Sistema de ficheros distribuido en Java (JavaAFS)

Se trata de un proyecto práctico de desarrollo **en grupos de 2 personas** cuyo plazo de entrega termina el **21 de mayo**.

Objetivo de la práctica

La práctica consiste en desarrollar un sistema de ficheros distribuido 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 SFD basado en el modelo de carga/descarga, con una semántica de sesión, con caché en los clientes almacenada en disco e invalidación iniciada por el servidor. Es, por tanto, un sistema con unas características similares al AFS estudiado en la parte teórica de la asignatura Se recomienda, por tanto, que el alumno revise el sistema de ficheros AFS en la documentación de la asignatura antes de afrontar la práctica.

Evidentemente, dadas las limitaciones de este trabajo, el sistema a desarrollar presenta enormes simplificaciones con respecto a las funcionalidades presentes en un sistema AFS real, entre ellas:

- Hay un único servidor.
- No hay directorios. Concretamente, el servidor (*vice*; directorio `servidor`) exportará solo los ficheros que se encuentran en el directorio `AFSdir` (o sea, que si un cliente solicita el fichero `f`, el servidor enviará el fichero `AFSdir/f`). Nótese que los ficheros pueden ser de texto o binarios.
- Se trata de un sistema monousuario, no existiendo el concepto de permisos de acceso a un fichero (o sea, todos los ficheros son accesibles).
- Cada nodo "cliente" (*venus*; directorios `cliente1`, `cliente2`...) se corresponde con una JVM, pudiendo estas estar ejecutando en distintas máquinas.
- Las copias de los ficheros en el cliente se almacenan en un directorio llamado `Cache` dentro de cada nodo "cliente", y tendrán el mismo nombre que el del fichero.
- Se asume que las aplicaciones no son concurrentes y que no va a haber sesiones de escritura simultáneas sobre un mismo fichero aunque sí puede haber múltiples sesiones de lectura mientras que se produce una de escritura.
- A las aplicaciones se les ofrece un API similar a la clase `RandomAccessFile` de Java, pero limitándose, para simplificar el trabajo, a las operaciones de lectura (`read`) y escritura (`write`) usando vectores de bytes, así como de modificación de la posición del puntero (`seek`) y de cambio de tamaño de un fichero (`setLength`). Se limita también los modos de apertura posibles de un fichero a solo dos, manteniendo la misma semántica que en `RandomAccessFile`:
  - Modo `r`: si el fichero existe se abre en modo lectura; en caso contrario, se genera la excepción `FileNotFoundException`.
  - Modo `rw`: si el fichero existe se abre en modo lectura/escritura; en caso contrario, también se crea previamente.
- Asimismo, se usa la clase `RandomAccessFile` para el acceso interno, tanto local como remoto, a los datos del fichero.

En cuanto a la tecnología de comunicación usada en la práctica, 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 incluyen, a continuación, dos fragmentos de código que permiten apreciar las diferencias entre el API de acceso de lectura/escritura a un fichero local usando la clase `RandomAccessFile`, en la que se inspira el API planteado, y el correspondiente al acceso remoto utilizando el servicio que se pretende desarrollar:

```
// acceso local
try {
    RandomAccessFile f = new RandomAccessFile("fich", "rw");
    byte[] b = new byte[1024];
    leido = f.read(b);
    f.seek(0);
    f.write(b);
    f.setLength(512);
    f.close();
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// acceso remoto (en negrilla aparecen los cambios respecto a un acceso local)
try {
    // iniciación de la parte cliente
    Venus venus = new Venus();

    // apertura de un fichero;
    // el modo de apertura solo puede ser "r" o "rw"
    // y tiene el mismo comportamiento que en RandomAccessFile.
    // Si el fichero no está en la caché, se descarga del servidor
    // y se almacena en el directorio Cache.
    // Finalmente, se abre el fichero en Cache como un RandomAccessFile.
    VenusFile f = new VenusFile(venus, "fich", "rw");

    // resto de las operaciones igual que en local;
    // de hecho, se realizan sobre la copia local
    byte[] b = new byte[1024];
    leido = f.read(b);
    f.seek(0);
    f.write(b);
    f.setLength(512);

    // si el fichero se ha modificado, se vuelca al servidor
    f.close();
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
catch (Exception e) {
    e.printStackTrace();
}
```

Para afrontar el trabajo de manera progresiva, se propone un desarrollo incremental en varias fases. Por cada fase, se indicará qué funcionalidad desarrollar como parte de la misma y qué pruebas concretas realizar para verificar el comportamiento correcto del código desarrollado.

