View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

U/PC: Un Patrón Arquitectónico para Aplicaciones Colaborativas

Luis A. Guerrero
luguerre@dcc.uchile.cl
Departamento de Ciencia de la Computación
Universidad de Chile,
Santiago, Chile

Sergio Ochoa[1], Oriel Herrera[2], David Fuller
{sochoa, oaherrer, dfuller}@ing.puc.cl
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile, Santiago,
Chile

RESUMEN

Los sistemas colaborativos pertenecen al área de investigación llamada CSCW ("Computer-Supported Cooperative Work"). Estos sistemas apoyan a grupos de personas trabajando en equipo, con el fin de alcanzar metas comunes. Un aspecto fundamental en todo proceso de colaboración es la comunicación, y aunque todos los sistemas siguen el mismo patrón arquitectónico, la falta de una definición explícita del patrón hace que sea imposible el reuso de soluciones en este ámbito. Es por eso que en el presente artículo se presenta el patrón arquitectónico *U/PC (Users/Processes Communication)* que modela los mecanismos de comunicación en los sistemas colaborativos, brindando una solución al problema antes mencionado.

Palabras Clave: Sistemas Colaborativos, Patrón Arquitectónico, CSCW, Comunicación.

1. Introducción

Un *sistema* colaborativo es un sistema basado en computadores que apoya a un grupo de personas que trabajan en una tarea o meta común, y que provee una interfaz a un ambiente compartido **[Ellis 91]**. Bajo este enfoque hay cuatro componentes importantes en aplicaciones colaborativas: las personas, las formas de interacción grupal, las estrategias de colaboración y el ambiente compartido. Para proveer interacción grupal, los sistemas colaborativos deben prestar mucha atención a tres elementos básicos: *Comunicación, Coordinación* y *Colaboración* **[Ellis 91]**. La comunicación es el proceso por el cual un emisor envía a un receptor, un mensaje a través de un canal. La coordinación es la encargada de coordinar las actividades de los usuarios, para obtener un todo coherente. Por último, la colaboración es el proceso por el cual dos o más personas comparten información. La efectividad en la comunicación y colaboración aumentará, siempre que las actividades del grupo estén coordinadas.

Según la taxonomía de tiempo-espacio presentada por Johansen **[Johansen 88]**, en aplicaciones colaborativas sólo son posibles dos tipos de comunicación: sincrónica y asincrónica. De este modo, hablamos de *comunicación sincrónica* cuando los usuarios están trabajando o interactuando al mismo tiempo, y de *comunicación* asincrónica cuando los usuarios están interactuando en tiempos distintos. El patrón arquitectónico **[Buschmann 96]** presentado en este artículo, aporta una *arquitectura de comunicación* genérica, para soportar ambos tipos de comunicación. Los patrones arquitectónicos expresan los esquemas fundamentales de la organización estructural en los sistemas de software **[Buschmann 96]**. Estos proveen el conjunto de componentes participantes y especifican sus responsabilidades, las reglas de interacción y los aspectos más importantes de las relaciones entre ellos.

Las soluciones basadas en patrones no sólo han permitido formalizar problemas y soluciones, sino que también han ayudado a profesionalizar la construcción de sistemas de software, pudiendo garantizar la calidad del producto. Además, los patrones han permitido reducir los tiempos involucrados en el desarrollo del sistema y reutilizar soluciones probadas. Por todo lo antes mencionado, consideramos que la mejor manera de avanzar en esta dirección en el área de los sistemas colaborativos, es definiendo un patrón arquitectónico general, orientado a la comunicación, que sea capaz de reflejar la problemática asociada a ese dominio específico.

---

[1] Profesor Adjunto – Instituto de Informática – Universidad Nacional de San Juan (UNSJ) - Argentina
[2] Profesor Auxiliar – Depto. de Matemática y Computación – Universidad Católica de Temuco - Chile

A continuación se describe el patrón arquitectónico propuesto, luego se presentan algunos trabajos relacionados (sección 3), y por último se presentan las conclusiones y el trabajo futuro (sección 4).

Definición del Patrón Arquitectónico

Para formalizar el patrón arquitectónico propuesto, se utilizará nomenclatura sugerida por Buschman en su libro: *Pattern-Oriented Software Architecture: A System of Pattern.*

NOMBRE: U/PC

U/PC (Users/Processes Communication) es un patrón arquitectónico que establece claramente el esquema de comunicación que deben respetar las aplicaciones colaborativas, para soportar los dos tipos de interacciones antes definidos (comunicación sincrónica y asincrónica).

