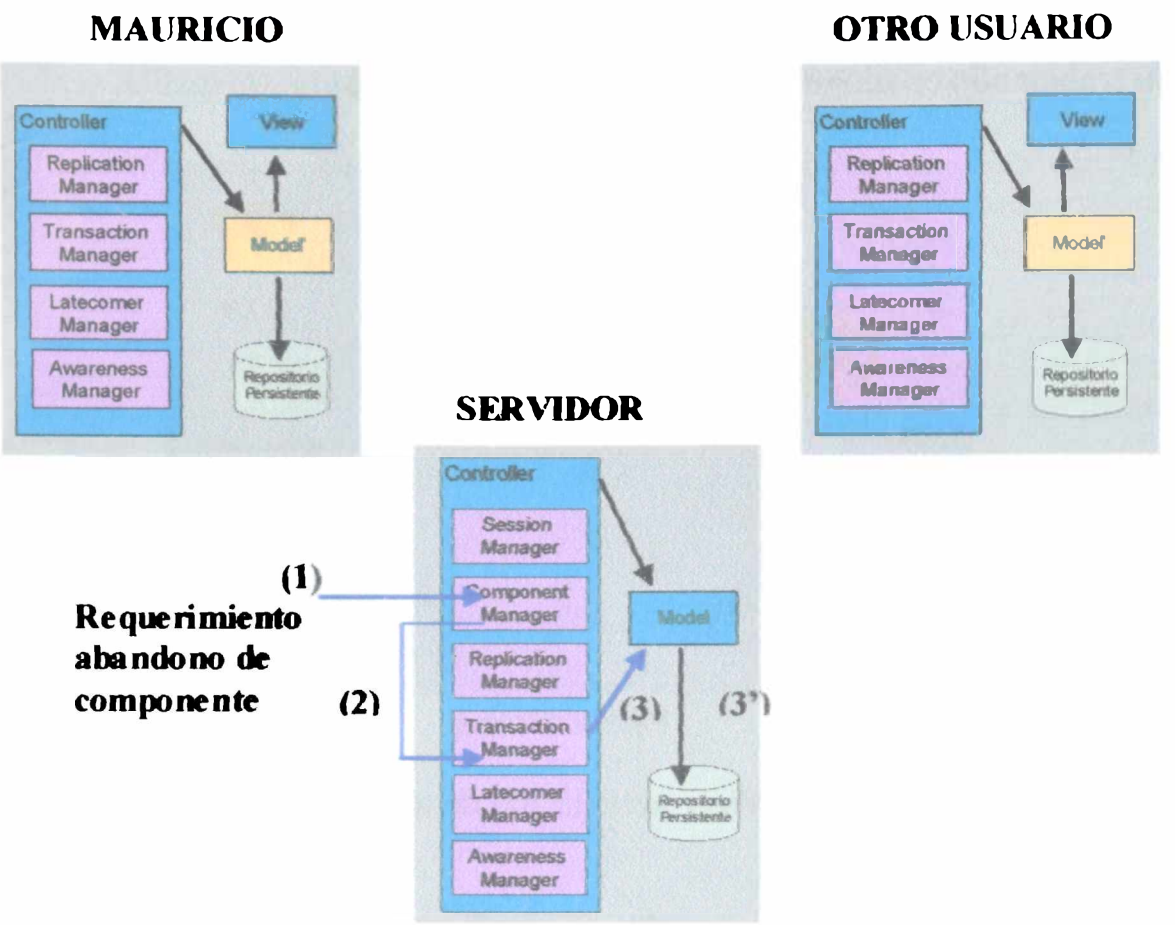


Figura 5.44 – Escenario 5: Pasos del 1 al 3'

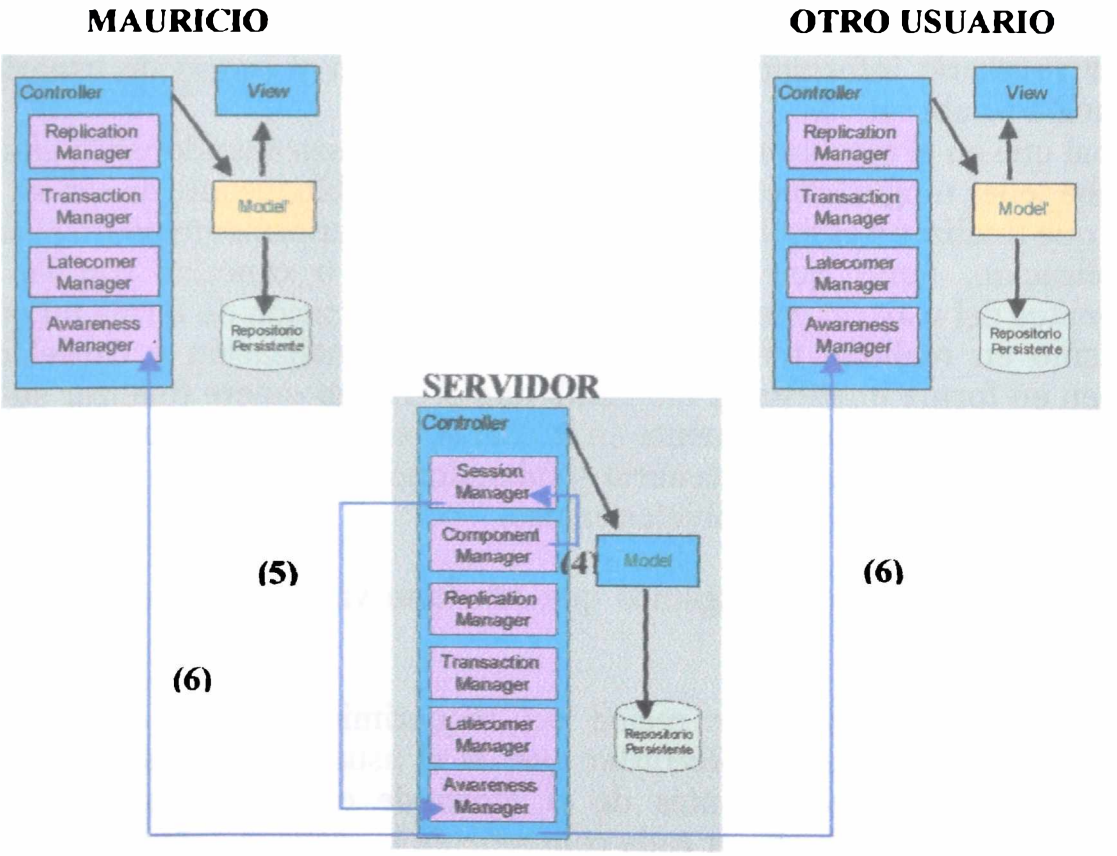


Figura 5.45 – Escenario 5: Pasos del 4 al 6

Una vez que *Mauricio* abandona el Editor Gráfico, en la vista de los tres usuarios se observa que *Aixa* se encuentra en el Editor Grafico, y *Mariana* y *Mauricio* no están trabajando en ningún componente. (Figura 5.46)

Figura 5.46 – View actualizada de todos los usuarios

5.3.6 Escenario 6 - Usuario que abandona la sesión

Cuando un usuario va a dejar de utilizar la aplicación colaborativa este cambio debe ser notificado al resto de los usuarios, y será reflejado en el *model* y almacenado en el repositorio persistente del servidor para así poder tener en caso de ser necesario información relacionada al ingreso y egreso de usuarios a la aplicación colaborativa.
Al igual que en el escenario anterior si un usuario desea abandonar la aplicación, no debe tener ningún objeto de ningún componente bloqueado, de ser así deberá tener que finalizar o cancelar el uso del objeto y posteriormente podrá abandonar la aplicación, notar que el proceso de finalizar o cancelar con las tareas pendientes del usuario puede ser provisto por la aplicación de forma tal que para el mismo en realidad todas las operaciones de finalización o cancelación se realicen en forma implícita al momento que el mismo quiere finalizar su sesión. De ser así, para cada componente en la cual el mismo se encuentre se realizarán los pasos descriptos en el escenario 5 (abandono de un componente) o los del escenario 3 en caso de tener objetos en uso.

Para nuestro ejemplo supongamos que *Mariana* va a abandonar la aplicación colaborativa.

(1) El *Session Manager* recibe un requerimiento para que el usuario abandone la sesión. Recordar que si el usuario se encuentra utilizando algún componente antes de el envío de este requerimiento se debe proceder a realizar por cada componente la secuencia de pasos descriptas en el escenario 5. (Figura 5.47)

(2) El *Session Manager* envía el requerimiento al *Transaction Manager* del servidor para reflejar en el *model* del servidor una transacción especificando fecha y hora del fin de la sesión de usuario. (Figura 5.47)

(3) El *Transaction Manager* refleja la nueva transacción en el *model* la cual posteriormente o inmediatamente será almacenada en el repositorio persistente del servidor (3'). (Figura 5.47)

(4) Al mismo momento que el *Session Manager* notifica al *Transaction Manager* notifica al *Awareness Manager* del servidor para que el mismo notifique a todos los usuarios con los cambios realizados a través de sus *Awareness Managers* (5). (Figura 5.48-5.49)

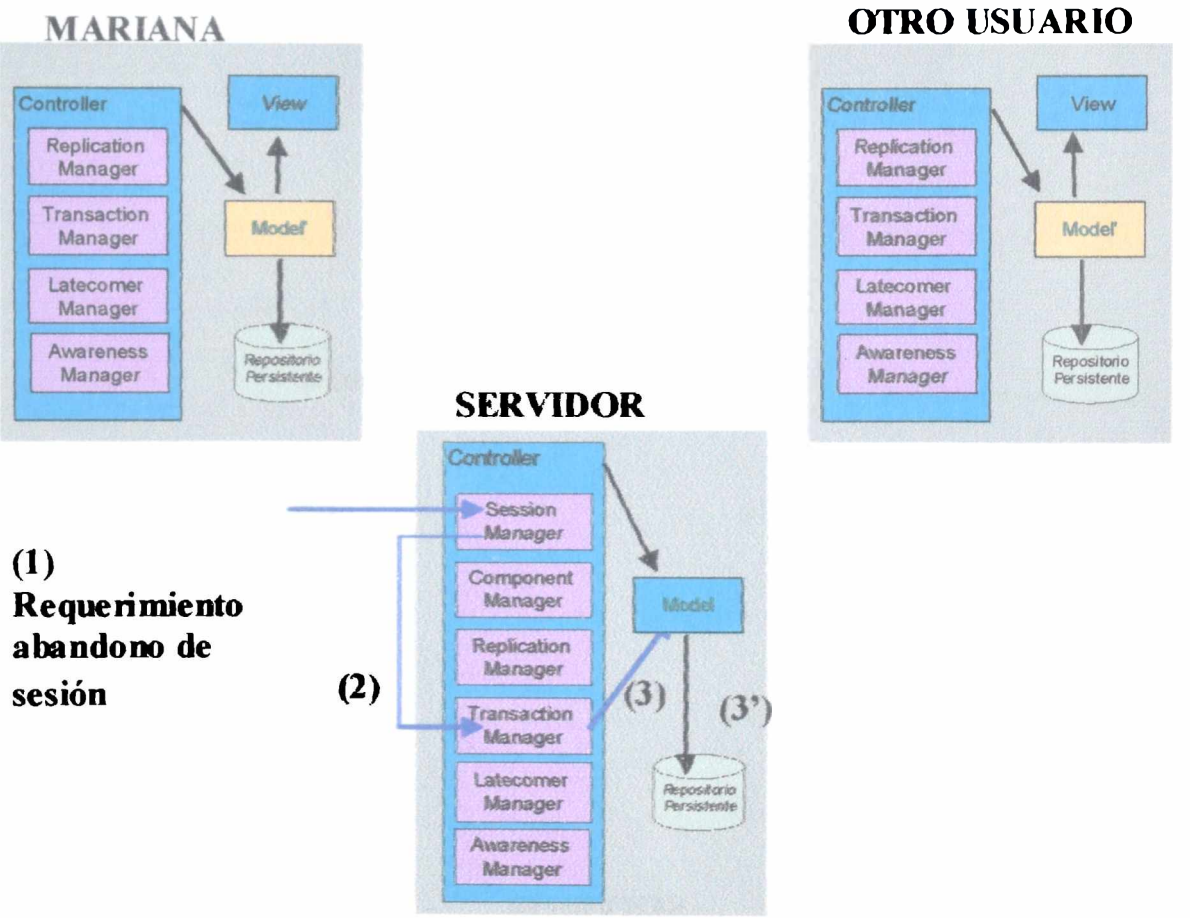


Figura 5.47 – Escenario 6: Pasos del 1 al 3'

