Un Modelo de Comunicación para Aplicaciones Colaborativas

Luis A. Guerrero, Sergio F. Ochoa[1], Oriel A. Herrera[2], David A. Fuller
{luguerre, 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. En el presente artículo se propone un esquema de comunicación que permite modelar los mecanismos de distribución de mensajes que se requieren en todos los sistemas colaborativos. Estos mecanismos proveen comunicación tanto entre usuarios como entre procesos del sistema a través del cual se está colaborando.

Palabras Clave: Sistemas Colaborativos, Modelo de Comunicación, CSCW.

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 **[Ell91]**. Los sistemas colaborativos proveen las herramientas necesarias para que un grupo de usuarios se comunique y trabaje de forma coordinada. La comunicación es, por lo tanto, un aspecto fundamental tanto en el diseño como en la construcción de las aplicaciones colaborativas. Según la taxonomía de tiempo-espacio presentada por Johansen **[Joh88]**, 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 modelo presentado en este artículo, aporta un esquema de comunicación genérico, para soportar ambos tipos de comunicación.

En todo proceso de comunicación se tiene básicamente un *emisor*, un *receptor*, un *mensaje* y un *canal*. En ese escenario, un emisor se comunica con un receptor por medio de mensajes que son enviados a través del canal. Aquí, tanto el emisor como el receptor puede ser un proceso de usuario o un proceso interno. Un *proceso de usuario* es aquel que interactúa con el sistema siguiendo las órdenes indicadas por el usuario. En cambio un *proceso interno*, es aquel proceso que interactúa con el sistema, y que no refleja una orden explícita del usuario (sincronización de eventos, actualización de objetos compartidos, etc.).

---

[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

El canal es el responsable de la distribución de los mensajes entre el emisor y el receptor. Esta distribución podrá realizarse de dos formas: sincrónicamente o asincrónicamente, según el tipo de colaboración definido para los usuarios. Además, entenderemos por *mensaje* a cualquier objeto que pueda ser representado por un flujo de bytes, de tal modo que pueda ser enviado a través de una red de computadores, a un grupo de uno o más destinatarios.

La presente investigación sobre el proceso de comunicación en aplicaciones colaborativas está enfocada en el componente *canal*, más que en los componentes *emisor*, *receptor* y *mensaje*. En la sección 2 se muestra la arquitectura del modelo propuesto. En la sección 3 se muestra una implementación de éste. En la sección 4 se muestra cómo algunos de los principales sistemas actuales pueden ser especificados a partir de este modelo. Finalmente en la sección 5 se presentan las conclusiones y algunos tópicos de trabajo futuro.

2. Definición del Modelo

La estrategia de comunicación en aplicaciones colaborativas, tiende a estar fuertemente acoplada a la implementación utilizada para el canal. Esto le resta generalidad a las soluciones propuestas, limitando su reusabilidad y forzándolas a soportar sólo los tipos de interacción implementados por ese canal. Esto no sólo aumenta la complejidad de la solución (en cuanto al esfuerzo de desarrollo y de mantenibilidad), sino que además aporta un conjunto de debilidades sobre la propuesta (rendimiento pobre, interacción poco natural, etc.). Para atacar este problema, se propone un modelo que establece claramente el esquema de comunicación que deben respetar las aplicaciones colaborativas, para soportar los dos tipos de interacciones antes definidas (sincrónica y asincrónica) independientemente de la arquitectura que utilicen. De esa manera, al menos las ideas podrán ser reutilizadas, tanto por plataformas de desarrollo (ver punto 4) como por aplicaciones específicas.

El escenario de trabajo de las aplicaciones colaborativas (ver figura 1) 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. La comunicación entre las aplicaciones se realiza a través de mensajes, y tanto el envío como la recepción de éstos son considerados eventos. Un *evento* es todo estímulo externo reconocible por la aplicación.