EJEMPLO

Suponga que estamos desarrollando una aplicación colaborativa para que un grupo de usuarios realice una "lluvia de ideas" estructurada (*brainstorming*). Es decir, los usuarios aportan ideas sobre un tópico particular, y pueden opinar a favor o en contra de las ideas de los otros usuarios. En esta aplicación, como en la mayoría de las aplicaciones colaborativas, están involucrados los procesos internos y los de usuario. Los *procesos de usuario* son aquellos que ejecutan una orden explícita de éste; mientras que los *procesos internos* sólo responden a solicitudes internas del sistema, como por ejemplo la distribución de las actualizaciones sobre un objeto compartido. En la lluvia de ideas, los usuarios pueden enviar mensajes (con ideas y argumentos a favor o en contra) que son vistos por los otros usuarios. Estos mensajes son explícitamente enviados por un usuario (*procesos de usuario*), y deben ser mostrados a todos los otros usuarios. Puede ocurrir además, que un usuario nuevo se incorpore a la sesión de trabajo, o que algún usuario que estaba participando abandone la sesión. En este caso, el usuario simplemente entra a la sesión o sale de ella. Sin embargo, la entrada o salida de un usuario debería ser notificada al grupo. Aquí ocurre un evento que es automáticamente comunicado a todos los usuarios del grupo, a través de mensaje generado por un *proceso interno* (el usuario no tiene conciencia de la ocurrencia del evento).

Es posible también, que un usuario que acaba de ingresar al sistema quiera ver qué es lo que ha ocurrido durante la sesión de trabajo (la historia). En este caso, solicita ver paso a paso las ideas que han sido generadas, así como su proceso de aceptación o rechazo según la argumentación de los otros usuarios. Es posible también que decida replantear con nuevos argumentos, una idea previamente rechazada. Esto implica que el sistema debe mantener el registro de la historia de la sesión de trabajo.

CONTEXTO

En un ambiente de colaboración entre varios usuarios, éstos necesitan comunicarse (intercambiar mensajes). En algunas ocasiones es necesario mantener registro de los mensajes para posteriores operaciones sobre ellos. El rango de comunicación que es posible encontrar en los sistemas colaborativos va desde punto a punto hasta broadcast, y a su vez esta puede ser sincrónica o asincrónica.

PROBLEMA

En sistemas colaborativos y en muchos sistemas distribuidos, se producen mensajes generados por usuarios o por procesos del sistema que deben ser enviados a otros usuarios o procesos. Además, algunas veces es necesario almacenar estos mensajes para operaciones posteriores. El problema principal consiste en definir los elementos participantes en el sistema de comunicación de las aplicaciones colaborativas, su función y las relaciones entre ellos. Una solución en esta línea, permitiría fundamentalmente el reuso de diseños **[Jacobson 97]**.

FUERZAS

- Un usuario puede explícitamente enviar un mensaje a otro usuario o grupo de usuarios.
- Un proceso interno puede enviar un mensaje a otro(s) proceso(s) o a otro(s) usuario(s).

- Un usuario (o proceso) puede recibir mensajes en cualquier momento.
- Un usuario o módulo del sistema puede requerir ver mensajes enviados con anterioridad.

SOLUCIÓN

Introducir un componente "*users/processes communication*" (U/PC) para proveer comunicación entre usuarios y/o procesos, y para el almacenamiento y recuperación de los mensajes. Este componente permite enviar mensajes a un usuario de la sesión de trabajo, a un grupo de usuarios o a todos los usuarios de la sesión. Lo mismo para procesos internos. Este componente provee comunicación tanto sincrónica como asincrónica. Si un usuario miembro del grupo de trabajo no está conectado a la sesión, y un mensaje le es enviado, el componente lo almacena y se lo envía cuando el usuario ingresa al sistema.

ESTRUCTURA

El patrón arquitectural U/PC comprende cinco tipos de componentes participantes: procesos, manejador de eventos, canal, bitácora y mensajes pendientes.

Los procesos pueden ser de usuario o internos. En un *proceso de usuario*, es el usuario quien explícitamente genera un mensaje y lo envía a un grupo de destinatarios. En un *proceso interno*, el mensaje no es explícitamente enviado por el usuario, sino por un módulo del sistema. Ocurre generalmente por una acción hecha por el usuario, como ingreso o salida de una sesión de trabajo, bloqueo o desbloqueo de un archivo, modificación o borrado de un objeto compartido, etc. Los procesos además de enviar mensajes, también reciben y procesan mensajes.

Clase Proceso	**Colaboradores** - Manejador de Eventos
Responsabilidades - Genera mensajes a ser enviados - Procesa los mensajes recibidos	

Figura 1 : Componente *Proceso*

El manejador de eventos recibe mensajes de los procesos y los envía a los otros manejadores de eventos de los otros usuarios, a través del canal. También recibe mensajes enviados por otros manejadores de eventos, y los despacha a los procesos locales correspondientes (de usuario o internos).

Clase Manejador de Eventos	**Colaboradores** • Proceso • Canal
Responsabilidades • Envía los mensajes a través del canal • Recibe del canal los mensajes y los entrega al proceso correspondiente	

Figura 2: Componente ***Manejador de Eventos***

El canal recibe un mensaje de un manejador de eventos, y lo redirecciona a los manejadores de eventos correspondientes. En caso de necesitar la historia de los mensajes enviados por el canal, éste además envía una copia del mensaje al componente "bitácora".