1. Acceso de lectura a un fichero remoto sin tener en cuenta aspectos de coherencia (valoración de 3 puntos).
2. Acceso de escritura a un fichero remoto sin tener en cuenta aspectos de coherencia (valoración de 3 puntos).
3. Incorporación de un modelo de coherencia asumiendo que solo puede haber una sesión de escritura, con múltiples de lectura, en cada momento (valoración de 4 puntos).

Arquitectura del software del sistema

Antes de pasar a presentar cada una de las fases, se especifica en esta sección qué distintos componentes hay en este sistema.

En primer lugar, hay que resaltar que la práctica está diseñada para no permitir la definición de nuevas clases (a no ser que se trate de clases anidadas), estando todas ya presentes, aunque mayoritariamente vacías, en el material de apoyo.

No todas ellas serán necesarias en las primeras fases de la práctica, como se irá explicando en esta misma sección y a lo largo del documento. Por tanto, no es necesario que entienda el objetivo de cada clase en este punto, ya que se irá descubriendo a lo largo de las sucesivas fases.

El software de la práctica está organizado en tres directorios: `cliente` (realmente, hay varios directorios cliente para asegurarse de que cada JVM tiene su propia caché si se ejecutan varios "nodos cliente" en la misma máquina real), `servidor` y `afs`. Empecemos por este último, que contiene las clases que proporcionan la funcionalidad del servicio a desarrollar, que estarán incluidas en el paquete `afs`. A continuación, se comenta brevemente el objetivo de cada una, que será posteriormente explicado en detalle en las secciones del documento dedicadas a presentar progresivamente la funcionalidad requerida.

- `Venus`: Clase de cliente que proporciona acceso al servicio realizando la iniciación de la parte cliente. La aplicación deberá instanciar un objeto de esta clase antes de interaccionar con el servicio desarrollado. Esta clase se requerirá desde la primera fase de la práctica.
- `VenusFile`: Clase de cliente que proporciona el API del servicio AFS, existiendo un objeto de esta clase por cada uno de los ficheros a los que está accediendo la aplicación. La aplicación deberá instanciar un objeto de esta clase para acceder a un fichero. Esta clase se requerirá desde la primera fase de la práctica.
- `Vice y ViceImpl`: Interfaz remota, y clase que la implementa, que ofrece en el servidor el servicio AFS proporcionando métodos remotos para iniciar la carga y descarga de ficheros. Al arrancar el servicio, el servidor, que ya está programado, instancia y registra un objeto de esta clase. Estas clases se requerirán desde la primera fase de la práctica.
- `ViceReader y ViceReaderImpl`: Interfaz remota, y clase que la implementa, que ofrece en el servidor los servicios para completar la descarga de un fichero. Se creará una clase de este tipo en el servidor cada vez que se produce una operación de descarga. Estas clases se requerirán desde la primera fase de la práctica.
- `ViceWriter y ViceWriterImpl`: Interfaz remota, y clase que la implementa, que ofrece en el servidor los servicios para completar la carga de un fichero. Se creará una clase de este tipo en el servidor cada vez que se produce una operación de carga. Estas clases se requerirán a partir de la segunda fase de la práctica.
- `VenusCB y VenusCBImpl`: Interfaz remota, y clase que la implementa, que ofrece en el cliente el servicio de *callback* requerido para implementar el protocolo de coherencia. Se instanciará un objeto de esta clase en la parte cliente al iniciarse la interacción con el servicio. Estas clases se requerirán solo en la tercera fase de la práctica.
- `LockManager`: Clase de servidor que gestiona cerrojos de lectura/escritura para sincronizar el acceso a un fichero. Ya está implementada y **no debe modificarse**. Se requerirá en la tercera fase de la práctica.

