

Universitat de Lleida

COMPUTACION DISTRIBUIDA Y APLICACIONES

Java Remote Method Invocation – Aplicación de
Mensajería

Paula Gallucci Zurita
47128784J

Contenido

Introducción	2
Diseño de la aplicación.....	3
Características de la implementación	4
Clases adicionales.....	4
Servidor – MessageManagerServant	4
Pruebas de funcionalidad.....	5
Permisos RMI	5
Verificación de Conexión.....	6
Errores y problemas	7

Introducción

En el presente documento se detallará la implementación de la Práctica de RMI de la asignatura Computación Distribuida y Aplicaciones, cursada en la Universidad de Lleida durante el curso académico 2019 – 2020.

El objetivo de dicho proyecto es la implementación de un sistema de mensajería mediante el mecanismo RMI de Java, el cual debe admitir múltiples usuarios y la creación de grupos de estos.

Adicionalmente, también se muestra el funcionamiento de la aplicación, así como se explican los errores más importantes que han aparecido durante el desarrollo.

Diseño de la aplicación

Siguiendo con las características de implementación de un Middleware Orientado a Mensajes (Message-Oriented Middleware, o MOM) y con la idea de hacer uso del mecanismo RMI (Java Remote Method Invocation), se ha comenzado definiendo el esquema de la comunicación cliente – servidor, el cual se puede encontrar en la *figura 1*.

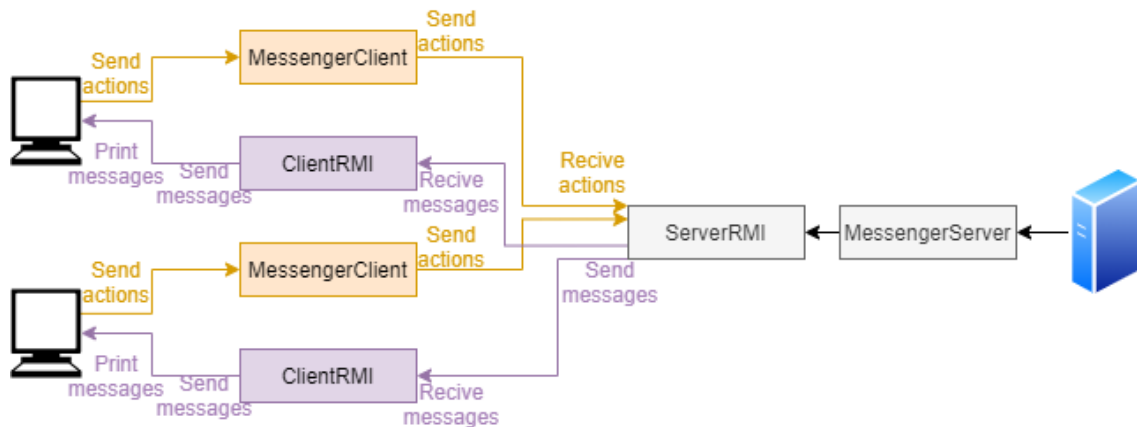


Figura 1: Esquema de comunicación entre los clientes y el servidor mediante la utilización de RMI

Tal y como se puede observar, se definen dos interfaces: **ServerRMI** y **ClientRMI**:

- **ServerRMI** → En esta interfaz se definirán los métodos que implementará el gestor de mensajes, tales como el envío de mensajes, la conexión de usuarios o la creación de nuevos grupos, entre otros.
- **ClientRMI** → En la interfaz del cliente se definirán los métodos del callback; aquellos métodos a los que el gestor de mensajes invocará cuando necesite informar de nuevos mensajes al cliente.

La definición de ambas interfaces se ha realizado en función a lo establecido por los compañeros en el foro de la asignatura.

No obstante, se han añadido métodos adicionales en **ClientRMI**, con el objetivo de poder notificar eventos adicionales, tales como la desconexión de usuarios o informar de la unión de usuarios a un grupo.

De la misma forma, para favorecer un control de errores más elaborado, también se han añadido excepciones personalizadas en los métodos de **ServerRMI**.

Características de la implementación

Clases adicionales

Se ha optado por crear dos clases adicionales para definir los agentes que interactúan con el servidor:

- User
 - Almacena:
 - Nombre del usuario
 - Contraseña
 - Estado
 - Cliente al que se deberán realizar las llamadas Callack pertinentes.
- Group
 - Almacena:
 - Nombre del grupo
 - Lista de usuarios miembros de dicho grupo

Ambas clases únicamente son utilizadas desde el lado del servidor.

