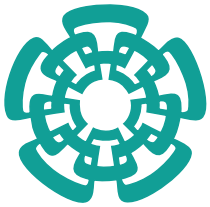


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Diseño e implantación de una agenda cooperativa flexible en un entorno no confiable

Tesis que presenta

Ing. Daniel Cruz García

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de Ingeniería Eléctrica

Opción

Computación

Director de Tesis

Dra. Sonia Guadalupe Mendoza Chapa

México, D.F. Febrero del 2009

Resumen

Los progresos recientes en las tecnologías computacionales y comunicacionales han revolucionado la forma de utilizar las computadoras. Esta revolución tecnológica refleja un cambio en el énfasis de emplear sistemas computacionales para facilitar la interacción humana, además de utilizarlos para resolver problemas. El presente trabajo de investigación se inscribe en el ámbito del Trabajo Cooperativo Asistido por Computadora (TCAC), el cual estudia los aspectos sociales de las actividades individuales y colectivas (disciplina etnometodológica), así como los aspectos tecnológicos de los soportes de información y comunicación (disciplina de los sistemas cooperativos). Particularmente, las soluciones propuestas en esta tesis de maestría repercuten en la disciplina de los sistemas cooperativos, cuya principal meta es permitir a grupos de personas, posiblemente distribuidas, cooperar efectivamente para lograr sus objetivos comunes, mediante el uso de sistemas cooperativos (generalmente distribuidos). Algunos ejemplos de éstos son: editores multiusuario, sistemas de mensajes, herramientas para la toma de decisiones en grupo, salas de reuniones electrónicas, conferencias por computadora, flujos de trabajo ("*workflows*") y ambientes de aprendizaje cooperativo. Uno de los dominios de aplicación del TCAC que no ha sido estudiado exhaustivamente, a pesar de ser uno de los primeros temas de interés de los investigadores, es el de las agendas cooperativas. Excepto por algunos productos comerciales (e.g., IBM Lotus Notes, Apple iCal y Microsoft Outlook) y de software libre (e.g., Darwin Calendar Server, Opengroupware.org y Open-Xchange) la mayoría de los estudios consideran a las agendas cooperativas como aplicaciones triviales o herramientas de decoración ofrecidas por otros sistemas, e.g., sistemas de conferencia por computadora. La motivación del presente trabajo reside en explorar dos problemas fundamentales de estas aplicaciones: 1) la limitada flexibilidad de sus bloques de construcción para adaptarlos a los diversos patrones de uso de los grupos y 2) la falta de mecanismos para garantizar la disponibilidad de la información, cuando estas aplicaciones se llevan del entorno fiable de una LAN (infraestructura de comunicación en una organización local) al entorno no fiable del Internet (infraestructura de comunicación en una organización distribuida o de múltiples organizaciones cooperantes). Para hacer frente al problema de flexibilidad limitada se recurrió al paradigma de la Programación Orientada a Aspectos (POA), cuyo principal objetivo es facilitar la reutilización de código entre aplicaciones. Para lograr este objetivo, la POA evita la dispersión y el entrelazamiento de código, separando la funcionalidad básica de los aspectos no funcionales de una aplicación. Los aspectos no funcionales son precisamente los módulos de una aplicación que pueden ser reutilizados por otras aplicaciones sin tener que realizar una nueva implementación. Respecto al problema de

la latente indisponibilidad de la información, la replicación de información en los sitios de los usuarios es una solución bien conocida en sistemas distribuidos para garantizar la disponibilidad de la información. Particularmente, se utilizó la tecnología Web de *servlets* para permitir el manejo de peticiones concurrentes mediante hilos de ejecución.

Abstract

The recent progress in computational and communicational technologies has revolutionized the way of using computers. This technological revolution reflects a change in the emphasis of employing computing systems to facilitate human interaction, in addition to use them for the resolution of problems. The present research work inscribes in the Computer Supported Cooperative Work (CSCW) field, which studies the social aspects of individual and collective activities (ethnomethodology discipline), as well as the technological aspects of information and communication supports (groupware discipline). Particularly, the solutions proposed by this thesis fall into the groupware discipline, whose main goal is to allow a group of possible distributed persons to effectively collaborate in order to achieve their common objectives by using (generally distributed) groupware systems. Some examples of these ones are message systems, multi-user editors, group decision-support systems, electronic meeting rooms, computer conferencing, workflows, and cooperative learning environments. One of the CSCW application domains that have been only superficially analyzed, although it was one of the first research topics to be investigated, is that of cooperative calendars. Excepting for some commercial products (e.g., IBM Lotus Notes, Apple iCal, and Microsoft Outlook) and open source (e.g., Darwin Calendar Server, Opengroupware.org and Open-Xchange), the majority of the studies consider them as trivial applications or decoration tools provided by others systems, e.g., computer conferencing systems. The motivation of this master thesis resides in exploring two fundamental problems of these applications: 1) the limited flexibility of their building blocks to be adapted to the several usage patterns of groups; and 2) the lack of mechanisms to ensure the information availability when these applications are transposed from the reliable environment of a LAN (communication infrastructure in a physically centralized organization) to the unreliable environment of the Internet (communication infrastructure in a physically distributed organization or in several cooperating organizations). To cope with the limited flexibility problem, we use the Aspect Oriented Programming (AOP) Paradigm, whose main goal is to facilitate code reutilization among applications. To achieve this goal, the AOP avoids code scattering and tangling by separating the basic functionality from the non-functional aspects of an application. The non-functional aspects precisely are the application modules that can be reused by other applications without having to carry out a new design. Relatively to the information latent unavailability problem, the information replication on user sites is well known solution in distributed systems to ensure information availability. Particularly, we use the servlets Web technology to allow the management of concurrent requests by means of execution threads.

Agradecimientos

Un agradecimiento especial a mi madre Elía, por sus consejos, por enseñarme a vivir la vida y por su apoyo incondicional que siempre me ha brindado, porque sin él nunca habría logrado mis metas.

A Luis y Lourdes, por ser pilares en este proyecto de vida.

A mis hermanos Zoila, Adriana y Julio, por enseñarme muchas cosas de la vida.

Al Centro de Investigaciones y Estudios Avanzados del IPN, porque me brindo las herramientas necesarias para realizar esta tesis.

Al Consejo Nacional de Ciencia y Tecnologia, por brindarme los recursos económicos que me ayudaron a concluir mis estudios de maestría.

A mi asesora, la Dra. Sonia Guadalupe Mendoza Chapa, de quien he aprendido mucho a lo largo de este proyecto, pero sobre todo por su paciencia y apoyo que me permitieron concluir la maestría.

Al Dr. Dominique Decouchant y al Dr. José Guadalupe Rodríguez, por sus valiosos comentarios.

A los profesores de la Departamento de Computación, por sus enseñanzas y buenos consejos.

Al Dr. Francisco Rodríguez Henríquez, por brindarme su apoyo en el momento que lo necesité.

Al Dr. Ulises Juárez Martínez, por compartir conmigo un poco de su sabiduría.

A todas esas personas que conocí durante mi estancia en el CINVESTAV y a quienes considero mis amigos. Especialmente a mis amigos de generación: Fabiola Ortega, Fernado García, Cuauhtemoc Mancillas, Christian I. Mejía, Marco A. Negrete, Jorge Ortíz, Eduardo F. Vázquez, Saúl Zapotecas, William De la Cruz, Juan C. Fuentes, Christopher E. Charles, Patricia García, Víctor Serrano y Carlos Valle.

A Sofía Reza, Felipa Rosas, Flor Córdova y Arcadio Morales, por su excelente labor que desempeñan dentro de la Departamento de Computación y que además es fundamental para

nuestro desarrollo como estudiantes.

Finalmente a todos aquellos que tuvieron que ver en el desarrollo de esta tesis y que por descuido no mencione.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Contexto científico

El principal objetivo del campo de investigación denominado Trabajo Cooperativo Asistido por Computadora (TCAC)[1] es permitir a un grupo de personas realizar un proyecto común de manera concertada, aún cuando éstas no puedan reunirse físicamente en el mismo lugar y/o al mismo tiempo (ver figura 1.2 Ref. 1). De esta manera, el TCAC incrementa las oportunidades de trabajo en grupo y provee acceso a los recursos requeridos por dicho trabajo, e.g., datos compartidos y destrezas diversas de los participantes [Prakash et al., 1999]. El interés en este campo de investigación no se restringe a los ámbitos académico y científico, sino que también se ve reflejado en el número creciente de: 1) productos comerciales que han aparecido en el mercado (e.g., IBM Lotus Notes, Microsoft NetMeeting, Microsoft Exchange Server, Novell GroupWise, Apple iCal) y 2) proyectos de código abierto (e.g., Darwin Calendar Server, Horde, Opengroupware.org y Open-Xchange). Incluso, el consorcio W3 puso en evidencia que el TCAC constituye un campo de investigación importante para la evolución de la Web [W3C, 2004].

El fenómeno de la globalización no solamente ha dado lugar a la distribución de las organizaciones, sino también a la cooperación entre organizaciones distribuidas en diferentes partes del mundo. En consecuencia, las personas necesitan trabajar conjuntamente a pesar de la distancia física y de la diferencia horaria. En respuesta a este requerimiento, los desarrolladores de sistemas principalmente han concentrado sus esfuerzos en la construcción de aplicaciones cooperativas que soporten personas distribuidas alrededor del mundo (ver figura 1.2 Ref. 2). Un resultado fructífero de estos esfuerzos es el sistema Alliance [Romero-Salcedo and Decouchant, 1997], el cual permite a un grupo de colaboradores distribuidos en Internet producir conjuntamente documentos estructurados.

[1] Computer Supported Cooperative Work (CSCW) por sus siglas en inglés

Desde el punto de vista conceptual, los servicios de un sistema cooperativo abarcan tres espacios funcionales (ver figura 1.1) [Ellis and Wainer, 1994]:

1. el **espacio de producción** designa los objetos que resultan de la actividad del grupo, e.g., un artículo, un reporte técnico o un programa de cómputo;

2. el **espacio de coordinación** define actores (i.e., individuos, grupos, roles o agentes de software), identifica actividades y tareas así como sus relaciones temporales y designa los actores responsables de dichas actividades y tareas.

3. el **espacio de comunicación** ofrece a los actores la posibilidad de intercambiar información.

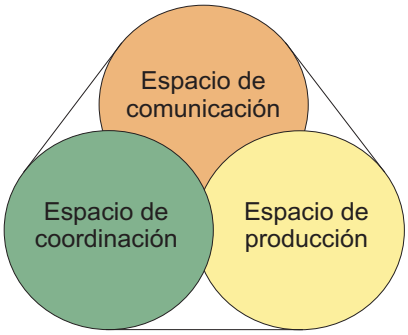


Figura 1.1: Espacios funcionales de un sistema cooperativo

Según el dominio de aplicación de un sistema cooperativo, los tres espacios funcionales no tienen la misma importancia, e.g., los sistemas de edición de documentos privilegian la producción; los sistemas de flujo de trabajo (*workflow*) hacen hincapié en la coordinación; y los espacios de medios (*mediaspaces*) enfatizan la comunicación.

Particularmente, nuestro estudio se enfoca en el dominio de aplicación de las agendas cooperativas que, habiendo estado disponibles desde principios de 1980, constituyen una de las primeras tecnologías cooperativas que salieron a la luz [Palen, 1999]. De acuerdo con el modelo conceptual de los sistemas cooperativos, las agendas privilegian los espacios funcionales de coordinación y comunicación (ver figura 1.2 Ref. 3). Por una parte, las agendas cooperativas crean nuevas oportunidades de coordinación social, ya que permiten a los miembros del grupo hacer inferencias sobre la calidad de la asignación del tiempo [Palen, 1999], e.g., los empleados de una organización pueden deducir la carga de trabajo de sus colegas no sólo por el número de citas en sus agendas, sino también por la naturaleza de las citas. Por otra parte, las agendas cooperativas presentan algunas características especiales que impactan la comunicación interpersonal, e.g., revelación de información y mecanismos de planificación de reuniones (ver figura 1.2 Ref. 4).

Finalmente, cabe mencionar que la agenda cooperativa, propuesta en este trabajo de tesis de maestría, está diseñada para asistir grupos pequeños, entre 3 y 15 (ver figura 1.2 Ref. 5)

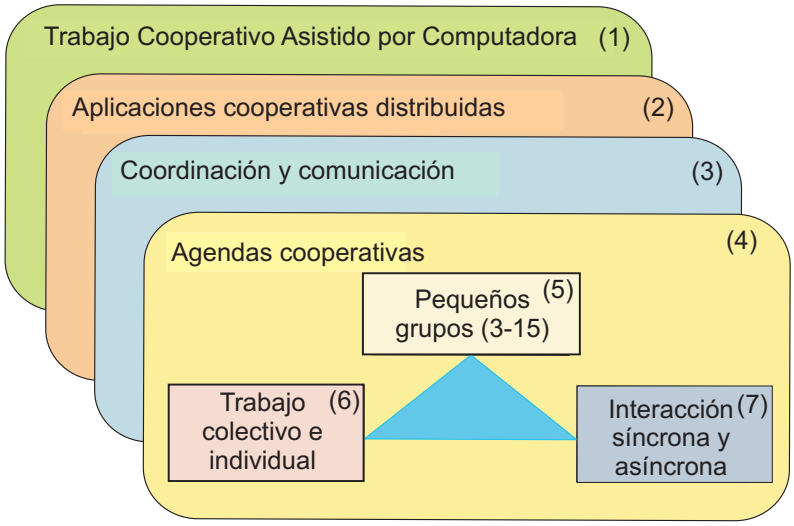


Figura 1.2: Contexto de investigación del trabajo de tesis

individuos, en la planificación de reuniones tanto personales como colectivas (ver figura 1.2 Ref. 6). Puesto que los miembros del grupo pueden estar distribuidos en diferentes partes del mundo y conectados mediante Internet, la agenda cooperativa les permitirá interactuar tanto en forma síncrona (al mismo tiempo) como asíncrona (en diferentes momentos) a través del mismo soporte de colaboración (ver figura 1.2 Ref. 7).

1.2. Problemática

Citadas generalmente como ejemplos triviales en la literatura científica (e.g., [Ellis et al., 1991, Amor et al., 2004, Garrido et al., 2005]), las agendas cooperativas han sido descritas, de manera superficial, como aplicaciones que ofrecen funciones carentes de las complejidades que presentan otros sistemas diseñados para el soporte de la cooperación. Excepto por algunos productos comerciales (e.g., Apple iCal, IBM Lotus Notes y Microsoft Outlook) y de software libre (e.g, Darwin Calendar Server, Opengroupware.org y Open-Xchange), estas aplicaciones han sido raramente consideradas como sistemas cooperativos en todo el sentido de la palabra. La desvalorización de estas aplicaciones se debe principalmente a dos razones:

1. han sido utilizadas como "banco de pruebas" *(testbed)* para estudiar algunos problemas fundamentales de los sistemas cooperativos como la repartición de objetos en tiempo real, e.g., RTCAL [Sarin and Greif, 1985];

2. han sido diseñadas como una funcionalidad secundaria (decoración) de los sistemas de conferencia, e.g., RTCAL y de administración de documentos, e.g., IBM Lotus Notes.

La facilidad de evolución de un sistema cooperativo depende, en gran medida, de su flexibilidad, definida en términos de la capacidad de adaptarse a la diversidad de patrones de

trabajo de los grupos que han sido observados mediante estudios sociológicos [Grudin, 1988], [Grudin and Palen, 1995]. Este trabajo de investigación aborda, por una parte, los problemas relacionados con la flexibilidad limitada de la tecnología de los sistemas cooperativos para: 1) adaptarse a dicha diversidad de patrones de trabajo y 2) facilitar la reutilización de bloques de construcción o módulos del sistema que son comunes a varios tipos de aplicaciones.

Por otra parte, también se pretende resolver el problema al que se enfrenta el desarrollador de aplicaciones cuando pretende implantar una agenda cooperativa en un entorno no fiable, como el de Internet, en términos de la disponibilidad de información. De hecho, las fallas potenciales tanto a nivel de red como a nivel de servidor, así como las eventuales latencias en la transmisión de datos, no permiten garantizar a los usuarios el acceso oportuno a información coherente y actualizada. Este problema pone en evidencia la necesidad de ofrecer, a los usuarios de la agenda cooperativa, un soporte tolerante a estos tipos de fallas.

1.2.1. Fexibilidad limitada en los sistemas cooperativos actuales

Resulta prácticamente imposible ofrecer un abanico completo de patrones de trabajo en un sólo sistema cooperativo. Por esta razón, para construir sistemas cooperativos flexibles, los investigadores de TCAC recurrieron principalmente a la Programación Orientada a Objetos (POO). Gracias a sus numerosas ventajas, e.g., encapsulación, polimorfismo y modularización, la POO constituye innegablemente una tecnología importante que ha facilitado la construcción de innumerables sistemas cooperativos (e.g., COAST [Schuckmann et al., 1999], DOC2U [Morán et al., 2002], DreamObjects [Lukosch, 2004] y Agilo [Guicking and Grasse, 2006]) en la última década.

La POO reduce los problemas de desarrollo y mantenimiento gracias a que favorece la reutilización de código. Aunque la reutilización de código es un requerimiento factible en muchos casos, existen algunos en donde la POO podría no satisfacer este requerimiento [Kiczales et al., 1997]. Este es el caso de aplicaciones cuya construcción no se limita a la simple definición de un conjunto de servicios dados, sino que también necesita tomar en cuenta diferentes propiedades no funcionales como el tiempo real, la distribución, la persistencia y la sincronización de la información.

En este caso, la POO conduce, por una parte, al entrelazamiento del código fuente de los servicios de la aplicación con el código fuente de las diferentes propiedades no funcionales. Por otra parte, da lugar a la dispersión del código de cada una de las propiedades no funcionales. En consecuencia, la reutilización de código se ve comprometida, ya que no es siempre posible reutilizar los diferentes módulos (i.e., servicios o propiedades no funcionales) de una aplicación, independientemente unos de otros.

Un paradigma de programación que promete facilitar la evolución de las aplicaciones y aumentar el grado de reutilización de los módulos constituyentes es la **Programación Orientada a Aspectos (POA)**. Este paradigma ofrece una solución a los problemas de entrelazamiento y de dispersión de código de las propiedades no funcionales de un sistema. Los principios de la POA serán explorados en detalle en el capítulo 3.

1.2.2. Entorno de área local vs entorno de área amplia

La mayoría de las agendas cooperativas propuestas hoy en día han sido utilizadas en el entorno local de una organización para planificar reuniones de trabajo cara a cara. El efecto de la globalización ha conducido, por una parte, a la descentralización de las organizaciones y, por otra parte, a la colaboración entre organizaciones distribuidas. En consecuencia, las reuniones de trabajo cara a cara han sido traspasadas a un entorno virtual soportado por tecnologías especializadas como las conferencias por computadora. Este trabajo de tesis propone la implantación de una agenda cooperativa en Internet, con el fin de soportar la planificación de reuniones tanto físicas como virtuales, cuyos participantes puedan eventualmente estar localizados en diferentes partes del mundo, regidas por zonas horarias diferentes.

Puesto que la mayoría de las agendas cooperativas, propuestas hasta el momento, han sido implantadas en entornos fiables de LANs, estas aplicaciones siguen generalmente un arquitectura centralizada o cliente/servidor, que consiste en un servidor central y varios sitios cliente que permiten a los usuarios interactuar entre sí, a través del servidor central. Sin embargo, estas arquitecturas resultan inadecuadas en un entorno como el de Internet, ya que las fallas potenciales de redes de computadoras o sitios podrían eventualmente imposibilitar a los usuarios el acceso a la información sobre sus reuniones. De aquí, surge la necesidad de diseñar soportes tolerantes a este tipo de fallas y a lentitudes en la transmisión de datos. Una solución viable para resolver el problema de indisponibilidad latente de la información es la utilización de un mecanismo de replicación de información (cf. capítulo 3).

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar una agenda cooperativa flexible e implantable en un entorno no confiable como el de Internet (en términos de la disponibilidad de información), con el fin de permitir a un grupo de colaboradores planificar sus reuniones individuales y colectivas.

1.3.2. Objetivos particulares

1. Definir los servicios de la agenda cooperativa, siguiendo una arquitectura de distribución híbrida, con el objeto de garantizar a los usuarios el acceso oportuno a información coherente y actualizada sobre sus citas personales y colectivas.

2. Diseñar e implementar los servicios de la agenda cooperativa mediante el paradigma de la POA, con el fin de poder ser adaptada facilmente a los diversos patrones de trabajo de los grupos y permitir a otras aplicaciones con requerimientos similares reutilizar y especializar eficazmente sus bloques de construcción.

1.4. Organización de la tesis

El presente documento de tesis está organizado de la siguiente manera ver figura 1.3). En el capítulo 2, se presenta una síntesis de los trabajos más representativos que han sido propuestos en el dominio de aplicación de las agendas cooperativas. Los sistemas analizados provienen tanto de trabajos de investigación como de organizaciones comerciales y de software libre. En el capítulo 3, se exponen los principios del paradigma de la Programación Orientada a Aspectos (POA), así como los diferentes enfoques de implementación de la POA. Este paradigma de programación no solo favorece la reutilización de propiedades no funcionales, en diferentes dominios de aplicación, sino también facilita el mantenimiento y la evolución del sistema. En el capítulo 4, se describen las diferentes estrategias que permiten abordar cada una de las propiedades no funcionales contempladas en el desarrollo del sistema propuesto. Puesto que se pretende dar solución al problema de la falta de disponibilidad de la información en Internet, se estudian particularmente las técnicas de replicación de datos y de control de concurrencia. También se describen las principales arquitecturas de distribución para sistemas cooperativos y algunas de las tecnologías más importantes para la comunicación de datos en Internet. En el capítulo 5, se presenta el diseño del sistema propuesto, particularmente los módulos que lo conforman y sus interacciones, así como los diferentes escenarios que puede soportar el sistema. En el capítulo 6, se exponen los detalles de la implementación del sistema, además de las pruebas realizadas para validarlo y de los resultados obtenidos. Finalmente en el capítulo 7, se enuncian las conclusiones del presente trabajo de tesis y algunas de sus posibles extensiones.

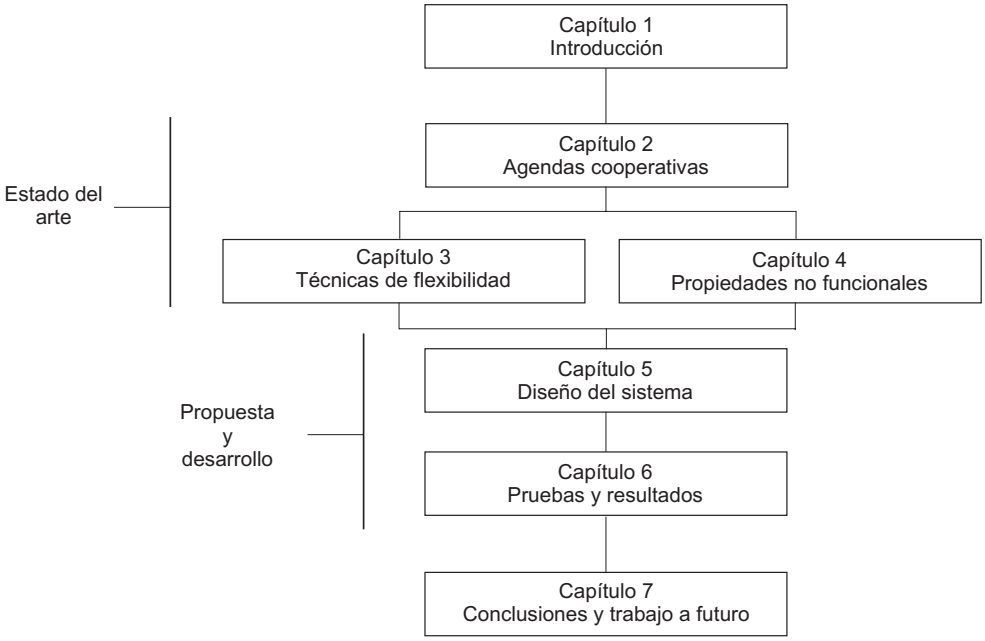


Figura 1.3: Organización del documento de tesis

Capítulo 2

Agendas cooperativas

El objetivo del presente capítulo es realizar un estudio de las agendas cooperativas más relevantes que han sido desarrolladas tanto en trabajos de investigación, como en organizaciones comerciales y de software libre. Primero, se provee una introducción a los sistemas cooperativos, particularmente a las formas de interacción e intercambio de información que permiten, a los usuarios de sistemas cooperativos, tener conciencia de las acciones propias y de los demás, cuando trabajan conjuntamente para lograr un objetivo (sección 2.1). Enseguida, se presenta una síntesis de las diferentes agendas cooperativas estudiadas, así como de los estándares propuestos por algunas comunidades internacionales, e.g., IETF (sección 2.2). Finalmente, se realiza un análisis comparativo de estos sistemas con el fin de poner en evidencia algunos de los problemas recurrentes en el desarrollo de agendas cooperativas (sección 2.3).

2.1. Introducción a los sistemas cooperativos

La mayoría de los sistemas de software soportan únicamente la interacción entre el usuario y el sistema. Incluso, los sistemas diseñados para ambientes multi-usuario (e.g., los sistemas de información de oficina) proveen un soporte mínimo para la interacción entre los diferentes usuarios. Este tipo de soporte es necesario porque una parte significativa de las actividades de una persona ocurre en un contexto grupal en vez de individual.

El término **colaboración** se refiere a un grupo de personas que elaboran un producto o proveen un servicio [Bannon and Schmidt, 1989]. Un punto crucial para la colaboración efectiva es la relación entre el trabajo de cada uno de los colaboradores y el trabajo del grupo, ya que las personas tienden a tomar decisiones autónomas cuando trabajan de manera separada. En la mayoría de los casos, las decisiones se toman bajo condiciones cambiantes e impredecibles que el grupo no puede planear. Por lo tanto, los miembros del grupo deben coordinarse entre sí para colaborar de manera efectiva [Malone y Crowston, 1994].

La colaboración entre las personas que se comprometen en un proyecto común requiere de la coordinación de las actividades relacionadas con el proyecto y de los recursos utilizados en su realización. La coordinación a nivel humano [Schlichter et al., 1997] con frecuencia se

apoya en sistemas de software distribuidos, cuyos componentes negocian entre sí a nivel de aplicación para facilitar la coordinación requerida (ver figura 2.1).

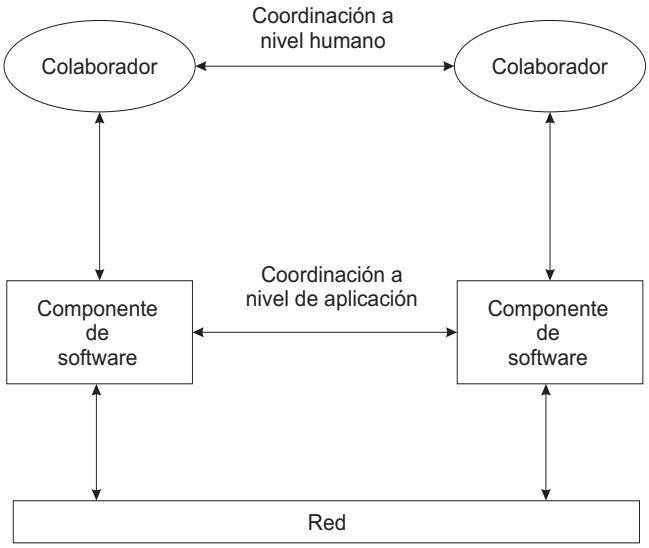


Figura 2.1: Niveles de coordinación para soportar colaboradores distribuidos

La comunicación entre los miembros del grupo también es importante para facilitar la colaboración, especialmente si los colaboradores no trabajan cara a cara ni al mismo tiempo. Específicamente, la comunicación satisface dos requerimientos esenciales [Schlichter et al., 1997]:

1. a **nivel de objetos**, permite el intercambio de información compartida y
2. a **nivel de relación**, facilita la coordinación de las actividades del grupo y del acceso y uso de los recursos compartidos (ver figura 2.2).

A nivel de objetos, las dependencias de coordinación entre los colaboradores están definidas de forma implícita. En consecuencia, la coordinación es soportada mediante el intercambio de información, suficiente para obtener un conocimiento mutuo del progreso y de la situación actual de la colaboración. Por el contrario, a nivel de relación, las dependencias de coordinación entre los colaboradores están definidas de forma explícita. La comunicación en este contexto tiene la intención de iniciar transiciones entre los estados de la colaboración, e.g., una notificación puede informar al usuario que el documento compartido ha alcanzado un cierto estado en el que puede comenzar a trabajar.

El intercambio de información entre los colaboradores puede llevarse a cabo mediante comunicación directa o indirecta (ver figura 2.3). La primera establece un enlace directo de comunicación entre los colaboradores, a través del cual intercambian información, ya sea de manera síncrona (e.g., videoconferencia) o de manera asíncrona (e.g., correo electrónico). La comunicación indirecta asume un espacio de trabajo compartido por los colaboradores en el

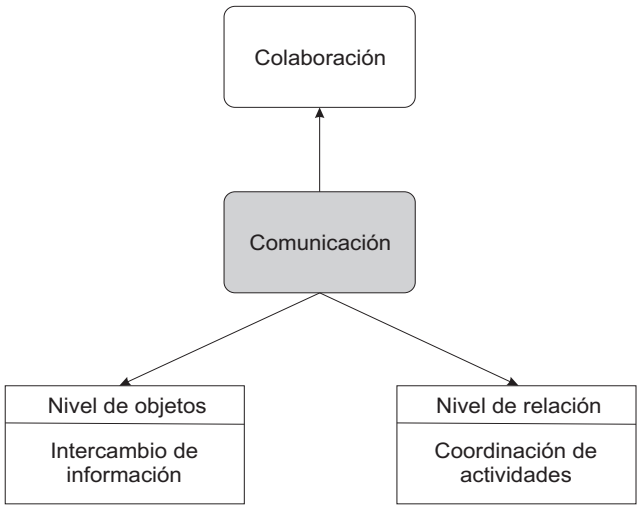


Figura 2.2: Comunicación como soporte de la colaboración

que manipulan objetos para intercambiar y propagar información (e.g., tablero de boletines) [Beaudouin, 1994].

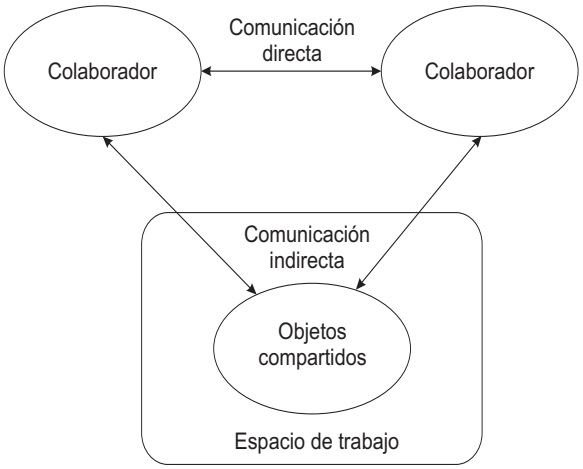


Figura 2.3: Soporte para la colaboración a nivel de objetos

Cuando se habla de sistemas de software que soportan grupos distribuidos, las tareas pueden clasificarse en dos tipos, de acuerdo a su posibilidad de estructuración [Schlichter et al., 1997]: 1) **tareas estructuradas**, las cuales se realizan de acuerdo a un procedimiento preestablecido (e.g., la autorización de un viaje de negocios o de un crédito bancario); y 2) **tareas no estructuradas**, las cuales nunca se efectúan de la misma manera, tales como el trabajo creativo (e.g., la escritura de un artículo científico) o el trabajo sujeto a influencias externas (e.g., las acciones de un corredor de la bolsa de valores).

Dado que estos tipos de tareas son diametralmente opuestos, se requieren mecanismos diferentes para su coordinación (ver figura 2.4). El mecanismo apropiado para las tareas estructuradas es la coordinación explícita. El calificativo de "explícito" se refiere a que la coordinación es realizada mediante acciones iniciadas expresamente para coordinar la tarea. El iniciador puede ser un componente de software que sigue un modelo abstracto, el cual describe los pasos necesarios para completar una tarea determinada. Este tipo de tareas conduce a los llamados sistemas de flujo de trabajo [Domingos et al., 1998].

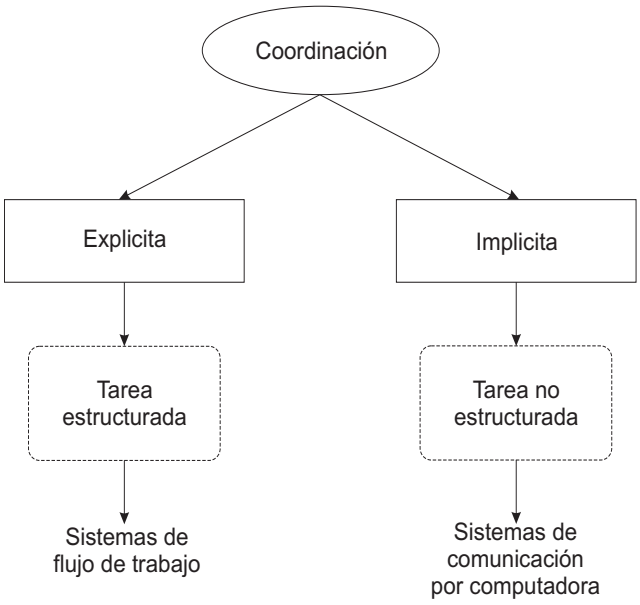


Figura 2.4: Coordinación para diferentes tipos de tareas

En contraste, las tareas no estructuradas carecen de un modelo abstracto que describa los pasos necesarios para completarlas. Por esta razón, el sistema debe permitir a los colaboradores efectuar cualquier acción necesaria para alcanzar su objetivo común. En este contexto, el mecanismo apropiado para las tareas no estructuradas es la coordinación implícita. El calificativo de "implícito" se refiere a que la coordinación debe adaptarse dinámicamente a la situación real, lo cual requiere de mecanismos de conciencia de grupo que permitan a los colaboradores conocer las actividades pasadas, presentes y futuras de sus colegas en el espacio compartido. La información de conciencia de grupo puede transmitirse a los colaboradores de manera directa o indirecta con la ayuda de un sistema de comunicación por computadora [Domingos et al., 1998].

Con el fin de permitir que los colaboradores intercambien información y realicen consensos de sus actividades en el curso de la colaboración, los sistemas cooperativos deben facilitar la interacción humano-humano. Para lograr este objetivo, Dix propone un marco de referencia que permite analizar diferentes formas de interacción en base a cuatro aspectos [Dix, 1997]: 1) comunicación y control, 2) meta-información, 3) efecto de acciones propias y de las acciones de otros y 4) conciencia de grupo.

1. **Comunicación y control**: los colaboradores se comunican directamente, pero el intercambio de información se lleva a cabo ya sea de manera síncrona o asíncrona (ver figura 2.5). En algunos casos, los colaboradores tienen los mismos derechos de acceso a los objetos compartidos, i.e., todos los colaboradores pueden controlar la totalidad de objetos compartidos y percibir su estado.

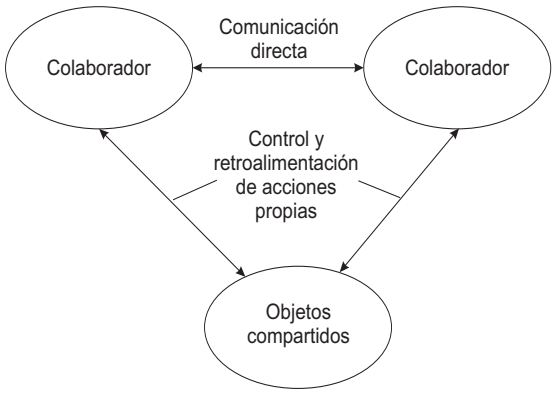


Figura 2.5: Comunicación y control con los mismos objetos compartidos

La compartición de objetos no es un requerimiento de la colaboración. El caso más simple es cuando un solo colaborador tiene el control sobre todos los objetos compartidos (ver figura 2.6 Ref. a). También se puede dar el caso en donde cada colaborador controla diferentes objetos y adquiere conciencia de los cambios realizados sobre los objetos que no manipula (mediante comunicación directa con los demás colaboradores). En estos casos, los objetos no están compartidos (ver figura 2.6 Ref. b). En general, un espacio de trabajo incluye tanto objetos compartidos por el grupo como objetos controlados por un solo colaborador.

2. **Meta-información**: la colaboración no solo depende de la comunicación, sino también requiere que los colaboradores tengan un nivel similar de comprensión sobre el trabajo que están realizando. Al menos, se requiere que los colaboradores hablen un mismo lenguaje y que tengan expectativas de comportamiento similares, así como un nivel de cultura y entendimiento común. Un colaborador puede entonces comunicarse directamente con otro con el fin de compartir conocimientos que permitan comprender el entorno de trabajo (ver figura 2.7).

   Cuando los objetos compartidos están presentes, en el curso de una conversación, los colaboradores puede referirse a dichos objetos ya sea por nombre, descripción o sobrenombre. A estos tipos de referencias se les conoce como *deixis*. Si los colaboradores se encuentran en lugares físicamente diferentes, el uso de la deixis se vuelve más difícil, ya que se requieren expresiones más complejas para referirse a un objeto compartido.

3. **Efecto de acciones propias y de otros**: cuando un grupo de colaboradores comparte un objeto, éste no es el sujeto de la comunicación sino un medio de comunicación.

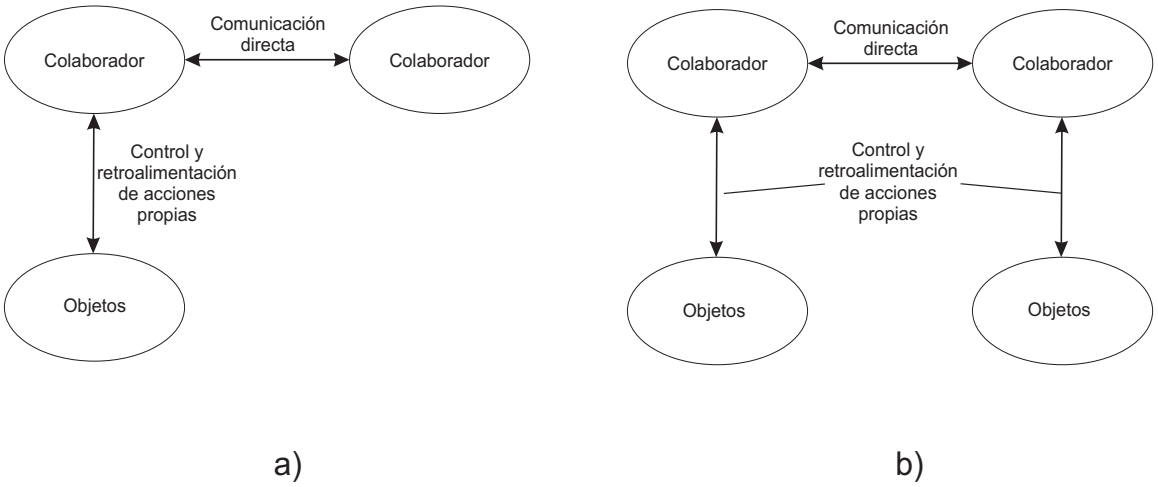


Figura 2.6: Comunicación y control con diferentes objetos

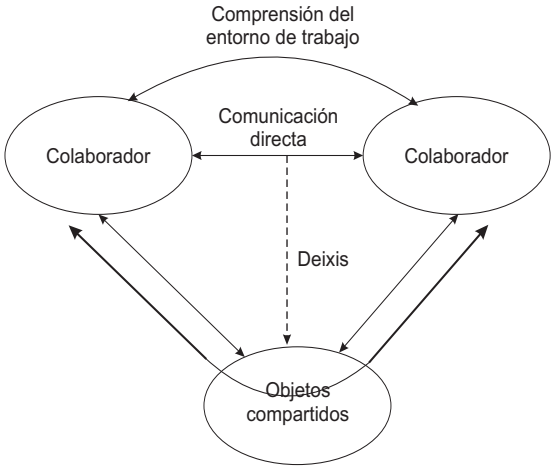


Figura 2.7: Meta-información

Cuando un colaborador realiza modificaciones sobre un objeto compartido, sus colegas observarán los efectos de dichas acciones. En ese sentido, existen dos canales de comunicación por cada colaborador: 1) en uno de los canales, recibe el efecto de sus propias acciones (*feedback*) y 2) en el otro canal, recibe el efecto de las acciones de los demás (*feedthrough*) sobre los objetos compartidos (ver figura 2.8).

4. **Conciencia de grupo**: la información de conciencia de grupo se utiliza a menudo en sistemas cooperativos para permitir que un usuario perciba: 1) la presencia de otros usuarios en el sistema, 2) su disponibilidad para interactuar activamente y 3) lo que está pasando (¿Qué? ¿Cómo? y ¿Por qué?) en el espacio de trabajo compartido (ver

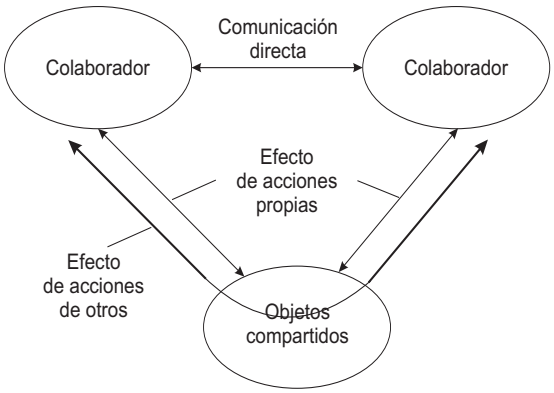


Figura 2.8: Efecto de acciones propias y de otros

figura 2.9). El desarrollo de elementos gráficos (*widgets*) de conciencia de grupo es relativamente complejo debido a que las acciones de un colaborador sobre los objetos compartidos deben reflejarse en las interfaces gráficas de los demás miembros del grupo.

Un problema importante de retardo en la provisión de información de conciencia puede presentarse cuando el sistema requiere mostrar el efecto de las acciones de otros. De hecho, el retardo que conlleva la visualización de esta información en la interfaz gráfica de todos colaboradores es causado principalmente por limitaciones técnicas, e.g., la latencia de red. Aun más, este retardo puede verse agravado por el comportamiento propio del sistema, e.g., la frecuencia con la que los cambios, realizados por un colaborador, son notificados a sus colegas o la frecuencia con la que un colaborador necesita estar enterado del nuevo estado de los objetos.

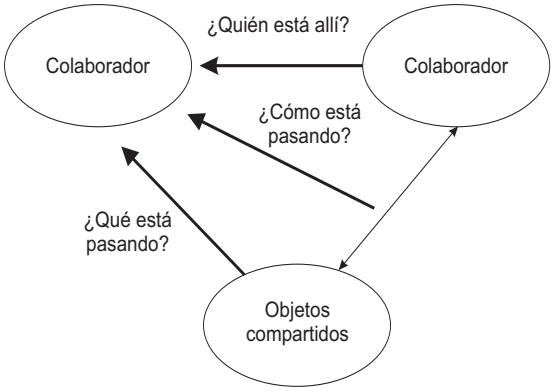


Figura 2.9: Información de conciencia de grupo

2.2. Trabajo relacionado

A pesar de la creciente proliferación de las agendas cooperativas, la investigación empírica conducida al respecto es relativamente poca [Palen, 1999]. Los primeros trabajos remontan a 1982, cuando se llevó a cabo un estudio sobre agendas convencionales con el fin de realizar el diseño de las entonces emergentes agendas electrónicas sin soporte de red [Kelley and Chapanis, 1982]. Posteriormente, los estudios se centraron en el desarrollo de agendas electrónicas personales [Kincaid et al., 1985], aunque también se lanzó el reto de la naturaleza dual (i.e., individual y compartida) de las agendas con soporte de red [Payne, 1993]. Algunos otros estudios hicieron énfasis en el diseño de funciones de base para agendas electrónicas personales [Greif, 1984], pero igualmente analizaron el impacto de algunas de estas funciones en agendas cooperativas [Beard et al., 1990].

Asimismo, los investigadores analizaron algunos problemas sociales y psicológicos (e.g., rechazo, insatisfacción, temor al cambio) que resultan de la adopción de nuevas tecnologías de comunicación (e.g., agendas cooperativas) en el ámbito de una organización. Estos problemas tienen una influencia directa en el diseño de nuevos sistemas, debido a que los sistemas "adoptados" no siempre cumplen con las expectativas de quienes los utilizan. Además, un cambio en las tecnologías de comunicación implica nuevas formas de organización y planeación de las actividades de los empleados para poder alcanzar las metas comunes [Ehrlich, 1987a, Ehrlich, 1987b].

Posteriormente, tomando la planificación de reuniones como principal foco de atención, los investigadores pusieron en evidencia algunos problemas importantes en la adopción de sistemas cooperativos por parte de los empleados de una organización [Grudin, 1988]. La falla ha sido atribuida principalmente a problemas de implantación debido a que la tecnología: 1) no es suficiente para quienes hacen uso del sistema [Francik et al., 1991] o 2) es innecesaria para quienes no lo utilizan [Grudin, 1988].

Años más tarde, algunos trabajos sobre agendas cooperativas identificaron algunos factores sociales y tecnológicos que contribuyeron a la adopción extendida de sistemas cooperativos [Grudin and Palen, 1995]. Sin embargo, estos trabajos están menos centrados en los impactos de la funcionalidad propia de las agendas cooperativas. Más adelante, Mosier y Tamaro realizaron un estudio a corto plazo con el fin de examinar algunas implicaciones del uso individual y colectivo de las agendas cooperativas. Los resultados mostraron que si los usuarios hacen poco uso de la agenda de otro usuario, entonces podría no valer la pena mantener una agenda cooperativa [Mosier and Tammaro, 1997].

Tanto los trabajos de investigación como las organizaciones comerciales y de software libre han realizado estudios y desarrollos en el dominio de aplicación de las agendas cooperativas. La figura 2.10 muestra algunos de los trabajos más representativos que han sido desarrollados en los diferentes ámbitos. Estos trabajos serán estudiados en las sub-secciones 2.2.1 - 2.2.7.

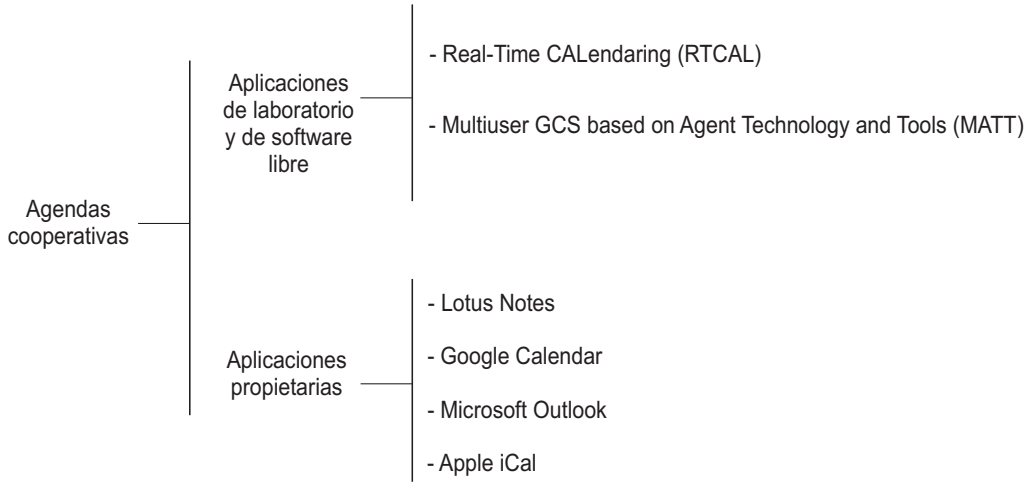


Figura 2.10: Aplicaciones representativas en el dominio de las agendas cooperativas

2.2.1. RTCAL

"Real-Time CALendaring" (RTCAL) [Sarin and Greif, 1985] permite a los miembros de un grupo planificar horarios de reuniones en tiempo real, pero no soporta la selección automática de las ranuras de tiempo asociadas a cada cita. Además, provee información y herramientas para soportar la toma de decisiones.

A nivel de la interfaz gráfica, RTCAL ofrece espacios compartidos y personales donde despliega respectivamente datos públicos y privados. Por cada ranura de tiempo, RTCAL provee la siguiente información a cada miembro del grupo: 1) en la columna compartida, indica si la ranura está disponible para todos los colaboradores y 2) en la columna privada, muestra los detalles de la cita del usuario en esa ranura. Cuando el usuario se desplaza a lo largo de la columna compartida, la columna privada también se desplaza, de manera que ambas columnas siempre muestran la misma fecha y ranura de tiempo.

Cuando un usuario propone un horario de cita, RTCAL envía un aviso a cada participante para solicitarle que acepte o rechace la invitación. Los votos son mostrados y tabulados en una pantalla compartida hasta que todos los participantes hayan votado. Si un participante tarda en dar su voto, el presidente de la conferencia puede dar por terminada la votación para continuar con la conferencia. Los usuarios tienen la opción de ignorar los horarios de reuniones aceptados por el grupo en su ausencia. El presidente de la conferencia se encarga de supervisar las actividades y determinar quien toma el control en un momento dado. Además, el presidente es el único que puede dar por terminada la conferencia.

Las funciones de RTCAL están divididas en funciones de aplicación y funciones de control de conferencia. Las funciones de aplicación pueden ser utilizadas en cualquier momento por el presidente de la conferencia para manipular el calendario. Las funciones de control de conferencia permiten pasar el control entre los diferentes participantes, así como entrar a una conferencia y salir de ella. Los comandos específicos del calendario son enviados a las vistas

de todos los participantes, pero sólo uno a la vez puede tomar el control de estos comandos. Una ventana de resumen presenta el estado de la conferencia, e.g., qué participantes están presentes, quién es el presidente y quién tiene el control.

2.2.2. MATT

"Multiuser GCS based on Agent Technology and Tools" (MATT) [Grosso et al., 2005] es un sistema que provee las funciones básicas de una agenda cooperativa implementada mediante servicios Web. Por lo tanto, MATT puede ser accedido desde cualquier dispositivo de cómputo que soporte esta tecnología. La figura 2.11 muestra la arquitectura de MATT que sigue un modelo de comunicación cliente/servidor [Schussel, 1996]. La comunicación entre los usuarios es de forma asíncrona (i.e., en momentos diferentes) debido a las fallas de comunicación latentes en Internet.

Cada usuario está representado por un avatar. De esta manera, los diferentes avatars comparten información relacionada con las agendas y tareas de los usuarios que representan. La estructura social de los avatars está regida por un modelo organizacional jerárquico basado en unidades, donde cada unidad mantiene las reglas que definen interacciones, rangos y roles específicos.

MATT utiliza una infraestructura multi-agente. Un agente es una entidad autónoma (software) que reacciona a cambios producidos en el sistema y que toma iniciativas para lograr las metas del usuario. MATT define dos tipos de agentes: 1) de asistencia personal y 2) de servicio jerárquico. Los primeros realizan las siguientes operaciones sobre la agenda personal de cada usuario: a) registro de citas, b) administración de solicitudes de citas, c) notificación de citas, d) negociación con otros agentes para encontrar un horario factible para una cita entre varios usuarios y e) sincronización entre la agenda general (almacenada en un servidor central) y la agenda personal del usuario (almacenada en el sitio local).

Los agentes de servicio jerárquico operan sobre el modelo organizacional. Particularmente, estos agentes actualizan y negocian cambios en la estructura social, e.g., creación de nuevos grupos. MATT permite definir un agente maestro jerárquico, el cual se encarga de administrar las reglas de unidad con el fin de mantener actualizadas y coherentes las estructuras de comportamiento.

2.2.3. Estándar iCalendar

iCalendar [Dawson and Stenerson, 1998] es una especificación estándar para el desarrollo de aplicaciones que requieren intercambiar información de agendas. Esta especificación estándar es el resultado del grupo de trabajo de Calendarios y Agendas (*Calendaring and Scheduling Working Group*) del IETF (*Internet Engineering Task Force*). También es conocida como iCal porque es el nombre de la primera aplicación que la implementó en sistemas operativos Apple Macintosh.

iCalendar se basa en un proyecto de código abierto, llamado vCalendar (*Virtual Calendar*) que ofrece herramientas para mantener eventos en línea con formato de agenda. iCalendar permite a los usuarios invitar a reuniones o asignar tareas a otros usuarios mediante correo

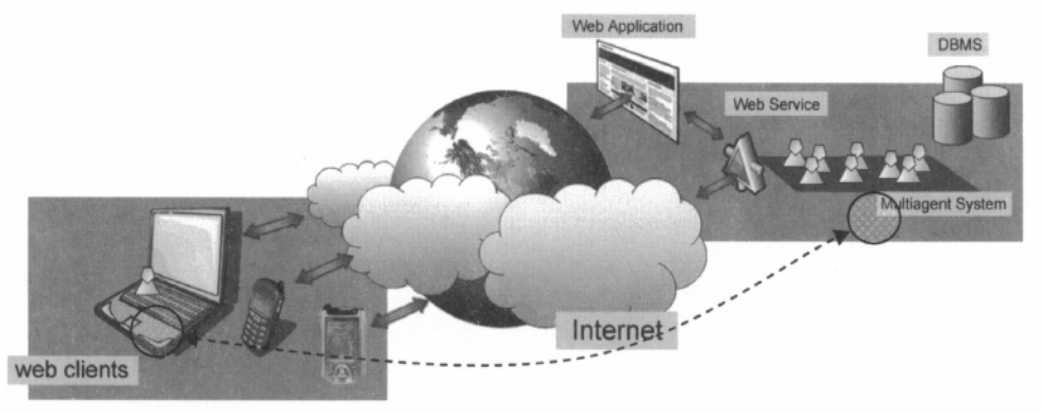


Figura 2.11: Arquitectura de MATT

electrónico. El destinatario puede aceptar o rechazar la invitación o incluso proponer otra fecha y hora para la reunión.

El formato de iCalendar es definido en términos de un tipo de contenido MIME (*Multipurpose Internet Mail Extensions*) [Freed and Borenstein, 1996], el cual permite que un objeto pueda ser intercambiado utilizando varias formas de transporte con respecto a las capas del modelo OSI: a) capa de aplicación: sistemas de archivos, memoria basada en portapapeles, HTTP (*Hypertext Transfer Protocol*) [Fielding et al., 1999] y WebDAV (*Web-based Distributed Authoring and Versioning*) [Slein and Davis, 1999], b) capa de presentación: protocolos de escritorio interactivos, c) capa de red: SMTP (*Simple Mail Transfer Protocol*) [Postel, 1982], d) capa de enlace de datos: comunicación asíncrona punto a punto y e) capa física: red cableada y comunicación infrarroja.

