Arquitectura Centrada en la Web para el Control y Monitoreo de Funcionalidad Domótica

E. **Echeverría;** T. **Ballari;** H. **Molina;** E. **Wainerman;** L. **Olsina**
GIDIS - Dpto. de Computación, Facultad de Ingeniería, UNLPam
Calle 9 y 110 – (6360) Gral. Pico, La Pampa
TE +54 (0)2302 430497 Interno 6501
E-mail olsinal@ing.unlpam.edu.ar

Resumen: en el presente paper se ilustra una arquitectura para el desarrollo de sistemas en ambientes distribuidos homogéneos centrados en la Web, que a efectos de ser llevado a la práctica, se discute el diseño e implementación de una aplicación para el control y monitoreo con funcionalidad domótica. La arquitectura empleada se basa en el patrón arquitectural denominado *Web Delivery*. Se presentan consideraciones finales y futuros avances en el empleo de arquitecturas de software con potencial repercusión para el campo de la Domótica.
Palabras claves: Domótica, Arquitectura, Aplicación Web, Patrones, RMI.

1. Introducción

El surgimiento de nuevas tecnologías y arquitecturas hicieron que las aplicaciones centradas en Internet evidenciaran un rápido crecimiento. Particularmente, las aplicaciones Web ganaron en interacción y funcionalidad para pasar de ser sólo una forma de presentar información y contenido, a ser aplicaciones con soporte a complejidad de software tradicional. Este crecimiento también está incluyendo a las funcionalidades domóticas [4, 8] que es una de nuestras áreas de estudio reciente [7]. (Los autores en [8] definen a Domótica –palabra surgida de doméstico e informática- como la disciplina que estudia el desarrollo de infraestructuras inteligentes en casas y edificios, como así también las tecnologías de información para soportarlas). De esta manera, enfocamos el trabajo hacia la selección y discusión de una arquitectura de desarrollo de sistemas en ambientes homogéneos basados en la Web, ilustrándolo con una aplicación de control y monitoreo domótico. Los módulos de funcionalidad que se implementaron fueron los de control de iluminación, control de riego y cochera. A través de la misma, desde cualquier navegador, una persona podría configurar el subsistema de luces, acceder ordenadamente a la cochera, accionar los sistemas de riego, entre otros aspectos.

Para lograr dicho desarrollo, en primer lugar, se optó por un modelo cliente-servidor basado en un enfoque de arquitectura de tres capas. Básicamente, este modelo propone tener al cliente como una de sus capas, el cual se comunica con una capa intermedia que es la del servidor, siendo éste último el encargado de la manipulación de los datos con la tercera capa. Esta distribución permite que varios clientes interactuen con un servidor (o más de uno), el cual accederá a una BD (o varias) a efectos de responder a los requerimientos de los clientes. Es también la capa del servidor quien se encarga de la lógica de negocio y comunicación con el dispositivo físico. En la literatura reciente, se ha dado en llamar *Web Delivery* [1] al patrón arquitectónico que se utiliza para modelar el tipo de aplicaciones como el discutido en el presente trabajo. "*The greatest strength of this architecture is its ability to leverage existing business objects in the context of a Web application*", p. 110.

Luego se optó por un ambiente que permitiera el desarrollo de aplicaciones en intranets e Internet basado en el paradigma de Orientación a Objetos (se seleccionó la plataforma Java 2 [6]).

Por una parte, el entorno Java brinda conectividad con diferentes BD relacionales a través del mecanismo JDBC [11], el cual permite trabajar transparentemente con distintos motores de BD. Por otra parte, el ambiente brinda soporte para desarrollar aplicaciones servidoras, aplicaciones clientes (applets), y la respectiva intercomunicación entre procesos. La comunicación entre ambas capas se logra a través del mecanismo llamado RMI [9], el cual habilita a los clientes a invocar métodos de objetos que residen en diferentes máquinas remotas. Este modelo no soporta la distribución de objetos independientemente del lenguaje, por lo que, tanto el cliente como el servidor deberán estar implementados en Java; de allí el concepto de ambiente distribuido homogéneo. Además, otro aspecto a destacar es la posibilidad de incorporar código no-Java a través de métodos nativos lo que posibilita comunicar al servidor con los módulos funcionales del dispositivo físico. Finalmente, cabe indicar la necesidad de que el cliente sea notificado de los eventos ocurridos en el servidor que afecten la vista del cliente; para tal fin se utilizó el mecanismo callback.

