Un Modelo de Coordinación Cliente-Servidor, de Apoyo a la Construcción de Aplicaciones Colaborativas Portables

Sergio Ochoa[1], David Fuller
Pontificia Universidad Católica de Chile
Escuela de Ingeniería. DCC
Santiago, Chile
{sochoa, dfuller}@ing.puc.cl

Resumen

En la actualidad existe una gran variedad de plataformas de apoyo al desarrollo de aplicaciones colaborativas, que trabajan bajo una arquitectura cliente-servidor **[Chabert 98, Guerrero 98, Licea 98, Trevor 97]**. Estas poseen un alto grado de incompatibilidad entre ellas, lo cual produce una fuerte dependencia de la aplicación cliente, hacia el servidor usado. Debido a esto, las capacidades de las aplicaciones están sometidas a la funcionalidad permitida por el servidor. Esto entorpece el avance de las investigaciones y de los desarrollos, ya que para resolver un problema, antes hay que resolver un conjunto de problemas asociados (sistemas de percepción, manejadores de eventos, etc.). La solución obvia a estos problemas pasa por la portabilidad, por eso que en este trabajo se presenta un *modelo de coordinación*, como referencia para la construcción de aplicaciones colaborativas portables.

Palabras Claves: Modelo de Coordinación, Arquitectura Cliente/Servidor, Framework de Componentes de Software, Aplicaciones Colaborativas, Portabilidad.

1. Introducción

En los últimos años se han consolidado conceptos como la portabilidad de las aplicaciones y el reuso de software, diseños, etc. Esto se debe a la implementación de propuestas que han demostrado su utilidad, como por ejemplo: ODBC, JDBC, Java, JavaBeans, ActiveX, CORBA, etc. **[Pressman 96]**. Otro factor influyente fue la definición de patrones de análisis, diseño y arquitecturas, que ha permitido la reutilización de soluciones a problemas recurrentes **[Buschmann 96, Fowler 97, Gamma 95]**.

Las ventajas del reuso y de la portabilidad han sido ampliamente discutidas por la comunidad científica **[Jacobson 96, Mili 95]**. Éstas básicamente pasan por la reducción de tiempos y costos de desarrollo, y por la posibilidad de abordar desafíos mayores, garantizando la calidad del producto final. En el caso del área de groupware, aún falta trabajo en la formalización de problemas y soluciones, debido a que es un área relativamente nueva. Como un ejemplo de esto, podemos ver una alta incompatibilidad entre las distintas plataformas de desarrollo de este tipo de aplicaciones, *TOP* **[Guerrero 98]**, *COCHI* **[Licea 98],** *MetaWeb* **[Trevor 97]**, *Habanero* **[Chabert 98]**, etc. Aunque también es cierto que todas estas plataformas siguen un *patrón arquitectónico* **[Buschmann 96]** similar, ya que están compuestas por:

- *Un servidor (o plataforma servidora)*: que implementa un conjunto básico de funcionalidades asociadas al manejo de las comunicaciones, conexiones de usuarios, eventos, administración de objetos distribuidos, e implementación de políticas (seguridad, acceso a los recursos, etc.). Muchas de estas capacidades son exportadas, para que puedan ser invocadas por las aplicaciones clientes.
- *Una librería de clases*: que implementa un conjunto de funcionalidades, orientadas a facilitar el desarrollo de aplicaciones cliente, y a aprovechar la funcionalidad ofrecida por la plataforma

[1] Profesor Adjunto, Instituto de Informática, UNSJ, Argentina.

servidora. Ésto le permite al programador abstraerse de los detalles de implementación (manejo de sockets, threads, protocolos de comunicación, eventos, etc.), y hacer un mejor uso de los recursos disponibles.

Este trabajo trata de sacar ventaja de esta similitud, definiendo un *modelo de coordinación* que captura la esencia del escenario de trabajo, de una familia de aplicaciones de groupware. La familia de las aplicaciones colaborativas basadas en arquitecturas cliente-servidor. Este modelo de coordinación, como cualquiera de los otros modelos existentes (Layers, Blackboard, Pipes and Filters, etc.) **[Buschmann 96]**, identifica a los actores que participan en el escenario de trabajo, establece sus responsabilidades, y la dinámica de las comunicaciones entre ellos. Su composición consiste en dos elementos básicos, que permiten a las aplicaciones independizarse de la plataforma servidora que se utilice. El primero es una *arquitectura* de comunicación, basada en la tradicionalmente conocida como cliente-servidor **[Adler 95]**. Ésta identifica a los grandes grupos que participan en el escenario de trabajo, y establece una dinámica de comunicación entre ellos. El segundo elemento es un *framework de componentes de software* **[Roberts 97]**, que realiza la misma función que la arquitectura, pero en forma detallada y sobre la aplicación cliente. Las aplicaciones desarrolladas utilizando este *framework*, y que respetan la dinámica propuesta por la *arquitectura*, son portables entre servidores que cumplan con un conjunto de requisitos mínimos.