Con respecto a los directorios `cliente`, en los mismos se encuentra la clase `Test`, que es un programa interactivo que sirve para probar la funcionalidad de la práctica, permitiendo que el usuario pueda realizar operaciones de lectura, escritura, posicionamiento y cambio de tamaño sobre ficheros. Asimismo, este programa recibirá como variables de entorno la siguiente información:

- en qué máquina y por qué puerto está dando servicio el proceso `rmiregistry` en las variables `REGISTRY_HOST` y `REGISTRY_PORT`, respectivamente.
- el tamaño de bloque usado en las transferencias entre los clientes y el servidor: `BLOCKSIZE`.

Mediante el uso de enlaces simbólicos, los directorios de cliente comparten todos los ficheros (excepto, evidentemente, el directorio `Cache`), no habiendo que realizar ningún desarrollo de código en los mismos a no

ser que uno quiera preparar sus propios programas de prueba.

En cuanto al directorio `servidor`, donde tampoco hay que hacer ningún desarrollo, este incluye la clase `ServidorAFS` que inicia el servicio dándole de alta en el `rmiregistry` con el nombre `AFS` (instancia un objeto de la clase `ViceImpl` y lo registra). Este programa recibe como argumento el número de puerto por el que escucha el proceso `rmiregistry` previamente activado. Este directorio contiene un subdirectorio denominado `AFSDir` que será donde se ubiquen los ficheros del servidor.

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 estos pueden residir en distintas máquinas.

Ejecución de la práctica

Aunque para toda la gestión del ciclo de desarrollo del código de la práctica se puede usar el IDE que se considere oportuno, para aquellos que prefieran no utilizar una herramienta de este tipo, se proporcionan una serie de *scripts* que permiten realizar toda la labor requerida. En esta sección, se explica cómo trabajar con estos *scripts*.

Para probar la práctica, debería, en primer lugar, compilar todo el código desarrollado que se encuentra en el directorio, y paquete, `afs`, generando los ficheros JAR requeridos por el cliente y el servidor.

```
cd afs
./compila_y_construye_JARS.sh
```

A continuación, hay que compilar y ejecutar el servidor, activando previamente `rmiregistry`.

```
cd servidor
./compila_servidor.sh
./arranca_rmiregistry 12345 &
./ejecuta_servidor.sh 12345
```

Por último, hay que compilar y ejecutar el cliente de prueba.

```
cd cliente1
./compila_test.sh
export REGISTRY_HOST=triqui3.fi.upm.es
export REGISTRY_PORT=12345
export BLOCKSIZE=... # el tamaño que considere oportuno
./ejecuta_test.sh
```

Nótese que el servidor y el cliente pueden ejecutarse en distintas máquinas. Además, tenga en cuenta que, si ejecuta varios clientes en la misma máquina, debería hacerlo en diferente directorio de cliente (`cliente1`, `cliente2`...).

Fase 1: lectura de un fichero sin tener en cuenta aspectos de coherencia

El objetivo de esta fase es permitir el acceso de lectura a ficheros remotos almacenados en el servidor (ubicados en el subdirectorio `AFSDir`).

Antes de pasar a describir la funcionalidad de esta fase, se considera conveniente hacer dos reflexiones previas:

- Dado que los ficheros pueden tener un tamaño considerable, no es factible implementar un método remoto para descargar un fichero que retornara el contenido del mismo, puesto que eso requeriría que tanto en el cliente como en el servidor se almacenara en memoria todo el fichero. Nótese que esa misma circunstancia

se produce en la operación de carga. Será necesario, por tanto, ir enviando el contenido del fichero por bloques.
- Obsérvese que el modelo de RPC/RMI síncrono de Java RMI no es el idóneo para este tipo de transferencias puesto que requiere una interacción cliente/servidor (la invocación de un método remoto que implica una petición y una respuesta) por cada bloque involucrado.

Fase 1: funcionalidad del servidor

Para realizar la descarga, se plantea usar un esquema de fábrica de referencias remotas a objetos (véase en la guía de Java RMI la sección dedicada a este esquema), tal que se cree un objeto remoto para encapsular cada sesión de descarga de un fichero. Con este esquema, el servicio `Vice` ofrece una operación (`download`) para iniciar la descarga de un fichero que genera una referencia remota de tipo `ViceReader` que ofrece métodos remotos para ir descargando el fichero bloque a bloque.

En `ViceReaderImpl` se usará la clase `RandomAccessFile` para los accesos al fichero real. De hecho, cada objeto `ViceReaderImpl` almacenará internamente un objeto `RandomAccessFile` que corresponderá a la sesión de acceso de lectura del fichero.

