Sistema de memoria compartida distribuida en Java (JavaDSM)

Se trata de un proyecto práctico de desarrollo **en grupos de 2 personas** cuyo plazo de entrega termina el **20 de mayo**.

Objetivo de la práctica

La práctica consiste en desarrollar un sistema de memoria compartida distribuida (*Distributed Shared Memory*) en un entorno Java que permita que el alumno llegue a conocer de forma práctica el tipo de técnicas que se usan en estos sistemas, tal como se estudió en la parte teórica de la asignatura.

Con respecto al sistema que se va a desarrollar, se trata de un entorno con las siguientes características (se recomienda revisar la teoría de la asignatura para repasar todos los conceptos que se nombran a continuación):

- Es una implementación de tipo RT-DSM, aplicable, en principio, a cualquier tipo de objeto de Java (más adelante, se explicarán qué restricciones existen finalmente en cuanto a qué tipo de objetos compartidos puede gestionar este sistema).
- Usa un modelo de coherencia de entrada (EC) existiendo, por tanto, dos tipos de entidades en este sistema: cerrojos y objetos compartidos. Ambas entidades tienen asociado un nombre (`String`) que les identifica de manera única y global durante una ejecución del sistema DSM.
- Como exige el protocolo EC, el programador debe vincular cada objeto compartido con el cerrojo que se usa para asegurar la coherencia en los accesos a dicho objeto.
- Los cerrojos ofrecerán tanto acceso exclusivo como compartido (*múltiples lectores/único escritor*).
- Utiliza un único gestor centralizado (`ServidorDSM`) para manejar los cerrojos.
- Como especifica el modelo de coherencia de entrada, cuando un proceso entra en una sección crítica, hay que asegurarse de que vea los últimos cambios que se hayan hecho sobre los objetos compartidos asociados a ese cerrojo desde la última vez que este proceso accedió en sección crítica a los mismos (recuerde que si hay accesos fuera de una sección crítica no es necesario garantizar ningún comportamiento específico: el resultado es impredecible).
- En el modelo de coherencia EC, no es necesario realizar ninguna acción a la salida de una sección crítica de un cerrojo. Los datos modificados se quedarán en ese proceso hasta que otro entre en la sección crítica controlada por ese mismo cerrojo. Sin embargo, en el sistema que se propone en este ejercicio, este esquema plantearía el problema de qué hacer si el primer proceso termina antes de que el segundo quiera entrar en esa sección crítica. Para solventarlo, aunque sea menos eficiente, el gestor centralizado también actuará como almacén de objetos compartidos.
- En consecuencia, la politíca de actualización será la siguiente:
    - Al entrar en la sección crítica de un determinado cerrojo, el proceso contactará con el gestor informándole de qué versión tiene de cada objeto compartido asociado a dicho cerrojo. El gestor enviará aquellos objetos cuya versión sea más actual que la que posee el proceso que actúa como cliente.
    - Al salir de una sección crítica en modo exclusivo, el proceso enviará al gestor la nueva versión de todos los objetos asociados al cerrojo. Nótese que, por simplicidad, no se va a implementar ninguna estrategia para detectar qué objetos han sido modificados: se considerará que todos lo han sido.

En cuanto a la tecnología de comunicación usada en la práctica, dadas las características de la misma, se ha elegido Java RMI (si no está familiarizado con el uso de esta tecnología puede consultar esta guía sobre la programación en Java RMI).

Para completar esta sección introductoria, se incluye, a continuación, un ejemplo de un cliente de este sistema, lo que permite tener un primer contacto con el API del mismo.

```
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import dsm.*;

public class ClienteDSM {
    static public void main (String args[]) {

       if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        // ¿"StringBuffer []"? ¿Por qué no "StringBuffer"?
        StringBuffer [] v1 = new StringBuffer[1];
        v1[0] = new StringBuffer();

        ObjetoCompartido o1 = new ObjetoCompartido("objecto1", v1);

        DSMCerrojo cerrojo = null;
        try {
            cerrojo = new DSMCerrojo("cerrojo");
            cerrojo.asociar(o1);

            if (!cerrojo.adquirir(true)) {
                System.err.println("Error en adquirir en modo exclusivo");
                return;
            }
            System.out.println("Valor de objeto al entrar:");
            System.out.println(v1[0]);
            System.out.println("S.critica exclusiva: cambia valor objetos");
            v1[0].append("|hola");
            if (!cerrojo.liberar()) {
                System.err.println("Error en liberar");
                return;
            }
            cerrojo.desasociar(o1);
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteDSM:");
            e.printStackTrace();
        }
    }
}
```

Dadas las características de este proyecto práctico, donde existe un proceso gestor con un doble rol de manejador de cerrojos y de almacén de objetos, se propone un desarrollo incremental en tres fases:

- Implementación de un servicio remoto de cerrojos.
- Implementación de un almacén remoto de objetos.
- Implementación de la funcionalidad final: la clase *DSMCerrojo*, que se construirá usando los dos servicios remotos previamente implementados.