Está claro que la aplicación debe manejar eventos, y para esto hay dos posibilidades: incorporar el manejo de eventos al código de la aplicación, o encapsularlo a través de un *manejador de eventos* capaz de brindar ese servicio a cualquier aplicación. Es obvio que la segunda opción es mejor que la primera, no sólo por las ventajas en reusabilidad, legibilidad y mantenibilidad, sino porque el manejo de eventos es una función genérica que no depende de una aplicación en particular. Debido a esto, en nuestro modelo las aplicaciones interactúan entre ellas a través de un *manejador de eventos*, que es el encargado de enviar y recibir los mensajes que otras aplicaciones envían por el canal. Cuando un emisor desea enviar un mensaje a un grupo, debe especificar: los receptores, el contenido a distribuir, y el modo de distribución (sincrónico o asincrónico). Esta especificación (*mensaje*, figura 1) es entregada al manejador de eventos local, luego es traducida a una especificación equivalente (*mensaje'*, figura 1) entendible por el canal, y posteriormente es enviada en forma de mensaje. El canal revisa el conjunto de destinatarios y lo distribuye según la forma especificada. Si el mensaje es sincrónico, el canal hará de puente entre el emisor y el receptor. En cambio si es asincrónico, el canal validará la integridad del mensaje, y si está correcto, aceptará la

responsabilidad sobre la entrega del mensaje. El canal sólo reportará al emisor los errores que pudieran surgir de la comunicación entre él y el manejador de eventos del receptor.

