Prácticas de laboratorio

Un chat distribuido orientado a objetos basado en RMI (3 sesiones)

Concurrencia y Sistemas Distribuidos

Introducción

El objetivo de esta práctica es introducir a los estudiantes a los sistemas distribuidos y al diseño orientado a objetos de aplicaciones distribuidas. Se utilizará RMI como middleware distribuido subyacente, pero cualquier otro middleware que soporte objetos distribuidos podría ser utilizado. La aplicación seleccionada para aprender y practicar los sistemas distribuidos es una aplicación de Chat distribuida. Existen muchas aplicaciones de Chat en el mercado y también algunos estándares. El objetivo no es desarrollar una aplicación completa ni seguir un estándar en particular, sino centrarse en un sistema distribuido orientado a objetos claro y sencillo. Esta práctica también trata algunos aspectos importantes de los sistemas distribuidos no específicos de los sistemas orientados a objetos, como servicios de nombre, y el paradigma de cliente/servidor. Se utilizará Java como lenguaje de programación y RMI como soporte de ejecución de objetos.

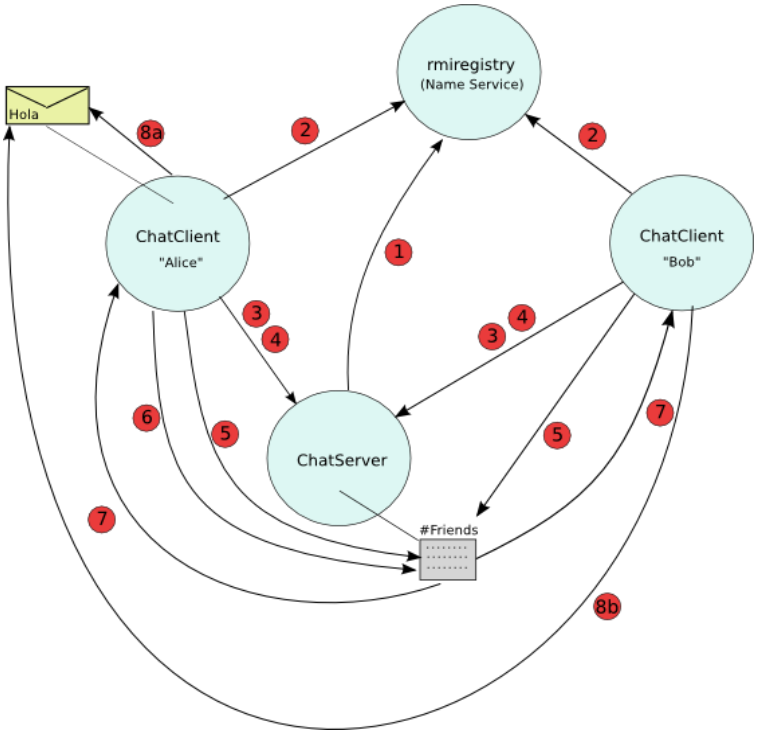
Una vez completada esta práctica, se habrá aprendido a:

- Identificar los diferentes procesos que conforman una aplicación distribuida.
- Compilar y ejecutar aplicaciones distribuidas sencillas.
- Entender la función de servicio de nombres.
- Entender el paradigma de cliente/servidor y su extensión orientada a objetos para diseñar e implementar aplicaciones distribuidas.

Se necesitan aproximadamente tres semanas para completar la práctica. Cada semana hay que asistir a una sesión de laboratorio, donde el profesor puede resolver cuestiones técnicas que surjan. También puede ser necesario dedicar tiempo adicional para completar la práctica. La práctica se puede realizar conectándose al escritorio remoto de los laboratorios, o bien utilizando directamente el ordenador personal. Es necesario destacar que, para la correcta visualización de la interfaz gráfica, la profundidad de color seleccionada en el cliente de escritorio remoto debe ser “Color verdadero (24) bits”.

