PRACTICAS DE SISTEMAS DISTRIBUIDOS.

Práctica 4: Multienvío ordenado FIFO en Java RMI

En esta práctica ampliaremos de nuevo la funcionalidad de la práctica 3 para implementar un multienvío ordenado FIFO de mensajes entre los miembros de un grupo (grupo cerrado).

Se pretende implementar una semántica FIFO en el envío de mensajes, de modo que los receptores reciban todos los mensajes procedentes de un mismo emisor en el mismo orden en que éste los ha enviado. Obsérvese que esta semántica no implica que los mensajes procedentes de distintos emisores se reciban en el mismo orden por todos los receptores.

Para ello asociaremos a cada mensaje enviado por cada miembro del grupo un número de secuencia (de dicho miembro del grupo), de forma que los receptores recogerán los mensajes atendiendo a dichos números. Para no complicar excesivamente el problema se inhibirán las altas y bajas una vez que se haya ordenado un primer multienvío sobre el grupo, de otra forma habría que resincronizar los números de secuencia entre todos los miembros al dar de alta a nuevos miembros.

Supondremos que los envíos son fiables, de modo que si se produce algún fallo en algún envío sencillamente mostraremos el fallo en pantalla, aunque no lo trataremos. El fallo en algún envío de hecho conllevará que los clientes fallidos si posteriormente se recuperan no puedan seguir recibiendo mensajes debido a la inconsistencia en los números de secuencia. Estas situaciones no las contemplamos.

PASOS A SEGUIR:

1. Incluir dentro de *GroupMessage* un nuevo campo entero, *seqnum*, para el número de secuencia de un mensaje. Habrá que modificar el constructor en consecuencia.

2. Incluir en *ObjectGroup* una lista de números de secuencia (para cada miembro del grupo). Podemos considerar que si un miembro no aparece en la lista es porque aún no ha enviado mensajes al grupo, y su primer número de secuencia es 0. Esta lista nos servirá así para determinar el número de secuencia a utilizar para el siguiente mensaje enviado por un miembro del grupo.

   Añadir entonces un nuevo método a *ObjectGroup*:

   - *int nextSeqNumber(int memberid)*: Devuelve el siguiente número de secuencia que debe ser utilizado para el miembro con identificador numérico indicado. Actualiza el número almacenado en la lista (o lo inserta) para la siguiente invocación. Retorna -1 en caso de fallo (no existe ese miembro de grupo).

3. Modificar la implementación de *Client* para incluir:

   - Lista de números de secuencia esperados de los miembros de cada grupo al que pertenece. Puede crearse una lista de grupos, donde en cada nodo guardamos a su vez la lista de números de secuencia de los miembros del grupo. Para minimizar el coste de inicialización de la lista, podemos partir de listas vacías, de modo que si en la lista de grupos no tenemos información de un grupo, es porque aún no se ha recibido nada de él, y si en la lista de un grupo no tenemos información de uno de sus miembros es

porque su número de secuencia esperado es 0, al no haber recibido aún ningún mensaje proveniente de dicho miembro del grupo.

Se propone implementar dos métodos privados para el manejo de esta lista de listas:

* *int seqnumber(GroupMember gm)*: Que devuelve el siguiente número de secuencia correspondiente a ese miembro de grupo. El siguiente mensaje recibido de ese objeto debe contener ese número de secuencia.
* *updatenumber(GroupMember gm)*: Para actualizar el número de secuencia esperado de *gm*, incrementándolo en uno (o dándolo de alta si ha llegado el primer mensaje proveniente de él).

- El método *receiveGroupMessage* debe ser modificado para controlar que sólo puede recoger el siguiente mensaje en la cola del grupo indicado cuyo número de secuencia se corresponda con el esperado para el emisor del mismo (método *seqnumber*), en otro caso se bloquea hasta que llegue uno del grupo que cumpla esas condiciones. Una vez extraído un mensaje, debe actualizarse el número de secuencia esperado de ese emisor (método *updatenumber*).

4. Modificaremos la implementación de *sendGroupMessage* en el servidor, de modo que debe determinar el siguiente número de secuencia de mensaje a utilizar para el emisor de que se trate (método *nextSeqNumber*). La invocación a este método debe inhibir definitivamente las altas y bajas de miembros del grupo. Para ello se eliminará la invocación a *EndSending*, que puede ser a su vez eliminado.

5. Comprobar el funcionamiento correcto del multienvío FIFO creando varios clientes, y lanzando varios mensajes casi simultáneamente desde varios de ellos. Cada uno debe enviar varios mensajes, y debe comprobarse que se reciben en el orden correcto (orden de envío del emisor), aunque posiblemente intercalando mensajes de diferentes emisores en diferente orden en cada receptor.

CUESTIONES:

1. En esta versión se han inhibido las altas y bajas de miembros cuando se inician los envíos de mensajes en un grupo, y si se intenta realizar una de estas operaciones, el invocador quedará bloqueado indefinidamente. ¿Crees que en el caso concreto de las bajas se podrían permitir sin causar problemas?. ¿Cómo podríamos dar de alta un nuevo miembro e incorporarlo al sistema de números de secuencia de forma dinámica?.

2. Hemos partido de un multienvío fiable, pero normalmente será bastante probable que se produzcan fallos de conexión con los clientes. Si un cliente tiene un fallo temporal y no se le pueden enviar algunos mensajes, ¿de qué forma podríamos reincorporarle al grupo y que pudiera seguir recogiendo los futuros mensajes sin problemas?.

3. Supongamos que asociamos los números de secuencia al grupo en su conjunto, en lugar de a sus miembros, es decir, que tenemos un contador de mensajes enviados por grupo, y en cada invocación a *sendGroupMessage* éste es incrementado y asociado al mensaje. ¿Qué modelo semántico de entrega se estaría implementando de esta forma?.

4. **Ejercicio avanzado:** Implementar en casa un multienvío fiable FIFO, es decir, *sendGroupMessage* sólo tiene éxito si todos los mensajes han sido entregados a todos los miembros del grupo.

   IDEA: Utilizar la confirmación de mensajes en una segunda ronda por parte del servidor, el cual debe esperar a que todos los mensajes hayan sido entregados para enviar mensajes de confirmación

a todos los clientes. Java RMI garantiza las entregas si en las invocaciones a *DepositMessage* no se eleva una excepción *RemoteException*. Cuando todos los mensajes han sido confirmados el método *sendGroupMessage* podrá retornar *true*, mientras tanto permanecerá esperando la finalización del *thread SendingMessage* (puede usarse el método *join* de *Thread* para esperar su terminación, pero debe liberarse el *lock* previamente).

Si una entrega falla será necesario anular los mensajes enviados a los demás miembros del grupo y habrá que invalidar ese número de secuencia en los clientes. Puede incluso contemplarse la posibilidad en la implementación de dar de baja a los clientes fallidos que no respondan al cabo de un cierto número de reintentos.

Obsérvese que es también posible la situación en que todos los clientes reciben los mensajes, pero luego falla el envío de alguno de los mensajes de confirmación (reiteradas veces). En este caso, dado que los demás clientes recogerán el mensaje, puede contemplarse la baja del cliente para el cual no se ha logrado la confirmación, de esta forma se mantiene la consistencia del multienvío en los restantes miembros del grupo.