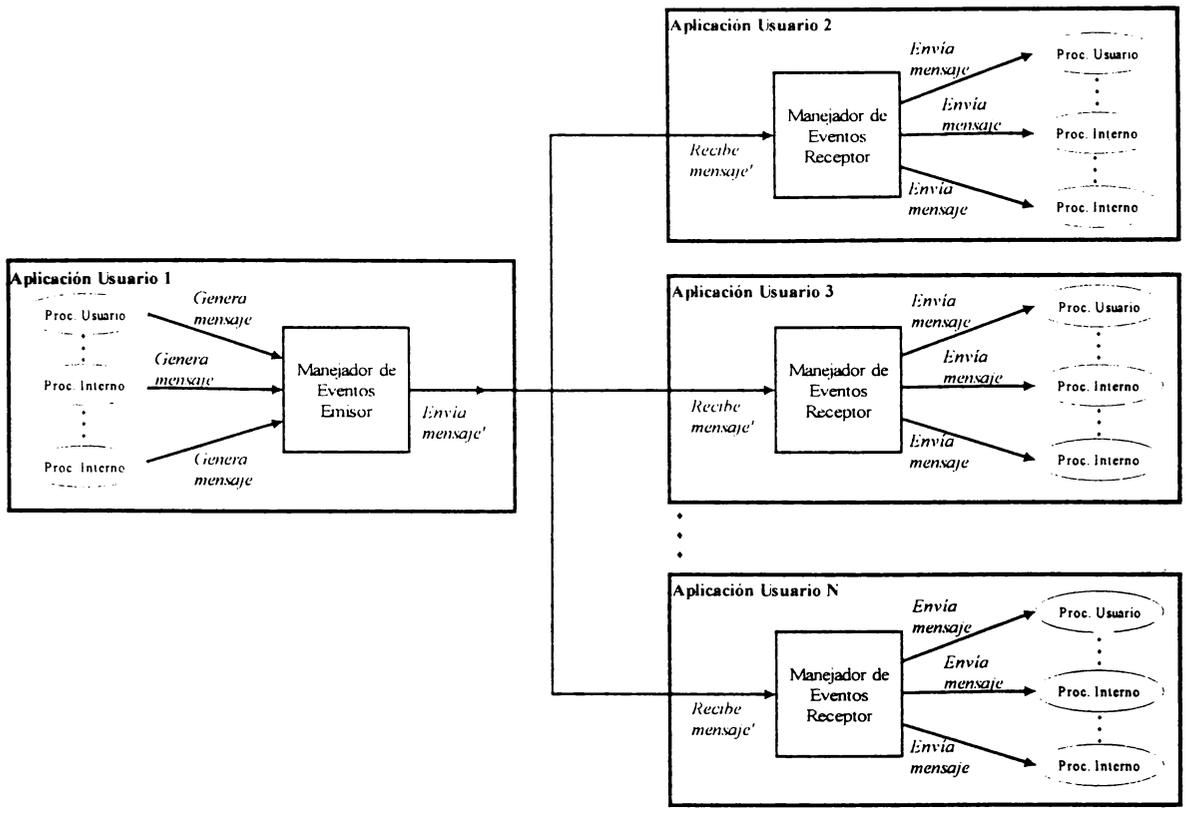

Figura 1. Esquema de Comunicación Propuesto

Una vez que el manejador de eventos del receptor recibe el mensaje (*mensaje'*), analiza si la aplicación destino está disponible. Si es así, el mensaje es entregado a la aplicación, y se retorna al canal la retroalimentación acerca de la operación realizada. En caso contrario se revisa el modo de distribución asociado al mensaje. Si el modo es sincrónico, el mensaje de retroalimentación retornado al canal indicará un error en la disponibilidad de la aplicación destino. Si por el contrario es asincrónico, la retroalimentación indicará que el mensaje no ha podido ser entregado, por lo tanto el canal se tendrá que hacer cargo de la persistencia y de la posterior redistribución a los destinatarios que no lo recibieron.

Básicamente, esa es la dinámica de la comunicación que respetan la mayoría de los sistemas colaborativos. La diferencia entre ellos radica en la forma en que se implementa el canal, para lo cual hay dos enfoques a seguir: *cliente-servidor* o *par a par.* Las ventajas y desventajas de cada una de estas estratégias de comunicación han sido ampliamente discutidas por investigadores del área de los sistemas distribuidos **[Tan96]**, por lo que no abordaremos este tema.

En la figura 2 se muestra la estructura de clases que representa el modelo de comunicación típico entre aplicaciones colaborativas. Allí, el proceso de comunicación se lleva a cabo entre dos aplicaciones (emisor y receptor) que se comunican por medio de mensajes que son enviados a través de un canal. La comunicación sigue siempre la misma secuencia: proceso emisor → manejador de eventos del emisor → canal → manejador de eventos del receptor → proceso receptor. Puede ser que, al igual que en los sistemas estratificados, la comunicación

se produzca entre dos niveles intermedios (por ejemplo: manejadores de eventos). Esto indica que es posible que se dé cualquier segmento de la secuencia de comunicación antes mencionada, y que no siempre hay necesidad de requerir la intervención de un usuario. La secuencia siempre es respetada.

Si bajamos en el nivel de abstracción es posible ver distintas estratégias de implementación de los manejadores de eventos. Éstas van desde las totalmente propietarias y cerradas, hasta aquellas que utilizan APIs (Application Programing Interface) y agentes de inteligencia artificial para lograr protabilidad.