Un Chat distribuido básico

El siguiente esquema muestra una disposición básica del chat distribuido. En este caso, hay un servidor de nombres (llamado *rmiregistry* en RMI), un ChatServer y dos ChatClient (lanzados por los usuarios Alice y Bob). Estos 4 procesos conforman un sistema sencillo que, sin embargo, es más complejo e interesante que un esquema de cliente/servidor puro.



El esquema muestra los pasos ordenados que nuestra aplicación sigue cuando los usuarios Alice y Bob se unen a un canal de charla denominado “#Friends” y empiezan a chatear. Alice envía un mensaje “Hola” y ambos usuarios lo reciben al estar conectados al canal.

A continuación se detallan los pasos seguidos. Cada paso está marcado con un círculo rojo numerado en el esquema:

1. ChatServer registra su objeto principal “ChatServer” en el servidor de nombres.
2. Los dos ChatClients buscan el objeto “ChatServer” utilizando el servidor de nombres. Nótese que los tres procesos tienen que estar de acuerdo en el nombre del objeto “ChatServer”.
3. Los clientes del Chat se conectan al ChatServer. Para ello crean un objeto *ChatUser* local y lo registran en el servidor, mediante el método *connectUser* de ChatServer.
4. Los clientes del Chat preguntan la lista de canales, mediante el método *listChannels* de ChatServer.
5. Los clientes del Chat se unen al canal “#Friends”. Para ello obtienen primero el objeto *ChatChannel* del canal solicitado (mediante el método *getChannel* de ChatServer) y se unen a dicho canal, usando el método *join* de ChatChannel.
   Nótese que el primer usuario en unirse al canal también recibe una notificación cuando el segundo cliente se une. Esta notificación de canal no se ha dibujado en el

esquema.

6. Alice envía un mensaje de chat sencillo al canal. Para ello, *ChatChannel* invoca a *ChatUser* para retransmitir los mensajes.
7. A partir de dicho envío, el canal retransmite el mensaje a todos los usuarios conectados (al canal).
8. Los usuarios, cuando reciben el mensaje, piden su contenido. 8a es una invocación local, mientras que 8b es una invocación remota.

Nótese que, aunque el rol de los *ChatClient* consiste principalmente en actuar como clientes de chat, también actúan como servidores, pues tienen objetos que son invocados de forma remota. Más concretamente, *ChatChannel* invoca a *ChatUser* para retransmitir los mensajes, y cualquiera que necesite ver el contenido de un mensaje dado tiene que invocar al dueño del mensaje por su contenido. En el esquema, *ChatClient* pregunta al usuario Alice por el contenido del mensaje "Hola". Este patrón de invocación de objetos no es habitual en la mayoría de entornos de chat pero es lo bastante completo para comprender las aplicaciones orientadas a objetos distribuidas.

Instalaciones de laboratorio y recomendaciones del entorno

Se puede utilizar cualquier entorno de desarrollo para gestionar proyectos Java (BlueJ, Eclipse, etc.) o simplemente un editor de texto sencillo, el compilador estándar por línea de comandos y la máquina virtual de Java. Nuestra recomendación es utilizar las herramientas de línea de comandos, pues aunque se utilice un IDE para el desarrollo, existen ciertas acciones que se deben llevar a cabo a través de la línea de comandos.

Si realiza la práctica conectándose al escritorio remoto de los laboratorios, hay que tener en cuenta que la máquina a la que se conecta tiene un firewall configurado, el cual bloquea la mayoría de puertos. Se pueden utilizar los puertos **9000-9499**. Algunos de los programas a utilizar en la práctica y la herramienta *rmiregistry* necesitarán que se indique el puerto a usar.

El software proporcionado

Inicialmente, se ha proporcionado el código binario del DistributedChat, para que el alumno pueda comprobar la funcionalidad del chat distribuido que se desea implementar.
Descargue el archivo "**DistributedChat.zip**" del material de la práctica y descomprímalo. Verá que contiene un conjunto de ficheros de tipo .class (es decir, ficheros con código binario Java). De estos ficheros nos interesan los programas **ChatClient.class** y **ChatServer.class**. Los otros archivos son interfaces y clases necesarias para estos programas y todos ellos conforman un chat distribuido básico.