En base a los mecanismos antes descriptos se centró el desarrollo de la aplicación domótica. No obstante, están surgiendo nuevas tecnologías que brindan mayores prestaciones, principalmente en cuestiones de performance. La estrategia RMI que se presenta tiene ciertas limitaciones para desarrollos de aplicaciones críticas en cuanto a restricciones de tiempo, entre otros aspectos. Este punto se profundizará en las consideraciones finales, donde se brindará un mayor detalle y explicación de estas desventajas y posibles soluciones.

El presente trabajo se estructura de la siguiente manera: en la sección 2, se describen los componentes de la arquitectura distribuida homogénea en función del patrón arquitectónico *Web Delivery*. A continuación, en la sección 3, se discuten distintos aspectos de diseño e implementación de la aplicación domótica. Finalmente, en la sección 4, se realizan algunas consideraciones y delinean futuros avances.

2. Arquitectura Distribuida Homogénea: Modelo de tres Capas

Como se mencionó anteriormente, la arquitectura utilizada es un modelo de tres capas, a saber: capa de Presentación, de Aplicación y de Datos. En la figura 1, se muestra un esquema general de esta arquitectura (además, el lector puede remitirse a [1] p.112, para apreciar una vista lógica - especificada en UML, del patrón arquitectónico *Web Delivery*. "*The Web Delivery architectural pattern is so named because the Web is used primarily as a delivery mechanism for an otherwise traditional client/server system*").

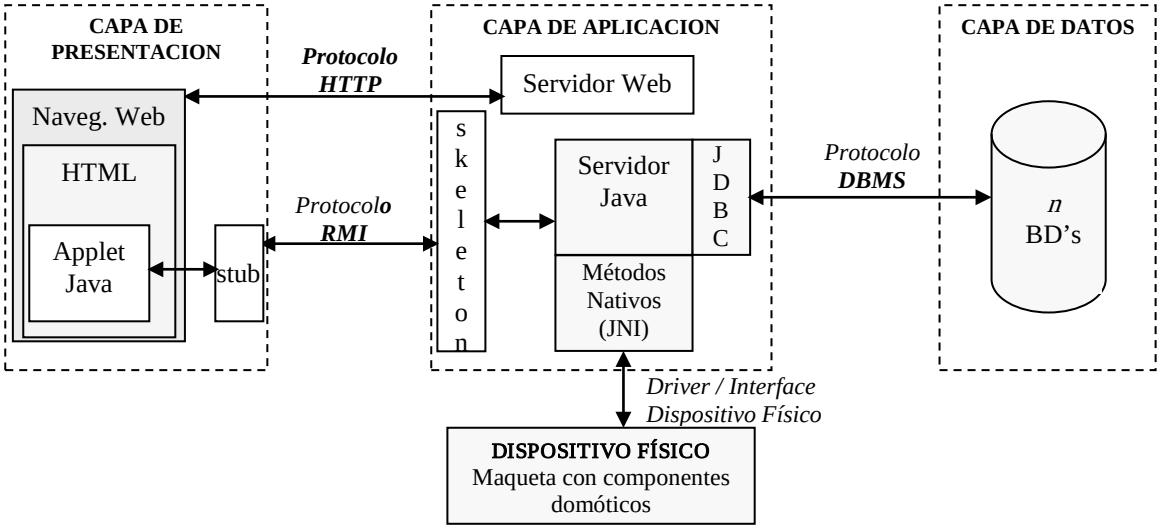

Fig. 1: Modelo de tres capas para una arquitectura de software distribuida homogénea.

En la capa de Presentación se distribuye la lógica del cliente respectiva y parte de la lógica de la aplicación (a través de los applets). Los applets pueden ser ejecutados potencialmente en cualquier navegador y pueden invocar tanto métodos locales como remotos. Para ejecutar métodos de objetos remotos, Java posee el mecanismo RMI que permite la comunicación con el servidor de una manera transparente para el usuario. Este mecanismo será descripto en párrafos siguientes.

En la capa de Aplicación, se encuentra básicamente el servidor Web y el servidor de aplicaciones (o varios). Esta capa implementa gran parte de la lógica de negocio específica del dominio; además se encarga de la manipulación de los datos (con la capa de Datos) y de la comunicación y manejo de eventos con el dispositivo físico –ver fig. 1.

Para la manipulación de los Datos, Java utiliza una interface de acceso a bases de datos llamada JDBC la que será presentada en la siguiente sección. En la capa de Datos se encuentran toda la información a la que tendrá acceso la aplicación servidora. La comunicación entre ambas capas se realiza a través del protocolo DBMS. Por otra parte, para la comunicación con el dispositivo físico (maqueta con componentes domóticos –ver fig. 3a), se utiliza la facilidad de métodos nativos (protocolo JNI), los que permiten ejecutar código no-Java desde clases Java.