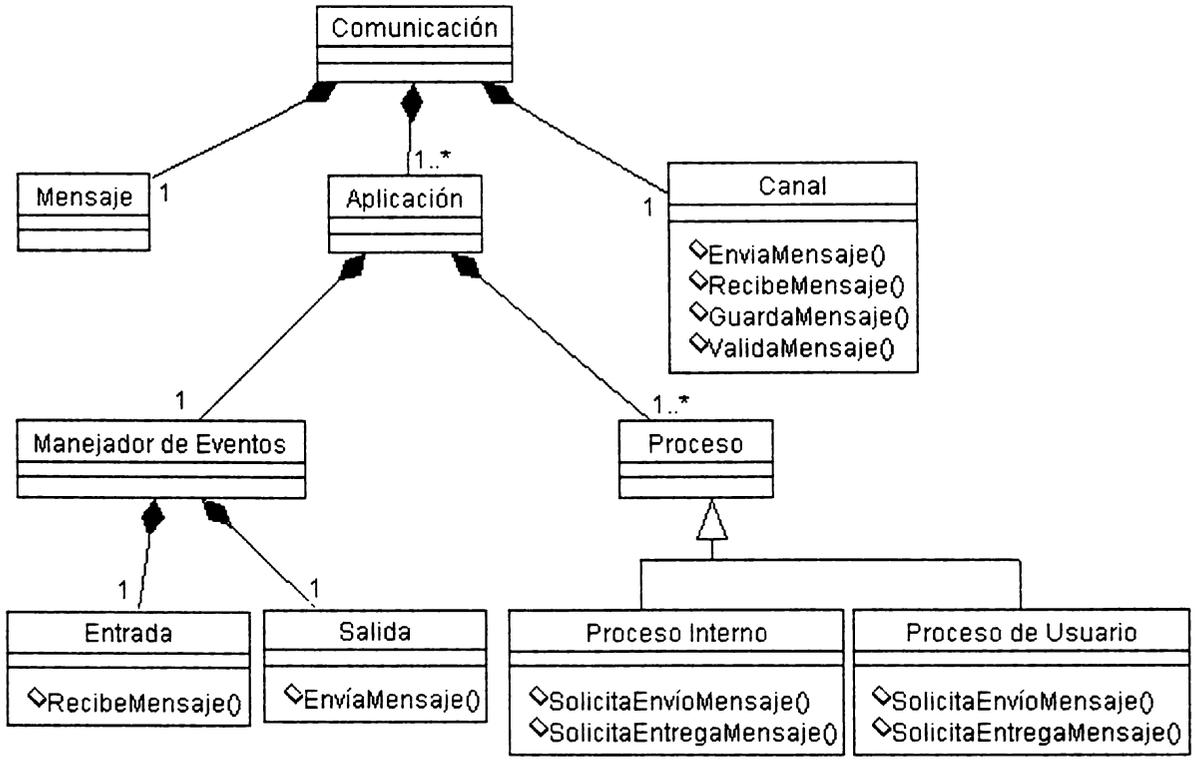

Figura 2. Diagrama de Clases del Modelo

En nuestro modelo de comunicación el emisor y el receptor son procesos (internos o de usuario) que interactúan a través de manejadores de eventos. El canal de comunicación es capaz de manejar comunicación sincrónica y asincrónica, y de almacenar mensajes en caso de ser necesario. Esto nos permite generalizar nuestro problema y asignar roles a cada uno de los elementos participantes. Luego estos elementos podrán ser implementados de distintas maneras, pero siempre deberán respetar el esquema de comunicación y las responsabilidades y atribuciones asignadas.

La mayor parte de la complejidad del desarrollo de aplicaciones colaborativas consiste en la implementación del *canal* y del *manejador de eventos*. Para ello se deben tomar un conjunto importante de decisiones de diseño, como por ejemplo: un canal centralizado (Cliente-Servidor) vs. distribuido (Par a Par); mensajes con persistencia en el canal vs. persistencia en el manejador de eventos; manejadores de eventos inteligentes vs. aplicaciones inteligentes; utilización de APIs vs. operaciones a medida. Una vez tomadas estas decisiones la complejidad de desarrollar sistemas colaborativos baja enormemente debido a que:

1. El modelo de comunicación ha sido estructurado y la estrategia de comunicación está explícitamente definida.
2. Es posible reutilizar desarrollos anteriores sobre la misma plataforma o sobre una distinta. Esta reutilización puede hacerse a nivel de ideas, diseños o incluso de código fuente.

Este ingrediente adicional de la reutilización es muy importante en el área de groupware, ya que por ser un área nueva, aún no se han definido patrones de diseño ni arquitectónicos **[Bus96]** capaces de reflejar la realidad de este tipo de aplicaciones.

2.1 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*. Esto significa que la tasa de errores en mensajes transmitidos por ese canal, debe mantenerse dentro de límites despreciables (menores al 0.025 %) **[Gal97]**. 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 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.

2.2. 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 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.

2.3. 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 **[Och99]**. 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 poder 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.
> **Entregar mensajes:** a los procesos destinatarios que forman 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.

3. Implementaciones

En esta sección se presenta una implementación del modelo propuesto, dónde el canal ha sido centralizado a través de un servidor. Los componentes son los siguientes:

> **TOP Server (Ten Object Platform) [Gue98]:** que cumple el rol de *canal*. Para más información revisar el punto 4.1.
> **TopInterface:** que cumple el rol de *manejador de eventos*. Éste fue desarrollado como un Java applet, y permite la comunicación entre aplicaciones Web, bajo una arquitectura Cliente-Servidor.
> **Aplicaciones:** que están escritas en HTML y Javascript, y utilizan a *TopInterface* para interactuar con otras aplicaciones a través de un servidor *TOP* (canal). Estas aplicaciones pueden ser de cualquier tipo (chat, presentadores distribuidos, foros de discusión, etc.), y pueden involucrar tanto comunicación sincrónica como asincrónica.