Clase Bitácora	**Colaboradores** • Canal
Responsabilidades • Almacena la historia de los mensajes enviados en el sistema	

Figura 3: Componente *Canal*

Clase Canal	**Colaboradores** • Manejador de Eventos • Bitácora • Mensajes Pendientes
Responsabilidades • Distribuye los mensajes por la red hacia los correspondientes destinatarios	

Figura 4: Componente *Bitácora*

En caso de que un mensaje deba ser enviado a un usuario, y éste no se encuentre conectado a la sesión de trabajo, el canal envía este mensaje al componente "mensajes pendientes". Cuando un nuevo usuario se incorpora a la sesión de trabajo, el canal chequea en el componente "mensajes recibidos", y en caso de haber mensajes para este usuario, se los envía.

Clase Mensajes Pendientes	**Colaboradores** • Canal
Responsabilidades • Almacena mensajes que no se han podido entregar • Devuelve los mensajes almacenados de usuario	

Figura 5: Componente *Mensajes Pendientes*

El siguiente diagrama muestra los objetos que componen el patrón U/PC.

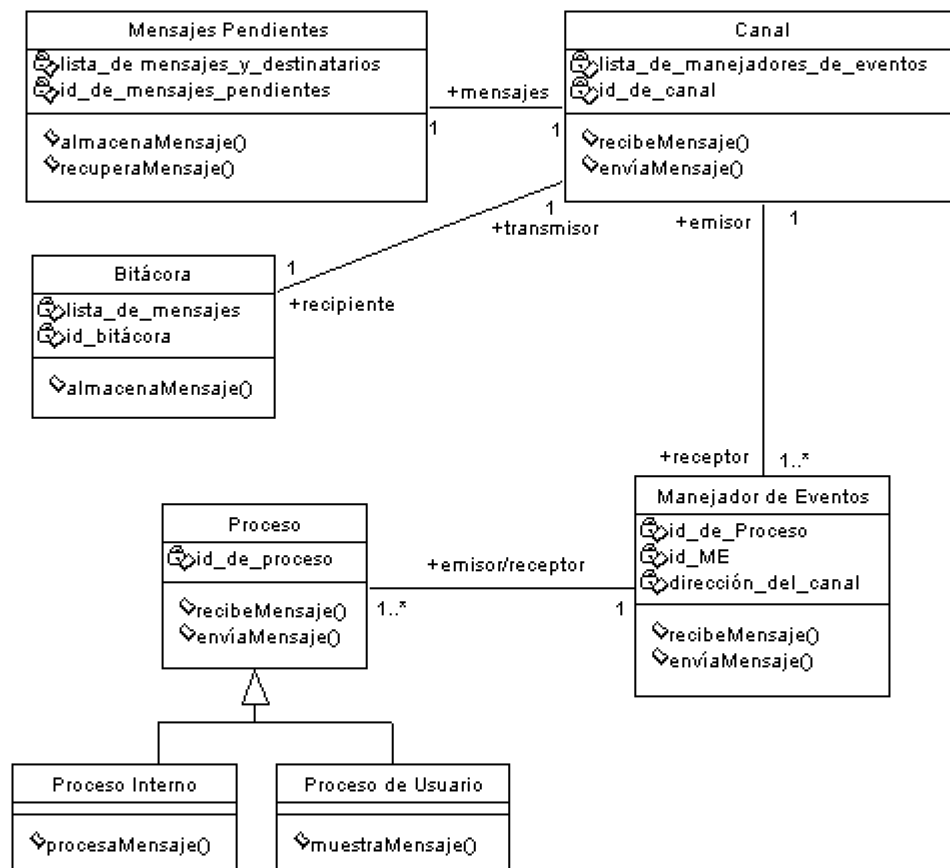

Figura 6: Diagrama de Clases del Modelo

Procesos

El escenario de trabajo de las aplicaciones colaborativas (figura 7) está compuesto por *aplicaciones*, por medio de las cuales los *procesos* (internos y de usuario) colaboran utilizando el *canal* que los comunica. Este tipo de aplicaciones tiene al menos un proceso de usuario; y además por cada uno de éstos, tiene uno o más procesos internos. Un proceso de usuario es aquel que ejecuta una orden expresa del usuario. En cambio un proceso interno, es el que trabaja sin una orden expresa del usuario, como por ejemplo los *daemons* de Unix. El proceso, ya sea interno o de usuario, siempre interactúa a través de manejadores de eventos.