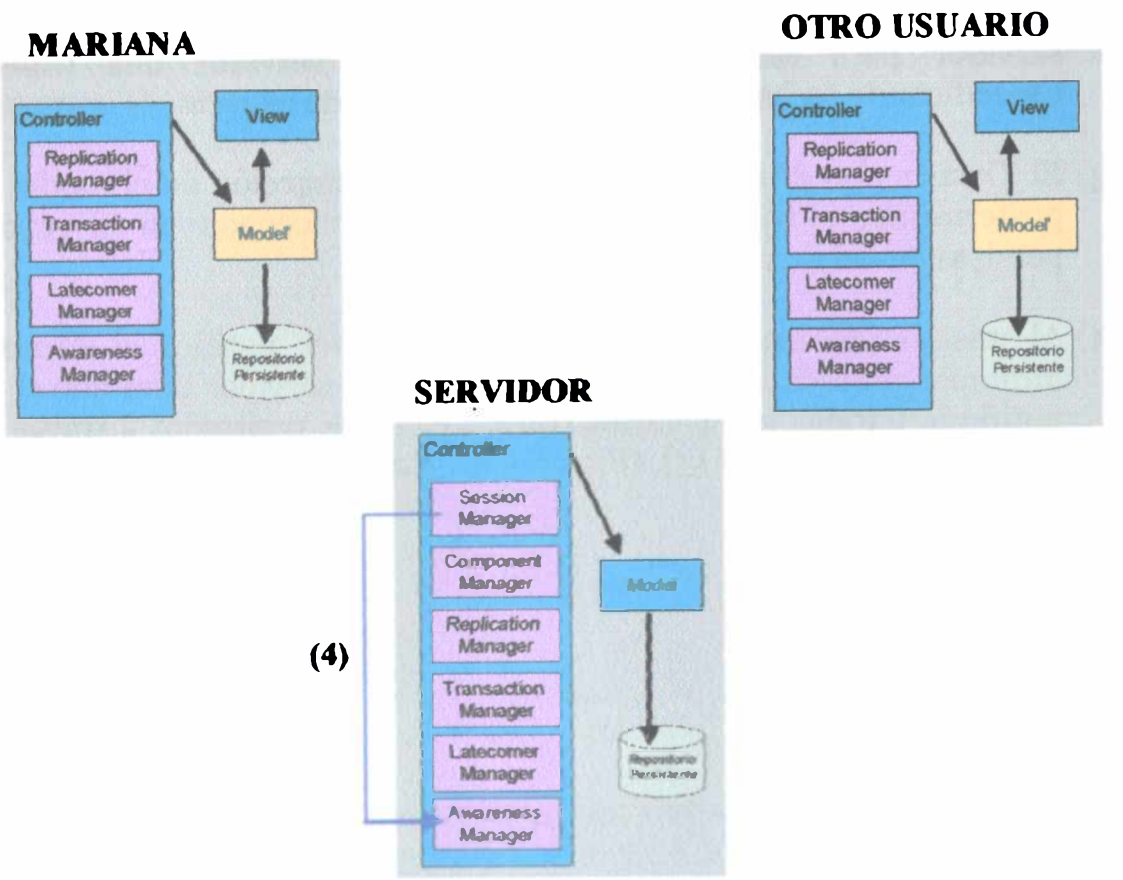


Figura 5.48 – Escenario 6: Paso 4

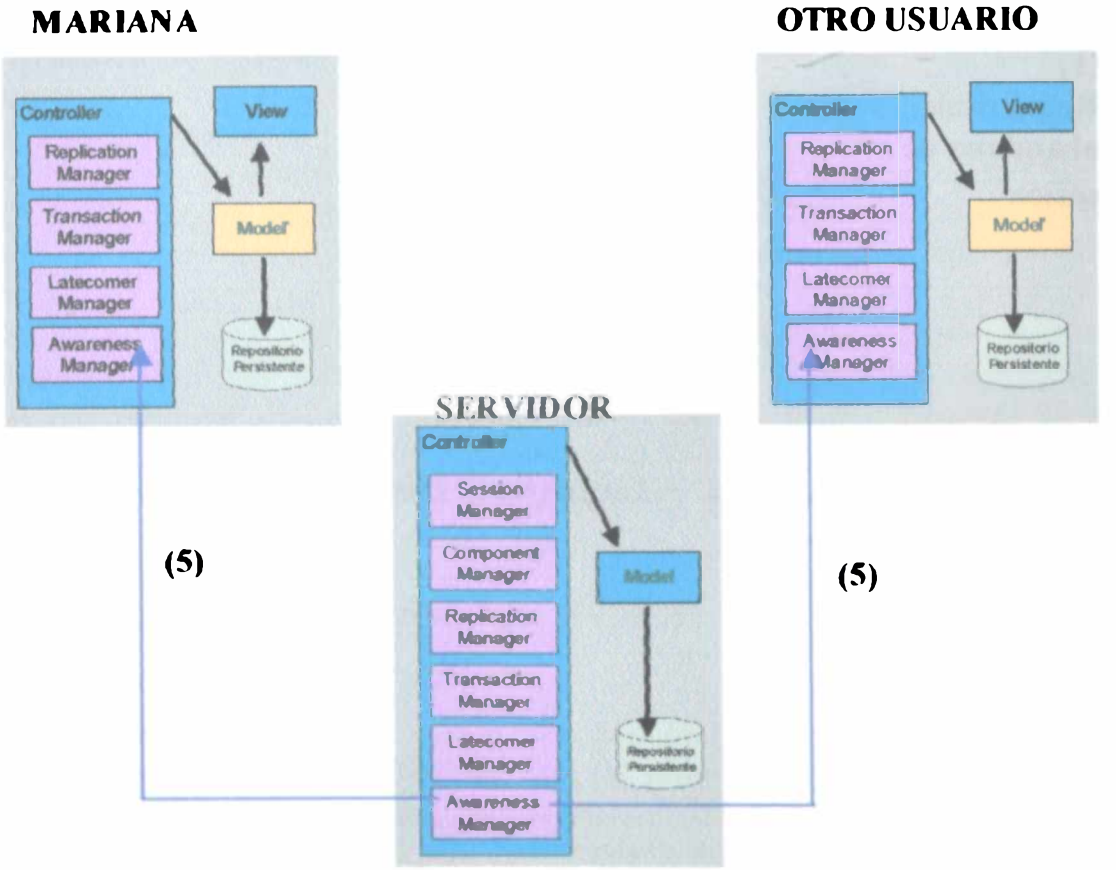


Figura 5.49 – Escenario 6: Paso 5

Luego de que *Mariana* abandona la aplicación el resto de los usuarios verifican tal situación en el listado de usuarios que se encuentran con sesiones iniciadas. (Figura 5.50)

Figura 5.50 – View actualizada de todos los usuarios

Capítulo 6

Comentarios Finales

En este capítulo se resumirán los resultados obtenidos en este trabajo de grado, de manera de presentar al lector una visión global de todos los temas tratados durante el desarrollo de esta tesis.
En la sección 6.1 se presentarán algunos trabajos relacionados con los principales temas tratados en el presente trabajo.
En la sección 6.2 se describirán los resultados obtenidos y conclusiones sobre ellos.
En la sección 6.3 se detallarán algunos lineamientos generales para una futura implementación de la arquitectura propuesta.
Y por último, en la sección 6.4, se enumerarán algunas líneas de trabajo futuro y posibles continuaciones a esta tesis.

6.1 Trabajos Relacionados

Como dijimos a lo largo del desarrollo de este trabajo de grado, actualmente se han llevado a cabo muchos trabajos de investigación, que se han centrado en solucionar problemas puntuales, relacionados al desarrollado de aplicaciones colaborativas sincrónicas, dado que las mismas abarcan muchas áreas de investigación.
En particular, analizamos dos frameworks especialmente diseñados para el desarrollo de tales aplicaciones, y algunas herramientas específicas desarrolladas con cada uno de ellos. El objetivo fue fundamentalmente, poner especial atención en los métodos de control de concurrencia utilizados para garantizar consistencia de información, y en los métodos de latecoming en caso de la llegada tarde de un usuario a la sesión de trabajo compartida.

Vimos que el framework COAST [SKSH96] implementa una arquitectura totalmente replicada, y utiliza transformación de operaciones como método de control de concurrencia para mantener consistentes los datos. Ofrece la posibilidad de que un usuario llegue tarde a una sesión, informando solo el estado final del trabajo compartido.
Por su parte DyCE [TR02] ofrece la posibilidad de desarrollar aplicaciones centralizadas o totalmente replicadas, utilizando transformación de operaciones como método de control de concurrencia, y en algunos casos ofrece la posibilidad de bloquear los datos compartidos, ya sea de manera automática, es decir, bloqueos donde la aplicación es la responsable de efectuarlos o delegando dicha responsabilidad al usuario que usa la aplicación. DyCE también informa del estado final de un trabajo compartido a cualquier usuario que se incorpore tarde a una sesión colaborativa.
El resultado de una serie de pruebas realizadas sobre herramientas especialmente diseñadas y desarrolladas mediante estos dos frameworks, nos hizo llegar a la

conclusión de existen algunos problemas de inconsistencias temporales, aún no resueltos hoy en la actualidad.
Sin embargo el objetivo de DyCE fue fundamentalmente lograr una clara separación entre los componentes y el modelo de objetos propiamente dicho. Sin dudas fue un gran avance en el desarrollo de aplicaciones colaborativas (Groupware).

Además, durante nuestra investigación se pudo comprobar que existe mucho trabajo relacionado a solucionar problemas específicos relacionados a la construcción de aplicaciones colaborativas sincrónicas.
Tal es el caso del framework DreamTeam [RU99]. Este framework ofrece una plataforma para el desarrollo, testeo y ejecución de aplicaciones sincrónicas compartidas en ambientes distribuidos heterogéneos, poniendo especial atención en la capa de red, en el desarrollo de componentes móviles y en la capa de comunicaciones. Este último concepto fue desarrollado en la sección 4.4 donde se describe el funcionamiento y la arquitectura del Servidor Sametime, cuya principal contribución radica en ofrecer una óptima arquitectura de comunicación, independientemente de la aplicación colaborativa a desplegar.

Sin embargo uno de los principales problemas que afecta especialmente a las aplicaciones colaborativas sincrónicas es cómo garantizar consistencia de información, dado que las tareas que llevan a cabo los usuarios se realizan en el mismo momento y en diferentes lugares.
Una buena referencia es [WARP94], donde se detallan los principales problemas de control de concurrencia.

Por otra parte, también se han encontrado buenas referencias sobre el modelado de colaboración mediante objetos compartidos [SSS99], donde se exponen los aspectos principales y los beneficios de la reusabilidad de los objetos compartidos.

En lo que respecta a la propiedad de latecoming no se ha encontrado mucha información relacionada al tema, dado que este concepto es un tema muy reciente. En [ITW01] se expone una posible solución al problema de latecoming en ambientes de aprendizaje colaborativos basados en la web implementado mediante JAVA. Otra buena referencia es [IM01] donde se presenta el desarrollo de un framework para la construcción de aplicaciones colaborativas sincrónicas, donde su principal contribución radica en ofrecer dos algoritmos: uno para retornar el estado final de una sesión de trabajo compartida, y el otro que retorna la sucesión de eventos que se generaron a lo largo de toda una sesión colaborativa, hasta llegar al estado actual del trabajo compartido.

En resumen, existen muchos trabajos de investigación relacionados a la construcción de aplicaciones colaborativas sincrónicas. Cada uno de ellos se centra en un problema específico, ya sea para dar solución al problema de consistencia de información en ambientes distribuidos o para ofrecer soporte a la propiedad de latecoming.
Sin embargo no se han encontrado hoy en día aplicaciones que ofrezcan simultáneamente buena consistencia de información y permitan reproducir la sucesión de eventos de una sesión colaborativa en caso de la llegada tardía de un usuario a la sesión de trabajo.

Por tal motivo, nuestro mayor aporte radica en ofrecer una solución al problema de consistencia y latecoming a través de un camino sencillo y flexible, poniendo especial atención, no en los componentes en sí, sino en los datos que ellos manipulan. Es decir, no en cómo desarrollar tales aplicaciones, dado que existe mucho trabajo en la actualidad que ofrece soporte a eso y con buenas soluciones, sino en como diseñar el modelo de datos que utilizará cada componente a desarrollar para garantizar un buen manejo de consistencia, como así también ofrecer soporte a la propiedad de latecoming, dando la posibilidad al usuario de decidir ver el estado final de una sesión colaborativa o, la sucesión de eventos hasta llegar al estado final.

6.2 Resultados obtenidos

El principal objetivo de este trabajo de grado fue definir una arquitectura para el desarrollo de aplicaciones colaborativas sincrónicas, dando soporte principalmente a los problemas de inconsistencias temporales en la información y a la propiedad de latecoming (llegada tarde de un usuario a la sesión colaborativa). Sin embargo durante el transcurso de nuestra investigación pudimos notar que existe mucho trabajo relacionado a dar soporte a la construcción de dichas aplicaciones. Cada solución aporta un buen camino a seguir y ofrece la posibilidad al desarrollador de construir sus propios componentes colaborativos.

Si bien todos los trabajos investigados durante el desarrollo del capítulo 4 dieron su aporte en los resultados finales de nuestra tesis, la mayor contribución fue a partir del estudio del framework DyCE donde pudimos notar que existe una buena separación entre los componentes, como ser pizarra, editor gráfico, editor de documentos, chat, etc., y el modelo de objetos utilizados por cada componente; por ejemplo, en el caso de un componente editor de diagramas, el modelo de objetos será un diagrama propiamente dicho.
De este modo decidimos utilizar este gran avance en la construcción de aplicaciones colaborativas, separando los componentes del modelo de objetos que éstos utilizan, centrando nuestra atención en cómo manipular y definir el modelo de objetos para garantizar consistencia de información y dar soporte a la propiedad de latecoming; dado que ambos problemas radican fundamentalmente en el modelo de objetos y no en los componentes en sí.

De esta manera en el capítulo 5 se define la arquitectura HA4CAC (Hybrid Architecture for Collaborative Applications Content), como su nombre lo indica, es una arquitectura híbrida definida para el contenido de aplicaciones colaborativas sincrónicas. Se define un modelo de arquitectura híbrida, dado que durante el desarrollo del capítulo 3, donde se mencionan las principales arquitecturas para el desarrollo de tales aplicaciones, pudimos comprobar que solo tres de ellas son las más utilizadas, estamos hablando de las arquitecturas centralizadas, replicadas e híbridas. Sin embargo, ninguna aplicación en la actualidad implementa una arquitectura híbrida dada su complejidad, a pesar de ser una buena arquitectura que toma las ventajas tanto de las arquitecturas centralizadas como de las replicadas.
En el capítulo 5, además se exponen los principales componentes de la arquitectura propuesta, detallando cómo es su funcionamiento tanto del lado del

cliente como del lado del servidor., y cómo es la comunicación entre ambos, mediante los controladores correspondientes.
En la sección 5.3 se muestran varios escenarios que intentan demostrar como será el funcionamiento de la arquitectura en diferentes casos de uso. De esta manera, se muestra mediante ejemplos concretos, que la arquitectura funciona adecuadamente, garantizando consistencia de información y permitiendo que el usuario decida ver el estado final de una sesión colaborativa o la sucesión de eventos hasta llegar al estado final, haciendo una descripción detallada de cómo es la estructura de los objetos.

6.2.1 Contribuciones de este trabajo de grado

Hoy en día, la mayor parte de las actividades que desarrollan las personas, habitualmente se lleva a cabo en un contexto mas bien grupal que individual, es por ello que surge la necesidad de proveer un mecanismo que soporte no solo la interacción entre el usuario y el sistema, sino que también garantice la interacción Usuario-Usuario, para que varias personas puedan realizar una tarea en común, sin la necesidad de estar físicamente reunidos.

Los sistemas que hacen más sencillo el proceso de compartir información, entre un grupo de personas que realizan una tarea común, son llamados Aplicaciones Groupware o Aplicaciones colaborativas.
Sin embargo, a pesar de la existencia de una gran variedad de aplicaciones groupware, varios estudios de la CSCW encontraron que tales aplicaciones no soportan adecuadamente la variabilidad del trabajo cooperativo en las organizaciones.
Desarrollar esta clase de aplicaciones no es una tarea sencilla, dado que involucran diversas áreas de trabajo, tales como sistemas distribuidos, comunicaciones, interfaces humano-computador, bases de datos, y algunas otras áreas más.

Específicamente en aplicaciones groupware, que desempeñan funciones en un espacio de trabajo compartido, donde se permite la interacción Usuario-Usuario en un mismo momento, lo que conocemos como Aplicaciones colaborativas sincrónicas, es imprescindible mantener alguna clase de consistencia entre dos o más representaciones del mismo espacio de trabajo compartido, a pesar de las actividades concurrentes de los usuarios.
La fundamental causa de inconsistencia, es que las acciones se llevan a cabo en el mismo instante de tiempo. Cuando un usuario lleva a cabo alguna acción sobre un conjunto de datos, otro usuario puede estar modificando los mismos datos en el mismo instante de tiempo, pero debido al tráfico de información en la red, tales acciones se procesan en diferente orden, y es allí donde se producen las inconsistencias.
Otro de los grandes inconvenientes en sistemas con espacio de trabajo compartido es cómo permitir la incorporación tardía (latecoming) de un nuevo usuario en el sistema, ya que esta acción requiere que al mismo se le informe la secuencia ordenada de las tareas efectuadas hasta llegar al estado actual del sistema. Esto nos enfrenta al problema de cómo informar al nuevo usuario sin afectar el trabajo actualmente realizado por el resto de los usuarios en el sistema.

El presente trabajo de grado ofrece de esta manera una arquitectura óptima para el modelo de objetos, utilizado por cualquier componente colaborativa que se desarrolle, resolviendo los problemas antes mencionados.

El modelo propuesto ofrece las siguientes ventajas:

- Hay una clara separación entre los componentes (editor gráfico, editor de diagramas, chat, etc...) y el modelo de objetos que ellos manipulan (documento gráfico, diagrama, documento de texto respectivamente). De esta manera, los datos son separados de los componentes que se utilizan para acceder a esos datos, de una manera similar en la que se separa las aplicaciones de los modelos de dominios.

- El estado compartido es modelado como objetos compartidos, los cuales son replicados dinámicamente siempre que se trate de objetos simples centralizados, de lo contrario serán manipulados centralizadamente, evitando así cualquier tipo de inconsistencia en la información.

- Ofrece soporte a la propiedad de latecoming, dando la libertad al usuario de elegir el método que desea para obtener el estado final de una sesión colaborativa. Podrá optar entre ver solo el estado final o la sucesión de eventos en el orden en que se fueron generando. Durante el proceso de latecoming no se bloquean componentes, no se bloquea al resto de usuarios conectados, como así tampoco se pierden eventos que son ejecutados por el resto de los usuarios durante el proceso de latecoming. De esta manera no se afecta el trabajo anteriormente realizado por el resto de los usuarios previamente conectados en la sesión colaborativa.

- Se utilizan bloqueos como métodos de control de concurrencia, lo que garantiza que los objetos compartidos centralizados sean manipulados por un usuario a la vez, evitando de esta manera inconsistencias temporales. En cuanto a los objetos simples compartidos que son replicados en el cliente cuando un usuario solicita su uso, son bloqueados de manera tal que el resto de los usuarios que intenten acceder a él en el mismo momento que esta siendo modificado por otro usuario recibirán las notificaciones correspondientes de que no podrá utilizarlo hasta tanto el usuario que lo tiene en uso libere el recurso compartido.

