PRACTICAS DE SISTEMAS DISTRIBUIDOS.

Práctica 2: Programación de un servidor de grupos en Java RMI.

En esta práctica se implementará un servidor de grupos de objetos distribuidos, 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.

Usaremos una versión centralizada, de modo que un nodo ejerce la función de servidor del grupo, y los clientes (remotos) se conectan a él para solicitarle los servicios mencionados.

PASOS A SEGUIR:

Crearemos un nuevo proyecto desde *netbeans*, de nombre *CentralizedGroups*, de tipo *Java→Java Application*. Con el botón derecho creamos:

1. La clase serializable *GroupMember*, que recogerá la información siguiente de cada miembro de un grupo: alias del miembro (*String*), su hostname (*String*), identificador numérico del miembro dentro del grupo (int) e identificador numérico del grupo (int). Se creará el correspondiente constructor de la clase con los argumentos necesarios.

2. Interface remoto para el servidor de grupos, con nombre *GroupServerInterface*, facilitando los servicios:

   - *int 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*. Se retorna el identificador numérico del nuevo grupo en el servidor (mayor o igual a 0) o -1 en caso de error (por ejemplo, ya existe un grupo con ese alias).
   - *int findGroup(String galias)*: Para localizar un grupo por su identificador textual. Retorna su identificador o -1 si no existe.
   - *boolean removeGroup(String galias, String oalias)*: Para eliminar el grupo con identificador textual *galias*, si existe y su propietario es el objeto con identificador textual *oalias*. Retorna *true* si pudo borrarse, *false* en caso contrario.
   - *boolean removeGroup(int gid, String oalias)*: Para eliminar el grupo con identificador numérico *gid*, si existe y su propietario es el objeto con identificador textual *oalias*. Retorna *true* si pudo borrarse, *false* en caso contrario.
   - *GroupMember addMember(int gid, String alias, String hostname)*: Para añadir como nuevo miembro del grupo al objeto con el alias indicado, que está ubicado en *hostname*. Retorna *null* si ya existe.
   - *GroupMember isMember(int gid, String alias)* : Obtiene un miembro del grupo *gid* por su alias como miembro del grupo. Retorna *null* si no existe.
   - *boolean StopMembers(int gid)*: Se bloquean los intentos de añadir/eliminar miembros del grupo. Devuelve *false* si no existe ese grupo.
   - *boolean AllowMembers(int gid)*: Para permitir de nuevo las inserciones y borrados de miembros del grupo. Devuelve *false* si no existe ese grupo.

3. La clase *ObjectGroup*, para implementar cada grupo de objetos, con campos para identificar textual y numéricamente el grupo, su lista de miembros actuales (GroupMember) y a su propietario (GroupMember).

   Para asignar números de identificación diferentes a los nuevos miembros se empleará un contador estrictamente creciente, del cual se irán asignando los nuevos identificadores al dar de alta a los nuevos miembros. Cuando se da de baja un miembro del grupo su número identificador sencillamente se deja de usar.

   Dado que desde diferentes objetos se pueden requerir operaciones simultáneas a un grupo, se usará un cerrojo (*Lock*) para el control de la **exclusión mutua** durante todas las operaciones. Para ello puede utilizarse la implementación *ReentrantLock*. Asimismo, el control de bloqueos de las altas y bajas se realizará mediante una variable de tipo *Condition*, vinculada al cerrojo de exclusión mutua[1].

   Métodos (Identificar argumentos):

   - Constructor *ObjectGroup*: Crea un nuevo objeto de grupo cuyo alias e identificador numérico se indica en los argumentos, y cuyo primer miembro es el indicado en los argumentos (alias y hostname). Este último se convierte además en *propietario* del grupo. En este caso no es necesario el bloqueo de exclusión mutua.
   - *GroupMember isMember*: Se comprueba si un objeto cuyo alias se indica como argumento es miembro del grupo. Si lo es retorna su correspondiente objeto de clase *GroupMember*, en caso contrario retorna *null*.
   - *GroupMember addMember*: Se incluye el objeto cuyo alias se indica en los argumentos como nuevo miembro del grupo, salvo que ya existiese en dicho grupo uno con el mismo alias. Al nuevo miembro se le asignará un nuevo identificador. Retorna *null* si ya existe un miembro con el alias indicado. Debe bloquearse al invocador si las inserciones y borrados de sus miembros están bloqueadas.
   - *boolean removeMember*: Se elimina del grupo al objeto cuyo alias se indica como argumento. No puede eliminarse al propietario del grupo ni un objeto que no es miembro del grupo, retornando *false*.
   - *StopMembers()* y *AllowMembers()*: para controlar los bloqueos de nuevos miembros.

4. La implementación del servidor, con nombre de clase *GroupServer*, a partir de *UnicastRemoteObject*. Debe implementar una lista de grupos (*ObjectGroup*), para lo cual puede utilizarse la clase predefinida *LinkedList* de Java. Para evitar problemas con los accesos concurrentes, los servicios ofertados deben ejecutarse en exclusión mutua, usando un cerrojo (*Lock*) a tal efecto, pero debe considerarse su liberación cuando se invoquen las operaciones específicas sobre un *ObjectGroup*, una vez localizado, y siempre que sea lo último realizado en el método en concreto desde el que se invoquen (de otra forma se producirán deadlocks o se vulneraría el principio de exclusión mutua en las operaciones del servidor, respectivamente).

5. El programa principal del servidor, que debe establecer la política de seguridad desde un fichero de permisos (puede utilizarse el mismo para diferentes prácticas, y puede ubicarse en la misma carpeta *NetBeansProjects*), con la orden:

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

[1] Estos conceptos relativos a la programación concurrente en Java se asumen ya conocidos.

A continuación debe establecerse el gestor de seguridad (*SecurityManager*), y lanzar el registro sobre el puerto 1099 del servidor:

LocateRegistry.createRegistry(1099);

Teniendo en cuenta que si el registro ya estaba en ejecución se elevará la excepción *RemoteException*[2]. Una vez en ejecución el registro el servidor tendrá que darse de alta (*bind* o *rebind*).

6. Finalmente crearemos los clientes, los cuales serán también servidores (*UnicastRemoteObject*), en previsión de *callbacks* de posteriores prácticas. Utilizaremos un interface *ClientInterface*, que de momento estará vacío. De momento no será necesario crear un registro para ellos y darlos de alta en el mismo, aunque al invocar al constructor de *UnicastRemoteObject* con "*super*" sí se habrán exportado como servidores.

   A continuación se implementará una clase *Client* que implemente dicho interface, 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 de idéntica forma que el servidor.
   - El cliente pedirá su alias y abrirá un menú[3] para pedir las distintas opciones sobre el servidor: crear grupo, eliminar grupo, añadir/eliminar miembro de grupo, bloquear/desbloquear altas y bajas, y terminar su propia ejecución.

     Para finalizar su ejecución el cliente debe darse de baja como servidor con el método *unexportObject* de *UnicastRemoteObject*.

7. Finalmente se comprobará el correcto funcionamiento lanzando el servidor y varios clientes, y probando los distintos servicios.

[2]Nos puede ocurrir al ejecutar el programa varias veces.
[3]Debe implementarse también, aunque puede ser simplemente textual.