Los objetos de tipo iCalendar asocian este contenido MIME a un conjunto de mensajes para soportar operaciones de planificación tales como: solicitud, notificación, modificación y cancelación de citas y reuniones.

En la tabla 2.1, se muestran las aplicaciones más relevantes, tanto de código abierto como propietarias, que soportan el estándar iCalendar. Además, se indica si las diferentes aplicaciones proveen ($\surd$) o carecen (χ) de soporte para planificar reuniones colaborativamente.

En las secciones 2.2.4 - 2.2.6, se estudian algunas aplicaciones propietarias con soporte para iCalendar.

A continuación se describen brevemente algunas especificaciones estándares complementarias.

Código abierto	**Soporte cooperativo**	**Propietarias**	**Soporte cooperativo**
Darwin Calendar Server	√	Apple iCal	χ
Chandler	χ	FirstClass	√
Mozilla Calendar	χ	Google Calendar	χ
KOrganizer	χ	Lotus Notes	√
Horde	√	Microsoft Outlook	χ
Novell Evolution	χ	Novell GroupWise	√
Opengroupware.org	√	Microsoft Exchange Server	√
PHPicalendar	χ	Microsoft Entourage	χ
phpGroupWare	√	Nuvvo	χ
Simple Groupware	√	Jalios JCMS	√
Webical	χ	Sun Java Calendar Server	χ
Zimbra Collaborative	√	Windows Calendar	χ

Tabla 2.1: Aplicaciones con soporte de iCalendar [ApliCalendar]

CalDAV

CalDAV está diseñada para implementar sistemas cooperativos cliente/servidor que requieren acceso a eventos o datos compartidos. Esta especificación sigue un modelo de eventos para representar citas, reuniones o ranuras de tiempo bloqueadas. Cada evento es expresado mediante el formato iCalendar para permitir que cualquier navegador Web pueda descargar una representación estándar del evento. Los eventos están organizados en colecciones WebDAV [Slein and Davis, 1999] para facilitar la sincronización y la búsqueda.

Un servidor CalDAV realiza las siguientes funciones: 1) analiza archivos de tipo iCalendar, 2) soporta el protocolo de control de acceso de WebDAV [Sedlar et al., 2004] y 3) realiza reportes de tiempo libre y ocupado. Mediante estas funciones, un usuario puede sincronizar su agenda con el servidor CalDAV y compartirla con otros usuarios utilizando múltiples dispositivos. El protocolo también soporta agendas para sitios u organizaciones [Daboo et al., 2007].

GroupDAV

GroupDAV es un protocolo utilizado para conectar aplicaciones cooperativas cliente/servidor de código abierto. Aunque GroupDAV es más limitado que CalDAV, ofrecen algunas funciones similares que son un subconjunto de las funciones de WebDAV [GroupDAV]. Se utiliza principalmente en aplicaciones de software libre que pretenden resolver problemas del mundo real.

hCalendar

hCalendar es una especificación de microformatos abierta (basada en iCalendar) cuya implementación puede estar embebida en formatos (X)HTML, Atomo, RSS y XML. Además, define herramientas que pueden extraer detalles de eventos desde sitios Web. Estos detalles pueden ser visualizados en otros sitios Web mediante aplicaciones de agenda.

hCalendar tiene una representación idéntica a iCalendar en HTML. Los *bloggers* pueden embeber eventos hCalendar directamente en sus páginas Web. Además, hCalendar permite que las aplicaciones recuperen información de dichos eventos, directamente de las páginas Web, sin tener hacer referencia a un archivo separado.

El formato básico de hCalendar utiliza pares de nombres <*objeto, propiedad*> de iCalendar con el fin de transformar directamente objetos iCalendar en elementos XHTML [hCalendar].

vCard

vCard o *VersidCard* es un formato de archivo estándar (para tarjetas electrónicas de negocio) destinado a almacenar nombres, direcciones, números telefónicos, URLs, logos, fotografías, e incluso clips de audio. Fue propuesta originalmente en 1995 por el consorcio *Versit* compuesto por Apple Computer, AT&T, IBM y Siemens. Usualmente, las vCard son adjuntadas a mensajes de correo electrónico, aunque también pueden ser intercambiadas por otros medios.

Así como existe un microformato hCalendar para iCalendar también existe un microformato hCard para vCard. Las funciones de hCard tienen una representación idéntica a las de vCard en (X)HTML para poder embeber datos vCard en páginas Web. Existen también extensiones para navegadores y tecnologías *X2V*, que convierten hCards en vCards, con el fin de proveer interoperabilidad entre publicaciones hCards en la Web y clientes vCard.

El envío de vCards por Bluetooth es, en términos generales, una de las formas más compatibles de *placecasting*, i.e., transmisión de un mensaje a todos los dispositivos con soporte de *bluetooth* localizados dentro de un área geográfica específica. Puesto que el envío de vCards por *bluetooth* no requiere mecanismos de sincronización, algunas aplicaciones las utilizan para el difusión de mensajes anónimos [Dawson and Howes, 1998].

xCal

xCal es una representación en XML de los componentes, de las propiedades y de los parámetros del estándar iCalendar [Dawson et al., 2002]. Particularmente, define un XML/DTD (*Extensible Markup Language/Document Type Definition*) que corresponde a la especificación de objetos básicos para agendas basadas en iCalendar.

Webcal

Webcal en un esquema URI (*Uniform Resource Identifier*) no oficial[1] que se ha convertido en un estandar *de facto* para acceder a archivos con formato iCalendar mediante WebDAV. Webcal fue utilizado por primera vez en Apple iCal (cf. sección 2.2.7).

El prefijo "webcal:" del protocolo Webcal se usa para lanzar un manejador de protocolo externo, el cual pasa la URL de un archivo *.ics* en lugar de su contenido, como suele hacerse en una descarga normal. También se utiliza para lanzar lectores RSS externos [Webcal].

[1]Algunos esquemas URI oficiales registrados en la IANA (*Internet Assigned Numbers Authority*) son HTTP:, ftp:, mailto:, etc.

2.2.4. Lotus Notes

Lotus Notes es un sistema cliente/servidor creado en 1973 por David Woolley en la Universidad de Illinois, pero actualmente es propiedad de *IBM Software Group*. Provee un soporte de colaboración (destinado al ámbito empresarial) que facilita la administración de cuentas de correo electrónico e información de contactos personales y grupos, así como la planificación de citas y tareas. También puede actuar como cliente de Internet multipropósito con el servidor *Lotus Domino* de IBM [McCoy et al., 2001].

Lotus Notes replica los datos en los clientes con el fin de permitirles trabajar fuera de línea y de reducir la carga del servidor. La sincronización entre cada cliente y el servidor tiene lugar en la siguiente conexión, a través de la red o de un enlace punto a punto. La replicación se puede realizar automáticamente por medio de un planificador o manualmente mediante una solicitud del cliente.

Lotus Notes organiza sus datos en estructuras llamadas **documentos**, cuyo contenedor persistente es el sistema de archivos NSF (*Notes Storage Facility*). Un **documento** es un paquete semiestructurado de información que consiste de un conjunto de campos. A diferencia de una tabla en una base de datos relacional, un **documento** no está limitado a un número específico de campos. Durante el ciclo de vida de un **documento**, el número de campos puede variar.

Otra de las funciones principales de Lotus Notes es la mensajería instantánea. Debido a que los mensajes pueden ser enviados de forma automática, este servicio puede facilitar la coordinación de tareas pendientes, la planificación de eventos entre grupos de colaboradores y la administración de sesiones de conversación en "tiempo real". Del servicio de mensajería instantánea se desprende la funcionalidad de la agenda y de la planificación de reuniones.

La agenda de Lotus Notes se encuentra almacenada en la base de datos del servidor *Lotus Domino*. Mediante una replica de dicha base de datos, el colaborador puede gozar de una funcionalidad completa de la agenda. Sin embargo, se debe mantener una conexión al servidor para obtener un mejor desempeño de la agenda. Al utilizar las funciones de planificación de Lotus Notes es posible compartir la agenda con otros usuarios, así como hacer uso de las agendas de los demás.

Mediante una red de servidores Domino, la agenda facilita la organización de grupos de trabajo, la planificación de reuniones laborales, el control de subordinados o compañeros de trabajo y la coordinación de acciones.

Las entradas en la agenda pueden ser codificadas individualmente por color, lo que permite distinguir visualmente, e.g., citas personales, reuniones de negocios o acontecimientos relacionados con los distintos departamentos o proyectos de la empresa.

Las citas de la agenda son sensibles a las zonas horarias, i.e., se puede designar una ranura de tiempo específica para una reunión independientemente de la zona horaria regente. De esta manera, se facilita la coordinación de reuniones y teleconferencias en organizaciones separadas geográficamente.

2.2.5. Google Calendar

Google Calendar es una aplicación Web que sirve para manejar información personal. Está disponible al público a partir de abril del 2006 y actualmente se encuentra en la versión beta. Se requiere tener una cuenta en Google para poder tener acceso a las funcionalidades de la aplicación. Permite crear múltiples agendas, las cuales pueden ser visualizadas en una misma pantalla y compartidas con personas designadas explicitamente por el usuario.

Google Calendar también permite definir el número de días visibles en la interfaz gráfica, la cual presenta una similitud con aplicaciones de escritorio, tales como Microsoft Outlook y Apple iCal. Google Calendar utiliza una interfaz Ajax (*Asynchronous JavaScript and XML*) [Ajax Framework] que permite a los usuarios ver, añadir y arrastrar eventos de una fecha a otra sin necesidad de recargar la página.

Como es una aplicación Web, se puede ejecutar en cualquier sistema operativo utilizando un navegador que soporte las tecnologías Web requeridas. Los eventos de la agenda están almacenados en los servidores de Google. Por lo tanto, la privacidad de la información referente a la agenda del usuario queda a la merced de los administradores de aplicaciones Google.

Los datos de la agenda pueden ser integrados a otras aplicaciones de Google, tales como: a) servicio de correo Gmail, b) página principal iGoogle donde se puede elegir y organizar contenido en forma de "*gadgets*"[2] (la agenda se muestra como un módulo de la página principal), c) escritorio de búsqueda Google Desktop. Inclusive, los datos de la agenda también pueden ser integrados en otras aplicaciones, e.g., *Novell Evolution* y *Windows Calendar* de Windows Vista.

Google Calendar puede ser sincronizado entre dispositivos móviles o computadoras personales. Así los recordatorios de eventos pueden ser enviados vía correo electrónico o SMS en 80 países y regiones [Wikipedia-Google Calendar]. Google Calendar contiene una API que permite crear aplicaciones Web capaces de hacer uso de los datos almacenados en la agenda [Google Calendar].

2.2.6. Microsoft Outlook

Microsoft Outlook es un administrador de información personal que provee una agenda, un administrador de tareas y de contactos personales, un block de notas, un diario y un navegador Web. Puede utilizarse como aplicación independiente o en conjunto con Microsoft Exchange Server y Microsoft Office SharePoint Server para soportar funciones de intercambio de información entre múltiples usuarios. Algunas de las funciones que ofrece son: administración de buzones, carpetas públicas e información de agenda y localización de ranuras de tiempo para citas. Cuando Outlook se utiliza en combinación con Microsoft Exchange Server, los usuarios se benefician de un mejor soporte de colaboración y de una mayor seguridad de sus datos. Outlook no ofrece soporte para otros formatos de agendas, e.g., iCalendar, CalDAV, SyncML y vCard 3.0. La versión de 2007 permite crear una representación HTML de la agenda con el fin de poderla compartir por medio de la Web [MsOutlook].

[2]Un "*gadget*" es una mini-agenda que permite revisar la agenda general sin tener que utilizar un navegador

2.2.7. Apple iCal

Apple iCal es una aplicación para la administración de información personal que se ejecuta en sistemas operativos Mac OS X. Fue la primera aplicación en ofrecer soporte para múltiples agendas y en publicar/suscribir agendas en servidores WebDAV. iCal realiza un seguimiento de eventos y citas, con el fin de ofrecer múltiples vistas de agendas para identificar conflictos de horario y tiempo libre. Para resolver conflictos de horario, iCal comprueba la disponibilidad de los usuarios y busca el horario más conveniente para todos antes de invitarlos a una reunión. Asimismo, da la posibilidad de consultar, modificar y proponer citas fuera de línea, las cuales serán actualizadas automáticamente en la siguiente conexión al servidor [iCal]. iCal puede ser compartido a través de la Web por HTTP o WebDAV. Los usuarios pueden suscribirse a las agendas de otros usuarios para mantener contacto con ellos y compartir horarios de eventos. También permite la recepción de notificaciones de eventos próximos por correo electrónico y SMS (*Short Message Service*). iCal además integra los servicios de Apple Sync para sincronizar datos en dispositivos móviles, e.g., PDA's, iPod y iPhone [Apple iCal].

2.3. Análisis comparativo

En esta sección, se realiza un análisis comparativo de los diferentes sistemas estudiados en el estado del arte. Después de describir las principales características que serán consideradas en el análisis, se presenta una tabla de los diferentes sistemas estudiados, así como la discusión consecuente que concluye este capítulo. Por medio de este análisis, se evidencian algunos de los problemas recurrentes (en el diseño de agendas cooperativas) que motivaron el desarrollo del presente trabajo de tesis de maestría.

- **Flexibilidad**: se refiere a la capacidad del sistema que le permite poder ser adaptado por el desarrollador de aplicaciones con el fin de añadir nuevas funcionalidades, modificar las existentes o eliminar aquellas que ya no cumplen con los requerimientos del grupo.
- **Técnica de diseño**: se refiere a la(s) técnica(s) de ingeniería de software utilizada(s) para construir un sistema que posea alguna(s) propiedad(es) específica(s), e.g., modularidad o reutilización de código. Los paradigmas orientados a objetos y a aspectos son algunos ejemplos de técnicas de diseño e implementación
- **Privacidad de datos**: se refiere a la capacidad del sistema que ofrece al usuario de decidir qué datos permanecen privados y cuáles se vuelven públicos.
- **Localización de datos**: se refiere a la propiedad del sistema que provee al usuario de mantener sus datos privados localmente, mientras que sus datos públicos pueden ser transferidos eventualmente a otros sitios, e.g., los de sus colegas.
- **Entorno de comunicación**: se refiere al alcance de la infraestructura de comunicación en la que el sistema puede ser implantado, e.g., red de área local (LAN por sus siglas en inglés) o red de área amplia (WAN por sus siglas en inglés).

- **Sensibilidad a zona horaria**: se refiere a la capacidad del sistema para soportar múltiples usuarios dispersos en lugares regidos por zonas horarias diferentes.

- **Interacción entre usuarios**: es el tipo de soporte que el sistema ofrece a los usuarios para interactuar entre ellos. Se pueden distinguir dos tipos de interacciones: síncrona y asíncrona. Si los usuarios pueden interactuar al mismo tiempo y enviar/recibir actualizaciones tan pronto como se produzcan, entonces los usuarios interactúan síncronamente. De lo contrario, la interacción es asíncrona.

- **Disponibilidad de la información**: se refiere a la propiedad del sistema que ofrece en todo momento información pública relevante sobre la agenda de un colaborador (e.g., ranuras de tiempo libre), aún cuando éste no se encuentre en línea en todo momento.

- **Manejo de la concurrencia**: se refiere al mecanismo que emplea el sistema para resolver el problema que surge cuando dos o más usuarios quieren hacer uso simultáneo de la misma ranura de tiempo. Los mecanismos que permiten el manejo de la concurrencia pueden ser clasificados en: 1) esquemas automáticos (e.g., candados y serialización de operaciones) y 2) esquemas manuales también llamados protocolos sociales (i.e., provisión de herramientas y de la información necesaria y suficiente para permitir a los colaboradores coordinar ellos mismos sus acciones).

- **Soporte de diferentes tipos de grupos**: se refiere a los tipos de grupos soportados por el sistema. Así, un sistema puede soportar usuarios jerarquizados, democráticos o ambos. Este soporte tiene una influencia directa en los tipos de herramientas adicionales ofrecidas por el sistema para resolver eventuales conflictos en la planificación de reuniones.

La tabla 1.2 muestra las características previamente descritas (columna izquierda) de algunos de los sistemas analizados a lo largo del capítulo (fila superior).

A partir del estado del arte, se puede apreciar que numerosas agendas personales y cooperativas han sido desarrolladas, sin embargo la mayoría de ellas proviene del ámbito comercial. En consecuencia, difícilmente podrían ser adaptadas a los requerimientos específicos de los diversos grupos de colaboradores, aunque ofrezcan mecanismos de configuración a nivel de usuario final (e.g., los agentes en MATT). De hecho, estos mecanismos ofrecen un limitado soporte de adaptación que solo satisface parcialmente las necesidades de los grupos, las cuales son susceptibles de cambiar dependiendo de la etapa del proyecto.

Las referencias bibliográficas sobre sistemas provenientes de laboratorios de investigación o de software libre (e.g., RTCAL y MATT) no mencionan si se utilizó alguna técnica de diseño que facilite la reutilización de código o el mantenimiento del sistema. En cuanto a los sistemas de índole comercial es prácticamente imposible conocer esta información. Aún si se publicara esta información, el beneficio sería irrelevante dado que las únicas personas susceptibles de reutilizar el código son los propios empleados de la empresa.

Desde la aparición de los primeros sistemas (e.g., RTCAL), los desarrolladores de aplicaciones tomaron en cuenta las implicaciones concernientes a los problemas de privacidad

Características del sistema	RTCAL	MATT	Lotus Notes	Google Calendar	Apple iCal	Microsoft Outlook
Flexibilidad	χ	χ	χ	χ	χ	χ
Técnica de diseño	No especifica	Agentes	No especifica	No especifica	No especifica	No especifica
Privacidad de datos	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Localización de datos	$\surd$	$\surd$	$\surd$	No soportado	No soportado	No soportado
Entorno de comunicación	LAN	WAN (Servicios Web)	LAN y WAN	WAN(Web)	LAN y WAN(Web)	LAN y WAN(Web)
Sensibilidad a zona horaria	χ	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Interacción entre usuarios	Síncrono	Asíncrono	Asíncrono	Asíncrono	Asíncrono	Asíncrono
Disponibilidad de la información	χ	$\surd$	$\surd$	χ	$\surd$	χ
Manejo de la concurrencia	Manual mediante votación	Automática mediante agentes	Manual mediante mensajes	Manual, haciendo visible los horarios disponibles de los participantes	Manual, haciendo visible los horarios disponibles de los participantes	Manual, haciendo visible los horarios disponibles de los participantes
Soporte de diferentes tipos de grupos	χ	Por jerárquias	Por jerárquias	χ	χ	χ

Tabla 2.2: Comparación entre distintos sistemas que ofrecen agendas cooperativas

de la información. Sin embargo, la administración de la información contenida en algunas agendas propietarias (e.g., Google Calendar, Apple iCal y Microsoft Outlook) es realizada por organizaciones completamente ajenas a los usuarios. Este hecho podría ser una de las causas principales de rechazo de estos sistemas, ya que algunos usuarios prefieren administrar ellos mismos su información personal o, al menos, mantenerla en sus propios dispositivos de almacenamiento.

Puesto que RTCAL salió a la luz a mediados de los 80s, resulta comprensible que este sistema solo puede ser implantado en una red de área local. La mayor parte de los subsiguientes sistemas (e.g., MATT, Lotus Notes, Google Calendar, Apple iCal y Microsoft Outlook) proveen soporte para redes de amplia (principalmente la Web), aunque algunos de ellos (e.g., Lotus Notes, Apple iCal y Microsoft Outlook) también pueden ser implantados en una red de área local.

De igual manera, resulta evidente que RTCAL no requiere ser un sistema sensible a zonas horarias, puesto que está destinado a soportar grupos de colaboradores que trabajan en un mismo edificio. Por el contrario, los demás sistemas son sensibles a zonas horarias debido a que pueden ser implantados en una red de área amplia. En acorde con esta característica,

estos sistemas permiten a los colaboradores interactuar de forma asíncrona (en diferentes momentos), mientras que RTCAL soporta la interacción síncrona (al mismo tiempo).

Sistemas como MATT, Lotus Notes y Apple iCal dicen proveer mecanismos de replicación para garantizar la disponibilidad de información relevante aún cuando no todos los participantes estén en línea en un momento dado. Por otra parte, a excepción de MATT, que utiliza la tecnología de agentes como mecanismo para controlar la concurrencia, la mayoría de los sistemas se basa en protocolos sociales para resolver eventuales conflictos entre los colaboradores. En otras palabras, estos sistemas ofrecen información y/o herramientas para asistir a los usuarios en la toma de decisiones o resolución de conflictos.

Finalmente, sistemas como MATT y Lotus Notes soportan grupos de usuarios que siguen una jerarquía, mientras que los demás sistemas soportan grupos cuyos miembros son socialmente iguales. Sin embargo, a nuestro saber, ninguno soporta ambos tipos de grupos.

Capítulo 3

Técnicas de flexibilidad

El objetivo de este capítulo es justificar la utilización del paradigma de la Programación Orientada a Aspectos (POA) como medio para dotar, a la agenda cooperativa propuesta en esta tesis, de la propiedad de flexibilidad. Primero, se pone en evidencia la superioridad de la POA sobre la Programación Orientada a Objetos para resolver los problemas de entrelazamiento y dispersión de código que dificultan la reutilización de módulos y el mantenimiento del sistema (sección 3.1). A continuación, se describen y ejemplifican los conceptos básicos de la POA (sección 3.2) y los diferentes enfoques de implementación (sección 3.3). Estos enfoques difieren básicamente en la representación y composición de los aspectos no funcionales, lo cual tiene repercusiones en la flexibilidad y eficiencia del sistema. Finalmente, se presenta una síntesis de los diferentes lenguajes para el desarrollo de aspectos (sección 3.4).

3.1. Antecedentes

Como se mencionó en la introducción de esta tesis, el paradigma de la Programación Orientada a Objetos (POO) favorece la reutilización de código y provee un enfoque que permite resolver adecuadamente problemas reales. Sin embargo, existen algunos problemas que no pueden ser solucionados mediante el paradigma de la POO.

Kiczales et al. [Kiczales et al., 1997], definen los paradigmas de la programación imperativa, modular u orientada a objetos como Lenguajes de Propósito General (LPG) ya que tienden a implementar todos los requerimientos de un sistema a lo largo de una sola dimensión. Esta única dimensión resulta conveniente para las funciones básicas del sistema, pero es inadecuada para implementar otros requerimientos, e.g., propiedades no funcionales. Esta limitación da lugar a dos problemas importantes: 1) entrelazamiento y 2) dispersión de código. El primer problema se presenta cuando dos o más propiedades no funcionales conviven conjuntamente en un mismo módulo del sistema. El segundo problema se produce cuando una propiedad no funcional se encuentra dispersa en varios módulos [Laddad, 2002].

Una solución a los problemas de entrelazamiento y dispersión de código de las propiedades no funcionales consiste en desacoplar sus definiciones según el principio de la separación de asuntos ("*separation of concerns*"). Partiendo de este principio, Kiczales et al.

[Kiczales et al., 1997], formalizaron las diferentes etapas de la realización de un sistema para desembocar en el paradigma de la Programación Orientada a Aspectos (POA).

Los mismos autores mencionan que un desarrollador de aplicaciones que utiliza LPG puede lograr la separación de asuntos mediante: 1) componentes o 2) aspectos. Un componente puede ser claramente encapsulado en un procedimiento general, ya que es una unidad de la descomposición funcional del sistema. Por el contrario, un aspecto no puede ser encapsulado en un procedimiento general porque es una propiedad no funcional que afecta el desempeño o la semántica de los componentes del sistema. Algunos ejemplos de este tipo de propiedad son: la persistencia, la distribución, la actualización, la administración, el control de concurrencia, la sincronización, el manejo de memoria, el chequeo de errores y la seguridad de redes.

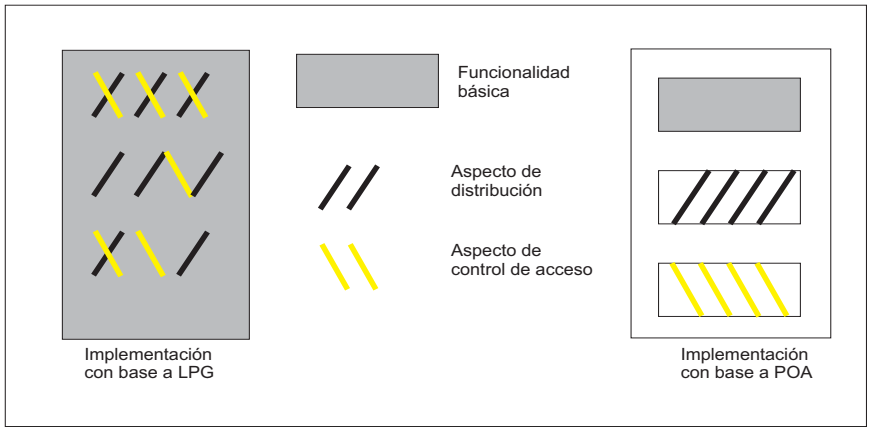


Figura 3.1: Implementación de una aplicación en LPG y en POA

La POA permite al desarrollador de aplicaciones escribir y visualizar (de manera eficiente e intuitiva) una propiedad dispersa en el sistema como una entidad separada. Para lograrlo, la POA requiere que el desarrollador separe: 1) la funcionalidad básica de las propiedades no funcionales y 2) las propiedades entre sí, mediante mecanismos de abstracción y composición. La parte izquierda de la figura 3.1 muestra un ejemplo de una implementación basada en LPG en donde las propiedades no funcionales (i.e., aspectos de distribución y de control de acceso) están entremezcladas no sólo con la funcionalidad básica, sino también entre ellas. Por el contrario, la parte derecha de la figura 3.1 muestra una implementación basada en la POA en donde las propiedades no funcionales están separadas tanto de la funcionalidad básica como entre sí.

3.2. Fundamentos de la POA

Como es bien sabido, para llevar a cabo una implementación tradicional se necesita de: 1) un lenguaje de programación, 2) un compilador o interprete del leguaje y 3) un programa escrito en ese lenguaje que implemente el sistema (ver la parte izquierda de la figura 3.2). A diferencia de la implementación tradicional, la implementación basada en la POA requiere de: 1) un lenguaje que implemente el aspecto de base (funcionalidad básica), 2) uno o más lenguajes que implementen los aspectos no funcionales (propiedades no funcionales), 3) un módulo de configuración que defina el entrelazamiento de cada uno de los aspectos no funcionales con el aspecto de base y finalmente 4) un mecanismo de composición (*weaver*) que cree el sistema final (ver la parte derecha de la figura 3.2).

Según el paradigma de la POA, un sistema se realiza en dos fases:

1. los diferentes aspectos se definen separadamente unos de otros; las definiciones de los aspectos deben estar desacopladas, i.e., carentes de dependencias entre si;

2. el sistema se construye por medio de la composición de los aspectos así definidos. El retardo en la composición permite relajar el acoplamiento entre los diferentes aspectos, ofreciendo nuevas perspectivas de reutilización.

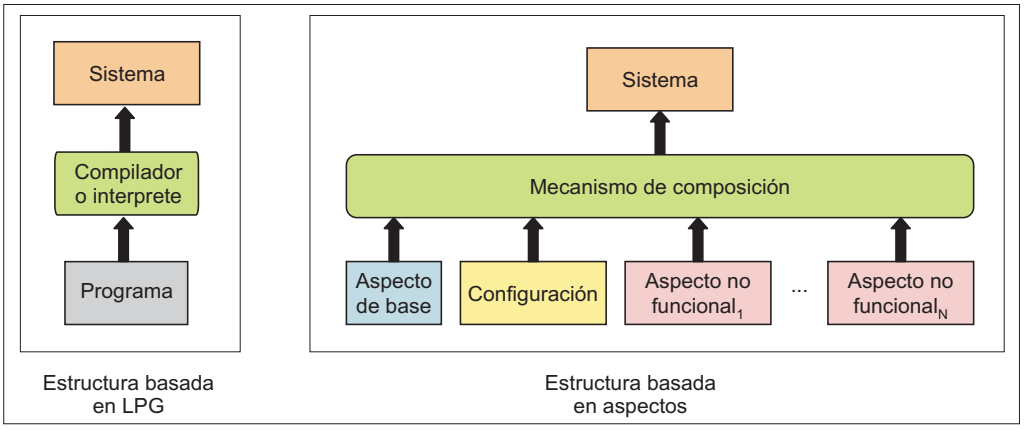


Figura 3.2: Descomposición de un sistema según los enfoques LPG y POA

Particularmente, el paradigma de la POA define un aspecto de base y varios aspectos no funcionales:

- el **aspecto de base** define las funciones básicas del sistema (i.e., el aspecto de base corresponde al "Qué" del sistema) e.g., el aspecto de base de una librería electrónica podría definir las funciones "buscar libro" o "pasar una solicitud de adquisición";

- los **aspectos no funcionales** definen los mecanismos que rigen la ejecución del sistema (i.e., los aspectos no funcionales definen el "Cómo" del sistema) e.g., la librería electrónica podría contener los aspectos no funcionales de distribución, persistencia y sincronización de la información. Cada aspecto no funcional es independiente de los otros y define un mecanismo de ejecución particular.

Puesto que los diferentes aspectos son módulos independientes unos de otros, es necesario entrelazarlos con el fin de construir el sistema. La unión de cada aspecto no funcional con el aspecto de base es definida en un **módulo de configuración** en términos de **puntos de unión** y **cortes en punto**.

Un punto de unión (*join point*) es un punto definido en el flujo de ejecución del aspecto de base en donde es posible agregar comportamiento adicional. Algunos ejemplos de puntos de unión son: invocación a un método y asignación de un valor a un atributo.

Un corte en punto (*pointcut*) es un predicado que debe ser cumplido por un conjunto de puntos de unión y que expone el contexto de ejecución de dichos puntos [Miles, 2004]. Un corte en punto siempre está asociado a un aviso (*advice*), que hace referencia al código que se ejecuta cuando se invoca a un aspecto no funcional. Más precisamente, un aviso es la entidad que implementa el comportamiento adicional o que sustituye el comportamiento definido por defecto en el sistema. Contiene un conjunto de reglas que, cuando finaliza su ejecución, devuelve el control al aspecto de base para que prosiga con las tareas pendientes.

Finalmente, el mecanismo de composición (*weaver*) realiza el entrelazado de los diferentes aspectos, con base en el módulo de configuración, para construir el sistema final. Existen tres formas de entrelazar el aspecto de base con los aspectos no funcionales:

- **Pre-compilación**: como resultado del entrelazamiento entre el aspecto de base y los aspectos no funcionales, se produce un nuevo código fuente en donde se inserta el código de los avisos en los puntos de unión correspondientes.

- **Compilación**: se genera el código objeto que contiene el aspecto de base y los aspectos no funcionales.

- **Ejecución**: se realiza un entrelazado dinámico que controla la ejecución del programa, ya que cada vez que se llega a un punto de unión que hace referencia a un corte en punto de un aspecto, se ejecuta el código del aviso asociado.

La figura 3.3 muestra gráficamente un sistema que evoluciona de una implementación basada en LPG a una implementación basada en POA. En la parte derecha de la figura 3.3 Ref. 1, se muestran los módulos del sistema construido en LPG. Las cajas sombreadas representan código mezclado y disperso en los módulos del sistema, i.e., los aspectos no funcionales. En consecuencia, las cajas sombreadas realizan el comportamiento adicional del sistema, e.g., una solicitud de acceso a un recurso compartido o de actualización del estado de dicho recurso.

El poder de la POA reside en la capacidad de poder reutilizar un aspecto no funcional (e.g., control de concurrencia) en diferentes sistemas con requerimientos similares (e.g.,

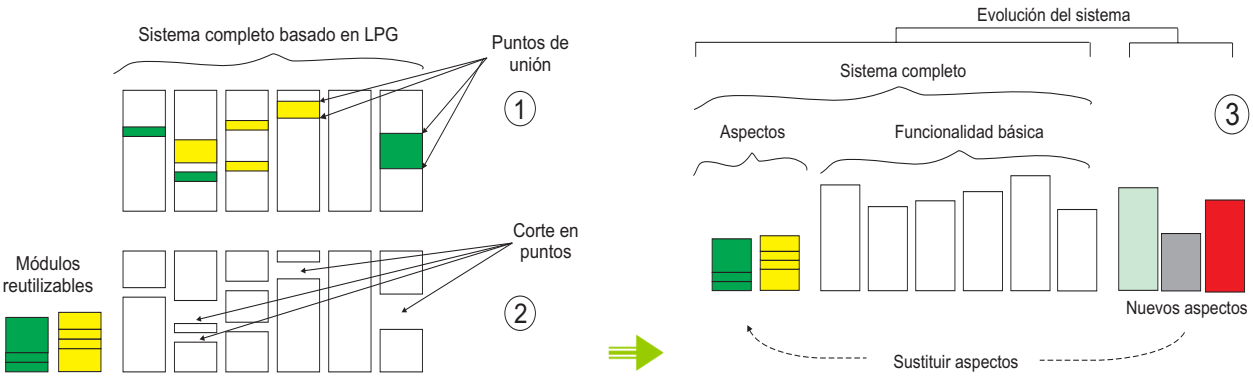


Figura 3.3: Evolución de un sistema: de una implementación en LPG a una implementación en POA

agenda cooperativa, editor multiusuario y pizarrón compartido), aunque el comportamiento adicional que ofrece dicho aspecto pueda ser implementado de diferentes maneras (e.g., candados optimistas vs. candados pesimistas). Mediante la aplicación de los conceptos básicos de la POA (e.g., puntos de unión, cortes en punto y avisos) el código del comportamiento adicional puede ser encapsulado en módulos reutilizables e independientes.

Así, de una manera sencilla pero eficaz, es posible intercambiar un aspecto no funcional por otro que realice la misma tarea de manera más óptima o adaptada a los requerimientos del sistema. Incluso es posible intercambiarlo por otro aspecto no funcional que realice una tarea diferente sin tener que llevar a cabo un rediseño de los módulos del sistema (ver figura 3.3 Ref. 2). Mediante el enfoque de la POA, un sistema se vuelve flexible gracias a que el aspecto de base está separado de los aspectos no funcionales. En consecuencia, se puede añadir, substituir o eliminar fácilmente el comportamiento adicional (ver figura 3.3 Ref. 3).

3.2.1. Ejemplo para ilustrar la diferencia entre POO y POA

Para ilustrar las limitaciones de reutilización de la POO, considere el ejemplo de una librería electrónica (ver figura 3.4). Los clientes utilizan una red para consultar la lista de libros disponibles y eventualmente comprarlos. Varios clientes pueden estar conectados al mismo tiempo y varias solicitudes pueden ser procesadas en paralelo.

El diseño orientado a objetos del sistema, conduce a definir las clases `Librería`, `Cliente`, `Libro` y `Banco`. Por otro lado, se pueden identificar al menos tres aspectos diferentes que corresponden a las propiedades no funcionales de este sistema: **distribución**, **persistencia** y **sincronización**. El aspecto de distribución proviene del hecho de que las instancias de las clases identificadas están localizadas en diferentes sitios remotos entre sí. Este aspecto reagrupa, entre otras funciones, la administración de las comunicaciones distantes y la definición del protocolo de comunicación utilizado. El aspecto de persistencia aparece en respuesta a la necesidad de asegurar que los objetos, e.g., libros puestos en venta, "sobrevivan" a un fallo del sistema. El almacenamiento de estos objetos sobre un soporte masivo y la administración de las actualizaciones de dicho almacenamiento forman parte de la definición del aspecto de

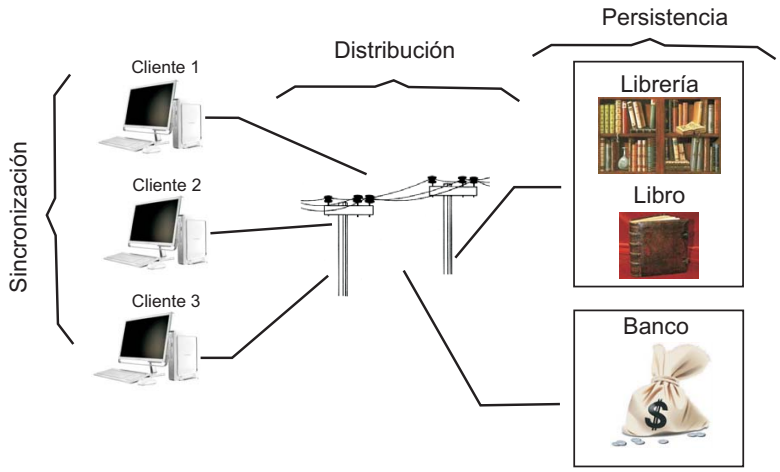


Figura 3.4: Ejemplo de los aspectos de una librería electrónica

persistencia. Finalmente, el aspecto de sincronización surge de la necesidad de sincronizar los accesos concurrentes a las estructuras de los objetos, e.g., un cliente solicita el precio de un libro mientras que la librería modifica este precio.

La implementación de la librería electrónica en un lenguaje a objetos convencional, tal como C++ o Java, introduce dos problemas que comprometen la reutilización [Kiczales et al., 1997]. Por una parte, el paradigma a objetos conduce a un entrelazamiento del código fuente correspondiente a los servicios del sistema con el de los aspectos de distribución, persistencia y sincronización. Por otra parte, el paradigma a objetos da lugar a la dispersión del código fuente de cada uno de estos aspectos.

Entrelazamiento de código

Se ilustrará el entrelazamiento de código mediante una implementación posible de la clase `Libro` y de la sincronización de los accesos a la variable de instancia `precio`. En la figura 3.5, se muestra el código correspondiente en una sintaxis Java. Las líneas de código que representan el aspecto de sincronización (*en cursivas*) están íntimamente entrelazadas con las líneas correspondientes a los servicios realizados por la clase `Libro`. Tales entrelazamientos afectan la legibilidad del código haciendo difícil su comprensión, mantenimiento y reutilización, e.g., es dificil reutilizar la clase `Libro` en un contexto que necesita otra política de sincronización.

La solución polimórfica, que consiste en utilizar el mecanismo de herencia múltiple para introducir otra política de sincronización, no es satisfactoria. De hecho, la definición de subclases que modifiquen las reglas de sincronización plantea un problema de anomalía de herencia [Matsuoka and Yonezawa, 1993]. Este problema se traduce en la obligación de redefinir algunos e incluso todos los métodos de la superclase en las subclases con el fin de obtener el comportamiento deseado. Este requerimiento reduce aún más las posibilidades de reutilización ofrecidas normalmente por el mecanismo de herencia.

```
1:  public class Libro{
2:    private String titulo, autor;
3:    private int precio;
4:    Monitor monitor = new Monitor();
5:
6:    public void escribirPrecio (int nuevoPrecio){
7:        /*Se prohíbe la lectura y escritura*/
8:        monitor.ProtegerLecturaEscritura();
9:
10:       this.precio = nuevoPrecio;
11:
12:       /*Se autoriza la lectura o escritura*/
13:       monitor.AutorizarLecturaEscritura();
14:
15:   }
16:   public int leerPrecio(){
17:       int precioActual;
18:       /*Se prohíbe la escritura*/
19:       monitor.ProtegerEscritura();
20:
21:       precioActual = this.precio;
22:
23:       /*Se autoriza escritura*/
24:       monitor.AutorizarEscritura();
25:
26:       return precioActual;
27:   }
```

Figura 3.5: Ejemplo de entrelazamiento de código mediante la POO

Una solución a la anomalía de la herencia podría ser la construcción de sistemas mediante patrones de diseño ("*desing patterns*"), e.g., "Estrategia" o "Estado" [Gamma et al., 1995]. Sin embargo, esta solución puede revelarse muy pesada, ya que aumenta considerablemente el número de clases, desvíos ("*indirections*") y comunicaciones, lo cual resulta perjudicial para el rendimiento del sistema. Por otra parte, el incremento de clases e interdependencias complica el mantenimiento y la evolución del sistema.

La anomalía de la herencia y la limitación de las posibilidades de reutilización no son específicas de la sincronización. Los mismos problemas se plantean cada vez que un objeto es regido por una o varias propiedades no funcionales, e.g., un objeto `Libro` debe ser persistente y estar sincronizado. La definición de los servicios realizados por el sistema no puede ser fácilmente reutilizada independientemente de las propiedades no funcionales. Además, es difícil reutilizar una propiedad no funcional en otro contexto.

Dispersión de código

La reutilización de una propiedad no funcional también es difícil ya que su definición está dispersa; por lo tanto, no puede ser aislada con el fin de ser reutilizada. De hecho, una misma propiedad puede involucrar varios objetos, e.g., tanto los clientes como los libros deben ser persistentes. Con base en la POO, el código que define la persistencia está disperso y parcialmente duplicado en las clases `Libro` y `Cliente`. En consecuencia, es difícil modificar o reutilizar este código aún cuando la política de persistencia del sistema es considerada (en términos de diseño) como una propiedad única.

3.3. Enfoques de implementación para la POA

Los principales problemas de la implementación de la POA son la representación y la composición de los aspectos. A continuación, se exponen los diferentes enfoques que intentan resolver estos problemas. Primero, se introduce el criterio de clasificación, propuesto por Bouraqadi y Ledoux [Bouraqadi and Ledoux, 2001], de los diferentes enfoques estudiados. Después, se describen y ejemplifican los diferentes enfoques que permiten implementar un sistema con base en la POA.

3.3.1. Criterio de clasificación

Para definir el criterio de clasificación de las diferentes técnicas de implementación de la POA, Bouraqadi y Ledoux parten del concepto de programa computacional [Bouraqadi and Ledoux, 2001]. Un programa es una secuencia de instrucciones destinadas a producir un resultado. La ejecución de este programa es efectuada por un sistema operativo o una máquina virtual que interpreta la secuencia de instrucciones.

Reconsiderando las definiciones del aspecto de base y de los aspectos no funcionales en este contexto, el aspecto de base describe los servicios de un sistema, mientras que los aspectos no funcionales intervienen en la manera de realizar estos servicios. En la ausencia de aspectos no funcionales, el aspecto de base puede ser considerado como un programa P_0 escrito para un interprete por defecto I_0. La ejecución de P_0 por I_0 produce un resultado dado R_0. Cuando los aspectos no funcionales intervienen en el procesamiento del programa R_0 se produce un resultado final R_1, diferente del resultado R_0. Este nuevo resultado R_1 puede ser obtenido ya sea transformando el programa P_0 o el interprete I_0. En efecto, puesto que todo resultado de un procesamiento computacional es inducido por el par $< Programa, Interprete >$, es posible modificar este resultado interviniendo en el programa, en el interprete o en ambos. Cada uno de estos tipos de intervención definen una categoría de implementación de la POA.

3.3.2. Enfoque por transformación de programa

Esta sección describe las técnicas de implementación que hacen variar únicamente el programa en el par $< Programa, Interprete >$. El objetivo es construir un nuevo programa P_1 a partir del aspecto de base (programa P_0) y de los aspectos no funcionales para producir un nuevo resultado R_1, sin cambiar el interprete por omisión I_0. Más precisamente, se trata de transformar el programa P_0 de con el fin de introducir los procesamientos que representan los aspectos no funcionales.

Descripción del enfoque

Este enfoque consiste en definir cada aspecto no funcional como un conjunto de reglas de transformación que serán aplicadas al aspecto de base. Por ejemplo, considere el aspecto no funcional que consiste en producir una traza en un archivo de historial para ciertas operaciones del aspecto de base. Este aspecto de historial puede ser representado por reglas de transformación que insertan el código para la producción de traza al principio de los métodos que representan las operaciones a rastrear. Este enfoque ha sido implementado en diversos trabajos [Chiba, 1995, Kiczales et al., 1997, Chiba and Tatsubori, 1998, Fradet and Südholt, 1998, Lieberherr et al., 1999].

Una de las dificultades de este enfoque reside en construir reglas de transformación genéricas, i.e., independientes del aspecto de base. Esta dificultad puede sortearse parametrizando las reglas de transformación (i.e., los aspectos no funcionales). En este contexto, las reglas de transformación pueden ser definidas en términos de puntos de unión abstractos, los cuales permiten referenciar, sin acoplamiento, a los elementos del aspecto de base. La configuración consiste en asociar los puntos de unión abstractos a los elementos concretos de la definición del aspecto de base (e.g., afectaciones y ciclos). Una vez que esta asociación ha sido establecida, la composición de los aspectos puede tener lugar, ya que las reglas de transformación pueden aplicarse al aspecto de base.

El proceso de transformación que representa la composición de los aspectos produce un nuevo programa en donde los diferentes aspectos están fusionados. En este programa monolítico, las líneas de código que representan los diferentes aspectos (de base y no funcionales) están enmarañadas. Este resultado es análogo al obtenido cuando se desarrolla un sistema mediante un lenguaje convencional de programación orientado a objetos. Asimismo, el enfoque por transformación de programa se asemeja a automatizar la mezcla de código que hubiera sido realizada manualmente por el desarrollador de aplicaciones.

Cuando un sistema conste de varios aspectos no funcionales, la composición se traduce en la aplicación de reglas de transformación que representan el conjunto de estos aspectos no funcionales. El orden de realización de las diferentes transformaciones es importante. De hecho, la aplicación de las diferentes reglas de transformación, siguiendo ordenamientos diferentes, no conduce generalmente al mismo resultado. Por ejemplo, la transformaciones que introducen la encriptación de los argumentos de ciertas llamadas de métodos (aspecto de criptografía) y la generación de trazas (aspecto de historial) no pueden ser aplicadas en un orden cualquiera. Para remediar esta exigencia semántica, ciertas implementaciones ofrecen

construcciones que permiten el ordenamiento de los aspectos no funcionales (i.e., de las reglas de transformación).

Ejemplo del enfoque por transformación de programa: el lenguaje AspectJ

La utilización de AspectJ para implementar el aspecto de sincronización de la clase `Libro` (cf. Sección 3.2.1) desemboca en la definición de tres módulos (ver figuras 3.6 y 3.7). El primer módulo corresponde al aspecto de base que implementa la definición de la clase `Libro`. El segundo módulo concierne la definición del aspecto de sincronización (`Sincronizar`). Finalmente, el tercer módulo corresponde a la definición de la configuración (`LibroSincronizar`).

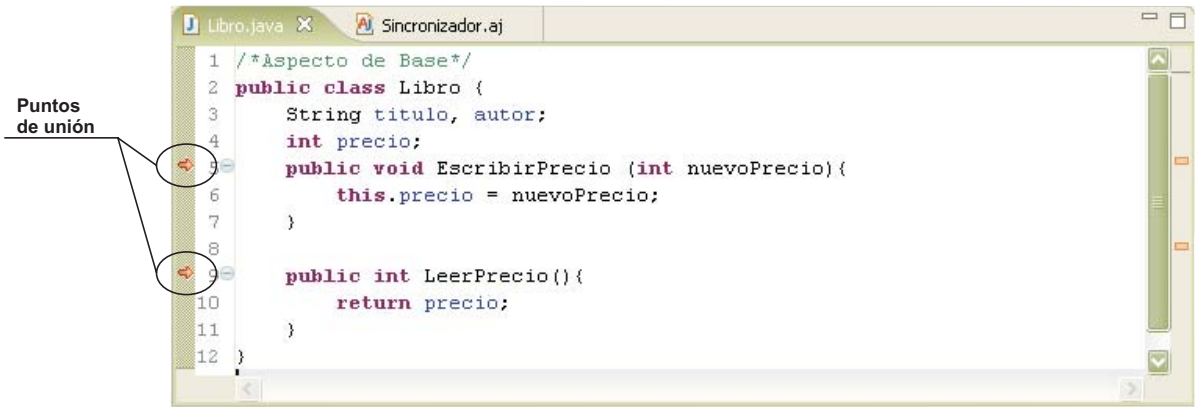


Figura 3.6: Aspecto de base

La palabra clave `aspect` de AspectJ introduce una construcción que representa un aspecto no funcional y eventualmente la configuración asociada. Esta construcción se asemeja a una definición de clase extendida con las definiciones de los puntos de unión utilizados por el aspecto, así como las transformaciones realizadas por el aspecto. Los puntos de unión son reagrupados en conjuntos identificados por un nombre e introducidos por la palabra clave `pointcut`. Estos conjuntos permiten encontrar las líneas de código a transformar. En el ejemplo, las líneas 28 a 30 definen el conjunto `sincronizarCorte` que reagrupa los puntos de unión correspondientes a la ejecución de los métodos `LeerPrecio()` y `EscribirPrecio()` de la clase `Libro`.

Las reglas de transformación son introducidas por medio de palabras clave, e.g., `before` y `after`. La palabra clave utilizada designa el tipo de transformación a realizar. Esta palabra clave es seguida del identificador del conjunto de los puntos de unión que deben ser afectados por la transformación. Las líneas 6 y 11 de la Figura 3.7 representan dos reglas de transformación que afectan el conjunto de los puntos de unión `sincronizarCorte`. Como los puntos de unión de este conjunto corresponden a recepciones de invocaciones a métodos, las transformaciones introducen el código de sincronización al principio (palabra clave `before`) y al final (palabra clave `after`) de los cuerpos de estos métodos.

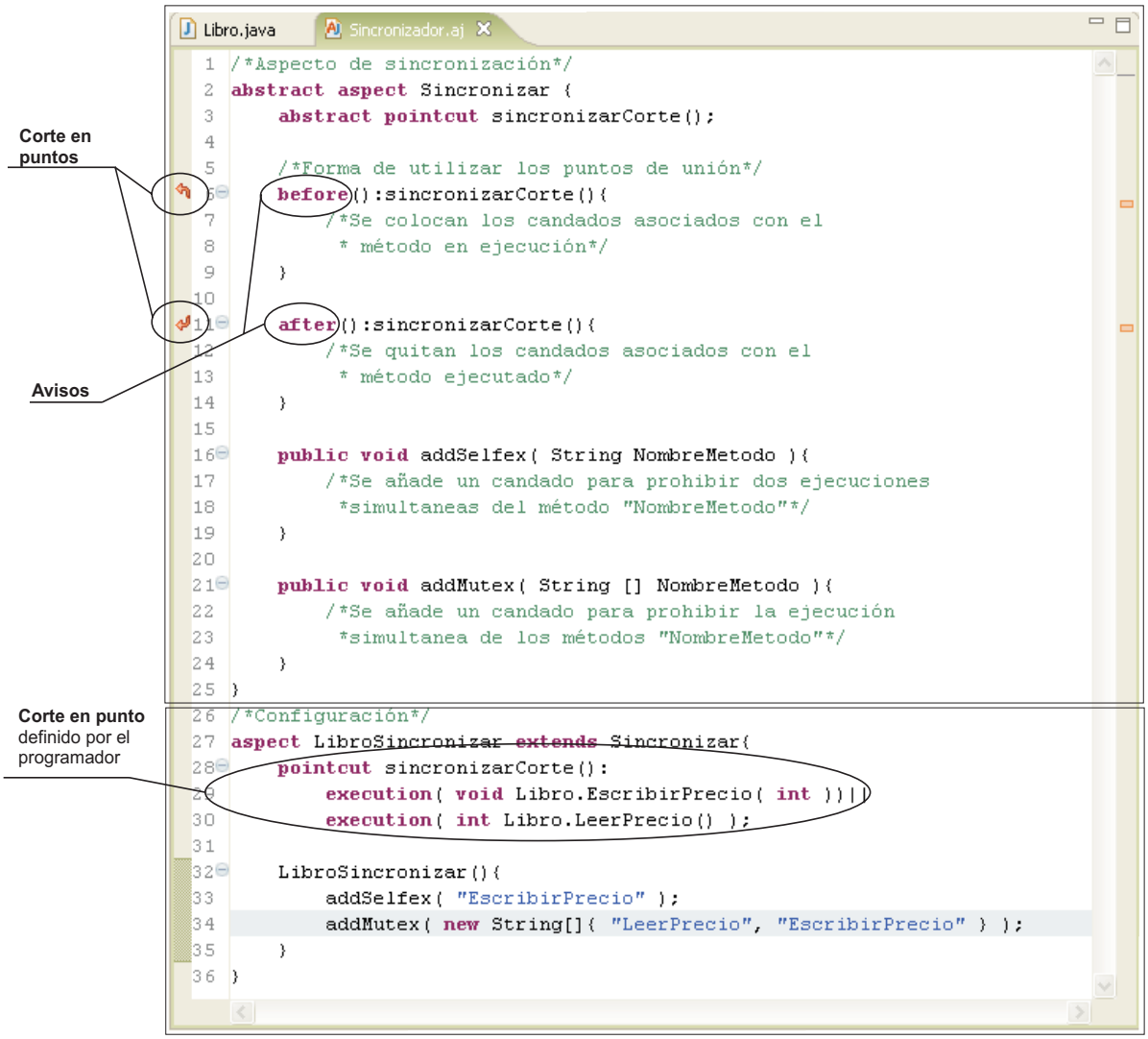


Figura 3.7: Aspecto de sincronización y módulo de configuración

La construcción `aspect` soporta la herencia gracias a la palabra clave `extends` de Java. Este concepto de herencia es utilizado en el ejemplo de la Figura 3.7 para favorecer la reutilización. Un aspecto de sincronización desacoplado de cualquier sistema es definido en el módulo `Sincronizar`. De hecho, este módulo define las reglas de transformación a realizar, además de las variables y los métodos necesarios para asegurar la sincronización. La definición del conjunto de puntos de unión en el corte `sincronizarCorte` (calificado como abstracto) debe ser realizada en los aspectos que heredan de `Sincronizar`. Esta definición se lleva a cabo en `LibroSincronizar` que está acoplada a la clase `Libro` y juega el rol de configuración. Además, el constructor `LibroSincronizar` (líneas 32 a 35) utiliza los métodos definidos en `Sincronizar` para fijar las reglas de sincronización (métodos auto-exclusivos o

mutuamente exclusivos).

3.3.3. Enfoque por transformación de interprete

Esta sección describe las técnicas de implementación que hacen variar únicamente el interprete en el par $< Programa, Interprete >$. El punto de partida es el aspecto de base que representa un programa P_0 cuya interpretación por el interprete por defecto I_0 produce el resultado R_0. El objetivo de este enfoque es construir un nuevo interprete I_1 a partir de los aspectos no funcionales y del interprete por defecto I_0 para producir un nuevo resultado R_1, sin cambiar el programa P_0. Más precisamente, consiste en modificar la semántica de interpretación del programa P_0 para tomar en cuenta los aspectos no funcionales.