A continuación se describe el componente *TopInterface* en detalle. El siguiente extracto de código muestra las variables, el método de inicialización, el destructor y el método de comunicación con el servidor (canal).

```
import java.applet.Applet;
import java.net.*;
import java.io.*;
import netscape.javascript.JSObject;
import java.lang.Integer;

public class TopInterface extends Applet implements Runnable {
```

```
  String srvname;                    // Server name
  int srvport;                       // Server port
  JSObject handWindow;               // Handler for the current browser window
  Socket s;                          // Communication socket
  DataInputStream i;                 // Communication input
  PrintStream o;                     // Communication output
  Thread notificator;                // For notifications handling
  ServerSocket socketClient;         // For notifications handling
  private int ntPort;

  public void init() {
  // Applet initialization
    handWindow = JSObject.getWindow(this); //handler for netscape window
    URL appletCodeBase = this.getCodeBase();
    srvname = appletCodeBase.getHost();
    srvport = Integer.parseInt(getParameter("WebServerPort"));
    ntPort = Integer.parseInt(getParameter("notificationsPort"));
    if (ntPort != 0) {
      notificator = new Thread(this);
      notificator.start();
    }
}

  public void destroy() {
  // Destroy the thread
    if (ntPort != 0)
      notificator.stop();
  }

  public String requestService(String service) {
  // Java server request services method
    String answer = "";
    try {
        this.s = new Socket (srvname,srvport);
        this.i = new DataInputStream(new BufferedInputStream(s.getInputStream()));
        this.o = new PrintStream(new BufferedOutputStream(s.getOutputStream()));
        o.println(service);      // Send the message
       o.flush();
        answer = i.readLine();  // Wait for an ACK
        s.close();
        } catch (IOException ex) {
          answer = "ERROR: " + ex.toString();
          }
     return(answer);
  }
}
```

El método de inicialización requiere como parámetro si se desea o no recibir notificaciones. En caso de querer recibir notificaciones, el applet crea un thread para capturar los eventos que llegan al puerto de notificación.

Un típico llamado al applet TopInterface desde una hoja HTML es el siguiente:

```
<APPLET CODE="TopInterface.class" WIDTH=0 HEIGHT=0 NAME="TopInterface" MAYSCRIPT>
<PARAM name="WebServerPort" value="1234">
<PARAM name="notificationsPort" value="9595">
</APPLET>
```

En este ejemplo se indica que la interfaz va a recibir notificaciones en el puerto 9595. En caso de no querer usar este servicio, se debe pasar como parámetro el puerto 0. También se pasa el puerto en que escucha el servidor. El método destructor detiene y elimina el thread notificador de eventos, en caso de haberse creado. Cada vez que el cliente envía un mensaje al applet, éste lo pasa al servidor, espera la respuesta y la regresa nuevamente al proceso que invocó el servicio. De esta forma, se pueden tener funciones, por ejemplo en JavaScript, que invoquen métodos de un servidor Java. El siguiente ejemplo muestra una invocación de servicios desde una hoja HTML con una función JavaScript.

```
function fnCreateBox() {
if ((document.frm.name.value != "") && confirm("Create a new box?")) {
    msg = "createNewBox:" + env + "," + session + "," + document.frm.name.value
    serverMsg = top.menu.document.TopInterface.requestService(msg)
    if (serverMsg != "OK")
       alert(serverMsg)
    else
       alert("The box object have been created!")
}
```

En el ejemplo anterior se invoca, desde JavaScript, un método para crear un objeto *box* en un servidor hecho en Java, imprimiendo luego un mensaje con el resultado de la operación. El programador de la interfaz no tiene que preocuparse de los aspectos de comunicación con el servidor. Solo invoca al applet y usa los métodos que el servidor provee.

En cierto tipo de aplicaciones, por ejemplo en sistemas colaborativos, las notificaciones a los usuarios de ciertos eventos que ocurren, son fundamentales. El applet TopInterface también implementa un mecanismo de manejo de notificaciones, de modo que si una notificación llega desde el servidor, el applet permite tomarla y propagarla al manejador de eventos definido en el cliente.