Arquitectura del software del sistema

Antes de pasar a presentar cada una de esas tres fases, se especifica en esta sección qué distintos componentes (clases) hay en este sistema:

- `FabricaCerrojos` (y `FabricaCerrojosImp`): interfaz remota, y clase que la implementa, que proporciona la funcionalidad requerida para la creación de cerrojos remotos. **La interfaz ya está definida pero la implementación, obviamente, no está desarrollada.**
- `Cerrojo` (y `CerrojoImp`): interfaz remota, y clase que la implementa, que proporciona las operaciones de *adquirir*, de forma compartida o exclusiva, y *liberar* un cerrojo remoto. **La interfaz ya está definida pero la implementación, obviamente, no está desarrollada.**
- `Almacen` (y `AlmacenImp`): interfaz remota, y clase que la implementa, que proporciona la funcionalidad para el almacenamiento y recuperación de objetos compartidos. **La interfaz ya está definida pero la implementación, obviamente, no está desarrollada.**
- `ObjetoCompartido`: clase que permite asociar un nombre a un objeto del programa haciendo de este forma posible su uso como objeto compartido. Asimismo, incluye el número de versión asociado a ese objeto compartido. Esta clase se apoya en la clase `CabeceraObjetoCompartido` que gestiona la información de control propiamente dicha del objeto compartido (su nombre y versión, pero no el objeto en sí mismo). **Ambas clases están completamente implementadas y no se deben modificar**.
- `DSMCerrojo`: clase que se apoya en los dos servicios remotos implementados para desarrollar la funcionalidad final requerida. Este tipo de cerrojos también ofrece las operaciones de *adquirir* y *liberar*, basadas en las operaciones del mismo nombre proporcionadas por el servicio de cerrojos remotos, pero añade a las misma la funcionalidad de los volcados y recuperaciones de objetos compartidos usando el servicio de almacenamiento remoto. **Esta clase no está implementada**.
- `ServidorDSM`: se encarga de activar las implementaciones de los dos servicios remotos registrándolos en el `rmiregistry`. Recibe como argumento el número de puerto en el que está ejecutando el `rmiregistry`, que debe haber sido arrancado previamente. **Esta clase ya está implementada**.
- Clientes de prueba para las distintas partes del proyecto.

Los ficheros de la práctica están organizados en tres directorios:

- `cliente`: contiene todas las clases que prueban la funcionalidad del sistema. Este directorio sólo se debería modificar si se requieren programas de prueba adicionales.
- `servidor`: contiene la clase que corresponde al proceso servidor (`ServidorDSM`). No debería modificarse este directorio.
- `dsm`: contiene todas las clases restantes, que son las que implementan la funcionalidad del sistema. En aras de mejorar la organización del proyecto, todas estas clases están incluidas en un paquete Java denominado `dsm`.

Además de las diversas clases, en los distintos directorios se incluyen *scripts* para facilitar la compilación de las clases y la ejecución de los programas, así como la distribución de las clases requeridas por el cliente y el servidor, en forma de ficheros JAR, teniendo en cuenta que éstos pueden residir en distintas máquinas.

Servicio remoto de cerrojos

Dado que es necesario crear dinámicamente cerrojos, se plantea usar el esquema más habitual en estos casos: utilizar una interfaz remota (`FabricaCerrojos`) registrada en el `rmiregistry` que proporcione un

método para crear cerrojos, cada uno de los cuales será a su vez una interfaz remota (`Cerrojo`).

A continuación, se muestra la interfaz `FabricaCerrojos`:

```
package dsm;
import java.rmi.*;

public interface FabricaCerrojos extends Remote {
        Cerrojo iniciar(String s) throws RemoteException;
}
```

La clase que implementa esta interfaz debe usar algún tipo de contenedor para almacenar qué cerrojos existen en el sistema, guardando en cada entrada del contenedor la vinculación entre el nombre global del cerrojo y el objeto de tipo `Cerrojo` asociado al mismo. El alumno puede usar el contenedor que considere oportuno.

