Prácticas de laboratorio

Un chat distribuido orientado a objetos basado en RMI

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:

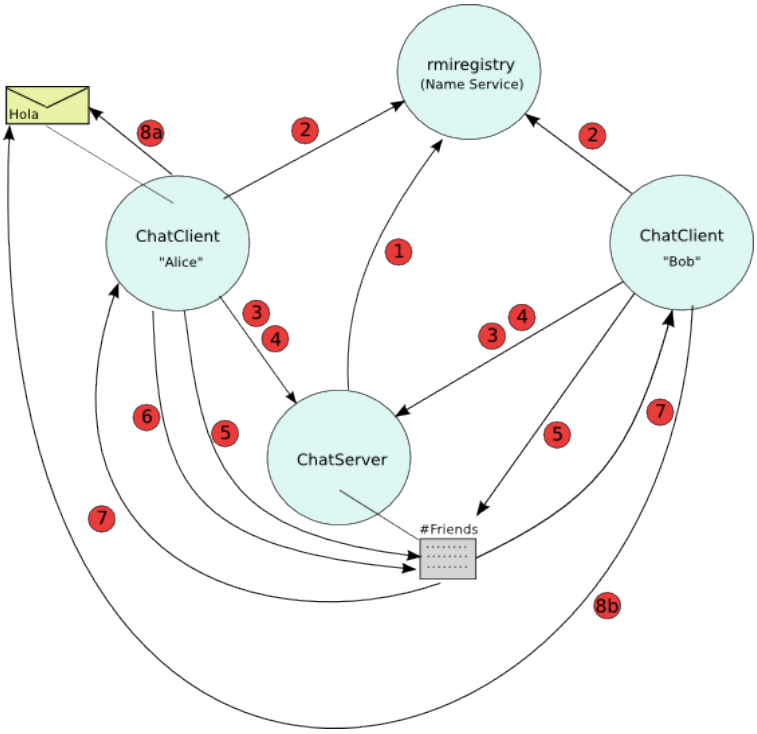
- Identificar los diferentes procesos que conforman una aplicación distribuida.
- Compilar y ejecutar aplicaciones distribuidas sencillas.
- Entender la función de servicio de nombres.
- Entender el paradigma de cliente/servidor y su extensión orientada a objetos para diseñar e implementar aplicaciones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

Esta práctica de laboratorio tiene que ser desarrollada por **grupos de dos estudiantes**.

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. Para este fin se pueden utilizar los laboratorios durante sus periodos de acceso libres de tiempo. También se puede utilizar vuestro ordenador personal.

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.

Dadas las instalaciones de laboratorio disponibles para esta práctica, se recomienda utilizar Windows, y las herramientas por línea de comandos *java*, *javac* y *rmiregistry*.

Los laboratorios tienen ordenadores con un firewall configurado, el cual bloquea la mayoría de puertos. Se pueden utilizar los puertos **9000-9499** para las prácticas de laboratorio. Algunos de los programas a utilizar en la práctica y la herramienta *rmiregistry* necesitan ser configurados para utilizar estos puertos.

El software proporcionado

Para esta práctica, se ha proporcionado un paquete *jar* denominado "DistributedChat.jar". Éste debe ser descargado y desempaquetado, comprobando que contiene los archivos con el código fuente java. También se debe comprobar que se pueden compilar, por ejemplo en

Windows abriendo el proyecto *DistributedChat.jar* con BlueJ y compilándolo; o bien utilizando ***javac *.java*** *(*en Linux o en Windows). Asimismo, compruebe que se pueden ejecutar los programas básicos.

NOTA: En las instalaciones de laboratorio, el programa **javac** se encuentra en el directorio "**C:\Archivos de programa\java\jdk1.8.0_71\bin"**

El paquete contiene dos programas ya preparados para ser ejecutados. Estos programas son **ChatClient.java** y **ChatServer.java**. 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 o consola de Linux. (También se puede ejecutar en Windows o Mac si es necesario).

```
rmiregistry 9000
```

NOTA: En las instalaciones de laboratorio, el programa **rmiregistry** se encuentra en el directorio **"C:\Archivos de programa\java\jdk1.8.0_71\bin"**

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 grupo 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 compiladas** (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, y opcionalmente 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 (en Windows).

Si el proceso *rmiregistry* se ha lanzado en la máquina local, no es necesario suministrar el nombre del host. Es decir, bastaría con:

```
java ChatClient myport=9002
```

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 1: empezar a chatear

En este caso se debe aprender cómo chatear utilizando el Chat distribuido proporcionado. Además, se debe aprender cómo está construido el Chat distribuido orientado a objetos y qué invocaciones de objeto ocurren cuando se realizan las actividades básicas del Chat.

Para chatear necesitamos al menos 2 personas (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.