Esta propuesta, a diferencia de las planteadas por las plataformas antes mencionadas, presenta un modelo de coordinación genérico, que puede ser usado como referencia para la construcción de nuevas y mejores plataformas de desarrollo. Además, el framework de componentes propuesto está orientado a lograr la portabilidad de las aplicaciones, y no solamente a facilitar la tarea del desarrollador. Hay otros trabajos relacionados, que pertenecen al área de los sistemas de cómputo cliente-servidor y de los sistemas distribuidos, pero éstos no tienen en cuenta la naturaleza colaborativa de las aplicaciones. En la siguiente sección se describe el modelo de coordinación propuesto, y en las secciones posteriores, se profundiza sobre la estructura y el comportamiento de los elementos participantes.

2. Modelo de Coordinación Propuesto

La arquitectura refleja el escenario típico de una gran parte de las aplicaciones colaborativas, donde una aplicación cliente se comunica con un servidor a través de un conjunto de mensajes predefinidos. Estos mensajes son administrados por manejadores de eventos (uno en el cliente, y otro en el servidor) quienes tienen a su cargo la responsabilidad de coordinar los mensajes, y ocultar los detalles de comunicación entre el cliente y el servidor.

Para poder reflejar mejor el escenario de aplicación típico, planteamos una arquitectura basada en el patrón de diseño llamado *Model-View-Controller* (MVC) **[Gamma 95]**, organizada según la estructura propuesta por el patrón arquitectónico conocido como *Client-Server* **[Adler 95]**. El MVC organiza a una aplicación a través de tres componentes (figura 1):

- *Modelo:* Encapsula los datos y el núcleo funcional. Exporta procedimientos a las vistas y a los controladores, y los notifica a cerca de los cambios en su estado.
- *Vista*: Presenta información al usuario. Básicamente es la interfaz de la aplicación.
- *Controlador*: Es el intermediario entre la vista y el modelo. Su función principal consiste en recibir los requerimientos del usuario en forma de eventos, y traducirlos en uno o más requerimientos válidos al modelo.

Por otro lado, debido a que la arquitectura Cliente-Servidor es un modelo de coordinación, identifica a los participantes en el escenario de trabajo, establece sus responsabilidades y la dinámica de las comunicaciones entre ellos. Específicamente establece las características del:

- *Cliente*: Interactúa con un servidor, a través de un conjunto de servicios predefinidos. Cuando un cliente solicita un servicio a un servidor, se bloquea en espera de la respuesta; y sólo continúa su ejecución cuando recibe del servidor una respuesta válida.
- *Servidor*: Recibe solicitudes de servicio por parte de los clientes, las procesa y luego devuelve los resultados a cada solicitante.

En el caso de la arquitectura propuesta (figrura 2), se ha extendido el patrón de diseño MVC a una arquitectura, y luego se ha fusionado con el patrón arquitectónico Cliente-Servidor. El resultado de eso es una arquitectura que se ajusta mejor al esquema de trabajo de las aplicaciones colaborativas planteadas. El cliente está representado por una aplicación que opera a nivel de vistas y controladores, y el servidor, por el modelo que encapsula los datos y gran parte del núcleo funcional. Debido a que nuestra propuesta apunta obtener aplicaciones portables, hemos introducido una variante, que separa al controlador en dos partes: el controlador de la aplicación y el controlador de la vista.

Aquí, las comunicaciones entre los participantes seguirán el esquema general establecido por la arquitectura Cliente-Servidor. Y la secuencia de comunicación siempre será: Vista -- Controlador de Vista -- Controlador de Aplicación -- Servidor. Si un usuario genera a través de la interfaz (*View*), una solicitud de servicio por medio de un evento, este evento es analizado por el controlador de la vista (*ViewController*). Si corresponde, se le envía al controlador de la aplicación (*ApplicationController*), una o más solicitudes orientadas a satisfacer el pedido inicial. Nuevamente si corresponde, se le envía una o más solicitudes equivalentes al *Servidor*. Éste las procesa y devuelve los resultados al *ApplicationController*, quién las retorna siguiendo el mismo camino.