El comportamiento del método `iniciar` será el siguiente:

- Si no existía un cerrojo con ese nombre, crea un nuevo objeto remoto, lo incluye en el contenedor asociándole el nombre especificado y retorna una referencia de ese nuevo objeto remoto.
- Si ya existía un cerrojo con ese nombre, simplemente se retorna una referencia al mismo.

Por simplicidad, y teniendo en cuenta que este tipo de sistemas se usan para aplicaciones paralelas donde todos los procesos son cooperantes, una vez creado un cerrojo remoto, permanecerá en el sistema todo el tiempo que esté arrancado el servidor. No se proporcionará ningún mecanismo, implícito o explícito, para destruir un cerrojo remoto.

A continuación, se muestra la interfaz `Cerrojo`:

```
package dsm;
import java.rmi.*;

public interface Cerrojo extends Remote {
    public void adquirir (boolean exclusivo) throws RemoteException;
    boolean liberar () throws RemoteException;
}
```

El parámetro del método `adquirir` especifica si se trata de un acceso exclusivo o compartido.

Por lo que se refiere a la implementación, se trata básicamente de la construcción de un clásico esquema de sincronización que permite múltiples lectores pero un único escritor. El alumno puede optar por el mecanismo que considere oportuno. Por ejemplo, puede usar una solución de bajo nivel usando métodos sincronizados (ya se han definido como tal en el código inicial de apoyo suministrado) y los métodos `wait` y `notify|notifyAll` de `Object`.

Por simplicidad, el único control de errores que se va a plantear es asegurarse de que un cerrojo estaba previamente *cerrado* cuando llega una petición de liberarlo (ese método devolverá `false` en caso de que no se cumpla este requisito). Un mejor control de errores sería asegurarse de que la llamada a liberar proviene del mismo cliente que tiene el cerrojo adquirido. Ese tipo de garantías no son para nada triviales en un entorno como Java RMI, puesto que la única información del cliente que le llega al servidor es la dirección de su máquina, pero, obviamente, podría haber varios clientes (varias JVM) en la misma máquina. Una solución podría basarse en entregar algún tipo de descriptor (número aleatorio, clave, ...) en la operación adquirir y que el cliente tenga que pasar como parámetro ese descriptor en la operación de liberar. Sin embargo, por simplicidad, obviamos este aspecto.

Para probar la funcionalidad de esta parte del proyecto, se proporciona el cliente `TestCerrojoRemoto`. Este programa de prueba recibe como argumentos la máquina y el puerto donde ejecuta el proceso `rmiregistry`.

Almacén remoto de objetos

La capacidad de Java para *serializar* objetos y el uso de este mecanismo que hace Java RMI para enviar los objetos especificados como parámetros de un método o recibir el objeto retornado por el mismo, permiten plantearnos la idea de construir un repositorio remoto de objetos, funcionalidad requerida para la construcción de nuestro sistema DSM. Este servicio de almacén proporcionaría un método para enviar uno o más objetos (múltiples objetos por eficiencia, como se comentará más adelante), `escribirObjetos`, y otro para recuperarlo(s) desde el almacén remoto, `leerObjetos`, encargándose Java RMI de todo el proceso de *serialización*, por muy complicado que fueran los objetos involucrados (por ejemplo, una lista de empleados). Además, gracias al mecanismo de *serialización*, se trata de un *almacén tipado*: permite detectar si por error se intenta recuperar un objeto compartido sobre una clase de un tipo diferente.

Dado que este mecanismo puede usarse con cualquier objeto (excepto en caso de que éste no sea *serializable* por su propia esencia, como, por ejemplo, un descriptor de fichero), el planteamiento inicial del proyecto era poder aplicarlo a cualquier objeto de una aplicación Java. Sin embargo, ciertas características de Java han limitado el tipo de objetos que se podrán usar como compartidos en este sistema. Sin entrar en muchos detalles, estas restricciones están relacionadas con la falta de mecanismos genéricos de copia en Java, a diferencia de lo que ocurre con otros lenguajes como C++ (["There is no automatic way to copy any given object in Java"]). Aunque estos problemas pueden afrontarse usando técnicas como la reflexión, por el momento se ha optado por una solución más pragmática: *envolver* el objeto a compartir en un *array*, lo que proporciona un nivel de indirección que posibilita una copia indirecta mediante el método `arraycopy` de `System`. Así, por ejemplo, si se desea compartir un `StringBuffer`, habría que definir un objeto de tipo `StringBuffer []`. Hay que resaltar que el uso de un *array* como un envoltorio trae a su vez sus propias limitaciones por la propia definición del lenguaje Java (por un lado, no se pueden crear *arrays* de tipos genéricos; por otro lado, un *array* de tipos primitivos no es de tipo `Object []`).