A continuación, se detallan los cambios a realizar:

- En el método `download` de la clase `ViceImpl` se creará una instancia de la clase `ViceReaderImpl` que se retornará como resultado del método. El cliente usará los métodos remotos de esta clase para completar la descarga.
- En cuanto a la clase `ViceReaderImpl`, en el constructor se debería instanciar un objeto de la clase `RandomAccessFile` asociado al fichero en el servidor (es decir, *abrir el fichero*) y en los métodos remotos de acceso al fichero se debería invocar el método correspondiente de `RandomAccessFile`.
- Como parte del desarrollo, tendrá que realizar los `imports` requeridos (por ejemplo, de `java.io`), así como la propagación de las excepciones (`throws`) pertinentes. Así, por ejemplo, tanto el constructor de `ViceReaderImpl` como el método `download` que activa este constructor deben propagar la excepción `FileNotFoundException` que generará la clase `RandomAccessFile` si el fichero no existe, para que, de esta forma, el cliente sea notificado de esta circunstancia.

Para terminar esta sección, se considera conveniente realizar una aclaración sobre el método remoto `read`. De forma intutiva, parecería más razonable usar una declaración similar a la del método del mismo nombre de la clase `RandomAccessFile` permitiendo de esta forma una implementación directa del mismo:

```
public int read(byte[] b) throws ...
    return f.read(b); // siendo f un objeto de tipo RandomAccessFile
}
```

Sin embargo, este modo de operación no es correcto en Java RMI ya que este método remoto devolvería la información leída en un parámetro pasado por referencia y Java RMI no permite ese tipo de paso de información (véase la guía sobre la programación en Java RMI para profundizar sobre este tema). Es por ello que en la definición propuesta la información leída se devuelve como retorno del método remoto.

Fase 1: funcionalidad del cliente

Con respecto a la parte cliente, en esta fase entran en juego dos clases:

- `Venus`: esta clase corresponde a la iniciación de la interacción de una aplicación con el servicio proporcionado y encapsula todo lo que tiene que ver con la operación de búsqueda del servicio `AFS` en el `rmiregistry` (es decir, el acceso a las tres variables de entorno previamente descritas, `REGISTRY_HOST`, `REGISTRY_PORT` y `BLOCKSIZE`, y la propia operación de *lookup*) para evitar de esta forma que se tenga que realizar este proceso por cada fichero accedido por la aplicación. Esta clase ofrecerá esta información a la clase `VenusFile`, ya sea a través de atributos públicos o métodos de tipo *getter*.

- `VenusFile`: habrá un objeto de este tipo por cada fichero abierto por la aplicación. El constructor deberá comprobar si el fichero existe en `Cache` y, en caso negativo, deberá proceder a su descarga del servidor usando los métodos remotos (primero, usará la referencia remota obtenida por `Venus` para iniciar la descarga y, a continuación, utilizará los métodos remotos del objeto `ViceReader` creado en el servidor para completarla descargando el fichero bloque a bloque usando el tamaño especificado en la variable de entorno correspondiente) y almacenando el contenido en una copia local en el directorio `Cache`. El constructor debe dejar abierto el fichero local, existiera este desde el principio o no. Con respecto a los métodos de esta clase, trabajarán directamente sobre el fichero local.

Fase 1: pruebas

En esta sección se comentan qué pruebas se pueden llevar a cabo para verificar la funcionalidad pedida.

1. Arranque el programa `Test` y abra un fichero no existente en el servidor. El programa `Test` debería imprimir el mensaje `Fichero no existente` lo que significaría que ha recibido la excepción `FileNotFoundException` que se ha propagado desde el servidor hasta la aplicación.
2. Arranque el programa `Test` y abra un fichero que previamente ha creado en el servidor de forma externa a la práctica y que ocupe más de un bloque teniendo un tamaño que no sea múltiplo del tamaño de bloque. Use las operaciones `read` y `seek` del programa de prueba para comprobar que el contenido es correcto.
3. Siguiendo con la prueba anterior, queremos probar ahora que cuando el fichero existe en `Cache` se usa la copia local. Para verificarlo, cierre el fichero, modifique externamente alguna parte del contenido de la copia del fichero en `Cache` y compruebe que al volver a abrirlo se accede al contenido modificado y no al servidor.
4. Como última prueba, compruebe el comportamiento del código desarrollado si se lee un fichero existente pero vacío.