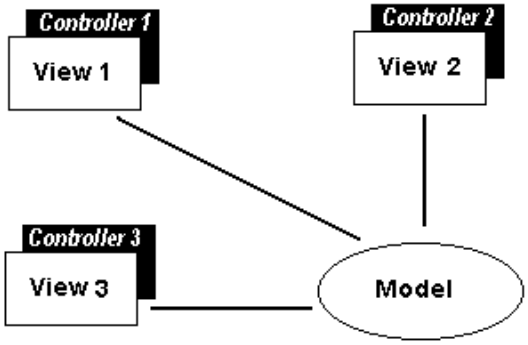


Figura 1. Patrón MVC

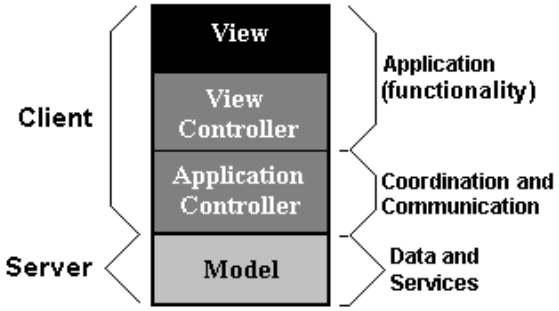


Figura 2. Arquitectura Propuesta

No todas las solicitudes generadas a través de una vista llegan al Servidor, muchas son resueltas por el *ViewController* o por el *ApplicationController*. Sólo las solicitudes que no pueden ser resueltas por los intermediarios (*ViewController* y *ApplicationController*) llegan al Servidor, como por ejemplo: solicitudes de conexión al sistema, actualizaciones de objetos compartidos, etc.

En esta arquitectura, el *ApplicationController* hace de puente entre las aplicaciones cliente y el servidor. Este, implementa una *API (Application Programming Interface)* que es la encargada de exportar un conjunto de servicios estándares, que permite a las aplicaciones cliente comunicarse con el servidor. La *API* traduce este conjunto de solicitudes estándares, en solicitudes equivalentes al servidor. El servidor recibe sólo solicitudes válidas, por lo tanto le responde al *ApplicationController* de la misma manera que a cualquier aplicación propietaria. Está claro que para que la aplicación sea portable, deberá existir una *API* para la plataforma destinataria. Esta *API* es independiente de la aplicación que la utilice. Por lo tanto si existe una *API* para una cierta plataforma servidora, todas las aplicaciones que respeten este modelo de coordinación, serán portables a ésta. A continuación se describe en forma más detallada a cada uno de los componentes de esta arquitectura, su comportamiento, y la relación entre ellos.

Client: Es una aplicación que está compuesta por las tres capas superiores de la arquitectura propuesta (fig. 2). Cada aplicación tendrá un único controlador de aplicación, y una o más

combinaciones de vista-controlador de vista. Toda la funcionalidad propia de la aplicación, estará en los dos últimos niveles de la arquitectura. Y será implementada según la forma de trabajo establecida por el *framework de componentes*, que es descripto en el punto 4.

View (Layer 4): Al igual que en la definición original del patrón MVC **[Gamma 95]**, la vista representa la interfaz con el usuario. Específicamente, una ventana que es controlada por un único controlador de vista.
Controller (Layer 2 and 3): El controlador, como se mencionó antes, está compuesto por el controlador de la vista (ViewController) y el controlador de la aplicación (ApplicationController). Esto responde a una decisión de diseño que apunta a encapsular en el controlador de la vista, todo el manejo de eventos internos propio de la aplicación. Por otra parte, el rol del controlador de aplicación es implementar las funciones de interacción entre la aplicación cliente y el servidor.

ViewController (Layer 3): Realiza la misma tarea que el componente *mediator*, en el patrón de diseño del mismo nombre **[Gamma 95]**. En otras palabras, es el encargado de controlar y coordinar las interacciones entre todos los componentes de la vista, manteniendo en todo momento la coherencia de la misma. Se comunica con el *ApplicationController* para solicitar servicios, para recibir respuestas o notificaciones.
ApplicationController (Layer 2): Además de las funciones ya mencionadas, tiene la responsabilidad de administrar la comunicación entre el cliente y el servidor, de forma similar al componente *broker*, dentro del patrón arquitectónico del mismo nombre **[Gamma 95]**. Los detalles del funcionamiento de este componente, se describen en la próximo sección (punto 4).