- Se define el concepto de Transacción (sección 5.2.4), la cual encapsula un conjunto de eventos. Tiene como ventaja que pueden implementarse operaciones undo y redo de manera sencilla en caso de que se ejecute un rollback en el repositorio persistente.

- Hay replicación en dos niveles:

  - ✓ Objetos que residen localmente en el cliente, lo que introduce el concepto de *model'* (sección 5.2.3), objetos que son de uso exclusivo del cliente, con lo cual toda manipulación de los mismos se hace de manera local, lo que ofrece mayor disponibilidad en información que solo será utilizada por un único cliente. Sin embargo el resto de los usuarios en la aplicación podrá ver siempre

actualizado el modelo ya que cualquier cambio en los objetos del *model'* de un cliente será replicado inmediatamente al modelo del servidor enviando las notificaciones correspondientes a todos los clientes activos. La replicación es desde el cliente al servidor y desde el servidor al resto de los usuarios conectados.

- ✓ Objetos replicados dinámicamente, objetos compartidos centralizados. Si el usuario desea utilizar un objeto simple se replicará en el cliente para que el mismo tenga mayor disponibilidad. En este caso la replicación es desde el servidor al cliente que solicita el objeto, una vez que finaliza su utilización el usuario notifica al servidor que el objeto fue liberado, luego el servidor replicará los cambios efectuados localmente en el usuario al resto de los usuarios conectados.

- Define un modelo de objetos (objetos HA4CAC-ver sección 5.2.3.3) dando la libertad al desarrollador de optar cómo será la implementación de los mismos, respetando ciertas restricciones que se deben cumplir para un buen funcionamiento de la arquitectura propuesta.

- Otorga una lista de permisos asignados a los usuarios para poder manipular los objetos HA4CAC definidos en la arquitectura. El modelo ofrece extensibilidad en este aspecto, ya que estos permisos serán definidos de acuerdo a los criterios de cada diseñador. Existen ciertas restricciones al respecto en el sentido de que todo desarrollador deberá respetar los conceptos de "objetos de uso exclusivo" y "objetos compartidos". Dentro de estas dos categorías podrán definir todos los permisos que deseen.

- Ofrece al desarrollador la posibilidad de construir componentes mediante modelos o frameworks ya existentes, utilizando la arquitectura propuesta solo para el manejo de los datos. Esto tiene como ventaja que independientemente del componente que se quiera desarrollador el modelo de datos puede cambiar sin demasiado esfuerzo extra por parte del diseñador del sistema. Como así también cualquier cambio que sufra el componente será totalmente independiente al modelo de datos que se esté utilizando.

Desventajas del modelo propuesto:

- El modelo define una arquitectura híbrida para el contenido. Como ya hemos dicho, una arquitectura híbrida toma las ventajas de las arquitecturas centralizadas y replicadas. Sin embargo, nuestro modelo no es una arquitectura híbrida propiamente dicha, sino que se han definido un par de modificaciones al modelo estándar. De todas maneras, existe un modelo central que siempre esta actualizado con las últimas modificaciones. Como ya sabemos, toda arquitectura centralizada tiene sus desventajas como ser un único punto de fracaso y el famoso cuello de botella en caso de que se produzcan numerosas solicitudes en el mismo instante de tiempo. Sin embargo, existen muchas soluciones a esos problemas. Se puede utilizar una buena arquitectura de comunicación utilizando clusters en la capa del servidor, esto hace que haya una mayor

disponibilidad del sistema en caso de que una componente falle, dando soporte también al tema del cuello de botella, ya que las solicitudes pueden ser atendidas por cualquier cluster que se encuentre disponible. Lo que intentamos decir, es que existen muchas alternativas al problema de las arquitecturas centralizadas, solo será cuestión de hacer un buen análisis del problema y elegir la arquitectura de comunicación mas adecuada para el mismo.

- En cuanto al proceso de latecoming, explicamos que la arquitectura propuesta ofrece la posibilidad al usuario de elegir ver el estado final o la sucesión de eventos hasta llegar al estado actual de una sesión colaborativa. Básicamente, esto se debe a que el método de mostrar la sucesión de eventos, si bien es un mecanismo para ofrecer al usuario un mayor nivel de detalle de lo que ocurrió antes de su llegada a la sesión de trabajo, tiene varios inconvenientes. Entre ellos los más importantes son: a) requiere de gran capacidad de almacenamiento, ya que una sesión de trabajo crecerá indeterminadamente. Con lo cual se debe contar con una política que controle este crecimiento y disminuirlo cuando sea apropiado y b) si un usuario decide ver la sucesión de eventos, puede ocurrir que se le reproduzcan todos los eventos para ver finalmente solo un documento vacío. Para entender mejor este caso, supongamos que durante una sesión de trabajo se crean y modifican muchos objetos, y luego un participante borra todos objetos en el documento compartido. Al latecomer que ingresa luego de esta última acción, se le reproducirá *toda* la sucesión de eventos para finalmente vea solo un documento vacío. Por tal motivo, se pensó que el modelo propuesto proporcione los dos métodos para la propiedad de latecoming.

6.3 Lineamientos generales para una futura implementación de la arquitectura propuesta

6.3.1 Java como lenguaje de desarrollo

Java es un lenguaje relativamente reciente dentro del mundo del desarrollo del software, pero que se ha hecho muy conocido debido a sus capacidades de portabilidad entre plataformas y fácil interfaz para el desarrollo de aplicaciones cliente / servidor y servicios sobre la plataforma de Internet.
Las características principales que nos llevan a recomendar JAVA como lenguaje de desarrollo fueron las siguientes:

Portable – Se pretende lograr que el marco de trabajo sea lo más portable posible, el hecho de que Java proporcione una plataforma de desarrollo independiente del tipo de hardware y sistema subyacente lo hizo especialmente atractivo para el desarrollo.
Además el hecho de poder utilizar métodos nativos del sistema para el manejo de dispositivos físicos, con relativa facilidad, lo hacen suficientemente flexible para

poder realizar cualquier tipo de aplicación o servicio que sea capaz de aprovechar de forma eficiente las capacidades concretas de la máquina en la que se ejecuta, sin perder por esto su portabilidad (o al menos parte de ella), ya que el único código a modificar correspondería al de los dispositivos específicos.
Una buena planificación aislando las clases que utilicen el código nativo permitirá un rápido desarrollo y adaptación de cualquier producto creado sobre cualquier plataforma.

Seguro –La seguridad en Java se implementa mediante un gestor de seguridad y un monitor de seguridad (en el caso de las Applets). Mediante su utilización, se consigue mejorar la privacidad de los datos en las aplicaciones que impliquen la conexión o interacción de usuarios remotos y se pueden controlar los intentos de acceso no legítimos de las aplicaciones a recursos de las maquinas como discos duros o zonas de memoria no locales a la aplicación.

Orientado a Objetos – Este paradigma de desarrollo ha mostrado sus inmejorables cualidades de reutilización y versatilidad a la hora de resolver problemas frente a la programación modular clásica. El uso de la programación orientada a objetos permite una buena definición de los elementos del sistema, el uso de herencia, polimorfismo y por encima de todo la reutilización eficiente de las clases ya desarrolladas; sin bien no es un lenguaje de programación orientado a objetos puro se aproxima bastante bien a los conceptos fundamentales de tal paradigma salvando algunas diferencias. Aún así es especialmente útil al construir nuevas aplicaciones o servicios, ya que el uso o adaptación adecuada de los conectores y componentes básicos ofrecidos por el sistema, permite centrarnos en el desarrollo del código específico, ya que el esquema básico y las relaciones entre los objetos del sistema ya están definidas, por lo que solo hemos de realizar las modificaciones necesarias en aquellos puntos que no se adapten exactamente al comportamiento del servicio específico a desarrollar.
Todo esto mejora considerablemente el tiempo de desarrollo y la fiabilidad de las aplicaciones desarrolladas mediante este paradigma de programación.
Aunque Java es un lenguaje joven y con algunos fallos, se trata de un lenguaje bastante flexible y que evoluciona positivamente con cada nueva versión que se crea. La versión actual ha mejorado en gran medida el desarrollo de aplicaciones visuales con Java, el cual resultaba bastante más complicado en versiones anteriores del lenguaje.
Además se ha mejorado especialmente la velocidad del código interpretado por la JVM, lo cual era uno de los principales handicaps del lenguaje.

Para concluir podemos decir que Java es un lenguaje adecuado para el desarrollo de aplicaciones y servicios que se ejecuten especialmente sobre la WWW, pero en sus orígenes el lenguaje estaba muy orientado al paradigma cliente / servidor, quizás influenciado por el tipo de lenguajes a los que sustituiría, como cgi, pero con el tiempo, las necesidades de los consumidores han cambiado y se tiende a distribuir más el trabajo para no sobrecargar a los servidores, por ello se implementaron una serie de cambios, como la aparición de los servlets que permitían una mejor distribución del trabajo.
Por tanto, la impresión general es la de un lenguaje que se está construyendo de a poco según la demanda del mercado, por lo que tarde o temprano se impondrá una revisión completa del mismo desde sus mismas bases.

Hoy por hoy, resulta imposible hablar de Internet, o intranets o incluso dispositivos de red sin hablar de Java. Aparentemente, Java se ha establecido como la plataforma estándar para la construcción de aplicaciones sobre la Web.
El lenguaje de programación Java ha sido adoptado más rápidamente que cualquier otro lenguaje de programación en la historia, y la Plataforma Java ha sido reconocida por la mayoría de las corporaciones como la clave tecnológica para soportar el acceso a un gran número de recursos corporativos de forma independiente a las plataformas.
Además, no hay duda que es increíblemente popular entre los desarrolladores de software. Ha revolucionado el desarrollo y expansión de las aplicaciones de Internet e intranets, ya que ofrece un nuevo modelo de computación sobre la red gracias a los objetos remotos. Este nuevo modelo es fácil de entender y también fácil de programar.
Pero, además, hay otra razón por la que Java ha tenido mucho éxito: ofrece la promesa de ser capaz de que un programa escrito una vez sea ejecutado en cualquier sitio, sin importar el hardware o el sistema operativo sobre el que lo haga. Sólo este hecho implica un enorme potencial de ahorrar tiempo y costo en el desarrollo de software.

6.3.2 Problemas Generales de las Aplicaciones Colaborativas

En esta sección mostraremos algunos de los problemas generales de los sistemas de trabajo colaborativo, como se han explicado durante el desarrollo de este trabajo de grado, y cómo el lenguaje Java es capaz de solucionarlos de forma efectiva.

Homogeneidad de los entornos de ejecución

Uno de los problemas principales por los que las aplicaciones colaborativas no han logrado expandirse en el mercado es por el hecho de tener que ejecutarse sobre sistemas operativos o plataformas con iguales características o al menos compatibles. Java mediante las características del propio lenguaje, obvia esos problemas, permitiendo la ejecución de sus programas en multitud de plataformas sin tener que cambiar una sola línea de código en los mismos.

Disponibilidad local de una copia de la aplicación

Otro de los problemas más comunes, principalmente desde el punto de vista de la instalación y mantenimiento de las aplicaciones colaborativas, se encuentra en mantener la compatibilidad de diferentes versiones del programa, para lo que normalmente se debía recurrir a la instalación de forma local de una copia del mismo en la máquina de cada usuario.
Con Java, mediante el uso de Applets, podemos lograr que todos los usuarios utilicen la misma versión del programa y este se tenga que instalar en una sola máquina que sea accesible para el usuario mediante un navegador. De esta forma ejecutar la aplicación es tan sencillo como abrir tan solo un navegador web.

Mantenimiento de la integridad de datos entre clientes de una sesión

Este sea tal vez el aspecto más importante de las aplicaciones colaborativas, ya que de esto depende el correcto funcionamiento o no de la aplicación. Debemos advertir que según la naturaleza de la aplicación este problema puede presentarse como mas o menos importante.
En general, la complejidad del mantenimiento de la integridad de la información del sistema (consistencia), dependerá en primer lugar de si la aplicación es asincrónica o sincrónica. En nuestro caso es totalmente sincrónica por lo cual dependerá totalmente del tipo de información que se maneje. En sistemas en tiempo real donde la interfaz gráfica es compartida y se reflejan sobre ella los cambios el mantenimiento de la consistencia es vital para el correcto funcionamiento de la aplicación como vimos durante el desarrollo de este trabajo de grado.
Java no resuelve directamente este tipo de problèmas, estando la solución en las manos del programador, que deberá adaptarla al sistema según las características del mismo.

Colisión de Eventos y su Tratamiento

Como se planteaba anteriormente uno de los problemas básicos a la hora del uso de aplicaciones colaborativas, se encuentra en el control de los intentos de acceso simultáneo por parte de distintos usuarios a la misma información.
Si el acceso se produce para consulta no habrá ningún tipo de problema, ya que ambos compartirán la misma información sin producirse ningún tipo de inconsistencia sobre la misma. El problema surgirá si alguno o ambos tratan de modificar la información compartida. En este caso la solución dependerá en gran medida del tipo de sistema colaborativo en el que nos encontremos.
Si el sistema sigue un modelo de control explícito del flujo de información, (Explicit floor control model), será el propio sistema quien determine las acciones a tomar, ya que normalmente este tipo de sistemas se basan en turnos, por lo que en un momento dado solo un participante podrá modificar o tener acceso a una determinada parte de la información compartida del sistema y para poder modificarla, se debe poseer turno, y para tenerlo, ha de haber sido concedido por el resto de los usuarios de la aplicación de forma explícita, tal es el caso particular de la herramienta SnapChat desarrollada con DyCe, cuando un usuario intenta escribir en un documento de texto. Es responsabilidad del usuario solicitar acceso al documento de texto y es responsabilidad de este liberar el recurso.
Por otra parte tenemos los servicios con modelo de control de flujo implícito (Explicit floor control model), en este caso, el sistema implementa algún tipo de mecanismo o política que resuelve los conflictos de acceso sobre la información compartida de forma transparente al usuario. Este tipo de sistemas es lo que se plantea en este trabajo de grado, son más difíciles de implementar, pero resultan más eficientes y rápidos que los anteriores.
Java no proporciona ningún mecanismo de control de este tipo de errores, por lo que es responsabilidad del programador establecerlo en función de la naturaleza y comportamiento de la aplicación concreta.

Sincronismo y Ordenación de Eventos

Como se comento durante el desarrollo de este trabajo de grado existe una división entre sistemas sincrónicos y asincrónicos. En el primer tipo, se plantea un problema relacionado con la emisión y la recepción de los eventos por parte de cada participante en la sesión.

Todos los protocolos de comunicación ofrecidos hoy en día para Internet, incluyen el envío de mensajes entre sus funciones básicas, pero el envío de mensajes en orden desde un destino, no aseguran su recepción en el orden correcto, es más no se asegura su recepción en absoluto. Por lo tanto se debe habilitar algún tipo de mecanismo que permita la ordenación y ejecución de los eventos recibidos en el orden correcto en el que fueron generados, así como el tratamiento de los posibles errores provocados por la pérdida de eventos durante su transmisión por la red.
La solución más interesante expuesta en este trabajo de grado es la de utilizar un sistema de transacciones similar al utilizado por las bases da datos para su actualización, para ello sería necesaria la implementación de un mecanismo de bloqueo simple para evitar el acceso de otros usuarios a los datos que estamos modificando. El uso de técnicas de bloqueo conlleva a su vez la prevención de deadlock (problemas de lazo mortal o inanición) que prevengan la posibilidad de bloqueos indefinidos en el sistema.

Por último y no menos importante sería proporcionar a las aplicaciones la capacidad de ejecución reversible, es decir ser capaces de deshacer las operaciones realizadas sobre los datos comunes en cualquier momento, posibilitando la corrección de errores de colisión e inconsistencia de los datos compartidos. Esto también se puede llevar a cabo mediante las transacciones que hacen más sencillo el proceso de implementar operaciones undo y redo.