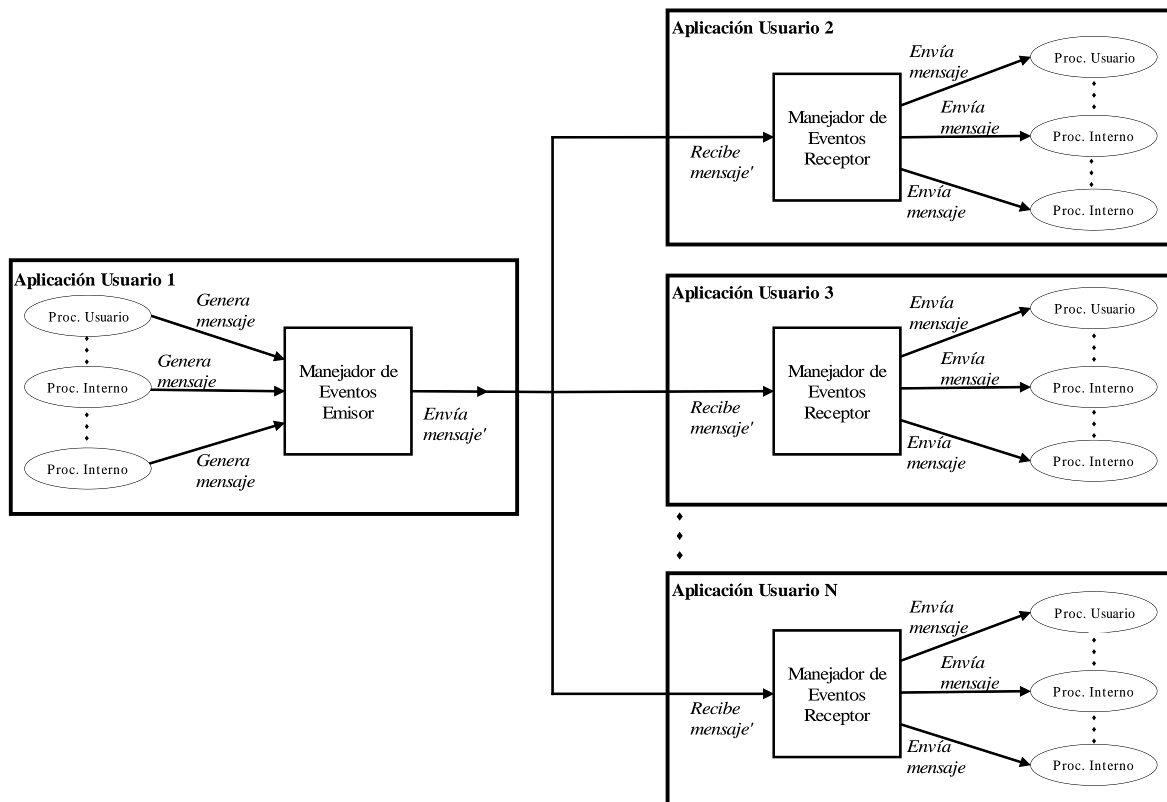

Figura 7: Escenario de Trabajo de las Aplicaciones Colaborativas

Canal

La complejidad asociada a este componente es muy grande debido a que tiene que ocultar los detalles de implementación y aparecer ante el usuario como un canal de comunicación *seguro* **[Gallardo 97]**. Las funciones de este componente dependen mucho de la cantidad de inteligencia depositada en él, y también de su estrategia de comunicación (par a par, cliente-servidor o combinaciones). En la actualidad, la mayor parte de los canales son implementados bajo una arquitectura cliente-servidor, aunque la cantidad de inteligencia depositada en ellos varía mucho. La función básica del canal es la distribución de mensajes en forma sincrónica y asincrónica, donde el destinatario puede ser un proceso de usuario, un proceso interno, uno o más grupos de éstos, etc. Además, dependiendo de la estrategia de comunicación elegida, puede ser necesario que el canal realice algunas de las siguientes funciones:

- **Persistencia de mensajes:** en caso de ser necesario, los mensajes deberían poderse almacenar.
- **Administración de objetos compartidos:** creación, actualización, eliminación, almacenamiento y control de concurrencia sobre estos objetos compartidos. También la serialización de las operaciones sobre ellos.
- **Administración de conexiones:** creación, actualización, eliminación, y validación de conexiones de usuarios al sistema. Además de la creación, actualización y eliminación de sesiones de trabajo.