Para poder ejecutar *ChatServer* y *ChatClient*, se necesita una instancia de ***rmiregistry*** ya en funcionamiento de modo que nuestros programas de chat puedan utilizarlo como servidor de nombres. Los servicios de nombre, como se ha explicado en clase, permiten registrar un nombre simbólico para el objeto remoto y asociarle su referencia, y así que puedan ser localizados por los clientes. Para empezar *rmiregistry*, sólo hay que ejecutarlo en una terminal. Importante: debe ejecutarlo **en el mismo directorio donde se encuentran las clases .class de DistributedChat.**

```
rmiregistry 9000
```

El parámetro 9000 implica arrancar *rmiregistry* en el puerto 9000. Si este parámetro no es proporcionado, *rmiregistry* se arrancará en su puerto por defecto 1099. Debe recordarse que en los laboratorios del DSIC tiene que utilizar puertos dentro del rango 9000-9499.

NOTA: Si al ejecutar esta orden aparece un error del tipo "Port already in use", es posible que se deba a que otro alumno haya lanzado una instancia sobre la misma máquina virtual. En dicho caso, vuelva a lanzar el *rmiregistry* con otro número de puerto, por ejemplo el 9100.

Para empezar un *ChatServer*, debe ejecutarse el siguiente comando **en el mismo directorio donde se encuentran las clases .class** (los parámetros nsXX hacen referencia a *name-server*):

```
java ChatServer nsport=9000 myport=9001
```

Esta instrucción empieza un *ChatServer* en el puerto 9001, y se utiliza el puerto local 9000 cuando *ChatServer* necesita conectarse al *rmiregistry*. El puerto local por defecto para *ChatServer* y *ChatClient* es 9001, y el puerto por defecto para *rmiregistry* es 9000, por lo que la instrucción anterior tiene el mismo efecto que esta instrucción más sencilla:

```
java ChatServer
```

Para empezar un *ChatClient*, se pueden proporcionar los mismos parámetros. Por ejemplo:

```
java ChatClient nsport=9000 myport=9002
```

Esta instrucción lanza un ChatClient en el Puerto 9002 y se utiliza el puerto local 9000 para conectarse al *rmiregistry.*

Por otro lado, se puede proporcionar también un parámetro adicional (*nshost*) para especificar el anfitrión donde el servidor de nombres está corriendo. Por ejemplo:

```
java ChatClient nshost=192.168.1.1 nsport=9000 myport=9002
```

Con estos parámetros, este *ChatClient* intentará encontrar un *rmiregistry* que se esté ejecutando en el puerto 9000 del ordenador 192.168.1.1. En este ejemplo hemos supuesto que el *rmiregistry* se encuentra en dicha máquina, pero en general habrá que descubrir cuál es la IP del ordenador con el que estamos trabajando, lo que puede hacerse fácilmente usando por ejemplo el comando **ipconfig** en una terminal de Windows, o bien **ifconfig** en Linux.

Este ChatClient atenderá conexiones en el puerto 9002. Recordemos que, si va a haber más de un cliente o servidor corriendo en el mismo ordenador físico, hay que especificar un puerto diferente al puerto por defecto 9001, asignando un puerto nuevo por cada cliente que se lance.

Hay un parámetro adicional para *ChatClient* y *ChatServer*, y es el nombre del objeto *ChatServer*. Por defecto es “TestServer”. Se puede modificar este nombre utilizando el argumento “server” en ambos programas. Por ejemplo, las siguientes instrucciones empiezan un *ChatServer* y un *ChatClient* que utilizarán un objeto ChatServer denominado “NewServer”

```
java ChatServer server=NewServer
java ChatClient server=NewServer myport=9002
```

Importante: En esta práctica se asume que rmiregistry y ChatServer son ejecutados en el mismo ordenador.