Fase 2: Escritura en un fichero sin tener en cuenta aspectos de coherencia

Una operación de escritura en un fichero puede implicar una operación de descarga, al abrirlo si no está en `Cache`, y una de carga, al cerrarlo si se ha escrito o se ha cambiado su tamaño durante la sesión de acceso.

Aunque la operación de carga ya se ha implementado en la fase previa, habrá que reajustarla dado el diferente comportamiento de la operación de apertura de un fichero dependiendo del modo de apertura en caso de que este no exista previamente: en una sesión de lectura (modo `r`) se produce la excepción `FileNotFoundException`, mientras que en una de escritura (modo `rw`) el fichero se crea y se abre normalmente.

Fase 2: funcionalidad del servidor

Revisemos los cambios requeridos en las distintas clases:

- En `ViceImpl` habrá que programar el método `upload` para que cree una instancia de la clase `ViceWriterImpl` que se retornará como resultado del método. El cliente usará los métodos remotos de esta clase para completar la carga.
- Con respecto a la clase `ViceReaderImpl`, habrá que reajustar su constructor para tener en cuenta el modo de apertura tal como se ha explicado previamente.
- En cuanto a la clase `ViceWriterImpl`, en el constructor se debería instanciar un objeto de la clase `RandomAccessFile` asociado al fichero en el servidor y en los métodos remotos de acceso al fichero se debería invocar el método correspondiente de `RandomAccessFile`.
- Como parte del desarrollo, tendrá que realizar los `imports` y la propagación de las excepciones requeridos.

Fase 2: funcionalidad del cliente

Con respecto a la parte cliente, los cambios requeridos serían:

- `VenusFile`: habrá que controlar si se modifica o se cambia de tamaño el fichero durante la sesión de acceso de manera que, si ocurre esta circunstancia, en la operación de cierre se realice la carga del fichero al sevidor: primero, usará la referencia remota obtenida por `Venus` para iniciar la carga y, a continuación, utilizará los métodos remotos del objeto `ViceWriter` creado en el servidor para completarla enviando el contendo del fichero bloque a bloque.

Fase 2: pruebas

En primer lugar, se debería comprobar que las pruebas de la fase anterior siguen funcionando correctamente. Estas serían las pruebas propuestas para esta fase:

1. Pruebe a abrir en modo lectura un fichero existente y, a continuación, escriba en el mismo. El programa de prueba debería imprimir que se ha producido una excepción de E/S y continuar operando correctamente.
2. Repita la prueba anterior cambiando la longitud del fichero.
3. Abra un fichero no existente en modo escritura, escriba en el mismo y ciérrelo. Compruebe que tanto el contenido almacenado en `Cache` como en `AFSDir` es correcto.
4. Abra un fichero existente en modo escritura, escriba en el mismo y ciérrelo. Compruebe que tanto el contenido almacenado en `Cache` como en `AFSDir` es correcto.
5. Repita la prueba anterior cambiando solo la longitud del fichero.
6. Pruebe el uso de la caché haciendo dos sesiones sucesivas de escritura sobre un fichero existente.

Fase 3: Implementación del modelo de coherencia

En esta fase se aborda el protocolo de coherencia de AFS pero solo para un escenario simplificado donde se asume que en cada momento solo hay una sesión de escritura activa en cada fichero, que, eso sí, podría ejecutarse concurrente con sesiones de lectura simultáneas sobre ese mismo fichero.

Antes de pasar a describir la funcionalidad concreta de esta fase, hay que revisar dos aspectos relacionados con los problemas de coherencia debido a accesos simultáneos.

Sincronización de cargas y descargas

Tal como se han implementado las fases previas se pueden producir en paralelo cargas y descargas de un mismo fichero, lo que puede causar problemas de coherencia (el lector puede realizar una descarga parcial de un fichero porque el escritor todavía no ha completado la carga).

La solución más directa es usar cerrojos de lectura/escritura sobre un fichero mientras se está descargando (cerrojo de lectura) o cargando (cerrojo de escritura), permitiendo, de esta forma, un modelo de múltiples descargas pero solo una carga. Habría que solicitar un bloqueo de un cerrojo de lectura al principio de la descarga para liberarlo al final de la misma y uno de escritura siguiendo la misma pauta para la carga.