Mensajes

Los mensajes pueden ser de diversos tipos según el emisor, el contenido, y la forma de distribución del mismo. Un mensaje puede ser enviado explícitamente por un proceso de usuario a otro(s) proceso(s) de usuario del grupo. En este caso el mensaje puede contener texto, imagen, sonido o cualquier otro objeto. Ejemplo de esto son los “mailing list”, el correo electrónico, los sistemas de chat, etc. Sin embargo, también un proceso interno de una aplicación puede enviar mensajes a otro(s) proceso(s) que pertenecen a aplicaciones de otros usuarios. En este caso, el mensaje puede ser una coordenada de un área de dibujo, el bloqueo de un objeto compartido, el arribo de un nuevo usuario a la sesión de trabajo, un comando, etc. Ejemplo de sistemas con este tipo de mensajes son las pizarras de dibujo compartidas, telepunteros, editores de texto colaborativos, sistemas de memoria compartida

distribuida, sistemas que usen RPC ("Remote Procedure Call"), etc. Los mensajes serán simplemente "objetos enviados a través del canal", sin importar si éstos son generados por un proceso de usuario o por un proceso interno del sistema. Tampoco importa si éstos obedecen a una notificación de eventos, un comando RPC, un mensaje de correo electrónico, etc. Bajo este esquema, un *emisor* será cualquier tipo de proceso que genere mensajes, y un *receptor* será cualquier tipo de proceso que los reciba.

Manejador de Eventos
Este componente, por su parte, es el encargado de manejar los eventos que deben ser enviados por el canal y viceversa. El manejador de eventos debería ser independiente de la estrategia de comunicación implementada en el canal, pero lamentablemente en la mayoría de los casos no sucede así. Los manejadores de eventos más avanzados utilizan una API para encapsular toda la funcionalidad dependiente de la implementación del canal. De esa manera al cambiar la implementación del canal, sólo hay que cambiar la implementación de la API, y el resto de la funcionalidad asociada a las aplicaciones y al propio manejo de los eventos no sufrirá cambio alguno. Al igual que en el caso del canal, las funciones de este componente dependen de la cantidad de inteligencia que se desee colocar en él. Sin embargo, existe un conjunto de funciones básicas que el canal debe brindar, como por ejemplo:

- **Entregar y recibir los mensajes en forma correcta:** esto se refiere tanto a la integridad de datos, como a la secuencia de propagación o aceptación de mensajes. Para ello el manejador de eventos deberá interactuar con el canal para garantizar una entrega y una recuperación segura de los mensajes.
- **Almacenar mensajes:** se refiere a que el manejador de eventos debe cumplir funciones de buffer de los mensajes de entrada y de salida al canal, debido a que la contraparte no siempre está disponible en el momento que se la solicita.
- **Distribuir mensajes:** se refiere a la entrega de mensajes a los procesos destinatarios correspondientes, que formen parte de una aplicación.

Dependiendo de la implementación del canal, también podría encargarse de la administración de los objetos compartidos, la administración de usuarios y sesiones, y la persistencia de los mensajes.

DINÁMICA

Se muestran algunos de los principales escenarios que se pueden presentar al usar el patrón U/PC.

Escenario I. Muestra cómo un usuario ingresa a una sesión de trabajo, registrándose con el *canal*.

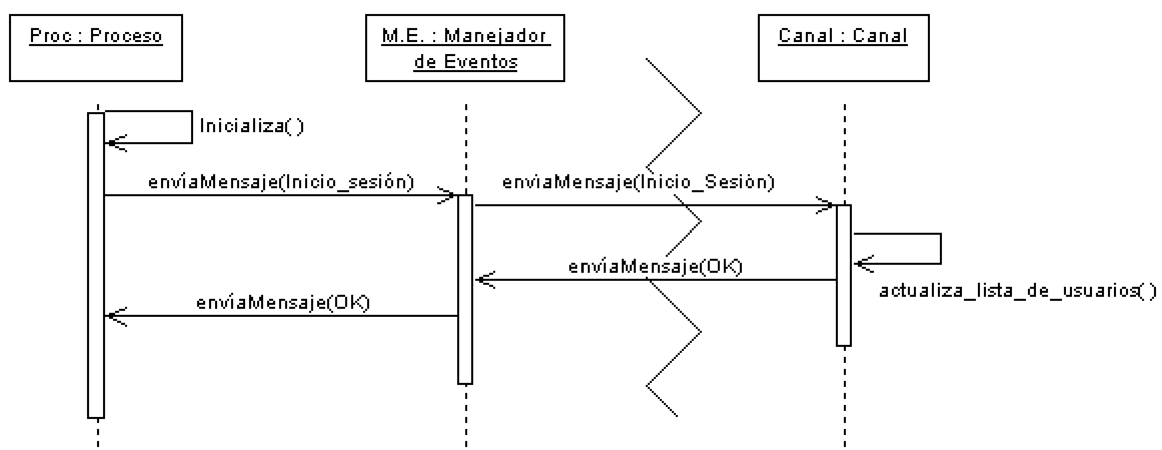

Figura 8: Escenario I. Registro de un usuario a una sesión de trabajo, a través del canal

- El usuario (*proceso*) inicia su aplicación.
- El proceso envía al *manejador de eventos* el mensaje de solicitud de incorporación a la sesión de trabajo.
- El *manejador de eventos* le indica al *canal* que un nuevo usuario desea incorporarse a la sesión.

- El *canal* registra al nuevo usuario y devuelve un "OK" al *manejador de eventos*.
- El *manejador de eventos* le indica al *proceso* que ya ha ingresado a la sesión de trabajo.