Descripción del enfoque

Un interprete es una entidad que generalmente es compleja y difícil de entender y modificar. De hecho, seleccionar los mecanismos internos pertinentes y transformarlos, con el fin de introducir los aspectos no funcionales, no son tareas triviales. Por lo tanto, es necesario un marco que guíe y facilite las transformaciones. En general, los diferentes representantes del enfoque por transformación de interprete, e.g., [Watanabe and Yonezawa, 1988, Kiczales and Pæpcke, 1994, Stroud and Wu, 1996, Dempsey and Cahill, 1997, Lunau, 1998, Pryor and Bastán, 1999], adoptan el mismo marco: la **reflexividad**.

Según Smith, la reflexividad es la capacidad de una entidad de auto-representarse y de manipularse a sí misma, de la misma forma que representa y manipula su dominio de aplicación [Smith, 1990]. La reflexividad caracteriza la propiedad de un sistema que es capaz de razonar y de actuar sobre sí mismo. Así, un sistema reflexivo puede controlar su propio comportamiento y adaptarlo en función de la evolución de los requerimientos o del dominio de aplicación del sistema.

La reflexividad permite separar naturalmente el aspecto de base de los aspectos no funcionales. De hecho, un sistema desarrollado con la ayuda de un lenguaje reflexivo comprende dos niveles[1]: un nivel de base y un nivel meta (ver figura 3.8). El nivel de base describe los servicios realizados por el sistema y representa el aspecto de base. El nivel meta describe la manera de interpretar el nivel de base y representa el conjunto de los aspectos no funcionales. La separación de los aspectos no funcionales (unos de otros) puede ser obtenida mediante la estructuración del nivel meta, de manera que cada clase describa un único aspecto no funcional.

El nivel meta está poblado de objetos particulares llamados "meta-objetos", los cuales pueden estar relacionados con objetos del nivel de base, por medio de una relación llamada "meta-relación". La actividad de cada objeto del nivel de base está regida por los meta-objetos a los que está asociado por medio de meta-relación, e.g., los métodos invocados sobre un objeto Φ son interpretados por los meta-objetos asociados al objeto Φ.

[1]Una aplicación reflexiva virtualmente puede comprender una infinidad de niveles [Smith, 1990]. Sin embargo, la presente explicación se limita a los dos primeros, ya que son los únicos niveles relevantes en este contexto

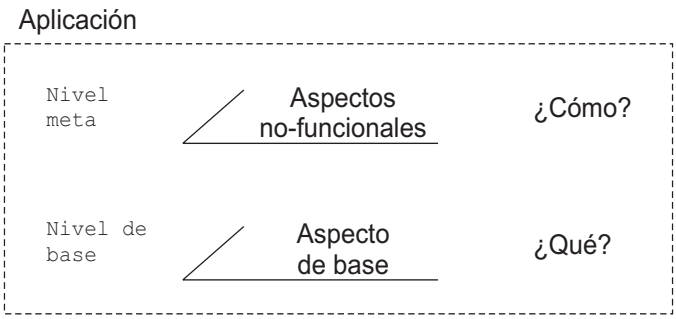


Figura 3.8: Correspondencia entre aspectos y niveles de un sistema reflexivo

La configuración de los aspectos no funcionales consiste en definir lo(s) meta-objeto(s) encargado(s) de controlar la ejecución de cada objeto del nivel de base. Esta definición está acompañada de la designación de los puntos de unión donde los meta-objetos deben intervenir. Fuera de los puntos de unión designados, la interpretación del nivel de base es idéntica a la realizada con el interprete por defecto.

Partiendo de una configuración dada, la composición de los aspectos se realiza en dos pasos. Primero, el interprete por defecto es extendido con las definiciones de los meta-objetos y, en consecuencia, con las definiciones de los aspectos no funcionales. Después, el aspecto de base es ensamblado con los aspectos no funcionales por medio de la meta-relación.

La dificultad de este enfoque consiste en componer los meta-objetos (representantes de los aspectos no funcionales) que intervienen en un mismo punto de unión. Esta composición se traduce ya sea por composición de meta-objetos (cooperación de meta-objetos [Mulet et al., 1995, Oliva and Butazo, 1998]) o por composición de clases de meta-objetos (principalmente mediante herencia [Bouraqadi, 1999a]).

Ejemplo del enfoque por transformación de interprete: el lenguaje MetaclassTalk

El lenguaje MetaclassTalk es una extensión reflexiva de SmallTalk en donde las clases juegan el rol de meta-objetos [Bouraqadi, 1999b]. Las clases son instancias de meta-clases (i.e., clases de clases) que definen métodos para controlar la estructura (e.g., reservación y acceso) y el comportamiento de las instancias (e.g., envío y recepción de mensajes). Las definiciones de las meta-clases describen ciertos mecanismos de ejecución de los objetos del nivel de base. Si solo se define un aspecto no funcional por meta-clase, MetaclassTalk separa claramente las definiciones de los diferentes aspectos no funcionales. La separación entre el aspecto de base y los aspectos no funcionales es automática, ya que éstos últimos son definidos en las meta-clases, mientras que el aspecto de base es definido en las clases del nivel de base.

Retomando el ejemplo de sincronización de la clase `Libro`, en el marco de la programación por meta-clases en MetaclassTalk, el aspecto de base corresponde a la definición de los servicios realizados por un libro, e.g., leer o modificar el precio. La definición de la clase `Libro` describe estos servicios independientemente de la manera de realizarlos (ver figura 3.9). El aspecto de sincronización aparece en forma de la definición de la meta-clase `Sincronizar` que

redefine los métodos de control de las lecturas/escrituras de las variables de instancia (respectivamente, "`atlV: ivName of: object`" y "`atlV: ivName of: object put: newValue`") con el fin de introducir la sincronización. Estos métodos son invocados implícitamente cuando se realizan los accesos a la estructura de los objetos del nivel de base.

"Aspecto de base"

```
1:  "La clase Libro es una instancia de la meta-clase MetaLibro"
2:  MetaLibro new
3:    intanceVariableName: "titulo autor precio";
4:    installAtName: #Libro;
5:    category: #'LibreriaElectronica-AspectoDeBase'.
6:
7:  EscribirPrecio:nuevoPrecio
8:    precio:= nuevoPrecio.
9:
10: LeerPrecio
11:  ^precio
```

"Aspecto de sincronización"

```
1:  MixEnMetaclase new
2:    installAtName: #Sincronizar
3:    category: #'LibreriaElectronica-AspectoDeSincronización'.
4:
5:  "Método de control de lectura a variables instanciadas"
6:  altV:ivName of:object put:newValue
7:  "prohíbe cualquier escritura a la variable ivName de la instancia object hasta el final de
8:   la lectura"
9:
10: "Método de control de escritura a variables de instancia"
11: altV:ivName of:object put:newValue
12: "prohíbe cualquier acceso a la variable ivName de la instancia object hasta el final de la
13:  escritura"
14:
15: sincronizarLecturaEscritura: ivName
16: "añade la variable ivName a la lista de variables a sincronizar"
```

"Modulo de configuración"

```
1: MetaLibro addSuperClass: Sincronizar.
2: Libro SincronizarLecturaEscritura: 'precio'.
```

Figura 3.9: Representación de los aspectos mediante meta-clases

Ya que la definición de la meta-clase `Sincronizar` es independiente de la definición de la clase `Libro`, existe una clara separación entre el aspecto no funcional de sincronización y el aspecto de base. Gracias a esta separación, es posible reutilizar un aspecto independientemente del otro. De manera general, se puede disponer de una biblioteca de meta-clases que podría ser utilizada en diferentes aplicaciones.

La configuración del aspecto de sincronización se realiza mediante las expresiones que aparecen en el módulo de configuración. La primera de ellas (ver la línea 1 del Aspecto de base en la figura 3.9) hace que `MetaLibro` (la meta-clase de `Libro`) se vuelva una subclase

de la meta-clase `Sincronizar`. Así, el comportamiento de la clase `Libro` es extendido, principalmente con el método `sincronizarLecturaEscritura:` (ver la línea 2 del Módulo de configuración en la figura 3.9). Este método toma como argumento el nombre de la variable de instancia `precio`, cuyos accesos representan puntos de unión utilizados por el aspecto de sincronización.

3.3.4. Enfoques híbridos

En las dos secciones precedentes, se describieron dos enfoques opuestos para la implementación de la POA. Entre estos dos extremos, existen enfoques híbridos que transforman tanto el programa como el interprete para introducir los aspectos no funcionales. De hecho, la transformación del programa o del interprete no son tareas fáciles. Dependiendo del lenguaje utilizado o de las herramientas disponibles, un tipo de transformación puede revelarse más adaptado que el otro para ciertos aspectos particulares. De aquí el interés de los enfoques híbridos.

Para ilustrar un enfoque híbrido, considere el ejemplo de dos aspectos no funcionales, optimización y sincronización, que son más fáciles de realizar con técnicas de implementación diferentes. En este caso, la transformación de programa es más adaptada para el aspecto de optimización, mientras que la transformación de interprete es más conveniente para el aspecto de sincronización.

El aspecto de optimización consiste en limitar el número de ciclos `for` de un programa Java. La Figura 3.10(a) muestra el ejemplo de una clase de vectores que define un método `sumar()` utilizando ciclos `for` para realizar la adición de vectores. Un programa que adiciona tres vectores requiere de dos invocaciones de este método. De esta manera, dos ciclos `for` son ejecutados. Como ya se mencionó, la optimización de este programa es más fácil de realizar por transformación de programa que consiste en fusionar los dos ciclos como se muestra en la Figura 3.10(b). Una solución que utilice la transformación de interprete es difícil de implementar ya que el interprete debe analizar el programa antes de ejecutarlo.

El aspecto de sincronización consiste en controlar los accesos a los campos `public` de una clase Java. Por su carácter público, estos campos pueden ser accedidos directamente en las diferentes clases. Por esta razón, la sincronización por transformación de programa es difícil de realizar, pues necesita un análisis pesado de tipo para encontrar el conjunto de los accesos. En comparación, el aspecto de sincronización es fácil de implementar por transformación del interprete, puesto que todos los accesos a los campos de un objeto son realizados por el interprete.

Javassist [Chiba, 2000] es uno de los raros representantes del enfoque híbrido. De hecho, ofrece una biblioteca de herramientas de transformación de programa y capacidades reflexivas que permiten transformar el interprete. Según lo que se quiera privilegiar, la eficiencia o la flexibilidad, es posible elegir entre aplicar transformación de programa o de interprete. Los aspectos no funcionales en donde la eficiencia es primordial serán introducidos por transformación de programa, mientras que los aspectos en donde la flexibilidad es más importante serán introducidos por transformación de interprete.

```
1:  public class Optimizacion{
2:    public Vector a,b,c,suma;
3:
4:    ...
5:
6:    suma = a.sumar(b.sumar(c));
7:
8:    ...
9:  }
10:
11: class Vector{
12:   Vector sumar( Vector otro){
13:       Vector resultante = new Vector( this.size() );
14:       for( int i = 0; i < this.size(); i++ ){
15:            resultante.colocarElemento(i,
16:            this.obtenerElemento(i) +
17:            otro.obtenerElemento(i));
18:       }
19:       return resultante;
20:   }
21: }
```

(a)

```
1:  public class Optimizacion{
2:    public Vector a,b,c,suma;
3:
4:    ...
5:
6:    suma = new Vector(a.size());
7:    for( int i = 0; i < this.size(); i++ ){
8:        suma.colocarElemento(i,
9:        a.obtenerElemento(i) +
10:       b.obtenerElemento(i) +
11:       c.obtenerElemento(i));
12:   }
13: }
14:
```

(b)

Figura 3.10: Aspecto de optimización

3.4. Lenguajes para el desarrollo de aspectos

En la actualidad existen dos formas de implementación de sistemas con el enfoque de aspectos: los lenguajes de propósito específico y los lenguajes de propósito general.

Los lenguajes de propósito específico manejan uno o más aspectos, pero no pueden manejar aspectos para los que no fueron diseñados. Expresan los conceptos del dominio especifíco en un nivel de abstracción más alto que el lenguaje de base. Para garantizar que los aspectos del dominio sean escritos de manera eficiente y evitar conflictos de control y de dependencia con el lenguaje base, el lenguaje de propósito especifico impone restricciones en la utilización del lenguaje de base. Un ejemplo es COOL (*COOrdination Aspect Language*), un lenguaje para el manejo de hilos creado por Xerox [Lopes and Kiczales, 1997], donde el lenguaje de base carece de los métodos `wait`, `notify`, `notifyAll` y de la palabra reservada `syncronized` para evitar que pueda expresar sincronización.

Los lenguajes de propósito general estan diseñados para describir cualquier tipo de aspecto, por lo que no pueden imponer restricciones al lenguaje de base. El nivel de abstracción del lenguaje de base y el de los aspectos es el mismo. Además, contienen el mismo conjunto de instrucciones, ya que debe ser posible programar cualquier tipo de aspecto. La principal desventaja de los lenguajes de propósito general es que no garantizan la separación de asuntos ("*separation of concerns*"). Sin embargo proveen un ambiente más adecuado para el desarrollo de aspectos, i.e., el programador puede pasar de un contexto de sincronización a uno de manejo de excepciones sin cambiar de herramienta de programación. Un ejemplo es AspectJ[2], en el que el lenguaje de base es Java y las instrucciones de los aspectos también

[2]ver ApéndiceA

se escriben en Java.

Una diferencia muy importante entre los lenguajes para aspectos de propósito especifico y de propósito general es que el primero fuerza la separación de asuntos, mientras que el segundo no lo garantiza del todo. Las ventajas y desventajas de utilizar uno u otro dependen de la aplicación que se quiere desarrollar. Mediante un lenguaje para aspectos de propósito especifico, el paso de una aplicación a otra significa comenzar todo desde el principio, con otros lenguajes para aspectos y con otras restricciones que quizas sean similares a otras aplicaciones ya desarrolladas. Por medio de un lenguaje para aspectos de propósito general, el programador puede pasar de un contexto a otro sin cambiar de lenguaje de desarrollo, lo que implica reducir costos de aprendizaje y desarrollo.

Capítulo 4

Propiedades no funcionales

El objetivo del presente capítulo es describir los diferentes mecanismos para implementar las propiedades no funcionales, las cuales permiten asegurar la disponibilidad de la información en entornos no confiables (e.g., Internet). Primero, se introducen los principios de las técnicas de replicación de datos (sección 4.1) y de control de concurrencia (sección 4.2). Estas técnicas permiten respectivamente mantener disponibles y coherentes los datos compartidos, los cuales pueden estar distribuidos en diferentes sitios. Enseguida, se describen y ejemplifican las arquitecturas de distribución más importantes en el diseño de sistemas cooperativos (sección 4.3). Cada arquitectura de distribución presenta diversas ventajas y desventajas, por lo que la selección de una arquitectura específica depende de los requerimientos del sistema. Finalmente, se analizan los principales modelos de comunicación (sección 4.4), así como las tecnologías más importantes para facilitar la transferencia de información en Internet (sección 4.5).

4.1. Replicación de datos

Coulouris et al. definen la **replicación de datos** [Coulouris et al., 1994] como el mantenimiento en línea de copias idénticas de los datos de un sistema con el fin de proveer buen desempeño, alta disponibilidad y tolerancia a fallas en entornos distribuidos (e.g., Internet). Así, la principal motivación de considerar la replicación de datos en sistemas distribuidos reside en estas tres características, por las siguientes razones:

1. **Buen desempeño del sistema**: si los datos son compartidos por un número relativamente grande de clientes, entonces un solo servidor no debería almacenar todos los datos. Este único servidor puede convertirse en un cuello de botella, haciendo más lentos sus tiempos de respuesta y minimizando su capacidad de rendimiento en términos de solicitudes procesadas por segundo. Una solución preferible consiste en replicar los datos compartidos en varios servidores, con el fin de que actúen como proveedores de una comunidad pequeña de clientes cercanos en la red.

2. **Alta disponibilidad de los datos**: si los datos compartidos están replicados en dos o más servidores que ejecutan software equivalente y que son alcanzables mediante enlaces de comunicación independientes, entonces los clientes podrán dirigir sus solicitudes a un servidor alterno cuando el servidor por defecto falle o sea inalcanzable.

3. **Tolerancia a fallas**: si los servidores que forman parte de una colección son capaces de atender en paralelo las solicitudes de los clientes, entonces es posible garantizar el correcto procesamiento de una solicitud, aún si uno (o más) de los servidores falla(n).

   La técnica de tolerancia a fallas es más confiable que la de alta disponibilidad, ya que puede garantizar la detección de fallas en tiempo real, tales como fallas arbitrarias, bizantinas y falla-paro (*fail-stop*). En particular, cuando se presenta este último tipo de fallas, el servidor detiene su proceso para cambiar a un estado en el cuál la falla puede ser fácilmente detectada [Coulouris et al., 1994].

La razón de operaciones de lectura/escritura realizadas en sistemas con datos replicados es una consideración importante en el diseño, especialmente cuando el número de réplicas es relativamente alto. El número de réplicas puede variar de cientos (e.g., usadas en uSENET o DNS) a algunas cuantas, que se utilizan principalmente en sistemas tolerantes a fallas.

4.1.1. Administración de réplicas

Un problema importante en sistemas distribuidos que soportan la replicación de datos es saber: 1) en qué lugar, en qué momento y para qué tipo de usuarios podrían ser almacenadas las réplicas y 2) qué mecanismos podrían ser utilizados para mantenerlas consistentes. El problema del almacenamiento de datos puede dividirse en dos subproblemas: 1) la ubicación de los servidores de réplicas y 2) la ubicación de las réplicas [Tanenbaum and van Steen, 2002]:

1. **Ubicación de los servidores de réplicas**: se refiere a encontrar el lugar más óptimo para ubicar un servidor de réplicas. Existen varias formas de calcular la mejor posición del servidor k, sin embargo el problema se reduce a seleccionar la mejor posición K entre N lugares diferentes (donde $K < N$). Este problema solamente puede ser resuelto mediante heurísticas. Por ejemplo Qiu et al., toman como punto de partida la distancia entre los clientes (medida en términos de la latencia y el ancho de banda de la red) y su ubicación [Qiu et al., 2001]. La solución propuesta por Qiu et al., selecciona un servidor a la vez, de tal manera que la distancia media entre el servidor y sus clientes sea mínima suponiendo que k servidores ya han sido ubicados (i.e., existen $N - k$ lugares). Radoslavov et al., proponen otra alternativa de solución que consiste en tomar solamente la topología de Internet como formador de sistemas autónomos[1] [Radoslavov et al., 2001].

2. **Ubicación de las réplicas**: se pueden distinguir tres tipos de réplicas de datos (ver figura 4.1):

[1] Un sistema autónomo puede ser visto como una red en la que todos los nodos ejecutan el mismo protocolo de enrutamiento, el cuál es manejado por una organización.

Figura 4.1: Organización lógica de los diferentes tipos de réplicas de datos

a. **Réplicas permanentes**: pueden ser consideradas como el conjunto inicial de réplicas de un almacén de datos distribuido. En muchos casos, el número de réplicas permanentes es pequeño. Sin embargo, este tipo de réplicas sigue siendo útil para garantizar la coherencia de los datos.

b. **Réplicas inicializadas en el servidor**: son copias de los datos creadas por iniciativa del propietario de los datos para aumentar los tiempos de respuesta. Suponiendo que todos los servidores de réplicas funcionan correctamente, la decisión de dónde colocar las réplicas es más sencilla que la decisión de dónde colocar los servidores. Las réplicas inicializadas en el servidor están ganando adeptos en el ámbito de los servicios de almacenamiento en la Web, donde son utilizadas como copias de sólo lectura colocadas cerca de los clientes.

c. Réplicas inicializadas en el cliente: también conocidas como cachés del cliente, este tipo de réplicas permite mejorar el tiempo de acceso a los datos. Cuando un cliente desea acceder a algún dato, el cliente se conectará al servidor más cercano para obtener una copia. Si se trata de una operación de modificación, el dato actualizado podrá ser re-enviado al servidor que lo proporcionó. Sin embargo, cuando múltiples operaciones involucran datos de sólo lectura, el tiempo de acceso puede mejorar dejando que el cliente almacene los datos requeridos en un caché cercano. El caché puede estar localizado en la máquina del cliente o en una máquina de la red local a la que pertenece el cliente, pero el caché solo puede ser conservado por un tiempo limitado. Así, la siguiente vez que el cliente solicite el mismo dato, podrá obtenerlo del caché local.

4.1.2. Actualización de réplicas

La replicación de datos garantiza buen desempeño, alta disponibilidad y tolerancia a fallas en sistemas distribuidos. Sin embargo, los datos contenidos en las réplicas tienen un tiempo limitado de validez, debido a que los usuarios podrían realizar modificaciones sobre dichos datos. A fin de colaborar efectivamente, los usuarios necesitan acceder en todo momento a información actualizada y coherente. Para satisfacer este requerimiento, un sistema cooperativo debe ofrecer un mecanismo que garantice la actualización oportuna de todas las réplicas de datos, siempre que se aplique una operación de modificación sobre una de ellas. Se han propuesto varios protocolos de actualización de réplicas: estados vs operaciones, *pull* vs *push* y *unicasting* vs *multicasting*.

Estados vs operaciones

Un problema importante de la actualización de datos replicados se refiere a definir lo que será propagado. Básicamente existen tres esquemas [Tanenbaum and van Steen, 2002]:

1. **Propagación de una invalidación**: consiste en notificar a otras réplicas que los datos que contienen ya no son válidos, debido a que una actualización tuvo lugar. Este esquema ha sido implementado por los llamados protocolos de invalidación, cuya ventaja principal es que utilizan un mínimo del ancho de banda de la red. La invalidación podría especificar qué parte de la réplica de datos ha sido actualizada, así solo esta parte será invalidada.

2. **Transferencia de los datos actualizados**: este esquema es útil cuando la razón entre las operaciones de escritura y lectura es relativamente alta. En este caso, la alta probabilidad de que una actualización tenga lugar reside en la condición de que los datos sean leídos antes de que la siguiente actualización se lleve a cabo.

3. **Difusión de la operación de actualización**: asume que cada réplica está representada por un proceso capaz de mantener "activamente" sus datos actualizados. La principal ventaja de este esquema es que las actualizaciones pueden ser propagadas con un costo mínimo de ancho de banda, a condición de que el tamaño de los parámetros asociados a la operación sea pequeño. Sin embargo, este esquema podría requerir más poder de procesamiento por cada réplica, especialmente cuando las operaciones de actualización son relativamente complejas. Este tipo de propagación es también conocido como replicación activa [Schneider 1990].

Enfoques *pull* vs *push*

Según el esquema *push*, el servidor envía por iniciativa propia las actualizaciones de los datos a cada cliente. Por el contrario, conforme al esquema *pull*, un cliente o un servidor solicita a otro servidor la transmisión de las actualizaciones que tenga almacenadas hasta el momento [Tanenbaum and van Steen, 2002].

Los mensajes que necesitan ser transmitidos entre un cliente y el servidor son diferentes en cada esquema de actualización. En el esquema *push*, la única comunicación que tiene lugar es cuando el servidor envía las actualizaciones a cada cliente. En el esquema *pull*, un cliente tiene que preguntar al servidor sobre la existencia de nuevas actualizaciones y, si es necesario, debe solicitar los datos actualizados.

El esquema *push* se utiliza en réplicas permanentes e inicializadas en el servidor. Este esquema también es conveniente cuando las réplicas requieren un alto grado de consistencia. Por el contrario, el esquema *pull* se emplea en réplicas inicializadas en el cliente. Este esquema es eficiente cuando el número de operaciones de lectura es relativamente bajo, debido a que las réplicas raramente son compartidas.

El principal inconveniente del esquema *pull*, en comparación con el esquema *push*, es el tiempo de respuesta puesto que tiende a incrementarse cuando los datos del caché ya no son válidos. Otro problema del esquema *pull* es que el servidor necesita mantener un seguimiento de todos los cachés de los clientes, lo que podría ocasionarle una sobrecarga.

Finalmente, el tiempo de respuesta en cada esquema es diferente. Cuando el servidor envía datos actualizados a los cachés de los clientes, el tiempo de respuesta en el cliente es prácticamente nulo. Sin embargo, cuando el servidor envía invalidaciones a los clientes, el tiempo de respuesta es el mismo que el del esquema *pull*, ya que está determinado por el tiempo que toma a cada cliente obtener los datos actualizados desde el servidor.

Unicasting vs *multicasting*

La transmisión *unicast* es el esquema utilizado en la mayoría de las comunicaciones vía Internet [Tanenbaum and van Steen, 2002]. Cuando un servidor, que forma parte del almacén de datos distribuido, requiere transmitir sus actualizaciones a N servidores diferentes, el servidor "transmisor" enviará N mensajes por separado, uno por cada servidor "receptor".

Por el contrario, la transmisión *multicast* ofrece una estrategia más eficiente que consiste en el envío simultáneo de mensajes a un grupo de servidores. De esta manera, el servidor "transmisor" envía un sólo mensaje a N servidores "receptores", i.e., entrega un mensaje sobre cada enlace de red y únicamente crea copias del mensaje cuando existe una bifurcación.

En muchos casos, la transmisión *multicast* resulta más económica. El peor caso sucede cuando todas las réplicas están localizadas en la misma red local, ya que el envío de un mensaje por *unicast* no es más caro que por *multicast*. La transmisión *multicast* puede ser más eficiente si se combina con el esquema *push*. Por el contrario, cuando se combina con el esquema *pull*, la transmisión *multicast* se comporta como en el caso de un simple cliente o servidor que solicita la actualización de sus réplicas.

4.2. Control de concurrencia

La necesidad de implementar mecanismos de control de concurrencia en sistemas cooperativos se presenta cuando el núcleo funcional del sistema, la interfaz de usuario y los datos compartidos están distribuidos en varios sitios [Greenberg and Marwood, 1994]. La latencia potencial en las redes de comunicación incrementa la probabilidad de que ocurran inconsistencias en los datos compartidos.

Cuando un sistema cooperativo carece de un mecanismo de control de concurrencia, los eventos pueden ser intercalados en los diferentes sitios participantes: primero se ejecutan localmente al momento de ser creados, después se trasmiten al sitio remoto y finalmente se ejecutan al momento de ser recibidos. En consecuencia, los colaboradores podrían confundirse a causa de las inconsistencias producidas por la ejecución no sincronizada de los eventos en los diferentes sitios implicados.

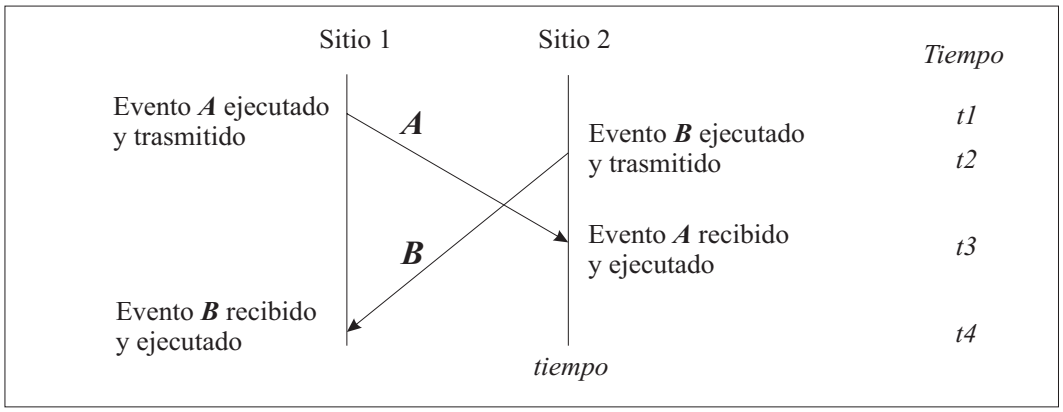


Figura 4.2: Diferente orden de ejecución de los eventos A y B en los sitios 1 y 2

Este problema de sincronización se ilustra en la figura 4.2. El sitio 1 crea, ejecuta y envía el evento A en el tiempo t_1. El sitio 2 realiza las mismas operaciones con el evento B en el tiempo t_2. El evento A es recibido y ejecutado por el sitio 2 en t_3; más tarde el evento B es recibido y ejecutado por el sitio 1 en t_4. En este punto, el sitio 1 ejecutó el evento A seguido del B, mientras que el sitio 2 ejecutó estos eventos en orden inverso. El resultado del diferente orden de ejecución conduce a inconsistencias en ambos sitios.

Los mecanismos tradicionales de control de concurrencia estudiados en sistemas distribuidos [Bernstein et al., 1987, Fujimoto, 1990] no pueden ser utilizados directamente en sistemas cooperativos, dado que estos últimos están destinados a soportar la interacción entre personas por medio de computadoras. En este ámbito, se han adoptado dos tipos de mecanismos que solucionan el problema de la ejecución no sincronizada de eventos: 1) la serialización y 2) los candados.

4.2.1. Mecanismo de serialización

Todos los algoritmos de control de concurrencia están basados en el criterio de **equivalencia serial** [Coulouris et al., 1994] que se deriva de las reglas de tratamiento para operaciones conflictivas de lectura y escritura (ver tabla 4.1).

Operaciones de diferentes transacciones		**Conflicto**	**Razón**
Lectura	Lectura	No	Porque el efecto de un par de operaciones de lectura no depende del orden en que fueron ejecutadas
Lectura	Escritura	Si	Porque el efecto de una operación de lectura y de una de escritura depende del orden en que fueron efectuadas
Escritura	Escritura	Si	Porque el efecto de un par de operaciones de escritura depende del orden en que fueron ejecutadas

Tabla 4.1: Reglas de tratamiento para operaciones conflictivas

Los algoritmos de serialización trabajan sincronizando eventos intercalados de forma que las transacciones atómicas (que pueden consistir de varios eventos) sean ejecutadas de manera serial en todo el sistema. Estos algoritmos también trabajan reparando los efectos de eventos desordenados para dar la impresión de que fueron ejecutados de manera serial. Un planificador se encarga de definir el orden de ejecución de los eventos o de detectar y reparar inconsistencias de orden [Bernstein et al., 1987]. El mecanismo de serialización ofrece dos niveles de optimismo:

1. **Serialización pesimista**: asegura que los eventos sean ejecutados en el orden correcto en todos los sitios, impidiendo la recepción de eventos en desorden. Una de las principales desventajas de la serialización pesimista es que la ejecución total de una secuencia podría ser muy lenta, ya que el planificador retarda la ejecución de cada evento hasta que sus predecesores lleguen.

2. **Serialización optimista**: se basa en dos suposiciones: 1) rara vez se reciben eventos conflictivos en desorden y 2) es más eficiente continuar con la ejecución de los eventos y después reparar los problemas, que garantizar el orden correcto en todo momento. Un esquema de reparación frecuentemente utilizado consiste en regresar el sistema al estado inmediato anterior a la ocurrencia del evento en desorden y enseguida ejecutar nuevamente los eventos en orden. El regreso a un estado anterior puede simularse mediante operaciones "deshacer/rehacer" [Prakash and Knister, 1992, Karsenty and Beaudouin-Lafon, 1993]. Otro esquema de reparación conocido se refiere a transformar, mediante un conjunto de reglas, un evento recibido en desor-

den de tal manera que su efecto sea el mismo como si hubiera llegado en orden [Ellis and Gibbs, 1989].

Una de las principales desventajas de la serialización optimista es que la implementación del almacenamiento del estado y de las operaciones "deshacer/rehacer" puede resultar costosa y compleja, debido a que no todos los eventos son susceptibles de deshacerse/rehacerse o transformarse (e.g., un evento resultante del mundo real).

Mientras que la serialización garantiza la integridad de los datos, también puede conducir a extraños comportamientos de interacción, e.g., los colaboradores pueden terminar peleando por el control de un objeto compartido. Sin embargo, este esquema es aceptable en algunas situaciones, particularmente cuando la serialización es instantánea. En algunas aplicaciones, como la de dibujo de mapas de bits (*bitmaps*), el grano de interacción es tan pequeño que los colaboradores podrían no percibir los efectos de la serialización.

Debido a que la serialización instantánea rara vez ocurre en un entorno real, el esquema pesimista interfiere con el flujo de la interacción, pues obliga a los colaboradores a esperar a que la sincronización ocurra antes de permitirles continuar. Por el contrario, el esquema optimista tiene repercusiones en la interfaz de usuario, debido a la dificultad de mostrar cómo reparar las ocasionales secuencias en desorden mediante operaciones de "deshacer/rehacer" o transformaciones.

4.2.2. Mecanismo de candados

Otro mecanismo para la administración de la concurrencia es la utilización de candados, un esquema que consiste en ganar acceso privilegiado a algunos datos por un periodo determinado. Los candados pueden ser empleados por un planificador para asegurar serialización global controlando el orden en que los sitios obtienen y liberan sus candados [Bernstein et al., 1987]. El mecanismo de candados ofrece tres niveles de optimismo (ver la tabla 4.2):

Nivel de optimismo	**Puede manipular el dato mientras espera su candado**	**Puede liberar el dato mientras espera su candado**
Pesimista	χ	χ
Optimista	$\surd$	$\surd$
Semi-optimista	$\surd$	χ

Tabla 4.2: Niveles de optimismo del mecanismo de candados

1. **Candados pesimistas**: fuerza a un sitio a esperar hasta que la solicitud de un candado sea resuelta antes de permitirle manipular el dato. Si la solicitud es bloqueante, el sitio hará la solicitud, esperará la respuesta y entonces manipulará el dato si y solo si recibió el candado. Si la solicitud es no bloqueante, el sitio podrá ejecutar otras acciones mientras la solicitud está siendo atendida. Una de las desventajas de este esquema son

los retardos perceptibles que vuelven poco natural la interfaz de usuario, interfiriendo con el flujo de la interacción. Además, la interfaz de usuario debería proporcionar información que muestre que el dato está en espera de un candado.

2. **Candados optimistas**: asume que los sitios conceden frecuentemente sus solicitudes de candados. Después de pedir un candado, el sitio solicitante adquiere una aprobación tentativa que le permitirá comenzar a manipular el dato antes de saber si realmente obtuvo el candado o no. Si el candado es otorgado, entonces el sitio podrá continuar con el procesamiento del dato. Así, este esquema evita los retardos en la interfaz de usuario, ya que no es necesario esperar a que el colaborador que ha puesto candado al dato lo libere.

   Cuando el sitio, al que le ha sido otorgado el candado tentativo, haya terminado de manipular el dato, el sitio podrá proponer una "liberación pendiente" y continuar con otras acciones (que podrían requerir más solicitudes de candados). Si el candado es negado, el sitio deberá recordar todas las operaciones realizadas sobre el dato para revertirlo a su estado original. La dificultad de este esquema se presenta cuando el sitio ha modificado otros datos que dependen del estado tentativo del primero. En este caso, se requiere un sistema de restauración de estado que deshaga todos efectos negativos en cascada de esta manipulación no autorizada.

3. **Candados semi-optimistas**: todos los efectos negativos precedentes son evitables mediante candados semi-optimistas. Según este esquema, un sitio está autorizado a manipular un dato con un candado tentativo, pero tiene prohibido manipular otros datos hasta que se le otorgue o se le niegue dicho candado. La dificultad de utilizar este esquema se presenta cuando otros colaboradores desean hacer uso de un dato que tiene un candado tentativo colocado por otro colaborador. Además, si la interfaz de usuario no refleja las modificaciones (tentativas y verdaderas) realizadas sobre el dato compartido, entonces esta situación podría ocasionar confusiones en el grupo de colaboradores.

El mecanismo de candados garantiza que solo un colaborador a la vez acceda a los datos compartidos. El tamaño de grano de un candado determina el nivel de acceso a un dato compartido, manejado por un simple candado. Desde la perspectiva del sistema, el grano grueso implica menos solicitudes de candados, pero también menos oportunidad para la concurrencia debido a que los candados podrían ser negados con más frecuencia.

Finalmente, mientras que el problema de la sincronización de eventos es inaceptable en sistemas distribuidos, en el ámbito de los sistemas cooperativos este problema podría ser soportable en algunas situaciones, ya que las personas pueden recurrir a protocolos sociales (e.g., toma de turno) para resolver los eventuales conflictos en la coordinación de sus acciones. En muchas aplicaciones cooperativas es raro que se produzcan inconsistencias pero cuando aparecen, los colaboradores tienen la capacidad de solucionar, por este medio, sus efectos negativos. El análisis de protocolos sociales está fuera del alcance de esta tesis.

4.2.3. Selección de mecanismos de control de concurrencia

La selección de un mecanismo de control de concurrencia es difícil. Una selección errónea puede conducir a un sistema inutilizable. Un mecanismo demasiado poderoso puede ser exagerado para la aplicación, sin contar el tiempo empleado en el desarrollo de esquemas innecesarios o rara vez utilizados. Greenberg y Marwood distinguen dos tipos de consideraciones [Greenberg and Marwood, 1994]:

Consideraciones humanas

Cuando los colaboradores tienen la expectativa de ganar el control sobre un área del espacio de trabajo compartido, el mecanismo más adecuado es el de los candados. Por el contrario, si los colaboradores desean trabajar muy cerca unos de los otros, entonces el mecanismo de serialización es suficiente.

La latencia de red tiene un mayor impacto sobre la forma en que los colaboradores perciben sus interacciones. Si el sistema puede otorgar/negar un candado antes de que el usuario esté listo para manipular el dato, entonces un esquema de candados no-optimistas resulta conveniente en este caso. Si se trata de una interacción más lenta, pero el sistema puede otorgar/negar el candado antes de que el dato sea liberado, entonces se puede optar por un esquema de candados semi-optimistas. Los candados optimistas pueden ser considerados cuando el tiempo de respuesta es bastante lento.

De manera similar, si la serialización ocurre de forma casi inmediata, entonces se puede optar por un esquema de serialización pesimista.

Consideraciones técnicas

Los mecanismos de serialización y de candados optimistas son considerados más complejos de implementar que sus homólogos pesimistas.

También se deben considerar los recursos computacionales ocupados por los diferentes esquemas. La complejidad computacional de algunos algoritmos de serialización optimista es bastante alta, particularmente aquellos que utilizan operaciones complejas de "deshacer/rehacer". Similarmente, el número de transacciones de red también debe tomarse en cuenta, puesto que algunos esquemas necesitan un gran intercambio de mensajes entre los sitios participantes, provocando sobrecarga y alta latencia de red.

4.3. Arquitecturas de distribución

En esta sección, se presenta una clasificación de las arquitecturas de distribución más relevantes para sistemas cooperativos. Esta clasificación, propuesta por Roth y Unger, se basa en un esquema de aplicación y en un esquema de distribución [Roth and Unger, 2000]. El esquema de aplicación define los componentes del sistema y la forma en que interactúan. En cambio, el esquema de distribución pone en evidencia las ventajas y desventajas de centralizar, distribuir o duplicar estos componentes en los sitios de los colaboradores y en servidores.

Los autores de esta clasificación no consideran a los datos compartidos como un componente del esquema de aplicación, sino que los incorporan a otros componentes. Ya que en el desarrollo de la agenda cooperativa es importante considerar la distribución de los datos compartidos, la clasificación de las arquitecturas de distribución ha sido extendida con los componentes "estado privado" y "estado compartido" [Phillips, 1999]. Asimismo, los componentes definidos en el esquema de aplicación fueron adaptados a la terminología empleada por el paradigma de la POA (cf. Capítulo 3) para la descomposición de un sistema.

4.3.1. Esquema de aplicación

El esquema de aplicación (ver figura 4.3) está integrado por cuatro componentes principales:

1. el **estado** se refiere a los datos compartidos y privados que son accedidos en lectura y modificación por los diferentes componentes del sistema;

2. el **sistema de ventanas** es utilizado principalmente en terminales "tontas"; simplemente despliega las ventanas de la aplicación y recibe los eventos del usuario desde los dispositivos físicos de entrada;

3. el **núcleo de la aplicación** está formado por el módulo de presentación y el aspecto de base:

   - el **módulo de presentación** transforma la entrada proveniente de los dispositivos físicos (o del sistema de ventanas) en eventos de la interfaz lógica y envía la vista interactiva a los dispositivos físicos de salida (o al sistema de ventanas);
   - el **aspecto de base** define el comportamiento básico del sistema, i.e., transforma los eventos de la interfaz lógica en actualizaciones sobre el estado compartido y calcula la vista interactiva a partir del estado compartido;

4. los **aspectos no funcionales** definen el comportamiento adicional del sistema, e.g., el control de acceso, el manejo de la concurrencia y el almacenamiento persistente del estado compartido.

El esquema de aplicación está representado por los siguientes elementos gráficos:

1. un **rectángulo con línea continua** representa un componente que realiza algún tipo de procesamiento, e.g., la actualización del estado compartido o de la interfaz de usuario;

2. una **elipse** representa tanto el estado privado como el estado compartido;

3. una **línea continua dirigida** de un componente a otro indica el flujo de datos.

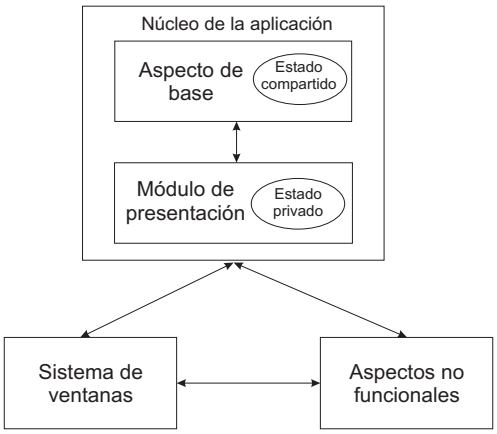


Figura 4.3: Esquema de aplicación

4.3.2. Esquema de distribución

El esquema de distribución está formado por un grupo de colaboradores, dispositivos, procesos concurrentes y estados centralizados, duplicados y/o distribuidos en los sitios de los colaboradores o en servidores.

El esquema de distribución está representado por los siguientes elementos gráficos:

1. un **rectángulo con línea punteada** representa un servidor central;
2. un **rectángulo con línea punteada asociado a un avatar** representa un sitio de colaborador o una terminal "tonta".

A continuación, se describen tres clases principales de arquitecturas de distribución para sistemas cooperativos:

1. Arquitecturas con componentes centralizados

Esta clase de arquitecturas tiene al menos un componente centralizado (i.e., núcleo de aplicación, estado compartido o aspectos no funcionales) sobre el cual trabaja un grupo de colaboradores. Según esta clase, los sitios de los colaboradores interactúan a través de un servidor central, ya que en ningún momento existe comunicación directa entre ellos. Tres subclases han sido identificadas:

1. **Núcleo de aplicación centralizado**: el servidor alberga la funcionalidad completa de la aplicación, i.e., una sola instancia del aspecto de base y una instancia del módulo de presentación por cada colaborador. Cada terminal mantiene únicamente una instancia del sistema de ventanas (ver figura 4.4). La principal desventaja de esta arquitectura reside en la sobrecarga de la red causada por la constante interacción entre el núcleo de aplicación y los múltiples sistemas de ventanas.

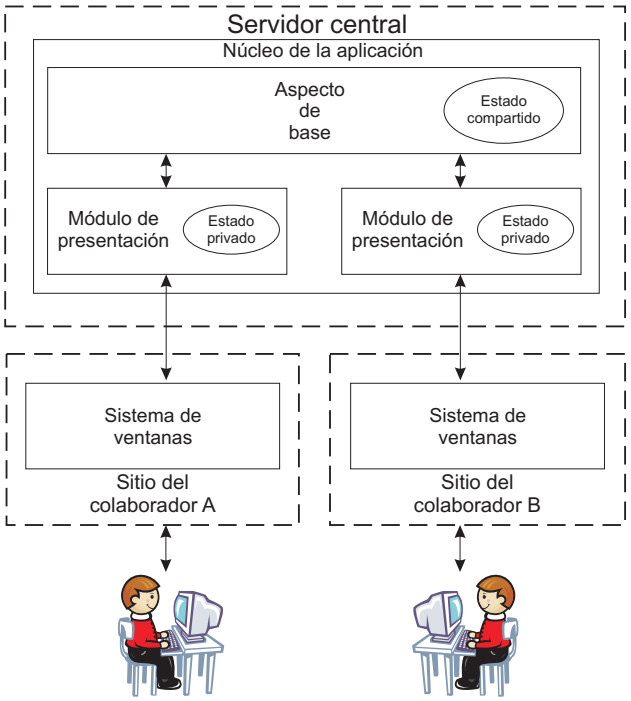


Figura 4.4: Arquitectura con núcleo de aplicación centralizado

2. **Estado compartido centralizado**: el aspecto de base reside en el servidor, mientras que el módulo de presentación está descentralizado en cada uno de los sitios participantes (ver figura 4.5). La principal ventaja de esta arquitectura es la disminución en la cantidad de mensajes transferidos ya que, a diferencia del sistema de ventanas, el módulo de presentación realiza una parte del procesamiento global del sistema. La principal desventaja es la falta de soporte para el manejo de información en tiempo real.

3. **Aspectos no funcionales centralizados**: en el servidor se localizan los aspectos no funcionales de la aplicación que aseguran: 1) el mantenimiento de la consistencia y 2) el ordenamiento de los eventos provenientes de cada uno de los sitios participantes donde se encuentra una instancia del núcleo de aplicación (ver figura 4.6). Una de las principales desventajas de esta arquitectura es la generación de cuellos de botella en el servidor.

En general, las arquitecturas centralizadas presentan una relativa simplicidad para mantener la consistencia del estado compartido mediante algoritmos centralizados [Phillips, 1999]. Sin embargo, una de sus limitaciones principales es que si el servidor falla, todos los colaboradores quedarían incomunicados.

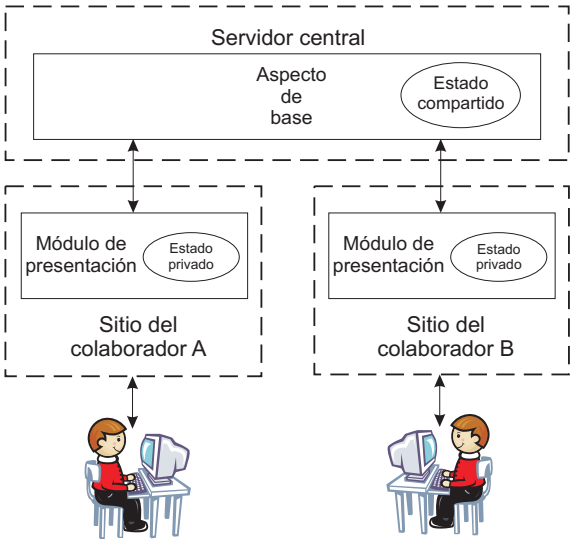


Figura 4.5: Arquitectura con estado compartido centralizado

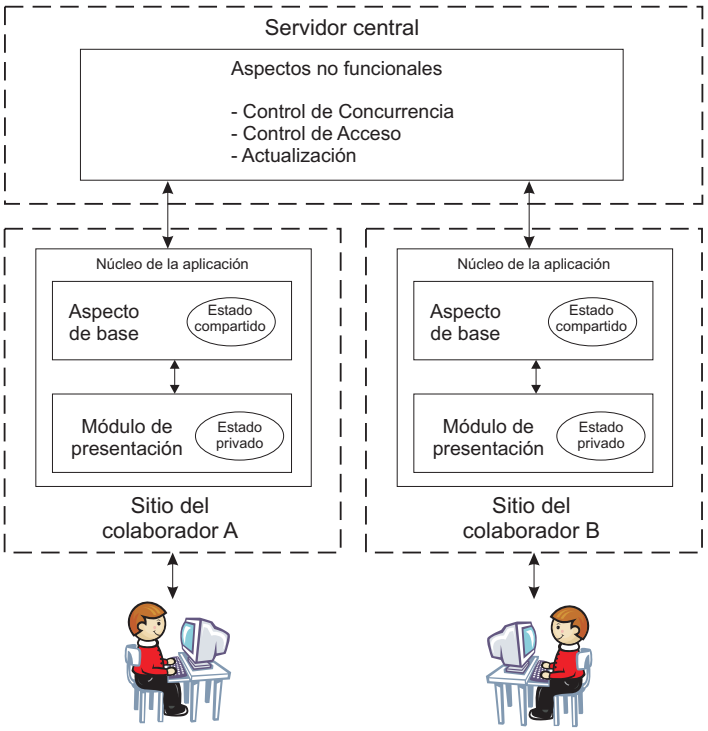


Figura 4.6: Arquitectura con aspectos no funcionales centralizados

2. Arquitecturas con comunicación directa (replicada)

Esta clase de arquitecturas no tiene un servidor central, sino que cada sitio de colaborador está conectado con todos los demás [Lukosch, 2003]. Se han identificado cuatro subclases:

1. **Estado semi-replicado**: el estado compartido está replicado en más de un sitio de almacenamiento, pero no necesariamente en todos (ver figura 4.7). Esta arquitectura incrementa la disponibilidad del estado compartido porque está almacenado en varios sitios predefinidos. Estos sitios también desempeñan la función de sitio de colaborador, por lo tanto albergan otros componentes, e.g., el módulo de presentación. Los demás sitios sólo contienen componentes que soportan la interacción de los colaboradores con el sistema, e.g., el sistema de ventanas o el módulo de presentación.

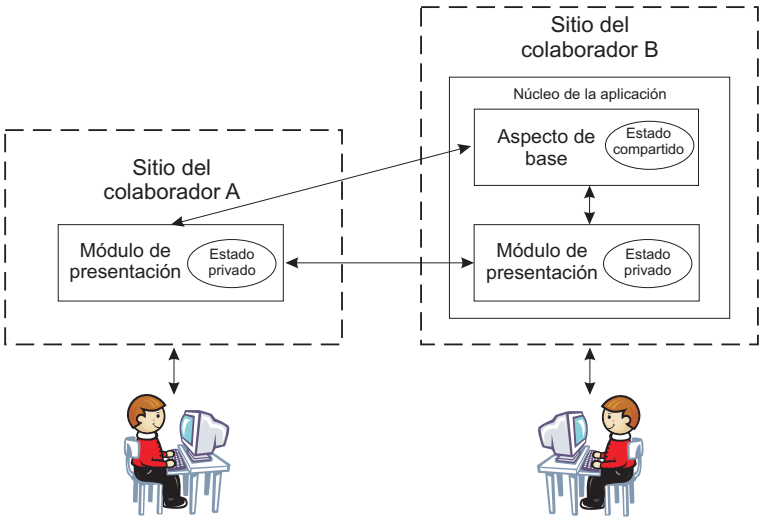


Figura 4.7: Arquitectura con estado semi-replicado

2. **Estado replicado**: el estado compartido está replicado en todos los sitios de los colaboradores, cuya comunicación se lleva a cabo mediante un enlace directo entre las instancias del aspecto de base (ver figura 4.8). Esta arquitectura mejora el tiempo de respuesta en la interfaz de usuario ya que las solicitudes son procesadas localmente. Para asegurar la consistencia de datos compartidos se requiere de un mecanismo de control de concurrencia, sin embargo la mayoría de los algoritmos utilizados son complejos. El tráfico en la red se incrementa debido a que cualquier cambio realizado por un colaborador sobre el estado compartido es propagado a los sitios de todos los colaboradores.

3. **Estado replicado con enlace directo entre módulos de presentación**: esta arquitectura es similar a la precedente. La diferencia principal reside en el enlace adicional de comunicación directa que existe entre las instancias del módulo de presentación (ver

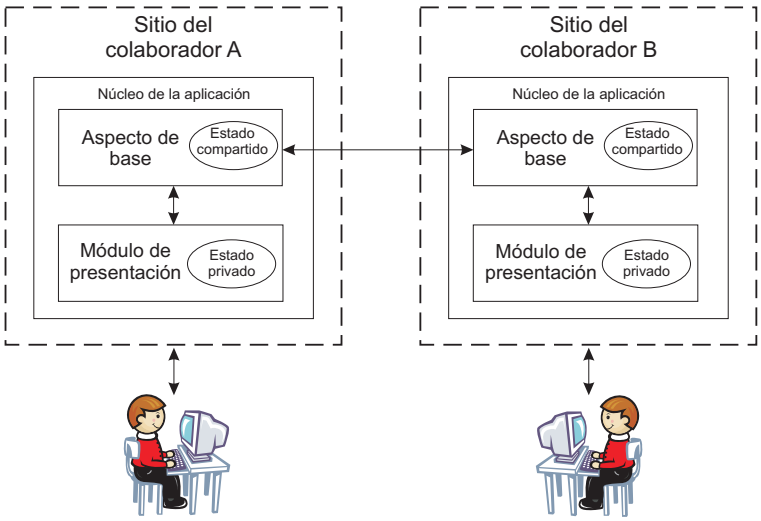


Figura 4.8: Arquitectura con estado replicado

figura 4.9). La principal desventaja de esta arquitectura es que se eleva el número de mensajes en la red, ya que existe información compartida, e.g., las coordenadas del apuntador del ratón, que deben ser actualizadas constantemente durante una sesión de trabajo.

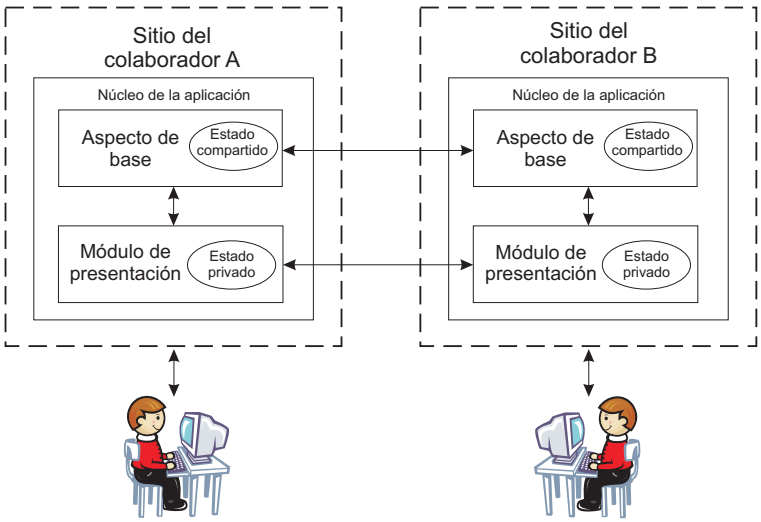


Figura 4.9: Arquitectura con estado replicado y módulos de presentación conectados

4. **Aspectos no funcionales descentralizados**: como en las dos arquitecturas precedentes, todos los componentes están replicados. Sin embargo, a diferencia de estas arquitecturas, el único enlace directo entre los sitios de los colaboradores se establece a

través de las instancias de los aspectos no funcionales (ver figura 4.10). Una de las ventajas de esta arquitectura es que evita los cuellos de botella causados por el servidor de la arquitectura centralizada. Sin embargo, los algoritmos empleados son más complejos debido a su carácter distribuido.