El siguiente fragmento de código muestra el mecanismo usado en el applet para recibir las notificaciones del servidor, y transmitirlas al cliente.

```
public void run() {
    try {
      socketClient = new ServerSocket(ntPort);
       while (true) {
         Socket socketClass = socketClient.accept();
         DataInputStream dis = new DataInputStream(new BufferedInputStream _
               (socketClass.getInputStream()));
         String notify = dis.readLine();
         String args[] = {""} ;
         args[0] = notify ;
         handWindow.call("notificator",args);
         socketClass.close();
       }
    } catch (IOException ex) {
         ex.printStackTrace();
       } finally {
          validate();
          try {
             socketClient.close();
          } catch (IOException ex) {
               ex.printStackTrace();
             }
        }
  }
```

En el código anterior, el applet recibe la notificación en la variable `notify` y la envía a una función llamada `notificator(args)` que debe estar definida en el cliente. Por ejemplo, el siguiente fragmento de código JavaScript (de una aplicación chat), define esta función manejadora de eventos.

```
function notificator(notify) {
  if (notify.substring(0,13)=="<chatMessage>") {
    document.formData.text_chat.value = notify.substring(13) + "\n" +
       document.formData.text_chat.value
  }
  else
    if (notify.substring(0,13)=="<userArrived>") {
      newUser = new Option(notify.substring(13))
      fnInsertUser(newUser)
```

```
    }
  else
    if (notify.substring(0,12)=="<userDeparture>") {
      userDepart = notify.substring(12)
      fnRemoveUser(userDepart)
    }
}
```

En el ejemplo anterior la función `notificator` o manejador de eventos, maneja tres tipos de notificaciones: la llegada de un mensaje de texto, el arribo de un nuevo usuario y la partida de un usuario.

4. Trabajos Relacionados

Debido a que los sistemas distribuidos forman parte de CSCW, muchos de los trabajos en esta área necesariamente están relacionados con el modelo propuesto. Aunque ellos no tienen en cuenta la variedad ni la heterogeneidad de la comunicación entre procesos. Tampoco reparan en la actividad de colaboración entre usuarios, que es el principal motivo de desarrollo de aplicaciones en CSCW.

Por otra parte, en los últimos años han aparecido varias implementaciones de la combinación "Canal-Manejador de Eventos", éstas son comúnmente conocidas como "plataformas de apoyo al desarrollo de aplicaciones colaborativas" o GP (Groupware Platforms). Su objetivo es abstraer al programador de todos los detalles de implementación, liberándolo de toda la problemática antes mencionada. Las plataformas más reconocidas son: GroupKit **[Ros97]**, Habanero **[Cha98]**, MetaWeb **[Tre97]**, NSTP **[Pat96]** y TOP **[Gue98]**. A continuación se muestra cómo las distintas plataformas cumplen con el modelo propuesto.

4.1. TOP (Ten Objects Platform)

Nuevas versiones de esta plataforma aún continúan desarrollándose en el Departamento de Ciencia de la Computación de la Pontificia Universidad Católica de Chile. TOP funciona bajo una arquitectura cliente-servidor. Ésta implementa el componente "*canal*" como un servidor que es el 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 consta de un puerto de entrada/salida implementado a través de un socket. A través de esta puerta comunican a las aplicaciones cliente, utilizando al servidor como medio de comunicación.

4.2. MetaWeb

Este proyecto continúa desarrollándose en el Centro Nacional de Investigaciones Alemán (GMD FIT.CSCW). MetaWeb está diseñado para brindar soporte a aplicaciones sincrónicas que trabajan sobre la Web, bajo una arquitectura cliente-servidor. El componente "*canal*" está representado por dos servidores:

- El MetaWeb Server, encargado de brindar persistencia a los objetos compartidos, administración de las conexiones de usuarios, propagación de eventos, etc.
- El Web Server tradicional, que es el encargado de servir las tradicionales páginas web.

El componente "*manejador de eventos*" está implementado por dos componentes, una API y el MetaWeb Client:

- La API es la encargada de manipular el modelo de objetos propuesto por la plataforma (Usuarios, Ubicaciones y Sesiones), ocultándole al programador muchos de los detalles de implementación.
- El MetaWeb Client tiene dos funciones, la primera es brindar la representación de un usuario del MetaWeb Server, y la segunda es actuar como proxy de los objetos almacenados en el servidor.

4.3. NSTP (Notification Servers for Synchronous Groupware)

Este proyecto ha sido desarrollado en conjunto entre Lotus Development Corporation y el Laboratorio de Ciencia de la Computación del M.I.T. (Massachusetts Institute of Technology). Posee una arquitectura cliente-servidor, 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. NTSP trata con cuatro tipos de objetos: *places, things, facades* y el *server* propiamente tal. Cuando un objeto *thing,* perteneciente a un *place,* es modificado por un cliente, el *server* notifica a todos los clientes pertenecientes a ese *place*. También el server provee los servicios de autenticación de usuarios, creación de *places* y recuperación de *places* existentes.