6.3.3 Rendimiento

Uso de RMI para establecer el sistema de comunicación del Marco de trabajo

Pueden en un primer momento parecer perjudiciales para el rendimiento del sistema. Esto es cierto en cuanto al tiempo de ejecución del mismo, pero a cambio de la pérdida en el rendimiento obtenemos una mayor estabilidad del sistema y una mayor defensa ante posibles inconsistencias o problemas de programación, ya que estas clases y su implementación han sido probadas en muchos sistemas y aplicaciones, y por tanto resultan fiables y libres de fallos conocidos.
El uso de la interfaz socket esta recomendada en aquellos casos en los que deseamos obtener un rendimiento óptimo del sistema, pero estaremos expuestos a fallos más frecuentes de programación y al consumo de más tiempo de desarrollo al tener que realizar más pruebas para poder validar el programa.
Sin embargo existen hoy un día muchas alternativas para implementar un sistema de comunicación optimo. Incluso existen plataformas que ofrecen soporte a este tema. Será cuestión de hacer un buen análisis y utilizar aquella que mejor se adapte a las características del sistema a desarrollar.

Identificación de usuarios

La forma clásica de controlar el acceso por parte de un usuario a un sistema consiste en la identificación del mismo mediante el uso de una clave o algún otro tipo de verificador de identidad. Para lograr esto, se debe mantener una base de datos con la información de los usuarios, identificadores y permisos en algún lugar del sistema.

Existen servidores especializados en este tema como los actuales servidores LDAP (Lightweight Directory Access Protocol) de licencia gratuita. La última versión en el mercado es la versión 3. También hoy en día se encuentra la versión de OID (Oracle Internet Directory) que es parte del Identity Management perteneciente a la empresa Oracle, para la administración de usuarios en Internet. Empresa que ha tenido un gran crecimiento en los últimos años, sin embargo se debe contar con el licenciamiento correspondiente para su uso; no ocurre lo mismo en el caso de los servidores LDAP de licenciamiento gratuito.

Integración con sistemas de seguridad existentes

Para evitar la introducción de un número indeterminado de passwords para acceder a un servicio, se debe intentar integrar los mecanismos de identificación de la plataforma con aquellos propios del sistema local, aprovechándose de la seguridad ya probada de los mismos.

Control de Acceso

No solo se debe identificar al usuario, sino determinar a qué servicios puede acceder y a cuales no, dentro del sistema.

Auditoria de Seguridad

La normativa vigente, recomienda controlar todos los intentos de entrada al sistema, así como aquellas operaciones que pudieran considerarse peligrosas al exponer datos confidenciales a terceros, por lo que se recomienda crear una utilidad capaz de generar informes "logs" sobre los sucesos del sistema.
También deberíamos ser capaces de mantener información de lo que hace cada usuario dentro del sistema para comprobar quien realiza acciones dudosas o extrañas dentro del mismo.
Java posee dos mecanismos para controlar la seguridad del sistema: uno de ellos está en establecer una política de permisos, lo que nos permite determinar con toda facilidad que permisos tiene la aplicación sobre los recursos del sistema, como se explicó durante el desarrollo de este trabajo de grado.

Autenticación

Ya hemos hablado de seguridad con anterioridad, la forma habitual de resolver el problema es mediante el uso de password y ficheros de identificación de usuarios, pero las tecnologías actuales permiten la creación de registros firmados, que son mas seguros que los anteriores ya que previenen la suplantación de personalidad desde otros hosts, tanto para los usuarios como para los servidores, por medio de centros certificadores que generan permisos utilizando RSA (empresa de seguridad de software fundada inicialmente por Rivest, Shamir, & Adleman autores de un algoritmo muy difundido para garantizar autenticación).
La solución puede ser un tanto ineficiente, pero dependiendo del grado de seguridad que deseemos obtener para nuestro servicio nos plantearemos su uso o no.
Hoy en día este método, junto al cifrado de clave pública sonde los más seguros que existen.

Gestión de permisos

Este es un problema externo y de auditoria, se debe mantener la confidencialidad de las passwords y usuarios del sistema mediante un buen sistema de gestión y auditoria.

6.4 Trabajo futuro

El presente trabajo de grado ha dejado un modelo de arquitectura propuesto para el manejo de contenido de aplicaciones colaborativas sincrónicas. Actualmente existen diversos puntos por explorar, tales como buenas construcciones de componentes colaborativas, buen soporte de awareness para actualizaciones y notificación de las acciones de un usuario al resto de los usuarios conectados en una sesión colaborativa, y lograr la integración de la arquitectura propuesta con otras arquitecturas desarrolladas sobre diferentes niveles de información. Para esto se proponen las siguientes ramas como continuación del presente trabajo:

- Análisis de arquitecturas desarrolladas para la construcción de componentes colaborativos
- Análisis de arquitecturas específicas para el manejo óptimo de las comunicaciones
- Implementación de la arquitectura propuesta para el contenido de componentes, mediante algunas directivas que fueron detalladas en la sección 6.3
- Integrar a la arquitectura propuesta, componentes colaborativos desarrollados a través de arquitecturas especialmente diseñadas para dicho propósito, y manejo de comunicación óptima para un mejor rendimiento del sistema
- Construcción de una aplicación completa utilizando el modelo de arquitectura propuesto para detectar posibles errores que no permitan un buen funcionamiento del mismo

Apéndice A

Patrón Model View Controller (MVC)

A.1. ¿Que son los patrones?

Los patrones de software son soluciones reutilizables a los problemas que ocurren durante el desarrollo de un sistema de software o aplicación. Estos proporcionan un proceso consistente o diseño que uno o más desarrolladores pueden utilizar para alcanzar sus objetivos. También proporciona una arquitectura uniforme que permite una fácil expansión, mantenimiento y modificación de una aplicación.
Para ampliar información sobre este tema, puede acudirse al libro "Patrones de Diseño" de E. Gamma et al, Ed. Addison-Wesley.
Para ampliar información sobre los patrones que se suelen usar en aplicaciones J2EE :

http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/

A.1.2 Estructura de los patrones

Los patrones usualmente se describen con la siguiente información:

- Descripción del problema: Que permitirá el patrón y ejemplos de situaciones del mundo real.
- Consideraciones: Que aspectos fueron considerados para que tomara forma esta solución.
- Solución general: Una descripción básica de la solución en si misma.
- Consecuencias: Cuales son los pro y contra de utilizar esta solución.
- Patrones relacionados: Otros patrones con uso similar que deben ser considerados como alternativa.

A.1.3 Tipos de Patrones

Existen diferentes tipos de patrones. Dependiendo del nivel conceptual de desarrollo donde se apliquen, se distinguen (de más abstractos a más concretos): patrones de análisis, patrones arquitectónicos, patrones de diseño y patrones de implementación o *idioms*.

Al respecto puede encontrarse más información en:

http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

Dependiendo del propósito funcional del patrón, se distinguen los siguientes tipos:

- Fundamental: construye bloques de otros patrones.
- Presentación: Estandariza la visualización de datos.
- De creación: Creación condicional de objetos.
- Integración: Comunicación con aplicaciones y sistemas y recursos externos.
- De particionamiento: Organización y separación de la lógica compleja, conceptos y actores en múltiples clases.
- Estructural: Separa presentación, estructuras de datos, lógica de negocio y procesamiento de eventos en bloques funcionales.
- De comportamiento: Coordina/Organiza el estado de los objetos.
- De concurrencia: Maneja el acceso concurrente de recursos.

El patrón Modelo-Vista-Controlador (MVC) es un ejemplo de patrón **arquitectónico estructural**.

A.2 Patrón MVC

A.2.1. Introducción

Las aplicaciones Web pueden desarrollarse utilizando cualquier arquitectura posible. La arquitectura del patrón Modelo-Vista-Controlador es un paradigma de programación bien conocido para el desarrollo de aplicaciones con interfaz gráfica (GUI).

A.2.2 ¿En que consiste el MVC?

El principal objetivo de la arquitectura MVC es aislar tanto los datos de la aplicación como el estado (modelo) de la misma, del mecanismo utilizado para representar (vista) dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo (controlador).

Las aplicaciones MVC se dividen en tres grandes áreas funcionales:

- **Vista**: la presentación de los datos

- **Controlador**: es el objeto que proporciona significado a las ordenes del usuario, actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio, entra en acción, bien sea por cambios en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al propio Modelo.
- **Modelo**: la lógica del negocio o servicio y los datos asociados con la aplicación

El propósito del MVC es aislar los cambios. Es una arquitectura preparada para los cambios, que desacopla datos y lógica de negocio de la lógica de presentación, permitiendo la actualización y desarrollo independiente de cada uno de los citados componentes.

El MVC consta de:

- Una o más vistas de datos

- Un modelo, el cual representa los datos y su comportamiento

- Un controlador que controla la transición entre el procesamiento de los datos y su visualización.

A continuación mostramos un esquema de este modelo:

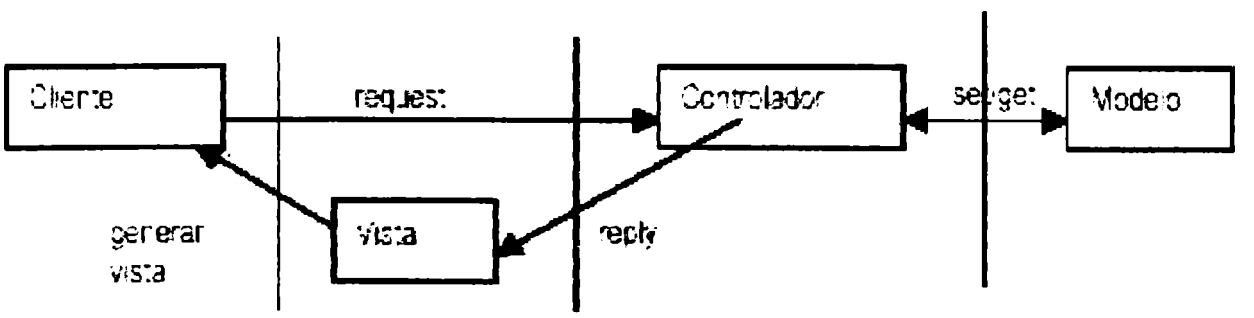


Figura A.1 – Esquema del Patrón Model-View-Controller

A.3 MVC y J2EE

Las aplicaciones de MVC pueden ser implementadas con J2EE utilizando JSP para las vistas, servlets como controladores y JDBC para el modelo.

Por ejemplo:

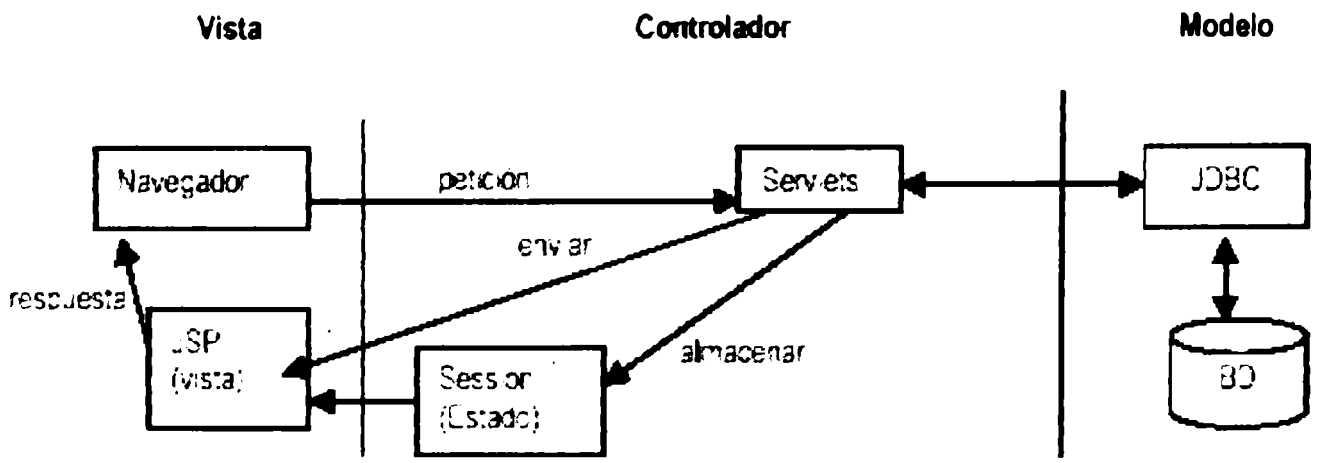


Figura A.2 – Ejemplo de MVC implementado con J2EE

Apéndice B

UML

Cualquier rama de ingeniería o arquitectura ha encontrado útil desde hace mucho tiempo la representación de los diseños de forma gráfica. Desde los inicios de la informática se han estado utilizando distintas formas de representar los diseños de una forma más bien personal o con algún modelo gráfico. La falta de estandarización en la manera de representar gráficamente un modelo impedía que los diseños gráficos realizados se pudieran compartir fácilmente entre distintos diseñadores.

Se necesitaba por tanto un lenguaje no sólo para comunicar las ideas a otros desarrolladores sino también para servir de apoyo en los procesos de análisis de un problema. Con este objetivo se creo el Lenguaje Unificado de Modelado (UML: *Unified Modeling Language*).

UML se ha convertido en ese estándar tan ansiado para representar y modelar la información con la que se trabaja en las fases de análisis y, especialmente, de diseño.
El lenguaje UML tiene una notación gráfica muy expresiva que permite representar en mayor o menor medida todas las fases de un proyecto informático: desde el análisis con los casos de uso, el diseño con los diagramas de clases, objetos, etc., hasta la implementación y configuración con los diagramas de despliegue.

B.1 Historia de UML

El lenguaje UML comenzó a gestarse en octubre de 1994 [1], cuando Rumbaugh se unió a la compañía *Rational* fundada por Booch (dos reputados investigadores en el área de metodología del software). El objetivo de ambos era unificar dos métodos que habían desarrollado: el método Booch y el OMT (*Object Modelling Tool*). El primer borrador apareció en octubre de 1995. En esa misma época otro reputado investigador, Jacobson, se unió a *Rational* y se incluyeron ideas suyas. Estas tres personas son conocidas como los "tres amigos". Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.
Esta primera versión se ofreció a un grupo de trabajo para convertirlo en 1997 en un estándar del

OMG (*Object Management Group* http://www.omg.org). Este grupo, que gestiona estándares relacionados con la tecnología orientada a objetos (metodologías, bases de datos objetuales, CORBA, etc.), propuso una serie de modificaciones y una nueva versión de UML (la 1.1), que fue adoptada por el OMG como estándar en noviembre de 1997.
Desde aquella versión han habido varias revisiones que gestiona la *OMG Revision Task Force.*

B.2 Modelo Visual

Tal como indica su nombre, UML es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del sistema. Para facilitar este modelado, se realiza una abstracción y se plasma en una notación gráfica. Esto se conoce como modelado visual.
El modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. De la misma forma que para construir una choza no hace falta un modelo, cuando se intenta construir un sistema complejo como un rascacielos, es necesario abstraer la complejidad en modelos que el ser humano pueda entender.
UML sirve para el modelado completo de sistemas complejos, tanto en el diseño de los sistemas software como para la arquitectura hardware donde se ejecuten.
Otro objetivo de este modelado visual es que sea independiente del lenguaje de implementación, de tal forma que los diseños realizados usando UML se puedan implementar en cualquier lenguaje que soporte las posibilidades de UML (principalmente lenguajes orientados a objetos).
UML es además un método formal de modelado. Esto aporta las siguientes ventajas:

- Mayor rigor en la especificación.
- Permite realizar una verificación y validación del modelo realizado.
- Se pueden automatizar determinados procesos y permite generar código a partir de los modelos y a la inversa (a partir del código fuente generar los modelos). Esto permite que el modelo y el código estén actualizados, con lo que siempre se puede mantener la visión en el diseño, de más alto nivel, de la estructura de un proyecto.

