Práctica 2

Instant Messaging P2P con RMI

Pablo Martínez
Xavi Moreno

Extra Features

A parte de los puntos obligatorios, hemos añadido en nuestra aplicación una serie de funcionalidades que mejoran el rendimiento y la experiencia de usuario.

- **Server state auto-detect**: El peer sabe cual es el estado del servidor en todo momento (*polling*).
- **Disconnection notifier**: Cuando el peer detecta que el servidor ha caído se pone en marcha un mecanismo mediante el cual, cada vez que ve algún cliente desconectado lo notifica a todos los demás.
- **Client is typing**: En las comunicaciones punto a punto notificamos la acción de “está escribiendo”.

Manual de uso

Para poder ejecutar el chat, lo primero que hay que hacer es **poner en marcha el Servidor**. Para ello hay que poner en líneas de comando: *java -jar Server.jar <port>*. Éste pone en marcha automáticamente el *rmiregistry*. Luego hay que dejarlo encendido y entonces poder dar paso a ejecutar los Peers. Para ello hay que poner: *java -jar Peer.jar <ip> <port>*. Hay que poner la *ip* donde se encuentra el Servidor y el puerto (*port*) por donde escucha éste.

Una vez se ha puesto en marcha los peers deseados, pasamos a las instrucciones de la interfaz gráfica. Lo primero que nos aparece es una **pequeña ventana** en el centro de la pantalla que nos pide que introduzcamos nuestro nombre para poder chatear, este nombre será único mientra esté en uso. Una vez introducido un nombre válido se dará entrada a una **ventana más grande**, donde podremos empezar a chatear con quien deseemos. A la izquierda encontramos una lista de usuarios conectados en ese instante, y si miramos más abajo podemos encontrar una pestaña de grupos donde se listan los grupos en los cuales estás incluido. Más en el centro podemos ver donde se mostrarán los chats (tanto si es de grupo, éstos se distinguen porque incluyen paréntesis, o no). **Por cada chat abierto, se abrirá una pestaña**. Para finalizar si le damos a grupos podemos ver que el botón de ‘*Create group*’ se podrá hacer click en él (si es que hay usuarios). Si clicamos en él, nos aparecerá una ventanita donde nos pide que introduzcamos un nombre para el grupo, y seleccionemos todos los usuarios que queramos meter en el grupo. A más podemos añadir un nuevo usuario a un grupo ya creado.