Server: Es la plataforma destinada a brindar servicios a aplicaciones clientes, fundamentalmente para la comunicación entre usuarios y para manejar objetos compartidos. Este servidor puede ser COCHI **[Licea 98]**, TOP **[Guerrero 98]**, MetaWeb **[Trevor 97]**, Habanero **[Chabert 98]**, o cualquiera que cumpla con los requisitos impuestos por el modelo.

Model (Layer 1): El modelo está representado por el servidor, y es el encargado de encapsular los datos y gran parte del núcleo funcional. Éste se comunica con la aplicación a través del *ApplicationController*. Bajo este esquema, no todos los servidores son aceptables, ya que éstos deben cumplir con el siguiente conjunto de requisitos mínimos, para que los servicios estándares definidos en la *API*, puedan ser implementados:

- Creación, eliminación y administración de sesiones.
- Administración, altas y bajas de conexiones de usuarios a sesiones.
- Creación, eliminación y actualización de objetos compartidos en forma sincrónica y asincrónica (Semántica Unix) **[Tanenbaum 96]**.
- Administración del control de piso sobre objetos compartidos.
- Mecanismos de bloqueo sobre objetos compartidos.
- Creación, eliminación, actualización y administración de objetos públicos y privados.
- Comunicación (Sincrónica/Asincrónica) entre los usuarios y entre aplicaciones, punto a punto y multicast.
- Manejo de transacciones Cliente-Servidor.

En la siguiente sección se describe al controlador de aplicaciones, que es el componente más activo, en la tarea en la tarea de construir aplicaciones portables.

3. ApplicationController

Este componente es el más complejo de todo el modelo, ya que es el encargado de desvincular a la aplicación cliente del servidor que le presta servicio. Éste encapsula toda la funcionalidad necesaria

para que cualquier solicitud de una aplicación, pueda ser procesada por el servidor. El siguiente diagrama de clases, descripto en UML (Unified Modeling Language) **[Uml 97]**, muestra la composición del *ApplicationController* y las clases que interactúan con él.

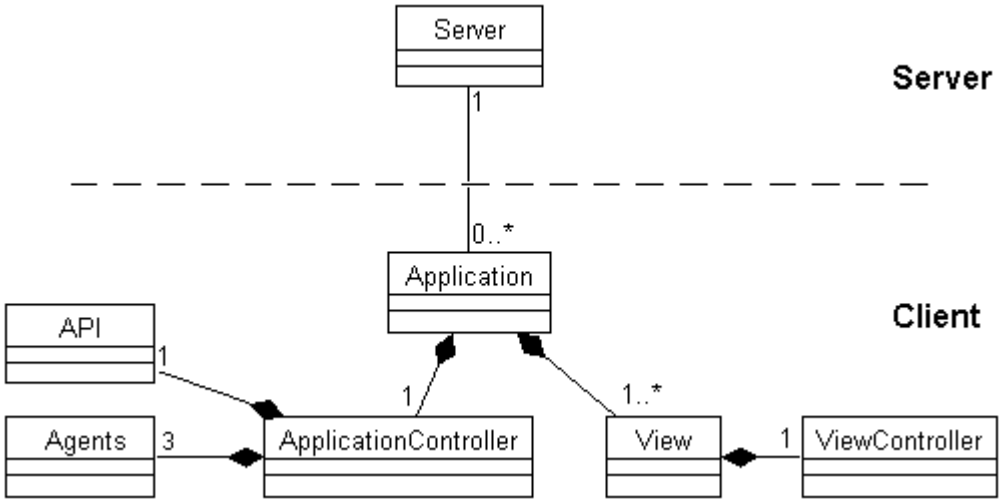


Figure 3. Diagrama de Clases de los Participantes en el Modelo

Como se puede ver en la figura 3, una aplicación tiene un único controlador de aplicaciones, y 1 o más vistas. El *ApplicationController* está compuesto por:

- *Agentes:* Hay tres agentes al servicio del *ApplicationController*. El primero escucha el canal de comunicación con los *ViewControllers*. El segundo escucha el canal de comunicación Servidor. Y el tercero, procesa requerimientos y los despacha hacia las vistas o hacia el servidor, según corresponda.
- *API:* Implementa dos conjuntos de funciones estándares. El primero sirve para que las aplicaciones clientes se comuniquen con el servidor. Y el segundo, sirve para manejar los servicios que el *ApplicationController* les brinda a las aplicaciones clientes. Para que una aplicación pueda ser portada a otra plataforma servidora, sólo se necesita que el primer conjunto de funciones de la *API* esté implementado sobre la nueva plataforma. Además, debido que la interfaz entre la *API* y el *ViewController* está bien definida, es posible cambiar una *API*, por otra que implemente las funciones en forma más eficiente. Esto no afectará al normal comportamiento de la aplicación.