B.3 ¿Qué es UML?

UML es ante todo un lenguaje. Un lenguaje proporciona un vocabulario y reglas para permitir una comunicación. En este caso, este lenguaje se centra en la representación gráfica de un sistema.
Este lenguaje nos indica cómo crear y leer los modelos, pero no dice cómo crearlos. Esto último es el objetivo de las metodologías de desarrollo.

Los objetivos de UML son muchos, pero se pueden sintetizar sus funciones:

- Visualizar: UML permite expresar de una forma gráfica un sistema de forma que otro lo puede entender.
- Especificar: UML permite especificar cuáles son las características de un sistema antes de su construcción.

- Construir: A partir de los modelos especificados se pueden construir los sistemas diseñados.
- Documentar: Los propios elementos gráficos sirven como documentación del sistema desarrollado que pueden servir para su futura revisión.

Aunque UML está pensado para modelar sistemas complejos con gran cantidad de software, el lenguaje es los suficientemente expresivo como para modelar sistemas que no son informáticos, como flujos de trabajo (*workflow*) en una empresa, diseño de la estructura de una organización y por supuesto, en el diseño hardware.

Un modelo UML esta compuesto por tres clases de bloques de construcción:

- Elementos: Los elementos son abstracciones de cosas reales o ficticias (objetos, acciones, etc.)
- Relaciones: relacionan los elementos entre sí.
- Diagramas: Son colecciones de elementos con sus relaciones.

Las distintas vistas de un modelo están definidas en UML a través de los distintos tipos de diagramas que incluye.

Un diagrama es la representación gráfica de un conjunto de elementos con sus relaciones. En concreto, un diagrama ofrece una vista del sistema a modelar.
Para poder representar correctamente un sistema, UML ofrece una amplia variedad de diagramas para visualizar el sistema desde varias perspectivas.

UML incluye los siguientes diagramas:

- Diagrama de casos de uso.
- Diagrama de clases.
- Diagrama de objetos.
- Diagrama de secuencia.
- Diagrama de colaboración.
- Diagrama de estados.
- Diagrama de actividades.
- Diagrama de componentes.
- Diagrama de despliegue.

En este apéndice solo se detallaran los diagramas de Estructura Estática (diagrama de clases) y los de Interacción (diagrama de secuencia y de colaboración) dado que fueron utilizados en el desarrollo de este trabajo de grado, por lo que nos centramos solo en éstos.

B.4 Elementos Comunes a Todos los Diagramas

B.4.1 Notas

Una nota se representa como un rectángulo con una esquina doblada con texto en su interior. Puede aparecer en un diagrama tanto, sola como también unida a un elemento por medio de una línea discontinua.
Puede contener restricciones, comentarios, el cuerpo de un procedimiento o un valor rotulado (*tagged value)*.

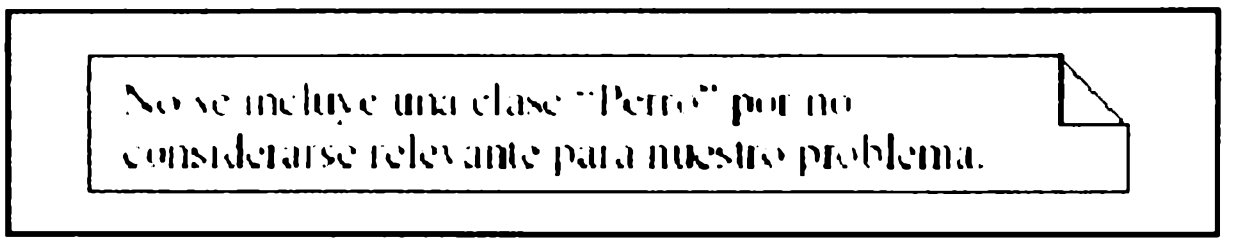


Figura B.1 – Ejemplo de nota.

B.4.2 Agrupación de Elementos Mediante Paquetes

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Cualquier grupo de elementos, sean estructurales o de comportamiento, puede incluirse en un paquete. Incluso pueden agruparse paquetes dentro de otro paquete.
Un paquete se representa como un rectángulo grande con un pequeño rectángulo sobre la esquina superior izquierda a modo de lengüeta. Si no se muestra el contenido del paquete entonces el nombre del paquete se coloca dentro del rectángulo grande. Si, por el contrario, se quiere mostrar el contenido del paquete, entonces el contenido va dentro del rectángulo grande y el nombre del paquete va en la lengüeta.
Se pueden indicar relaciones de dependencia entre paquetes mediante una flecha con la línea a trazos. Si el paquete A depende del paquete B, entonces irá una flecha de A a B, tal y como se muestra en la figura B.2.

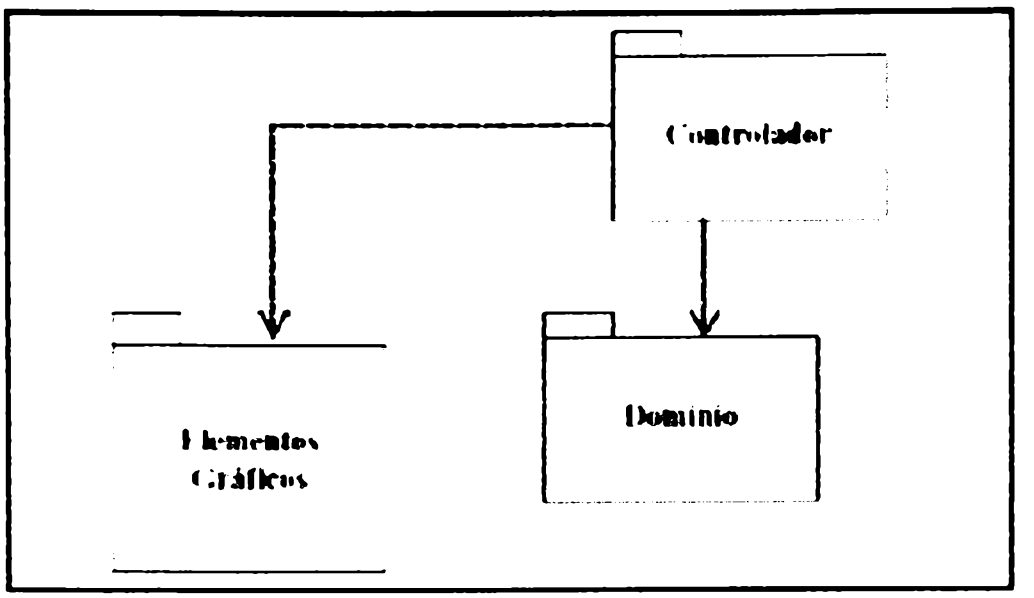


Figura B.2 – Agrupación por Paquetes.

B.5 Diagramas de Estructura Estática

Con el nombre de Diagramas de Estructura Estática se engloba tanto al Modelo Conceptual de la fase de Diseño de Alto Nivel como al Diagrama de Clases de Diseño. Ambos son distintos conceptualmente, mientras el primero modela elementos del dominio el segundo presenta los elementos de la solución software. Sin embargo, ambos comparten la misma notación para los elementos que los forman (clases y objetos) y las relaciones que existen entre los mismos (asociaciones) .

B.5.1 Clases

Una clase se representa mediante una caja subdividida en tres partes: En la superior se muestra el nombre de la clase, en la media los atributos y en la inferior las operaciones. Una clase puede representarse de forma esquemática (plegada), con los detalles como atributos y operaciones suprimidos, siendo entonces tan solo un rectángulo con el nombre de la clase. En la figura B.3 se ve cómo una misma clase puede representarse a distinto nivel de detalle según interese, y según la fase en la que se esté.

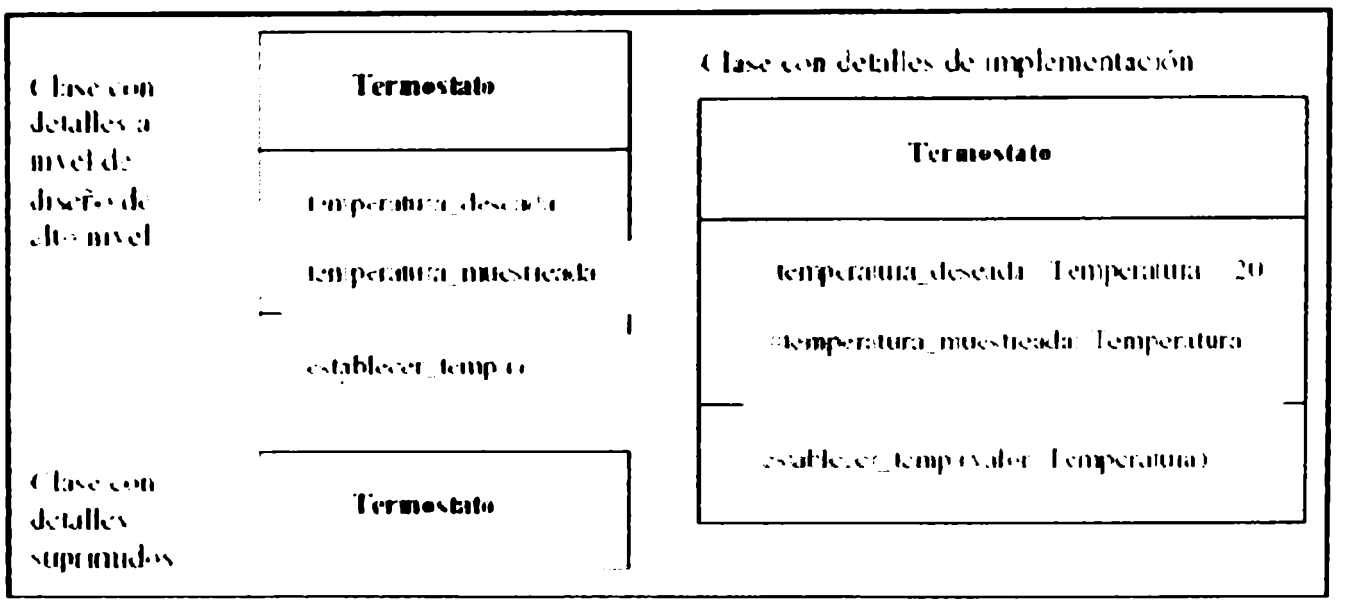


Figura B.3 - Notación para clases a distintos niveles de detalle.

B.5.2 Objetos

Un objeto se representa de la misma forma que una clase. En el compartimento superior aparecen el nombre del objeto junto con el nombre de la clase subrayados, según la siguiente sintaxis: *nombre_del_objeto: nombre_de_la_clase*
Puede representarse un objeto sin un nombre específico, entonces sólo aparece el nombre de la clase.

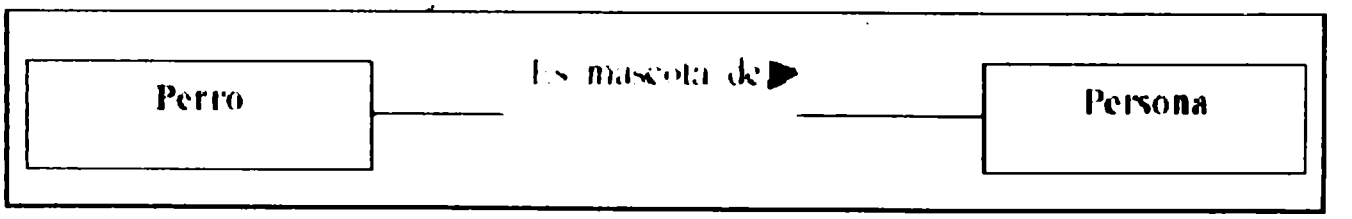


Figura B.4 - Ejemplos de objetos.

B.5.3 Asociaciones

Las asociaciones entre dos clases se representan mediante una línea que las une. La línea puede tener una serie de elementos gráficos que expresan características particulares de la asociación. A continuación se verán los más importantes de entre dichos elementos gráficos.

B.5.3.1 Nombre de la Asociación y Dirección

El nombre de la asociación es opcional y se muestra como un texto que está próximo a la línea. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual leer el nombre de la asociación. En el ejemplo de la

figura B.5 se puede leer la asociación como "Un objeto de la clase Perro es mascota de un objeto de la clase Persona".

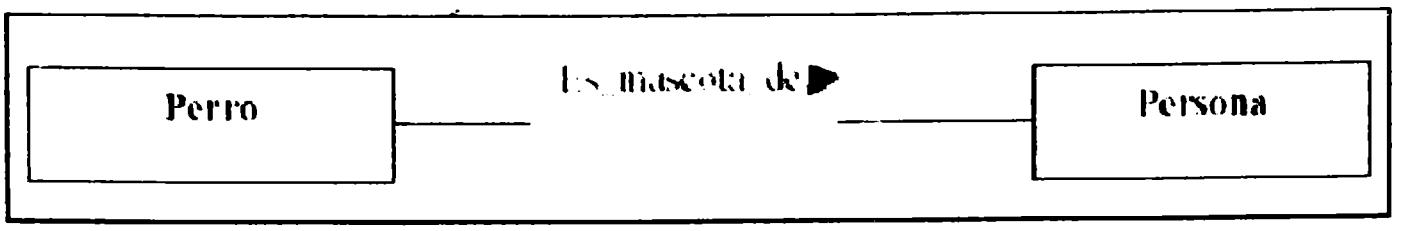


Figura B.5 - Ejemplo de asociación con nombre y dirección.

Los nombres de las asociaciones normalmente se incluyen en los modelos para aumentar la legibilidad. Sin embargo, en ocasiones pueden hacer demasiado abundante la información que se presenta, con el consiguiente riesgo de saturación. En ese caso se puede suprimir el nombre de las asociaciones consideradas como suficientemente conocidas.
En las asociaciones de tipo agregación y de herencia no se suele poner el nombre.

B.5.3.2 Multiplicidad

La multiplicidad es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener esa asociación con una instancia de la otra clase.
Puede expresarse de las siguientes formas:

- Con un número fijo: *1*.
- Con un intervalo de valores: *2..5*.
- Con un rango en el cual uno de los extremos es un asterisco. Significa que es un intervalo abierto. Por ejemplo, *2..** significa 2 o más.
- Con una combinación de elementos como los anteriores separados por comas: *1, 3..5, 7,15..**.
- Con un asterisco: * . En este caso indica que puede tomar cualquier valor (cero o más).

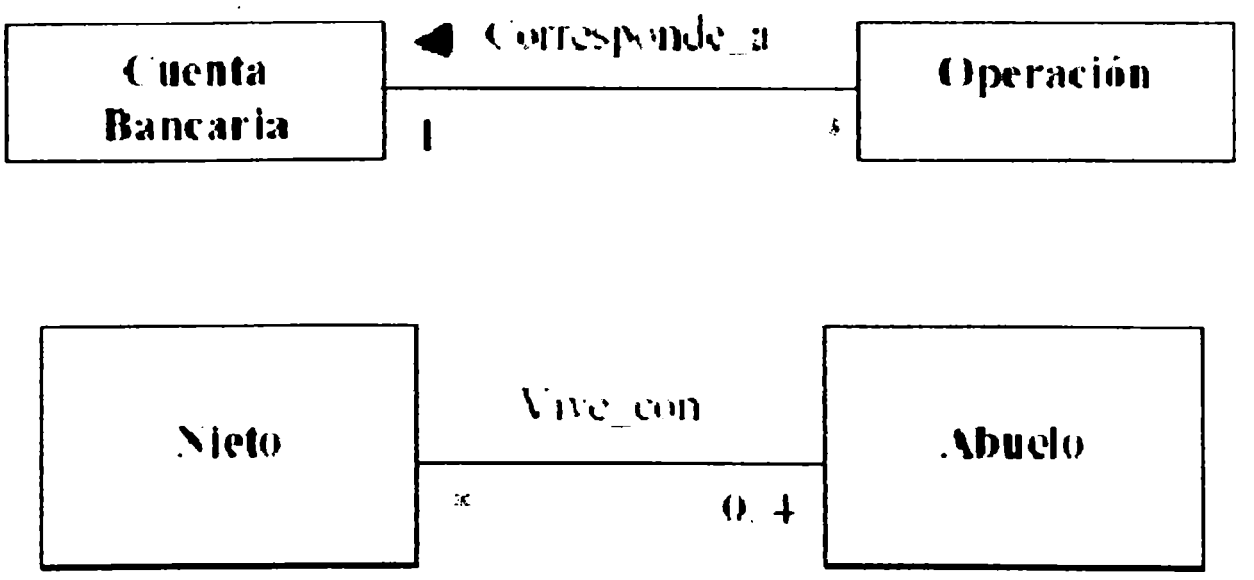


Figura B.6 - Ejemplos de multiplicidad en asociaciones.