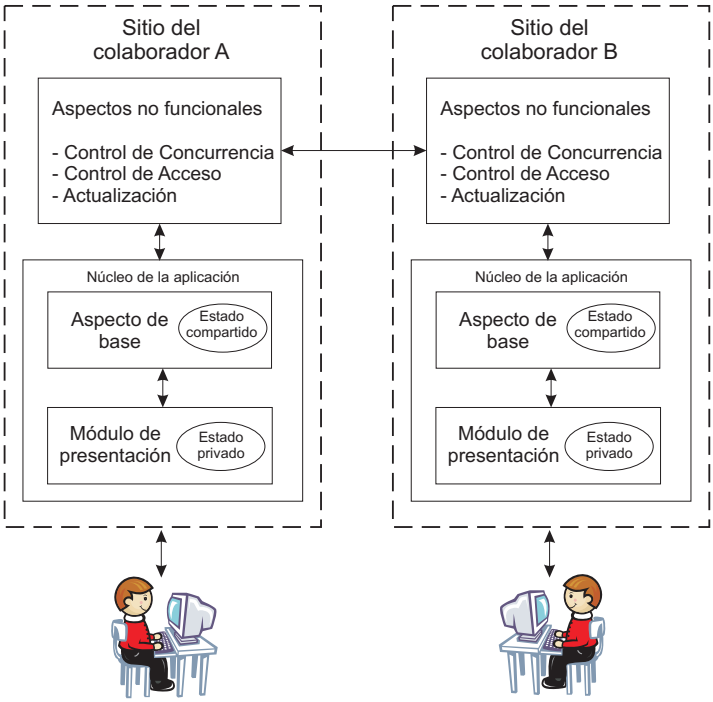


Figura 4.10: Arquitectura con aspectos no funcionales descentralizados

3. Arquitecturas híbridas

Estas arquitecturas se caracterizan por tener al menos un componente central, sin embargo los sitios de los colaboradores están comunicados entre sí directamente. En otras palabras, es una combinación de las arquitecturas de componentes centralizados y de las que tienen comunicación directa. Existen dos subclases principales:

1. **Híbrida con estado compartido centralizado**: es una combinación de las arquitecturas de estado centralizado y de estado replicado con enlace directo entre módulos de presentación. Más precisamente, el estado compartido está localizado en un servidor central, mientras que cada sitio de colaborador mantiene una instancia del módulo de presentación (ver figura 4.11). Los sitios de los colaboradores están conectados directamente mediante los módulos de presentación, los cuales a su vez están enlazados con el aspecto de base localizado en el servidor central.

2. **Híbrida con los aspectos no funcionales centralizados**: los aspectos no funcionales están ubicados en el servidor central, mientras que cada sitio de colaborador

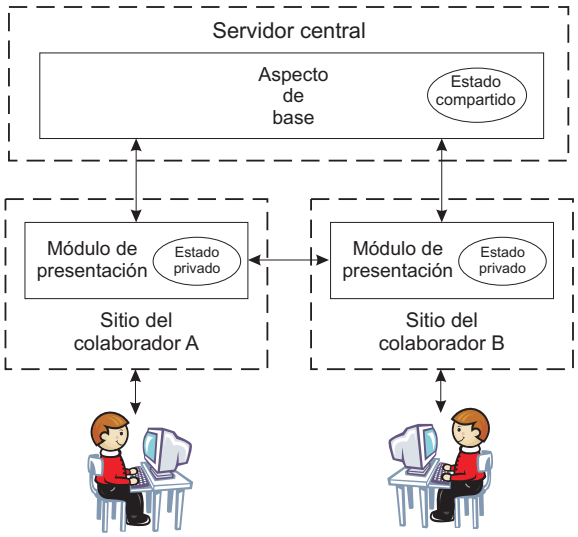


Figura 4.11: Arquitectura híbrida con estado compartido centralizado

mantiene una instancia del núcleo de la aplicación (ver figura 4.12). Los sitios de los colaboradores están enlazados a través de las instancias del núcleo de la aplicación, las cuales a su vez están conectadas a los aspectos no funcionales ubicados en el servidor central.

4.4. Modelos de comunicación de datos

En esta sección se analizan dos de los modelos de comunicación más importantes para diseñar sistemas cooperativos que requieren ser implantados en Internet. Estos modelos permiten a un grupo de colaboradores, posiblemente distribuidos en zonas geográficas diferentes, mantenerse en comunicación mediante el envío de mensajes y archivos multimedia.

4.4.1. Arquitectura cliente/servidor

El modelo cliente/servidor es una arquitectura modular basada en mensajes que pretende mejorar la usabilidad, flexibilidad, interoperabilidad y escalabilidad de las redes de computadoras. El cliente y el servidor actúan respectivamente como solicitante y proveedor de servicios [Schussel, 1996]. La interacción entre estos actores se lleva a cabo de la siguiente manera: el cliente envía una solicitud al servidor, el cual la procesa y transmite al cliente una respuesta en forma de mensaje. Así, esta arquitectura reduce el tráfico en la red cuando se envía la respuesta de una petición, puesto que no transfiere imperativamente un archivo como lo hacía su antecesora, la arquitectura de archivos compartidos [Sadoski, 1997].

Existen diferentes tipos de arquitecturas cliente/servidor: la arquitectura de dos niveles (*2-Tier*) y las diferentes configuraciones de tres niveles (*3-Tier*) [Schussel, 1996,

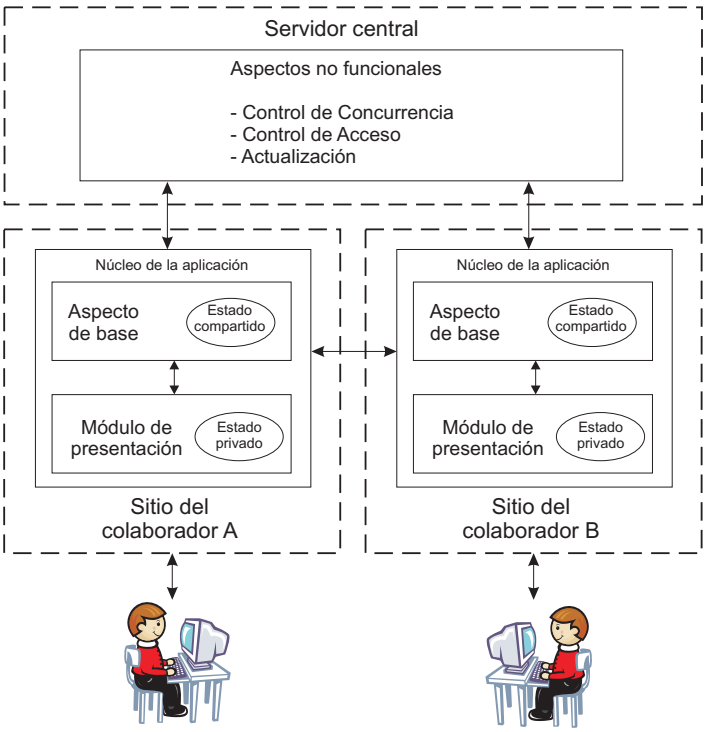


Figura 4.12: Arquitectura híbrida con los aspectos no funcionales centralizados

Edelstein, 1994].

Arquitectura de dos niveles (*2-Tier*)

Esta arquitectura se divide en dos módulos: 1) el sistema de usuario ubicado en el sitio cliente y 2) el servicio de base de datos localizado en el sitio servidor. Esta configuración es adecuada en sistemas distribuidos que soportan desde una docena hasta una centena de personas interactuando mediante una red LAN. Algunas de sus principales limitaciones son su bajo desempeño en redes WAN y su restringida flexibilidad en la administración de datos a causa del empleo de un manejador de base de datos propietario (e.g., SQL).

Arquitectura de tres niveles (*3-Tier*)

Esta arquitectura tiene el fin de cubrir las limitaciones de la arquitectura de dos niveles. Para lograr este objetivo incorpora un nivel intermedio entre el sistema de usuario y el servidor de base de datos. El nivel intermedio puede realizar diversas tareas, tales como encolamiento de peticiones, ejecución de aplicaciones, almacenamiento de la base de datos y planificación mediante prioridades del trabajo en proceso de ejecución.

La arquitectura de tres niveles puede soportar grupos de miles de personas y ofrecer mayor flexibilidad en comparación con la arquitectura de dos niveles. Una de sus principales

limitaciones depende en gran medida del rol que juega el nivel intermedio ya que, en caso de falla, los clientes quedarían incomunicados.

Existen diferentes tipos de configuraciones de la arquitectura *3-tier*:

1. **Arquitectura *3-Tier* con monitor para procesamiento de transacciones**: es la configuración básica del tipo *3-tier*, la cual consiste de un monitor para el Procesamiento de Transacciones (TP por sus siglas en inglés). El monitor TP sirve para encolar mensajes, planificar transacciones y dar prioridad a los servicios que solicitan los clientes. Cuando llega una transacción, el monitor TP la acepta, la encola y toma la responsabilidad de manejarla, así el cliente no debe esperar hasta que su petición sea atendida.

   La arquitectura de monitor TP tiene las siguientes ventajas:

   - actualiza múltiples servicios manejadores de bases de datos (DBMS por sus siglas en inglés) en una sóla transacción;
   - provee conectividad a varias fuentes de datos incluyendo archivos planos y DBMS relacionales;
   - otorga prioridades a las transacciones;
   - mantiene un alto grado de seguridad.

2. **Arquitectura *3-Tier* con servidor de mensajes**: el objetivo del servidor de mensajes es proveer conectividad a otros DBMS relacionales. Los mensajes son procesados de manera asíncrona con base a su nivel de prioridad. Un mensaje contiene información acerca de: sus lugares de procedencia y destino, su identificador y su prioridad. La diferencia entre la tecnología de monitor TP y el servidor de mensajes es que este último utiliza mensajes "inteligentes", los cuales contienen suficiente información para pasar por todos los servicios de la red.

   Para algunos tipos de mensajes, la capa intermedia de esta arquitectura puede servir como punto de enrutamiento entre dos servicios, mientras que para otros tipos de mensajes, la capa intermedia puede ejecutar un proceso de almacenamiento siguiendo las reglas del mensaje.

   El servidor de mensajes da robustez a la arquitectura mediante el almacenamiento y el reenvío lógico de mensajes. Además, la entrega de mensajes puede ser programada para soportar una falla del sistema, por lo que no se requiere de una conexión permanente entre el cliente y el servidor.

3. **Arquitectura *3-Tier* con servidor de aplicaciones**: el cuerpo principal de una aplicación no se ejecuta en el sitio del cliente sino en el servidor de aplicaciones. Más precisamente, el servidor de aplicaciones administra la lógica de negocios, el cómputo de operaciones y la recuperación de datos, mientras que los sitios cliente se encargan de manejar la interfaz de usuario.

El servidor de aplicaciones trabaja sobre un sistema operativo con tecnología *multicast* de 32 bits y con configuración de multiprocesamiento simétrico (SMP). Además, los servidores pueden configurarse para ejecutar en paralelo algunas aplicaciones.

Esta arquitectura tiene las siguientes ventajas:

- la seguridad aumenta a medida que disminuye el número de aplicaciones que se ejecutan en el cliente, debido a que el entorno de un servidor está más controlado;
- la aplicación resultante es más escalable;
- el costo/beneficio de instalación y soporte de una aplicación en un servidor es mayor que el que se obtiene al mantener la misma aplicación ejecutándose en cientos de clientes.

4. **Arquitectura *3-Tier* con Servicio Manejador de Base de Datos (DBMS)**: los datos están almacenados en forma normalizada para poder accederlos mediante diferentes DBMS (e.g., SQL y MySQL). Sin embargo, esta forma de almacenamiento puede ocasionar problemas en algunas aplicaciones.

   Por medio de un DBMS, una aplicación puede recuperar datos de un repositorio común para almacenarlos persistentemente mientras los esté utilizando.

5. **Arquitectura *3-Tier* con componentes distribuidos**: este tipo de arquitectura cambia la forma "tradicional" de construir aplicaciones cliente/servidor mediante la incorporación de componentes distribuidos tolerantes a fallas (e.g., replicación de componentes en múltiples servidores).

   Una arquitectura de componentes distribuidos ofrece las siguientes ventajas:

   - una misma aplicación puede ser utilizada en el cliente, ya sea de forma centralizada o completamente distribuida;
   - una aplicación puede ser desarrollada y probada localmente para conocer su comportamiento en entorno distribuido;
   - el desarrollador de este tipo de aplicaciones no tiene que preocuparse por resolver problemas tales como la gestión de colas y la coordinación de solicitudes, gracias al empleo de la tecnología ORB (*Object Request Broker*) para la invocación de servicios.

4.4.2. Arquitectura peer-to-peer

Una arquitectura par a par (P2P por sus siglas en inglés) no tiene clientes ni servidores fijos, sino que cuenta con un conjunto de nodos llamados **servents** que se comportan simultáneamente como clientes y servidores. Este tipo de red contrasta con el modelo cliente/servidor, donde rige una arquitectura monolítica carente de soporte para la distribución de tareas, i.e., generalmente todo el procesamiento es realizado en el servidor, mientras que el cliente no tiene ni siquiera la capacidad de efectuar operaciones simples, e.g.,

verificar que todos los campos de un formulario hayan sido proporcionados correctamente [Schoder and Fischbach, 2003].

El elemento fundamental de una red P2P es el par, el cual se define como la unidad de procesamiento básico, capaz de desarrollar algún trabajo útil y de comunicar los resultados a otra unidad de la red, ya sea directa o indirectamente. Los pares suelen tener una naturaleza dinámica y hetereogénea, i.e., se conectan a la red de forma intermitente y cada par cuenta con capacidades distintas [Tejedor and Jesús, 2006].

Las redes P2P son utilizadas para muchos propósitos, tales como la compartición de archivos de texto y multimedia, así como datos en tiempo real, e.g., el trafico de la red telefónica (VoIP). Una ventaja importante de estas redes es que todos los pares contribuyen a enriquecer la red con sus propios recursos, e.g., ancho banda, espacio de almacenamiento y poder computacional. La naturaleza distribuida de una red P2P permite incrementar su robustez en caso de fallas, e.g., mediante la replicación de datos en múltiples pares [Schoder and Fischbach, 2003].

Existen dos tipos de pares:

1. **Pares simples**: actúan como usuarios finales que tienen capacidades para proporcionar servicios y para utilizar los servicios ofrecidos por otros pares;

2. **Superpares**: ayudan a los pares simples a localizar otros pares y sus recursos. Los pares simples envían solicitudes de búsqueda de recursos a los superpares, los cuales les indican donde conseguirlos. Generalmente los superpares tiene una naturaleza estática (i.e., no se desconectan de la red) y son fácilmente alcanzables [Tejedor and Jesús, 2006].

Los pares pueden formar grupos que sirven a una causa común dictada por los demás miembros del grupo. Los grupos de pares pueden proporcionar a sus pares miembro servicios que no son accesibles por medio de otros pares de la red P2P. Algunos ejemplos de estos servicios son la transferencia de archivos, la provisión de información de estado del propio par o de otros a los que se encuentra conectado, la realización de un cálculo y el establecimiento de un enlace para comunicarse con otro par.

Los servicios pueden clasificarse en:

1. **Servicios de pares**: son funciones ofrecidas por un par de la red a otros pares de forma que si el par "proveedor" falla, los servicios estarán indisponibles;

2. **Servicios de grupo de pares**: son funciones proporcionadas por varios miembros del grupo de pares consiguiendo así un acceso redundante al servicio. Aunque un par del grupo falle, el servicio permanecerá disponible.

De acuerdo a su grado de centralización, las redes P2P se clasifican en: redes híbridas con componentes centralizados, puras o totalmente descentralizadas y mixtas o semicentralizadas [Androutsellis-Theotokis and Spinellis, 2004].

Redes P2P híbridas con componentes centralizados

Son redes P2P de la primera generación (ver figura 4.13). Napster y Publius [Waldman et. al., 2000] son ejemplos de redes P2P híbridas con componentes centralizados. Los pares interactúan con un servidor central que administra el ancho de banda, los enrutamientos y la comunicación entre los pares. El servidor central guarda los registros de conexión de cada par (e.g., IP, puerto y ancho de banda) y de los tipos de recursos compartidos que proporciona. Así, un par realiza una petición de recursos al servidor central, el cual la procesa y regresa la respuesta con la información necesaria para que el par solicitante se ponga en contacto con el par proveedor para descargar los recursos.

Cada par almacena sus propios recursos y los comparte con los demás pares de la red. Cuando un nuevo par desea unirse a la red, éste debe ponerse en contacto con el servidor central para reportarle el tipo de recursos compartidos que almacena.

Las principales ventajas de este tipo de red P2P son su relativa simplicidad de implementación y su eficiencia para localizar los recursos.

La principal desventaja es que este tipo de red se considera intrínsecamente no escalable, ya que está limitada por las capacidades del servidor central para almacenar los registros de los pares y dar respuesta a las solicitudes potencialmente concurrentes. También es vulnerable a fallas técnicas y ataques maliciosos.

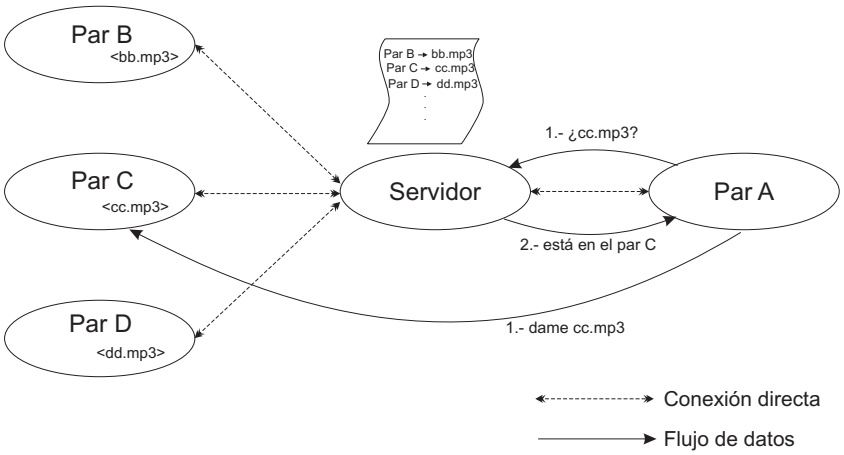


Figura 4.13: P2P centralizadas

Redes P2P puras o totalmente descentralizadas

Son redes P2P de la segunda generación, la cual es considerada como la más común (ver figura 4.14). Gnutella [Gnutella, 2003] es un ejemplo representativo de esta generación. Estas redes no requieren de ningún tipo de administración, lo cual permite eliminar la necesidad de utilizar un servidor central. Se opta por utilizar los mismos pares como almacenes de información y como nodos de conexión entre pares, i.e., la comunicación se establece de par

a par con la ayuda de un par intermediario. Además, cada par trata de mantener un cierto número de conexiones permanentes con otros pares.

En redes P2P puras, el tráfico está conformado esencialmente por peticiones, respuestas a estas peticiones y mensajes de control que facilitan el descubrimiento de otros pares. Las búsquedas de recursos compartidos se realizan de forma no-determinística debido a que no existe forma de saber donde se encuentra dichos recursos. Para minimizar la sobrecarga de mensajes, cada mensaje tiene un tiempo de vida (TTL por sus siglas en inglés) que se va decrementando en cada salto[2], de manera que cuando su valor es cero el mensaje se desecha.

Las redes de este tipo tienen las siguientes características:

- los pares actúan como cliente y servidor a la vez,
- no existe un servidor central que administre los recursos, por lo que se le considera una red robusta,
- no existe un enrutador central que sirva como enlace y administre las direcciones de otros pares.

Sus principales desventajas son: 1) el potencial incremento en el número de saltos entre los pares y 2) la sobrecarga del ancho de banda a causa de las búsquedas de información en la red. Además, es posible que el recurso solicitado no pueda ser encontrado a causa de dos razones principales: 1) que el TTL del mensaje de solicitud sea cero o 2) que el par que contiene dicho recurso no se encuentre conectado a la red al momento de realizar la solicitud.

Redes P2P mixtas o semicentralizadas

Son redes P2P de la tercera generación (ver figura 4.15). Kazza [Kazaa, 2003] es un ejemplo típico de esta generación. Estas redes son similares a las redes P2P con componentes centralizados, excepto que utilizan el concepto de **superpar**. Un superpar administra los recursos de ancho de banda, enrutamiento y comunicación entre los pares que están conectados a él, pero desconoce su identidad. El superpar puede cambiar dinámicamente a medida que nuevos pares se conectan. De hecho, ciertos pares de la red son seleccionados como superpares para ayudar a administrar el tráfico dirigido hacia otros pares.

Cada superpar mantiene un pequeño número de conexiones abiertas a los pares cliente. Asimismo, los superpares están conectados entre sí. Esta topología tiene la ventaja de hacer la red escalable gracias a la disminución de: 1) el número de nodos involucrados en el enrutamiento y manejo de mensajes y 2) la sobrecarga de tráfico hacia los nodos que están conectados a los superpares. Además, si uno de los superpares falla, la red puede continuar operando ya que los pares cliente conectados a él podrán establecer nuevas conexiones con otros superpares.

[2]Un salto se refiere a la conexión que se da entre dos pares cuando se realiza una solicitud.

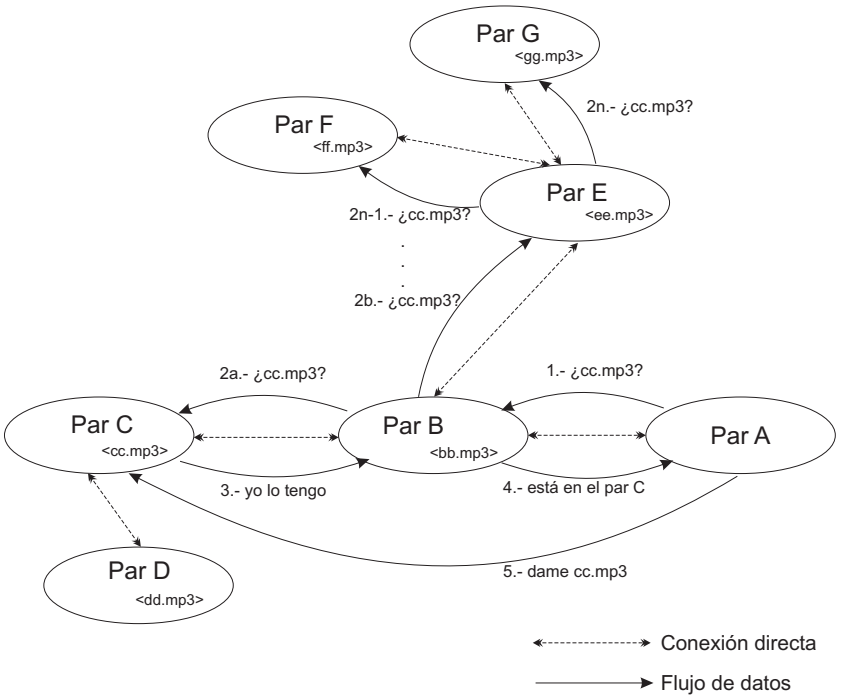


Figura 4.14: P2P pura sin componentes centralizados

Las redes P2P mixtas tienen dos ventajas adicionales:

1. el tiempo de búsqueda de un recurso disminuye en comparación con las redes P2P puras, debido a que la búsqueda se realiza únicamente en los superpares;

2. la mayor carga de red se produce en las conexiones entre los superpares, mientras que el tráfico entre los pares cliente y los superpares es relativamente ligero.

4.5. Tecnologías Web para la comunicación de datos

En las dos secciones precedentes, se han descrito las arquitecturas de distribución y los modelos de comunicación que permiten implantar sistemas cooperativos distribuidos en Internet. En esta sección, se exponen dos de las principales tecnologías para la comunicación de datos en la Web: CGI y *servlets*.

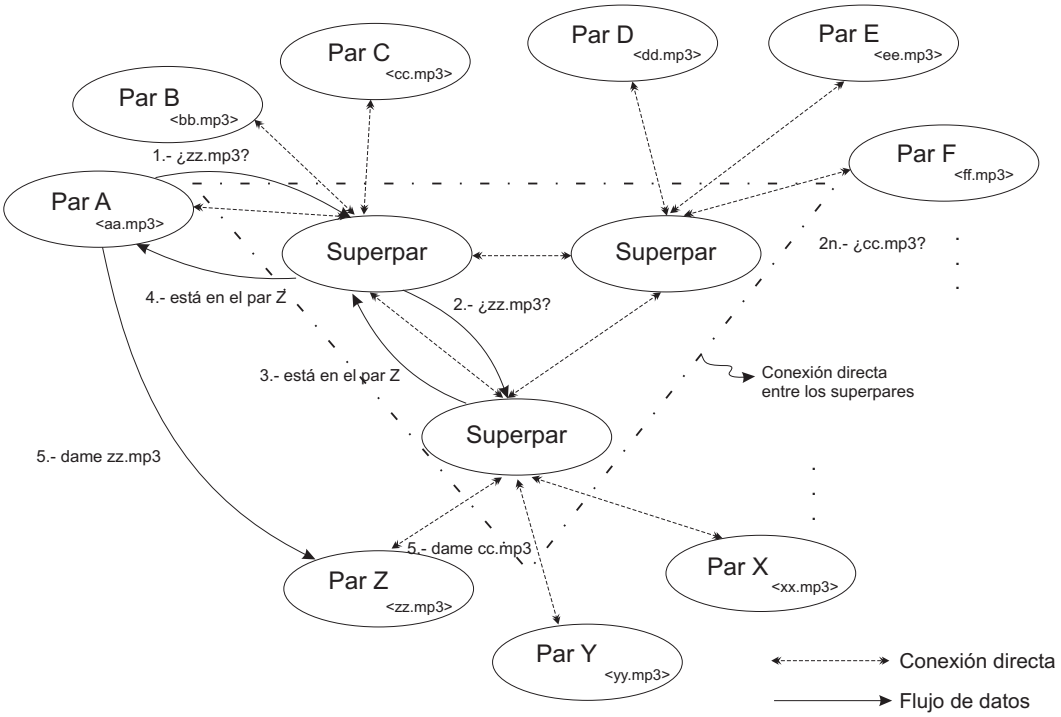


Figura 4.15: P2P mixtas

4.5.1. Tecnología CGI

La tecnología estándar CGI (*Common Gateway Interface*) permite crear una interfaz entre una aplicación externa y un servidor de información (e.g., un servidor Web). Esta tecnología permite a un cliente (e.g., un navegador Web) solicitar datos de un programa CGI que se ejecuta en el servidor Web. A su vez, éste último puede obtener información de otros programas antes de responder a las peticiones de los clientes.

La entrada de un programa CGI es obtenida de la entrada estándar (*input stream*) y de las variables de entorno[3] del sistema operativo donde se está ejecutando (ver figura 4.16). La salida (e.g., una página Web) de un programa CGI es enviada a la salida estándar (*output stream*). La tecnología CGI constituye una de las primeras maneras prácticas de añadir contenido dinámico a las páginas Web.

Un programa CGI se ejecuta en el servidor de la siguiente manera:

1. El servidor recibe una petición y comprueba si se trata de una invocación a un programa CGI. Los clientes realizan dichas peticiones mediante una URL (*Uniform Resource Locator*) que contiene el nombre del programa CGI sujeto a invocación.

2. El servidor prepara el entorno de ejecución del programa CGI.

[3]Las variables de entorno son un conjunto de valores dinámicos que normalmente afectan el comportamiento de los procesos en una computadora.

3. El servidor ejecuta el programa CGI.

4. El programa CGI genera un objeto MIME, el cual se envía a la salida estándar. El programa CGI debe aclarar qué tipo de objeto MIME creó para que el servidor pueda calcular su tamaño.

5. La salida estándar puede ser reenviada a un programa externo que se encuentre incluso en otro servidor. Dicho programa puede estar escrito en cualquier lenguaje que soporte el servidor, e.g., Perl o C. Por razones de portabilidad, a menudo se utilizan lenguajes de *script*[4] en vez de lenguajes compilados.

6. Finalmente, el servidor envía la información producida por el programa CGI de regreso al cliente, el cual se mantiene en estado de espera en lugar de realizar una descarga de archivos tradicional [CGI, 1998].

Una de las principales desventajas de esta tecnología es la latente saturación del servidor cuando varios clientes realizan solicitudes concurrentes al mismo programa CGI. De hecho, por cada solicitud, se crea una copia del programa CGI destinada a atenderla. Finalmente, esta tecnología se considera potencialmente peligrosa, ya que es posible que se produzcan errores que causen la falla completa del servidor.

4.5.2. Tecnología *servlets*

Los *servlets* son componentes del servidor que pueden ser ejecutados en cualquier plataforma, puesto que heredan las ventajas del lenguaje Java en términos de portabilidad ("*write once, run anywhere*"). Esta tecnología tiene como objetivo incrementar la funcionalidad de una aplicación Web. Un *servlet* se carga dinámicamente en el entorno de ejecución del servidor al momento de ser invocado, i.e., cuando se recibe una solicitud de un cliente, el contenedor del servidor Web inicia el *servlet* requerido. A continuación, el *servlet* procesa la solicitud del cliente y envía la respuesta al contenedor del servidor, el cual se encarga a su vez de transmitirla al cliente (ver figura 4.16) [Allamaraju et al., 2000].

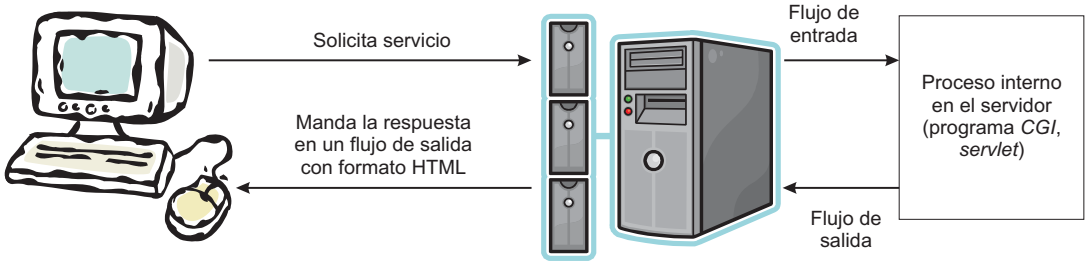


Figura 4.16: Modelo solicitud/respuesta en la Web

[4]Es un lenguaje diseñado para ser ejecutado por medio de un intérprete, e.g., Javascript, Perl, PHP, Pyton.

Cuando un proceso completo (también llamado hilo pesado) se inicia en un servidor, se requiere la asignación de varios recursos (e.g., procesador y memoria). El cambio de contexto del proceso actual al nuevo proceso conduce a la sobrecarga del servidor, ya que es necesario guardar el estado del proceso actual para continuar su ejecución más tarde. Por el contrario, por cada solicitud dirigida a un *servlet*, se crea un proceso ligero (también llamado hilo ligero) para atenderla. Un proceso ligero es un proceso hijo que es controlado por el proceso padre (en este caso, el servidor). El cambio de contexto se realiza fácilmente, ya que un proceso ligero puede pasar de activo a inactivo o a estado de espera [Vandana, 2004].

La API para la programación de *servlets* facilita la escritura de servicios complejos para aplicaciones Web, puesto que el desarrollador no tiene que centrarse en detalles referentes al protocolo HTTP, a los formatos de solicitud y a las cabeceras de mensajes. Además, pueden interactuar con diferentes bases de datos (e.g., Oracle o Servidores SQL) y con otros servidores Web.

Con el fin de aumentar el desempeño del servidor, los *servlets* permiten guardar en memoria caché algunos datos utilizados con frecuencia. Además, como cualquier programa Java, también manejan excepciones para hacer más robusta la implementación.

Características de los *servlets*

1. Son independientes del servidor y del sistema operativo, i.e., el servidor puede estar escrito en cualquier lenguaje de programación, obteniéndose exactamente el mismo resultado que si estuviera escrito en Java.

2. Pueden invocar a otros *servlets*, e incluso a métodos concretos de otros *servlets*. También pueden redireccionar solicitudes de servicios a otros *servlets* localizados en el mismo sitio o en un sitio remoto. De esta manera, el trabajo puede distribuirse en forma más eficiente, e.g., se podría tener un *servlet* encargado de la interacción con los clientes, mientras que otro *servlet* estaría destinado a manejar la comunicación con una base de datos.

3. Obtienen fácilmente datos acerca del cliente (pero sólo los datos permitidos por el protocolo HTTP) tales como dirección IP, puerto y método empleado (e.g., GET o POST).

4. Utilizan *cookies* y sesiones para guardar datos acerca de un usuario determinado en el sitio cliente, con el fin de personalizar la interacción cliente/servidor.

5. Sirven de enlace entre un cliente y una o varias bases de datos en arquitecturas cliente/servidor de 3 capas (*3-Tier*), aún si la base de datos está localizada en un servidor distinto.

6. Actúan como representantes (*proxies*) de un *applet*[5]. Por restricciones de seguridad, un *applet* no puede acceder directamente a un servidor de datos localizado en cualquier

[5]Es un programa Java diseñado para ejecutarse en una página Web a través de un *browser* que soporta Java. Los *applets* también soportan interfaces visuales.

sitio remoto, pero un *servlet* sí puede hacerlo sin necesidad de configurar políticas de acceso al servidor.

7. Al igual que los programas CGI, los *servlets* permiten la generación dinámica de código dentro de una página HTML.

4.5.3. Análisis comparativo

En esta sección se realiza un análisis comparativo entre las tecnologías CGI y *servlets* con el fin de poner en evidencia sus ventajas y desventajas en la construcción de sistemas distribuidos que comparten información a través de Internet.

Portabilidad

Aunque los programas CGI pueden ser soportados por una amplia variedad de servidores Web, no son portables entre plataformas. El programa CGI debe ser compilado en la plataforma donde va a ejecutarse.

Por el contrario, gracias a la tecnología Java, los *servlets* pueden ejecutarse en cualquier plataforma y pueden acceder a las clases de otras aplicaciones Java. Sun ofrece el código fuente de *servlets* y JSP (*Java Server Pages*) a las comunidades de desarrollo de las tecnologías Java y Apache, con el fin de asegurar la disponibilidad de *software* libre y el soporte de integración para aplicaciones Apache.

Desempeño

Generalmente, el servidor Web genera un nuevo proceso cada vez que un cliente invoca un programa CGI. Si éste último fue escrito en Perl, el servidor Web descargará y ejecutará el intérprete de Perl cada vez que se haga una petición. A medida que el número de solicitudes concurrentes crece, el rendimiento del servidor disminuye.

Los *servlets* están exentos de este problema, ya que se ejecutan una sola vez y permanecen en memoria para atender futuras solicitudes. Además, los *servlets* hacen más fácil la implementación de aplicaciones multi-hilo que los programas CGI.

Facilidad de desarrollo

Las aplicaciones Java no generan violaciones de acceso a la memoria que provoquen la falla del servidor, aunque son más lentas que las aplicaciones escritas en lenguajes Perl o C.

Por definición, HTTP es un protocolo sin estados. Esta característica no permite dar seguimiento del estado de una sesión en algunas aplicaciones Web. Para solucionar este problema, los programas CGI deben utilizar *cookies* o reescribir URLs, mientras que los *servlets* definen una interfaz de sesión dedicada.

Seguridad en el desarrollo

Los lenguajes para desarrollar programas CGI se caracterizan por ser flexibles y potentes. Los programadores con experiencia pueden utilizar su estructura abierta para desarrollar soluciones creativas. Sin embargo, estas características de flexibilidad y potencia en manos menos experimentadas podrían dar respuestas no deseadas.

En Perl, cuando un tipo de dato se pasa como argumento de manera errónea, se generan problemas que sólo pueden ser detectados durante la ejecución. Estos problemas no suceden en Java, ya que los errores de asignación tanto en tipos de datos como en objetos se detectan durante la compilación. El resultado es un ambiente de desarrollo más previsible.

Despliegue

Los procesos de instalación, actualización y despliegue de un programa CGI se realizan de forma manual. En consecuencia, estos procesos no sólo consumen tiempo, sino que también requieren de un gran número de tareas de administración en el servidor Web.

La interfaz *Java Servlet* provee un mecanismo de despliegue portable que facilita la implantación de aplicaciones. Mediante este mecanismo, se crea un archivo con extensión (*.war*) donde se empaqueta un proyecto que después será instalado en servidores Web compatibles con esta tecnología. De esta manera, se evita el proceso de instalación en servidores con características específicas.

Mantenimiento

Las aplicaciones Java son más fáciles de mantener que los programas CGI, gracias a la propiedad de reutilización de código. Para muchos desarrolladores, resulta más fácil entender la mecánica de un programa en Java, escrito por otra persona, que leer un programa en Perl o C. Además, Java genera menos errores en el código que muchos programas escritos en Perl o C, debido a la propiedad de encapsulación.

Seguridad de red

La Máquina Virtual de Java (JVM por sus siglas en inglés) restringe el acceso al entorno de ejecución del servidor Web. Sin embargo, en el caso de Perl, el administrador debe utilizar controles de bajo nivel (i.e., del sistema operativo) para restringir el acceso a las aplicaciones que se ejecutan en el servidor Web.

[7]Un entorno IDE (*Integrated Development Environment*) es un sistema que contiene un conjunto de herramientas útiles para un programador, tales como: 1) un editor de código, 2) un compilador, 3) un *debugger* y 4) una herramienta para construir interfaces gráficas (GUI, *Graphic User Interface*).

Características	Programa CGI	*Servlets*
Portabiblidad entre servidores Web	√	√
Portabiblidad entre plataformas	χ	√
Ejecución de múltiples sesiones sin generar procesos separados para cada una de ellas	χ	√
Protección contra el uso de memoria	√	√
Lenguaje de programación	C y Perl	Java
Seguridad en el desarrollo	Libre No recomendable para desarrolladores con poca experiencia	Fuerte Todos los valores son tratados por separado
Seguimiento de estado entre solicitudes	χ	√
Ejecución de código de forma segura	χ	√
Tiempo de desarrollo	Difícil, engorroso	Sencillo, principalmente utilizando un IDE[7]
Mantenimiento	Difícil	Sencillo
Despliegue	Engorroso	Sencillo, utilizando archivos *.war*
Seguridad de red	√	√

Tabla 4.3: Comparación entre programas CGI y *servlets* [Java Servlets, 1999]

Capítulo 5

Diseño de la agenda cooperativa

El objetivo del presente capítulo es describir el diseño de la agenda cooperativa propuesta en este trabajo de tesis. Particularmente, el diseño de este sistema hace énfasis en la provisión de dos propiedades: la flexibilidad de sus módulos constituyentes (cf. Capítulo 3) y la disponibilidad de los datos compartidos (cf. Capítulo 4). La propiedad de flexibilidad se refiere a la capacidad del sistema: 1) para poder ser "fácilmente" adaptado a las necesidades de los diversos grupos y 2) para permitir que otras aplicaciones con requerimientos similares reutilicen eficientemente algunos de sus módulos. Por otra parte, la propiedad de disponibilidad de los datos compartidos se refiere a la capacidad del sistema para garantizar que los colaboradores puedan acceder en cualquier momento a datos coherentes y actualizados, aunque el sistema esté implantado en un entorno susceptible de sufrir latencias y fallas potenciales de redes y servidores.

En este capítulo, primeramente se detalla la arquitectura del sistema en términos de los esquemas de aplicación y de distribución descritos en el Capítulo 4 precedente (cf. sección 5.1). A continuación, se describe el diagrama de clases de la agenda cooperativa, haciendo una clara separación entre las clases del aspecto de base y las clases de los aspectos no funcionales (cf. sección 5.2). Finalmente se explican, mediante diagramas de secuencia, las principales funcionalidades de los diferentes aspectos del sistema, así como algunas de sus interacciones más importantes (cf. sección 5.3).

5.1. Arquitectura del sistema

En esta sección se describen, mediante un esquema de aplicación, los principales componentes (tanto procesos como estados) de la agenda cooperativa. A continuación se precisa, mediante dos esquemas de distribución separados, el carácter centralizado o descentralizado de cada componente. Esta separación tiene el objetivo de detallar no sólo la distribución de los componentes de procesamiento, sino también la distribución de los datos manipulados por dichos procesos.

5.1.1. Esquema de aplicación

De acuerdo con la sección 4.3.1, el esquema de aplicación de la agenda cooperativa define los componentes que la constituyen y la forma de interacción entre ellos (ver figura 5.1). Así, el esquema de aplicación está formado por los siguientes componentes:

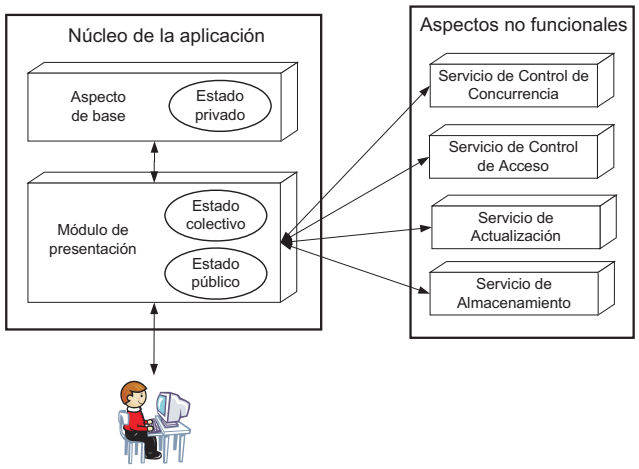


Figura 5.1: Esquema de aplicación de la agenda cooperativa

1. **Estado**: se refiere a los datos que son manipulados por los diferentes componentes de la agenda cooperativa. El estado puede dividirse en dos tipos:

   a) **Citas**: conciernen las reuniones registradas en la agenda cooperativa. Por su carácter consultable/modificable por parte de los colaboradores involucrados, las citas se clasifican en:

      i. **Citas privadas (estado privado)**: son las reuniones de un colaborador que no tienen relación con ninguno de sus grupos de trabajo. Por omisión, los datos de una cita privada sólo pueden ser consultados y modificados por el colaborador implicado (i.e., el propietario de la cita). De esta manera, se mantiene su privacidad y se evita que sus colegas sean interrumpidos con información irrelevante;

      ii. **Citas colectivas (estado colectivo)**: son las reuniones de un colaborador que tienen relación con alguno de sus grupos de trabajo. Por omisión, todos los colaboradores invitados a la reunión pueden consultar los datos de la cita. La modificación de dichos datos está a cargo del creador de la cita, sin embargo la agenda cooperativa permite a éste último otorgar derechos de modificación a sus colegas (a todos o a un grupo designado).

   b) **Ranuras de tiempo libre (estado público)**: son los horarios no utilizados por las citas privadas y colectivas de un colaborador. Por omisión, las ranuras

de tiempo libre de un colaborador están disponibles en consulta y modificación a todos los miembros de sus grupos de trabajo. Sin embargo, la agenda cooperativa permite a cada usuario denegar estos derechos de acceso (consulta, modificación o ambos) a sus colegas (a todos o a un grupo designado).

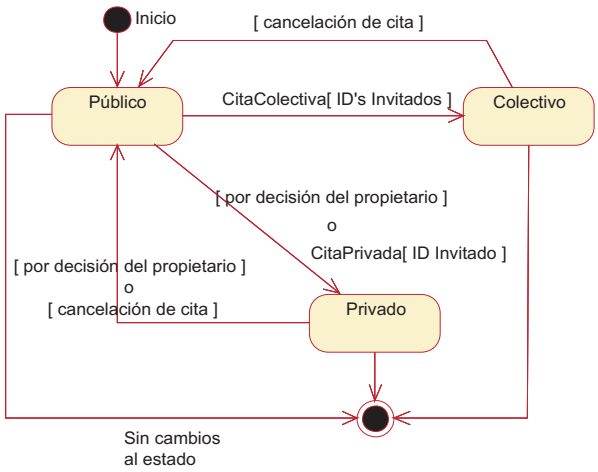


Figura 5.2: Transición de los estados de la agenda cooperativa

El estado de la agenda cooperativa transita por los tres valores mencionados anteriormente (i.e., público, colectivo y privado), cuando se realizan operaciones de creación, modificación, cancelación y reservación de una cita (ver figura 5.2). El valor inicial del estado es público en forma de ranuras de tiempo libre. Sin embargo, un elemento del estado público puede cambiar a privado cuando el colaborador correspondiente: a) crea una cita privada o b) deniega derechos de acceso a los miembros de sus grupos de trabajo. De manera similar un elemento del estado público, común a todos los miembros de un grupo, puede cambiar a colectivo cuando se propone una cita colectiva. Recíprocamente, un elemento del estado colectivo (una cita colectiva) o del estado privado (una cita privada) puede retomar su carácter público cuando los colaboradores implicados en la cita: a) la cancelan, ocasionando que la cita se convierta nuevamente en ranura de tiempo libre u b) otorgan derechos de acceso a todos los miembros de sus grupos de trabajo.

2. **Núcleo de la aplicación**: consta de dos componentes:

   a) **Aspecto de base**: realiza las funcionalidades básicas de la agenda cooperativa, i.e., crear, modificar, cancelar, reservar y confirmar una cita. Además, gestiona las transiciones de las ranuras de tiempo (e.g., libre, ocupada y tentativa) para informar, a los colaboradores, cuáles están disponibles para crear una cita (cf. sección 5.2). El aspecto de base también gestiona grupos e individuos con quienes es posible concretar citas;

b) **Módulo de presentación**: recibe las operaciones provenientes de los dispositivos físicos y las transforma en eventos de la interfaz lógica, con el fin de que puedan ser procesados por los demás módulos de la agenda cooperativa. El módulo de presentación también envía el resultado de las operaciones a los dispositivos físicos de salida para que pueda ser desplegado al usuario.

3. **Aspectos no funcionales**: añaden funcionalidad al aspecto de base de la agenda cooperativa, i.e., actualizan y controlan el acceso al estado compartido[1], mantienen la persistencia y garantizan la coherencia de dicho estado y permiten la implantación de la agenda cooperativa en un entorno distribuido. Además del aspecto de distribución (el cual se describe en la sección 5.1.2) se definieron cuatro aspectos no funcionales adicionales:

   a) **Servicio de Control de Concurrencia**: se encarga de atender las múltiples solicitudes concurrentes (tanto locales como remotas) que provienen de los colaboradores. Además, este servicio resuelve las eventuales incongruencias que se pudieran presentar en el estado compartido;

   b) **Servicio de Control de Acceso**: ofrece funciones a los colaboradores para permitirles otorgar o denegar derechos de acceso (e.g., consulta, modificación y supresión) a sus colegas sobre sus citas colectivas y ranuras de tiempo libre;

   c) **Servicio de Actualización**: envía mensajes de notificación referentes a la creación, modificación, confirmación o cancelación de una cita colectiva hacia la agenda de todo colaborador implicado en la cita. Asimismo, actualiza los cambios realizados en las ranuras de tiempo libre de cada colaborador, las cuales están replicadas en los sitios de sus colegas (i.e., los miembros de sus grupos de trabajo);

   d) **Servicio de Almacenamiento**: pasa los datos de una cita o de una ranura de tiempo libre que se encuentran en la memoria primaria a la memoria secundaria (en forma de archivos de texto), a fin de poderlos recuperar cuando se haga referencia a dicha cita o ranura en una sesión posterior.

En carácter flexible de la agenda cooperativa se obtiene a nivel de los aspectos no funcionales, los cuales añaden funcionalidad al aspecto de base, a fin de que la agenda cooperativa tenga un comportamiento diferente del original. Así, los servicios mencionados pueden fácilmente ser sustituidos por otros, cuando los requerimientos de los usuarios de la agenda cooperativa cambien, o pueden ser reutilizados por otro sistema con requerimientos similares. En la sección 5.2.1 se describen las técnicas utilizadas por cada aspecto no funcional.

[1]Por estado compartido nos referimos a las citas colectivas y las ranuras de tiempo libre de una agenda cooperativa.

5.1.2. Esquema de distribución

El esquema de distribución de la agenda cooperativa (cf. sección 4.3.2) muestra la ubicación tanto de sus componentes como de sus estados en los diferentes sitios de los colaboradores.

El diseño de la distribución de los componentes desembocó en una arquitectura completamente replicada, i.e., en cada uno de los sitios de los colaboradores se encuentra el núcleo de la aplicación y el módulo de cada uno de los aspectos no funcionales (ver figura 5.3). Esta arquitectura tienen las siguientes ventajas: 1) mejora el tiempo de respuesta en la interfaz de usuario, ya que las solicitudes son procesadas localmente y 2) evita los cuellos de botella causados por el único servidor de la arquitectura centralizada. Sin embargo, esta arquitectura también tiene algunas desventajas: 1) requiere un mecanismo que asegure la consistencia del estado compartido e 2) incrementa el tráfico en la red, ya que cualquier cambio realizado en el estado compartido es propagado a todos los sitios que contienen réplicas.

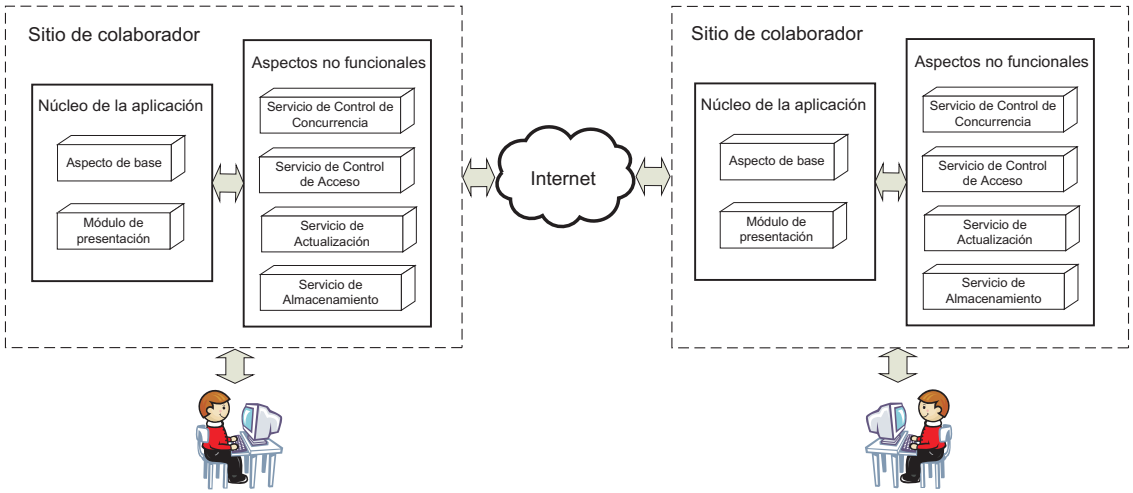


Figura 5.3: Esquema de distribución de los componentes de la agenda cooperativa

Por otra parte, el diseño de la distribución de los estados generó una arquitectura parcialmente replicada, debido a que la agenda cooperativa gestiona tres tipos de estados con diferentes requerimientos de control de acceso (ver figura 5.4).

Las citas privadas solamente están ubicadas en el sitio del colaborador propietario, ya que por cuestiones de privacidad no pueden ser accedidas por ningún otro colaborador. La figura 5.4 muestra que las citas privadas sólo se encuentran en el sitio correspondiente, donde están protegidas para evitar que sean accedidas desde otros sitios, e.g., el sitio A no tiene acceso a las citas privadas de los sitios B ni C y recíprocamente estos últimos no tienen acceso a las citas privadas del sitio A. De manera similar, el sitio B no tiene acceso a las citas privadas del sitio C y éste no tiene acceso a las citas privadas de sitio B.

Las citas colectivas están replicadas sólo en los sitios de los colaboradores involucrados en la cita. El motivo de replicar una cita colectiva se justifica por la necesidad de mantener

su disponibilidad para permitir posibles cambios por parte de los colaboradores implicados en la cita. Así, cuando un colaborador realiza un cambio en una cita colectiva, dicho cambio será notificado a todos los sitios que contienen réplicas. Como se muestra en la figura 5.4, mientras que el colaborador del sitio A mantiene citas colectivas con los colaboradores de los sitios B y C, estos últimos no tienen citas comunes.

Finalmente las ranuras de tiempo libre de cada colaborador están replicadas en los sitios de sus colegas. Por omisión, los miembros de los grupos de trabajo de un colaborador tienen derecho de visualizar sus ranuras de tiempo libre y de proponer una cita que comprenda algunas de estas ranuras. Al igual que las citas colectivas, cuando una ranura de tiempo libre cambia de valor (e.g., de público a colectivo o de público a privado), dicho cambio será notificado a cada uno de los sitios que contienen réplicas de dicha ranura. La figura 5.4 muestra que las ranuras de tiempo libre del sitio A están replicadas en los sitios B y C, las ranuras de tiempo libre del sitio B se encuentran replicadas en los sitios A y C y por último las ranuras de tiempo libre del sitio C están replicadas en los sitios A y B.

Las réplicas de las citas colectivas y de las ranuras de tiempo libre pueden considerarse como de datos de caché, ya que están almacenadas, por un tiempo de vida válido, en el sitio del colaborador que las solicita.

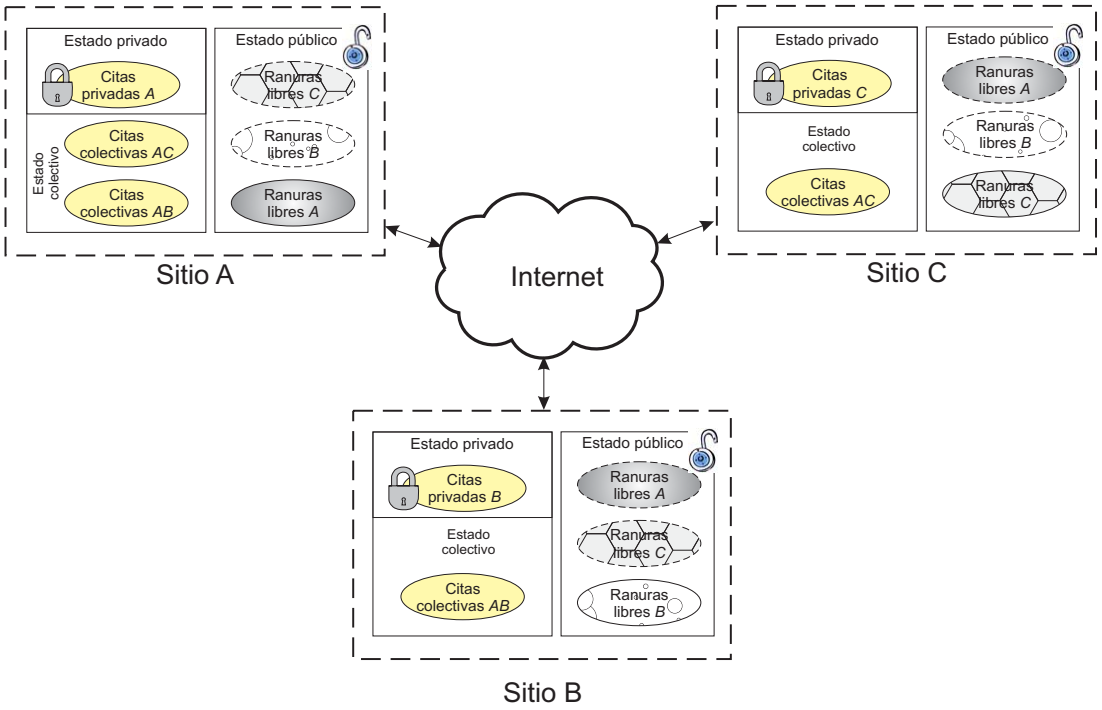


Figura 5.4: Esquema de distribución de los estados de la agenda cooperativa

En conclusión, el diseño completo de la arquitectura de la agenda cooperativa es una arquitectura replicada híbrida, dado que sus componentes se encuentran replicados en cada sitio de los colaboradores, mientras que sus estados están parcialmente replicados.