La elección de la arquitectura distribuida en un ambiente homogéneo para el desarrollo de la aplicación domótica, nos pareció la alternativa más conveniente, debido a que era un proyecto de software nuevo. No obstante, previo a dicha elección, se analizaron otras arquitecturas y estrategias posibles, como por ejemplo el ambiente heterogéneo basado en CORBA (Common Object Request Broker Architecture). Esta estrategia puede ser una buena elección para el diseño e implementación cuando hay que integrar sistemas existentes o componentes desarrollados en diferentes ambientes o lenguajes [2]. Otra ventaja de CORBA con respecto a RMI es que, como se indicó en la introducción, está brindando soporte para aplicaciones de tiempo real.

La característica principal de los ambientes homogéneos es que, tanto el cliente como el servidor deben estar implementados en el mismo lenguaje para permitir su intercomunicación. El mecanismo RMI permite distribuir las tareas o procesos sobre otros objetos en la red. Es decir, no sólo se puede realizar paralelismo local (threads) sino también paralelismo global (en otras máquinas, potencialmente con otros SO, etc.). Esta invocación a un objeto remoto es transparente al usuario, ya que éste no conoce en qué máquina reside físicamente dicho objeto, sino que los objetos remotos son vistos por el cliente como objetos locales. La transparencia es un atributo esencial de los sistemas distribuidos.

3. Detalles de Diseño e Implementación de la Aplicación Domótica

En esta parte del trabajo, se detallan los aspectos más importantes que tienen que ver con el diseño e implementación de la aplicación para control y monitoreo de funcionalidad domótica. En primer lugar, en la figura 2 se muestra un Diagrama de Clases simplificado (especificado en UML [10]), que grafica las principales clases e interfaces del sistema. En dicho diagrama, se especifican clases de soporte a métodos nativos, callback, y aspectos específicos del cliente y del servidor, como se detallará en las siguientes subsecciones.

3.1. Aspectos del Servidor

Ante todo, es necesario tener una interface que publique aquellos métodos del servidor de aplicaciones que podrán ser invocados por los clientes, a los efectos de separar la definición de los métodos de los objetos, de su implementación. Dicha interface debe extender de *java.rmi.Remote*; y todos sus métodos deben contener la cláusula *throws java.rmi.RemoteException* ya que durante su invocación remota y posterior ejecución pueden ocurrir errores en la comunicación con el servidor. Esta característica se convierte en la única diferencia entre invocar un método remoto de uno local.

