PRÁCTICAS DE SISTEMAS DISTRIBUIDOS.

Práctica 4: Programación de un servidor de grupos de usuarios en Java RMI.

En esta práctica se implementará un servidor centralizado de grupos de usuarios, facilitando rutinas para crear y eliminar grupos, dar de alta y baja a sus miembros, así como para bloquear y desbloquear las altas/bajas de sus miembros.

Los usuarios participantes tendrán un nombre textual (alias), así como los grupos en los que participan. Los clientes pueden crear nuevos grupos, convirtiéndose en sus propietarios, y sólo ellos podrán dar de baja los grupos que han creado.

Uno de los estudiantes del equipo puede preparar el código del servidor, y el otro el código de los clientes remotos.

PASOS A SEGUIR:

1. Crearemos un nuevo proyecto desde Netbeans, con nombre *CentralizedUserGroups*, de tipo *Java→Java Application*.

2. Con el botón derecho creamos una nueva clase de Java, con nombre *GroupMember*. Los objetos de esta clase serán los miembros de un grupo. Esta clase únicamente contendrá dos campos: nombre del miembro (*String*) y hostname (*String*).

   Se creará el correspondiente constructor de la clase con los argumentos necesarios.

3. Crearemos el interfaz remoto para el servidor de grupos (Java Interface), con nombre *GroupServerInterface*, facilitando los servicios:

   - *boolean createGroup(String galias, String oalias, String ohostname)*: Para crear un nuevo grupo, con identificador textual *galias*. El propietario del grupo es su creador, con alias *oalias*, ubicado en *ohostname*. Devuelve false si ya existe un grupo con ese alias, true en caso de éxito.
   - *boolean isGroup(String galias)*: Para determinar si existe un grupo con el nombre indicado. Devuelve true si existe, false en caso contrario.
   - *boolean removeGroup(String galias, String oalias)*: Se emplea para eliminar el grupo con identificador textual *galias*,. Sólo se puede eliminar si el argumento *oalias* coincide con el propietario del grupo. Devuelve true en caso de éxito, false en caso contrario.
   - *boolean addMember(String galias, String alias, String hostname)*: Para añadir como nuevo miembro del grupo *galias* al usuario con el alias *alias* indicado, que está ubicado en *hostname*. Retorna *false* si ya existe como miembro o cuando el grupo *galias* no existe. Retorna true en caso de éxito. Esta invocación será bloqueante cuando las altas y bajas estén bloqueadas. En tal caso, una vez desbloqueadas las altas y bajas podrá completarse la operación.
   - *boolean removeMember(String galias, String alias):* Para eliminar el miembro *alias* del grupo con alias *galias*. Determinar situaciones de error, y retornar *false* en ese caso, considerando en particular que no puede eliminarse al propietario del grupo (creador del grupo). Esta invocación será bloqueante cuando las altas y bajas estén bloqueadas.

- *boolean isMember(String galias, String alias)*: Determina si el usuario de alias indicado *alias* es miembro del grupo *galias*. Devuelve *false* si el grupo indicado no existe o no es miembro del grupo.
- *String Owner(String galias)*: Devuelve el alias del propietario del grupo indicado, o null si no existe dicho grupo.
- *boolean StopMembers(String galias)*: Se bloquean los intentos de añadir/eliminar miembros del grupo. Devuelve *false* si no existe ese grupo. Esta operación **no es bloqueante**, lo que debe bloquearse es la ejecución de sucesivas invocaciones de altas/bajas de miembros de ese grupo hasta que se ejecute *AllowMembers* sobre ese grupo.
- *boolean AllowMembers(String galias)*: Para permitir de nuevo las altas y bajas de miembros del grupo. Aquellos que estaban bloqueados deben ser desbloqueados y prosiguen su ejecución con la operación que estaban realizando (alta o baja). Devuelve *false* si no existe ese grupo.
- *LinkedList<String> ListMembers(String galias)*: Para devolver la lista de nombres de miembros de un grupo.
- *LinkedList<String> ListGroups()*: Para devolver la lista de nombres de grupos actual.

Recuerda que en Java RMI todos los métodos del interfaz elevan la excepción *RemoteException*.