El *ApplicationController* tiene además un núcleo de funcionalidad, que es el encargado del manejo de la información interna (que él almacena), los agentes, los servicios, y las particularidades de funcionamiento de la aplicación.

4. Framework de Componentes

Como se mencionó antes, el framework tiene como primer objetivo apoyar a la arquitectura, en el desarrollo de aplicaciones portables. Y como segundo objetivo, facilitar el desarrollo de aplicaciones colaborativas, abstrayendo al programador de detalles de funcionamiento de la arquitectura. Para cumplir con esto, se han diseñado un conjunto de *componentes* **[Booch 98, Brown 97]** que resumen las formas de interacción básicas en los sistemas colaborativos **[Antillanca 99, Favela 97]**.

Los componentes son "*piezas de software que soportan un conjunto de comportamientos estándares, independientemente de la funcionalidad específica para la que han sido diseñados*" **[Vander Veer 97]**. Éstos, han sido organizados en un *framework* que agrupa componentes relacionados, y le agrega semántica a estas relaciones entre ellos (formas de interacción válidas). Roberts y Johnson definen a un framework como *"diseños reusables de partes o de todo un sistema de software, descripto por un*

conjunto de clases abstractas, y por la manera en que las instancias de ellas colaboran" **[Roberts 97]**. En este caso, se trata de los escenarios de trabajo de las aplicaciones colaborativas cliente, que interactúan a través de la arquitectura propuesta.

El framework propuesto es genérico, y puede ser implementado de muchas formas, y sobre distintos lenguajes de programación. En este caso, se ha elegido implementarlo a través de componentes de software, en el sentido de *"elementos de programas que pueden ser usados por otros elementos de programas, sin necesidad de que se conozcan los autores de éstos"* **[Meyer 99]**. Específicamente se eligió la especificación JavaBean **[Sun 97]**, que "*componentes de software funcionalmente discretos, con los cuales se pueden construir complejas aplicaciones basadas en java*" **[Vander Veer 97],** y están orientados a "*aliviar los problemas de interfaces, mejorando la consistencia de la Interfaz de Usuario (UI), la productividad del desarrollador y el reuso de código*" **[Valdés 97]**. Además, "*tienen la capacidad de poder transportar el conocimiento almacenado en ellos*" **[Crane 97]**. Esto permite incorporarles funcionalidad adicional, para el manejo de información sobre conexiones, percepción (awareness), eventos e interactividad.