5.2. Clases de la agenda cooperativa

La implementación de los módulos de la agenda cooperativa, mediante el paradigma de la POA, derivó en dos tipos de clases (ver figura 5.5): las clases del aspecto de base marcadas en color claro y las clases de los aspectos no funcionales (clases de comportamiento) marcadas en color obscuro.

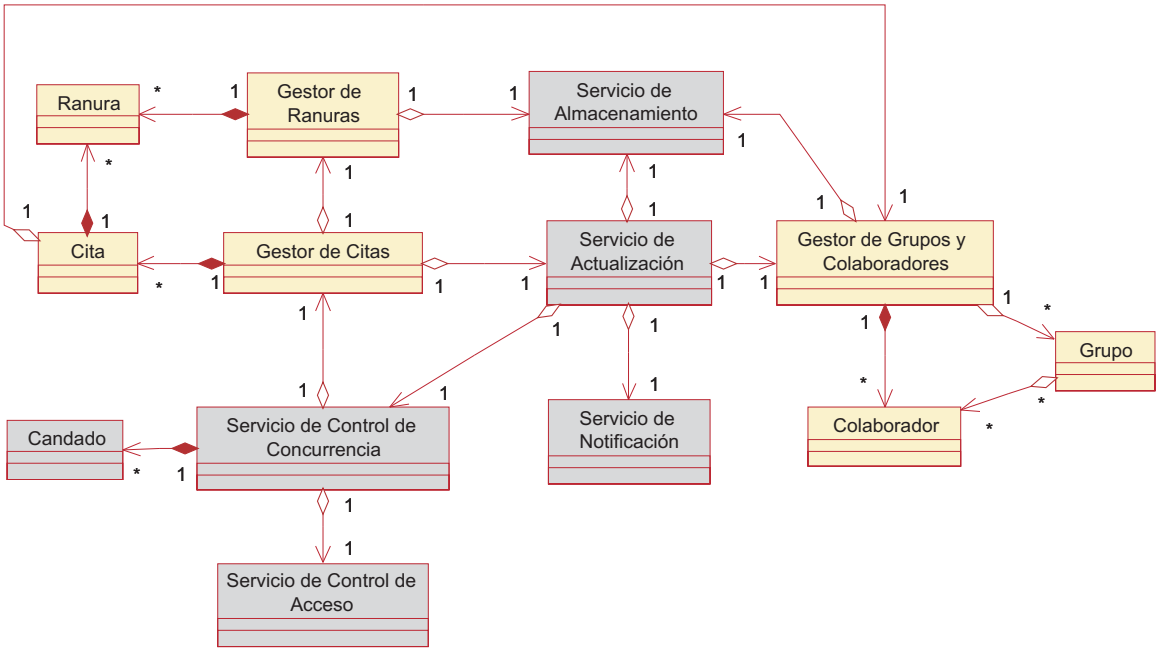


Figura 5.5: Clases de los aspectos de base y no funcionales de la agenda cooperativa

Como se mencionó anteriormente (cf. sección 5.1.1), las ranuras de tiempo definen tres diferentes estados que administra la agenda cooperativa, i.e., citas privadas, citas colectivas y ranuras de tiempo libre. Además, una ranura de tiempo puede transitar por los siguientes tres valores: 1) libre, 2) ocupado y 3) tentativo, dependiendo del tipo de operación que se realice sobre ella (e.g., crear, modificar, cancelar, reservar o consultar una cita). La figura 5.6 muestra gráficamente la transición de una ranura de tiempo por estos diferentes valores:

1. **Libre**: es el valor inicial de una ranura de tiempo. Puede retomar este valor cuando la cita correspondiente se cancela explícitamente o expira el tiempo de espera estipulado para confirmarla.

2. **Ocupado**: una ranura de tiempo toma este valor cuando se confirma la creación de la cita asociada. La ranura de tiempo mantiene este valor indefinidamente, pero si la cita confirmada se cancela, entonces la ranura de tiempo cambiará su valor de ocupado a libre.

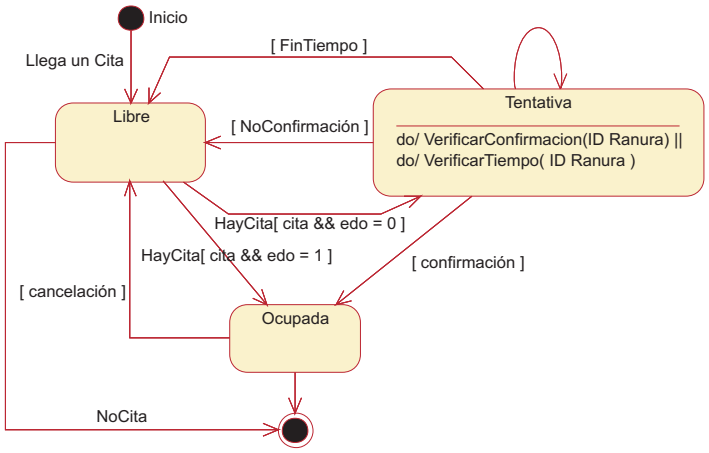


Figura 5.6: Transición de una ranura de tiempo

3. **Tentativo**: una ranura de tiempo toma este valor cuando se crea una cita cuya confirmación depende de la decisión unánime o al menos mayoritaría de los miembros del grupo. Si la cita correspondiente se confirma, entonces la ranura de tiempo cambiará su valor de tentativo a ocupado. En caso contrario, cambiará a libre. La ranura de tiempo mantiene el valor tentativo por un tiempo estipulado. Si este tiempo expira y la cita asociada aún no ha sido confirmada, entonces la ranura de tiempo automáticamente cambiará su valor a libre.

En las siguientes secciones, se describen las clases de los aspectos no funcionales y de base de la agenda cooperativa (secciones 5.2.1 y 5.2.2, respectivamente), haciendo énfasis en la forma de interacción entre ellas y con los distintos estados del sistema.

5.2.1. Clases de los aspectos no funcionales

Los aspectos no funcionales (i.e., el Servicio de Control de Concurrencia, el Servicio de Control de Acceso, el Servicio de Actualización y el Servicio de Almacenamiento) se refieren a los componentes encargados de administrar la agenda cooperativa en un entorno distribuido. Cada uno de ellos tiene una función específica y es independiente de los demás. Estos componentes de procesamiento permiten añadir el comportamiento adicional a la agenda cooperativa, i.e., actualizan y mantienen la coherencia del estado compartido, controlan el acceso a dicho estado y aseguran su persistencia. Cada uno de estos componentes fue implementado por medio de la herramienta de programación AspectJ. El entrelazado con el aspecto de base se realiza mediante el enfoque de implementación de la POA, llamado transformación por programa (cf. sección 3.3.2).

A continuación, se describen los aspectos no funcionales en términos de interacción entre componentes locales y remotos, los cuales permiten administrar el estado compartido de la agenda cooperativa.

Servicio de Almacenamiento

El Servicio de Almacenamiento guarda, en archivos de texto, los datos de una cita o de una ranura de tiempo libre que se encuentran en memoria primaria y los recupera (i.e., extrae los datos que fueron guardados en los archivos de texto) cuando se hace referencia a alguno de ellos.

Un colaborador designado como administrador se encarga de realizar el registro del grupo de trabajo, el cual permite determinar los sitios donde serán almacenadas las replicas del estado compartido. De esta manera, el Servicio de Almacenamiento local guarda los datos de las citas privadas y colectivas, así como de las ranuras de tiempo libre del colaborador local. Además, almacena las ranuras de tiempo libre de los colaboradores remotos que forman parte de los grupos de trabajo del colaborador local.

El Servicio de Almacenamiento utiliza un sistema de archivos que está estructurado de la siguiente manera: a) el almacén de datos locales está formado por carpetas llamadas `diaNum` donde `Num` significa el número del día del mes actual en que se está utilizando la agenda cooperativa; dentro de cada una de estas carpetas se encuentra almacenado un archivo plano, llamado `ranuras.txt`, que contiene los datos de las citas privadas, colectivas y ranuras de tiempo libre del día `Num` indicado y b) el almacén de datos remotos utiliza una carpeta llamada `AG_UsuariosRemotos` donde se guardan, en archivos de texto separados, cada una de las réplicas de los estados compartidos remotos, i.e., una relación 1:1 entre un archivo de texto y una réplica del estado compartido remoto. El estado compartido comprende tanto las citas colectivas y las ranuras de tiempo libre de un colaborador.

El Servicio de Almacenamiento implementa los métodos `GuardarArchivo()` y `LeerArchivo()`. Estos métodos tienen dos diferentes funciones dependiendo de si la operación que se realiza es local o remota.

- `GuardarArchivo()`
  Entrada: La ruta donde se va a almacenar el archivo de texto, los objetos `Ranura` que contienen los datos que van a ser almacenados, estos datos pueden ser un elemento del estado privado o del estado compartido y un entero `Num` que corresponde al día del mes en que se esta realizando la operación.
  Salida: 1 si los datos se pudieron guardar en archivos de texto o **0** en caso contrario.
  Descripción: El almacén de datos local guarda tanto datos que pertenecen a un elemento estado privado o compartido local. Este método recibe como parámetros: 1) una cadena que contiene la ruta donde se encuentra la carpeta `diaNum`, más el nombre del archivo `ranuras.txt`, e.g., `BD/diaNum/ranuras.txt`, 2) los objetos `Ranura` que contienen los datos que pueden pertenecer a una cita privada, una cita colectiva o una ranura de tiempo libre local y 3) un entero `Num` que corresponde al día del mes en que se esta realizando la operación.

  En el caso de una réplica del estado compartido remoto, el método `GuardarArchivo()` recibe: 1) una cadena `AG_UsuariosRemotos` que contiene la ruta en donde se va a almacenar los datos del estado compartido remoto, 2) el identificador del colaborador remoto `IDUserRemoto` que se agregará al nombre del archivo para poder identificar a

quién pertenece la réplica, 3) los objetos `Ranura` que contienen los datos de un elemento del estado compartido remoto, i.e., los datos de una cita colectiva o de una ranura de tiempo libre remota y 4) un entero `Num` que corresponde al día del mes en que se esta realizando la operación.

- `LeerArchivo()`
  Entrada: Los parámetros de la cita o de la ranura de tiempo libre que se van a recuperar de los archivos de texto almacenados en memoria secundaria.
  Salida: Una cadena `ValoresCita` que contiene los datos de la cita o de la ranura de tiempo libre solicitada o **0** si ocurrió un error al tratar de recuperar los datos.
  Descripción: El método `LeerArchivo()` extrae del almacén de datos locales tanto los datos de citas privadas y colectivas y de las ranuras de tiempo libre. Este método recibe como parámetros: 1) una cadena que contiene la `ruta` en donde se encuentra el archivo de texto, 2) un entero `Num` que corresponde al día del mes al que pertenece la cita privada o la cita colectiva o la ranura de tiempo libre y 3) un objeto `Ranura` para cargar los datos de la cita o de la ranura de tiempo libre solicitada.

  En el caso de una réplica del estado compartido remoto, el método `LeerArchivo()` recibe adicionalmente una cadena `IDUserRemoto` para extraer la réplica correcta.

Clase `Candado`

Es una clase que permite colocar un candado simple a una ranura de tiempo, con el propósito de bloquear el acceso concurrente a dicha ranura cuando está siendo manipulada por un colaborador. Esta clase ayuda al Servicio de Control de Concurrencia a aplicar el mecanismo de candados (cf. sección 4.2.2). La clase `Candado` implementa los siguientes métodos:

- `ColocarCandado()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo a la que se va a colocar el candado, el identificador `IDSolicitante` del colaborador que va a manipular la ranura de tiempo y una cadena `tipo` que contiene el tipo de operación (e.g., crear, modificar, cancelar, reservar o consultar una cita) que se va a realizar.
  Salida: 1 si se pudo colocar el candado o **0** en caso contrario.
  Descripción: Coloca el/los candados a/las ranuras de tiempo para bloquear el acceso concurrente a estas, mientras una solicitud está siendo atendida.

- `LiberarCandado()`
  Entrada: El identificador `IDRanura` de la ranura a la que se va a quitar el candado y el identificador `IDSolicitante` del colaborador que manipuló la ranura.
  Salida: 1 si se pudo liberar el candado o **0** en caso contrario.
  Descripción: Quita el/los candados colocados a las ranuras de tiempo una vez que la operación ha finalizado.

Servicio de Control de Concurrencia

El Servicio de Control de Concurrencia se encarga de resolver los problemas de inconsistencia de datos, ocasionados por el acceso concurrente al estado compartido. Para atacar dichos problemas se hace uso del mecanismo de candados pesimistas (cf. sección 4.2.2), el cual garantiza que un solo colaborador a la vez acceda a los estados público o colectivo, forzando al sitio solicitante a esperar hasta que la petición de un candado sea resuelta antes de permitirle manipular un elemento de dicho estado. Para reforzar la seguridad de acceso a la información, se hace uso de políticas que el Servicio de Control de Acceso aplica para decidir qué colaboradores tendrán acceso al estado compartido y quienes no.

El Servicio de Control de Concurrencia hereda las propiedades de la clase `Candado` para hacer uso de sus métodos `ColocarCandado()` y `LiberarCandado()`. Además de los métodos heredados, el Servicio de Control de Concurrencia implementa los siguientes métodos:

- `TiempoDeEspera()`
  Entrada: El identificador `IDRanura` de la ranura a la que se va añadir el tiempo de espera.
  Salida: **1** si se pudo completar la operación y **0** en caso contrario.
  Descripción: Añade un tiempo de espera a la ranura para que cambie su valor de tentativa a ocupada o libre.

- `TipoDeOperacion()`
  Entrada: El tipo de operación que se va a atender.
  Salida: No regresa nada.
  Descripción: Determina si se trata de una operación local o remota. En el caso de una operación local, el candado es otorgado automáticamente. Sin embargo, si se trata de una operación remota el Servicio de Control de Concurrencia hace uso del método `AplicarPoliticas()`, el cual hace una invocación al Servicio de Control de Acceso para saber si la operación solicitada puede ser atendida o no.

- `AplicarPoliticas()`
  Entrada: El identificador `IDRanura` de la ranura solicitada y el identificador `IDSolicitante` del colaborador que realiza la solicitud.
  Salida: No regresa nada.
  Descripción: Invoca al Servicio de Control de Acceso para aplicar las políticas de acceso y definir si la operación solicitada por el colaborador remoto tiene derechos de acceso a la ranura de tiempo.

Servicio de Control de Acceso

El Servicio de Control de Acceso aplica políticas de acceso al estado compartido de la agenda cooperativa, i.e., cuando un colaborador remoto envía una solicitud para realizar una operación sobre un elemento del estado compartido, dicha solicitud debe aprobar todas las políticas definidas para poder realizar las modificaciones solicitadas. En caso de que la solicitud no apruebe dichas políticas, esta no se llevará a cabo.

A continuación, se listan las políticas que el Servicio de Control de Acceso aplica a las solicitudes provenientes de colaboradores remotos:

- El colaborador designado para registrar el grupo de trabajo tiene la opción de definir a qué ranuras de tiempo tendrá acceso dicho grupo.

- Las operaciones de lectura se podrán otorgar a todos los miembros del grupo de trabajo que las soliciten, ya que este tipo de operaciones no crea cambios en las ranuras de tiempo. Las operaciones de lectura son atendidas de acuerdo al orden en que el Servicio de Control de Concurrencia les de acceso.

- Si la(s) ranura(s) de tiempo, que solicita un colaborador remoto, no está(n) disponible(s) la solicitud correspondiente queda en "espera", pero se envía una notificación al solicitante para que decida si espera a que la ranura eventualmente se libere o si propone otro periodo de tiempo para realizar su cita.

- Para ejemplificar la tercera politíca, suponga que dos colaboradores $C1$ y $C2$ están realizando una solicitud de creación de cita sobre una misma ranura de tiempo del colaborador $C3$. En este tipo de solicitud surgen varios casos que el Servicio de Control de Acceso debe resolver:

  1. Si la ranura de tiempo ya está ocupada, entonces las solicitudes serán rechazadas.
  2. Si $C1$ tiene mayor prioridad que $C2$, entonces se otorga el derecho de acceso a $C1$ y la solicitud de $C2$ queda en "espera" hasta que $C1$ libere eventualmente la ranura de tiempo. Si $C1$ deja la ranura de tiempo con el valor ocupado, entonces la solicitud de $C2$ será rechazada.
  3. Si $C1$, $C2$ y $C3$ realizan al mismo tiempo sus solicitudes, entonces el derecho de acceso se otorgará al colaborador con mayor jerarquía en el grupo. Por el contrario, si la prioridad es igual para todos, entonces el derecho de acceso se otorga al colaborador propietario de la ranura de tiempo.

El Servicio de Control de Acceso implementa el método:

- `AplicandoPoliticas()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo solicitada y el identificador `IDSolicitante` del colaborador que realiza la solicitud.
  Salida: `True` si la solicitud cumple con las políticas de acceso o `False` en caso contrario.
  Descripción: Implementa las políticas definidas anteriormente. La ventaja hacer una separación entre las políticas de acceso y el control de concurrencia reside en la facilidad de implementación de dichas políticas ya que se tratan de condiciones que deben cumplirse secuencialmente. Así, si alguna política no se cumple, la secuencia se romperá ocasionando que la solicitud sea rechazada.

Servicio de Notificación

El Servicio de Notificación prepara los mensajes que el Servicio de Actualización envía a sus pares remotos cuando ocurre una actualización del estado compartido. El Servicio de Notificación envía los mensajes de notificación sólo a los sitios de los colaboradores concernidos en la citas colectivas que han sido modificadas.

La información que se envía a los colaboradores remotos, en forma de mensajes, es diferente dependiendo de la operación que se haya realizado. Por lo tanto, una notificación se envía si:

- Una de las réplicas de una ranura de tiempo (cita colectiva o ranura de tiempo libre) ha cambiado su valor:
  1. una cita colectiva ha sido creada, así que los invitados deben ser notificados;
  2. un colaborador remoto responde a la invitación de una cita colectiva;
  3. una cita colectiva ha sido confirmada o anulada, en consecuencia los invitados deben ser notificados.

El Servicio de Notificación implementa los métodos:

- `SeleccionarCandidatos()`
  Entrada: El arreglo `IdentificadoresCR[]` de identificadores de los colaboradores implicados en la cita colectiva.
  Salida: Las direcciones físicas (IPs) de los sitios de los colaboradores implicados en la cita colectiva.
  Descripción: Selecciona las direcciones físicas de los sitios de los colaboradores que deben ser notificados de los cambios del estado compartido, i.e., dichos sitios remotos contienen una réplica del estado compartido.

- `ConstruirMensaje()`
  Entrada: Un arreglo de cadenas `IdentificadoresCR[]` que contiene los identificadores de los colaboradores remotos a los que se va a enviar el mensaje de notificación y una cadena `tipo` que guarda el tipo de operación que se realizó.
  Salida: El mensaje que se va a enviar.
  Descripción: Construye el mensaje que se va a enviar como notificación, dependiendo de la operación que se llevó a cabo. Recibe como parámetros: 1) los identificadores `IdentificadoresCR[]` de los colaboradores implicados en la cita colectiva y 2) un parámetro `tipo` para identificar el tipo de operación que se realizó y construir el mensaje adecuado que se va a enviar como notificación, e.g., 1) "se creó una cita", 2) "se canceló la cita" y 3) "la cita no se concretó".

- `InterpretarMensaje()`
  Entrada: El mensaje recibido.
  Salida: El tipo de actualización que se ha hecho al estado compartido.

Descripción: Interpreta los mensajes que resultan de los cambios efectuados en el estado compartido (e.g., una modificación al valor de una cita colectiva, a una ranura de tiempo libre local o a una ranura de tiempo libre remota).

Servicio de Actualización

El Servicio de Actualización se encarga de actualizar y notificar a sus pares remotos, que uno o varios elementos del estado compartido han sido modificados. Básicamente si un miembro del grupo de trabajo realiza cambios sobre un elemento del estado compartido (e.g, si crea, modifica o cancela una cita colectiva), entonces el Servicio de Actualización debe actualizar todas las réplicas del estado compartido que se encuentran alojadas en varios sitios remotos. Del mismo modo, el servicio de actualización debe de informar, a través del Servicio de Notificación, a todos los colaboradores concernidos en las citas colectivas, del tipo de actualizaciones que se han realizado.

El enfoque *push* en convinación con la transmisión *multicast* se utilizan cuando se lleva a a cabo una actualización al estado compartido que se encuentra replicado en varios sitios remotos. Por lo tanto, la actualización de las réplicas del estado compartido se lleva a cabo si:

- un colaborador hace cambios al estado compartido, i.e., modifica el valor de una cita colectiva o una ranura de tiempo libre que es elemento del estado compartido;
- una ranura de tiempo cambia su valor, e.g., de tentativo a ocupado o de tentativo a libre;
- una cita confirmada es cancelada;
- si un miembro del grupo de trabajo ha estado fuera de línea por un determinado tiempo, durante el cual dicho miembro ha realizado cambios al estado compartido (i.e., ha cambiado el valor de una cita colectiva o ha creado citas sobre una ranura de tiempo libre que pertenece al estado compartido) u otros miembros del grupo de trabajo han creado citas que lo conciernen, entonces las réplicas del estado compartido deben ser actualizadas para evitar posibles incoherencias, producto de las actualizaciones que se realizaron al estado compartido durante el tiempo que el colaborador estuvo fuera de línea.

El Servicio de Actualización implementa los siguientes métodos:

- `NotificarActualizacion()`
  Entrada: La lista de `IdentificadoresCR[]` de los sitos de colaboradores remotos invitados a la cita colectiva y el identificador `IDSolicitante` del colaborador solicitante.
  Salida: `True` si se envío correctamente la notificación o los identificadores de los colaboradores a los que no se envió correctamente la notificación.
  Descripción: Invoca a los Servicios de Actualización remotos que se encuentra replicados en los sitios de los colaboradores invitados a la cita colectiva y les envía un mensaje de notificación, informando que un elemento del estado compartido ha sido modificado.

- `CrearMensaje()`
  Entrada: La lista de identificadores `IdentificadoresCR[]` de los sitios remotos a los que se va a enviar el mensaje de notificación.
  Salida: El mensaje de notificación.
  Descripción: Invoca al Servicio de Notificación local para obtener el mensaje que enviará a sus pares remotos que contienen réplicas del estado compartido.

- `VerificarExistenciaReplicas()`
  Entrada: La lista `Ranuras[]` de las ranuras de tiempo utilizadas en la cita colectiva.
  Salida: La lista de las direcciones físicas de los sitios remotos que contienen réplicas de las ranuras utilizadas.
  Descripción: Verifica si las ranuras de tiempo modificadas están replicadas en sitios remotos.

- `ActualizarRanura()`
  Entrada: La lista de identificadores `IdentificadoresCR[]` de los sitios remotos que contienen réplicas del estado compartido, el identificador `IDSolicitante` del colaborador solicitante y todos los elementos modificados del estado compartido.
  Salida: `True` si se pudo actualizar el estado compartido o `False` en caso contrario.
  Descripción: Envía la actualización de todos los elementos modificados del estado compartido a los diferentes sitios remotos que contienen réplicas.

- `ActualizarEdo()`
  Entrada: El identificador `IDSolicitante` del solicitante y todos los elementos modificados del estado compartido.
  Salida: `True` si se pudo actualizar el estado compartido o `False` en caso contrario.
  Descripción: Solicita al `Gestor de Citas` guarde los cambios al estado compartido, i.e., el servicio de actualización hace uso del `Gestor de Citas`, que es una clase del aspecto de base, para que este último solicite al `Gestor de Ranuras` que almacene los cambios hechos al estado compartido.

- `EsUsuarioValido()`
  Entrada: El identificador `IDSolicitante` del colaborador solicitante.
  Salida: `True` si el identificador es válido y `False` en caso contrario.
  Descripción: Pasa la solicitud al Servicio de Control de Concurrencia para que determine si dicha solicitud debe ser atendida o no.

La operaciones que realiza el Servicio de Actualización se envían como solicitudes a sus pares remotos, por lo tanto, pasan a través del Servicio de Control de Concurrencia remoto, debido a que son operaciones que tienen la capacidad de modificar el valor de elementos del estado compartido.

5.2.2. Clases del aspecto de base

Las clases del aspecto de base definen el comportamiento básico de la agenda cooperativa. Al menos, estas clases deben existir como elementos básicos para que funcione la agenda cooperativa. La función del aspecto de base es manipular los elementos del estado privado local. El aspecto de base se compone de: 1) clases principales que se son: a) `Gestor de Ranuras`, 2) `Gestor de Citas` y 3) `Gestor de Colaboradores y Grupos` y 2) clases secundarias que son: a) `Ranura`, b) `Cita`, c) `Grupo` y d) `Colaborador`. Las clases secundarias tiene una funcionalidad más básica que las clases principales, sin embargo ayudan en el desempeño de las clases principales. Por lo tanto, en primera instancia se describen las clases secundarias para dar un panorama de su utilidad y después se describe el funcionamiento de las clases principales.

Clase `Ranura`

Es el elemento básico de los estados (privado, colectivo y público) de la agenda cooperativa, puede guardar valores de una cita privada o de una cita colectiva o una ranura de tiempo libre, por lo tanto, los cambios que se realizan a las citas privadas y colectivas o ranuras de tiempo libre, quedan registrados en el objeto `Ranura`.

Este objeto implementa todos los parámetros necesarios que hacen a una ranura de tiempo un elemento único, dentro de la agenda cooperativa, para soportar cualquier tipo de operación. Algunos parámetros asociados al objeto `Ranura` son: 1) el `estado` que es estado al cual pertenece la ranura (e.g., privado, colectivo o público), 2) un parámetro `valor` que es el valor en que se encuentra la ranura (e.g., ocupada, libre o tentativa), 3) la lista `IdentificadoresCR[]` de contiene a los colaboradores invitados a la cita, 4) el `tiempoEspera` que es el tiempo de espera estipulado para confirmar una cita, 5) los `detallesDeCita` que es la descripción de la cita que se llevará a a cabo en ese horario, 6) la `fechaReunion` que es la fecha en que se llevará a cabo la cita, 7) el identificador `IDSolicitante` es el identificador del colaborador que propuso la cita y un `identificador` para la ranura de tiempo. Además, en la ejecución de la agenda cooperativa, se crean los objetos `Ranura` del mes en curso (i.e., 24 ranuras, una por cada hora del día, por los días que comprenden al mes), de esta forma un colaborador pueda crear una cita, personal o colectiva, en cualquier horario de los días del mes en curso.

Los parámetros del objeto `Ranura` se encuentran encapsulados para evitar que los datos puedan ser modificados por otros objetos, diferentes al `Gestor de Ranuras`, durante la ejecución del sistema. Por lo tanto, la clase `Ranura` implementa los métodos `get()` y `set()` para obtener y modificar respectivamente el valor de un parámetro de manera segura.

Clase `Cita`

Implementa los métodos que realizan las operaciones básicas en la agenda cooperativa. Los métodos de esta clase son heredados por la clase `Gestor de Citas`, debido a que dicho gestor recibe todas las peticiones de operaciones en la agenda cooperativa. La clase `Cita` implementa los siguientes métodos:

- `CrearCita()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo, un entero `Num` que corresponde al día del mes es que se va a realizar la operación, el identificador `IDSolicitante` del colaborador solicitante, una cadena `DetallesCita` donde se especifican los datos de la cita y el `tipo` de cita que se va a crear (privada o colectiva).
  Salida: `True` si se pudo completar la operación o `False` en caso contrario.
  Descripción: Crea una cita si la(s) ranura(s) de tiempo solicitada(s) se encuentra(n) disponible(s); el valor final de cada ranura después de aplicar esta operación es `Ocupado`.

- `ReservarRanura()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo, un entero `Num` que corresponde al día del mes es que se va a realizar la operación, el identificador `IDSolicitante` del colaborador solicitante, una cadena `DetallesCita` donde se especifican los datos de la cita y el `tipo` de cita que se va a crear.
  Salida: `True` si se pudo completar la operación o `False` en caso contrario.
  Descripción: Reserva una o varias ranuras de tiempo que potencialmente son candidatas a crear una cita; el valor final de cada ranura después de aplicar esta operación es `Tentativo`.

- `EliminarCita()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo que almacena la cita que se va a cancelar y el identificador `IDSolicitante` del colaborador solicitante.
  Salida: `True` si se pudo cancelar la cita o `False` en caso contrario.
  Descripción: Libera una o varias ranuras de tiempo ocupadas por citas confirmadas o tentativas; el valor final de cada ranura después de aplicar esta operación es `Libre`.

- `SeleccionarGrupo()`
  Entrada: Los identificadores `IdentificadoresCR[]` de los colaboradores invitados a la cita.
  Salida: El tipo de grupo[2] con el cual se va a realizar la cita y el nombre del grupo.
  Descripción: Invoca al `Gestor de Colaboradores y Grupos` para obtener el tipo de grupo con el cual se va a concretar la cita. Los tipos de grupos son: 1) `Común`, la cita se crea con miembros de un mismo grupo; 2) `Mixto`, la cita se crea con miembros de diferentes grupos y 3) `Individual`, la cita se crea con un único colaborador de cualquier grupo. Además, el método `SeleccionarGrupo()` devuelve el nombre del grupo al que pertenece(n) el(los) colaborador(es) invitado(s) a la cita.

[2]Un colaborador puede definir un grupo para cada actividad que va a realizar, i.e., se pueden definir diferentes tipos grupos para diferentes actividades.

Clase `Grupo`

La clase `Grupo` almacena los parámetros necesarios para identificar a los colaboradores que forman parte de uno o varios grupos dentro de la agenda cooperativa. Los parámetros que almacena esta clase son: 1) el identificador para el grupo (`IDGrupo`), 2) una cadena (`Nombre`) que define el nombre del grupo, 3) un entero (`Numero`) que guarda el numero de miembros en el grupo, 4) un arreglo `IdentificadoresCR[]` que almacena los identificadores de los colaboradores miembros del grupo, 5) una cadena (`Líder`) que contiene el nombre del líder del grupo y 5) una cadena (`Tipo`) que contiene la actividad que se realiza con ese grupo de trabajo.

Los parámetros de la clase `Grupo` están encapsulados dentro de la agenda cooperativa. Por lo tanto, la clase `Grupo` implementa los métodos `get()` y `set()` para obtener y modificar respectivamente el valor de un parámetro de manera segura.

Además de los metodos `get()` y `set()`, esta clase implementa los siguientes métodos:

- `CrearGrupo()`
  Entrada: Los identificadores `IdentificadoresCR[]` de los colaboradores que van a pertenecer a formar parte del grupo y el nombre del grupo que se va a crear.
  Salida: `True` si se pudo formar el grupo y `False` en caso contrario.
  Descripción: Toma los identificadores de los colaboradores y los coloca dentro del grupo que se va a crear. El colaborador designado como administrador crea el nuevo grupo y asigna el identificador a cada grupo. Los grupos registrados son almacenados en archivos de texto por el Servicio de Almacenamiento local.

- `EliminarGrupo()`
  Entrada: El identificador `IDGrupo` del grupo a eliminar.
  Salida: `True` si se pudo eliminar el grupo y `False` en caso contrario.
  Descripción: Elimina un grupo registrado, i.e., solicita al Servicio de Almacenamiento que elimine el archivo de texto que corresponde al grupo solicitado.

- `ModificarMiembrosGrupo()`
  Entrada: Una lista `IdentificadoresCR[]` con los identificadores de los colaboradores que van a cambiar de grupo y el identificador `IDGrupo` del grupo a modificar.
  Salida: `True` si se pudo modificar el grupo y `False` en caso contrario.
  Descripción: Busca si los identificadores de los colaboradores están dados de alta en la agenda cooperativa. Si no lo están los da de alta y los añade al grupo solicitado; en caso contrario, los añade al nuevo grupo. Si el grupo solicitado no existe, entonces en ese momento lo crea; en caso contrario, simplemente lo modifica.

- `BuscarGrupo()`
  Entrada: El identificador del grupo.
  Salida: La lista de los identificadores de los colaboradores que pertenecen al grupo o `False` si el grupo no existe.
  Descripción: Busca a un grupo específico. Si el grupo existe solicita al Servicio de

Almacenamiento que recupere los datos de los colaboradores que pertenecen al grupo y los guarda en una lista de objetos `Colaborador` (memoria principal).

- `BuscarIndividuos()`
  Entrada: Una lista `IdentificadoresCR[]` con los identificadores de los colaboradores a buscar.
  Salida: La lista de objetos `Colaborador` que contienen los valores de los datos de los colaboradores y el tipo de grupo al que pertenecen, para crear una cita privada o colectiva.
  Descripción: Busca a los colaboradores solicitados dentro de los diferentes grupos de trabajo para crear una cita potencial con ellos. Esto búsqueda se realiza debido a que no necesariamente una cita se crea con los mismos miembros de un grupo de trabajo.

Clase `Colaborador`

Esta clase contiene los parámetros necesarios para identificar a un colaborador. Además, facilita la manipulación de los estados de la agenda cooperativa, ya que brinda todos los datos relacionados con un colaborador, e.g., la `IP` de su sitio y su rol dentro del grupo. Cuando se hace referencia a un colaborador, el Servicio de Almacenamiento local transfiere sus datos de memoria secundaria a un objeto `Colaborador`.

Los parámetros que almacena este objeto son: 1) una cadena (`IP`) que almacena la IP de su sitio, 2) un entero (`Puerto`) para guardar el puerto de comunicación, 2) el identificador `IDColaborador` para colocar el identificador del colaborador, 3) una arreglo `Grupo` para identificar los grupos que pertenece el colaborador y 4) una cadena `Rango` que describe el rol del colaborador.

Los parámetros de la clase `Colaborador` se encuentra encapsulados dentro de la agenda cooperativa. Por lo tanto, esta clase implementa los métodos `get()` y `set()` para obtener y modificar respectivamente el valor de un parámetro de manera segura.

`Gestor de Ranuras`

La clase `Gestor de Ranuras` realiza consultas a los estados de la agenda cooperativa para conocer el valor de sus elementos, cada que se requiere crear una cita.

La clase `Gestor de Ranuras` solicita al Servicio de Almacenamiento que recupere los datos de la ranura de tiempo y los carga en memoria primaria para que estén disponibles cuando se haga referencia a ellos. La clase `Gestor de Ranuras` carga, en objetos `Ranura` los datos de: 1) citas privadas o 2) citas colectivas o 3) las ranuras de tiempo libre locales o remotas.

El `Gestor de Ranuras` implementa los siguientes métodos:

- `CargarEstado()`
  Entrada: Un objeto `DiaAgenda` que esta formado de objetos `Ranura` para almacenar los datos de las ranuras de tiempo que pertenecen al estado de la agenda solicitado (e.g., el estado privado o compartido) y una cadena `NomArchivo` que contiene la ruta

del archivo de texto que almacena los datos de las ranuras de tiempo pertenecientes al estado solicitado.
Salida: El objeto `DiaAgenda` cargado en memoria primaria con los datos de las ranuras de tiempo que pertenecen al estado solicitado.
Descripción: Carga en memoria primaria los datos de los elementos que pertenecen al estado solicitado. Para llevar la anterior operacion, el método `CargarEstado()` solicita al Servicio de Almacenamiento local que recupere de memoria secundaria los datos solicitados.

- `CargarEstadoRemoto()`
  Entrada: Un objeto `DiaAgenda` que esta formado por objetos `Ranura` para almacenar los datos de las ranuras de tiempo que pertenecen al estado compartido (i.e., los datos de una cita colectiva o de una ranura de tiempo libre remota), el identificador `IDPropietario` del colaborador propietario de la réplica.
  Salida: El objeto `DiaAgenda` cargado, en memoria primaria, con los valores de las ranuras de tiempo que pertenecen al estado compartido remoto.
  Descripción: Carga en memoria primaria los datos de las ranuras de tiempo que pertenecen al estado compartido. Debido a que la réplica del estado compartido se encuentra almacenada de manera local, este método también solicita al Servicio de Almacenamiento que recupere de memoria secundaria los datos de los elementos del estado compartido.

- `VerificandoDisponibilidad()`
  Entrada: El identificador de la ranura de tiempo `IDRanura` para verificar su disponibilidad y la cadena `fecha` que contiene el valor de la fecha en que se va a crear la cita.
  Salida: `True` si la ranura está disponible o `False` en caso contrario.
  Descripción: Verifica si la ranura de tiempo solicitada está disponible. La verificación de una ranura de tiempo aplica tanto para operaciones locales como remotas, debido a que es el mismo tipo de consulta.

`Gestor de Citas`

La clase `Gestor de Citas` recibe todas las operaciones de lectura/escritura que se van a realizar tanto en el estado privado como en el estado compartido. La clase `Gestor de Citas` atiende las operaciones de manera secuencial y así: 1) en modo monousuario, la clase `Gestor de Citas` recibe las solicitudes del módulo de presentación y las encamina a las distintas entidades de la agenda cooperativa para que realicen su función y 2) en modo multiusuario, la clase `Gestor de Citas` recibe las solicitudes del Servicio de Control de Concurrencia, debido a que éste último asigna prioridades de acceso, al estado compartido, también de manera secuencial.

En la recepción de cada solicitud, la clase `Gestor de Citas` solicita al Servicio de Control de Concurrencia un candado para cada ranura de tiempo implicada en la solicitud con el fin bloquear el acceso a dichas ranuras mientras dura el procesamiento de las operacio-

nes solicitadas, evitando de esta forma que los datos de las citas que se están creando se corrompan.

La clase `Gestor de Citas` hereda todos lo métodos de la clase `Cita` pero también implementa métodos propios para poder atender todas las operaciones solicitadas, locales o remotas. Los métodos que implementa son:

- `SolicitarCandado()`
  Entrada: El identificador `IDRanura` de la ranura de tiempo a la que se va a colorar el candado, el identificador `IDSolicitante` del colaborador solicitante y una cadena `tipo` que contiene el tipo de operación que se va a realizar (local o remota).
  Salida: `True` si se pudo colocar el candado y `False` en caso contrario.
  Descripción: Solicita al Servicio de Control de Concurrencia que coloque un candado sobre la ranura de tiempo solicitada. En una operación local el candado se otorga automáticamente y en una operación remota (e.g., crear, modificar, reservar y cancelar una cita e indagar el tiempo de otro miembro del grupo) el candado depende del Servicio de Control de Concurrencia remoto.

- `CambioValorR()`
  Entrada: El identificador `IDRanura` de la ranura que cambió su valor y el identificador `IDSolicitante` del colaborador solicitante.
  Salida: `True` si se pudo notificar el cambio y `False` en caso contrario.
  Descripción: Esta operación se realiza sólo en modo multiusuario, de forma que si el tipo de operación es local no se manda ninguna notificación. Además, este método invoca al Servicio de Actualización para que envíe la notificación, a su par remoto, del cambio al estado compartido.

- `IndagarTiempoColaboradorRemoto()`
  Entrada: El identificador `IDUserRemoto` del colaborador remoto al que se va a indagar su tiempo libre y dos enteros `tiempoIni` y `tiempoFin` que especifican el rango de tiempo solicitado.
  Salida: La lista de los valores de las ranuras solicitadas o `False` si ocurrio una falla al obtener los valores de las ranuras de tiempo solicitadas.
  Descripción: Obtiene los datos de las ranuras de tiempo solicitadas. Estos datos son obtenidos de la réplica del estado compartido que está almacenada localmente y que son las ranuras de tiempo libre del colaborador remoto solicitado. Por lo tanto, utiliza al `Gestor de Ranuras` el cual a su vez utiliza al Servicio de Almacenamiento para obtener los datos deseados.

- `VerificarDisponibilidad()`
  Entrada: El identificador `IDRanuras` de la ranura de tiempo que se va a verificar su disponibilidad.
  Salida: `True` si la ranura está disponible o `False` en caso contrario.
  Descripción: Toma el identificador de las ranura de tiempo y solicita al `Gestor de Ranuras` que verifique, por medio del método `VerificandoDisponibilidad()`, si la

ranura de tiempo está disponibles. Si una ranura de tiempo no está disponible y forma parte de una cita que engloba múltiples ranuras de tiempo, entonces el `Gestor de Citas` responde a la entidad, que solicitó verificar la disponibilidad de la ranura de tiempo que no se pudo completar la operación (en una operación local la notificación se envía al módulo de presentación y en una operación remota se envía al Servicio de Actualización).

`Gestor de Colaboradores y Grupos`

La clase `Gestor de Colaboradores y Grupos` maneja los datos de todos los colaboradores remotos e implementa métodos que ayudan al colaborador designado como administrador a crear grupos de trabajo. Cuando se hace referencia a los datos de un colaborador remoto, el `Gestor de Colaboradores y Grupos` solicita al Servicio de Almacenamiento local que recupere dichos datos para mantenerlos en memoria primaria. Si los datos ya no son requeridos, el `Gestor de Colaboradores y Grupos` los elimina de la memoria primaria, aunque los datos no se pierden ya que permanecen en memoria secundaria.

El Gestor de `Gestor de Colaboradores y Grupos` hereda los métodos de la clase `Grupo` y crea objetos `Colaborador` para cargar en ellos los datos recuperados por el Servicio de Almacenamiento.

5.3. Escenarios de la aplicación

Para modelar los escenarios en que se desenvuelve la agenda cooperativa, se hace uso del término **transacción**. Este término es utilizado comúnmente en bases de datos y una definición formal es: una transacción es un conjunto de operaciones que se realizan (todas o ninguna) para modificar el estado de una base de datos, sin que los datos pierdan consistencia; además, una transacción cumple con las propiedades de atomicidad, consistencia, aislamiento y durabilidad (propiedades **ACID**)[Ceri and Pelagatti, 1984]. En sistemas ideales, las transacciones deben garantizar todas las propiedades ACID sin embargo en la práctica, alguna de estas propiedades se simplifica para obtener un mejor rendimiento en el sistema.

En la agenda cooperativa, una transacción involucra operaciones de lectura y escritura, e.g., las operaciones de crear, modificar, cancelar, reservar y confirmar una cita.

Como se mencionó en la sección 5.2 de este capítulo, las operaciones locales son soportadas por el aspecto de base de la agenda cooperativa. Sin embargo, como se muestra en el escenario 1 figura 5.7, para realizar una transacción referente a la creación una cita colectiva sobre una ranura de tiempo libre local, el aspecto de base utiliza el Servicio de Control de Concurrencia para colocar un candado sobre la ranura de tiempo, valido mientras dura la transacción. La razón de colocar el candado a la ranura de tiempo es evitar que otro miembro del grupo de trabajo tenga acceso a dicha ranura y evitar inconsistencias en los datos. Cuando la transacción haya terminado, independientemente del resultado de la misma, la ranura de tiempo será liberada. Si otro miembro del grupo de trabajo desea modificar el valor de la

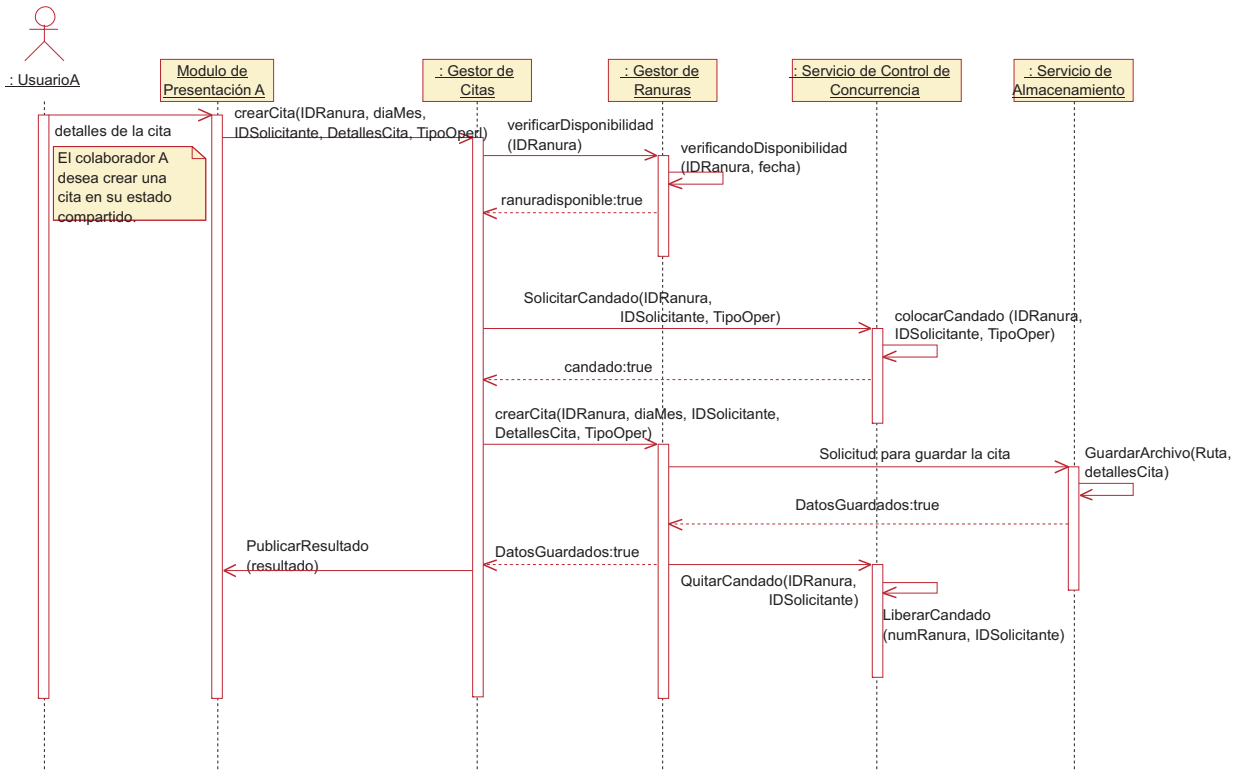


Figura 5.7: Escenario 1: El colaborador local crea una cita sobre una ranura de tiempo libre local

misma ranura de tiempo, entonces dicho miembro debe ser un invitado a la cita colectiva creada, en caso contrario, se le negará el acceso a dicha ranura.

La posibilidad de falla se muestra en el escenario 2, figura 5.8, en una trasacción referente a la creación de una cita sobre una ranura de tiempo libre local, ocurre cuando el colaborador local quiere hacer una actualización sobre una ranura de tiempo que previamente ha sido modificada por otro colaborador, i.e., la ranura de tiempo ya no está disponible. Existen dos casos en que la falla se confirma: 1) si el valor de la ranura de tiempo es "ocupado" y 2) el valor de la ranura de tiempo es "tentativo" pero finalmente se confirma. La posibilidad de no considerar la falla ocurre cuando el valor de la ranura de tiempo es "tentativo", pero la cita no se confirma o el tiempo de espera para que la cita se confirme expira.

El escenario 3 de la figura 5.9 muestra la transacción referente a la conversión de una ranura de tiempo libre remota en cita colectiva. Este tipo de transacción se considera la más compleja de realizar, debido a que involucra todos los componentes de la agenda cooperativa. El éxito de esta transacción depende de diferentes factores:

1. que algún miembro del grupo de trabajo no haya cambiado previamente el valor de la ranura de tiempo libre remota;

2. que el colaborador local tenga todos los privilegios de acceso sobre la ranura de tiempo

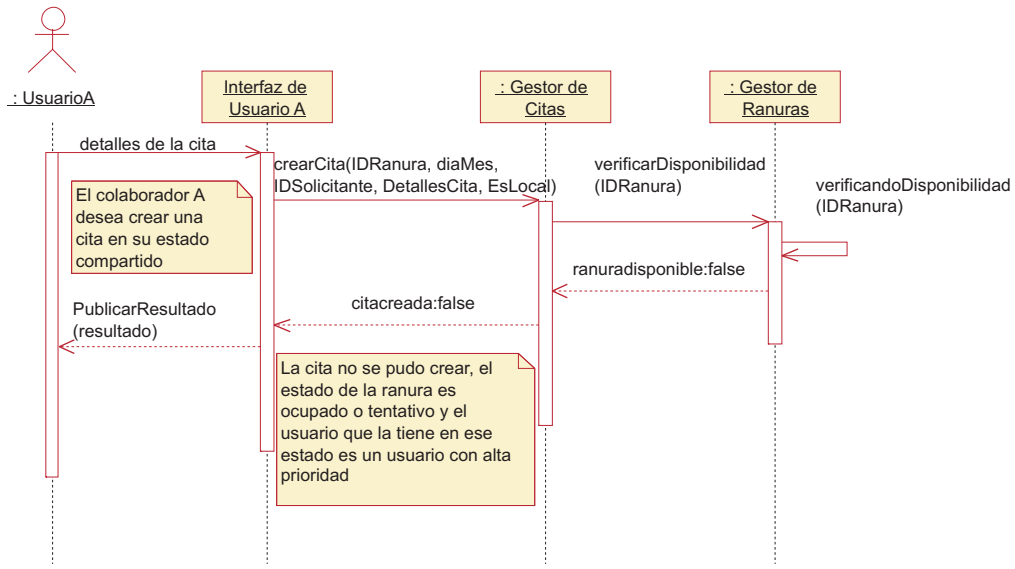


Figura 5.8: Escenario 2: Ocurrencia de una falla en la trasacción referente a la creación de una cita colectiva sobre una ranura de tiempo libre local

libre remota;

3. y que no existan fallas en la red.

La transacción referente a la actualización de una réplica del estado compartido sigue el mismo proceso referente a la creación de una cita colectiva sobre una ranura de tiempo libre remota. La diferencia radica en que la transacción referente a la actualización se realiza cuando ocurre alguna de las operaciones de crear, modificar, reservar y cancelar una cita colectiva y no simplemente cuando se crea una cita colectiva.

El escenario 4 de la figura 5.10 muestra la transacción referente a la solicitud de una réplica del estado compartido. Este tipo de transacción se considera como una operación de lectura, ya que no se realiza ningún cambio al estado compartido. Una falla potencial que puede ocurrir en este tipo de transacción se da cuando los sitios donde se encuentra almacenada dicha réplica, no estén en línea en el momento de realizar la transacción referente a la solicitud de una réplica del estado compartido o que el colaborador solicitante no sea un miembro valido dentro del grupo de trabajo (Escenario 5, figura 5.11).

El escenario 6 de la figura 5.12 muestra la transacción referente a la actualización de una réplica del estado compartido en un sitio remoto. En esta transacción, el colaborador local realiza modificaciones al estado compartido. Por lo tanto, el Servicio de Actualización local envía dichas modificaciones su par remoto, de esta forma el colaborador remoto conozca los cambios que realizó su colega. Debido a que pueden existir más de una réplicas del estado compartido almacenadas en diferentes sitios remotos, entonces la transacción de actualización se repite el numero de réplicas que existan de dicho estado.