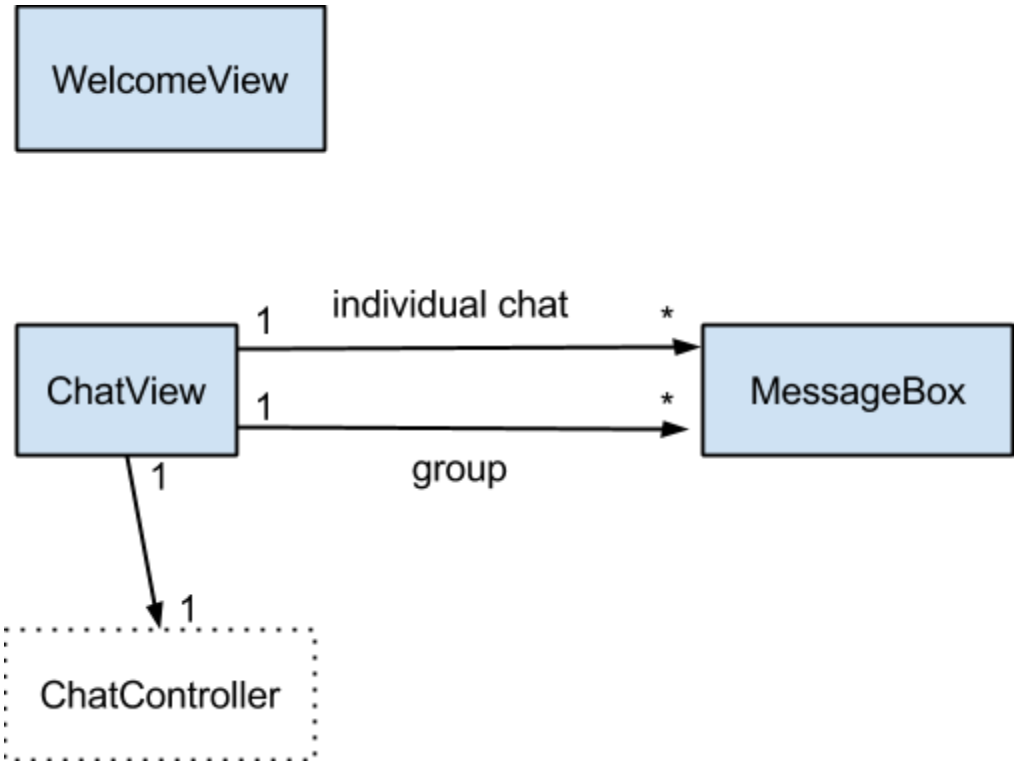
Estructura

Peer

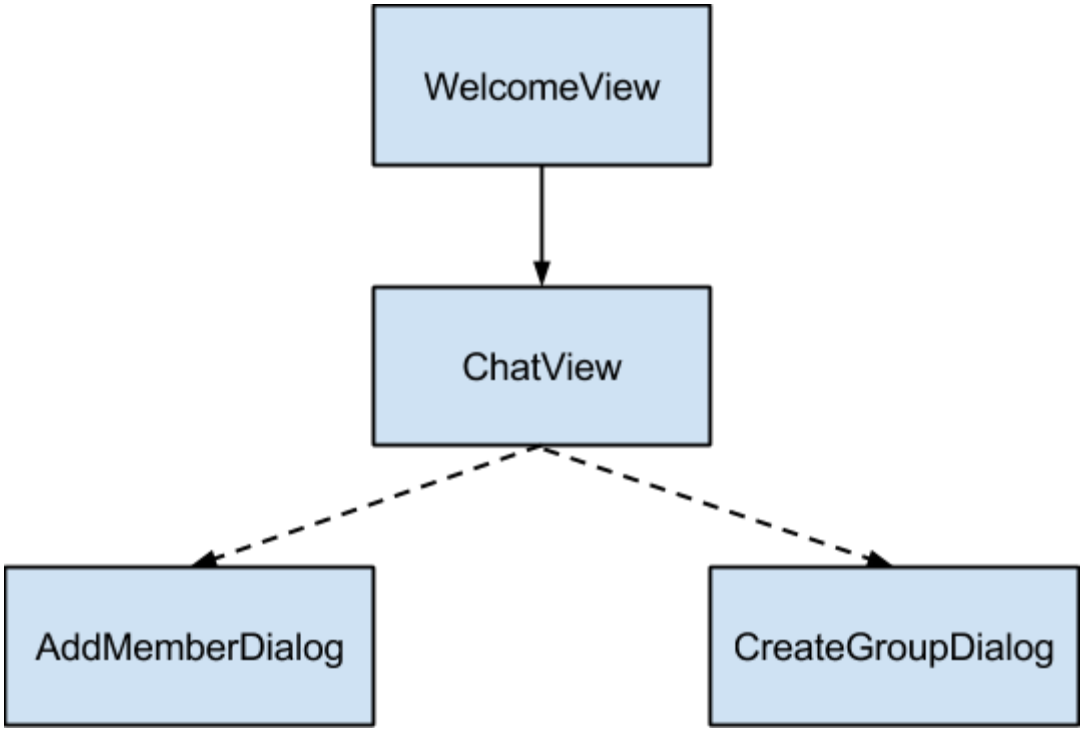
El peer está estructurado siguiendo el paradigma **MVC** (Model-View-Controller). Debido al tipo de aplicación que estamos diseñando, solo hay una clase en controlador (ChatController), que es la encargada de proveer las funciones necesarias a la vista.