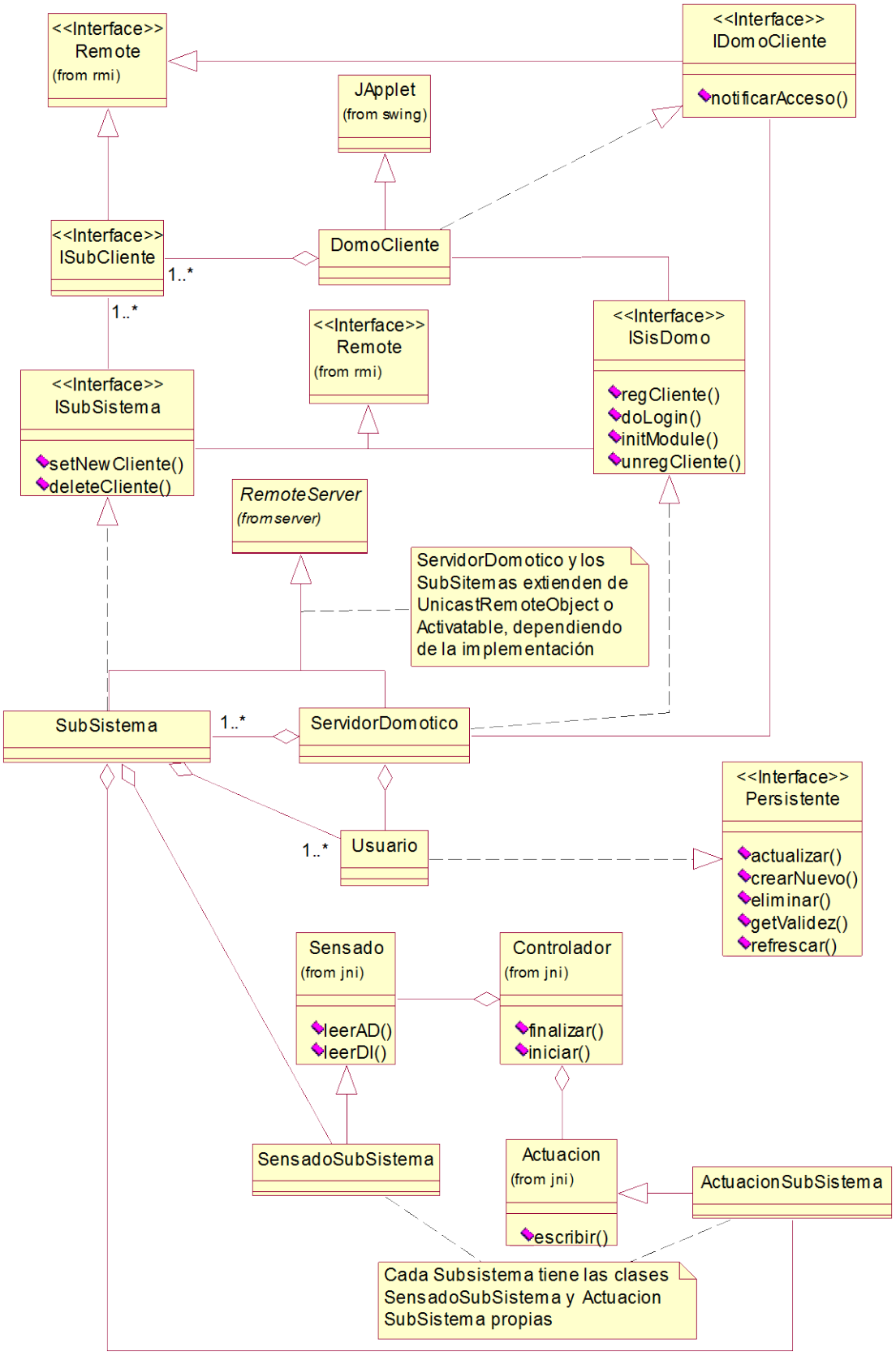

Fig. 2: Diagrama de clases simplificado del Sistema Domótico.

```
public interface ISisDomo extends java.rmi.Remote {
      String regCliente () throws java.rmi.RemoteException;
  // definiciones de otros métodos
}
```

La aplicación servidora, debe extender de una de las implementaciones de la clase *RemoteServer*, tal como *Activatable* o *UnicastRemoteObject*. El primer tipo de servidor, se crea y se ejecuta bajo demanda, mientras que el segundo, se mantiene corriendo todo el tiempo y puede consumir recursos innecesariamente.

La clase *ServidorDomotico* es quien debe implementar la interface correspondiente:

```
public class ServidorDomotico extends Activatable implements ISisDomo {
// otras clases agregadas
public String regCliente (IDomoCliente) throws RemoteException {
   // implementación del método
   }
// otros métodos
}
```

En el método *main* del servidor se puede especificar, entre otras cosas, un sistema de seguridad apropiado como puede ser, por ejemplo, *RMISecurityManager*.

```
try {
    System.setSecurityManager(new RMISecurityManager());
} catch (java.lang.SecurityException exc) {
System.out.println("Violacion de seguridad" + exc.toString());
    }
```

Además, en este método se debe crear una instancia local del objeto *ServidorDomotico* y publicarlo en el registro. Para arrancar este registro se debe ejecutar la aplicación *rmiregistry*. Este sistema de publicación de los objetos remotos se lo conoce como RMI Naming System.

```
try {
   ServidorDomotico servidor=new ServidorDomotico();
   Naming.rebind("SERVIDOR-DOMOTICO",servidor);
} catch (Exception e) {
   System.out.println("Error en el Servidor"+e.getMessage());
   }
```

Para poder efectuar la comunicación con el dispositivo físico se utiliza, como se indicó previamente, una tarjeta adquisidora de datos (PC-Lab 812G). Los drivers o controladores provistos en forma de librerías de enlace dinámico, ofrecen funciones que pueden ser utilizadas por rutinas, por ejemplo, en lenguaje C++ para enviar o recibir datos de la placa y así controlar y monitorear los distintos aspectos de los componentes del edificio. Esas rutinas pueden ser invocadas desde Java utilizando el protocolo JNI. Para ello, se deben crear clases cuyos métodos son implementados en lenguaje nativo, el cual se encuentra en una librería de enlace dinámico que contiene la implementación de esos métodos. Esta librería debe ser cargada durante la creación de la clase que declara los métodos nativos en una sección *static*.

```
public class Controlador {
   public static final native void iniciar();    // la placa adquisidora de datos
   public static final native void finalizar();
   static { System.loadLibrary("domotica");  }   // librería dinámica Domotica.dll
   }
```

La clase que define los métodos nativos debe ser utilizada para crear los archivos de encabezados (*file.h* de C++), con *javah*, que especifican las signaturas entre los métodos nativos definidos en Java y las correspondientes implementaciones de esos métodos. Esta función realiza una correspondencia de nombres de métodos que hace transparente su uso (pueden utilizarse como cualquier otro método Java).

```
Definición en archivo encabezado
   JNIEXPORT void JNICALL Java_Controlador_iniciar
(JNIEnv *, jclass);
```

Los archivos ".h" deben ser luego incluidos (*#include*) en el archivo con las implementaciones en código nativo, al igual que *jni.h* que contiene las definiciones de la comunicación entre clases Java y código nativo y en el caso de Windows, el archivo *windows.h*, proporcionado junto al compilador C++.