Actividad 0: empezar a chatear

En este caso se debe aprender cómo chatear utilizando el Chat distribuido proporcionado. Para chatear necesitamos al menos 2 usuarios (ejemplo: Alice y Bob). Deberemos seguir los pasos indicados en el apartado anterior, es decir:

- Paso 1: iniciar *rmiregistry*
- Paso 2: iniciar un ChatServer
- Paso 3: iniciar un ChatClient para Alice. En este paso se arranca un primer ChatClient en el mismo ordenador donde los otros programas se están ejecutando. Como el ChatServer iniciado en el paso anterior usa el puerto 9001, hay que arrancar el *ChatClient* de Alice en un puerto diferente (por ejemplo, el puerto 9002).
- Paso 4: iniciar un ChatClient para Bob. En este paso se inicia un segundo cliente de chat para Bob. Este cliente puede ejecutarse en la misma máquina donde hemos lanzado los anteriores procesos (asignándole un puerto diferente, por ejemplo el puerto 9003), o bien en otra máquina diferente.

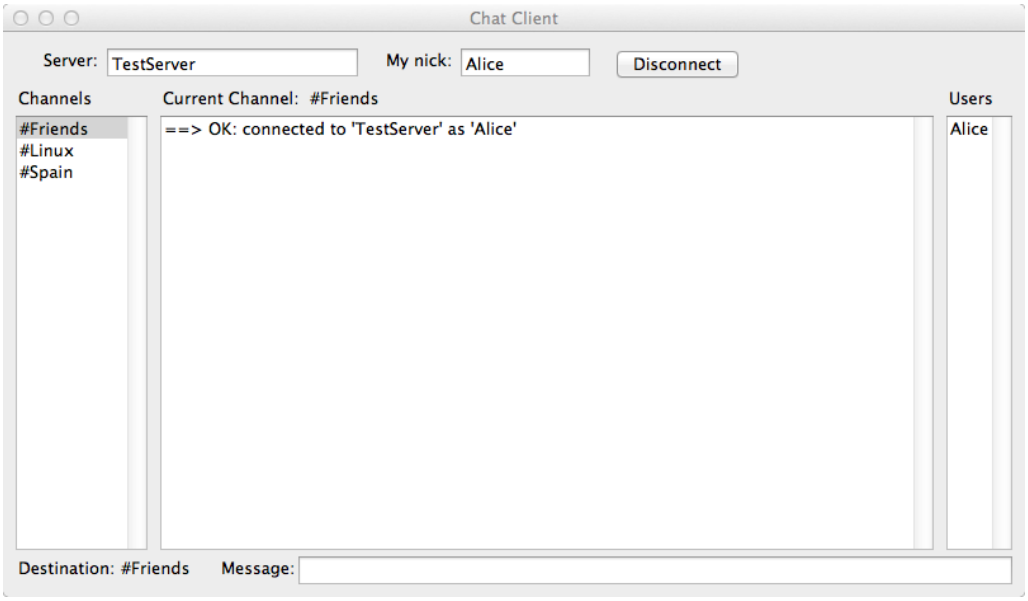
Para lanzar un ChatClient que se conecte al servidor de Chat de otra máquina diferente (accesible desde la nuestra, por ejemplo porque ambas máquinas están en la misma red local), habrá que lanzar el cliente especificando el ordenador donde se está ejecutando rmiregistry en la otra máquina. Por ejemplo, podemos asumir que es el ordenador 192.168.1.2 y que se ha lanzado rmiregistry también en el puerto 9000. El nuevo ChatClient lo lanzaremos en este caso con el puerto 9004.

```
java ChatClient nshost=192.168.1.2 nsport=9000 myport=9004
```

- Paso 5: chatear un poco.

La siguiente figura muestra la ventana del ChatClient. En ella se pueden observar tres áreas. Hay una línea superior para los parámetros de servidor. A continuación, hay un área central con barras de desplazamiento vertical (dividida en tres partes) y una tercera área en la parte inferior, donde se pueden escribir mensajes.