View

Este es el diagrama de clases de la vista:



Este es el diagrama de diseño que sigue que siguen las ventanas:



MessageBox es la clase que conté el contenidor de tota la conversa. És a dir que per a cada conversa (tant individual o grup) li correspon un MessaBox.

La clase **ChatView** es la clase principal de la vista, esta clase controla todas las conversaciones. Esta contiene una lista de usuarios (lista de MessageBox), una lista de grupos (lista de MessageBox) y una ventana con con pestañas con cada uno de los usuarios.

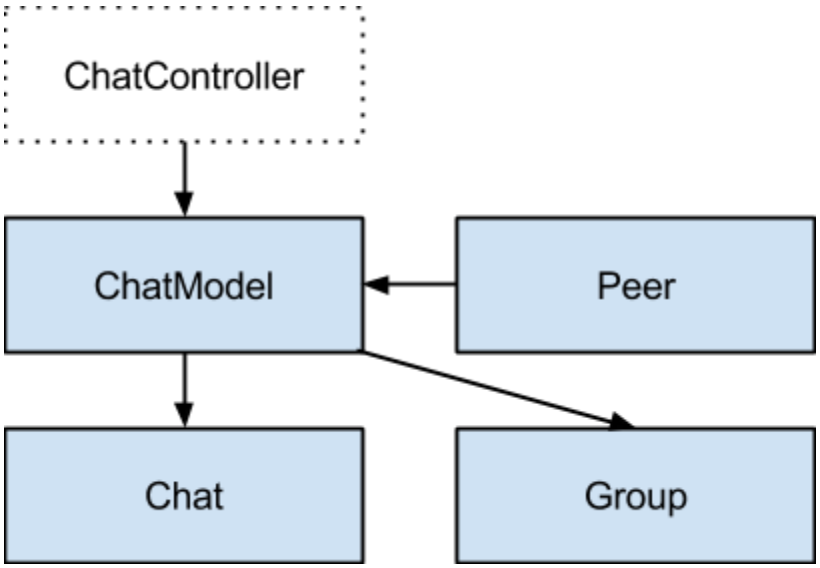
La clase que contiene el *main* es **WelcomeView**, es la clase de bienvenida del programa, que permite introducir al usuario su nombre de usuario y seguidamente registrarse en el servidor. Ésta da enlace a ChatView.

Como complementos hemos creado dos *JDialog*:

- **AddMemberDialog:** Diálogo que se muestra al añadir un nuevo o nuevos miembros a un grupo ya existente.
- **CreateGroupDialog:** Diálogo que se muestra para crear un grupo, pide que se seleccione un conjunto de usuarios y introducir un nombre al grupo.

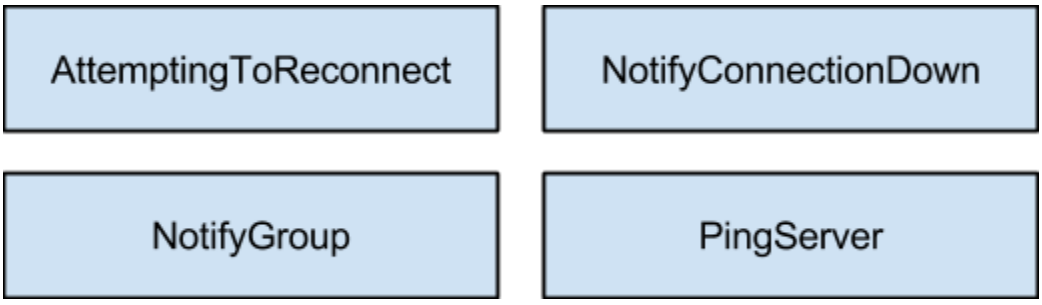
Model

El diagrama de clases del modelo es el siguiente:



Estas clases son las que nos darán las funcionalidades básicas de la aplicación. Donde el Peer es el objeto remoto que ponemos a disposición de los otros usuarios para que puedan “hablar” con nosotros.