3.2. Aspectos del Cliente

Los clientes RMI son los que realizarán invocaciones remotas a un servidor. Para ello es necesario obtener el objeto remoto (servidor) desde el registro utilizando el nombre con el que fue registrado y la direccion IP de la estación que la alberga.

```
try {
obj=(ISisDomo)Naming.lookup("//"+getCodeBase().getHost()+"/SERVIDOR-DOMOTICO");
} catch (Exception e) {
   System.out.println("Excepcion en el Applet"+e.getMessage());
    }
```

Una vez que se obtuvo el objeto remoto se puede invocar los distintos métodos que hayan sido publicados por el servidor a través de la interface.

```
try {
String s= obj.regCliente(this);   //Este trozo de código es utilizado en el Cliente
} catch (java.rmi.RemoteException exc) {  // que implementa la interface IDomoCliente
   notificar(12);
}
```

En la sección 3.5 discutiremos el mecanismo callback y su impacto sobre las aplicaciones clientes.

3.3. Aspectos de la Comunicación

Es importante mencionar la forma en que RMI hace posible la comunicación entre los objetos remotos. Esto se realiza a través de un mecanismo estándar (también utilizado por RPC – Remote Procedure Call) basado en el uso de *stubs y skeletons*. Es necesario generar los stubs y los skeletons a los efectos de generar las implementaciones de los métodos de la clase RMI referentes a la comunicación, duplicación de objetos, seguridad, etc. y de esa manera ocultarlos al programador. Un stub actúa del lado del cliente, o sea, el cliente invoca un método del stub local y este debe ser responsable de llevar esa invocación hasta el skeleton del objeto remoto (ver Fig. 1). Cuando se invoca un método del stub, este último realiza las siguientes acciones:

- Inicia una comunicación con la máquina virtual remota que contiene al objeto remoto.

- Escribe y transmite los parámetros a la máquina virtual remota.
- Espera por los resultados de la invocación al método.
- Lee los valores o la excepción retornada.
- Devuelve el resultado al cliente.

En la máquina virtual remota (servidor), cada objeto remoto debe tener su correspondiente skeleton, el cual es responsable de enviar la llamada a la implementación del objeto remoto actual. Cuando un skeleton recibe una invocación a un método realiza lo siguiente:

- Lee los parámetros para el método remoto.
- Invoca el método sobre la implementación del objeto remoto actual.
- Escribe y transmite los resultados (valor o excepción) al invocador.

Para generar los stubs y los skeletons es necesario ejecutar el compilador *rmic <nombre servidor>*.

3.4. Aspectos de los Datos

Para lograr la comunicación entre el servidor y la capa de datos, Java posee un mecanismo llamado JDBC, el cual es una interface de acceso a BD SQL estándar, que brinda acceso uniforme a un amplio rango de BD relacionales desde una aplicación implementada en el ambiente Java.

Para utilizar JDBC con un sistema administrador de BD particular, es necesario contar con un driver o controlador que medie entre JDBC y la BD, como por ejemplo JDBC-ODBC Bridge, el cual permite que la mayoría de los drivers de ODBC estén disponibles para los programadores de JDBC. La obtención del driver a utilizar para la conexión se realiza de la siguiente manera:

```
try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (java.lang.Exception e) {
 System.out.println(e);
  }
```

Para establecer una conexión con una BD, se debe invocar al método *DriverManager.getConnection*, el cual toma como parámetro tres objetos de tipo *String*: el primero representa un JDBC URL (Uniform Resource Locator), el segundo el ID del usuario y el tercero la contraseña de dicho usuario. La clase *DriverManager* intenta localizar un driver que pueda conectarse a la BD representada por ese URL, para lo cual, esta clase mantiene una lista de drivers registrados. Al momento en que se invoca el método *getConnection*, esta clase chequea la lista de drivers hasta encontrar uno que pueda conectarse a la BD especificada por el URL.

```
String url="jdbc:odbc:basedatos";
try {
  Connection con=DriverManager.getConnection(url,"","");
  ...
  con.close();
} catch (SQLException e) {
   }
```