Clase `ObjetoCompartido`

Esta clase, ya implementada, permite asociar un nombre con un objeto, que, por las restricciones recién explicadas, debe ser un *array* (`Object []`). Asimismo, incluye un número de versión.

Para facilitar su manejo, esta clase está organizada en dos partes: una cabecera (del tipo `CabeceraObjetoCompartido`), que incluye un nombre y una versión, y una referencia al objeto que se pretende compartir.

Además de los clásicos métodos *get* y *set*, es interesante resaltar el método que actualiza el valor del objeto que se pretende compartir, que usa el método `arraycopy` para realizar la copia, comprobando previamente que la clase es del tipo adecuado.

Interfaz `Almacen`

Esta interfaz define métodos para almacenar y recuperar objetos compartidos. Al tratarse de una aplicación distribuida, es importante minimizar el número de interacciones entre los clientes y el gestor (recuerde la segunda falacia: "*Network latency is zero*"). Por ello, se ofrecen métodos que permiten volcar

y recuperar simultáneamente un conjunto de objetos (organizados como una lista). Téngase en cuenta que precisamente este tipo de esquemas DSM híbridos están ideados para realizar todas las actualizaciones remotas de los objetos compartidos agrupadas al entrar en la sección crítica, en el caso del EC y del LRC, o al salir de la misma, en el caso del ERC.

El almacén estará implementado (clase `AlmacenImpl`) como un contenedor, seleccionado a criterio del alumno, que guarde los objetos compartidos existentes en el sistema.

A continuación, se muestra esta interfaz:

```
package dsm;
import java.util.*;
import java.rmi.*;

public interface Almacen extends Remote {
    List  leerObjetos(List  lcab)
                throws RemoteException;
    void escribirObjetos(List loc) throws RemoteException;
}
```

El comportamiento de estos métodos debe ser el siguiente:

- `leerObjetos`: Recibe como argumento una lista de las cabeceras (es decir, nombre y versión) de los objetos que se pretenden recuperar. Devuelve una lista en la que sólo aparecerán aquéllos de los objetos compartidos solicitados cuya versión sea más reciente que la que posee el cliente, retornando un `NULL` si éste posee la versión más actual de todos ellos. Por cada objeto solicitado, el tratamiento debe ser el siguiente:
  - Si la versión que posee el cliente es menor que la guardada en el almacén, se incluye la versión actual del objeto en la lista retornada.
  - Si la versión que posee el cliente es mayor o igual, no se incluye nada con respecto a ese objeto en la lista retornada.
  - Se hace lo mismo si el objeto no existe en el almacén. Se considera, por tanto, que la versión del cliente es la más actual. Tampoco se incorpora ningún tipo de información sobre ese objeto en el almacén: ya se hará cuando se escriba dicho objeto.
- `escribirObjetos`: Recibe como argumento una lista de los objetos compartidos que se desean escribir y los añade al almacén. Al tratarse de un esquema diseñado para programas cooperantes dentro de una misma aplicación, no se presupone nada sobre cuál debe ser el número de versión de cada objeto: simplemente se añaden al almacén directamente de la lista.

Como ocurría con los cerrojos remotos, tampoco en este caso se proporcionará ningún mecanismo, implícito o explícito, para eliminar un objeto compartido del almacén. Una vez creado el objeto al escribirlo por primera vez, permanecerá en el sistema todo el tiempo que esté arrancado el servidor.

Para facilitar la depuración de esta parte del proyecto práctico, se proporcionan varios clientes que usan distintos tipos de objetos compartidos:

- `TestAlmacenString`: Usa objetos compartidos de tipo `StringBuffer` (realmente, `StringBuffer [1]`).
- `TestAlmacenInts`: Comparte cinco números enteros (`Integer[5]`).
- `TestAlmacenContador`: Crea un objeto compartido asociado a un objeto de una clase definida por el usuario (`Contador [1]`). Obsérvese que la nueva clase debe implementar la interfaz `Serializable` (una interfaz sin métodos, que actúa sólo como un marcador: un *Marker Interface*) para poder ser tratada como un objeto compartido.