El escenario para salir de una sesión de trabajo es similar.

Escenario II. Muestra cómo un *proceso* envía un mensaje a un conjunto de procesos.

- El *proceso* envía el mensaje a su *manejador de eventos*.
- El *manejador de eventos* envía el mensaje al *canal*.
- El *canal* responde que recibió el mensaje, y procede a distribuirlo a cada *manejador de eventos* destinatario.
- Cada *manejador de eventos* recibe el mensaje y lo envía al *proceso* correspondiente.
- Cada *proceso* procesa, según corresponda, el mensaje.

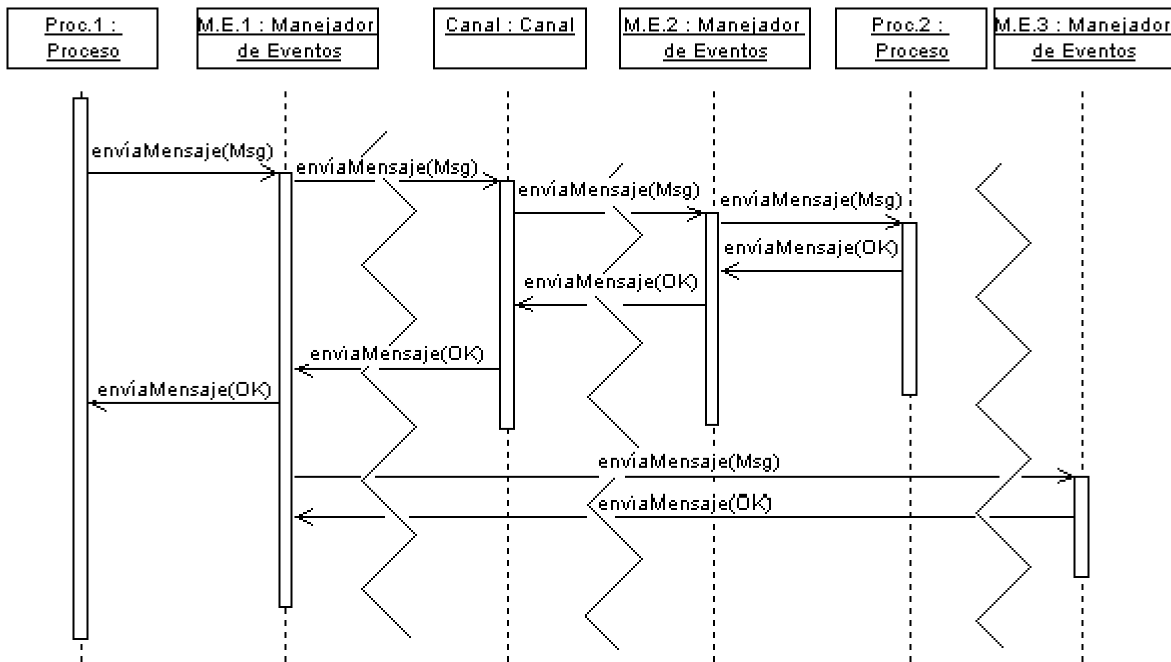

Figura 9: Escenario II. Un proceso envía un mensaje a un conjunto de procesos.

Escenario III. Muestra cómo el *canal* almacena en la *bitácora* todos los mensajes recibidos.

- Al recibir el mensaje, el *canal* lo envía a la *bitácora* antes de enviarlo a los *manejadores de eventos* destinatarios.
- La *bitácora* recibe el mensaje y lo almacena.

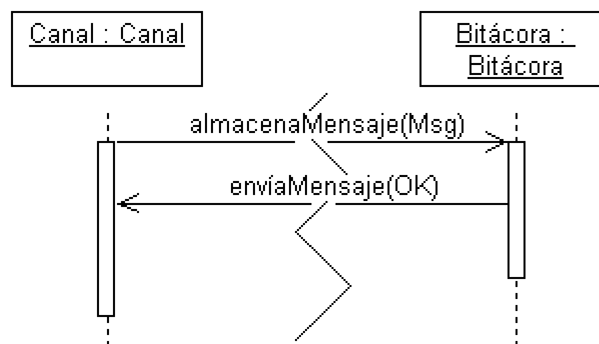

Figura 10: Escenario III. Almacenamiento de mensajes en la bitácora, por parte del canal.

Escenario IV. Muestra cómo el *canal* almacena un mensaje en el componente *mensajes pendientes*.

- El *canal* se da cuenta que un destinatario no está conectado (buscando en su lista de usuarios conectados), por lo que le envía el mensaje al componente *mensajes pendientes*.
- El componente *mensajes pendientes* almacena el mensaje.