Una vez que se obtuvo la conexión, se pueden enviar sentencias SQL a través de la misma, utilizando el objeto *Statement*, el cual es una interface que provee los métodos básicos para ejecutar dichas sentencias y recuperar los resultados. Para crear el objeto *Statement*, se debe invocar al método *createStatement* de la conexión, como se muestra a continuación:

```
Connection con=DriverManager.getConnection(url,"","");
Statement stmt=con.createStatement();
...
stmt.close();
con.close();
```

Para ejecutar sentencias SQL, deben invocarse los métodos *executeQuery*, *executeUpdate* o *execute*, según corresponda. La forma más universal de ejecutarlas, es invocando el método *execute*, el que es capaz de ejecutar cualquier tipo de sentencia. Sin embargo, los métodos *executeQuery* y *executeUpdate* son más eficaces y más utilizados.

```
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery(consulta);
...
rs.close();
int actualizadas = stmt.executeUpdate(actualizacion);
 //-devuelve la cantidad de filas afectadas
...
rs2.close();
stmt.close();
```

3.5. Aspectos del Mecanismo Callback

Un aspecto importante de determinadas aplicaciones cliente/servidor, y en particular de esta aplicación domótica, es la necesidad de que el cliente (o clientes) sea notificado de los eventos ocurridos en el servidor que afecten la vista del cliente.

En nuestro caso, los cambios ocurridos en algún módulo del sistema domótico, capturados mediante la aplicación servidora, deben reflejarse en la vista que todos los clientes tienen del fenómeno controlado o monitoreado. Para lograr esto hay dos formas de actualizar a los clientes: una es que el cliente pregunte al servidor en forma periódica sobre el estado del modelo con el consecuente costo (e ineficiencia); la otra es que el servidor notifique a los clientes cuando ocurran esos cambios. La diferencia es más que una cuestión de eficiencia, ya que en el segundo caso el servidor pasa de ser un agente pasivo a un activo participante en la comunicación de eventos. Para la implementación de nuestro sistema hemos seleccionado la segunda opción.

Tal como fue especificado en la sección 3.1, para que un objeto pueda ser accedido en forma remota debe implementar una interface que extienda de *Remote*. También se debe utilizar la utilidad *rmic* sobre el cliente para generar los *stubs* y *skeletons* del mismo.

Además, este objeto cliente debe ser publicado o exportado. Los servidores se registran en el RMI Registry con un nombre identificatorio; mientras que los clientes son exportados en forma anónima mediante el método:

```
Activatable.exportObject(Remote obj).
```

Luego el cliente debe pasar una referencia de sí mismo al servidor para que este pueda utilizar los métodos definidos en la interface remota del primero. El servidor recibirá esa instancia no como la clase del cliente sino como la interface remota que implementa (ver figura 2).

3.6. Aspectos del Prototipo, Interface y Funcionalidad Domótica

En la figura 3.a se muestra una foto del prototipo domótico construido en 1998 en el laboratorio de Computación II, por estudiantes y personal docente de la Facultad de Ingeniería de la UNLPam.

Este prototipo se viene empleando en proyectos de software en dos cátedras de dos carreras distintas en dicha Facultad.

La aplicación con funcionalidad domótica desarrollada, implementa de un modo integrado los módulos de riego, iluminación y cochera. El prototipo consta, por ejemplo, para estos dos últimos módulos, de 4 sectores de iluminación, es decir, subsuelo (cochera), planta baja, primer y segundo piso; y un sistema de acceso a cochera con un portón deslizable, 3 sensores de paso, 2 semáforos y un dispositivo zumbador. El acceso al sistema domótico requiere ingresar un nombre de usuario y clave de acceso que será validado contra una BD que mantiene información de todos los usuarios del sistema y sus permisos respectivos.

El módulo de iluminación permite controlar y manejar el encendido y apagado de las luces de tres modos (independientemente para cada sector), a saber:

- *Modo manual:* el usuario puede encender y apagar las luces cuando lo desee.
- *Modo automático:* permite programar el encendido de las luces de dos maneras: ya definiendo un nivel de iluminancia (luminosidad) que determina un porcentaje de luz por debajo del cual se encenderán las luces del piso especificado (*modo iluminancia*), ya definiendo un rango de tiempo dentro del cual se encenderán las luces (*modo temporizador*).