Java ofrece esta funcionalidad dentro de la clase `FileLock`, que se basa en el mecanismo nativo equivalente del sistema operativo subyacente. Sin embargo, este mecanismo no es aplicable al problema de sincronización planteado, puesto que los accesos que se requiere sincronizar corresponden a *threads* del mismo proceso (Java RMI va asignando *threads* para que procesen concurrentemente las peticiones que se van recibiendo), mientras que este mecanismo es solo válido para sincronizar procesos independientes (nótese que sucede lo mismo con los cerrojos de ficheros convencionales de Linux, aunque en este sistema operativo existe una variedad que sí sería válida: los *Open file description locks*, que quedan fuera de este proyecto práctico).

Para solventar esta limitación, vamos a usar un mecanismo de sincronización no vinculado con los ficheros, como son los *mutex* de lectura y escritura que ofrece Java. En el código de apoyo de la práctica se ofrece una clase denominada `LockManager`, que será instanciada por `ViceImpl`, y que ofrece esta funcionalidad permitiendo asociar un nombre (el fichero) con un mutex de lectura/escritura:

- Método `bind`: retorna un mutex de lectura/escritura (`ReentrantReadWriteLock`) asociado a ese nombre (a ese fichero): si ya existe, solo lo devuelve; en caso contrario, lo crea previamente. Recuerde que la clase `ReentrantReadWriteLock`, que corresponde al mutex retornado, ofrece los métodos:
  - `readLock().lock()`: solicitud de un bloqueo de lectura.
  - `readLock().unlock()`: eliminación de un bloqueo de lectura.
  - `writeLock().lock()`: solicitud de un bloqueo de escritura.
  - `writeLock().unlock()`: eliminación de un bloqueo de escritura.
- Método `unbind`: indica que se ha dejado de usar por parte de un *thread* el mutex de lectura/escritura asociado a ese nombre.

Protocolo de coherencia

Recuerde que el protocolo de coherencia de AFS conlleva que cuando se carga una nueva copia de un fichero al servidor, que ocurrirá al final de una sesión de escritura donde la aplicación ha modificado el contenido del fichero o su longitud, hay que invalidar todas las demás copias de las cachés de los clientes, tengan estos el fichero abierto (nótese que en esta fase si lo tienen abierto debe ser para una sesión de lectura) o no.

En la práctica, la invalidación de la copia del fichero en la caché se realizará directamente borrando el fichero del directorio `Cache`, lo que hará que posteriores accesos a ese fichero no lo encuentren en la caché y tengan que descargarlo del servidor. Recuerde que, dado como implementa el sistema operativo el acceso a los ficheros, aunque se borre el fichero de `Cache`, las sesiones de acceso a ese fichero que ya estén activas localmente continuarán sin incidencias.

Para implementar el mecanismo de invalidaciones se usará un esquema de tipo *callback* (véase la guía sobre la programación en Java RMI para profundizar sobre este tema), que permite a un servidor invocar un método remoto de un cliente. El modo de operación será el siguiente:

- El objeto de la clase `Venus` instanciará un objeto de la clase `VenusCBImpl`, que implementará el mecanismo de *callback*. Este objeto simplemente implementará la invalidación (método `invalidate`) borrando el fichero de la caché.
- Las invocaciones en `VenusFile` a los métodos de carga y descarga enviarán al servidor una referencia a ese objeto de tipo *callback*.
- La clase `ViceImpl` en el servidor gestionará una estructura de datos que asocie cada nombre de fichero con los objetos de *callback* de los clientes que tienen copias del fichero en su caché.
- Cada vez que un cliente descarga un fichero se añade su *callback* a la estructura de datos.
- Cada vez que un cliente carga un fichero se invoca el método de invalidación de todos los demás clientes (nótese la existencia del método `equals` para comprobar la igualdad entre dos objetos de tipo *callback*) y se eliminan de la lista, puesto que esos clientes ya no tienen copia.