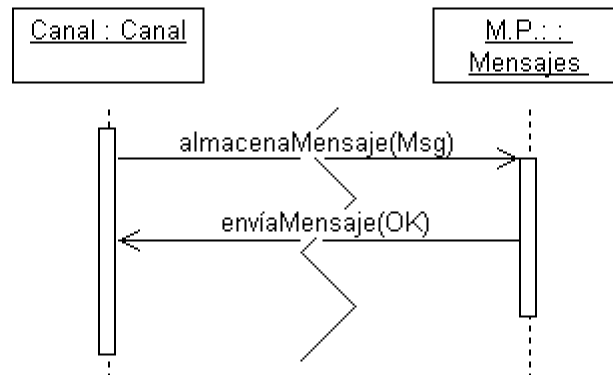

Figura 11: Escenario IV. Almacenamiento de mensajes en Mansajes Pendientes, por parte del Canal

IMPLEMENTACIÓN

1. Es necesario primero definir la funcionalidad requerida del patrón. Hay que decidir si se desea o no la bitácora, así como el tiempo de almacenamiento de mensajes en ésta: la última hora, la última semana, los últimos 500 mensajes, etc.
2. Es necesario decidir si la comunicación es sincrónica o asincrónica, para decidir si se utiliza o no el componente "mensajes pendientes".
3. Hay que especificar los mensajes que pueden ser generados tanto por el usuario como por los procesos internos. También hay que especificar los mensajes que pueden llegar a cada usuario o proceso, y qué hacer con ellos.
4. Se debe decidir si se quiere usar una arquitectura cliente-servidor o punto a punto. De esta decisión dependerá dónde se almacena la lista de usuarios conectados en la sesión de trabajo, así como su respectivo IP.
5. Si se usa *bitácora* o *mensajes pendientes*, es necesario definir dónde se van a implementar éstos, y dónde se almacenarán sus mensajes. Por ejemplo, en una arquitectura cliente-servidor estos dos componentes, así como el componente *canal,* pueden ser implementados como parte del servidor principal, o pueden implementarse como servidores independientes.

Un ejemplo de implementación es la plataforma TOP con TopInterface **[Guerrero 98]**.

VARIANTES

Una variante es el patrón arquitectónico *broker* **[Buschmann 96]**.

USOS CONOCIDOS

En las plataformas de apoyo al trabajo colaborativo GroupKit **[Roseman 97]** y Habanero**[Chabert 98]**.

CONSECUENCIAS

El patrón U/PC provee algunos importantes beneficios:

- Abstrae todos los aspectos de comunicación a través de las redes. Un proceso emisor simplemente envía el mensaje a su manejador de eventos, y se despreocupa. El manejador de eventos y el canal se encargan de hacerlo llegar a los destinatarios.

- La implementación de los componentes es independiente. Se puede rediseñar y/o modificar la arquitectura del canal y del manejador de eventos, y los procesos no se ven afectados.

- Se puede enviar cualquier tipo de mensajes por el mismo canal: notificaciones, eventos, resultados, excepciones, comandos RPC, imágenes, sonido, etc.

- Permite implementar comunicación sincrónica y asincrónica entre procesos.

- Permite mantener la historia del proceso de comunicación.

PATRONES RELACIONADOS

El patrón "broker" **[Buschmann 96]** es usado para estructurar sistemas distribuidos **[Tanenbaum, Adler 95]** que interactúan mediante invocación de servicios remotos, coordinando la comunicación.

3. Trabajos Relacionados

Un trabajo relacionado es el patrón arquitectónico *Broker* **[Buschmann 96]**, aunque éste está limitado a arquitecturas Cliente-Servidor **[Adler 95]**, y además no refleja la naturaleza colaborativa de las operaciones que allí se realizan. Otros trabajos relacionados son las plataformas de desarrollo de aplicaciones colaborativas, aunque están más orientadas a la implementación del patrón propuesto. Estas plataformas son herramientas que proveen un ambiente común a través del cual se comparte información, y las funciones para manejar dicho ambiente. A continuación se presentan las plataformas más representativas, las cuales son implementaciones específicas del patrón presentado en este trabajo:

3.1. TOP (Ten Objects Platform)

La plataforma TOP funciona bajo una arquitectura cliente-servidor **[Guerrero 98]**. Ésta implementa el componente "*canal*" como un servidor encargado de la persistencia de los objetos compartidos, distribución de mensajes, manejo de sesiones y conexiones de usuarios, y control de piso. El componente "*manejador de eventos*" forma parte tanto de los clientes como del servidor. Éste utiliza un puerto de entrada/salida para que las aplicaciones cliente se comuniquen, usando al servidor como intermediario.

3.2. MetaWeb

Está diseñado para brindar soporte a aplicaciones sincrónicas que trabajan sobre la Web, bajo una arquitectura cliente-servidor **[Trevor 97]**. El componente "*canal*" está representado por los servidores MetaWeb Server y un Web Server tradicional. El componente "*manejador de eventos*" está implementado por dos componentes, una API y el MetaWeb Client.