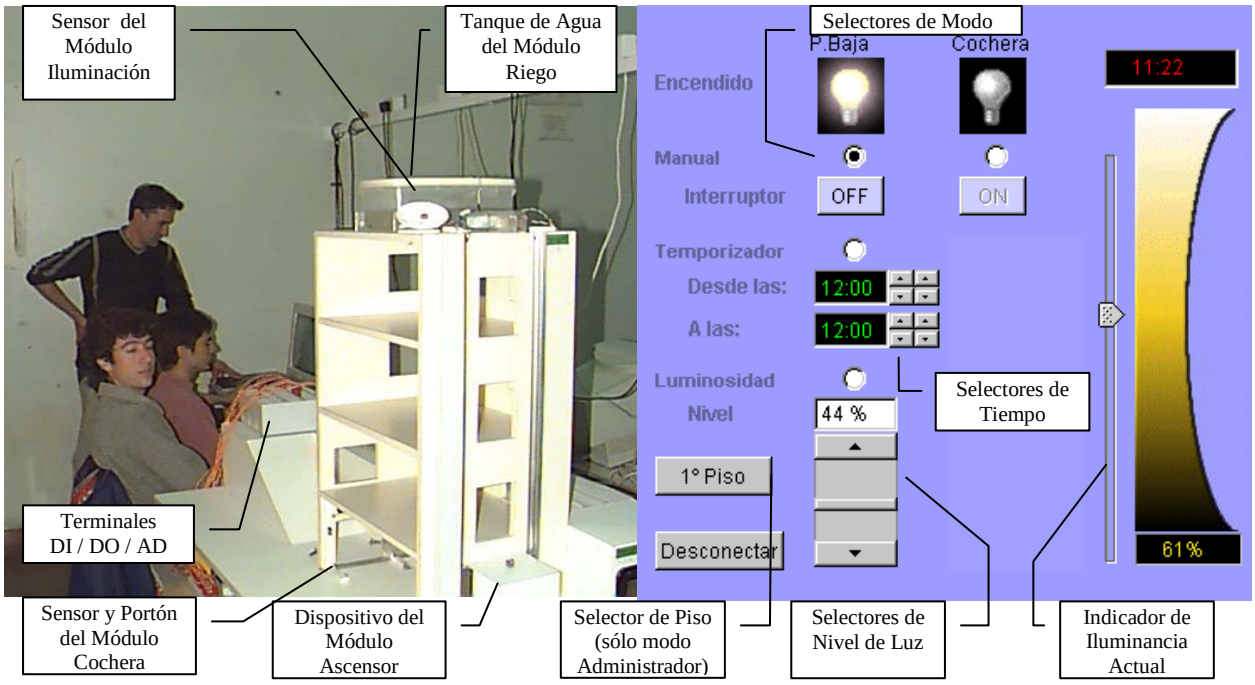

Fig. 3.a.: Foto del Prototipo Domótico; **3.b**: Vista parcial de la Interface de Usuario en el subsistema de Iluminación

Una vista de la interface de usuario del applet para este módulo se muestra en la figura 3.b. Además de los controles para cada modo se ofrecen controles que informan del estado de las luces de cada sector y el nivel de iluminancia del ambiente. La aplicación soporta múltiples usuarios entre los que se prevé un Administrador, quien tiene control sobre la configuración de todos los sectores. Por otro lado, los demás usuarios tendrán acceso sólo a su sector y al sector cochera (éste último, sólo en modo manual). Todos ellos serán notificados automáticamente de los cambios producidos por el fenómeno o por la acción de otros usuarios que estén controlando su correspondiente sector (por medio del mecanismo callback, discutido en la subsección anterior).

Finalmente, la lógica del módulo cochera está basada en distintos estados que determinan el comportamiento que tendrá el subsistema ante los eventos que ocurran en el modelo físico. Estos prevén los distintos sucesos que ocurren cuando un automóvil intenta entrar o salir de la cochera considerando los siguientes aspectos (el sistema construido sólo permite el ingreso o salida de un automóvil por vez):

- El sistema cuenta con 3 sensores de paso que determinan si un automóvil se encuentra frente a la puerta del lado exterior (desea entrar), interior (desea salir) o atravesando la puerta.
- Al bloquearse un sensor de entrada o salida, el sistema pone en rojo el semáforo del lado opuesto al mismo, de modo que los usuarios sepan que no pueden pasar al encontrarse otro vehículo del lado opuesto.
- El sistema sólo abrirá la puerta de entrada a un usuario que así lo solicite y esté bloqueando el sensor correspondiente (sensor exterior para entrada e interior para la salida). Por otra parte, si lo que se está produciendo es una salida, se accionará el zumbador para advertir a los peatones que un automóvil cruzará la acera.
- Luego, el sistema le da un lapso de tiempo para atravesar la puerta. Si no lo hace en este período, la puerta se cerrará.
- También se prevé que un automóvil no esté más de un tiempo determinado atravesando la puerta, lo que se considerará como una obstrucción indebida de la puerta lo cual se registrará en el sistema.
- Luego de que el automóvil deja de obstruir la puerta el sistema automáticamente la cierra.