La línea superior tiene 2 campos de texto, uno para indicar el nombre del ChatServer al que el cliente se quiere conectar y otro donde hay que escribir el apodo del usuario. En este caso hay que utilizar Alice y Bob en cada uno de los ChatClient iniciados. Una vez escritos ambos campos, el ChatClient puede conectarse al ChatServer. Si la conexión se produce, se observará un mensaje de éxito.



El área con barras de desplazamiento está dividida en 3 sub-áreas. El área de la izquierda contiene la lista de canales existentes en el servidor de Chat al que se ha conectado el cliente. El área de la derecha contiene la lista de usuarios del canal al que se ha conectado el usuario y el área central que es más grande contiene los mensajes entrantes tanto del canal seleccionado actualmente como mensajes privados.

La línea inferior tiene una etiqueta de *Destination* y una caja de texto para poder escribir mensajes. El destino muestra qué canal o usuario recibirán los mensajes. Se puede seleccionar un canal o un usuario haciendo doble-clic en sus nombres.

Una vez se hace doble-clic sobre un canal, el ChatClient se une a dicho canal saliendo del canal en que se estuviera previamente. Si se hace un clic a un usuario de dicho canal no se deja el canal actual, sólo se cambia el destino del mensaje. De este modo se envía un mensaje privado al usuario.

Actividad 1: Obtener el proxy del servidor

En esta actividad y siguientes aprenderemos cómo está constituido el Chat distribuido orientado a objetos y qué invocaciones de objeto ocurren cuando se realizan las actividades básicas del chat.

Tiene a su disposición una implementación parcial de la aplicación *ChatClient*, llamada *ChatClientCSD*, que está incluida en **DistributedChatSources**.

Descargue del material de la práctica el archivo "**DistributedChatSources.jar**" y descomprímalo.

En esta actividad y las siguientes deberá actualizar la clase *ChatClientCSD* según vayamos indicando, para conseguir una funcionalidad similar a la de *ChatClient*. En este caso, el componente de interfaz de usuario está totalmente implementado, pero en cambio parte de la funcionalidad del cliente está pendiente de implementar.

La primera tarea que debe realizar *ChatClientCSD* consiste en buscar el objeto ChatServer, utilizando el servidor de nombres, y así obtener su referencia con la que poder invocarlo. Recordemos que previamente el servidor *ChatServer* habrá registrado su objeto principal `ChatServer` en el servidor de nombres.

Instrucciones

En la clase *ChatClientCSD*, implemente en el método `doConnect` lo siguiente:

1.a. Obtenga una referencia al servidor de nombres, utilizando el método `getRegistry` de la clase *`LocateRegistry`*.

1.b. Busque el objeto `ChatServer` en el servidor de nombres, utilizando el método *`lookup`* de la interfaz *`Registry`*. Tenga en cuenta que el nombre con el que se ha registrado *ChatServer* en el servidor de nombres lo tenemos guardado en la variable `serverName`. Además, guarde la referencia remota obtenida en la variable `srv`, que ha sido previamente definida al principio de la clase *ChatClientCSD*. Como podrá observar, dicha variable es de tipo `IChatServer`, que es una interfaz que extiende de `Remote`.

1.c. Trate las excepciones que se puedan producir. En concreto, si no se puede encontrar `rmiregistry` en el host y/o puerto indicado; y si el nombre de objeto indicado no está registrado en el servidor de nombres. Para ello, puede descomentar las líneas de tratamiento de excepciones que se han incluido en el código.

Con esto habremos obtenido un proxy del objeto `ChatServer`, por lo que desde *ChatClientCSD* podremos realizar invocaciones remotas a dicho objeto, utilizando para ello los métodos definidos en `IChatServer`.

NOTA: para ayudarle a realizar esta actividad, en el código de la clase ChatServer, en su método work(), puede observar cómo ChatServer registra su objeto principal `ChatServer` en el servidor de nombres.
Puede también echarle un vistazo al código de la clase ChatConfiguration para ver cuáles son los parámetros de configuración de la aplicación y sus valores por defecto.