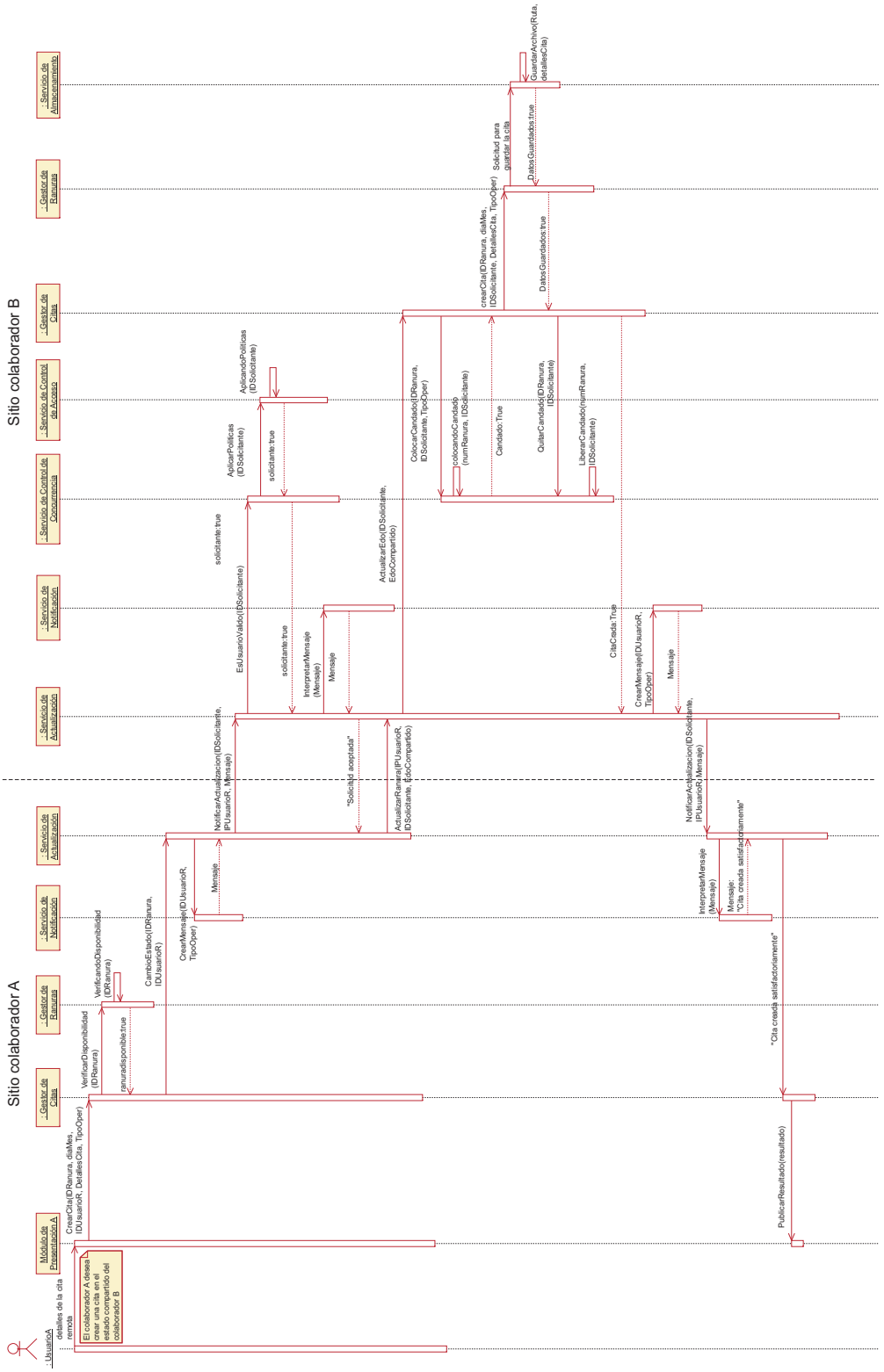


Figura 5.9: Escenario 3: El colaborador local convierte una ranura de tiempo libre remota en cita colectiva

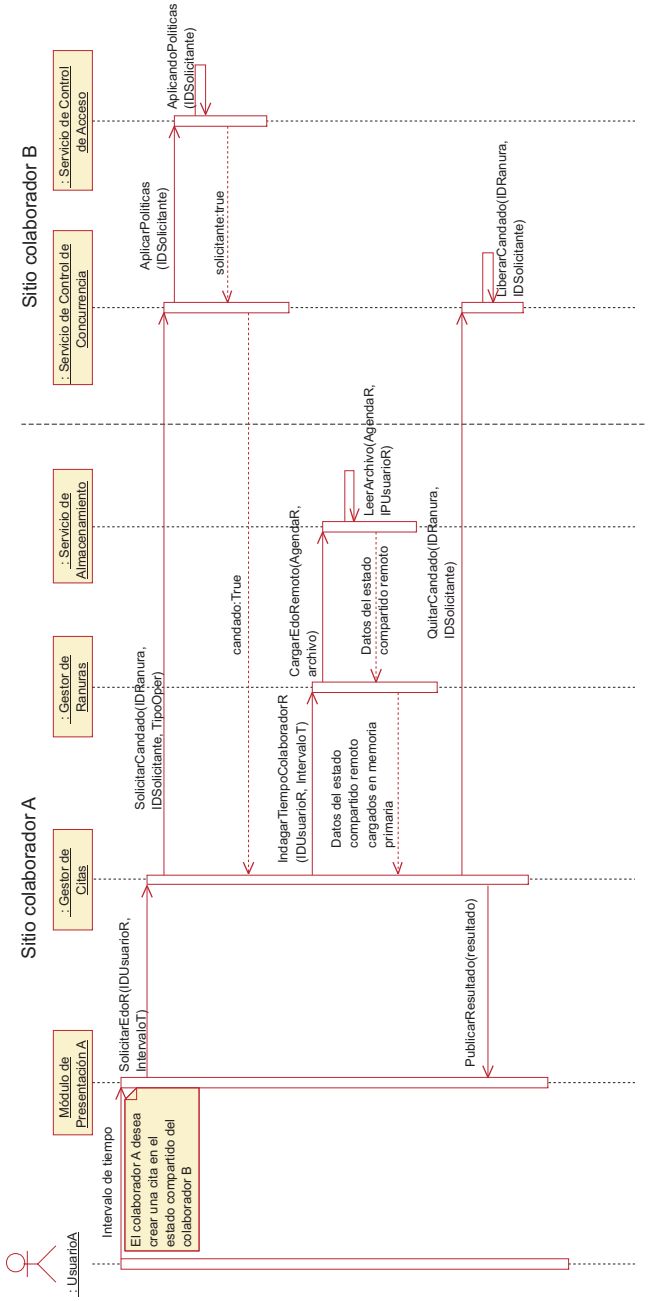


Figura 5.10: Escenario 4: Diagrama de secuencia cuando el sitio de un miembro del grupo de trabajo solicita una réplica del estado compartido

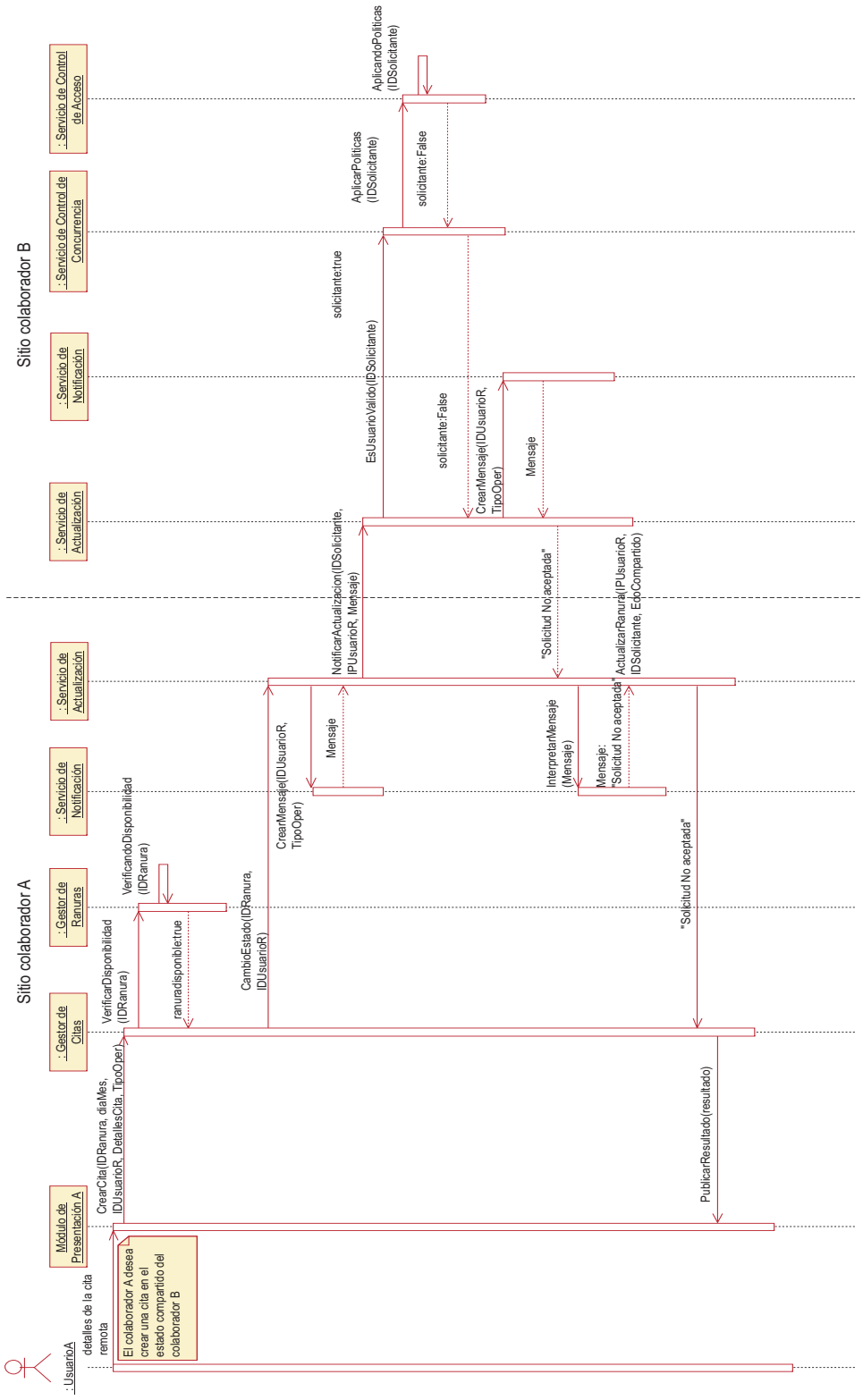


Figura 5.11: Escenario 5: Ocurrencia de una falla cuando se realiza una transacción referente a la solicitud de una réplica del estado compartido

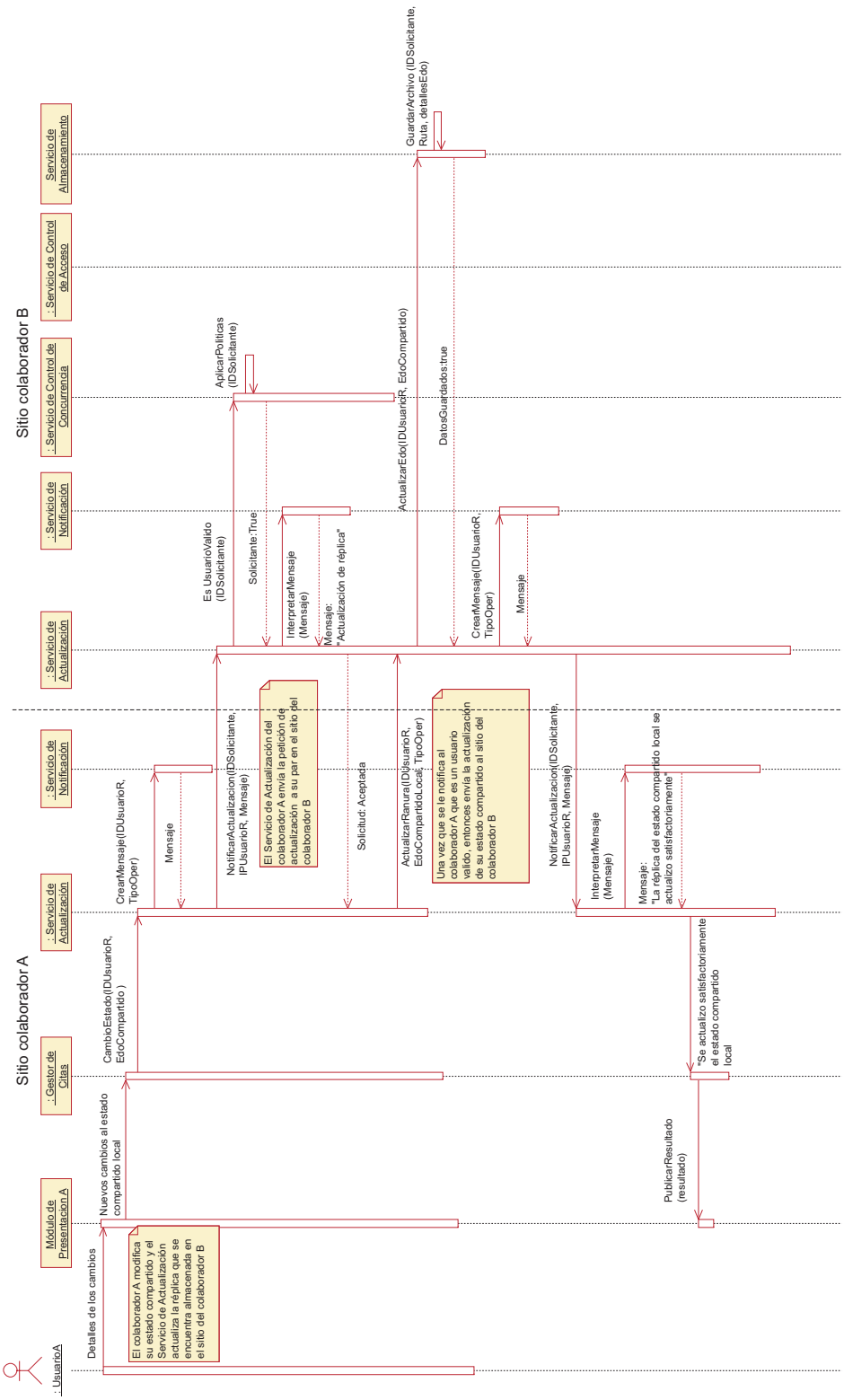


Figura 5.12: Escenario 6: Actualización de una réplica del estado compartido en un sitio remoto

Capítulo 6

Pruebas y resultados

En este capítulo se muestran los resultados obtenidos de las pruebas a la agenda cooperativa. Primero se muestra el esquema de distribución completo de la agenda (cf. sección 6.1) en términos de distribución de componentes y de estados.

Después de describir el esquema de distribución de los componentes y los estados de la agenda cooperativa, se prosigue con las pruebas realizadas al aspecto de base (cf. sección 6.2). Como se sabe el aspecto de base realiza las funciones básicas de la agenda, i.e., crear, modificar, cancelar, reservar y confirmar una cita. Acontinuación, se describen las pruebas de comunicación entre componentes de la agenda cooperativa mediante la tecnología de *servlets* (cf. sección 6.3).

Se concluye este capítulo con la descripción de las pruebas de flexibilidad (cf. sección 6.4), de los aspectos no funcionales, i.e., los servicios de Control de Concurrencia, Control de Acceso, Actualización y Almacenamiento.

6.1. Esquema de distribución de la agenda cooperativa

La figura 6.1 muestra la arquitectura general de la agenda cooperativa. Cada uno de los componentes que conforman la agenda cooperativa, i.e., el núcleo de la aplicación y los aspectos no funcionales están duplicados en los diferentes sitios de los colaboradores. Además, la utilización de Java como plataforma de desarrollo permite que la agenda cooperativa pueda ejecutarse en múltiples sistemas operativos (e.g., Linux, Mac OS o Windows).

En cuanto a los estados de la agenda cooperativa, el estado privado (citas privadas) sólo está almacenado en el sitio del colaborador propietario y los demás miembros de sus grupos de trabajo no tienen acceso a dicho estado. Por el contrario, los miembros del grupo de trabajo si tienen acceso al estado compartido de la agenda cooperativa (i.e., citas colectivas y ranuras de tiempo libre), lo cual implica hacer uso de mecanismos de control de concurrencia para mantener su consistencia. Además, el estado compartido está replicado en los sitios de los colaboradores concernidos. Por lo tanto, también se implementaron los mecanismos *pull* y *push* para realizar la actualización de las réplicas del estado compartido.

Para establecer la comunicación entre los componentes replicados de la agenda cooperativa

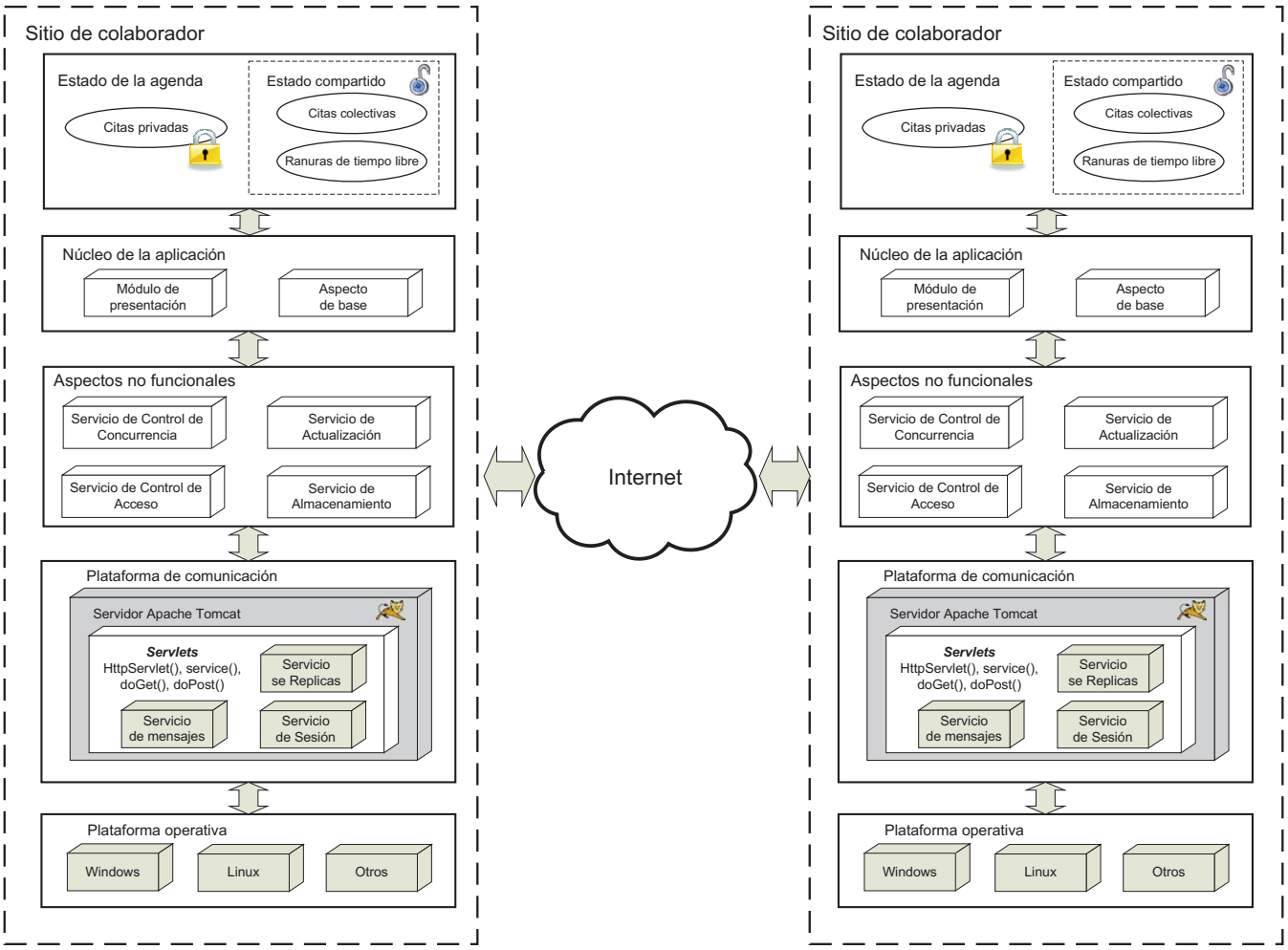


Figura 6.1: Arquitectura de los componentes y de los estados de la agenda cooperativa

se utilizaron *servlets*, los cuales son parte de la tecnología Java. Por lo tanto, cada sitio está compuesto por un servidor Tomcat [Apache-Tomcat, 2008], donde están alojados los *servlets*. Entre otras funcionalidades los *servlets* permiten la comunicación entre los diferentes aplicaciones y el envío de información mediante archivos. Un archivo se utiliza para guardar la infomación referente a los cambios realizados al estado compartido. Así, esta información se envía a los sitios remotos para actualizar la réplica del estado compartido que almacenan, con el fin de evitar las incongruencias en dicho estado.

Como se mencionó en la sección Arquitectura del Sistema (cf. sección 5.1), la agenda cooperativa tiene una arquitectura híbrida replicada, i.e., los sus componentes están replicados en los diferentes sitios de los colaboradores, mientras que los estados se mantienen semireplicados. A continuación, se mencionan las ventajas y desventajas de utilizar una arquitectura híbrida replicada como criterio de solución.

6.1.1. Ventajas y desventajas de la arquitectura con componentes replicados y estados semireplicados

El uso de componentes replicados permite crear múltiples instancias agenda cooperativa, cada una de las cuales se instala en los diferentes sitios de los colaboradores. Estos sitios deben de tener instalados los siguientes programas para poder ejecutar una instancia de la agenda cooperativa: 1) el jdk1.6.x, el cual contiene el interprete de Java, 2) aspectj1.5, que contiene el interprete de los aspectos, 3) el servidor Apache Tomcat 6.0 para la comunicación entre las diferentes instancias de la agenda cooperativa y 4) el paquete `servlet-api.jar` para la ejecución de los *servlets*; este último viene en el mismo paquete de Tomcat. Por otra parte, los estados de la agenda cooperativa semireplicados preservan la privacidad de las citas privadas de cada colaborador y proveen de disponibilidad del estado compartido mediante réplicas en los diferentes sitios remotos.

A continuación, se listan las ventajas y desventajas de utilizar los componentes de la agenda cooperativa replicados:

1. Mejora el tiempo de respuesta cuando se realiza una operación sobre los estados de la agenda cooperativa, ya que el procesamiento de una operación se realiza localmente. La operación más compleja que realiza la agenda cooperativa es la actualización de las réplicas del estado compartido.

2. Aumenta la disponibilidad en las peticiones de operaciones remotas, debido a que dichas peticiones de los colaboradores no pasan por un servidor central, sino que son enviadas directamente al sitio remoto.

3. La falla en un sitio remoto no afecta la interacción entre los miembros del grupo, debido a que cada sitio está comunicado con todos los demás.

4. Existe independencia de recursos, ya que cada sitio cuenta con su propio ambiente de ejecución para la instancia de agenda cooperativa.

Por otro lado, las desventajas de utilizar los componentes de la agenda cooperativa replicados son:

1. La latencia de red, i.e., los retardos en la transmisión de información a través de la red, sin embargo, un sitio no necesita obtener datos en tiempo real, por lo tanto este problema se puede tolerar.

2. Requiere de mecanismos complejos para mantener la consistencia del estado compartido, debido al acceso concurrente por parte de los colaboradores.

El estado compartido de la agenda cooperativa esta replicado en los sitios de los colaboradores. Por lo tanto las ventajas y desventajas de utilizar el estado compartido replicado son:

1. Aumenta la disponibilidad del estado compartido, ya que está replicado en los sitios remotos de los colaboradores, i.e., si un miembro del grupo de trabajo no esta conectado a la red en el momento en que se solicita una réplica de sus ranuras de tiempo libre, entonces, otro sitio que si contenga una réplica de dichas ranuras puede brindar el servicio.

2. Disminuye el tiempo de acceso a los elementos del estado compartido, debido a que el número de solicitudes que se realizan a un sitio remoto se reducen, i.e., para modificar una cita colectiva o una ranura de tiempo libre remota, la única solicitud que se realiza es la de actualización del estado compartido. Las modificaciones a una cita colectiva o a una ranura de tiempo libre se realizan sobre la réplica del estado compartido que está almacenada de manera local, sin embargo dichas modificaciones se deben actualizar en todas las réplicas del estado compartido.

Las desventajas de tener el estado compartido replicado se listan a continuación:

1. La actualización de las réplicas del estado compartido se vuelve un proceso complejo, ya que cualquier cambio debe ser propagado a todos los sitios que contienen réplicas. La misma desventaja se presenta cuando un miembro del grupo de trabajo ha estado fuera de línea por un determinado tiempo, durante el cual dicho miembro ha realizado cambios al estado compartido (i.e., ha cambiado el valor de una cita colectiva o ha creado citas en sus ranuras de tiempo libre o en las de sus colegas) u otros miembros del grupo de trabajo han creado citas que lo conciernen. Por lo tanto, las réplicas del estado compartido deben ser actualizadas para evitar posibles incoherencias en dicho estado.

2. Las políticas de control de acceso no garantizan que se otorguen derechos de acceso sobre el estado compartido a aquellos colaboradores que realmente las necesiten, debido a que el derecho de acceso se disputa entre todos los miembros del grupo de trabajo.

6.2. Pruebas del aspecto de base

Las operaciones que un colaborador puede realizar sobre el estado compartido son: crear, modificar, consultar, reservar y cancelar una cita, además de indagar el tiempo disponible de otro colaborador. Un colaborador utiliza dichas operaciones para modificar el valor (i.e., de libre a ocupada o tentativa) de las ranuras de tiempo libre de sus colegas. Por lo tanto, el caso más sencillo se presenta cuando el colaborador realiza operaciones sobre los elementos de su estado privado, ya que no afectan el estado compartido de la agenda cooperativa. Cuando un colaborador realiza operaciones sobre su estado privado, se dice que solo está utilizando la funcionalidad básica de la agenda, i.e., el aspecto de base. En esta sección se describe cómo un colaborador realiza modificaciones al estado privado de la agenda cooperativa.

La figura 6.2 muestra la pantalla principal del módulo de presentación de la agenda cooperativa (cf. sección 5.1.1). El módulo de presentación recibe las operaciones provenientes

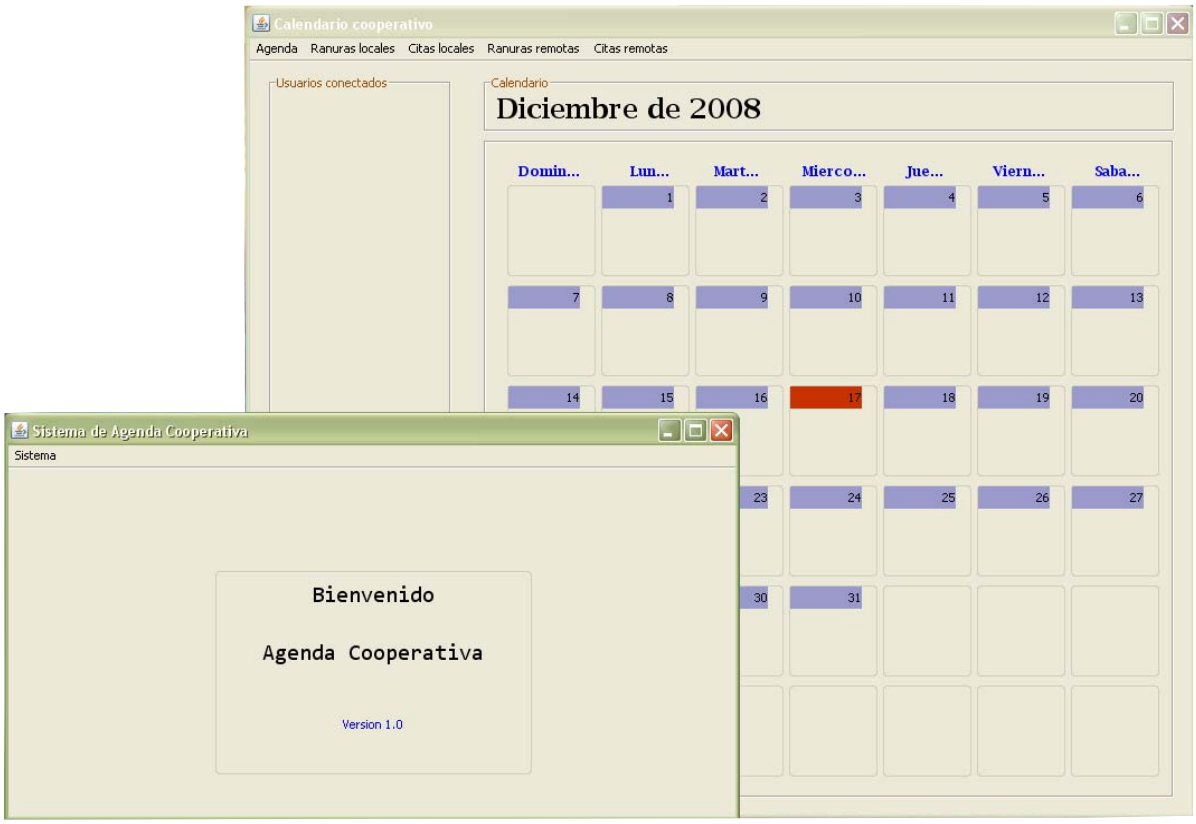


Figura 6.2: Módulo de presentación de la agenda cooperativa

de los dispositivos físicos y las transforma en eventos de la interfaz lógica, con el fin de que puedan ser procesados por los demás módulos de la agenda cooperativa. Por medio del módulo de presentación, el colaborador interactúa con la agenda cooperativa.

6.2.1. Manipulación de citas

Dependiendo del ámbito de la operación que se va a realizar (i.e., local o remota), el aspecto de base puede utilizar la funcionalidad adicional que otorgan los aspectos no funcionales. La funcionalidad del aspecto de base permite modificar el valor de los elementos del estado privado y no se necesita la intervención de la funcionalidad adicional. Sin embargo, para realizar una operación sobre un elemento del estado compartido si se necesita de la intervención de la funcionalidad adicional de la agenda cooperativa, ya que un cambio al estado compartido debe ser propagado a los sitios que contienen réplicas de dicho estado.

Un colaborador crea citas sobre elementos de su estado privado o del estado compartido dependiendo de sus necesidades, e.g. un colaborador crea citas privadas con miembros de sus grupos de trabajo o con personas ajenas a dichos grupos y citas colectivas que conciernen a varios miembros de sus grupos de trabajo.

A continuación se describe el funcionamiento del aspecto de base, cuando un colaborador realiza operaciones sobre su estado privado.

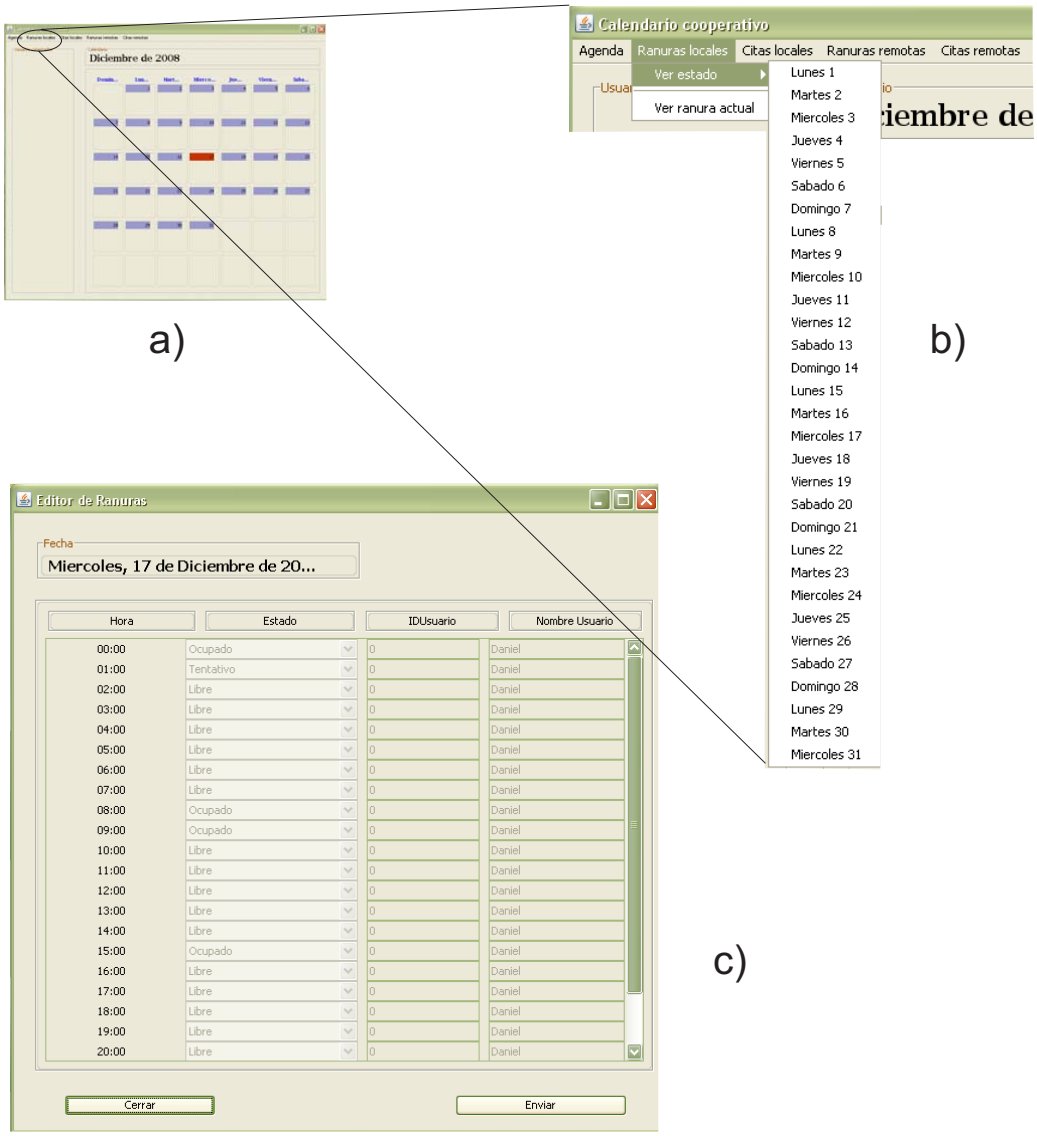


Figura 6.3: Operación de consulta al estado privado del colaborador propietario

La figura 6.3 muestra los resultados de la ejecución de una transacción referente a la consulta del estado privado del colaborador propietario de dicho estado (operación de lectura). Primero el módulo de presentación muestra una ventana que contiene el calendario del mes actual (ver figura 6.3 Ref. a). En esta ventana se encuentra una barra de menús para realizar las operaciones disponibles en la agenda cooperativa. Al seleccionar la opción `Ranuras locales` se despliega otro menú que contiene todos los días del mes en cuestión (ver figura 6.3 Ref. b). Al elegir cualquiera de estos días se habilita otra ventana que muestra el

contenido de las citas privadas creadas en el día seleccionado (ver figura 6.3 Ref. c); asimismo se muestra el valor de dichas citas (i.e., ocupada, tentativa o libre).

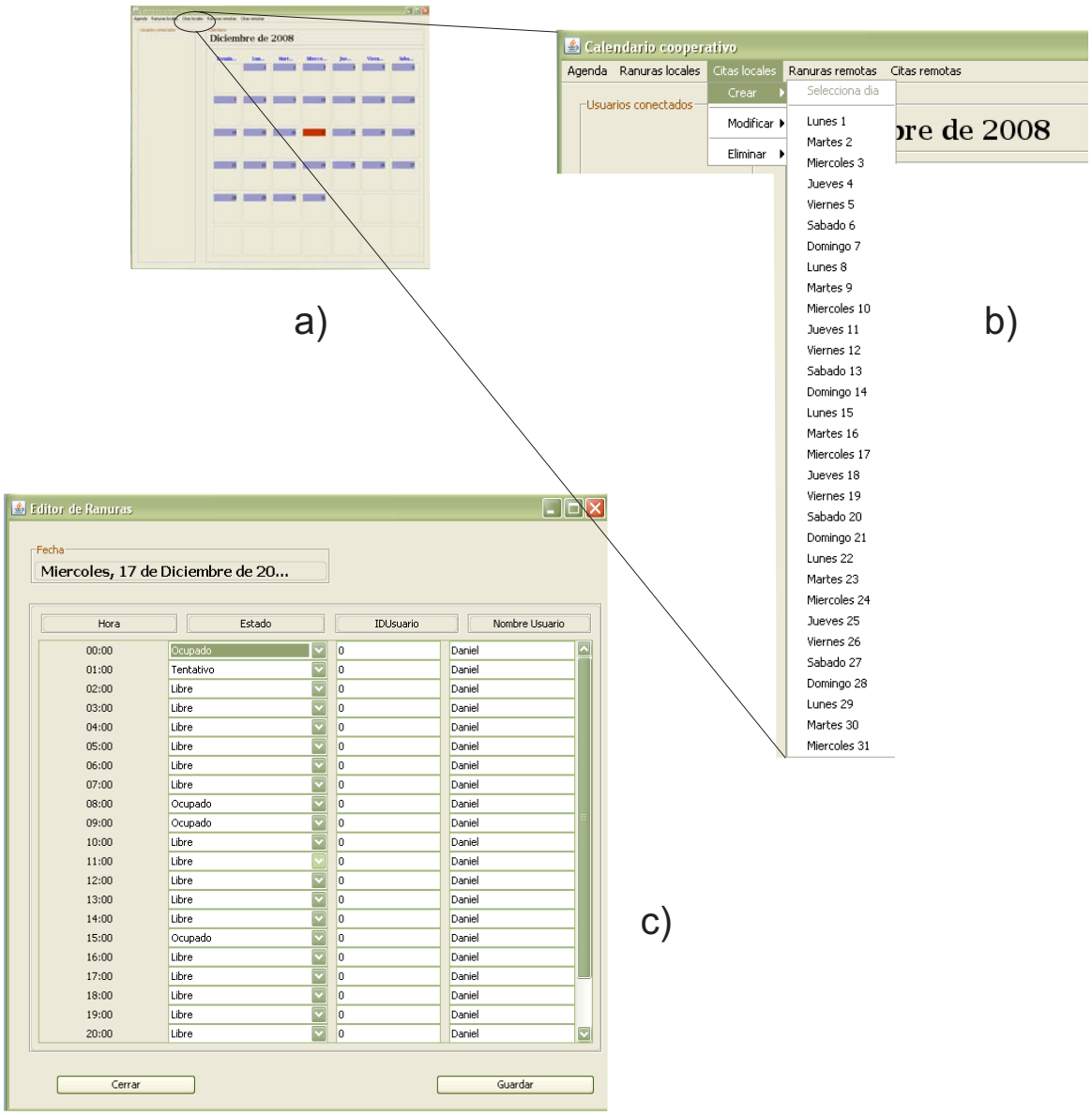


Figura 6.4: Resultados de la transacción referente a la creación de una cita privada

Las operaciones de escritura en el estado compartido de la agenda cooperativa (e.g., crear, modificar, reservar y cancelar una cita) se consideran más complejas que la operación de consulta, ya que requieren: 1) más tiempo de dedicación por parte del colaborador, ya que este último debe dar los detalles de la cita que se va a crear y 2) más tiempo de procesamiento de computadora ya que necesitan más componentes de la agenda cooperativa para ser procesadas

(e.g. el Servicio de Almacenamiento y el Gestor de Colaboradores y Grupos). En la figura 6.4 se muestra la forma en que son procesadas las operaciones de escritura en la agenda cooperativa.

De manera análoga a la operación de consulta, la barra de menús contiene la opción `Citas locales`. Al seleccionar dicha opción, se habilitan tres submenús (ver figura 6.4 Ref. b):

1. Opción `Crear`: habilita una lista de todos los días del mes en cuestión. Al seleccionar un día, se habilita otra ventana que muestra a detalle el conjunto de ranuras de tiempo de ese día (ver figura 6.4 Ref. c). Esta última ventana permite al colaborador crear citas privadas en el día seleccionado utilizando una o varias ranuras de tiempo libre. Cuando el colaborador presiona el botón `Guardar`, los datos de las citas creadas son almacenados por el Servicio de Almacenamiento en un archivo de texto.

   Para realizar una transacción referente a la reservación de una cita, los pasos son similares a la transacción referente a la creación una cita. La diferencia entre ambas transacciones es la siguiente: en una transacción referente a la creación de una cita, la ranura de tiempo libre cambia su valor a `ocupada`, mientras que para una transacción referente a la reservación de una cita, el valor de la ranura de tiempo libre cambia a `tentativa` asignándole un tiempo determinado a la ranura de tiempo, durante el cual se debe confirmar la cita ya que de no hacerse, el valor `tentativa` de la ranura de tiempo cambiará a `libre`. Dado que las transacciones referentes a la cración y reservación de una cita son muy similares, entonces se reutiliza la misma interfaz.

2. Opción `Modificar`: también habilita una lista de todos los días del mes en cuestión. Para realizar posibles modificaciones a una cita que se creo en cualquiera de esos días, se requiere seleccionar un día especifico para habilitar otra ventana que permite modificar la cita (ver figura 6.5 Ref. c). A diferencia de la ventana para crear citas privadas, la ventana para modificar una cita es más sencilla. Para fines prácticos y evitar distracciones al colaborador, no se muestra el detalle de todas las citas privadas del día seleccionado, sino únicamente el detalle de la cita que se va a modificar;

3. opción `Cancelar`: igual que las opciones anteriores, habilita una lista de todos los días del mes en cuestión. Cuando se selecciona un día especifico, se habilita una ventana que permite cancelar una cita (ver figura 6.6 Ref. c). Las ventanas que habilitan las opciones `Modificar` y `Cancelar` son similares, sin embargo, existe una diferencia que es la siguiente: la ventana que habilita la opción `Cancelar` tiene deshabilitadas las funciones que permiten modificar la cita y solo se muestra habilitado un botón `Cancelar`. Cuando el colaborador presiona dicho botón, la cita seleccionada se cancela.

Todas las operaciones de escritura mencionadas utilizan al Servicio de Almacenamiento, el cual guarda persistentemente los cambios realizados a las ranuras de tiempo.

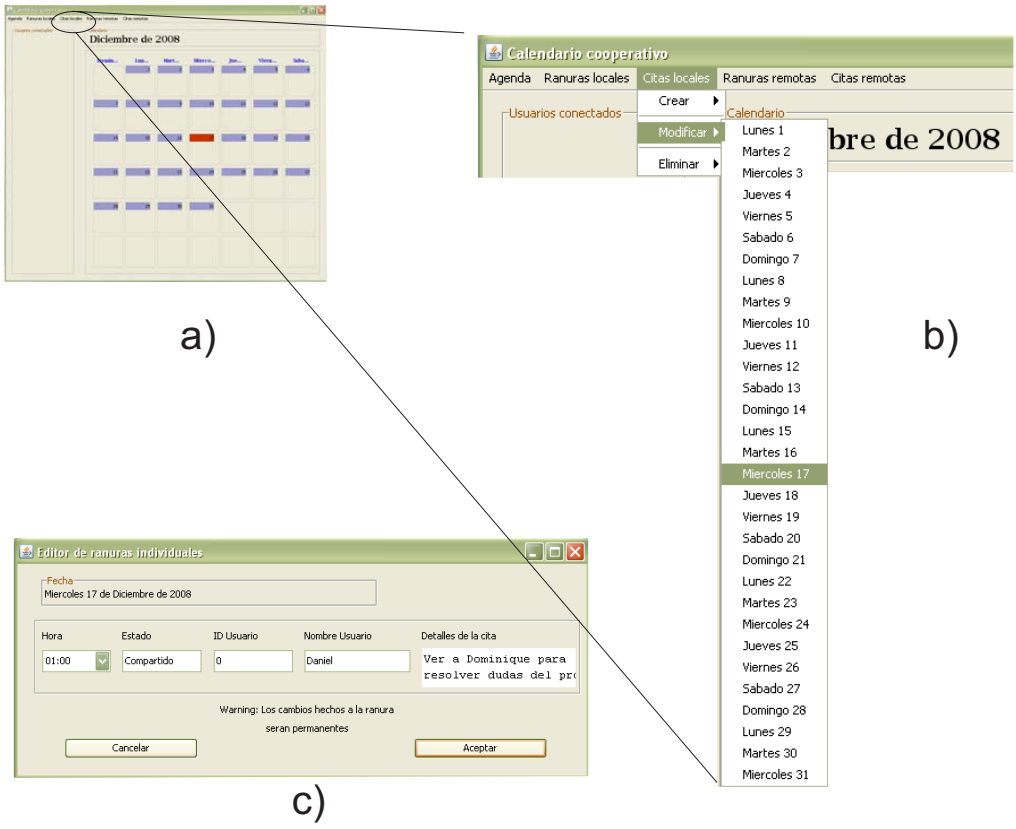


Figura 6.5: Resultados de la transacción referente a la modificación de una cita privada

6.3. Pruebas de comunicación mediante *Servlets*

Como se mencionó en la sección 4.5.2, se utilizó la tecnología de *servlets* como propuesta de solución para compartir e informar los cambios realizados al estado compartido de la agenda cooperativa. Las características más importantes por las que se eligieron los *servlets* son: 1) no se limitan a una sola plataforma de desarrollo y ejecución, 2) se cargan dinámicamente al momento de ser invocados, 3) se ejecutan una sola vez y permanecen en memoria para atender futuras solicitudes, 4) permiten la interacción entre aplicaciones, ya que pueden atender múltiples peticiones al mismo tiempo, 5) permiten el envío y la recepción de archivos y 6) invocan o reenvían solicitudes de servicios a otros *servlets* que se ejecutan en el mismo servidor o en otro servidor remoto.

Una limitación de los *servlets*, para el desarrollo de la agenda cooperativa, es que sólo atienden a peticiones que provienen de aplicaciones Web (e.g., *browsers*, *Applets* o de otros *servlets*). Sin embargo, la agenda cooperativa es una aplicación *standalone*[1], no una aplicación

[1]Una aplicación *standalone* se asemeja a un programa convencional que se ejecuta en una estación de trabajo convencional, no necesariamente en un servidor.

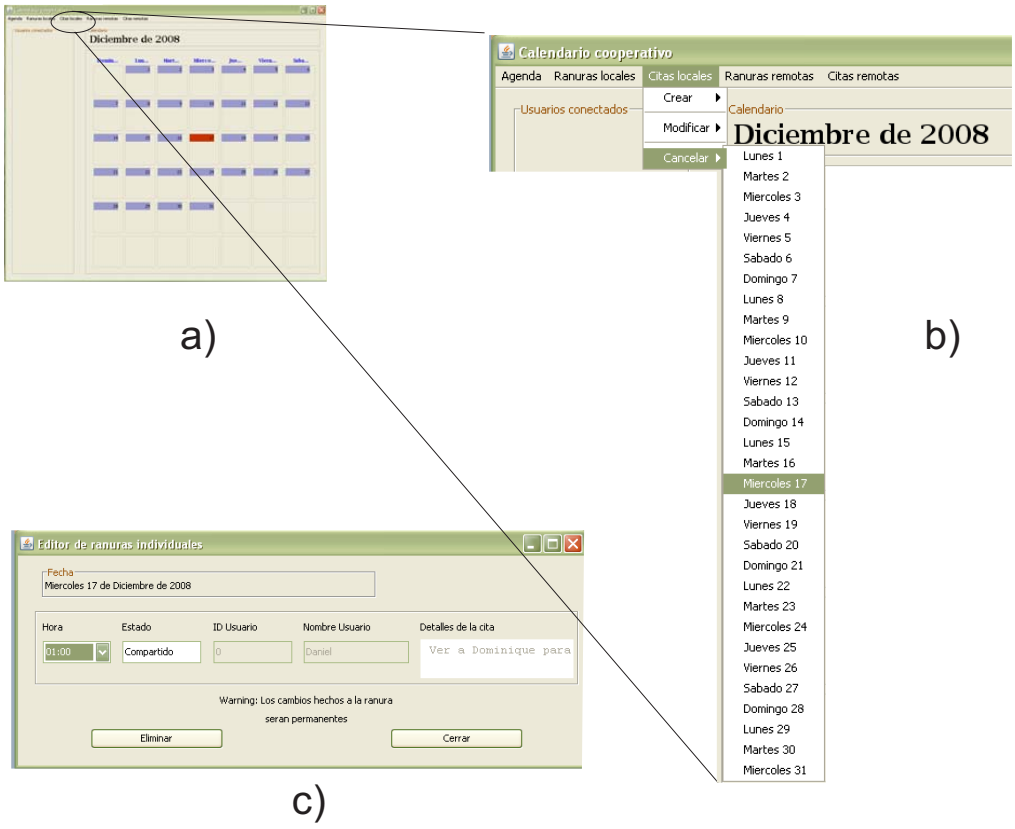


Figura 6.6: Resultados de la transacción referente a la cancelación de cita privada

Web. Para resolver dicha limitación, Java proporciona el paquete `java.net` el cual contiene las clases Java necesarias para implementar aplicaciones de red, i.e., permite crear las peticiones de la misma forma en que las hacen las aplicaciones Web.

Los principales recursos que el paquete `java.net` proporciona para permitir a una aplicación *standalone* hacer peticiones a los *servlets* son:

- una **URL** que permite localizar los recursos que se encuentran en la Web, e.g., los *servlets* que están almacenados en los servidores Tomcat, los cuales están localizados en los sitios de los colaboradores;
- la clase `HttpURLConnection` que da soporte al protocolo HTTP;
- la clase `OutputStream` que permite enviar un flujo de bytes a través de la red. En el flujo de bytes se envían las cabeceras del protocolo HTTP en forma de petición al *servlet*;
- la clase `InputStream` que permite recibir la respuesta proveniente del *servlet*.

Los *servlets* también implementan las clases `ServletOutputStream` y `ServletInputStream`, análogos a `OutputStream` e `InputStream` de `java.net`, para enviar y recibir flujos de bytes respectivamente. Estas clases son importantes ya que por medio de ellas se realizan las actualizaciones de las réplicas del estado compartido.

En la implementación de la agenda cooperativa se utilizaron tres tipos de *servlets*, los cuales realizan las siguientes tareas: 1) iniciar sesión con los sitios asociados a miembros de los grupos de trabajo, 2) solicitar una réplica de las ranuras de tiempo libre de los miembros del grupo y 3) actualizar las réplicas del estado compartido. Cabe mencionar que estos `servlets` están alojados en el servidor Tomcat, el cual a su vez está alojado en el sitio del colaborador.

Como se mencionó en la sección 4.5.2, un *servlet* necesita de una solicitud para poder realizar la tarea que le fue asignada y para poder envíar la respuesta adecuada. Por lo tanto, en los resultados de comunicación mediante los *servlets*, se muestra el código que realiza las peticiones a los *servlets* y el código de los *servlets* mismos.

1) *Servlet* de sesión

El *servlet* de sesión permite a un colaborador iniciar sesión con los demás miembros del grupo de trabajo. Si no ocurre ninguna falla (e.g., de red), entonces la respuesta esperada es la conexión entre los sitios remotos. En caso contrario se notifica al colaborador que los demás miembros del grupo de trabajo no están en línea. Si sólo hay algunos miembros del grupo de trabajo conectados, no todos, es con sus sitios que se establece la conexión. La ausencia de uno o varios miembros del grupo de trabajo no afecta la interacción entre los demás miembros del grupo, debido a que existe independencia de recursos, i.e., los componentes de la agenda cooperativa y el estado compartido están replicados en los sitios de los miembros del grupo que están conectados en ese momento.

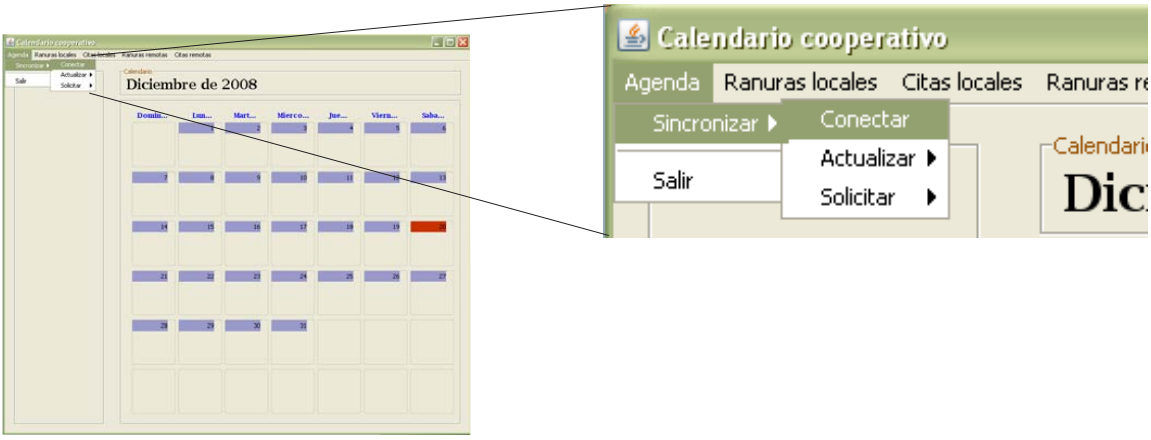


Figura 6.7: Un miembro del grupo de trabajo realiza una solicitud de conexión con otras instancias remotas de la agenda cooperativa

Como se mencionó anteriormente, la interacción colaborador/agenda se produce mediante el módulo de presentación. Por lo tanto, la figura 6.7 muestra la forma en que un colaborador

solicita al módulo de presentación, que se indague que miembros del grupo de trabajo están disponibles.

```java
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.net.HttpURLConnection;
import java.net.URL;

public class SolicitarSesion{
  private static String IP_user;
  private static long puerto;
  private static String nombre_user;
  private static final String CLASSNAME = "SolicitarSesion";

  public SolicitarConexion( String IP, String nombreUser, long puerto ) throws Exception{
    this.IP_user = IP;
    this.nombre_user = nombreUser;
    this.puerto = puerto;
    final String METHODNAME = "main";
    connect(false);
    log ("Se sale de la clase: " + CLASSNAME + "." + METHODNAME);
  }

  /**Se crea el método para enviar una solicitud GET al servlet de sesión*/
  private static void connect(boolean GET_METHOD) throws Exception{

    /**Se crea la URL para buscar el recurso en la red*/
    String urlString="http://"+ IP_user + ":" + puerto + "/Agenda/Sesion";
    String parameters="operacion=sesion";

    if(GET_METHOD)
      urlString += parameters;

    URL url = new URL(urlString);

    /**Se abre la conexión con el servlet*/
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setAllowUserInteraction(true);
    conn.setDoOutput(true);

    if (!GET_METHOD){
      /**Se envía la petición de lo que se quiere hacer*/
      OutputStream osw = (OutputStream) conn.getOutputStream();
    }

    /**Se espera por la respuesta y se envía a la interfaz de salida
     * para que se muestre al colaborador*/
    InputStream isw = (InputStream)conn.getInputStream();
    PrintWriter display = new PrintWriter(System.out);
    StringWriter sw = new StringWriter();

    int i = isw.read();

    for(;i!=-1;i=isw.read()) {
      sw.write(i);
    }

    /**Se cierra la conexión y se limpia el buffer de salida*/
    isw.close();
    conn.disconnect();
  }

  private static void log(String s) { System.out.println(s); }
}
```

a)

```java
import java.io.*;
import java.util.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Sesion extends HttpServlet {

  private static final long serialVersionUID = 1L;
  String[] DataUser = new String[3];
  String idUser;

  /**Se sobre escribe el método doPost de la clase HttpServlet para enviar
   * la respuesta al solicitate*/
  public void doPost(HttpServletRequest request, HttpServletResponse response)
                                        throws IOException, ServletException {
    String parametro;
    String parametros[];

    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()){
      parametro = (String)paramNames.nextElement();
      parametros = request.getParameterValues(parametro);
    }

    /**Se obtiene el ID del colaborador solicitante*/
    idUser = request.getRemoteAddr();

    /**Se obtiene los datos del colaborador receptor para construir el
     * mensaje de respuesta*/
    getDataUser( DataUser );
    /**Se envía de regreso el mensaje de respuesta. El mensaje que se
     * envía es el ID del colaborador receptor*/
    enviarMensaje(response);
  }

  public void enviarMensaje( HttpServletResponse response )
                  throws IOException, ServletException{
    byte[] msg = DataUser[1].getBytes();
    ServletOutputStream sos = response.getOutputStream();
    sos.write(msg);
    sos.flush();
    sos.close();
  }

  /**
   *Archivo que lee los datos del colaborador*/
  public void getDataUser( String[] DataUser ) {
    .
    .
    .
  }
}
```

b)

Figura 6.8: Código de la solicitud/respuesta para solicitar un inicio de sesión

El código para iniciar una sesión con un miembro del grupo se divide en dos partes: 1) la solicitud que realiza un colaborador a otro miembro del grupo (ver figura 6.8 Ref. a) y 2) la respuesta del *servlet* de `Sesión` si no ocurre ningún problema durante el envío de la solicitud. El mensaje de respuesta contiene el identificador del colaborador remoto que responde satisfactoriamente a la solicitud (ver figura 6.8 Ref. b). Este identificador permite saber qué miembros del grupo están disponibles en ese momento.