B.5.3.3 Roles

Para indicar el papel que juega una clase en una asociación se puede especificar un nombre de rol.
Se representa en el extremo de la asociación junto a la clase que desempeña dicho rol.

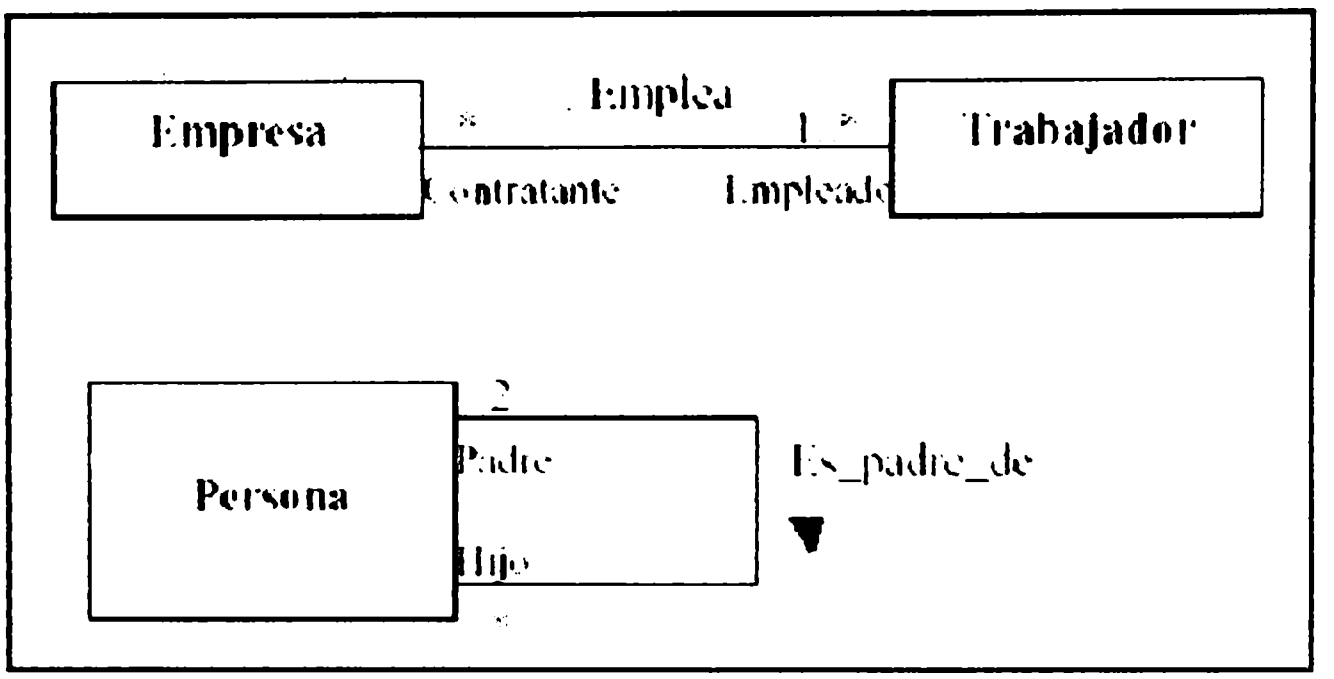


Figura B.7 - Ejemplo de roles en una asociación.

B.5.3.4 Agregación

El símbolo de agregación es un diamante colocado en el extremo en el que está la clase que representa el "todo".

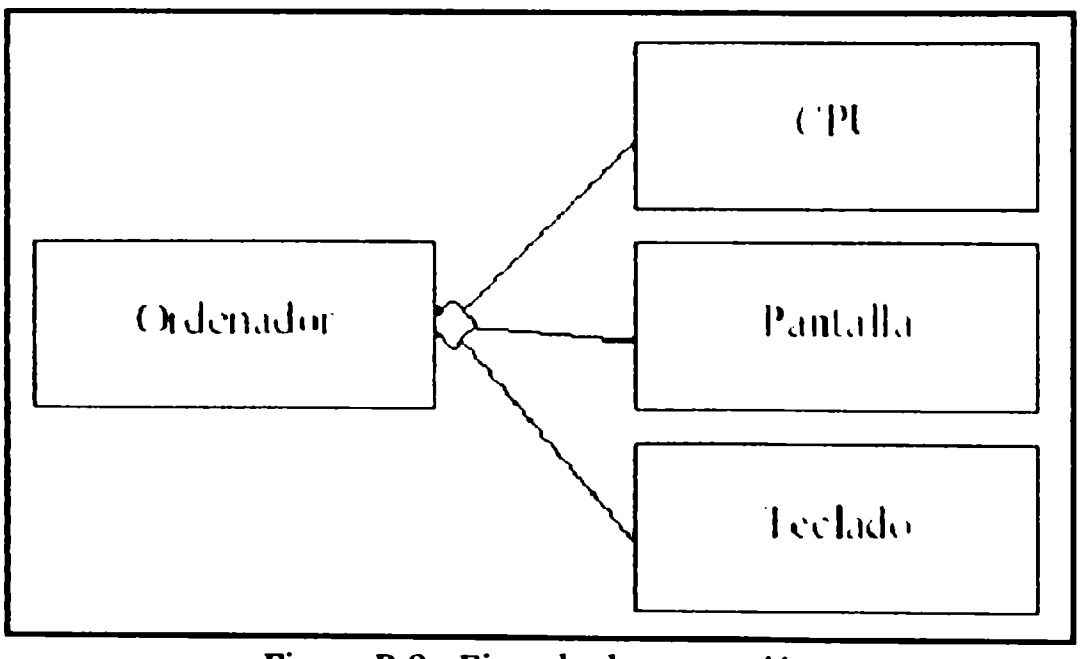


Figura B.8 - Ejemplo de agregación

B.5.3.5 Clases Asociación

Cuando una asociación tiene propiedades propias se representa como una clase unida a la línea de la asociación por medio de una línea a trazos. Tanto la línea como el rectángulo de clase representan el mismo elemento conceptual: la asociación. Por tanto ambos tienen el mismo nombre, el de la asociación. Cuando la clase asociación sólo tiene atributos el nombre suele ponerse sobre la línea (como ocurre en el ejemplo de la figura B.9). Por el contrario, cuando la clase asociación tiene alguna operación o asociación propia, entonces se pone el nombre en la clase asociación y se puede quitar de la línea.

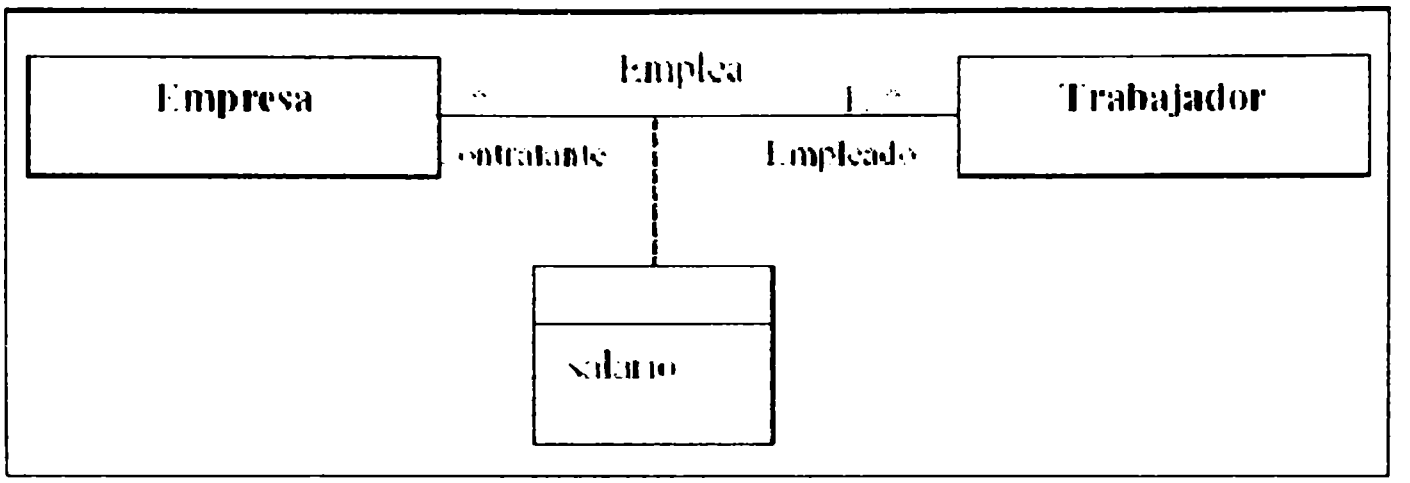


Figura B.9 - Ejemplo de clase asociación.

B.5.3.6 Asociaciones N-Arias

En el caso de una asociación en la que participan más de dos clases, las clases se unen con una línea a un diamante central. Si se muestra multiplicidad en un rol, representa el número potencial de tuplas de instancias en la asociación cuando el resto de los N-1 valores están fijos. En la figura B.10 se ha impuesto la restricción de que un jugador no puede jugar en dos equipos distintos a lo largo de una temporada, porque la multiplicidad de "Equipo" es 1 en la asociación ternaria.

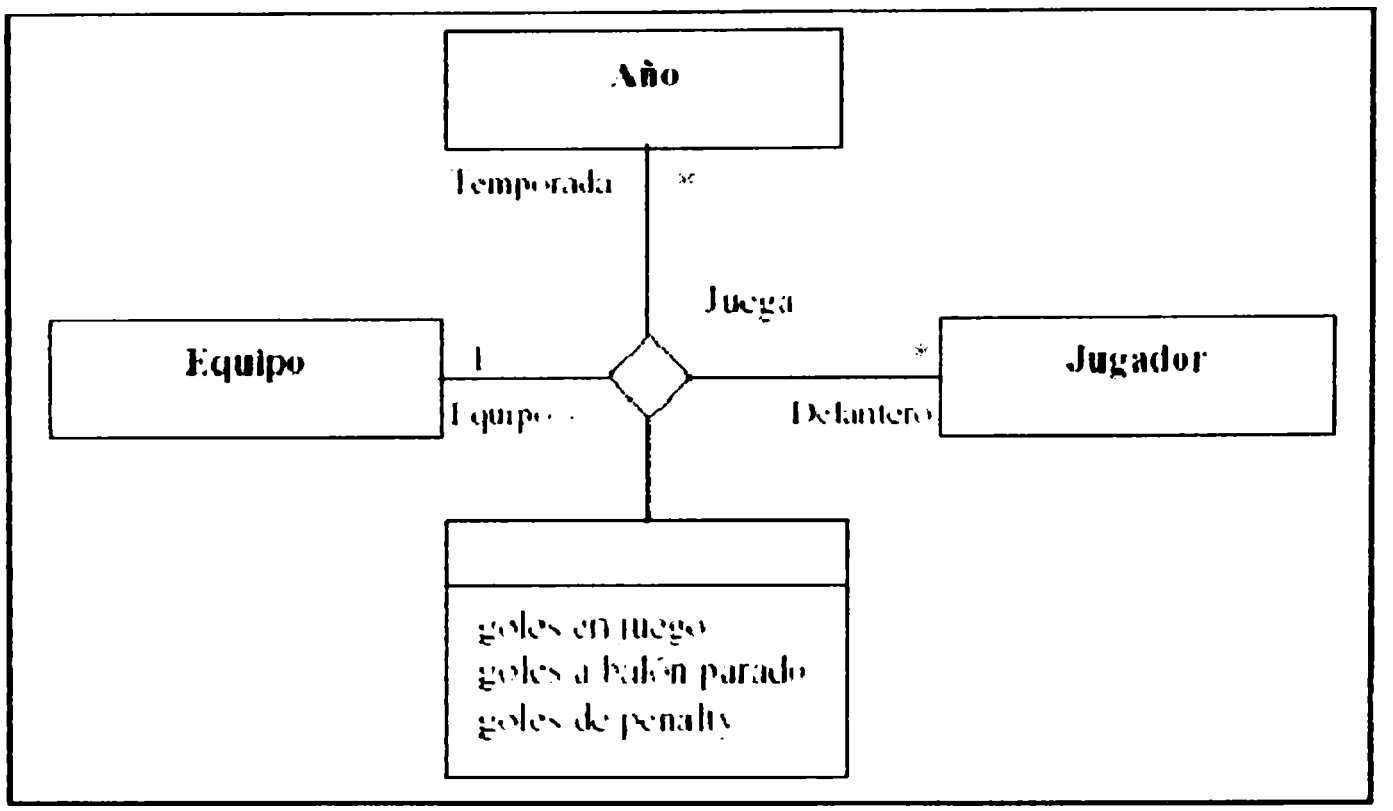


Figura B.10 - Ejemplo de asociación ternaria.

B.5.3.7 Navegabilidad

En un extremo de una asociación se puede indicar la navegabilidad mediante una flecha. Significa que es posible "navegar" desde el objeto de la clase origen hasta el objeto de la clase destino. Se trata de un concepto de diseño, que indica que un objeto de la clase origen conoce al objeto(s) de la clase destino, y por tanto puede llamar a alguna de sus operaciones.

B.5.4 Herencia

La relación de herencia se representa mediante un triángulo en el extremo de la relación que corresponde a la clase más general o clase "padre".

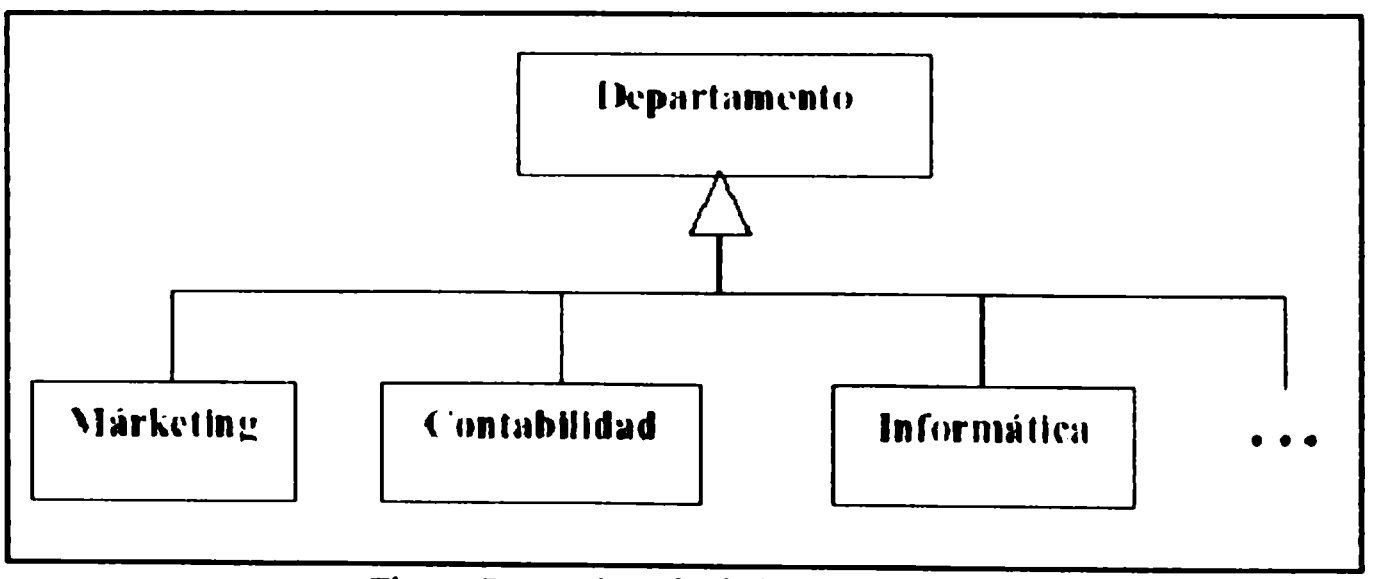


Figura B.11 - Ejemplo de herencia.