3.3. NSTP (Notification Service Transfer Protocol)

Éste posee una arquitectura cliente-servidor **[Patterson 96]**, en donde tanto el componente "*canal*" como el "*manejador de eventos*" se implementan por medio de un servidor de notificación que mantiene el estado compartido de las múltiples aplicaciones.

3.4. Habanero

Esta plataforma implementa el componente "*canal*" a través de un servidor, y los procesos (interno y de usuario) a través de clientes. Básicamente su arquitectura es cliente-servidor **[Chabert 98]**, con características similares al patrón Broker **[Buschmann 96]**. El "*manejador de eventos*" está constituido por el objeto *Wrapped.* Éste es el encargado no sólo de manejar los eventos de la aplicación, sino también de coordinar al resto de los objetos wrapped que forman parte de ella.

3.5. GroupKit

Es la plataforma más antigua, y es una extensión del lenguaje Tcl/Tk. Éste facilita el desarrollo de aplicaciones que apoyan el trabajo distribuido, en tiempo real, entre dos o más personas **[Roseman 97]**. El componente "*canal*" está implementado por tres tipos de procesos: un servidor principal llamado "*Registrador*", un "*Manejador de Sesiones*" por cada usuario en el sistema, y las aplicaciones que usan los usuarios, llamadas "*Conferencia*".

4. Conclusiones y Trabajo Futuro

En los últimos años se ha comprobado la utilidad del uso de patrones como un medio para mejorar la calidad y el tiempo de desarrollo de las aplicaciones. El patrón arquitectónico presentado es el primero en el área de CSCW, e intenta brindar una solución en esta línea. Éste brinda un marco de referencia para el desarrollo de futuras arquitecturas de apoyo al trabajo colaborativo y sistemas colaborativos en general, además presenta el primer paso en la estructuración del problema. A futuro, es posible formalizar de patrones de diseño **[Gamma 95, Buschmann 96]** específicos para el área de CSCW, basados en el patrón arquitectónico propuesto. También es necesario bajar el nivel de abstracción del problema y formalizar un patrón arquitectónico distribuido (par a par) y otro centralizado (cliente-servidor), específicamente diseñado para el trabajo colaborativo. Estas formalizaciones servirán como guía para la implementación de las futuras plataformas de apoyo al desarrollo de aplicaciones colaborativas. En la medida que se definan mayor cantidad de patrones en esta área, mejor será la calidad de las soluciones alcanzadas.

Reconocimientos

Este trabajo ha sido parcialmente financiado por el Fondo Nacional de Ciencia y Tecnología de Chile (FONDECYT), Proyecto Nº 198-0960.

Referencias

[Adler 95] R. Adler. *Distributed Coordination Models for Client/Server Computing*. IEEE Computer, Apr. 1995.

[Buschmann 96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[Chabert 98] Chabert, A., Grossman, E., Jackson, L., Piertrowiz, S., Seguin, C. *Java Object-Sharing in Habanero*. Communications of the ACM, Vol. 41, Num. 6, pp. 69-76. 1998.

[Ellis 91] Ellis, C., Gibbs, S., Rein, G. *Groupware, Some Issues and Experiences*. Communications of the ACM, Vol. 34, Num. 1, pp. 38-58. 1991.

[Gallardo 97] Gallardo, F. *Evaluación del Rendimiento y Afinamiento de Sistemas Distribuidos de Archivos*. Tésis de Magister. Pontificia Universidad Católica de Chile. Chile. pp. 180-181. Julio 1997.

[Gamma 95] Gamma, E., Helm, R., Johnson, R., Vissides, J. *Design Pattern: Elements of Reusable Object-Oriented Software*. Addison Wesley. 1995.

[Guerrero 98] Guerrero, L., Fuller, D. *Objects for Fast Prototyping of Collaborative Applications*. Proceedings of CRIWG'98, Rio de Janeiro, Brasil, Sep. 1998.

[Jacobson 97] Jacobson, I., Gris, M., Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley. 1997.

[Johansen 88] Johansen, R. *Groupware: Computer Support for Business Teams*. The Free Press, N. Y., 1988.

[Patterson 96] Patterson, J., Day, M., Kucan, J. *Notification Servers for Synchronous Groupware*. Proceedings of CSCW ´96. Boston, Massachusetts, EEUU. Nov. 1996.

[Roseman 97] Roseman, M. and Greenberg, S. *Building Groupware with GroupKit*. In M. Harrison (Ed.) Tcl/Tk Tools, pp. 535-564, O'Reilly Press, 1997.

[Tanenbaum] Tanenbaum, A. *Distributed Operating Systems*. 1º Ed., Prentice Hall. 1996.

[Trevor 97] Trevor, J., Koch, T., Woetzel, G. *MetaWeb: Bringing Synchronous Groupware to the World Wide Web*. Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW'97, Lancaster, 1997.