Además, para añadir otras funcionalidades, y mejorar la experiencia de uso de la aplicación, tenemos una serie de **workers** que se encargan de realizar tareas muy concretas en hilos de ejecución distintos. Estos workers son:



- **AttemptingToReconnect**: es un thread que es lanzado cuando cae el server, cuya única función es hacer intentos periódicos de conexión. En el momento que logra reconectarse el thread muere.
- **NotifyConnectionDown:** este worker se encarga de notificar que una conexión ha caido. Solo se lanza cuando el servidor cae, y por tanto, no es capaz de notificar las desconexiones de los clientes.
- **PingServer**: es el thread complementario a **AttemptingToRecconect**. Va haciendo ping al server, si el server ha caido este thread morirá.
- **NotifyGroup**: este hilo es el encargado de notificar algo a alguien de un grupo.

Conexión model-view

Uno de los detalles con el que más nos peleamos fué el como conseguir hablar desde el modelo a la vista para notificar los eventos asíncronos, como que te abran un nuevo chat, que te añadan a un nuevo grupo o que te llegue un mensaje.
Para solucionar este problema hemos hecho uso de las interfaces. La vista mantiene una interfaz para el **chatRoom** (que es la que dispone de las funciones de crear nuevos chats, conectar y desconectar clientes, notificar que el servidor ha caído, o ha vuelto a ponerse en linea) y una interfaz para cada chat o grupo.

Exceptions

También tenemos un par de excepciones que nos serán útiles (sobre todo en el momento de testing). Ambas son excepciones en tiempo de ejecución. Con lo cual, no es obligatorio gestionarlas, ya que en un principio no deberían de ocurrir.

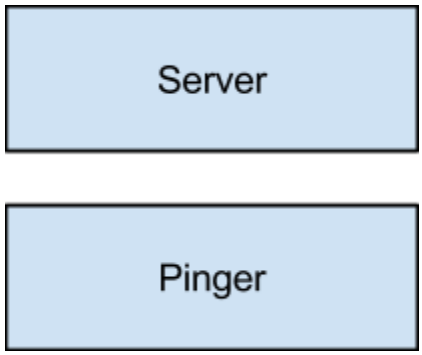
UserDisconnectedException: Esta excepción solo deberia saltar en un momento en que la concurrencia no estuviese bien gestionada y que se eliminara una conexión durante el proceso de escritura.

WrongAdresseeException: Esta excepción será lanzada si, el nombre de usuario que se escribe a través del controlador no existe en el sistema.

Server

El servidor con el que trabajamos en esta práctica es bastante sencillo. No contiene más que una clase principal que mantiene una lista de los clientes conectados (username, IPeer) y una serie de utilidades que hacen que la performance de la aplicación sea un poco mejor.

La estructura de clases en esta aplicación es la siguiente:



Donde **Server** es la clase principal, que implementa la interfaz **IServer** y se encarga de mantener los datos, y **Pinger** es un thread que corre en paralelo que va haciendo ping a todas las conexiones de forma periódica y en el caso de encontrar alguna desconexión, la notifica a todos los demás clientes.
Además disponemos en el server de una serie de **workers**. Que, tal y como lo hacíamos en el **peer** los utilizamos para notificar eventos a todos los clientes.

Sincronización

En cuanto a sincronización tenemos dos apartados. El primer apartado que hemos tenido que sincronizar ha sido todo lo referente a las comunicaciones multipunto. Es decir, aquellos mensajes que tenían que ser enviados a diferentes peers. Para ello hemos utilizado los **Workers**. Un worker, para nosotros es un **Thread** que contiene información relativa a la tarea que tiene que realizar. Por ejemplo el objeto **worker** que creamos para enviar un mensaje a una persona contendrá el username y el mensaje en sí. Cuando el envío acabe, el thread morirá o notificará una desconexión en caso de que el usuario esté desconectado.