- `TestAlmacenListaContadores`: Usa como objeto compartido una lista de objetos de la clase `Contador`.

Es interesante resaltar que en estos dos últimos ejemplos, al usarse una clase definida por el usuario, es necesario que el gestor pueda descargarse esa clase cuando se escribe por primera vez un objeto de la misma. Para permitir que esa descarga sea automática, se ha especificado la propiedad `java.rmi.server.codebase` en la ejecución cliente de manera que haga referencia a la URL donde está almacenada esa clase (realmente, esa propiedad ha sido especificada en el *script* que facilita la ejecución de todos los programas de cliente aunque sólo se requeriría en aquéllos que manejan objetos compartidos asociados a clases definidas por el usuario).

Todos los programas de prueba reciben como argumentos la máquina y el puerto donde ejecuta el proceso `rmiregistry`.

Implementación de la clase *DSMCerrojo*

Esta clase usa los dos servicios remotos ya implementados para construir la funcionalidad DSM planteada en este proyecto. Por tanto, necesitará buscarlos en el registro de RMI para lo que deberá conocer en qué máquina y por qué puerto está dando servicio el proceso `rmiregistry`. Esta clase obtendrá esta información accediendo a las variables de entorno `SERVIDOR` y `PUERTO`, respectivamente.

La clase deberá utilizar algún tipo de contenedor, seleccionado a criterio del alumno, para guardar qué objetos compartidos están vinculados con cada cerrojo.

A continuación, se muestra la versión inicial de esta clase:

```
package dsm;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class DSMCerrojo {

    public DSMCerrojo (String nom) throws RemoteException {
    }

    public void asociar(ObjetoCompartido o) {
    }
    public void desasociar(ObjetoCompartido o) {
    }
    public boolean adquirir(boolean exc) throws RemoteException {
        return true;
    }
    public boolean liberar() throws RemoteException {
        return true;
   }
}
```

Con respecto al método `asociar`, simplemente añadirá el objeto compartido recibido como parámetro al conjunto de objetos vinculados con el cerrojo. Por su parte, el método `desasociar` realizará justamente la labor contraria.

En cuanto al método `adquirir`, hará uso del método del mismo nombre del cerrojo remoto. Una vez

conseguido el acceso a la sección crítica, utilizará el método `leerObjetos` de la interfaz `Almacen` para recuperar la versión actual de todos los objetos asociados al cerrojo, actualizando localmente todos los objetos que estaban obsoletos. El método retornará un valor falso si se produce un error al actualizar la copia local de alguno de los objetos compartidos.

Por lo que se refiere al método `liberar`, si se trata de un acceso exclusivo, debe incrementar el número de versión de todos los objetos compartidos asociados al cerrojo y usar el método `escribirObjetos` de la interfaz `Almacen` para guardar estas nuevas versiones en el almacén. Por último, hará uso del método del mismo nombre del cerrojo remoto para abandonar la sección crítica, retornando directamente el valor devuelto por dicho método.

Para probar esta última parte, se dispone del programa `ClienteDSM`, que no recibe ningún argumento (recuerde que la máquina y el puerto del `rmiregistry` los obtiene la propia clase `DSMCerrojo` mediante variables de entorno) y que usa diversos objetos compartidos.

Material de apoyo de la práctica

Al descomprimir el material de apoyo se crea el entorno de desarrollo de la práctica, que reside en el directorio: `$HOME/DATSI/SD/JavaDSM.2015/`.

Entrega de la práctica

Se realizará en la máquina `triqui`, usando el mandato:

```
entrega.sd javadsm.2015
```

Este mandato recogerá los siguientes ficheros:

- `autores` Fichero con los datos de los autores:

  ```
  DNI APELLIDOS NOMBRE MATRÍCULA
  ```

- `memoria.txt` Memoria de la práctica. En ella se deben comentar los aspectos del desarrollo de su práctica que considere más relevantes. Asimismo, puede exponer los comentarios personales que considere oportuno.
- `dsm/FabricaCerrojosImpl.java` Implementación de la interfaz `FabricaCerrojos`.
- `dsm/CerrojoImpl.java` Implementación de la interfaz `Cerrojo`.
- `dsm/AlmacenImpl.java` Implementación de la interfaz `Almacen`.
- `dsm/DSMCerrojo.java` Implementación de la clase `DSMCerrojo`.

NOTA: La práctica está diseñada para no requerir la definición de nuevas clases.