PRACTICAS DE SISTEMAS DISTRIBUIDOS.

Práctica 3: Multienvío no ordenado en Java RMI

En esta práctica ampliaremos la funcionalidad de la práctica 2 para implementar un multienvío (no necesariamente ordenado) de mensajes entre los miembros de un grupo (grupo cerrado).

Para ello cada cliente implementará una cola local de mensajes, donde se depositarán los mensajes a él destinados. Los envíos se realizarán mediante un *thread* separado, para permitir los envíos simultáneos dentro de un mismo grupo. Asimismo, mientras haya envíos en curso habrá que inhibir las altas y bajas de miembros del grupo, para evitar inconsistencias.

Los clientes ahora implementarán *callbacks* para admitir los mensajes procedentes de los grupos a los que pertenece. Para ello será necesario crear un registro local (cada cliente en su máquina local), y darlos de alta en ellos con un nombre local (su alias por ejemplo).

PASOS A SEGUIR:

1. Incluiremos un nuevo campo en la clase *GroupMember* para identificar el número de puerto del registro de cada cliente. Habrá que incluir esta información como parámetro en el constructor. También habrá que incluir esta información en los argumentos de los métodos de creación de grupos y de alta de nuevos miembros en los grupos.

2. Crear una nueva clase serializable *GroupMessage*, cuyos objetos son mensajes de un grupo. Campos: Mensaje (array de bytes) y emisor (*GroupMember*). El grupo ya está indicado dentro del objeto de clase *GroupMember*.

3. Quitaremos (comentaremos) los servicios *StopMembers* y *AllowMembers* del interface *GroupServerInterface* y del servidor.

   En dicho interface *GroupServerInterface* incluimos un nuevo servicio:

   - *boolean sendGroupMessage(GroupMember gm, byte[] msg)*: Multienvío del mensaje *msg* por parte de *gm*, al grupo al que pertenece *gm*. Retorna *true* al terminar de enviar el mensaje a todos los miembros del grupo. Mientras hay envíos en curso en un grupo no pueden darse de alta o baja miembros en el mismo.

4. En la interface *ClientInterface* añadiremos dos servicios:

   - *void DepositMessage(GroupMessage m)*: Es un *callback* (método invocado desde el servidor), para depositar el mensaje *m* en la cola local del cliente. Como puede haber varios objetos enviando mensajes a la vez al cliente, éste debe garantizar la exclusión mutua en dichas operaciones[1]. Para ello se utilizará un *Lock* (implementación: *ReentrantLock*).
   - *byte[] receiveGroupMessage(String galias)*: Para recoger de su cola local el siguiente mensaje correspondiente al grupo de alias indicado. Si no hay ninguno se bloquea hasta que llegue uno. Si no existe ese grupo retorna *null*.

[1] La implementación de Java RMI podría desplegar un thread separado por cada petición, depende de la implementación concreta de Java RMI que usemos.

5. Crear en *ObjectGroup* dos nuevos métodos:

   - *void Sending()*: Para controlar el número de envíos en curso, incrementando un contador a tal efecto. Se invoca al iniciar un envío, y permite activar el bloqueo de altas/bajas de miembros del grupo. El mecanismo de bloqueo es similar al utilizado en la práctica anterior.
   - *void EndSending(GroupMember gm)*: Avisa del final de un envío por parte de *gm* a su grupo, para desbloquear las altas y bajas de miembros del grupo si no hay otros envíos en curso.

6. Crear una nueva clase *SendingMessage*, como extensión de *Thread* (hilo separado), para el envío de un mensaje (*GroupMessage*) a los miembros del grupo, exceptuándose al emisor. Deben identificarse los argumentos requeridos en el contructor, y en su método *run* deben realizarse los envíos, invocando el método *DepositMessage* de los destinatarios. Obsérvese que en esta versión no contemplamos un envío fiable u ordenado, si falla algún envío no lo tendremos en cuenta.

   Al finalizar todos los envíos se invocará el método *EndSending* del grupo.

7. Incluir la implementación del método *sendGroupMessage* en el servidor. Habrá que comprobar que el grupo del emisor existe realmente, y conformar un objeto de clase *GroupMessage*. Para enviar el mensaje se invoca el método *Sending* de *ObjectGroup* y se despliega un nuevo objeto de clase *SendingMessage* que se encargará del envío. Una vez creado dicho *thread* la invocación a *sendGroupMessage* termina retornando *true*.

8. En la clase *Client* habrá que implementar los nuevos servicios indicados en su interface. Debe implementarse la cola local de mensajes (se propone usar *Queue*, con la implementación *LinkedList*), y un *Lock* para controlar la exclusión mutua en su manejo. El bloqueo a la espera de mensajes puede hacerse mediante una variable de tipo *Condition* asociada a dicho *lock*.

   En el programa principal del cliente hay que incluir la creación del registro local en el puerto 1099 (si está en una máquina distinta al servidor) y darse de alta en él con su alias como identificador del servicio. Después desplegará el menú de opciones, del cual habremos eliminado las opciones de bloquear/desbloquear altas y bajas de miembros del grupo, y se incluirán las de envío y recogida de mensajes de grupos, con sus correspondientes implementaciones.

CUESTIONES:

1. ¿Es bloqueante o potencialmente bloqueante la invocación a *sendGroupMessage*?. Indica todas las posibles causas, si las hay, de bloqueo potencial.
2. Incluir dentro de *SendingMessage*, justo antes de invocar a *DepositMessage* para cada objeto miembro un *delay* aleatorio (entre 30 y 60 segundos), para comprobar que mensajes diferentes procedentes de un mismo emisor pueden llegar en orden distinto a los diferentes miembros del grupo.
3. ¿Qué ocurre si un cliente intenta recoger un mensaje de un grupo al cual ya no pertenece?. ¿Y si lo intenta de un grupo existente al que no ha pertenecido nunca?.
4. ¿Qué sucede si un objeto no recoge un mensaje procedente de un grupo y posteriormente desaparece el grupo?. ¿Podría posteriormente recoger el mensaje?. Indica soluciones posibles a este problema.