4. Implementamos una clase de nombre *GroupServer*, que implementa el interfaz anterior, utilizando la clase *UnicastRemoteObject*. Para representar los grupos utilizaremos una clase anidada *Group* (dentro de *GroupServer*), con los campos siguientes: nombre del grupo, propietario y lista de miembros (objetos de *GroupMember*), junto a otros que puedan necesitarse. Así, la lista de grupos será una lista privada de objetos de la clase *Group*.

   Además, para evitar problemas con los accesos concurrentes, los servicios ofertados por *GroupServer* **deben ejecutarse en exclusión mutua**. Para ello utilizaremos un cerrojo (*Lock*). Puedes utilizar la clase *ReentrantLock* para gestionarlo. Todos los métodos del servidor emplearán el cerrojo para garantizar la exclusión mutua.

   Para los bloqueos de altas y bajas se propone utilizar una variable booleana y una variable de la clase *Condition* **en cada uno de los grupos**. De esta forma, cuando se bloquean altas y bajas sólo afecta al grupo o grupos en que se haya indicado. Obsérvese que el hecho de bloquear altas y bajas no puede impedir que otros clientes puedan realizar otras operaciones con el servidor, o con ese grupo[1].

   Seguidamente se irán implementando los métodos del servidor, siguiendo las especificaciones indicadas en el punto 3. Es importante prestar atención al bloqueo de las altas y bajas y cerciorarse de que no se producen deadlocks u otros efectos colaterales indeseados, derivados de una implementación incorrecta. Por ello, es importante a la hora de probarlo someter el sistema a pruebas de diferente tipo.

5. Crearemos entonces la clase que encierra el programa principal del servidor. Para implementarlo seguir los pasos descritos en clase. Para establecer la política de seguridad usaremos un fichero de texto, que ubicaremos en la carpeta *NetBeansProjects*. Su contenido es el indicado en clase para el servidor.

[1]Se asumen conocidas las herramientas de programación concurrente necesarias para completar esta parte.

Para indicar cuál es el fichero que utilizamos, incluiremos dentro del programa principal la orden:

System.setProperty("java.security.policy", "pathname_fichero");

También debemos establecerse el gestor de seguridad (*SecurityManager*) y lanzar el registro sobre el puerto 1099 del servidor:

LocateRegistry.createRegistry(1099);

Debes tener en cuenta que si el registro ya estaba en ejecución se elevará la excepción *RemoteException*. Esto puede ocurrir durante las pruebas, al ejecutar el programa varias veces. Esta situación debe tratarse adecuadamente. Si se lanza el registro externamente (comando *rmiregistry*) debemos tener la precaución de hacerlo desde la carpeta en que se encuentran los paquetes del servidor, y con la variable CLASSPATH asignada a dicho directorio (puede ocurrir que no encuentre la clase del interfaz en caso contrario).

Una vez tengamos el registro en ejecución, podremos dar de alta el servidor con (*bind* o *rebind*).

6. A continuación se implementará en la máquina del cliente un proyecto *ClientGroupServer*, como aplicación java, el cual implementará el interfaz *GroupServerInterface*, considerando:

   - El cliente **puede** estar en una máquina distinta al servidor, en ese caso tendría que establecer su política de seguridad y gestor de seguridad según lo descrito en clase.
   - Será necesario escribir el código del interfaz *GroupServerInterface* de nuevo en el lado del cliente (puede copiarse desde el servidor).
   - El cliente, al inicio de su ejecución, **pedirá su alias, obtendrá automáticamente su hostname** y abrirá un menú[2] para pedir las distintas opciones sobre el servidor: crear grupo, eliminar grupo, añadirse/eliminarse como miembro de un grupo (este cliente invocador), bloquear/desbloquear altas y bajas en un grupo, mostrar miembros de un grupo, mostrar grupos actuales, comprobar si existe un grupo, mostrar propietario de un grupo, comprobar si es miembro de grupo y terminar su propia ejecución. Observad que al crear un grupo nuevo, automáticamente se debe incorporar al cliente invocador como propietario del mismo.

7. **Para realizar las pruebas es necesario desactivar el cortafuegos en ambas máquinas.** Para realizar las pruebas con dos nodos clientes remotos se copiará el proyecto del cliente a otra estación.

[2]Debe implementarse también, aunque puede ser simplemente textual.