Si se tiene una relación de herencia con varias clases subordinadas, pero en un diagrama concreto no se quieren poner todas, esto se representa mediante puntos suspensivos. En el ejemplo de la figura B.11, sólo aparecen en el diagrama 3 tipos de departamentos, pero con los puntos suspensivos se indica que en el modelo completo (el formado por todos los diagramas) la clase "Departamento" tiene subclases adicionales, como podrían ser "Recursos Humanos" y "Producción".

B.5.5 Elementos Derivados

Un elemento derivado es aquel cuyo valor se puede calcular a partir de otros elementos presentes en el modelo, pero que se incluye en el modelo por motivos de claridad o como decisión de diseño. Se representa con una barra "/" precediendo al nombre del elemento derivado.

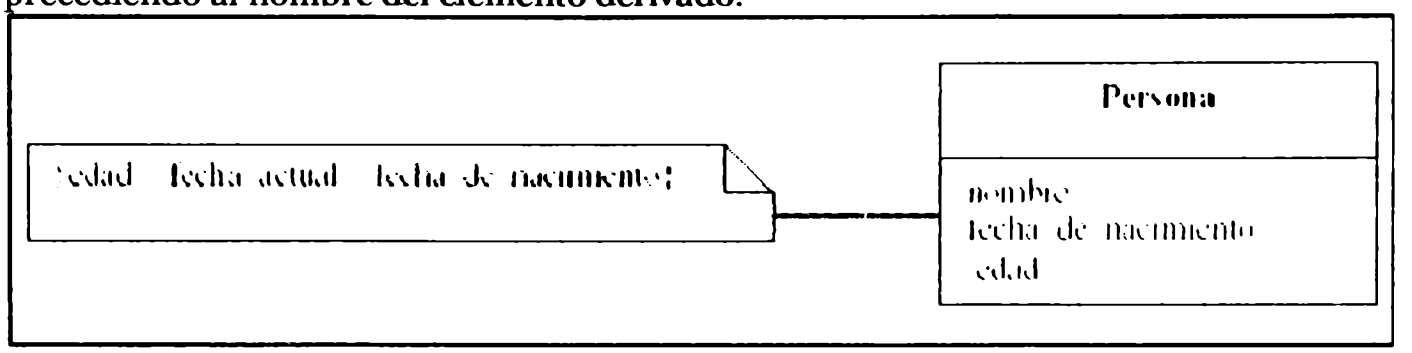


Figura B.12 - Ejemplo de atributo derivado.

B.6 Diagramas de Interacción

En los diagramas de interacción se muestra un patrón de interacción entre objetos. Hay dos tipos de diagrama de interacción, ambos basados en la misma información, pero cada uno enfatizando un aspecto particular: Diagramas de Secuencia y Diagramas de Colaboración.

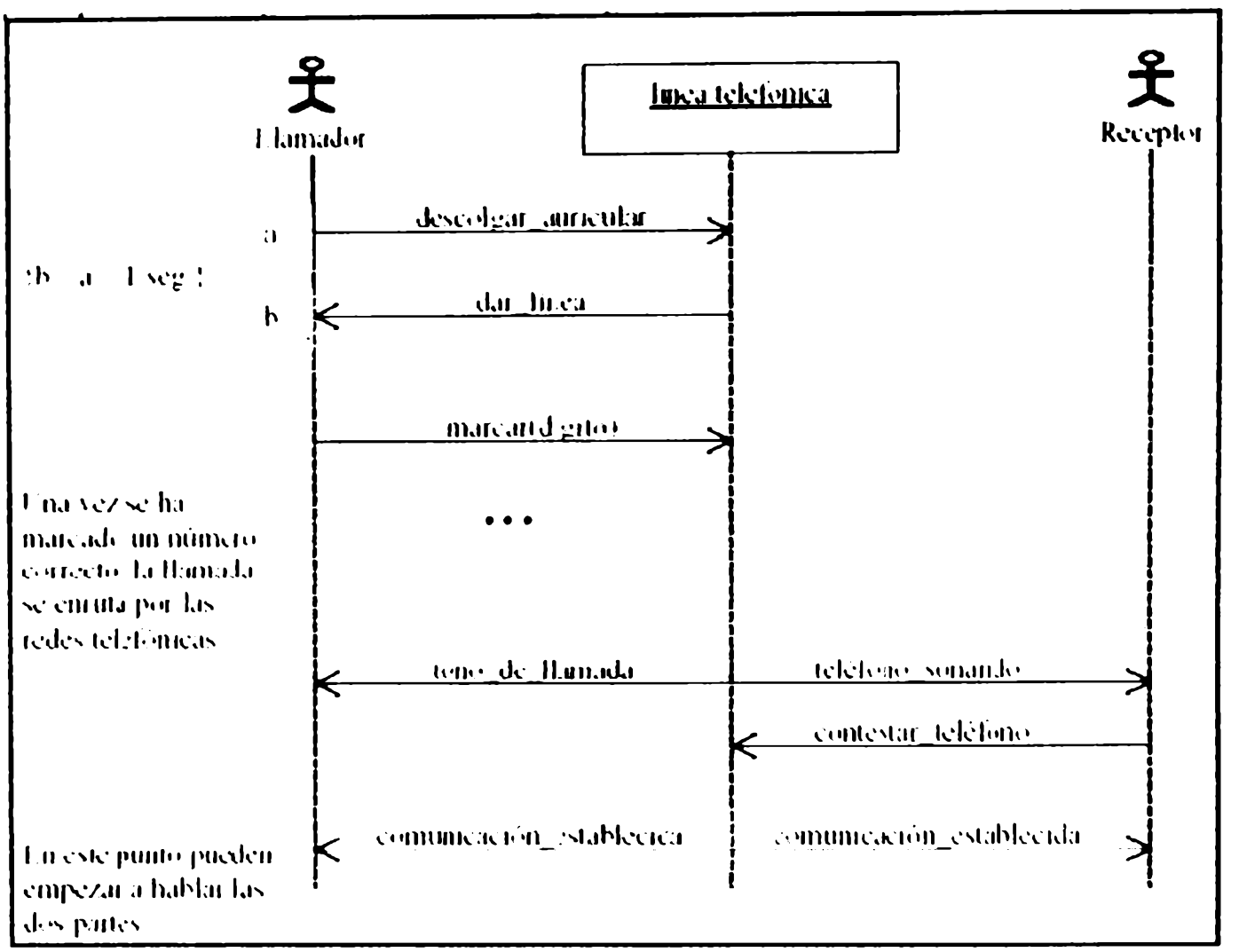


Figura B.13 - Diagrama de Secuencia.

B.6.1 Diagrama de Secuencia

Un diagrama de Secuencia muestra una interacción ordenada según la secuencia temporal de eventos. En particular, muestra los objetos participantes en la interacción y los mensajes que intercambian ordenados según su secuencia en el tiempo.
El eje vertical representa el tiempo, y en el eje horizontal se colocan los objetos y actores participantes en la interacción, sin un orden prefijado. Cada objeto o actor tiene una línea vertical, y Desarrollo Orientado a Objetos con UML
los mensajes se representan mediante flechas entre los distintos objetos. El tiempo fluye de arriba abajo.
Se pueden colocar etiquetas (como restricciones de tiempo, descripciones de acciones, etc.) bien en el margen izquierdo o bien junto a las transiciones o activaciones a las que se refieren. En la figura B.13 se representa el Diagrama de Secuencia para la realización de una llamada telefónica.

B.6.2 Diagrama de Colaboración

Un Diagrama de Colaboración muestra una interacción organizada basándose en los objetos que toman parte en la interacción y los enlaces entre los mismos (en cuanto a la interacción se refiere).
A diferencia de los Diagramas de Secuencia, los Diagramas de Colaboración muestran las relaciones entre los roles de los objetos. La secuencia de los mensajes y los flujos de ejecución concurrentes deben determinarse explícitamente mediante números de secuencia.

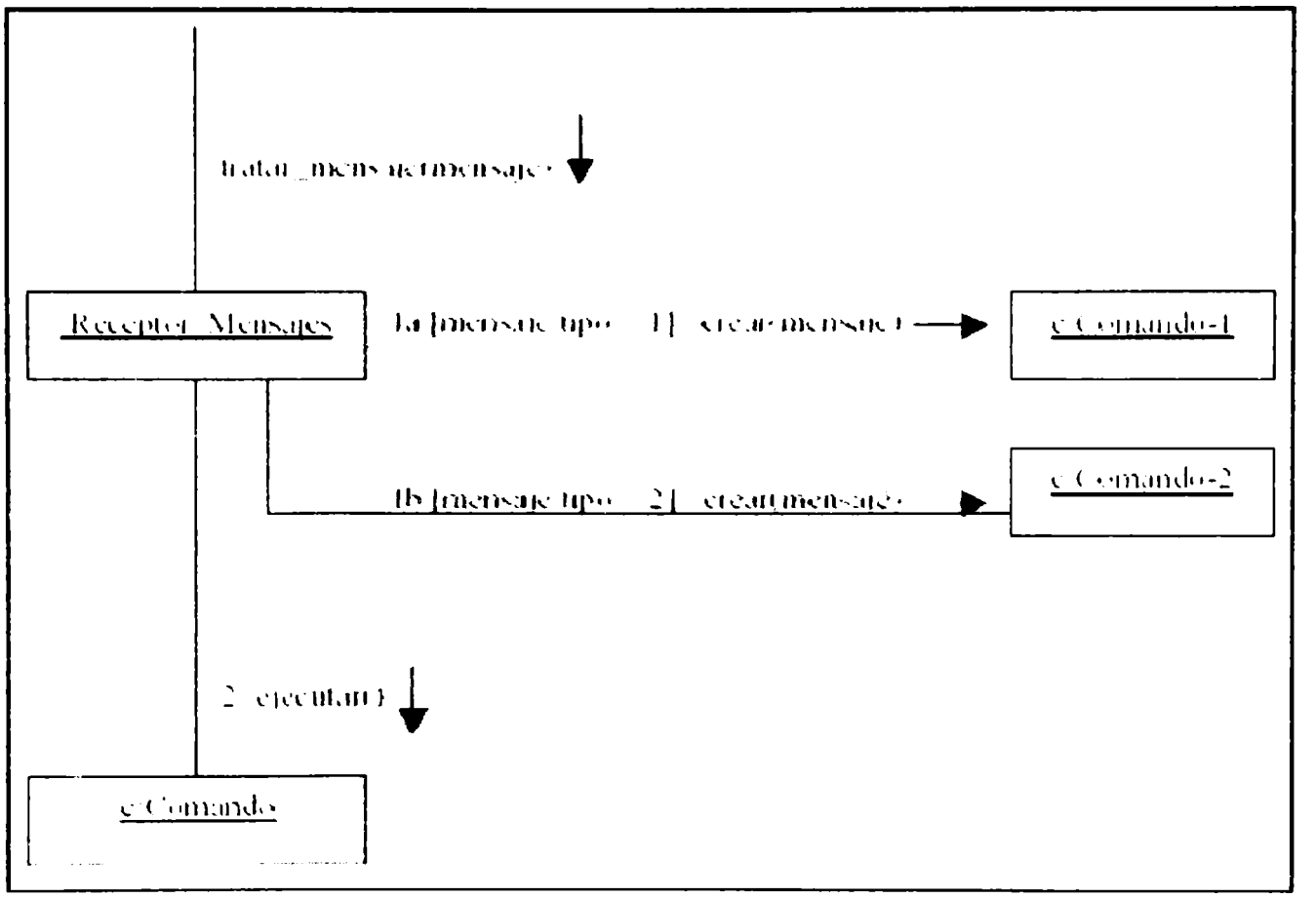


Figura B.14 - Diagrama de Colaboración.

En cuanto a la representación, un Diagrama de Colaboración muestra a una serie de objetos con los enlaces entre los mismos, y con los mensajes que se intercambian dichos objetos. Los mensajes son flechas que van junto al enlace por el que "circulan", y con el nombre del mensaje y los parámetros (si los tiene) entre paréntesis.

Cada mensaje lleva un número de secuencia que denota cuál es el mensaje que le precede, excepto el mensaje que inicia el diagrama, que no lleva número de secuencia. Se pueden indicar alternativas con condiciones entre corchetes (por ejemplo 3 [condición_de_test]: nombre_de_método()), tal y como aparece en el ejemplo de la figura B.14. También se puede mostrar el anidamiento de mensajes con números de secuencia como 2.1, que significa que el mensaje con número de secuencia 2 no acaba de ejecutarse hasta que no se han ejecutado todos los 2. x .

Apéndice C

Frameworks

Los frameworks orientados a objetos (llámense simplemente frameworks) son la piedra angular de la moderna ingeniería de software. El desarrollo de frameworks esta ganando rápidamente la aceptación debido a su capacidad para promover la reutilización del código fuente (source code).

Los frameworks son los Generadores de Aplicaciones que se relacionan directamente con un dominio específico, es decir, con una familia de problemas relacionados. Deben generar las aplicaciones para un dominio entero. Por lo tanto, debe haber puntos de flexibilidad que se puedan modificar de acuerdo a los requisitos particulares para ajustarse a la aplicación.

Los puntos flexibles de un framework se llaman **puntos hot-spots** (o puntos calientes). Los hot-spots son las clases o los métodos abstractos que deben ser implementados o puestos en ejecución.

Es importante destacar que *los frameworks no son ejecutables*, uno debe instanciar el framework y completar los hot-spots con las especificaciones propias de una aplicación en particular.

Sin embargo algunas características de los framework no son mutables ni tampoco pueden ser alteradas fácilmente.
Estos puntos inmutables constituyen el **núcleo** o **kernel** de un framework, también llamados puntos congelados o **frozen-spots**.

A diferencia de los hot-spots los puntos congelados o inmutables son los segmentos de código puestos en ejecución dentro del framwork que llaman a uno o a mas hot-spots. El núcleo o Kernel será la constante y presentara siempre la parte de cada instancia del framework.

Para entender mejor el concepto de un framework, podemos pensar en un framework como si fuese un motor. A diferencia de un motor tradicional, un motor o kernel de framework tiene muchas entradas de potencia. Cada una de esas entradas de potencia es un punto hot-spots del framework. Cada uno de estos puntos debe ser accionado para que el motor funcione.
Los generadores de potencia son el código específico de nuestra aplicación que se deben anexar a los hot-spots.
El código agregado de la aplicación será utilizado por el código Kernel del framework. El motor no correrá hasta que estén hechas todas las conexiones necesarias.