Para no crear un thread por cliente al que tenemos que mandar, cosa que escalaría de forma caótica (imaginamos el caso en que tenemos que notificar a mil clientes), lo que hacemos es utilizar un **ThreadPool** lo cual nos reserva una serie de threads (nosotros hemos utilizado una pool de diez threads por servicio). Lo cual hace que el número máximo de threads que corren es de diez, aunque tengamos cien tareas por realizar. De esta forma evitamos que el hecho de lidiar con conexiones lentas nos ralentice la aplicación.
Los servicios para los que utilizamos **ThreadPool**s son:

- **Server**
    - Notificar desconexión
    - Ping
- **Peer**
    - Enviar mensaje a grupos
    - Añadir persona a un grupo
    - Cambiar nombre del grupo
    - Dejar un grupo
    - Notificar desconexión (Cuando cae el servidor)

Debido a que las tareas son muy concretas el único error de concurrencia que tenemos que solucionar es el de encontrarnos un cliente desconectado cuando estamos notificando que otro cliente se ha desconectado.
Este fenómeno lo hemos llamado **Disconnectception** y para solucionarlo, en el lado del Peer lo único que hemos hecho ha sido sincronizar la función ***notifyDisconnection*** con lo cual nos aseguramos que las notificaciones serán secuenciales, es decir, pongamos por ejemplo que tenemos, en nuestra aplicación, a cinco clientes y además el servidor ha caído. A los clientes los conocemos como cliente1, cliente2, …, cliente5.
Imaginemos que cliente1 cae y cliente2 se da cuenta. En esta situación cliente2 ejecutará la función notifyDisconnection (que recordemos que es ***synchronized***), durante la ejecución de la función nos encontramos con que cliente3 también está desconectado, con lo cual hacemos una llamada recursiva a la función notifyDisconnection desde uno de los workers. Al ser una función sincronizada lo que ocurrirá es que primero se esperará a que acaben todos los workers y luego se dará una condición de carrera para ver que otra desconexión notificamos primero.

El mismo fenómeno en el lado del servidor lo hemos solucionado de una forma ligeramente distinta. En lugar de hacer llamadas recursivas a una función, mantenemos una lista de los clientes que se han desconectado y que aún no hemos notificado. De esta forma el objeto **pinger** que recordemos que, tal y como hemos explicado antes, es un thread que corre infinitamente mientras el servidor está activo va haciendo polling a todas las conexiones, cuando encuentra una desconexión la añade a la lista. Acto seguido coje todas las entradas de la lista y notifica las desconexiones.

Otro tema de sincronización, que es mucho más básico, es el hecho de que los objetos Chat y Group pueden ser leidos y modificados de forma remota y local, con lo cual, para evitarnos errores de concurrencia lo único que hemos hecho ha sido sincronizar los objetos susceptibles a ser modificados de forma concurrente (como por ejemplo la lista de mensajes) en el bloque de código en el que lo modificamos.

Test

Para el día del test llevábamos toda la práctica realizada con la versión completa implementada. Este día pusimos en marcha el servidor a principio de clase, y éste aguantó durante las dos horas. Durante toda la clase no hubo ningún problema de sincronización de *threads*.

También hicimos la prueba de **apagar el servidor** para probar que todos los peers pudieran seguir hablando, y así fué, no hubo ningún problema. Luego volvimos a encenderlo para que los peers se **registraran automáticamente** en el servidor.

Durante todo el proceso de *testing*, se crearon grupos, se añadía gente nueva a los grupos, se iban, volvían, etc., pasaron muchas acciones y como ya hemos dicho en ningún momento el servidor ni los peers nos lanzaron un error.