La figura 6.9 muestra los resultados del envío de una solicitud de inicio sesión. Cuando la solicitud tiene éxito, el mensaje de la repuesta contienen la dirección física (IP) del sitio remoto que respondió a la solicitud. Sin embargo, cuando no se hace contacto con algún miembro del grupo, entonces se lanza una excepción de Java que permite notificar al colaborador que uno de los miembros del grupo no está presente en la red (ver en la figura 6.9 Ref. a).

El módulo de presentación de la agenda cooperativa habilita el menú de los colaboradores remotos que respondieron a la solicitud de inicio de sesión (e.g., ver Sonia y Dominique, en la figura 6.9 Ref. b). Los colaboradores que aparecen habilitados en el menú de colaboradores

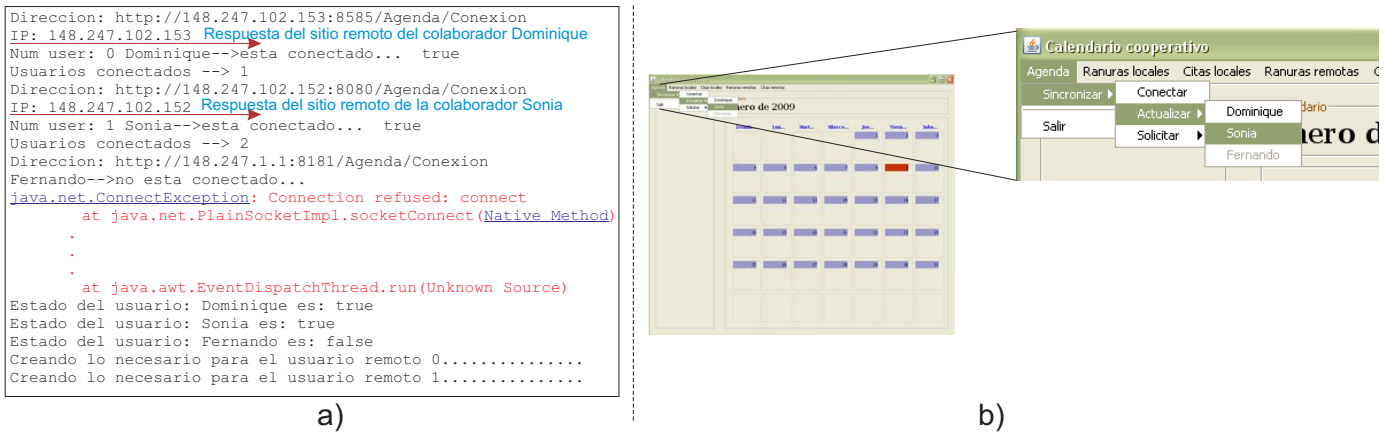


Figura 6.9: Resultado del *servlet* de sesión

remotos, permite conocer que colaboradores se encuentran disponibles en ese momento para solicitar réplicas de las ranuras de tiempo libre, asimismo permite al Servicio de Actualización actualizar las réplicas del estado compartido que está alojado en sus sitios. Cuando un miembro del grupo no está conectado a la red (e.g., ver Fernando en la figura 6.9 Ref. b), la actualización de la réplica del estado compartido que está alojada en su sitio se realiza hasta que dicho colaborador se conecte a la red.

2) *Servlet* de solicitud de réplica

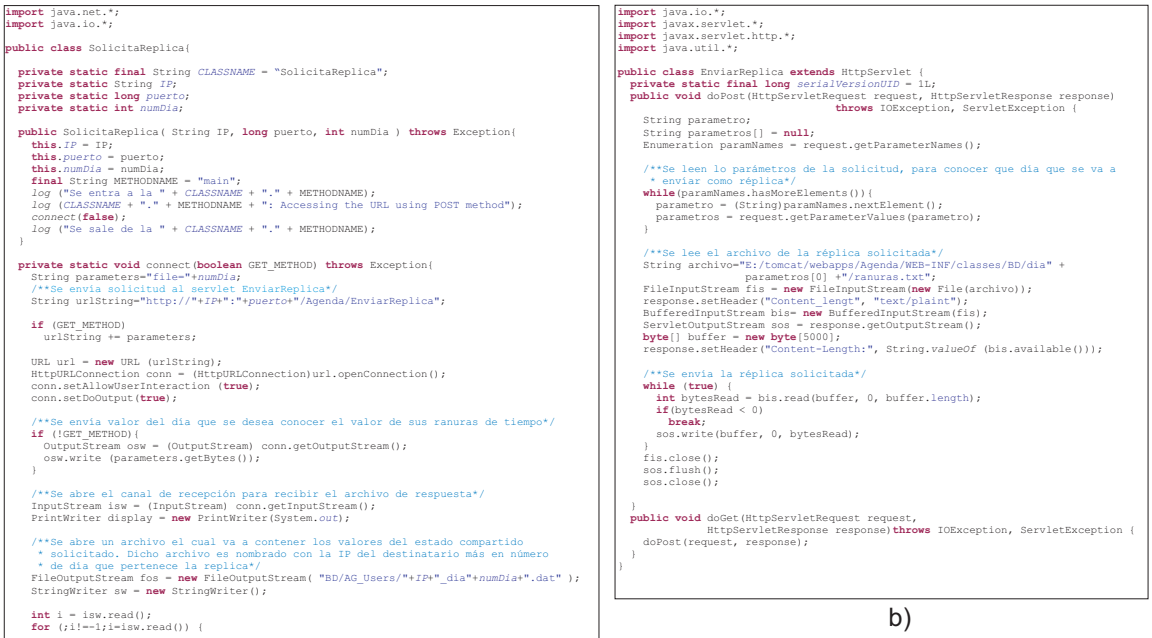
El *servlet* de solictud de réplica envía un archivo de texto que contiene los datos del estado compartido de la agenda cooperativa. Cuando un colaborador crea un grupo de trabajo, el servicio de actualización de la agenda cooperativa se encarga de obtener las réplicas de las ranuras de tiempo libre de los miembros que pertenecen al grupo. La solicitud se envía al *servlet* `EnviarReplica`, el cual envía en la respuesta un archivo de texto que contiene los datos de las ranuras tiempo libre del colaborador remoto.

Debido a que la solicitud de réplica se considera una operación de lectura, la única limitación que se presenta en esta solicitud es la posibilidad de ocurrencia de una falla en la red que impida el envío de la solicitud o de la respuesta.

El código para solicitar una réplica de las ranuras de tiempo libre de un miembro del grupo de trabajo está estructurado de la siguiente manera: 1) la solicitud que realiza el servicio de actualización cuando se crea un grupo de trabajo (ver figura 6.10 Ref. b) y 2) la respuesta que envía el *servlet* `EnviarReplica` (ver figura 6.10 Ref. b).

3) *Servlet* de actualización de réplica

La actualización de una réplica del estado compartido se lleva acabo mediante una analogía del método *push* en combinación con el método *pull* (ver sección 4.1.2). El colaborador local realiza modificaciones al estado compartido, en consecuencia el Servicio de



```java
import java.net.*;
import java.io.*;

public class SolicitaReplica{

  private static final String CLASSNAME = "SolicitaReplica";
  private static String IP;
  private static long puerto;
  private static int numDia;

  public SolicitaReplica( String IP, long puerto, int numDia ) throws Exception{
    this.IP = IP;
    this.puerto = puerto;
    this.numDia = numDia;
    final String METHODNAME = "main";
    log ("Se entra a la " + CLASSNAME + "." + METHODNAME);
    log (CLASSNAME + "." + METHODNAME + ": Accessing the URL using POST method");
    connect(false);
    log ("Se sale de la " + CLASSNAME + "." + METHODNAME);
  }

  private static void connect(boolean GET_METHOD) throws Exception{
    String parameters="file="+numDia;
    /**Se envia solicitud al servlet EnviarReplica*/
    String urlString="http://"+IP+":"+puerto+"/Agenda/EnviarReplica";

    if (GET_METHOD)
      urlString += parameters;

    URL url = new URL (urlString);
    HttpURLConnection conn = (HttpURLConnection)url.openConnection();
    conn.setAllowUserInteraction (true);
    conn.setDoOutput(true);

    /**Se envia valor del dia que se desea conocer el valor de sus ranuras de tiempo*/
    if (!GET_METHOD){
      OutputStream osw = (OutputStream) conn.getOutputStream();
      osw.write (parameters.getBytes());
    }

    /**Se abre el canal de recepción para recibir el archivo de respuesta*/
    InputStream isw = (InputStream) conn.getInputStream();
    PrintWriter display = new PrintWriter(System.out);

    /**Se abre un archivo el cual va a contener los valores del estado compartido
     * solicitado. Dicho archivo es nombrado con la IP del destinatario más en número
     * de dia que pertenece la replica*/
    FileOutputStream fos = new FileOutputStream( "BD/AG_Users/"+IP+"_dia"+numDia+".dat" );
    StringWriter sw = new StringWriter();

    int i = isw.read();
    for (;i!=-1;i=isw.read()) {

      sw.write(i);
      fos.write((byte) i);
    }

    String delimiter = sw.toString();
    display.println(delimiter);
    display.flush();

    /**Se cierra el canal de conexión y el archivo que contiene la réplica solicitada*/
    isw.close();
    fos.close();
    conn.disconnect();
  }

  private static void log(String s) { System.out.println(s); }
}
```

a)

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class EnviarReplica extends HttpServlet {
  private static final long serialVersionUID = 1L;
  public void doPost(HttpServletRequest request, HttpServletResponse response)
                                    throws IOException, ServletException {

    String parametro;
    String parametros[] = null;
    Enumeration paramNames = request.getParameterNames();

    /**Se leen lo parámetros de la solicitud, para conocer que dia que se va a
     * enviar como réplica*/
    while(paramNames.hasMoreElements()){
      parametro = (String)paramNames.nextElement();
      parametros = request.getParameterValues(parametro);
    }

    /**Se lee el archivo de la réplica solicitada*/
    String archivo="E:/tomcat/webapps/Agenda/WEB-INF/classes/BD/dia" +
                   parametros[0] +"/ranuras.txt";
    FileInputStream fis = new FileInputStream(new File(archivo));
    response.setHeader("Content_lengt", "text/plaint");
    BufferedInputStream bis= new BufferedInputStream(fis);
    ServletOutputStream sos = response.getOutputStream();
    byte[] buffer = new byte[5000];
    response.setHeader("Content-Length:", String.valueOf (bis.available()));

    /**Se envia la réplica solicitada*/
    while (true) {
      int bytesRead = bis.read(buffer, 0, buffer.length);
      if(bytesRead < 0)
        break;
      sos.write(buffer, 0, bytesRead);
    }
    fis.close();
    sos.flush();
    sos.close();

  }
  public void doGet(HttpServletRequest request,
            HttpServletResponse response)throws IOException, ServletException {
    doPost(request, response);
  }
}
```

b)

Figura 6.10: Código de la solicitud/respuesta para una solicitar una réplica del estado compartido

Actualización local comienza la transacción de actualización con su pares remotos. Primero, el Servicio de Actualización local envía a su par remoto un mensaje de notificación, en el cual se informa que se han realizado modificaciones al estado compartido (método *push*). Después, el Servicio de Actualización remoto responde con otra solicitud de actualización del estado compartido (metodo *pull*).

La principal motivación del diseño de la transacción referente a la actualización de una réplica del estado compartido, es que se evitan los problemas de restricciones que se tienen con el manejo del sistema de archivos en cuanto a la seguridad en la red, i.e., la transacción de actualización de una réplica del estado compartido necesita escribir un archivo de texto, que contiene los datos actualizados del estado compartido, en los sitios que almacenan réplicas de dicho estado.

Por lo tanto, para realizar la actualización de una réplica del estado compartido se hace uso de dos *servlets*: 1) el *servlet* `SolicitarActualizacion` (ver figura 6.11 Ref. a) que recibe el mensaje de notificación de los cambios al estado compartido, en forma de solicitud y 2) el

servlet `SolicitarActualizacion` que reenvía al *servlet* `EnviarActualizacion`[2] la solicitud que recibió para solicitarle el envío de los datos actualizados del estado compartido. De esta forma se actualiza la réplica del estado compartido que se encuentra almacenada en el sitio remoto.

```java
import java.net.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SolicitarActualizacion extends HttpServlet{
  private static final long serialVersionUID = 1L;
  String idUser;
  String mensaje = null;
  long port;
  String archivo;

  public void doPost(HttpServletRequest request, HttpServletResponse response)
                     throws IOException, ServletException {
    String parametro;
    String parametros[] = null;
    Enumeration paramNames = request.getParameterNames();

    /**Toma el valor del estado compartido que ha sido actualizado*/
    while(paramNames.hasMoreElements()){
      parametro = (String)paramNames.nextElement();
      parametros = request.getParameterValues(parametro);
    }

    /**Toma el valor de la IP del solicitante para reenviarle la petición que le envie
     * el archivo con los datos modificados del estado compartido*/
    idUser = request.getRemoteAddr();
    port = request.getRemotePort();
    archivo = "file="+parametros[0];
    String urlString="http://"+idUser+":8080/Agenda/EnviarActualizacion";
    System.out.println( "Direccion: " + urlString );
    URL url = new URL (urlString);
    HttpURLConnection conn = (HttpURLConnection)url.openConnection();
    conn.setAllowUserInteraction (true);
    conn.setDoOutput(true);

    OutputStream osw = (OutputStream) conn.getOutputStream();
    osw.write (archivo.getBytes());

    /**Abre el canal de recepción para recibir el archivo con la actualización
     * y la guarda en la carpeta ../BD/Actualizaciones*/
    InputStream isw = (InputStream) conn.getInputStream();
    PrintWriter display = new PrintWriter(System.out);
    FileOutputStream fos = new FileOutputStream(
                    "E:/tomcat/webapps/Agenda/WEB-INF/classes/BD/Actualizaciones"+parametros[0] );
    StringWriter sw = new StringWriter();
    int i = isw.read();
    for (;i!=-1;i=isw.read()) {
      sw.write(i);
      fos.write((byte) i);
    }
    //String delimiter = sw.toString();
    //display.println(delimiter);
    display.flush();
    isw.close();
    fos.close();
    conn.disconnect();

    /**Regresa un mensaje de que la actualización se ha recibido con éxito*/
    enviarMensaje(response);
  }

  public void enviarMensaje( HttpServletResponse response ) throws IOException, ServletException{
    mensaje = "Archivo guardado...";
    byte[] msg = mensaje.getBytes();
    ServletOutputStream sos = response.getOutputStream();
    sos.write(msg);
    sos.flush();
    sos.close();
  }
}
```

a)

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class EnviarActualizacion extends HttpServlet{
  private static final long serialVersionUID = 1L;
  String idUser;
  String mensaje = null;

  public void doPost(HttpServletRequest request, HttpServletResponse response)
                     throws IOException, ServletException {
    String parametro;
    String parametros[] = null;
    Enumeration paramNames = request.getParameterNames();

   /**Lee los parametros de la solicitud para que se envie la
    * actualización correcta*/
    while(paramNames.hasMoreElements()){
      parametro = (String)paramNames.nextElement();
      parametros = request.getParameterValues(parametro);
    }

    /**Se lee el archivo que contiene la actualización del estado compartido*/
    String archivo="E:/tomcat/webapps/Agenda/WEB-INF/classes/BD/dia" +
                   parametros[0] +"/ranuras.txt";
    System.out.println( "archivo a accesar: " + archivo );
    FileInputStream fis = new FileInputStream(new File(archivo));
    response.setHeader("Content_lengt", "text/plaint");
    BufferedInputStream bis= new BufferedInputStream(fis);
    ServletOutputStream sos = response.getOutputStream();
    byte[] buffer = new byte[5000];
    response.setHeader("Content-Length:", String.valueOf (bis.available()));

    /**Se envía el archivo con la actualización*/
    while (true) {
      int bytesRead = bis.read(buffer, 0, buffer.length);
      if (bytesRead < 0)
        break;
      sos.write(buffer, 0, bytesRead);
    }
    fis.close();
    sos.flush();
    sos.close();
  }
```

b)

Figura 6.11: Código de la solicitud/respuesta para actualizar una réplica del estado compartido

[2]El *servlet* `SolicitarActualizacion` está alojado en el sitio que almacena la réplica del estado compartido que se va a actualizar, mientras que el *servlet* `EnviarActualizacion` está alojado en el sitio del colaborador que inicia la actualización del estado compartido.

6.4. Pruebas de flexibilidad

En esta sección se describen los resultados de la funcionalidad adicional que proporcionan los aspectos no funcionales de la agenda cooperativa, i.e., los servicios de control de concurrencia, de control de acceso, de actualización y de almacenamiento. Para mostrar los resultados, primero se aborda la forma en que los cortes en punto, avisos y puntos de unión se entretejen con el código de la funcionalidad básica, así como los resultados obtenidos en la ejecución de la agenda cooperativa una vez que se añaden los aspectos no funcionales.

6.4.1. Servicio de Control de Concurrencia

En el servicio de control de concurrencia se implementó la técnica de candados pesimistas para bloquear el acceso concurrente al estado compartido de la agenda cooperativa, i.e., cada que se realiza una operación de creación, modificación, cancelación, reservación de una cita colectiva, se añade un candado a las ranuras de tiempo que se van a utilizar en dicha operación.

Específicamente, el servicio control de concurrencia actúa en el momento en que detecta que se va a realizar cualquiera de las operaciones antes mencionadas. La motivación es otorgar al colaborador solicitante total acceso a las ranuras de tiempo que solicita, de tal forma que cualquier otra operación, sobre las mismas ranuras de tiempo, queda en estado de espera hasta que el colaborador solicitante libere los candados de las ranuras de tiempo. De esta forma se asegura la integridad del estado compartido.

Para la implementación del servicio de control de concurrencia se utilizó el corte en punto `Candados`. Los puntos de unión de este corte se ven reflejados en el código Gestor de Citas, ya que este último es el que se encarga de recibir todas las operaciones que se van a realizar sobre el estado compartido (ver figura 6.12).

La figura 6.12 muestra en qué parte del código del aspecto de base actúan los puntos de únion. Para comprender lo que está sucediendo, se analiza la línea de código `GC.GR.crearCita( GC.GR.AG[diaModif] )`. Esta línea de código indica que el gestor de citas (`GC`) invoca a la función `crearCita()` del gestor de ranuras (`GR`) para que almacene temporalmente en memoria primaria una cita que se va a crear. La flecha naranja indica específicamente la parte del código del aspecto de base en que los avisos `before` y `after` están actuando. Dichos avisos están implementados dentro del aspecto `ServicioConcurrencia` (ver figura 6.13).

El aspecto `ServicioConcurrencia` implementa el cuerpo de los avisos y el módulo la configuración del Servicio de Control de Concurrencia, i.e., "Cómo" van a actuar el Servicio de Control de Concurrencia dentro del aspecto de base de la agenda cooperativa. El aviso `before` únicamente coloca un candado en una ranura de tiempo antes de iniciarse una operación (e.g. crear, modificar, cancelar, reservar una cita colectiva, e indagar el tiempo de un colaborador). El aviso `after` libera el candado de dicha ranura. El módulo de configuración indica el lugar en donde se va a añadir la funcionalidad adicional, i.e., antes y después de que se va a realizar una operación en el estado compartido de la agenda cooperativa (ver figura 6.14).

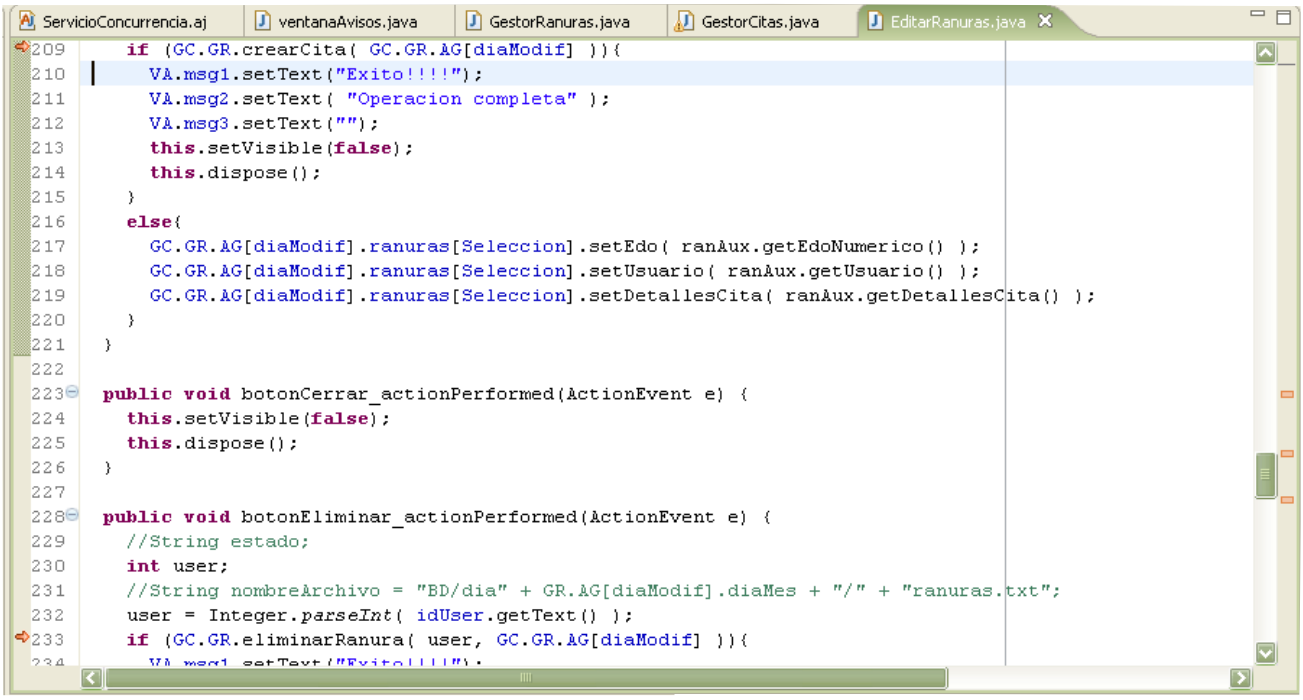


Figura 6.12: Puntos de unión donde va a actuar el Servicio de Control de Concurrencia

Los resultados de la implementación del Servicio de Control de Concurrencia se muestran en la figura 6.15. Estos resultados sólamente muestran unas líneas de texto que están implementadas en los avisos del aspecto `ServicioConcurrencia`. Sin embargo, internamente dichos avisos colocan y liberan candados sobre las ranuras de tiempo del estado compartido que van a ser manipuladas.

6.4.2. Servicio de Control de Acceso

El Servicio de Control de Acceso implementa políticas que administran el acceso al estado compartido de la agenda cooperativa. Estas políticas se aplican a los miembros del grupo de trabajo que solicitan realizar una operación (e.g., crear, modificar, cancelar y reservar una cita e indagar el tiempo de un colaborador) sobre el estado compartido de la agenda cooperativa. Además, este servicio complementa la función del servicio de control de concurrencia para definir la secuencia en que dichos miembros tendrán acceso al estado compartido.

El Servicio de Control de Acceso actúa en el momento en que la agenda cooperativa recibe una solicitud de una operación. Entonces, toma dicha solicitud, identifica al colaborador solicitante y le aplica las políticas que tiene definidas. El Servicio de Control de Acceso utiliza como referencia el identificador del solicitante y el tipo de operación que se va a realizar, para aplicar las políticas de acceso. Si la solicitud aprueba todas las políticas, entonces la operación solicitante se realiza, en caso contrario se rechaza.

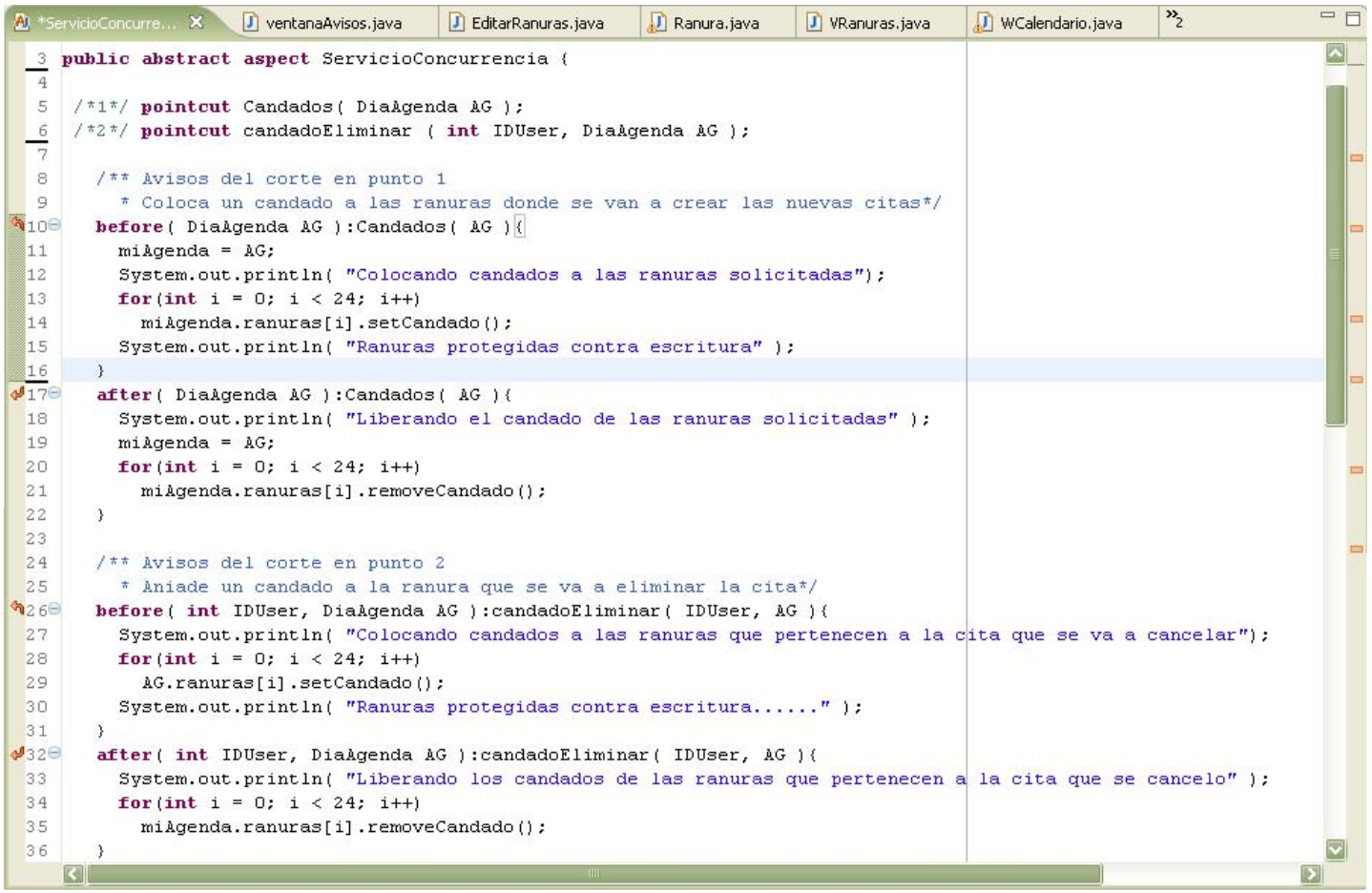


Figura 6.13: Avisos del Servicio de Control de Concurrencia

Normalmente, un *servlet* recibe una solicitud remota, la procesa y envía la respuesta sin verificar la procedencia de dicha solicitud. Para los *servlets* que se utilizan en agenda cooperativa lo anterior no cambia, i.e., cualquier persona ajena a los grupos de trabajo puede enviar una solicitud de acceso a su estado compartido. Por lo tanto, la funcionalidad adicional que presta el Servicio de Control de Acceso consiste en sustituir las partes del código de los *servlets* que procesan las solicitudes remotas, ya que antes de prestar un servicio, se debe verificar que dichas solicitudes provengan de miembros validos que sí pertenezcan a los grupos de trabajo y que además tengan derecho de acceso al estado compartido.

Para el desarrollo del Servicio de Control de Acceso se utilizó un aspecto llamado `ServicioAcceso`, el cual implementa el aspecto `EsMiembroValido()` con un aviso `around`, donde están definidas las políticas de acceso al estado compartido. Dentro del aviso `around` se definieron: 1) el código que va a reemplazar la parte de código de los *servlets* (donde son procesadas las solicitudes de acceso al estado compartido) y 2) las políticas de acceso que se van a aplicar a las solicitudes entrantes (ver figura 6.16). También, en la figura 6.16 se

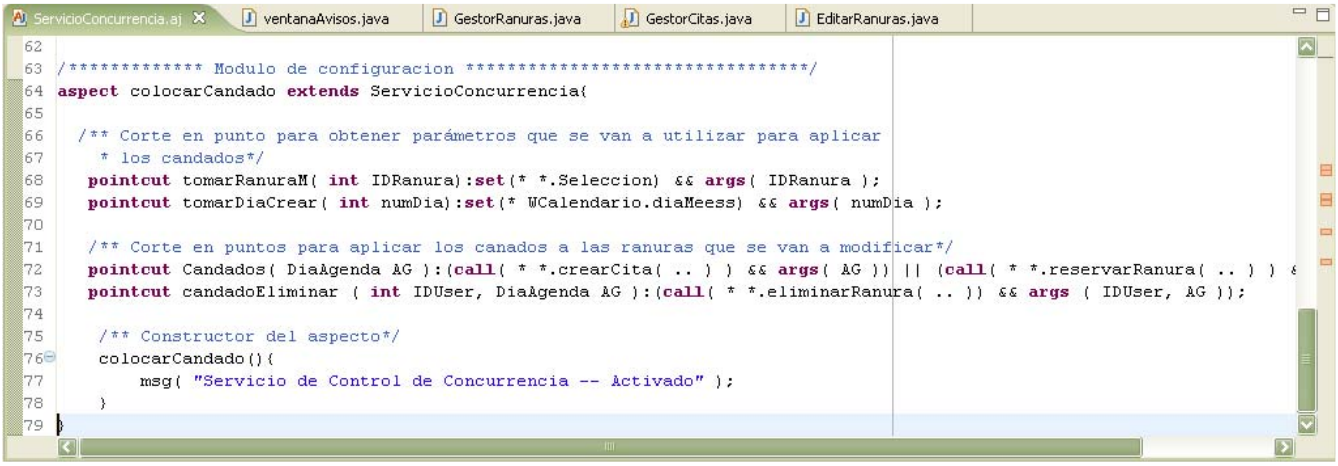


Figura 6.14: Módulo de configuración del Servicio de Control de Concurrencia

```
Servicio de Control de Concurrencia -- Activado
Dia del mes para crear la cita: Martes
Colocando candados a las ranuras solicitadas
Ranuras protegidas contra escritura......
Operacion en curso....
Liberando el candado de las ranuras solicitadas
El ID de la ranura donde se va a candelar la cita es: 2
Colocando candados a las ranuras que pertenecen a la cita que se va a cancelar
Ranuras protegidas contra escritura......
Operacion en curso....
Liberando los candados de las ranuras que pertenecen a la cita que se cancelo
```

Figura 6.15: Resultados de la ejecución del Servicio de Control de Concurrencia

muestra la definición del módulo de configuración del Servicio de Control de Acceso. Este último define el corte en punto `Politicas` que a su vez define el lugar específico donde se va a aplicar la funcionalidad adicional, i.e., cuando se realiza un solicitud de: 1) inicio de sesión, 2) una réplica del estado compartido, 3) actualización del estado compartido y 4) envío de actualización del estado compartido (ver figura fig6.17).

La figura 6.18 muestra la parte del código de los *servlets* donde actúan los puntos de unión del Servicio de Control de Acceso, i.e., dentro del código de los *servlets* de: a) `Sesion`, b) `EnviarReplica`, c) `SolicitudActuralización` y d) `EnviarActuralización`.

Las pruebas realizadas al Servicio de Control de Acceso se muestran en dos partes: 1) primero se presentan los resultados de la ejecución la agenda cooperativa sin la funcionalidad adicional del Servicio del Control de Acceso (ver figura 6.19) y 2) segundo, se muestran los resultados de la ejecución de la agenda cooperativa, pero ahora añadiendo la funcionalidad adicional del servicio del control de acceso (ver figura 6.20).

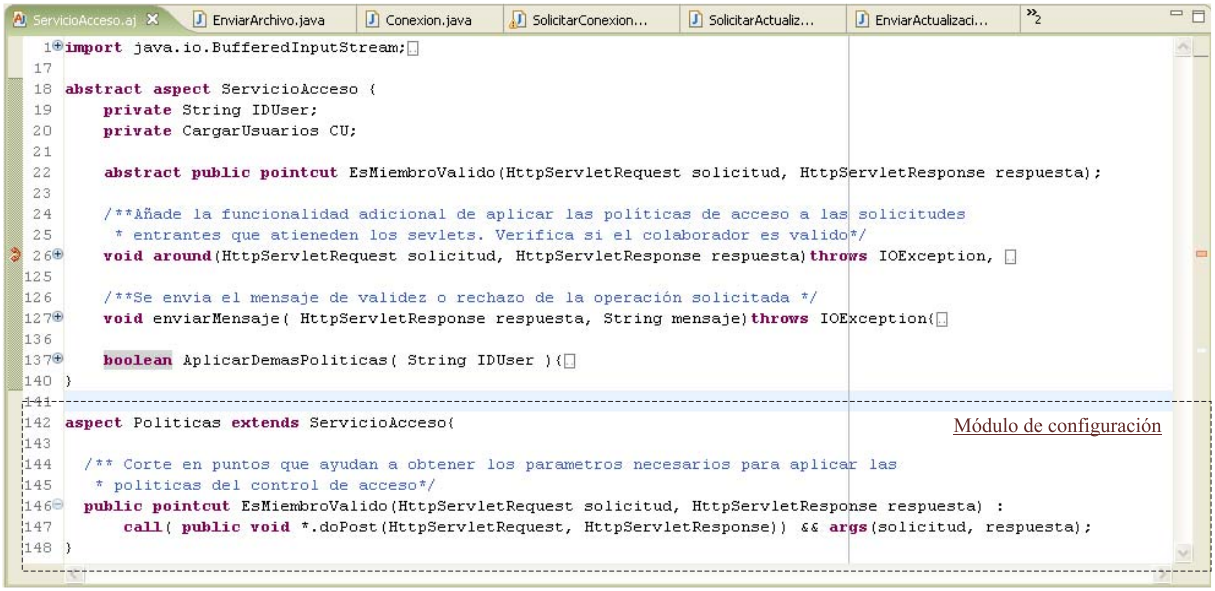


Figura 6.16: Aviso `around` y módulo de configuración del Servicio de Control de Acceso

Figura 6.17: Funciones del aspecto de base donde se va a añadir la funcionalidad adicional del Servicio de Control de Acceso, en referencia al aviso `around` y donde irán colocados puntos de unión

Los resultados de salida del Servicio de Control de Acceso se muestran mediante una línea en las imágenes 6.19 y 6.20. Antes de añadir la funcionalidad adicional, el *servlet* de `Sesión` regresa un mensaje que contiene el nombre del colaborador con el cual se inicio la sesión (ver figura 6.19, Ref. a) en tanto que el *servlet* de `EnviarReplica` regresa un archivo de texto que contiene los datos de la réplica del estado compartido (ver figura 6.19, Ref. b). Al aplicar la funcionalidad del Servicio del Control de Acceso, ahora el aviso `around` del aspecto `ServicioAcceso` se encarga de verificar que las solicitudes de acceso, al estado compartido de la agenda cooperativa provengan de miembros validos del grupo de trabajo. El aviso `around`, además de verificar la veracidad de las solicitudes, también envía los datos requeridos por la solicitud.

Figura 6.18: Defnición de los puntos de unión del Servicio de Control de Acceso dentro del aspecto de base

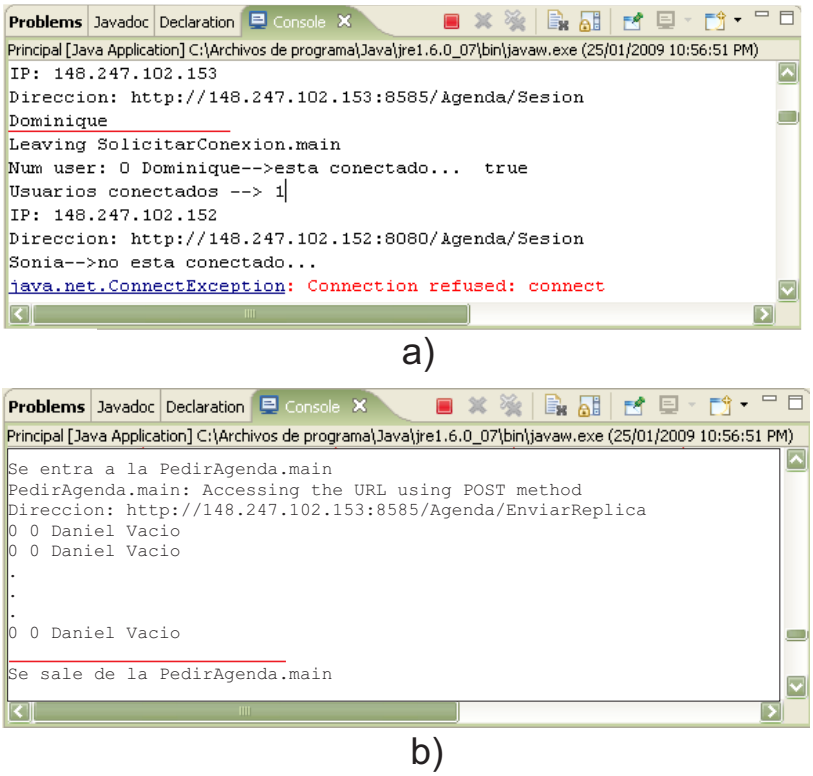


Figura 6.19: Resultados sin la funcionalidad adicional del Servicio de Control de Acceso

6.4.3. Servicio de Actualización

El servicio de actualización verifica los cambios que se hacen al estado compartido de la agenda cooperativa, i.e., a las citas colectivas o a las ranuras de tiempo libre de los miembros del grupo de trabajo. Cuando se realiza un cambio al estado compartido, este servicio se encarga de actualizar las réplicas de dicho estado que están almacenadas en los sitios remotos.

El proceso de actualización del estado compartido se realiza de la siguiente manera: 1) el Servicio de Actualización local inicia la transacción referente a la actualización de las réplicas del estado compartido con su pares remotos, i.e., el servicio de actualización local envía una notificación de que se han realizado cambios al estado compartido (método *push*), 2) el Servicio de Actualización remoto responde con otra solicitud de actualización del estado compartido, y 3) finalmente, el Servicio de Actualización local está en posibilidades de envíar la actualización del estado compartido (metodo *pull*).

Normalmente, para que se pueda escribir información en un servidor Web desarrollado con Java se requiere la configuración de diversas políticas definidas por Sun Microsystems. Este tipo de configuraciones se evitan por completo al implementar el Servicio de Actualización como se describió anteriormente.

Para implementar el servicio de actualización se realizó el aspecto `ServicioActualizacion` que contiene el corte en punto `Actualizar`, el cual añade la

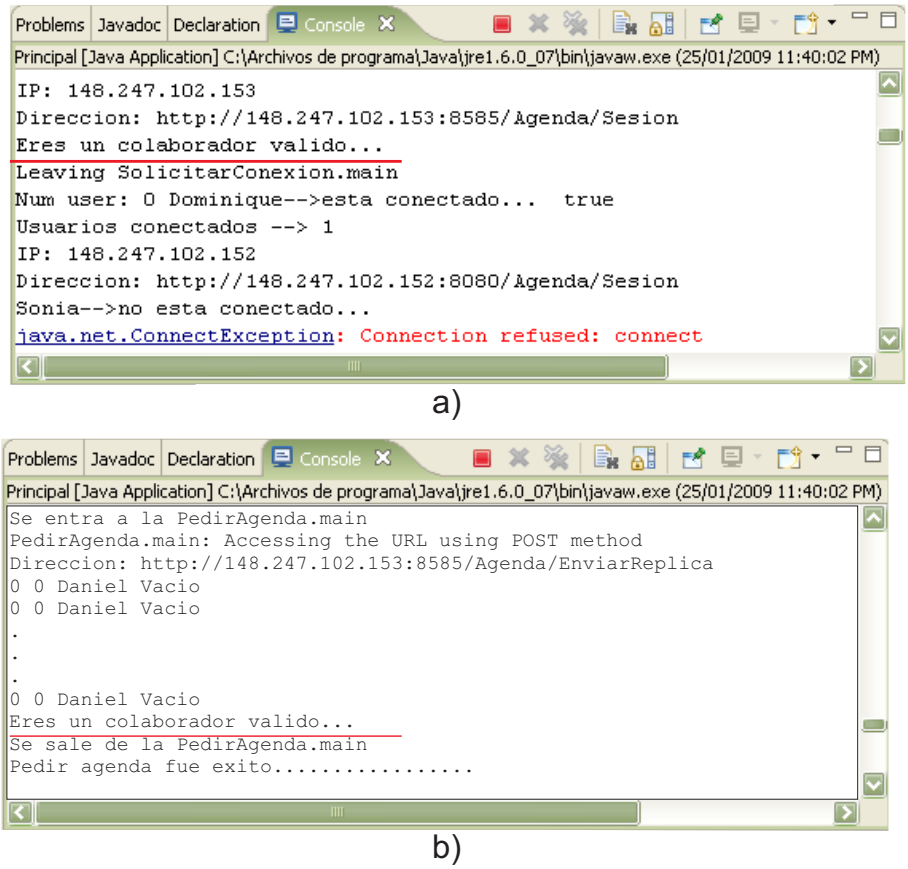


Figura 6.20: Resultados con la funcionalidad adicional del Servicio de Control de Acceso

funcionalidad adicional después de que un colaborador termina de modificar el estado compartido. Por lo tanto, se necesita de un sólo aviso `after` para añadir el servicio de actualización. El aviso `after` expone el código que realiza la actualización del estado compartido.

La figura 6.21 muestra la definición del aspecto `ServicioActualizacion`, el corte en punto `Actualizar` y el módulo de configuración que define el lugar donde van a actuar los puntos de unión dentro del aspecto de base (ver figura 6.22), i.e, el lugar en que se va a añadir la funcionalidad del Servicio de Actualización en el aspecto de base.

El lugar específico donde se añaden los puntos de unión del Servicio de Actualización se define después de que se guardan los cambios realizados a una ranura de tiempo libre local o remota o a una cita colectiva. Los puntos de unión que se generaron al añadir la funcionalidad adicional del módulo del Servicio de Actualización se muestran en la figura 6.23. Los puntos de unión donde se va a añadir la funcionalidad adicional del Servicio de Actualización se definen de dos formas: a) cuando se realizan modificaciones a las ranuras de tiempo libre del colaborador local y se deben actualizar las réplicas de dichas ranuras que se encuentran el los sitios remotos (ver figura 6.23, Ref. a) y b) cuando se realizan cambios a las ranuras de

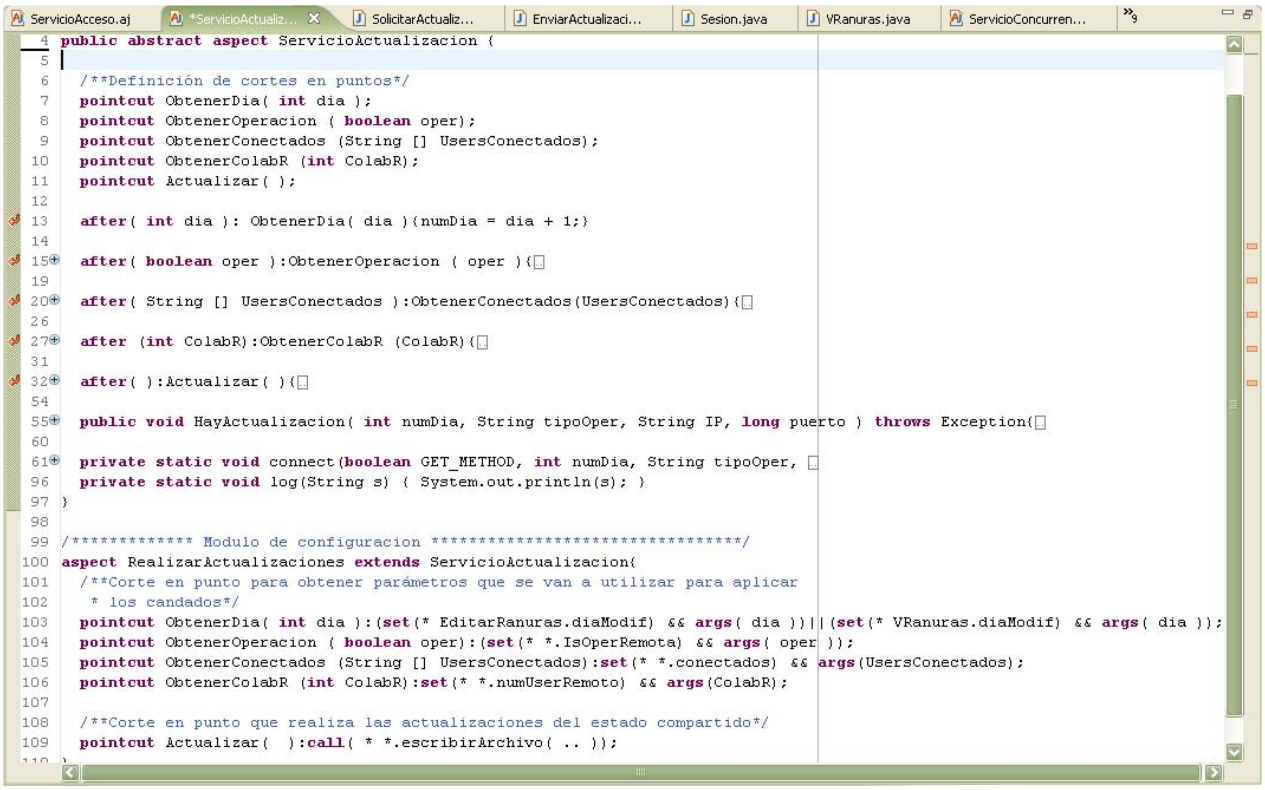


Figura 6.21: Implementación del aspecto `ServicioActualizacion` y del módulo de configuración del Servicio de Actualización

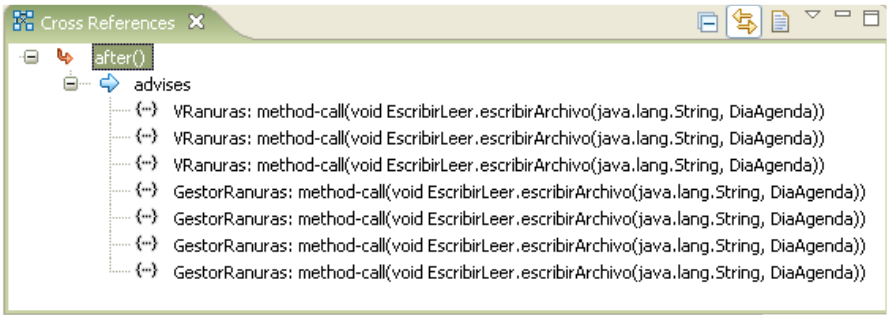
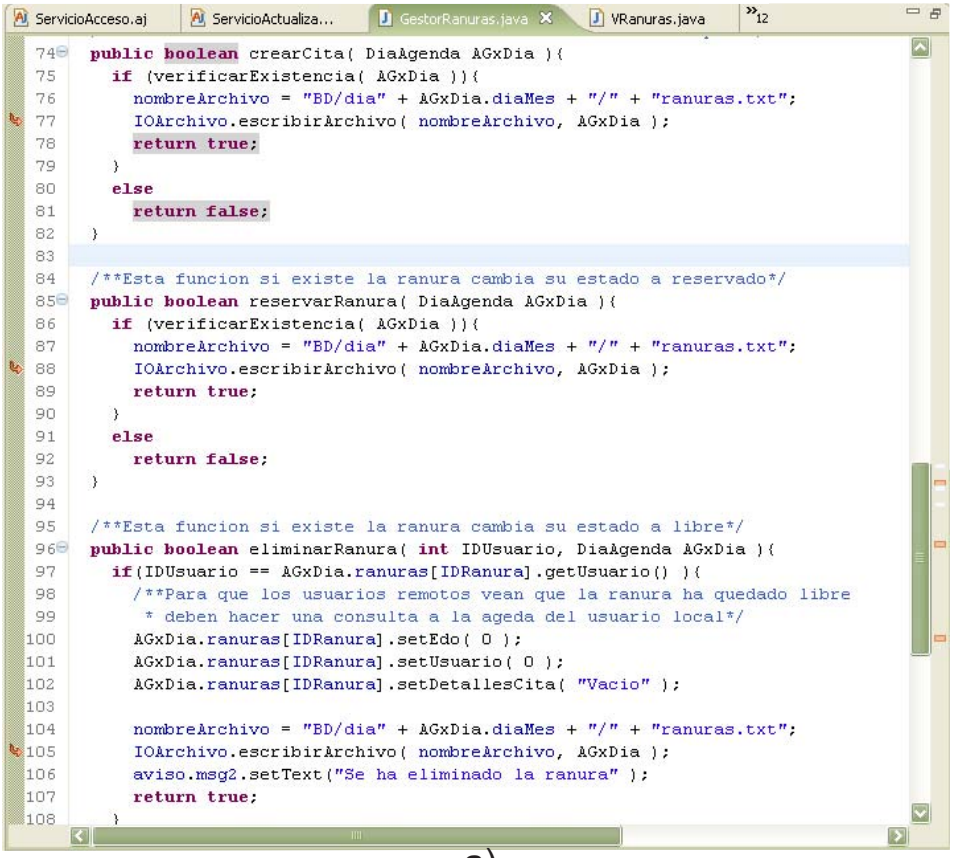


Figura 6.22: Funciones del aspecto de base donde se va a añadir la funcionalidad adicional del Servicio de Actualización, en referencia al aviso `after` y donde irán colocados puntos de unión

tiempo libre de un colaborador remoto. En este último caso se envía la actualización al sitio propietario de dichas ranuras, que finalmente se encargará de actualizar las réplicas de sus ranuras de tiempo libre que se encuentran en los demás sitios remotos (ver figura 6.23, Ref. b).

Al igual que los servicios anteriores, i.e., el de Control de Concurrencia y el de Control de



a)

b)

Figura 6.23: Defnición de los puntos de unión del Servicio Actualización sobre el aspecto de base

Acceso, los resultados de añadir la funcionalidad adicional del Servicio de Actualización, se presentan en forma de mensajes que los sitios remotos envían para notificar de la ocurrencia de éxito o falla al actualizar una réplica del estado compartido.

Como se mencionó anteriormente, es muy útil para un sitio de colaborador indagar que sitios (que corresponden a los miembros sus grupos de trabajo) se encuentran disponibles en la red, debido a que su disponibilidad implica que en esos instantes se les pueda envíar las actualizaciones a la réplica estado compartido que almacenan. En caso contrario, las actualizaciones a la réplica del estado compartido, que está almacenada en sus sitios, se realizará cuando los colaboradores asociados a dichos sitios se conecten a la red.

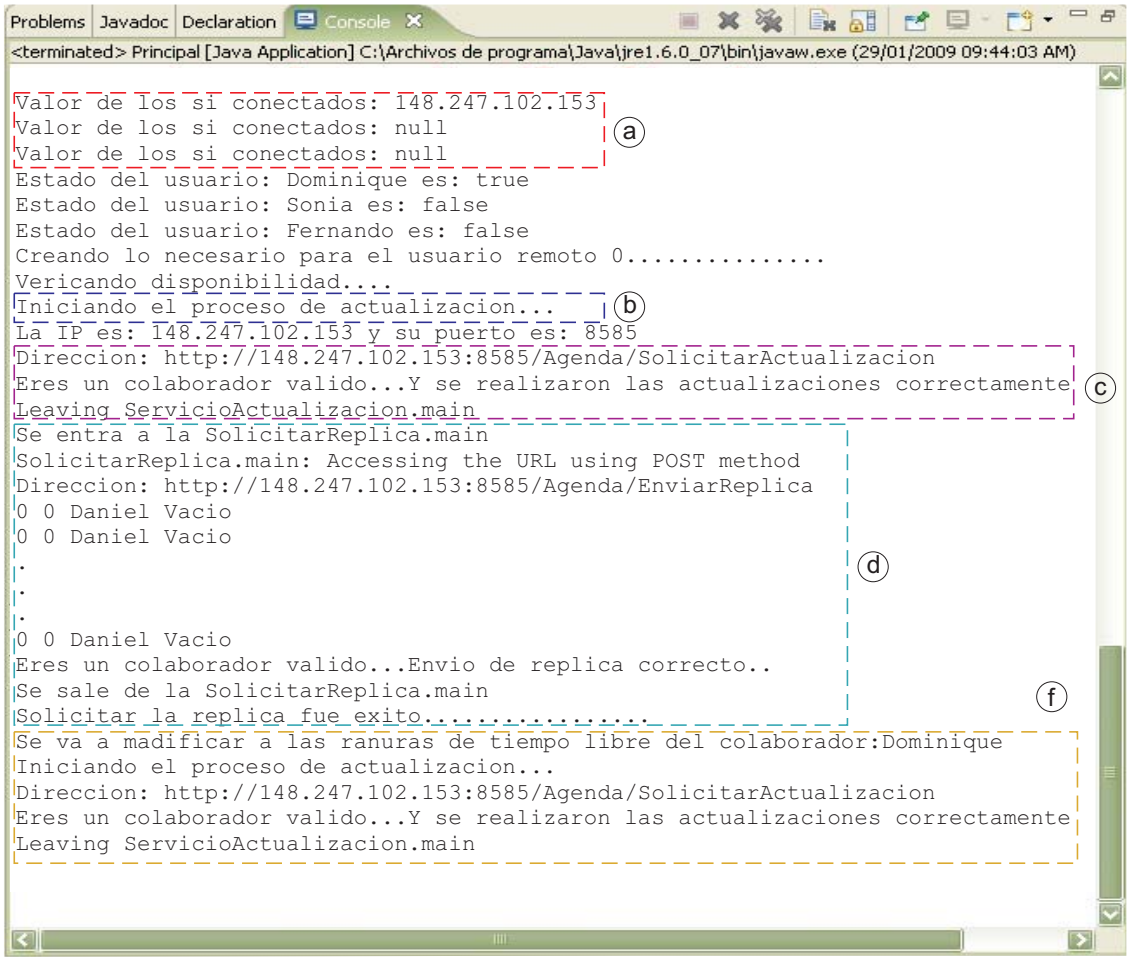


Figura 6.24: Resultados de añadir la funcionalidad adicional del Servicio de Actualización

Los resultados que se obtienen de añadir el Servicio de Actualización al aspecto de base se da en varias etapas:

1. el Servicio de Actualización verifica la disponibilidad de los sitios remotos (ver figura 6.24, Ref. a);

2. si el colaborador local realiza modificaciones a una cita colectiva o a una de sus ranuras de tiempo libre, el Servicio de Actualización debe propagar los cambios a las réplicas que hay de dichas ranuras, ya que su función es la de actualizar las réplicas del estado compartido en todo momento. Lo que muestra la figura 6.24, Ref. b, es que se activa la transacción referente a la actualización de la réplica. Sin embargo, en ese momento, ocurrió un error en la comunciación cuando se intentaba actualizar la réplica de las ranuras de tiempo modificadas, que se encuentra en el sitio del colaborador Dominique;

3. cuando el sitio del colaborador Dominique vuelve a estar disponible en la red, se le envían (al sitio de Dominique) las actualizaciones que estaban pendientes (ver figura 6.24, Ref. c).