4.4. Habanero

Este proyecto continúa desarrollándose en el NCSA (National Center for Supercomputing Applications) de la Universidad de Illinois, Urbana-Champaign (UIUC). Habanero junto con GroupKit son los dos máximos referentes en el área de las plataformas de groupware. Éste 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, con características similares al patrón Broker **[Bus96]**. 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. El servidor por su parte es el encargado de la sincronización de eventos, administración de sesiones y conexiones de usuarios, control de piso, persistencia de los objetos, y distribución de mensajes.

4.5. GroupKit

Es la plataforma más antigua, desarrollada en el Departamento de Ciencia de la Computación de la Universidad de Calgary, Alberta, Canadá. GroupKit es una extensión del lenguaje Tcl/Tk, que facilita el desarrollo de aplicaciones que apoyan el trabajo distribuido, en tiempo real, entre dos o más personas. 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*".

Estos tres tipos de procesos se comunican a través de sockets. Cada manejador de sesiones mantiene una conexión, a través de un socket, con el "*Registrador*" (cliente-servidor). De este modo, los manejadores de sesiones se comunican entre ellos a través del servidor "*Registrador* ". Las conferencias mantienen un socket hacia el "*Registrador*", un socket al manejador de sesiones que las creó, y un socket hacia cada conferencia en la sesión. Así, las conferencias (aplicaciones de los usuarios) se comunican punto a punto (par a par), y también tienen conexión con el servidor central "*Registrador*" y con su manejador de sesiones. Los

"*mensajes*" enviados a través del canal son comandos (RPC). El "*manejador de eventos*" de nuestro esquema estaría representado por los manejadores de sesiones.

5. Conclusiones y Trabajo Futuro

Las aplicaciones colaborativas pueden ser vistas como un conjunto especial de aplicaciones distribuidas. El modelo de comunicación presentado brinda un marco de referencia para el desarrollo de futuras arquitecturas de apoyo, tanto al trabajo colaborativo como a gran parte del distribuido. Esta formalización presenta el primer paso en la estructuración del problema, posibilitando la especificación de patrones arquitectónicos **[Bus96]** para el área de CSCW. Además, el modelo permite que las soluciones que respeten el esquema de comunicación propuesto puedan ser reutilizadas por otras plataformas o aplicaciones específicas.

A futuro es necesario bajar el nivel de abstracción del problema y formalizar una propuesta distribuida (par a par) y otra centralizada (cliente-servidor), como estrategias para la implementación del canal. Estas formalizaciones deberán ser definidas como patrones arquitectónicos, para que sirvan como guía para la implementación de plataformas de apoyo al desarrollo de aplicaciones colaborativas.

Reconocimientos

Este trabajo ha sido parcialmente financiado por el Fondo Nacional de Ciencia y Tecnología de Chile (FONDECYT), Proyecto N° 198-0960.

Referencias

[Bus96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[Cha98] 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.

[Ell91] Ellis, C., Gibbs, S., Rein, G. *Groupware. Some Issues and Experiences*. Communications of the ACM, Vol. 34, Num. 1, pp. 38-58. 1991.

[Gal97] 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.

[Gue98] Guerrero, L., Fuller, D. *Objects for Fast Prototyping of Collaborative Applications*. Proceedings of CRIWG'98, Rio de Janeiro, Brasil, Sep. 1998.

[Joh88] Johansen, R. *Groupware: Computer Support for Business Teams*. The Free Press, N. Y., 1988.

[Och99] Ochoa, S., Fuller, D. *A Client-Server Coordination Model to Support Portable Collaborative Applications Construction*. Submitted for Review to Fifth International Workshop on Groupware (CRIWG '99).

[Pat96] Patterson, J., Day, M., Kucan, J. *Notification Servers for Synchronous Groupware*. Proceedings of CSCW '96. Boston, Massachusetts, EEUU. Nov. 1996.

[Ros97] Roseman, M. and Greenberg, S. *Building Groupware with GroupKit*. In M. Harrison (Ed.) Tcl/Tk Tools, pp. 535-564, O'Reilly Press, 1997.

[Tan96] Tanenbaum, A. *Distributed Operating Systems*. 1° Ed., Prentice Hall. 1996.

[Tre97] 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.