Servidor – MessageManagerServant

Entendiendo por servidor el gestor de mensajes establecido en el objeto remoto, se han creado tres HashMaps:

1. RegisteredUsers<String, User> → Almacena a los usuarios registrados (conectados o no) indexados por su nombre.
2. CreatedGroups<String, Group> → Almacena los grupos creados indexados por el nombre.
3. ConnectedUsers<ClientRMI, String> → Almacena los nombres de los usuarios conectados indexados por el cliente que se ha conectado.

Si bien la última tabla podría haber sido obviada, se ha optado por implementarla dada la definición del método Logout, al cual se le pasa un ClientRMI en lugar de un usuario. De esta forma, se puede recuperar el nombre del usuario desconectado sin necesidad de iterar en cada uno de los usuarios registrados hasta encontrarlo.

Pruebas de funcionalidad

Para probar el correcto funcionamiento de la clase `MessageManagerServant`, la cual implementa `ServerRMI`, durante el desarrollo de la aplicación, se han realizado tests de forma paralela mediante JUnit 5.

En estos, se ha corroborado el correcto funcionamiento de los métodos de la interface.

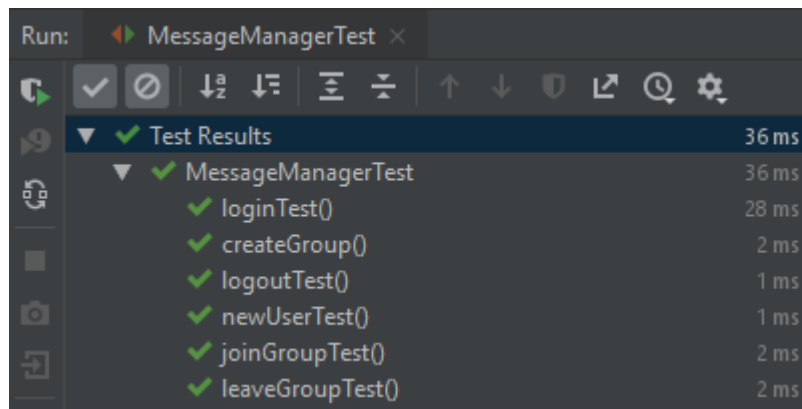


Figura 2: Resultado de la ejecución de los Tests

Debido a la simplicidad de los métodos de `ClientRMI`, no se ha considerado necesario la creación de Dummies para simular las respuestas y probar `ClientRMI` de forma independiente.

Permisos RMI

Se han definido los archivos de permisos para la aplicación RMI.

Por lo referente al servidor, se han establecido el siguiente archivo de política de permisos, el cual se puede encontrar en la carpeta del servidor:

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
    "connect,resolve,listen";
    permission java.util.PropertyPermission
    "java.rmi.server.hostname", "write";
};
```

En cuanto al cliente, se ha excluido la necesidad del acceso a la escritura de la propiedad `java.rmi.server.hostname`, mientras que se ha añadido el acceso al puerto HTTP (80):

```
grant {
    permission java.net.SocketPermission "*: 1024-65535", "connect,
    accept, resolver";
    permission java.net.SocketPermission "*:80",
    "connect,accept,resolve,listen";
};
```

Verificación de Conexión

Debido a las características de la configuración de la red local, no ha sido posible establecer conexión entre dos equipos independientes.

No obstante, para poder realizar las pruebas sin las interferencias que el router pueda ocasionar, se ha procedido a configurar una máquina virtual mediante VirtualBox. Las características de las máquinas que han realizado esta comprobación son:

- **Servidor - Máquina virtual**
 - Ubuntu 18.04.3 LTS
 - Red:
 - Adaptador NAT: IP 192.168.56.102
- **Cliente**
 - Windows 10 Home 18363.720
 - Red:
 - Adaptador Ethernet 4 (VirtualBox): IP 192.168.56.1

También se han deshabilitado todos los firewalls de ambos sistemas para poder realizar la conexión.

No obstante, debido a algún problema de enrutamiento relacionado con el cliente, los Callbacks no son capaces de llegar correctamente a dicho equipo, mientras que el servidor sí que recibe correctamente todos los comandos enviados por los clientes, tal y como se puede observar en la *Figura 3*.