Comprobación

1. Compile desde el terminal todos los ficheros *.java* con la instrucción:
   ```
   javac *.java
   ```

2. Lance a ejecución tanto `rmiregistry` como `ChatServer`. Recuerde que debe lanzarlos en el mismo directorio donde se tenga el código de `ChatClientCSD`.

3. Lance a ejecución `ChatClientCSD`, indicando el número de puerto del servidor de nombres (en párametro `nsport`) y facilitando un número de puerto no usado aún (en parámetro `myport`), para que sea usado por el cliente de chat. Por ejemplo:

   ```
   java ChatClientCSD nsport=9000 myport=9005
   ```

4. Verá que aparece la interfaz gráfica del cliente. Compruebe su funcionamiento actual.

a) Al indicar un nick y pulsar en `Connect`, deberá salir un mensaje del tipo "`Server has no channels`".
b) Si no se indica ningún nick y se pulsa en `Connect`, deberá salir un mensaje del tipo "`Nick`

`cannot be empty`".
c) Si no se indica ningún nombre de Server pero sí en el nick y se pulsa en `Connect`, deberá salir un mensaje del tipo "`Server name cannot be empty`".

Cuestión: *averigüe en qué clases del proyecto **DistributedChatSources** se detectan e imprimen todos estos mensajes indicados.*

Actividad 2: Conectarse a ChatServer y obtener la lista de canales

La segunda tarea a realizar consiste en que los clientes del Chat se conecten al *ChatServer*, utilizando el método *`connectUser`* de *ChatServer*. Una vez conectado, los clientes preguntan la lista de canales, y dicha lista será lo que devolverá el método `doConnect`. Este método, recordemos, habrá sido invocado por la interfaz de usuario (clase `ChatUI`) al pulsarse el botón "`Connect`". La lista obtenida se mostrará en la interfaz.

Nota: El método `connectUser` requiere que se le pase un objeto de tipo ChatUser, que contiene el nick o apodo del usuario y un objeto de tipo "MessageListener", que permite tratar los mensajes recibidos por el usuario.

Instrucciones

2.a. Cree un nuevo objeto `ChatUser`. El nick o apodo a utilizar se ha suministrado anteriormente como parámetro de entrada del método `doConnect`. Debe pasar como "MessageListener" al propio *ChatClientCSD*.

2.b. Utilizando el método `connectUser` de `srv`, conecte el cliente al ChatServer. Lance una excepción si la conexión no se ha realizado. La conexión no se llevará a cabo siempre que el apodo ya esté en uso.

2.c. Obtenga la lista de canales, usando el método *`listChannels`* de ChatServer. Asigne dicha lista al contenido de la variable `channels`.

Comprobación

Asegúrese de que está en ejecución tanto `rmiregistry` como `ChatServer`. Debe haberlos lanzado en el mismo directorio en donde se tenga *ChatClientCSD*.
1. Recompile y lance a ejecución *ChatClientCSD* (como en la actividad anterior).

Nota: puede lanzar *ChatClientCSD* desde el entorno de programación (por ejemplo, desde BlueJ) o bien desde el terminal. En el primer caso, deberá proporcionar en el main los parámetros de entrada, por ejemplo: `{"nsport=9000", "myport=9003"}`.
Si lo lanza desde el terminal, para evitar problemas de versiones de Java, deberá volverlo a compilar con la instrucción:

```
javac *.java
```

2. Compruebe el funcionamiento actual de la interfaz gráfica del cliente:
a) Al indicar un nick y pulsar en `Connect`, deberá salir la lista de canales y un mensaje del tipo "`OK: connected to 'TestServer' as XX`", siendo XX el nick que se haya indicado.

b) El botón de "`Connect`" ha pasado a mostrar "`Disconnect`". Compruebe que al pulsarlo se desconecta al usuario del canal.

c) Si se pincha en algún canal, se mostrará un mensaje de error del tipo "`BUG. Tell professor there are no users after joining`". Esto es porque todavía no se ha implementado el código correspondiente para unir el usuario al canal.