Fase 3: funcionalidad del servidor

Esta funcionalidad requiere los siguientes cambios en las clases de servidor:

- En `ViceImpl` (y en `Vice`) añadir la referencia al *callback* como parámetro de los métodos `download` y `upload`. Asimismo, dado que va a ser necesario acceder a funcionalidad de esta clase desde `ViceReaderImpl` y `ViceWriterImpl`, sería conveniente pasar una referencia de `ViceImpl` (`this`) en los constructores de `ViceReaderImpl` y `ViceWriterImpl` (dependiendo de cómo se implemente la funcionalidad de esta fase, también podría ser conveniente pasar una referencia al *callback*).

- Asimismo, en `ViceImpl` habrá que instanciar el `LockManager` y definir una estructura que relacione nombres de fichero con listas de `callbacks`, así como los métodos que se requieran para la gestión de esta estructura. Nótese que, dado que se pueden producir accesos concurrentes a esta estructura de datos (por ejemplo, descargas en paralelo de dos ficheros distintos), habrá que asegurar la coherencia de esta estructura usando, si es necesario, métodos `synchronized` para la gestión de la misma.
- En `ViceReaderImpl`, que tendrá nuevos parámetros en su constructor, habrá que pedir al `LockManager` un cerrojo asociado al fichero solicitando un bloqueo de lectura e incorporar el `callback` en la lista asociada a ese fichero (esta operación también se podría realizar en el propio método `download`). En el método `close` habrá que liberar el cerrojo e informar de que ya no se requiere el uso de ese cerrojo.
- Los cambios en `ViceWriterImpl` serán similares exceptuando dos aspectos. Por un lado, las operaciones sobre el cerrojo son de escritura. Por otro lado, cuando se completa la carga, pero antes de liberar el cerrojo, hay que invocar al método `invalidate` de cada objeto de tipo *callback* asociado a este fichero, exceptuando únicamente el correspondiente al cliente que está realizando la descarga (no queremos invalidar esa copia porque corresponde precisamente con la nueva versión del fichero).

Fase 3: funcionalidad del cliente

En cuanto a la parte cliente, requiere los siguientes cambios:

- En `Venus` hay que instanciar el *callback*.
- En `VenusFile` se tiene que pasar el *callback* como parámetro en los métodos que inician la carga y la descarga.
- En `VenusCB` hay que implementar el método de invalidación que borrará el fichero de `Cache`.

Fase 3: pruebas

Para probar el protocolo de coherencia puede ejecutar la siguiente secuencia de procesos:

1. En `cliente1` abra un fichero no existente en modo `rw`, escríbalo y ciérrelo.
2. En `cliente2` abra ese fichero en modo `r`, léalo y ciérrelo.
3. En `cliente3` abra ese fichero en modo `r` y lea parte del mismo pero no lo cierre.
4. En `cliente4` abra ese fichero en modo `rw`, y modifique parte del mismo. Antes de cerrarlo, compruebe el contenido de los directorios `Cache` de los cuatro clientes y del directorio `AFSDir` del servidor. Las copias en los tres primeros clientes deben ser iguales que la del servidor puesto que todavía no se ha cerrado el fichero en el cuarto cliente.
5. Cierre el fichero en el cuarto cliente. En ese momento, tienen que haber desaparecido las copias de los tres primeros clientes y la copia del cuarto debe ser igual que la del servidor.
6. Solicite una nueva lectura en `cliente3` y pruebe a leer con el descriptor obtenido en la primera apertura que todavía no se cerró y con el nuevo. Verifique que con el primero se accede al contenido original mientras que con el segundo al nuevo.

Material de apoyo de la práctica

El material de apoyo de la práctica se encuentra en este enlace.

Al descomprimir el material de apoyo se crea el entorno de desarrollo de la práctica, que reside en el directorio: `$HOME/DATSI/SD/javaAFS.2021/`.

Entrega de la práctica

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

```
entrega.sd javaAFS.2021
```

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 pueden comentar los aspectos del desarrollo de su práctica que considere más relevantes. Asimismo, puede exponer los comentarios personales que considere oportuno.
- `afs/Venus.java`
- `afs/VenusFile.java`
- `afs/VenusCB.java`
- `afs/VenusCBImpl.java`
- `afs/Vice.java`
- `afs/ViceImpl.java`
- `afs/ViceReader.java`
- `afs/ViceReaderImpl.java`
- `afs/ViceWriter.java`
- `afs/ViceWriterImpl.java`