Adicionalmente, el sistema registrará cada entrada y cada salida que realicen los usuarios en la BD, como así también los intentos fallidos y obstrucciones indebidas.

4. Discusión

Por una parte, una característica a destacar es que el reciente crecimiento de los PDAs (Personal Digital Assistant) y dispositivos semejantes, la telefonía móvil, los que están potencialmente permitiendo acceder a funcionalidades domóticas a través de aplicaciones centradas en la Web. Día a día, es creciente la utilización de este tipo de dispositivos electrónicos principalmente en los países altamente tecnificados.

Por otra parte, los mecanismos RMI y callbacks presentados proveen una plataforma base para desarrollar este tipo de aplicaciones distribuidas soportadas totalmente por el paradigma de Orientación a Objetos. Además, el entorno seleccionado permite la ejecución de código no-Java a través de la interface JNI, facilitando de un modo sencillo la comunicación de la aplicación con el dispositivo físico.

Sin embargo, como se mencionó en la sección 2, los mecanismos de implementación presentados en este trabajo tienen ciertas limitaciones para aplicaciones con restricciones críticas en cuanto al tiempo. En esta dirección, están surgiendo nuevas tecnologías de Java embebido como por ejemplo “real-time Java” que está brindando soporte a este tipo de aplicaciones [5]. De todas formas, al presente otras tecnologías como “real-time CORBA” y ActiveX/DCOM (Distributed Component Object Model) están en continuo progreso.

En referencia al mecanismo RMI básico (como el discutido) es oportuno agregar respecto de las desventajas o limitaciones, que nuevas estrategias surgidas como Servlets intentan potenciarlo. Por ejemplo, algunos navegadores no soportan RMI cuando se configura firewalls o proxy servers. Además, los paquetes del servidor que se crean para manipular las conexiones con los clientes, pueden vivir indefinidamente consumiendo recursos innecesariamente (esto puede parcialmente

evitarse mediante el empleo de la interface Activatable, como fue en nuestro caso). Por otro lado, si se sobreescriben las clases sockets para intercambiar datos en formas especiales (encriptada, comprimida, etc.), se pierde la capacidad que posee RMI de hacer HTTP tunneling.

Nuestros futuros avances están orientados a lograr que el monitoreo y, principalmente, el control de la aplicación domótica sea realizado en tiempo real, al menos en una intranet. Además, es nuestro interés incorporar las tecnologías recientes que se mencionaron en el párrafo anterior (como servlets) y demás tecnologías que puedan ir surgiendo.

Agradecimientos

Esta investigación es soportada por los proyectos UNLPam-09/F013, y UNLPam-Domótica. Agradecemos la participación de Giles, A; Miguel, F., Mauricio, B., y Lapine, P., tanto en el diseño como en la construcción de la maqueta del edificio y su circuitería (98-99).

Referencias

1. Conallen, J., (1999), *Building Web Applications with UML,* Addison-Wesley.
2. Evans, E., Rogers, D., (1997), *Using Java Applets and CORBA for multi-user Distributed Applications,* IEEE Internet Computing Vol 1, Nº 3, pp. 43-55.
3. Itschner, R., Pommerell, C., Rutishauser, M., (1998), *GLASS: Remote Monitoring of Embedded Systems in Power Engineering,* IEEE Internet Computing Vol 2, Nº 3, pp. 46-52.
4. Jong Jin Kim, Jin Kim Jong (1998), *Intelligent Buildings,* Butterworth-Heinemann Publisher.
5. Lee, B, H., (1998), *Embedded Internet Systems,* IEEE Internet Computing Vol 2, Nº 3, pp. 24-29.
6. Li Gong, (1999), *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley
7. Olsina, L., Echeverría, E., Miguel, F., Giles, A. (2000), *Domotic Monitoring by using the Web*, To appear in proceed. of AADECA 2000 (Asociación Argentina de Control Automático), Bs. As.
8. Quinteiro González, J., Lamas Graziani, J.; Sandoval, J., (1999), *Sistemas de Control para Viviendas y Edificios: Domótica*, Ed. Paraninfo, Mad. Spain.
9. Sun Microsystems Inc., (1999), *The Java Remote Method Invocation Home Page*, http:// www.java.sun.com/products/jdk/rmi/index.html
10. UML – Unified Modeling Language-, (1997), UML Notation Guide, Version 1.0, and ulterior releases.
11. White, S.; Fisher, M.; Cattell, R.; Hamilton, G.; Hapner, M., (1999), *JDBC API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*, Addison-Wesley