Cuestión: *Cuando los clientes del Chat preguntan la lista de canales, ¿qué proxies se emplean? ¿Qué métodos de esos proxies? ¿Se envía algún objeto como parámetro o valor de retorno de esos métodos? ¿Para qué?*

Actividad 3: Unirse a un canal

La tercera tarea a realizar consiste en que los clientes del Chat se conecten al canal seleccionado a través de la interfaz gráfica. Para ello, obtienen primero el objeto `ChatChannel` del canal solicitado (mediante el método *`getChannel`* de *ChatServer*) y se unen a dicho canal, usando el método *`join`* de *ChatChannel*.

Puede observarse que en la clase `ChatUI` se ha implementado el método *`onChannelSelected()`*, al cual se llama cada vez que se selecciona un canal en la interfaz gráfica. Y dentro de este método se llama al método `doJoinChannel` de la clase *ChatClientCSD*, que es donde realizaremos la conexión al canal seleccionado.

En esta actividad realizaremos también la salida del canal indicado, en el método `doLeaveChannel` de *ChatClientCSD*, para así completar la operatividad de la interfaz gráfica respecto a la selección de canales.

Instrucciones

3.a En el método `doJoinChannel` de la clase *ChatClientCSD*, implemente la unión al canal indicado en la variable `ch`. Puede consultar la interfaz *IChatChannel* para determinar el método y parámetros correspondientes a utilizar.

3.b En el método `doLeaveChannel` de la clase *ChatClientCSD*, implemente la salida del canal indicado en la variable `ch`. Puede consultar la interfaz *IChatChannel* para determinar el método y parámetros correspondientes a utilizar.

Comprobación

Recompile y lance a ejecución varios *ChatClientCSD* (como se realizaba en las actividades anteriores), y utilice en cada uno un usuario (nick) diferente. Compruebe el funcionamiento actual de la aplicación:
a) Si se pincha en algún canal, aparecerá el usuario en la lista de *Users* de ese canal. Y como "*Current Channel*" figurará el canal seleccionado.
b) La lista de usuarios de un canal debe actualizarse de forma apropiada conforme los usuarios se unan o abandonen los canales.
c) Si se escribe texto en la casilla de "Message", no se envía ningún mensaje pues todavía no está implementada esta funcionalidad.

Cuestiones: *Cuando un cliente del Chat se une a un canal, el canal avisa a los demás usuarios del canal.*
1) ¿Qué mensaje le llega a cada usuario del canal?
2) ¿Se ha creado algún objeto para el envío del mensaje? ¿Cuál o cuáles?
3) ¿A qué clases y métodos de dichas clases se ha llamado? Explique detalladamente los pasos que se siguen para que el canal avise a los demás usuarios de la unión de un nuevo usuario,

indicando: (i) Clase y método que se invoca; (ii) si se trata de un objeto remoto (y, por tanto, se hace uso de su proxy); (iii) parámetros del método (y si son por valor o referencia).

Actividad 4: Envío de un mensaje al canal

La cuarta tarea a realizar consiste en que los clientes del Chat puedan enviar mensajes de chat al canal en el que se encuentran. El cliente construye un objeto de tipo `ChatMessage` y lo envía al canal correspondiente. Y el canal se ocupa de retransmitir el mensaje a todos los usuarios conectados a ese canal. Por su parte, cada usuario, al recibir un mensaje, pide su contenido y lo mostrará en su interfaz gráfica.

Instrucciones

4.a En el método `doSendChannelMessage` de la clase *ChatClientCSD*, implemente el envío del mensaje al canal destino correspondiente. Puede consultar la interfaz *IChatChannel* para determinar el método y parámetros correspondientes a utilizar.

4.b Observe cómo se implementa el reenvío del mensaje a todos los usuarios conectados al canal en la clase *ChatChannel*.