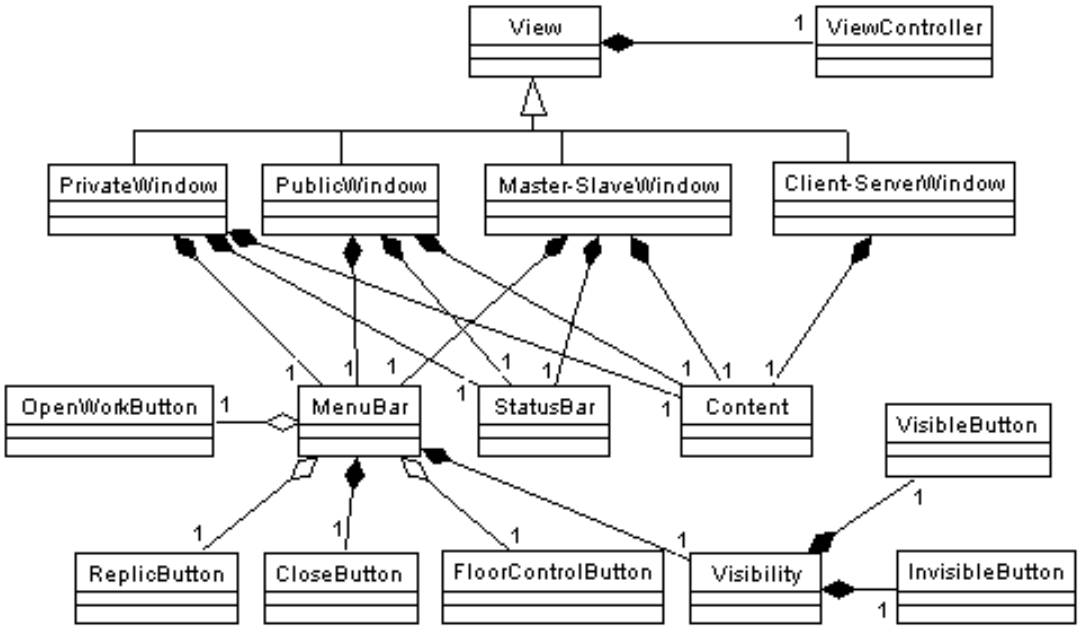
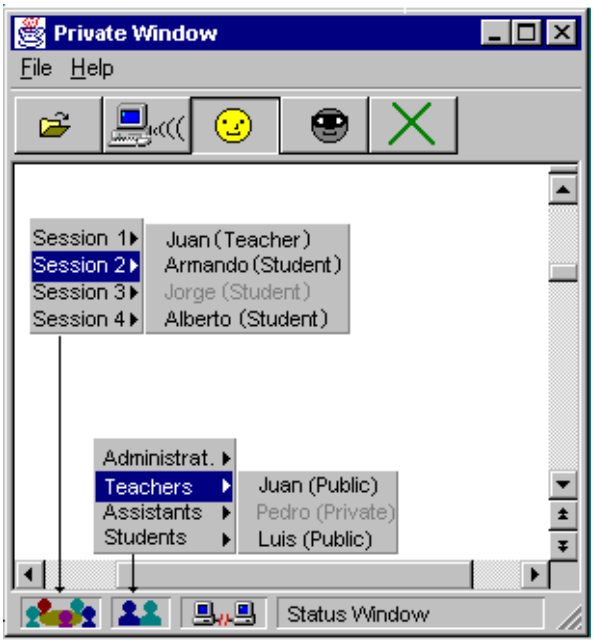


Figura 4. Esquema general del framework de componentes propuesto

El diagrama de clases de la *figura 4*, refleja la composición general del framework propuesto, mostrando las clases participantes y las relaciones entre ellas. La semántica de las relaciones se describirá junto con los componentes involucrados. Los componentes del framework son en su mayoría gráficos, y esto se debe a que "*la interfaz gráfica de usuario (GUI) requiere la mayor parte del esfuerzo de desarrollo del producto*" **[Myers 88]**. Además, "*la calidad del diseño de una interfaz gráfica de usuario (GUI) refleja el conocimiento especializado a cerca de factores humanos, psicológicos, fisiológicos y de diseño gráfico*" **[Schneiderman 98]**, que puede ser almacenado y reutilizado a través del diseño de componentes apropiados.

Los componentes del tipo ventanas, han sido diseñados para soportar las distintas formas de interacción clásicas en aplicaciones colaborativas **[Favela 97]**. Éstos encapsulan los dos niveles superiores de la arquitectura propuesta (fig.2). Su interfaz (*View*) ha sido diseñada siguiendo el patrón de "*Organización y Estructura Visual*" **[Sano 95, Schneiderman 98]**, para garantizar un alto grado de coherencia de la interfaz de la aplicación. Su comportamiento (*ViewController*), ha sido diseñado para realizar su función específica, ajustándose a la dinámica definida para la arquitectura propuesta. La combinación *vista-controlador de vista* (fig.3), está encapsulada en las ventanas que se describen a continuación (para mayor detalle ver **[Ochoa 98]**):

PrivateWindow*:** Este tipo de ventana permite a un usuario trabajar en forma aislada del resto, dándole la posibilidad de compartir su trabajo cuando él lo desee. Cuando un usuario está trabajando en una *PrivateWindow*, el *ViewController* asociado a su ventana, registra en una bitácora (*Binnacle)* todas las operaciones de actualización realizadas sobre los objetos compartidos. En la figura 5 se muestra una implementación de la *PrivateWindow*. Esta ventana está compuesta por otros componentes (figura 4) como la barra de menú (MenuBar***), la barra de estado (***StatusBar***), y el contenido (***Content***).