```
18:00:00: Utiliza Login (Usuario: [Contraseña]) para conectar o Register (Usuario: [Contraseña]) (Image) para registrarte.
----- Introduce el comando. -----
Usuario Paula 1234
18:00:00: No se completó el registro. Ahora ya puedes conectarte.
----- Introduce el comando. -----
newsgroup: Roger
----- Introduce el comando. -----
Login Paula 123
ERROR: El usuario y la contraseña introducida no coinciden
----- Introduce el comando. -----
Login Paula 1234
ERROR: Error remoto - java.rmi.RemoteException: RemoteException occurred in server thread; nested exception is:
java.rmi.ConnectIOException: Exception creating connection to: 192.168.134.2; nested exception is:
java.net.SocketTimeoutException: no route to host (Host unreachable)

/Computribuida-RMI/Serv$ java MessengerServer 192.168.56.102
Cargando servicios RMI. Levantando gestor de mensajes
El gestor de mensajes se va a levantar en la IP 192.168.56.102
La ruta del objeto es: rmi://192.168.56.102/Messenger
Objeto bindeado rmi://192.168.56.102/Messenger
DEBUG: Solicitud de registro entrante: Usuario Paula, Contraseña 1234
DEBUG: Creando al usuario Paula.
DEBUG: Enviando mensaje de tipo 5
DEBUG: Solicitud de conexión entrante: Usuario Paula, contraseña 123
DEBUG: La contraseña introducida por el usuario Paula no es correcta
DEBUG: Solicitud de conexión entrante: Usuario Paula, contraseña 1234
DEBUG: Conectando al usuario Paula y avisando a los otros usuarios
DEBUG: Enviando mensaje de tipo 0
```

Figura 3: Establecimiento de conexión entre dos equipos

Sin embargo, se confía en que pueda ser posible ejecutarlo en remoto en equipos correctamente configurados.

En caso de realizar una ejecución local, no hay ningún tipo de error a la hora de recibir los mensajes, tal y como se puede apreciar en la *Figura 4*.

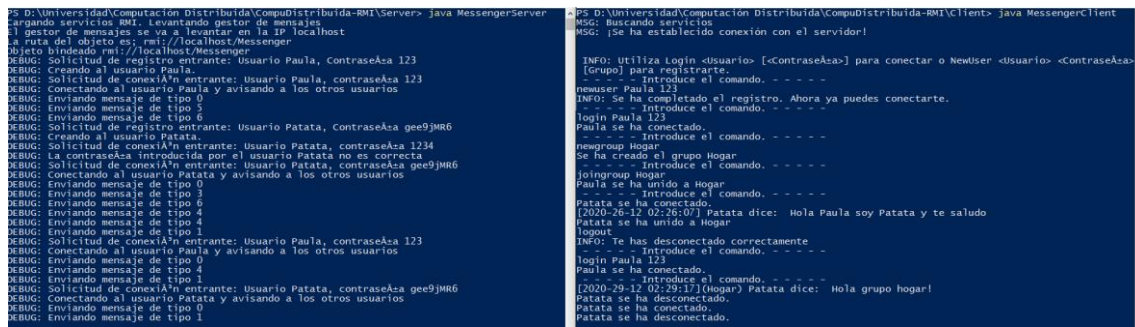


Figura 4: Conexión dentro de la misma máquina.

Errores y problemas

Algunos de los errores que se han producido durante la implementación del proyecto son:

- Nula comunicación entre dos equipos (**Solucionado**):
 - Problema: El cliente era incapaz de conectarse al servidor aun pudiendo establecer Ping entre ambos equipos. El análisis de la red mostró que el servidor estaba bloqueando todos los paquetes salientes por el puerto 1099.
 - Solución: Desactivación total del Firewall, tanto por parte del cliente como por parte del servidor (si bien no sería la solución recomendada en un caso de proyecto en producción)
- No se encuentra una clase compilada (**Solucionado**):
 - Problema: A la hora de ejecutar el cliente y/o el servidor desde el equipo que utiliza Ubuntu (hosteado en una máquina virtual), la creación del objeto devolvía el error `java.lang.ClassNotFoundException` a pesar de estar correctamente compilada en el mismo directorio.
 - Solución: Especificar el classpath al ejecutar el programa.
- Imposible recibir llamadas a métodos (**Solucionado**):
 - Problema: Aun estableciendo conexión entre el cliente y el servidor, en cuanto el cliente realizaba una llamada remota a alguno de los métodos del *ServerRMI*, este no devolvía ni recibía nada, y el cliente acababa lanzando la excepción `java.util.concurrent.TimeoutException`.
 - Solución: Definir la propiedad `java.rmi.server.hostname` en el servidor, asignándole el valor de la IP del servidor.
- Mal enrutamiento de los Callbacks (**Sin solucionar**):
 - Problema: Cuando se realiza la conexión entre cliente y servidor utilizando máquinas distintas (y, por tanto, saliendo de localhost), cuando el servidor llama a una de las funciones Callback, esta devuelve un error

java.net.NoRouteToHostException, dado que intenta acceder a la IP 169.254.154.2, imposibilitando la recepción de mensajes.

- Soluciones probadas:
 - Deshabilitar Npcap Loopback Adapter
 - Reasignar direcciones IP locales.
 - Modificación del archivo Hosts