Comprobación

Recompile y lance a ejecución varios *ChatClientCSD*, usando un usuario diferente para cada cliente. Compruebe el funcionamiento actual de la aplicación:
a) Si un usuario envía un mensaje al canal, en su interfaz de usuario debe mostrarse dicho mensaje.
b) Además, también debe aparecer en la interfaz de todos los usuarios conectados al canal donde se ha enviado el mensaje.

Cuestiones: *¿Cómo recibe un usuario un mensaje? ¿Qué métodos y clases se invocan? Identifique en qué parte del código se procede a analizar el tipo de mensaje recibido por parte de un usuario y a mostrar dicho mensaje por pantalla.*

Actividad 5: Envío de un mensaje privado a un usuario

Esta tarea consiste en que los clientes del Chat puedan enviarse mensajes privados directamente. El cliente emisor construye un objeto de tipo `ChatMessage` y lo envía al usuario destino correspondiente.

Instrucciones

5.a En el método `doSendPrivateMessage` de la clase *ChatClientCSD*, implemente el envío del mensaje al usuario destino correspondiente. Puede consultar la interfaz *IChatUser* para determinar el método y parámetros correspondientes a utilizar.
5.b Observe en el método `messageArrived` de la clase *ChatClientCSD* cómo el usuario trata los mensajes privados recibidos.

Comprobación

Recompile y lance a ejecución varios *ChatClientCSD*. Compruebe que si un usuario envía un mensaje a otro usuario del canal al que está conectado (fíjese que en "Destination" se indique el nombre del usuario destino), este mensaje solamente se refleja en las interfaces de estos dos usuarios, pero no en las de otros usuarios conectados a su mismo canal.

Cuestiones: *¿Cómo recibe un usuario un mensaje privado? ¿Cómo sabe que el mensaje es privado (y no del canal)? ¿Qué métodos y clases se invocan? Identifique en qué parte del código se procede a analizar el tipo de mensaje recibido por parte de un usuario y a mostrar dicho mensaje por pantalla.*

Actividad 6: Ampliación de la funcionalidad --> ChatRobot

Si ha resuelto con éxito todas las actividades anteriores, ¡enhorabuena!, habrá conseguido desarrollar la misma funcionalidad que el código utilizado en la Actividad 0.

En esta última actividad ampliaremos esta funcionalidad. Se desea implementar un *ChatRobot,* que es un proceso encargado de conectarse a un servidor dado y conectarse al canal indicado. Cada vez que un usuario se conecta a dicho canal, el `ChatRobot` le tiene que saludar enviando un mensaje "Hola apodo" al canal, siendo "apodo" el nick empleado por el usuario que se ha unido al canal.

Para ello, complete la implementación del `ChatRobot` en el archivo *ChatRobot.java* que se proporciona con la práctica.

Cuestiones

1) Explique qué objetos y qué operaciones son invocados cada vez que un usuario se conecta al canal "#Friends", asumiendo que el ChatRobot ya está conectado a ese canal.

2) Justifique la afirmación, "si lanzamos más de una aplicación ChatClient o ChatRobot en la misma máquina, deberán tener valores *myport* distintos. Si sólo se lanza una aplicación por máquina no es necesario modificar dicho valor".

3) La ubicación del servidor de nombres (*nsport*, *nshost*) debe ser conocida por todos, pero los clientes no necesitan conocer el *host* ni el *port* que corresponde al ChatServer. Indique la razón.

4) Asumiendo que *rmiregistry* y *ChatServer* están en una máquina A, y que lanzamos la aplicación *ChatRobot* en una máquina B, y una aplicación *ChatClient* en una máquina C, y sabiendo que estas dos aplicaciones indican la ubicación del servidor de nombres lanzado en la máquina A, describa:

a) dónde se ubican cada uno de los siguientes objetos (en qué máquina)
b) indique si se emplean una o varias instancias de cada objeto
c) explique cómo pueden comunicarse las instancias entre sí, es decir, cómo se obtienen sus referencias, qué tipo de invocaciones se realizan (locales o remotas), etc.

ChatServer	ChatChannel	ChatUser
ChatMessage	ChatClient	ChatRobot