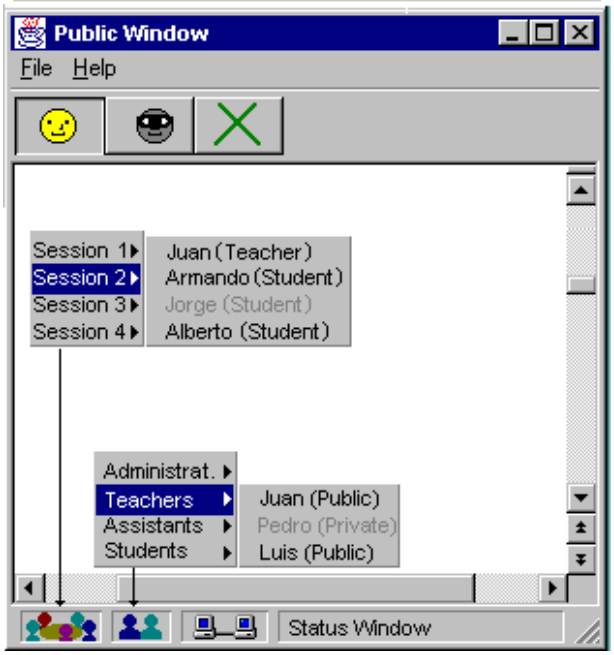


Figura 5. Prototipo de Ventana Privada **Figura 6.** Prototipo de Ventana Pública

Este tipo de ventana captura la esencia de la interacción *asincrónica* en aplicaciones colaborativas. Y puede ser utilizado en aplicaciones como Electronic Meeting Systems (EMS), que involucran una cuota de trabajo asincrónico **[Nunamaker 91]**.

***PublicWindow*:** En este tipo de ventana (fig. 6) todas las operaciones de actualización realizadas sobre objetos compartidos, son replicadas en forma sincrónica, a un grupo limitado de usuarios

(multicast). Durante una replicación, estas operaciones son enviadas al servidor a través del *ApplicationController*, quien de acuerdo al tipo de operación, tipo de ventana, usuario y sesión, genera un conjunto de solicitudes equivalentes para el servidor actual.

Master-SlaveWindow: Este tipo de ventana sigue el tradicional modelo *Master-Slave* **[Tanenbaum 96],** donde sólo un usuario a la vez, puede tener el permiso de uso activo (piso) de la aplicación. El cliente poseedor del permiso (piso) envía, al servidor las operaciones que realiza en forma local, a través del *ApplicationController*. El servidor las replica, en forma sincrónica, al resto de los clientes conectados a la sesión del emisor (multicast). Los clientes pasivos (receptores) pueden solicitar el control de piso al servidor, que es el encargado administrar el recurso, utilizando alguna política preestablecida. Este tipo de ventana puede ser utilizada para cualquier aplicación que respete el modelo Maestro-Esclavo, como por ejemplo en una para realizar presentaciones distribuidas.

Client-ServerWindow: Esta ventana es la encargada de realizar transacciones cliente-servidor puras, a través del *ApplicationController*. Está compuesta únicamente por las funciones de comunicación con el *ApplicationController* y por el componente contenido (*Content*), que puede ser especializado según las necesidades (figura 4).

4. Conclusiones

El modelo de coordinación presentado, sirve como guía para el desarrollo de aplicaciones colaborativas portables, en arquitecturas cliente-servidor. Éste no sólo ayuda a portar aplicaciones entre distintas plataformas servidoras, sino que también aporta un marco de referencia para el desarrollo de nuevas y mejores plataformas futuras. Por otra parte, el framework de componentes propuesto, también puede ser tomado como referencia para el desarrollo de nuevos frameworks, o para el mejoramiento de los existentes. Este framework no sólo facilita el desarrollo de aplicaciones, sino que además alienta el desarrollo de software basado en componentes, y ofrece la posibilidad de extenderlo o especializarlo en cualquier área.

Una desventaja de esta solución es que no aprovecha al máximo las capacidades de las plataformas servidoras, ya que la comunicación se basa en un conjunto de servicios estándares. Otra desventaja, consiste en que las aplicaciones construidas utilizando el framework, tendrán un rendimiento menor que las que accedan directamente al servidor, debido a que no tienen que utilizar los controladores como intermediarios.

A futuro, este modelo puede ser extendida a múltiples servidores, con sólo agregarle funcionalidad adicional al *ApplicationController*. Además, es posible mejorar la ejecución de operaciones, exigiendo bloqueo sólo en los casos donde no hay otra alternativa. También se puede mejorar en lo referente a la semántica de las operaciones, ya que la semántica Unix puede ser dura cuando se trata de replicar el trabajo asincrónico (realizado en una ventana privada).