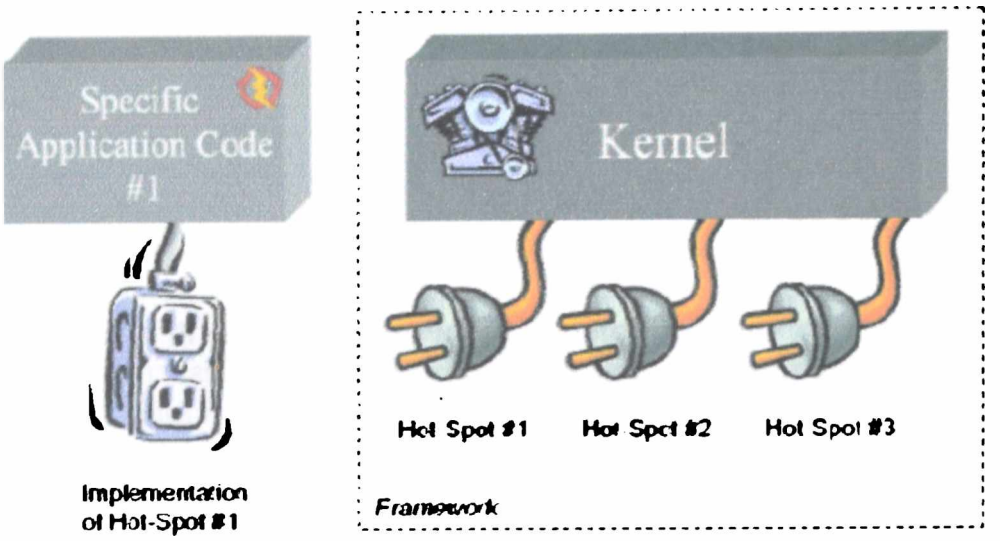


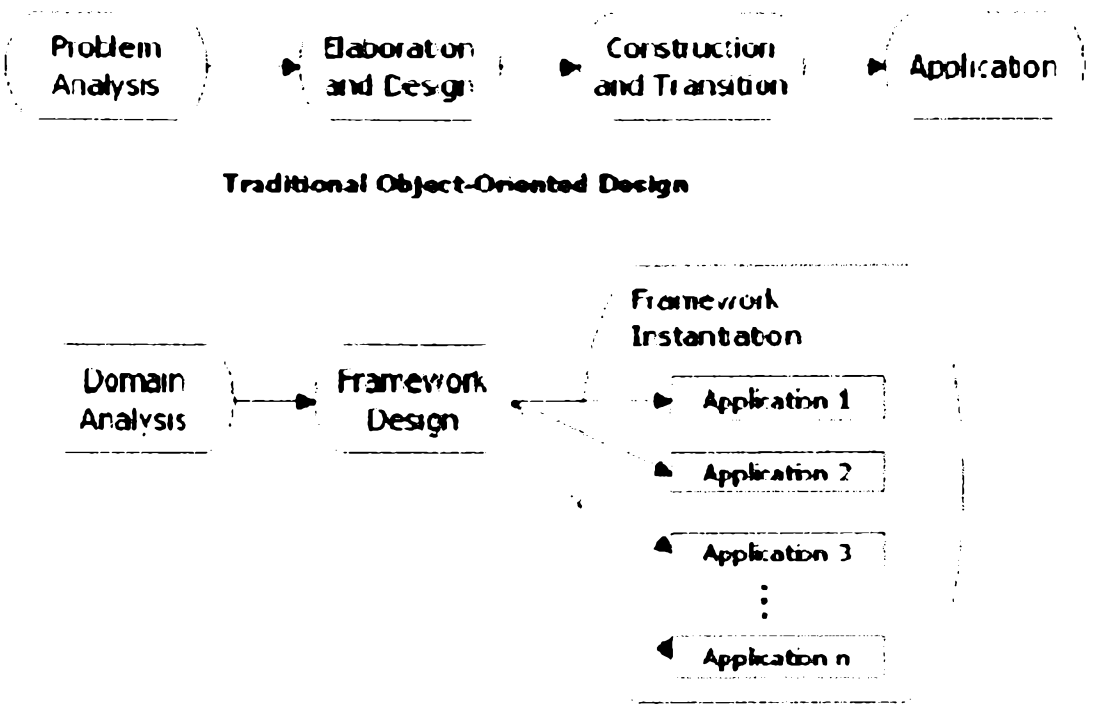
Figura C.1 – Concepto de Framework

La capacidad de reutilización del código y del diseño de los frameworks orientados a objetos permite una productividad mayor y un tiempo de Mercado breve en el desarrollo de aplicaciones, en comparación con el desarrollo tradicional de los sistemas de software.
La configuración flexible de frameworks, permite la reutilización del núcleo o Kernel del mismo.

Las tres etapas principales en el desarrollo de un framework son:

- Análisis del dominio
- Diseño del framework
- Instanciación

El análisis del dominio procura descubrir los requisitos del dominio y los posibles requerimientos futuros.
Durante el análisis del dominio, los puntos hot-spots y los puntos frizen-spots o congelados se destapan parcialmente.
La fase del diseño del framework define las abstracciones de éste. Se modelan los hot-spots y los frizen-spots (quizás con diagrama de UML, Modelo Unificado del Lenguaje, *Unified Modeling Language* [BJR98]) , y la extensión y la flexibilidad propuesta en el análisis del dominio se esboza en líneas generales.
Según lo mencionado antes, los modelos del diseño se utilizan en esta fase.
Finalmente, en la fase de "instanciación", los hot-spots del framework son implementados, generando un software del sistema especifico para una aplicación en particular.
Es importante observar que cada una de estas aplicaciones tendrá los firzen-spots del framework en común.
Las fases del proceso del desarrollo del framework son comparados con las tradicionales fases del diseño orientado a objeto, según se puede apreciar en el cuadro de la figura C.2, nombramos las fases del desarrollo según lo descrito en [JBR99].



FiguraC.2 – Proceso de desarrollo de un framework

En el desarrollo tradicional Orientado a Objeto, la fase de análisis del problema, también llamada inicio, estudia solamente los requisitos de un solo problema. En cambio, el desarrollo del framework captura los requisitos para un dominio entero.
Además, el resultado final del desarrollo orientado a objeto tradicional es una aplicación que es completamente ejecutable, mientras que muchas aplicaciones son las que resultan a partir de la fase de "instanciación" del desarrollo del framework. La fase de "instanciación" abarca las fases de construcción y de transición del desarrollo tradicional.
Así, la construcción y las fases separadas de la transición están presentes en cada uno de las instancias del framework. Para cada una de las instancias del framework hay un esfuerzo de implementación o puesta en práctica introducido por estas fases..

Aunque el desarrollo del framework promete ser muy eficiente, hay varias asuntos que deben ser discutidos.

Hay una serie de puntos que deben considerarse al elegir un modelo de framework, en donde cada uno de estos puntos deben ser analizados cuidadosamente; pues no son ni buenos ni malos, sino que son los puntos precisos a chequear.

Como habíamos indicado antes, los frameworks generan aplicaciones no por defecto sino que personalizando los requisitos particulares.
Ellos en sí mismos no son aplicaciones sino que son construcciones más complejas.
Es importante tener presente que el desarrollo de un framework será por lo menos tan costoso como el solo desarrollo de la aplicación.

Uno debe analizar cuidadosamente la necesidad de flexibilizar un framework al evaluar los requisitos que se deben resolver para un cliente o futuro usuario.

Es común integrar frameworks para satisfacer requerimientos de una aplicación. Sin embargo, Michael Mattson discute que haya por lo menos seis problemas comunes que los desarrolladores de frameworks y aplicaciones encuentran al integrar dos o más frameworks [MBF99].
Todos estos problemas derivan de un conjunto de cinco causas comunes: comportamiento cohesivo, cobertura del dominio, intención del diseño, carencia del acceso al código fuente, y carencia de los estándares para el framework.
Estos problemas se detallan detenidamente [MBF99], donde se proponen muchas soluciones para cada uno.
Los frameworks se abandonan o se abortan a menudo porque no pueden ser fácilmente integrados con otros frameworks. La integración del framework no es una tarea fácil, y *la composición debe ser considerada seriamente durante su desarrollo.*

Durante el desarrollo de un framework se debe considerar también que los requisitos del producto pueden cambiar rápidamente.
Una buena referencia de frameworks es [FSJ99], aquella que hace una buena evaluación de su metodología y sus últimos avances.

Los conceptos desarrollados anteriormente fueron citados a modo de establecer una clara definición de framework, explicar su proceso de creación y principalmente su estructura en general.

Bibliografía

[BAK95] R. Balter, S. B. Atallah, R. Kanawati. *Architecture for Synchronous Groupware Application Development.* HCI'95 – 1995 Tokoy Japan.

[BJR98] G. Booch, I. Jacobson, J. Rumbaugh, J. The *Unified Modeling Language User Guide. Addison* – Wesley Pub Co, 1998.

[BMRSS96] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. (1996) *Pattern-Oriented Software Architecture. A system of Patterns*, John Wiley & Sons.

[BS93] L.Bannon, K. Schmidt: *CSCW: Four Characters in Search of a context*, R. Baecker (Ed.),Readings in Groupware and ComputerSupported Cooperative Work, Morgan Kauffmann, San Mateo, California, 1993, pp. 50-56.

[Coleman97] COLEMAN, David (ed). *Groupware: Collaborative Strategies for Corporate LANs and Intranets.* Prentice Hall, Upper Saddle River, NJ, 1997.

[CSCW88] Suchman, L., (ed.), *CSCW 88 : Proceedings of the conference on computer-supported cooperative work*, September 26-29, 1988, Portland, Oregon. Association for Computing Machinery, New York, 1988.

[Dewan00] P. Dewan. *Architectures for Collaborative Applications.* University of North Carolina. 2000

[Egid88] Egido, C., *Videoconferencing as a technology : A review of its failures.* In [CSCW88], p. 13-24.

[EGR91] Clarence A. Ellis and Simon J. Gibbs and Gail Rein , 1991. *Groupware: some issues and experiences.* Communications of the ACM. Volume 34 , Issue 1. Pp. 39-58

[FKL+88] Fish, R.S., R.E. Kraut, M.D.P. Leland and M. Cohen, *Quilt : A collaborative tool for cooperative writing.* In R.B. Allen (ed.), Conference on office information systems, March 23-25, 1988, Palo Alto, CA, USA, vol. 9(2&3), SIGOIS bulletin. ACM Press, New York, 1988, p. 30-37.

[FSJ99] M. E. Fayad, D. C. Schmidt, R. E. Johnson. *Building Application Frameworks.* Addison-Wesley Pub Co, 1st edition, 1999.

[GK90] Galegher, Jolene. Kraut, Robert E. *Technology for intellectual teamwork:*
Perspectives on research and design. Intellectual teamwork: Social and Technological Foundations of Cooperative Work. GALEGHER, J. KRAUT, R.E., EGIDO, C (eds). Lawrence Erbaum Associates, Hillsdale, NJ, USA, 1990: pp 1-20.

[Grudin94] J. Grudin, Computer-Supported Cooperative Work: *History and Focus, Computer*, Mayo 1994, Vol. 27, No. 5., 19-26.

[GT00] S. Greenberg, M. Roseman. *Department of Computer Science.* Uiversity of Calgary. (Reveer)

[HBRPW94] H. Hill, T. Binck, S. Rohall, J. Patterson, W. Wilmer. *The Rendezvous architecture and language for constructing multiusers applications.* ACM Transactions on Computer Human Interaction. June 1994

[Heer96] Heeren, E., *Technology support for collaborative distance learning*. Ph.D. thesis, University of Twente, Enschede, the Netherlands, 1996, file://ftp.cs.utwente.nl/pub/doc/ctit/phd/heeren/heeren.zip.

[Hofte98] Henri ter Hofte - *Working Apart Together*. Foundations for Component Groupware. ISSN 1388-1795; No. 001.
ISBN 90-75176-14-7. Copyright © 1998, Telematica Institute, The Netherlands.

[HW92] Haake, J., Wilson, B. *Supporting collaborative writing of hyperdocuments in SEPIA*. Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW '92) Toronto, Canada (October 31 –November 4, 1992), 138 - 146.

[IM01] *Latecomer and Crash Recovery Support in Fault-Tolerant Groupware*. Mihail Ionescu and Ivan Marsic, Rutgers University IEEE Distributed Systems Online. http://computer.org/dsonline Vol. 2, No. 7, 2001. Collaborative Computing.

[ITW01] T. Illmann, R. Thol, M. Weber. *Transparent Latecomer Support for Web-Based Collaborative Learning Environments.* Department of Multimedia Computing. University of Ulm, Germany.2001

[JBR99] I. Jacobson, G. Booch, Rumbaugh, J. *The unified Software Development Process.* Addison-Wesley, 1999

[JCBZ99] J.Schümmer, C.Shucmman, L.M. Bibbó, and J.J. Zapico. *Collaborative Hypermedia Design Patterns in OOHDM.* Proceeding of the 4th International Workshop of Hypermedia Design Patterns. Darmstadt, Germany, 1999.

[Johansen88] Johansen, R. *Groupware: Computer Support for Business Teams.* New York: The Free Press, 1988.

[KBAW94] R. Kazman, L. Bass, G. Abowd, M. Webb. *SAAM: A Method for analyzing the properties of software architectures*. In Proceeding of International Conference on Software Engineering, ICSE'94

[KBL93] Karsenty, A., Beaudouin-Lafon, M. *An algorithm for distributed groupware applications*. Proceedings of the 13th International Conference on distributed Computing Systems ICDCS'93, (Pittsburgh,May 25-28).

[KP88] Krasner, G. E., Pope S. T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk*. Journal on Object Oriented Programming, August/September 1988.

[LJLR90] T. J. Lauwers, K. Lantz, A. Romanow. *Replicated Architectures for shared window systems*. A Critigue, ACM, pp 249, March 1990.

[LL02] D. Li, R. Li , 2002. *Transparent Sharing and Interoperation of heterogeneous Single-User Applications*. Department of Computer Science. Texas A&M University. College Station, Texas.2002

[MBF99] M. Mattsson, J. Bosch, M. E. Fayad. *Framework Integration Problems, Causes, Solutions*. Communication of the ACM October 1999. Vol. 42, Nº 10.

[OCL+93] Olson, J.S., S.K. Card, T.K. Landauer, G.M. Olson, T. Malone and J. Leggett, *Computer supported co-operative work : Research issues for the 90s*. Behaviour & Information Technology, 12 (1993), 2, p. 115-129.

[Patterson94] J. F. Patterson. *A Taxonomy of Architectures for synchronous Groupware Applications*. In Proceeding of the CSCW'94. Workshop on software architectures for cooperative systems. Chapel Hill, North Catalina.1994

[PS94] A. Prakash, H. S. Shim. *Distview: Support for building efficient collaborative applications using replicated active objects*. In Proceedings of the ACM conference on Computer Supported Cooperative Work, CSCW'94

[Rodden93] Rodden, T. 1993. *Technological support for cooperation. En: CSCW in practice: An introduction and case studies*, DIAPER, D. y SANGER, C., (eds) Springer-Verlag,
Londres, 1993: pp.1-22.

[Root88] Root, R.W., *Design of a multi-media vehicle for social browsing*. In [CSCW88], p. 25-38.

[RU00] J. Roth, C. Unger. *An extensible cassification model for distribution architectures of synchronous groupware*. In Dieng R. et al(ads): Fourth International Conference on the Design of Cooperative Systems. Sophia Antipolis. France 2000.

[RU99] J. Roth, C. Unger. *DramTeam- A platform for synchronous collaborative applications.* Prakatische Informatik II, FernUnivesität Gesamthochule Hagen.

[SG96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentiee Hall, New Jersey 1996.

[SSS00] J. Schümmer, T. Schümmer, C. Schuckmann, *COAST An Open Source Smalltalk Framework to Build Synchronous Collaborative Applications.* GMD - IPSI, Darmstadt, Germany intelligent views, 2000

[SSS99] J. Schümmer, T. Schümmer, C. Schuckmann, Peter Seitz, *Modeling Collaboration usin Shared Objects.*
GMD - German National Research Center for Information Technology - Integrated Publication and Information Systems Institute – IPSI. Dolivostr. 15, D – 64293 Darmstadt, Germany.

[TaIR94] Tang, J.C., E.A. Isaacs and M. Rua, *Supporting distributed groups with a montage of lightweight interactions.* In [CSCW94], p. 23-34,
http://www.sun.com/tech/projects/coco/papers/montage/CSCW94.ps

[TR02] Daniel A. Tietze, Jessica Rubart: *DyCE – A Framework for Component-Based Groupware.*
IPSI - Integrated Publication and Information Systems Institute
GMD - German National Research Center for Information Technology
Julius-Reiber-Strabe 15A - 64293 Darmstadt, Germany, 2002

[WARP94] WARP Report W1-94. *Concurrency Control for Real Time Groupware.* Division of Computer Science, University of St. Andrews.

Facultad de Informática

Universidad Nacional de La Plata