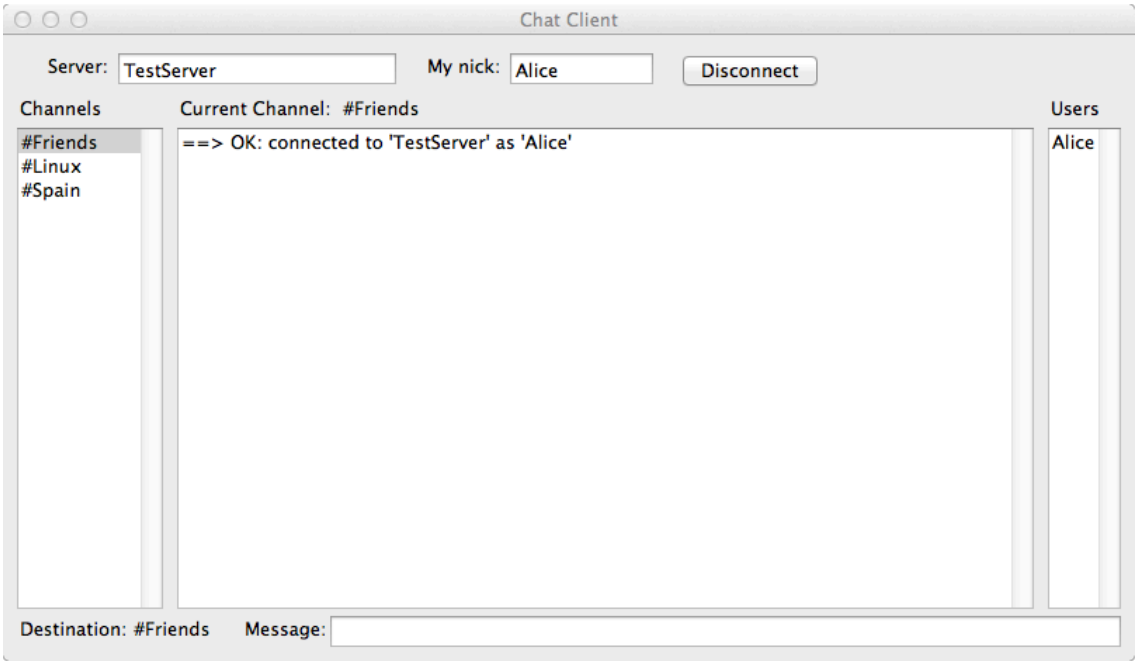
Para lanzar un ChatClient que se conecte al servidor de Chat de otra máquina diferente (por ejemplo, el ordenador del grupo vecino), habrá que lanzar el cliente especificando el ordenador donde se está ejecutando rmiregistry del grupo vecino. Por ejemplo, podemos asumir que es el ordenador 192.168.1.2 y que han 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 una 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.

Cuestiones

1) Cuando ChatServer registra su objeto principal ChatServer en el servidor de nombres, ¿qué acciones realiza para ello? ¿Qué interfaces se han requerido extender o implementar?

2) Cuando ChatClient busca el ChatServer utilizando el servidor de nombres, ¿qué acciones se realizan? ¿Qué nombre del objeto se ha utilizado? ¿Qué obtiene ChatClient del servidor de nombres?

3) Cuando los clientes del Chat se conectan al ChatServer, ¿qué proxies se emplean? ¿Qué métodos de esos proxies? ¿Se envía algún objeto como parámetro en esos métodos? ¿Para qué?

4) 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?

5) Cuando un cliente del Chat se une a un canal, ¿qué proxies emplea el cliente de chat para realizar la acción? ¿Qué métodos de esos proxies?

6) Cuando un cliente del Chat se une a un canal, el canal avisa a los demás usuarios del canal. Para ello, ¿qué proxies se emplean? ¿qué métodos de esos proxies? ¿Se envía algún objeto como parámetro de un método? ¿Cuál y para qué?

7) Si Alice envía un mensaje de chat sencillo al canal, ¿qué acciones se realizan? ¿Se crea algún objeto nuevo? Si es así, ¿para qué se utiliza?

8) Cuando el canal retransmite el mensaje de Alice a todos los usuarios conectados (al canal), ¿qué acciones realiza? ¿Se envía algún objeto como parámetro de un método?

9) Los usuarios, al recibir un mensaje, piden su contenido. ¿Qué acciones se realizan? ¿Qué diferencias hay entre una invocación local y una invocación remota?

10) ¿Qué objetos se crearán y qué invocaciones ocurrirán cuando el usuario Bob envíe un ***mensaje privado*** *a la usuaria Alice?*

Actividad 2

Se desea implementar un *ChatRobot,* que es un proceso encargado de conectarse a un servidor dado y conectarse al canal "#Friends". 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 al 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.