Es importante trabajar en la formalización de problemas y soluciones recurrentes en el área de groupware (arquitecturas de software, modelos de coordinación, patrones de diseño, etc.), porque esto va a ayudar a acordar caminos, hacia la definición de futuros estándares para esta área.

5. Agradecimientos

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

6. Referencias

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

[Antillanca 99] H. Antillanca, D. Fuller. *Refining Temporal Criteria to Classify Collaborative Systems*. International Journal of Human-Computer Studies, Vol. 50, No. 1, Jan. 1999, pp. 1-40.

[Booch 98] G. Booch, W. Kozaczynski. *Component-Based Software Engineering*. IEEE Software, Vol.15, Nro.5, pp34-36, Sep/Oct. 1998.

[Brown 97] A. Brown, K. Short. *On Components and Objects: Foundations of Component-Based Development.* Proc. of the 5th International Symposium on Assesment of Software Tools. IEEE Computer Society Press. Jun. 1997.

[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., Pietrowicz, S., Seguin, C. "*Java Object-Sharing in Habanero*". Communications of the ACM, Vol. 41, Jun. 1998.

[Crane 97] A. Crane, S. Clyde. "*Extending Patterns for GUI Design*". Research Paper. Utah State University. 1997.

[Favela 97] Favela, J., Soriano, M., Zapata, M. 1997. *A Design Space and Development Process for Collaborative Systems.* Proc. of CRIWG 97. Third CYTED-RITOS International Workshop on Groupware. San Lorenzo del Escorial, Madrid, Spain.

[Fowler 97] Fowler, M. 1997. "*Analysis Patterns*". Addison-Wesley.

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

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

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

[Licea 98] Licea, G., Favela, J. 1998. *Design Patterns for the Development of Synchronous Collaborative Systems*. Taller Internacional de Tecnología de Software, Simposium Internacional de Computación CIC-98. México D.F., Nov. 1998.

[Meyer 99] Meyer, B. 1999. *On to Components*. IEEE Computer, Jan. 1999.

[Mili 95] Mili, H., Mili, F., Mili, A. 1995. *Reusing Software: Issues and Research Directions*. IEEE TOSE 21, No. 6, Jun. 1995, 528-561.

[Myers 88] Myers, B. 1988. *Creating User Interfaces by Demostration*. Perspectives in Computing, Vol. 22, Academic Press, Inc. San Diego, CA.

[Nunamaker 91] Nunamaker, J., Dennis, A., Valacich, J., Vogel, D., George, J. 1991. *Electronic Meeting Systems to Support Group Work*. CACM, July 1991. Págs. 40-61.

[Ochoa 98] S. Ochoa, G. Licea, D. Fuller, J. Favela. *Widgets: Un Estilo Diferente para Crear Interfaces de Usuario Colaborativas.* XXIV Conferencia Latinoamericana de Informática, CLEI 98, Quito, Ecuador, 19 al 23 de Octubre de 1998. Págs. 653-665.

[Pressman 96] Pressman, R. 1996. *Software Engineering : A Practitioner's Approach*, 4º Ed., Mc. Graw Hill.

[Roberts 97] Roberts,D. , Johnson, R. 1997. *Patterns for Evolving Frameworks.*. Proc. of PLOP-3 (Pattern Language Oriented Programing).

[Sano 95] Sano, D., Mullet, K. 1995. *Designing Visual Interfaces: Communication Oriented Techniques*. SunSoft Press, Prentice Hall.

[Schneiderman 98] B. Schneiderman. 1998. *Designing User Interface: Estrategies for Effective Human Computer Interaction*, 3º Ed. Addison-Wesley Publising Company.

[Sun 97] Sun Microsystem Inc. 1997. *JavaBeans Specification, Version 1.01*. Jul. 1997. Graham Hamilton (Editor).

[Tanenbaum 96] Tanenbaum, A. 1996. *Distributed Operating Systems*. 1º Ed., Prentice Hall.

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

[Uml 97] *UML Notation Guide, version 1.1*, Sep. 1997, URL: http://www.rational.com/uml/resources/documentation/notation/index.jtmpl

[Vander Veer 97] Vander Veer, E. 1997. *Beans Basics*. Web Techniques Magazine, Vol. 2., Oct. 1997.

[Valdés 97] Valdés, R. 1997. *Creating Visual Components with JavaBeans*. Web Techniques Magazine, Vol. 2., Oct. 1997.