   Si varios sitios asociados a los miembros del grupo del colaborador local se encuentran disponibles en el momento que el colaborador local realiza cambios a sus ranuras de tiempo libre, entonces se aplicará el mismo criterio de actualización (como se aplicó con el colaborador Dominique) para poner al día las réplicas de dichas ranuras almacenadas en sus sitios;

4. debido a que el sitio del colaborador Dominique perdió contacto con la red, cuando vuelve a conectarse, el Servicio de Actualización le solicita una réplica de sus ranuras de tiempo libre, con el fin de tener la versión más actual de dichas ranuras (ver figura 6.24, Ref. d);

5. finalmente, si el colaborador local realiza actualizaciones sobre las ranuras de tiempo libre del colaborador Dominique, entonces el Servicio de Actualización debe notificar y actualizar los cambios sobre la réplica de las ranuras de tiempo libre que se encuentran en el sitio del colaborador Dominique. En este caso, sólo se envía la actualización al sitio de Dominique, el cual se encargará de actualizar las réplicas que se tienen de sus ranuras de tiempo libre que podrían estar localizadas en otros sitios remotos (ver figura 6.24, Ref. f).

Como conclusión del presente capítulo se puede decir que si se obtuvieron los resultados esperados, i.e., que el diseño de la agenda cooperativa mediante una arquitectura de distribución híbrida, el uso de *servlets* como medio de comunicación entre los diferentes componentes replicados de la agenda y el uso de la POA, resultaron ser buenas estrategias para alcanzar los objetivos del presente trabajo de tesis.

Capítulo 7

Conclusiones y trabajo futuro

Finalmente, en este capítulo se enuncian las conclusiones del presente trabajo de tesis, así como algunas posibles mejoras y extensiones de la agenda cooperativa propuesta.

7.1. Conclusiones

El estudio y desarrollo de una agenda cooperativa sigue siendo de gran relevancia, ya que la mayoría de los sistemas de agenda disponibles no permiten al usuario manejar su propia información. Además, algunos de estos sistemas sólo son organizadores de tiempo y no permiten la cooperación entre los miembros del grupo. De hecho, muchos usuarios optan por seguir utilizando agendas en papel, debido a que estos sistemas no garantizan la privacidad de sus citas.

La agenda cooperativa propuesta en el presente trabajo de tesis está destinada a los colaboradores que desean: 1) que la información de sus citas sea manejada por ellos mismos y almacenada en su propia computadora y 2) que la información de su tiempo libre sólo sea accesible a los miembros de sus grupos de trabajo.

Por lo tanto, las conclusiones del presente trabajo de tesis han sido divididas en dos partes: 1) las que se refieren al diseño de una agenda cooperativa, mediante una arquitectura de distribución híbrida, para garantizar a los usuarios el acceso oportuno a información coherente y actualizada sobre sus citas privadas y colectivas y 2) las que se refieren a la implementación de los servicios de la agenda cooperativa, mediante el paradigma de la POA, con el fin de poder ser adaptados fácilmente a los diversos patrones de trabajo de los grupos y de permitir que otras aplicaciones con requerimientos similares reutilicen dichos servicios.

A continuación se listan las conclusiones referentes al diseño de la agenda cooperativa en base a una arquitectura de distribución híbrida:

1. La arquitectura de distribución propuesta podría considerarse como una extensión al estudio de las arquitecturas de distribución, realizado por Roth y Unger [Roth and Unger, 2000], ya que se eliminó por completo el uso de un servidor central, distribuyendo el procesamiento de las solicitudes entre los sitios de los colaboradores.

2. La duplicación del estado compartido y de los componentes de la agenda cooperativa, en los sitios de todos los colaboradores, eliminó la necesidad de un servidor central, tanto para el procesamiento de las solicitudes como para el almacenamiento de la información, de manera similar a una arquitectura *peer to peer*.

3. La duplicación de los componentes de la agenda cooperativa mejora el tiempo de respuesta en el tratamiento de operaciones (e.g., crear, modificar, cancelar, reservar y confirmar una cita), puesto que cada una es procesada localmente en vez de ser enviada a un servidor central. Además, una falla en el sitio de algún colaborador no afecta la interacción entre los demás miembros del grupo porque no existe dependencia entre componentes.

4. La duplicación del estado compartido, en los sitios de todos los colaboradores, aumenta su grado de disponibilidad debido a que no se almacena un solo ejemplar en un servidor central. De esta manera, cuando alguno de los colaboradores no está conectado a la red, otro miembro del grupo puede brindar una réplica de la información requerida.

5. La distribución del estado privado garantiza la privacidad de cada colaborador, ya que dicho estado es almacenado únicamente en el sitio del colaborador propietario y no puede ser accedido por ningún otro miembro del grupo de trabajo. De esta manera, no sólo se mantiene la privacidad del colaborador, sino también se evita que sus colegas sean interrumpidos con información irrelevante.

6. El uso de la tecnología de los *servlets* en la implementación de la agenda cooperativa resultó ser una solución que tiene más ventajas que otras tecnologías (e.g., los *scripts* CGIs). Aunque los *servlets* requieren de un ambiente Java para poder ser ejecutados, pueden atender peticiones que provienen de aplicaciones Web desarrolladas en cualquier lenguaje de programación. Además, los *servlets* constituyen una de las tecnologías Web más utilizadas en la actualidad.

De igual manera, enseguida se listan las conclusiones referentes a la implementación de la agenda cooperativa mediante el paradigma de la POA:

1. La utilización de este paradigma permitió el manejo de dos tipos de estados: a) el estado privado, que es administrado por el aspecto de base y b) el estado compartido, que es manipulado por el aspecto de base en combinación con la funcionalidad adicional que brindan los aspectos no funcionales. Este punto podría considerarse como una extensión del trabajo realizado por Phillips sobre la descomposición genérica de un sistema cooperativo [Phillips, 1999], ya que en la presente tesis se pone en evidencia las ventajas de disponer de: 1) un estado compartido réplicado en los sitios de los colaboradores (garantía de disponibilidad de la información) y 2) un estado privado almacenado únicamente en el sitio del colaborador propietario (garantía de privacidad de la información).

2. En cualquiera de sus enfoques de implementación, la POA es un paradigma que tiene más ventajas que la POO, ya que facilita la evolución de una aplicación y aumenta el grado de reutilización de sus módulos constituyentes. Estas ventajas se deben a que la POA resuelve los problemas de entrelazamiento y dispersión de código que dificultan la reutilización de dichos módulos.

3. Los resultados obtenidos en la implementación de la agenda cooperativa, mediante la POA, muestran que los módulos de una aplicación pueden ser fácilmente sustituidos por otros que realicen tareas más complejas, sin tener que modificar el código del aspecto de base.

7.2. Trabajo futuro

Finalmente, se enuncian algunas posibles extensiones que podrían contribuir a la mejora de la agenda cooperativa propuesta en esta tesis:

1. Se realizó un estudio empírico del paradigma de la POA para poner en evidencia sus ventajas de uso sobre el paradigma de la POO, en cuanto a la facilidad de dotar de flexibilidad a una agenda cooperativa. Sin embargo, se obtendría un mejor diseño si se realizara el mismo estudio mediante Casos de Uso de UML, los cuales permiten ver explícitamente y de manera gráfica la dispersión y el entrelazamiento del código entre los diferentes módulos de una aplicación.

2. Se utilizó el mecanismo de candados pesimistas en el desarrollo del servicio de control de concurrencia, sin embargo esta técnica no es completamente adecuada para la agenda cooperativa, ya que no es necesario esperar la respuesta a una solicitud de candado para poder realizar otras operaciones. Por lo tanto, una mejora interesante sería implementar un nuevo servicio de control de concurrencia, mediante una técnica de candados semi-optimistas, que haga más eficiente el acceso al estado compartido de la agenda cooperativa.

3. Se desarrolló un módulo de presentación básico, el cual podría ser extendido con mecanismos y componentes gráficos que ofrezcan información de conciencia de grupo.

4. Aunque se implementaron los servicios de control de concurrencia, control de acceso, actualización y almacenamiento, se requiere añadir otros servicios a la agenda cooperativa, e.g., el servicio de autenticación propuesto en el trabajo de tesis de Elvia K. García [Elvia, 2009].

5. La persistencia de datos se llevó a cabo mediante el uso de archivos de texto, así que una mejora interesante sería substituirlo por un servicio de persistencia más complejo que administre una base de datos, ya que éstas también pueden ser utilizadas en conjunto con los *servlets*.

Apéndice A

AspectJ

AspectJ es el resultado de los trabajos realizados en Xerox-PARC sobre el paradigma de la Programación Orientada a Aspectos (POA) [Kiczales et al., 1997, Lopes et al., 1998, Xerox, 2000]. AspectJ es una extensión de Java que permite definir el aspecto de base en forma de clases Java. Los aspectos no funcionales y su configuración pueden ser definidos en un lenguaje dedicado mediante construcciones sintácticas que siguen la forma de reglas de transformación.

Inicialmente, AspectJ no permitía definir explícitamente la separación entre la configuración de los puntos de unión y el código de los aspectos no funcionales. De hecho, las reglas de transformación que representan los aspectos no funcionales hacían referencia explícita a nombres de clases o métodos definidos en el aspecto de base. Conscientes de ésta limitación, los diseñadores de AspectJ proponen, en las últimas versiones, la propiedad de **herencia de aspectos** y el concepto de **aspecto abstracto** para desacoplar las definiciones de los aspectos no funcionales de su configuración [Xerox, 2000]. Sin embargo, este desacoplamiento depende fuertemente de la disciplina de los desarrolladores de aplicaciones, ya que ningún mecanismo realmente lo fomenta.

AspectJ agrega cuatro entidades principales al lenguaje de base Java [Kiczales et al., 2001]:

1. **Puntos de unión**: son puntos bien definidos en la ejecución del aspecto de base, e.g., llamadas a métodos y constructores, asignaciones y recuperaciones de valores de atributos.

2. **Corte en puntos o corte**: agrupan puntos de unión y permiten exponer el contexto de ejecución de dichos puntos. Existen cortes primitivos y cortes definidos por el usuario.

3. **Avisos**: son acciones que se ejecutan al identificar un punto de unión incluido en un corte. Da un comportamiento adicional o sustituye el comportamiento que se estaba realizando en la aplicación.

4. **Introducciones o declaraciones**: permiten cambiar la estructura de las clases de un programa mediante: 1) la agregación de nuevas interfaces y clases o 2) la extensión de interfaces y clases existentes con nuevos atributos, constructores o métodos.

Las primeras tres entidades son dinámicas y permiten definir el comportamiento adicional de la aplicación que se llevará a cabo en tiempo de ejecución.

A continuación, se describe la forma en que AspectJ utiliza estas entidades en el desarrollo de aplicaciones flexibles.

A.1. Puntos de unión en AspectJ

Un punto de unión se puede ubicar en las siguientes partes del código de un programa:

- **Llamadas a métodos**: cuando se invoca a un método, pero no incluye llamadas al constructor `super`.
- **Ejecución de un método**: cuando se ejecuta el cuerpo de un método.
- **Llamada a un constructor**: cuando se invoca a un constructor, pero no incluye llamadas a los constructores `this` o `super`.
- **Ejecución de un constructor**: cuando se ejecuta el código de un constructor, después de la llamada a los constructores `this` o `super`.
- **Pre-inicialización de un objeto**: antes de que se ejecute el código de inicialización de una clase particular. Comprende el tiempo entre la llamada a su constructor y el comienzo de la llamada al constructor de la clase padre.
- **Inicialización de un objeto**: cuando se ejecuta el código de inicialización de una clase particular. Comprende el tiempo entre el retorno de la llamada al constructor de la clase padre y el retorno de la llamada a su constructor.
- **Referencia a un atributo**: cuando se hace referencia a un atributo, pero el atributo no debe tener el valor `final`.
- **Asignación a un atributo**: cuando se asigna un valor a un atributo.
- **Ejecución de un manejador**: cuando se ejecuta el manejador de una excepción.

El modelo de punto de unión es un elemento crítico en el diseño de cualquier mecanismo del lenguaje orientado a aspectos, ya que provee la intefaz entre el código de los aspectos no funcionales y el código del aspecto de base.

A.2. Corte en puntos o (cortes) en AspectJ

Un corte es un conjunto de puntos de unión más (opcionalmente) algunos de los valores en el contexto de ejecución de dichos puntos. Los cortes son principalmente usados por los avisos y pueden estar compuestos de operadores algebraicos (i.e., and(&&), or(||), not(!) y los paréntesis "()") para crear otros cortes. También pueden capturar puntos de unión de diferentes clases, i.e., un corte puede hacer referencia a diferentes clases o métodos de una clase. AspectJ provee dos tipos de cortes: **cortes primitivos** y **cortes definidos por el programador**.

A.2.1. Cortes primitivos

Los cortes primitivos que permite utilizar AspectJ son:

- `call`: captura los puntos de unión en llamadas a métodos o constructores cuyo encabezado coincide con el pátron. El código asociado al punto de unión se ejecuta exactamente antes o después de la llamada al método o constructor.

  ```
  call(PatronDeMetodo), call(PatronDeConstructor)
  ```

- `execution`: captura los puntos de unión en métodos o constructores cuyo encabezado coincide con el pátron. El código asociado al punto de unión se ejecuta exactamente dentro de la llamada al método o constructor, i.e., ya sea antes de la ejecución del códido del método o constructor o antes del retorno del método o constructor.

  ```
  execution(PatronDeMetodo), execution(PatronDeConstructor)
  ```

- `get`: captura los puntos de unión cuando se accede al atributo `PatronDeAtributo`.

  ```
  get(PatronDeAtributo)
  ```

- `set`: captura los puntos de unión cuando se modifica el atributo `PatronDeAtributo`.

  ```
  set(PatronDeAtributo)
  ```

- `initialization`: captura los puntos de unión cuando se inicializan objetos cuyos constructores coinciden con el `PatronDeConstructor`.

  ```
  initialization(PatronDeConstructor)
  ```

- `staticinitialization`: captura los puntos de unión cuando se inicializan las clases que coinciden con el `PatronDeClase`.

  ```
  staticinitialization(PatronDeClase)
  ```

- `handler`: captura los puntos de unión cuando se cachan excepciones que coinciden con el `PatronDeClase`, no cuando se lanza la excepción.

  ```
  handler(PatronDeClase)
  ```

- `this`: captura los puntos de unión cuando el objeto asociado a la referencia `this` es una instancia de una clase que coincide con el `PatronDeClase` o con la clase asociada al `Identificador`.

  ```
  this(PatronDeClase), this(Identificador)
  ```

- `target`: captura los puntos de unión que se encuentran dentro de la clase que coincide con el `PatronDeClase`.

  ```
  target(PatronDeClase)
  ```

- `args`: captura los puntos de unión cuando los valores de los argumentos coinciden con el `PatronDeClase` y el `Identificador` en la llamada a un método.

  ```
  args(PatronDeClase,Identificador)
  ```

- `within`: captura los puntos de unión que son invocados dentro de la ejecución del cuerpo de un aviso y que coinciden con el `PatronDeClase`. Incluye la inicialización de la clase, del objeto y de los puntos de unión de métodos y constructores.

  ```
  within(PatronDeClase)
  ```

- `withincode`: captura los puntos de unión que se encuentran dentro del ámbito de métodos y constructores que coinciden respectivamente con el `PatronDeMetodo` y el `PatronDeConstructor`. Los cortes `within` y `withincode` trabajan en la estructura léxica del código.

  ```
  withincode(PatronDeMetodo), withincode(PatronDeConstructor)
  ```

- `cflow`: captura los puntos de unión que se encuentran dentro del control de flujo de un programa, incluyendo un punto de unión inicial seleccionado por un corte.

  ```
  cflow(Corte)
  ```

- `cflowbelow`: captura los puntos de unión que se encuentran dentro del control de flujo de un programa, excluyendo un punto de unión inicial seleccionado por un corte.

  ```
  cflowbelow(Corte)
  ```

- `if`: captura los puntos de unión cuando el resultado de la comparación de valores, en tiempo de ejecución, es verdadera. Las expresiones sujetas a comparación pueden ser diversos elementos lógicos, incluyendo puntos de unión expuestos en el contexto, variables estáticas y declaraciones de otros cortes.

  ```
  if(ExpresionBoleana)
  ```

A.2.2. Cortes definidos por el programador

AspectJ permite al programador definir nuevos cortes, cuyo nombre debe estar precedido de la palabra reservada `pointcut`:

```
pointcut nombre(<Parametros>):<Corte>;
```

donde:

<Parametros> expone el contexto del corte y;

<Corte> puede ser un corte primitivo o definido por el programador.

Ejemplo:

```
/*Corte 1*/
pointcut Ver(): call(void Numero*(..));

/*Corte 2*/
pointcut VerSoloComplejo():Ver()&&target(NumComplejo);
```

El `Corte 1` llamado `Ver()` captura todas las llamadas al (a los) método(s) que comienza(n) con la palabra `Numero` y que recibe(n) cualquier número de parámetros (denotado en

AspectJ por `(..)`). El `Corte 2` utiliza el `Corte 1` para capturar las llamadas al (a los) método(s) que comienza(n) con la palabra `Numero`, pero que son invocados sólo sobre la clase `NumComplejo`.

Cuando el programador crea un corte, éste puede ser definido en una clase o en un aspecto y es tratado como miembro de la clase o del aspecto, ya que puede ser declarado como privado o público (`private` o `public`). Su alcance depende de la declaración de la clase o del aspecto que lo contiene. AspectJ no permite la sobrecarga de cortes definidos por el programador.

Los cortes también se pueden declarar como abstractos con la palabra reservada `abstract`. Los cortes abstractos son definidos sin especificar su cuerpo y sólo pueden existir dentro de un aspecto que es declarado abstracto.

AspectJ permite componer nuevos cortes a partir de los cortes primitivos o definidos por el programador. La composición se expresa mediante operadores algebraicos &&("y"), ||("o"), !(no) y paréntesis "()", los cuales se utilizan para asociar cortes. Los cortes compuestos se rigen por la tabla de verdad del operador algebraico que les precede.

La semántica asociada a los operadores es:

- $<corte>_1$ && $<corte>_2$: captura todos los puntos de unión asociados a $<corte>_1$ y $<corte>_2$.
- $<corte>_1$ || $<corte>_2$: captura todos los puntos de unión asociados a $<corte>_1$ o $<corte>_2$.
- **!**$<corte>_1$: captura todos los puntos de unión que no están asociados a $<corte>_1$.
- ($<corte>_1$): captura todos los puntos de unión asociados a $<corte>_1$.

A.3. Avisos

Los avisos definen el código adicional que se ejecuta en los puntos de unión capturados por los cortes. AspectJ soporta tres tipos de avisos: `before` (antes), `after` (después) y `around` (durante). El tipo de aviso determina la forma de interación con el punto de unión sobre el cual está definido.

- `before`: es el código que se ejecuta estrictamente antes del punto de unión. El lugar en donde será añadido o entrelazado el comportamiento referente al aspecto depende del punto de unión utilizado.

```
/*Llamada al aviso before*/
before():NombreCorte(Parametros){
  /*Cuerpo del aviso*/
  ...
}
```

- `after`: es el código que se ejecuta estrictamente después del punto de unión. Según el retorno del método este aviso puede dividirse en tres tipos: 1) al regresar un valor de manera normal, 2) al lanzarse una excepción o 3) sin importar el tipo de retorno. El código de estos tres tipos de avisos se muestra a continuación:

```
/*Al regresar un valor de manera normal*/
after()returning(ValorObjeto):NombreCorte(Parametros){
  /*Cuerpo del aviso*/
  ...
}

/*Al lanzarse una excepcion*/
after()throwing(Exception e):NombreCorte(Parametros){
  /*Cuerpo del aviso*/
  ...
}

/*Sin importar el tipo de retorno*/
after():NombreCorte(Parametros){
  /*Cuerpo del aviso*/
  ...
}
```

- `around`: se ejecuta en lugar del código asociado al punto de unión (ni antes ni después del mismo). El aviso puede retornar un valor como si fuera un método, por lo tanto se debe declarar un tipo de retorno.

```
/*Llamada al aviso around*/
around():NombreCorte(Parametros){
  /*Cuerpo del aviso*/
  ...
  return valor;
}
```

Apéndice B

Java Servlets

Los *servlets* son componentes del lado del servidor que pueden ser ejecutados en cualquier plataforma, gracias a la tecnología Java utilizada para implementarlos. Los *servlets* permiten extender las capacidades de un servidor de aplicaciones, las cuales son accesibles vía el modelo de programación solicitud/respuesta [Pursnani, 2001].

El API J2EESDK 1.x contiene dos paquetes, `javax.servlet` y `javax.servlet.http`, que a su vez contienen todas las clases e interfaces necesarias en la programación de *servlets* [Java Servlets, 2002].

La figura B.1 muestra la jerarquía de las clases en el API J2EESDK 1.x. La clase `GenericServlet` es abstracta e implementa dos interfaces `Servlet` y `ServletConfig`.

La interfaz `Servlet` declara los métodos más importantes en la vida de un *servlet*: 1) `init()` se utiliza cuando se lanza el *servlet* en ejecución, 2) `destroy()` se emplea cuando se requiere destruir el *servlet* y 3) `service()` se invoca cada vez que el *servlet* atiende una solicitud de servicio (ver figura B.2).

Cualquier clase derivada de `GenericServlet` debe redefinir el método `service()`. Este método recibe dos argumentos correspondientes a las interfaces `ServletRequest` y `ServletResponse`. La interfaz `ServletRequest` es el objeto que describe la solicitud de servicio que envía el cliente. Si la solicitud proviene de un formulario HTML, por medio de este objeto se puede acceder a los nombres de los campos y a sus respectivos valores introducidos por el usuario y obtener cierta información sobre el cliente (e.g., dirección IP, tipo de solicitud, navegador). La interfaz `ServletResponse` es el objeto que genera la respuesta al cliente. Además, este objeto permite al método `service()` conectarse al cliente y comunicale el resultado de su solicitud. Sin embargo, el método `service()` debe realizar tantas operaciones como sea necesario para desempeñar su cometido (e.g., escribir y/o leer datos de un fichero o comunicarse con una base de datos).

La clase `HttpServlet` dispone de una implementación del método `service()`. Dicha implementación detecta el tipo de servicio HTTP que ha sido solicitado desde el navegador y llama al método adecuado de esa misma clase (e.g., `doPost()` o `doGet()`). Cuando el programador crea una subclase de `HttpServlet`, por lo general no tiene que redefinir el método `service()`, sino uno de los métodos más especializados (normalmente `doPost()`) que tiene los mismos argumentos que `service()`.

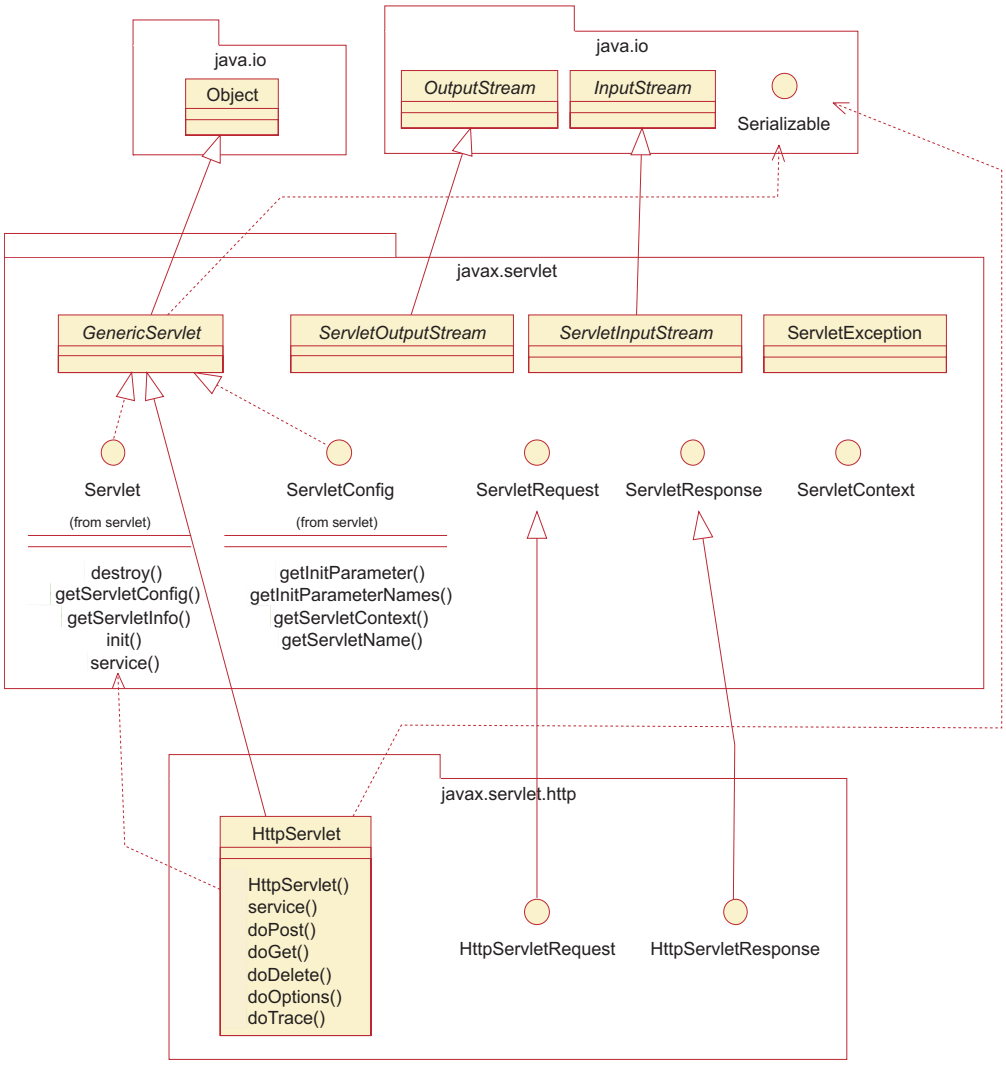


Figura B.1: Jerarquía de clases y sus métodos principales para crear *servlets*

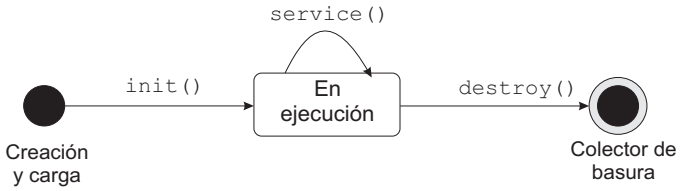


Figura B.2: Tiempo de vida de un *servlet*

Otras interfaces que se utilizan en la implementación de un `servlet` son las siguientes:

- `ServletContext` permite al *servlet* acceder a información sobre el entorno en que se

está ejecutando.

- `ServletConfig` define métodos que permiten al *servlet* mandar información sobre sus parámetros de inicialización.
- `ServletRequest` permite al método `service()` de `GenericServlet` obtener información sobre una petición de servicio de un cliente. Algunos de los datos proporcionados por `GenericServlet` son los nombres y valores de los parámetros enviados por el formulario HTML a una cadena de entrada.
- `ServletResponse` permite al método `service()` de `GenericServlet` enviar una respuesta al cliente que ha solicitado el servicio. Esta interfaz contiene variables (e.g., una cadena de salida o un `writer`) que permiten enviar un flujo datos binarios o caracteres hacia el cliente.
- `HttpServletRequest` deriva de `ServletRequest`. Esta interfaz permite a los métodos `service()`, `doPost()`, `doGet()`, `doDelete()`, `doOptions()` y `doTrace()` de la clase `HttpServlet` recibir y manejar peticiones de servicio HTTP que provienen de los clientes. Otra utilidad de esta interfaz es que permite al desarrollador de aplicaciones obtener información del encabezado de la petición de servicio HTTP.
- `HttpServletResponse` deriva de `ServletResponse`. Por medio de esta interfaz, los métodos de `HttpServlet` envían respuestas de tipo HTTP a las peticiones de los clientes que han solicitado algún servicio.

B.1. Interfaz `Servlet`

Todo *servlet* debe directa o indirectamente implementar la interfaz `Servlet`. Como cualquier otra interfaz de Java, un *servlet* también es una colección de declaraciones vacías de métodos. Los métodos de esta interfaz son:

- ```
  public abstract void init(ServletConfig config)
                                throws ServletException
  ```

  El método `init()` se usa para inicializar los parámetros proporcionados por el objeto `ServletConfig`. Se invoca sólo una vez, a menos que el *servlet* sea reiniciado. El método `init()` permite inicializar los parámetros de configuración, e.g., la conexión a una base de datos y la inicialización de archivos. Ningún método de `Servlet` puede ser invocado a menos que el *servlet* sea inicializado mediante el uso del método `init()`.

- ```
  public abstract ServletConfig getServletConfig()
  ```

El método `getServletConfig()` proporciona el objeto `ServletConfig` para la inicialización de los parámetros del *servlet*. Se pueden crear parámetros adicionales en el archivo `servlet.properties`. Una vez que hayan sido especificados en este archivo, se puede acceder a dichos parámetros mediante el objeto `ServletConfig`.

- ```
  public abstract void service(ServletRequest req,ServletResponse res)
                                        throws ServletException, IOException
  ```

  El método `service()` es el punto esencial del modelo solicitud/respuesta del protocolo HTTP. Recibe una petición del cliente en forma de objeto `ServletRequest`. Los parámetros del cliente se envían junto con el objeto de petición, aunque existen otras formas de enviar parámetros del cliente al *servlet*, e.g., por medio de *cookies* o reescritura de una URL. La respuesta resultante se envía al cliente mediante el objeto `ServletResponse`.

- ```
  public abstract String getServletInfo()
  ```

  El método `getServletInfo()` se usa para la extracción de metadatos del *servlet*, e.g., el autor, la versión del *servlet* y la información concerniente al *copyright*. Este método se debe redefinir dentro del *servlet* para que devuelva esta información.

- ```
  public abstract void destroy()
  ```

  El método `destroy()` se invoca para liberar todos los recursos solicitados, e.g., la base de datos. También se encarga de la sincronización de cualquier hilo pendiente. Este método se llama automáticamente una sola vez, como el método `init()`.

Además de los métodos que heredan de la clase `GenericServlet`, la clase `HTTPServlet` tiene métodos adicionales para cada uno de los métodos de respuesta HTTP:

- `doDelete(HttpServletRequest, HttpServletResponse)`
- `doGet(HttpServletRequest, HttpServletResponse)`
- `doOptions(HttpServletRequest, HttpServletResponse)`
- `doPost(HttpServletRequest, HttpServletResponse)`
  ```

- `doPut(HttpServletRequest, HttpServletResponse)`
- `doTrace(HttpServletRequest, HttpServletResponse)`

Bibliografía

[Allamaraju et al., 2000] S. Allamaraju, A. Longshaw, D. O'Connor, G. Van Huizen, J. Diamond, J. Griffin, M. Holden, M. Daley, M. Wilcox, and R. Browett, *Professional Java Server Programming J2EE Edition*, 2nd Edition, Wrox Press, Inc., 2000.

[Amor et al., 2004] M. Amor, L. Fuentes, D. Jimenez, and M. Pinto, "Adaptive Collaborative Virtual Enviroments: A Component and Aspect-based Approach", *Revista Iberoamericana de Inteligencia Artificial*, No. 24, pp. 33–43, 2004.

[Androutsellis-Theotokis and Spinellis, 2004] S. Androutsellis-Theotokis and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies", *ACM Computing Surveys*, Vol. 36, No. 4, pp. 335–371, December 2004.

[Bannon and Schmidt, 1989] L. J. Bannon and K. Schmidt, "CSCW: Four characters in search of a context", *In ECSCW'89 Proceedings of First European Conference on Computer Supported Cooperative Work, Computer Sciences House, Sloug*, Reino Unido, pp. 358–372, 1989.

[Beard et al., 1990] D. Beard, M. Palanlappan, A. Humm, D. Banks, A. Nair, and Shan, Y.-P, "A Visual Calendar for Scheduling Group Meetings", In *Proceedings of the 1990 ACM conference on Computer-Supported Cooperative Work (CSCW'90)*, ACM Press, pp. 279–290, Los Angeles CA (USA), 1990.

[Beaudouin, 1994] M. Beaudouin-Lafon, "Beyond the workstation: Mediaspaces an augmented reality", In *HCI'94 Proceedings of Conference on Human Computer Interaction, editores: G. Cockton, S. W. Draper y G. R. S. Weir*, Cambridge University Press, pp. 9–18, 1994

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

[Bouraqadi, 1999a] M. N. Bouraqadi-Saâdani, "Un cadre réflexif pour la programmation par aspects", In *Proceeding of LMO'99*, Villefranche sur Mer, January 1999.

[Bouraqadi, 1999b] M. N. Bouraqadi-Saâdani, "UN MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects", PHD Thesis, Université de Nantes, École des Mines de Nantes, July 1999.

[Bouraqadi and Ledoux, 2001] N. Bouraqadi and T. Ledoux, "Le point sur la programmation par aspects", *Technique et Science Informatique*, Vol. 4 , No. 20, pp. 505–528, 2001.

[Ceri and Pelagatti, 1984] S. Ceri and G. Pelagatti, *Distributed Databases Principles and Systems*, 2nd Edition, McGraw-Hill, Inc., New York, USA, 1984.

Distributed Systems: Principles and Paradigms, 2nd Edition, Prentice Hall, USA, 2002.

[Chiba, 1995] S. Chiba, "A Metaobject Protocol for C++", *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pp. 285–299, Austin, Texas, USA, 2005.

[Chiba, 2000] S. Chiba, "Load-time Structural Reflection in Java", Bertino E., Ed., In *Proceeding of ECOOP 2000*, Springer Verlag, Vol. 1850, pp. 313–336 , Sophia Antipolis and Cannes, June 2000.

[Chiba and Tatsubori, 1998] S. Chiba, and M. Tatsubori, "Yet Another java.lang.Class", *Workshop on Reflective Object-Oriented Programming and Systems, (ECOOP'98)*, Brussels, Belgium, Juillet, 1998.

[Coulouris et al., 1994] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems Concepts and Design*, 2nd Edition, Addison-Wesley, 1994.

[Dempsey and Cahill, 1997] J. Dempsey and V. Cahill, "Aspects of System Sopport for Distributed Computing", Lopes C., Kiczales G., Tekinerdogan B., Mens K. Eds, In *Proceding of the Aspects-Oriented Programming Workshop at ECOOP'97*, Jyväskylä, Finland, Juhe 1997.

[Dewan, 1990] P. Dewan, "A Tour of the Suite User Interface Software", In *Proceedings of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*, ACM Press, pp. 57–65, New York NY (USA), 1990.

[Dix, 1994] A. Dix, "Computer-Supported Cooperative Work - a Framework", In *Design Issues in CSCW*, D. Rosenburg and C. Hutchison (Eds.), pp. 23–37, Springer Verlag, 1994.

[Dix, 1997] A. Dix, "Challenges for Cooperative Work on the Web: An Analytical Approach", *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, Vol. 6, pp. 135–156, 1997.

[Domingos et al., 1998] H. J. L. Domingos, N. M. Preguica and J. L. Martins, "Coordination and Awareness Support for Adaptive CSCW Sessions", pp. 21–37, 1998.

[Dourish, 1997] P. Dourish. "Extending Awareness Beyond Synchronous Collaboration", In *CHI'97 Workshop on Awareness in Collaborative Systems, Conference on Human Factors in Computing Systems*, ACM Press, Atlanta, Georgia, E.U.A., 1997.

[Dourish, 1998] P. Dourish, "Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications", *ACM Transactions on Computer-Human Interaction*, Vol. 5 , No. 2, pp. 109–155, 1998.

[Edelstein, 1994] H. Edelstein, "Unraveling Client/Server Architecture", *DBMS*, Vol 5, No. 7, pp. 5, May 1994.

[Ehrlich, 1987a] S. F. Ehrlich, "Social and Psychological Factors Influencing the Design of Office Communications Systems", In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, ACM Press. pp. 323–329, Toronto, Ontario (Canada), 1987.

[Ehrlich, 1987b] S. F. Ehrlich, "Strategies for Encouraging Successful Adoption of Office Communication Systems", *ACM Transactions on Information Systems*, ACM Press, Vol. 5 , No. 4, pp. 340–357, 1987.

[Ellis and Gibbs, 1989] C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems", In *Proccedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 399–407, Seattle, Washington, USA.

[Ellis et al., 1991] C. Ellis, S. Gibbs, and G. Rein, "Groupware: Some Issues and Experiences", *Communication of the ACM*, ACM Press, Vol. 34 , No. 1, pp. 39–58, 1991.

[Ellis and Wainer, 1994] C. Ellis, and J. Wainer, "A Conceptual Model of Groupware", In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'94)*, ACM Press, pp. 79–88, Chapel Hill NC (USA), 1994.

[Fielding et al., 1999] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol - HTTP/1.1*, RFC 2616, IETF Standard, June 1999.

[Fradet and Südholt, 1998] P. Fradet and M. Südholt, "AOP: towards a generic framework using program transformation and analysis", In *Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, Juillet, 1998.

[Francik et al., 1991] E. Francik, S. E. Rudman, S. Cooper, and S. Levine, "Putting Innovation to Work: Adoption Strategies for Multimedia Communication Systems.", *Communications of the ACM*, ACM Press, Vol. 34, pp. 52–63, December 1991.

[Freed and Borenstein, 1996] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, RFC 2046, IETF Standard, November 1996.

[Fujimoto, 1990] R.M. Fujimoto, "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33, No. 10, pp. 31–53, October 1990.

[Gamma et al., 1995] E. Gamma, R. Helen, R. Johnson, and J. Vlissides, *Design Patterns-Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Garrido et al., 2005] J. L. Garrido, M. L. Rodríguez, M. Noguera, M. J. Hornos, and P. Paderewski, "Hacia la Satisfacción de Requisitos de Calidad de los Sistemas Colaborativos Mediante un Diseño Arquitectónico", *III Taller de Sistemas Colaborativos y Adaptativos (SIHICA 2005)*, pp. 33–43, 2005.

[Greif, 1984] I. Greif, "The User Interface of a Personal Calendar Program", In *Proceeding of Human Factors and Interactive System: Symposium on User Interfaces (NYU'82)*, Ablex Publishing Corp, pp. 207–222, Norwood NJ (USA), 1984.

[Greenberg and Marwood, 1994] S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface", In *Proccedings of the ACM CSCW Conference on Computer Sopported Cooperative Work*, ACM Press, pp. 207–217, North Carolina, October 22-26, 1994.

[Grosso et al., 2005] A. Grosso, C. Vecchiola, M. Coccoli, and A. Boccalatte, "A Multiuser Groupware Calendar System Based on Agent Tools and Technology", *International Symposium on Collaborative Technologies and Systems*, pp. 144–151, 2005.

[Grudin, 1988] "Why CSCW Applications Fail: Problems in the Design and Evaluation of Organization of Organizational Interfaces", In *Proceedings of the 1988 ACM Conference on Computer Supported Cooperative Work*, ACM Press, pp. 85–93, Portland OR (USA), 1988.

[Grudin and Palen, 1995] J. Grudin,and L. Palen, "Why Groupware Succeeds: Discretion or Mandate?", In *Proceedings of the Fourth Conference on European Conference on Computer-Supported Cooperative Work*, Kluwer Academic Publishers, pp. 263–278, Norwell MA (USA), 1995.

[Guicking and Grasse, 2006] A. Guicking, and T. Grasse, "A Framework Designe for Synchronous Groupware Applications in Heterogeneous Environments", In *Proceedings of the 12th International Workshop on Groupware: Design, Implementation, and Use (CRIWG'2006)*, Y. A. Dimitriadis, I. Zigurs, and E. Gómez-Sánchez (Eds.), Lecture Notes in Computer Science LNCS 4154, Springer, pp 203–218, Medina del Campo (Spain), 2006.

[Karsenty and Beaudouin-Lafon, 1993] A. Karsenty and M. Beaudouin-Lafon, "An Algorithm for Distributed Groupware Applications", In *Proceedings of the 13th International Conference on Distributed Computing Systems ICDCS'93*, pp. 195–202, Pittsburgh, May 1993.

[Kelley and Chapanis, 1982] J. F. Kelley, and A. Chapanis, "How Professional Persons Keep Their Calendars: Implications for Computerization", *Journal of Occupational Psychology*, Vol. 55, pp. 241–256, 1982.

[Kiczales et al., 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin, "Aspect-Oriented Programming", In *Proceedings European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science (LNCS 1241), Springer Verlag, pp 220–242, Jyväskylä (Finlande), June 1997.

[Kiczales et al., 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ", In *Proceedings of the 15th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer-Verlag, pp. 327–355, 2001.

[Kiczales and Pæpcke, 1994] G. Kiczales and A. Pæpcke, "Open Implementation and Metaobjects Protocols", Expanden tutorial notes, 1994.

[Elvia, 2009] Elvia K. García, "Disponibilidad de recursos compartidos en un ambiente colaborativo con carácter ubicuo", Tesis de Maestria, Departamento de Computación, CINVESTAV-IPN, Enero 2009.

[Kincaid et al., 1985] C. Kincaid, P. Dupont, and A. Kaye, "Electronic Calendars in the Office: An Assessment on User Needs and Current Technology", *ACM Transactions on Office Information Systems*, Vol. 3, No. 1, pp. 89–102, 1985.

[Lieberherr et al., 1999] K.J. Lieberherr, D. Lorenz, and M. Mezini, "Programming with Aspectual Components", Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, March 1999.

[Lopes and Kiczales, 1997] C. Lopes, and G. Kiczalez, "D: A Language Framework for Distributed Programming", PARC Technical report, SPL97-010 P9710047, February 1997.

[Lopes et al., 1998] C. Lopes, G. Kiczalez, M. Tekinerdogan, and W. de Meuter, "Recent Developments in Aspect", Eds. *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, pp. 398–401, Brussels, Belgium, July 1998.

[Lukosch, 2003] S. Lukosch, *Transparent and Flexible Data Sharing for Synchronous Groupware*, PhD Thesis, FernUniversität, Hagen (Alemania), Mayo 2003.

[Lukosch, 2004] S. Lukosch, "Flexible and Transparent Data Sharing for Synchronous Groupware", In *International Journal of Computer Applications in Technology (IJCAT): Special Issue on Current Approaches for Groupware Design, Implementation and Evaluation*, Vol. 19, No. 3-4, pp. 215–230, 2004.

[Lunau, 1998] C.P. Lunau, "Is Composition of Metaobjects = Aspect-Oriented Programming", *Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, July 1998.

[Malone et al., 1995] T. W. Malone, K.-Y. Lai, and C. Fry, "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work", *ACM Transactions on Information Systems*, Vol. 13, No. 2, pp. 175–205, 1995.

[Malone y Crowston, 1994] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination", *En ACM Computing Surveys*, Vol. 1, No. 26, pp. 87–119, 1994.

[Matsuoka and Yonezawa, 1993] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages", *Research Directions in Concurrent Object-Oriented Programming*, MIT press, pp.107–150, 1993.

[McCoy et al., 2001] C. McCoy, M. Riggsby, S. Haberman, and A. Falciani, *Mastering Lotus Notes and Domino 6*, SYBEX, 2003.

[Miles, 2004] R. Miles, *AspectJ Cookbook*, 1st Edition, O'Reilly Media, Inc, 2004.

[Morán et al., 2002] A. L. Morán, J. Favela, A. M. Martínez, and D. Decouchant, "Before Getting There: Potential and Actual Collaboration", In *Proceedings of the 8th International Workshop on Groupware: Design, Implementation and Use (CRIWG'2002)*, J. M. Haake, J. A. Pino (Eds.), Lecture Notes in Computer Science LNCS 2440, Springer, pp. 147–167, La Serena (Chile), September 1-4 2002.

[Mosier and Tammaro, 1997] J. Mosier, and S. Tammaro, "When Are Group Scheduling Tools Useful?" *Computer Supported Cooperative Work (CSCW)*, Vol. 6, No. 1, pp. 53–70, 1997.

[Mulet et al., 1995] P. Mulet, J. Malenfant, and P. Cointe, "Towards a Methodology for Explicit Composition of MetaObjects", *Proceedings of OOPSLA'95*, pp. 316–330, Austin, Texas, October 1995.

[Oliva and Butazo, 1998] A. Oliva and L.E. Butazo, "Composition of MetaObjects in Guarana", *Proceedings on the OOPLSA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[Palen, 1999] L. Palen, "Social, Individual & Technological Issues for Groupware Calendar Systems", In *Proceedings of the Conference on Human Factors in Computing Systems: The CHI in the Limit (CHI'99)*, ACM Press, pp. 17–24, Pittsburgh, PA (USA), May 1999.

[Payne, 1993] S. J. Payne, "Understanding Calendar Use", *Human Computer Interaction*, Vol. 8, No. 2, pp. 83–100, 1993.

[Phillips, 1999] W. G. Phillips, "Architectures for Synchronous Groupware", Technical Report 1999-425, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 6 1999.

[Prakash and Knister, 1992] A. Prakash, M.J. Knister, "Undoing Actions in Collaborative Work", In *Proceedings of the ACM CSCW Conference on Computer-Sopported Cooperative Work*, pp. 273–280, Toronto, Nov 1992.

[Prakash et al., 1999] A. Prakash, H. S. Shim, and J. H. Lee, "Data Management Issues and Trade-offs in CSCW Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 1, pp. 213–227, January/February 1999.

[Pryor and Bastán, 1999] J. Pryor and N. Bastán, "A Reflective Arquitecture for the Support of Aspect-Oriented Programming in Smalltalk", *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.

[Pursnani, 2001] V. Pursnani, *An Introduction to Java Servlet Programming*, Crossroads, Vol. 8, No. 2, pp. 3–7, ACM Press, 2001.

[Qiu et al., 2001] L. Qiu, V. Padmanabhan, and G. Voelker, "On the Placement if Web Server Replicas", *Proc. 20th INFOCOM Conference*, IEEE Computer Society Press, pp. 1587–1596, Los Alamitos CA., 2001.

[Radoslavov et al., 2001] P. Radoslavov, R. Govindan, and D. Estrin, "Topology-Informed Internet Replica Placement", *Proceedings of WCW'01: Web Caching and Content Distribution Workshop, Boston, MA*, pp. 296, Amsterdam: North-Holland, June 2001.

[Romero-Salcedo and Decouchant, 1997] M. Romero-Salcedo, and D. Decouchant, "Structured Cooperative Authoring for the World Wide Web", *Computer Supported Cooperative Work*, Vol. 6, No. 2-3, pp. 157–174, 1997.

[Roseman and Greenberg, 1992] M. Roseman, and S. Greenberg, "Groupkit: A Groupware Toolkit for Building Real-time Conferencing Applications", In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, ACM Press, pp. 43–50, Toronto, Ontario (Canada), 1992.

[Roth and Unger, 2000] J. Roth and C. Unger, "An Extensible Classification Model for Distribution Architectures of Synchronous Groupware", *Fourth International Conference on the Design of Cooperative Systems (COOP2000)*, IOS Press, pp. 113–127, Sophia Antipolis (France), May 23-26 2000.

[Sarin and Greif, 1985] S. Sarin, and I. Greif, "Computer-based Real-time Conferencing Systems", *IEEE Computer*, Vol. 18, No. 10, pp. 33–45, 1985.

[Schlichter et al., 1997] J. Schlichter, M. Koch and M. Bürger, "Workspace Awareness for Distributed Teams", In *Proceedings of Workshop Coordination Technology for Collaborative Applications*, editor: W. Conen, Singapore, 1997.

[Schneider 1990] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Comput. Surv.*, Vol. 22, No. 4, pp. 299–320, December 1990.

[Schoder and Fischbach, 2003] D. Schoder and K. Fischbach, "Peer-to-peer Prospects". *Communications of the ACM*, Vol. 46, No. 2, pp. 27–29, 2003.

[Schuckmann et al., 1999] C. Schuckmann, J. Schümmer, and P. Seitz, "Modeling Collaboration Using Shared Objects", In *Proceedings of the International Conference on Supporting Group Work (GROUP'1999)*, ACM Press, pp. 189–198, Phoenix, AZ (USA), November 14-17 1999.

[Slein and Davis, 1999] J. Slein and J. Davis, *Requirements for Advanced Collection Functionality in WebDAV*, Internet draft, WebDAV Working Group, June 1999.

[Smith, 1990] B. C. Smith, "What do You Mean, Meta?", *Workshop on Reflection and Metalevel Architechtures in OO Programming, ECOOP/OOPSLA'90*, Ottawa, Ontario, Canada, October 1990.

[Stroud and Wu, 1996] R. Stroud and Z. Wu, "Using Metaobjects Protocols to Satisfy Non-Functional Requirements", *Advances in Object-Oriented Metalevel Architectures and Reflection*, CRC press, pp 31–52, 1996.

[Tanenbaum and van Steen, 2002] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd Edition, Prentice Hall, USA, 2002.

[Tejedor and Jesús, 2006] M. Tejedor and R. Jesús, *Domine las Redes P2p (Peer To Peer)*, 1er Edition, Creaciones Copyright, 2006.

[Trevor et al., 1994] J. Trevor, T. Rodden, and G. Blair, "COLA: A Lightweight Platform for CSCW", *Computer Supported Cooperative Work*, Vol. 3, No. 2, pp. 197–224, 1994.

[Waldman et. al., 2000] M. Waldman, R. Aviel, C. Lorrie, "Publius: A Robust, Tamper-Evident, Censorship-Resistant Web Publishing System", In *Proceedings of the 9th USENIX Security Symposium.*

[Watanabe and Yonezawa, 1988] "Reflection in a Object Oriented Concurrency Language " *Proceeding of OOPSLA'88*, ACM Press, pp. 306–315, 1988.

Referencias Web

[Ajax Framework] Wikipedia, The Free Encyclopedia, *Ajax framework*, http://en.wikipedia.org/wiki/Ajax_framework

[Apache-Tomcat, 2008] *The Apache Software Foundation*, http://tomcat.apache.org/, 2008.

[Apple iCal] Apple Inc. *Características de iCal*, http://www.apple.com/es/macosx/features/300.html#ical

[ApliCalendar] Wikipedia, The Free Encyclopedia, *Lista de aplicaciones con soporte para iCalendar*, http://en.wikipedia.org/wiki/List_of_applications_with_iCalendar_support

[CGI, 1998] *The Common Gateway Interface*, http://hoohoo.ncsa.uiuc.edu/cgi/, 1998.

[Daboo et al., 2007] C. Daboo, B Desruisseaux, and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", http://tools.ietf.org/html/rfc4791#page-12, March 2007.

[Dawson and Howes, 1998] F. Dawson and T. Howes, "vCard MIME Directory Profile", ftp://ftp.isi.edu/in-notes/rfc2426.txt, September 1998.

[Dawson and Stenerson, 1998] F. Dawson and D. Stenerson, "Internet Calendaring and Scheduling Core Object Specification (iCalendar)", http://tools.ietf.org/html/rfc2445, November 1998.

[Dawson et al., 2002] F. Dawson, S. Reddy, D. Royer, and E. Plamondon, "iCalendar DTD Document (xCal)", http://www.ietf.org/proceedings/02mar/I-D/draft-ietf-calsch-many-xcal-01.txt, February 2002.

[Gnutella, 2003] The Gnutella web site: http://gnutella.wego.com

[Google Calendar] Google Calendar, *Google Calendar APIs and Tools*, http://code.google.com/apis/calendar/reference.html

[GroupDAV] GroupDAV. *Especificaciones de GroupDAV*, http://www.groupdav.org/

[hCalendar] Microformats. *Especificaciones de hCalendar*, http://microformats.org/wiki/hcalendar

[iCal] Wikipedia, The Free Encyclopedia. *iCal*, http://en.wikipedia.org/wiki/ICal

[Java Servlets, 1999] Java Servlet and JavaServer Pages Technology-Comparing Methods for Server-Side Dynamic Content, *Sun Microsystems*, http://java.sun.com/products/jsp/jspservlet.html, 1999.

[Java Servlets, 2002] Java Servlet Technology-The J2EE Tutorial. *Sun Microsystems*, http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html, 2002.

[Kazaa, 2003] The Kazaa web site: http://www.kazaa.com

[Laddad, 2002] R. Laddad, "I want my AOP!", *JavaWorld, part 1,2,3*, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, 2002.

[MsOutlook] Wikipedia, The Free Encyclopedia. *Microsoft Outlook*, http://en.wikipedia.org/wiki/Microsoft_Outlook

[Postel, 1982] J. B. Postel, "Simple Mail Transfer Protocol (SMTP)", http://tools.ietf.org/html/rfc821, August 1982.

[Sadoski, 1997] D. Sadoski, *Client/Server Software Architecture - Overview*, http://www.sei.cmu.edu/str/descriptions/clientserver_body.html, August 1997.

[Schussel, 1996] G. Schussel, *Client/Server: Past, Present, and Future*, http://www.dciexpo.com/geos/dbsejava.htm, 1996.

[Sedlar et al., 2004] E. Sedlar, J. Whitehead and U.C. Santa Cruz, "Access Control Protocol", *Web Distributed Authoring and Versioning (WebDAV)*, http://www.ietf.org/rfc/rfc3744.txt, 2004.

[Vandana, 2004] V. Pursnani, " An Introduction to Java Servlet Programming!", *Crossroads, The ACM Student Magazine*, http://www.acm.org/crossroads/xrds8-2/servletsProgramming.html, 2004.

[W3C, 2004] "W3C: Technology and Society domain", *World Wide Web Consortium*, http://www.w3.org/Collaboration/Overview.html, 2004.

[Webcal] Wikipedia, The Free Encyclopedia, *Webcal*, http://en.wikipedia.org/wiki/Webcal

[Wikipedia-Google Calendar] Wikipedia, The Free Encyclopedia. *Google Calendar*, http://en.wikipedia.org/wiki/Google_Calendar

[Xerox, 2000] "aspectj.org Site" http://www.aspectj.org, 2000.