Sistema de ficheros paralelo basado en GFS (miniGFS)

Se trata de un proyecto práctico de desarrollo **en grupos de 2 personas** cuyo plazo de entrega termina el **25 de mayo**.

Objetivo de la práctica

La práctica consiste en desarrollar un sistema de ficheros paralelo basado en el Google File System que permita llegar 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. Evidentemente, dadas las limitaciones de este trabajo, el sistema a desarrollar presenta enormes simplificaciones con respecto a las funcionalidades proporcionadas por un sistema GFS real. Una de las restricciones que se establecen para simplificar el sistema es que todas las operaciones sobre el sistema de ficheros manejarán datos con tamaños múltiplos del tamaño de un *chunk*.

En cuanto a la tecnología de comunicación usada en la práctica, se ha elegido Java RMI, explicada en la parte teórica de la asignatura (guía sobre la programación en Java RMI).

Para desarrollar la práctica de forma incremental, se plantea una serie de fases. Antes de describirlas, vamos a repasar el modo de operación del GFS y explicar cómo está organizado el software de la práctica.

Arquitectura del sistema

Aunque el modo de operación de GFS ha sido estudiado en detalle en la asignatura y puede profundizar sobre el mismo revisando el artículo que presentó a la comunidad esta tecnología, en esta sección se realiza un breve recordatorio del modo de operación de este sistema:

- Existen tres tipos de nodos: el maestro, que almacena todos los metadatos de los ficheros, los nodos de almacenamiento (*chunk servers*; en esta práctica les denominaremos *data nodes* usando la terminología del *Hadoop filesystem*, una versión de código abierto del GFS), que guardan los datos de los ficheros, y los clientes, donde las aplicaciones usan una biblioteca que les proporciona acceso a la funcionalidad del GFS.
- Los ficheros, que serán de gran tamaño, se dividen en fragmentos (*chunks*, con un tamaño por defecto de 64MiB), que se almacenan de forma replicada en los nodos de datos.
- El modo de operación del acceso a un fichero, a grandes rasgos, sería el siguiente:
  - Cuando una aplicación realiza una operación de lectura o escritura, la biblioteca de cliente determina qué *chunks* del fichero están involucrados en esa operación y contacta con el maestro indicándole el nombre del fichero y los números de los *chunks* afectados.
  - Si se trata de una escritura sobre un *chunk* que no tiene espacio asignado, el maestro genera un identificador único para ese *chunk* así como asigna tantos nodos de datos como establezca el grado de replicación configurado identificando uno de ellos como el primario para ese *chunk* y retornando toda esa información al cliente. Nótese que si se trata de una lectura o de una escritura que sobrescribe el *chunk*, ya tendrá asignados un identificador y nodos de datos enviando simplemente toda esa información al cliente.
  - A continuación, si se trata de una operación de escritura, el cliente envía al nodo de datos primario el identificador del *chunk* y su contenido. Ese nodo propaga la operación a los secundarios de forma sincronizada, de manera que si se producen dos escrituras concurrentes sobre un mismo *chunk*, se asegura de que todas las copias terminen con el mismo contenido. El nodo de datos escribe el *chunk* en un fichero convencional cuyo nombre corresponde al identificador del *chunk*. Terminada la operación en todos los nodos, el primario informa de esta circunstancia al cliente dándose por completada la operación.
  - En caso de una lectura, el cliente contacta con cualquiera de los nodos de datos que tienen copias enviándole el identificador del *chunk* y el nodo de datos lee el fichero correspondiente y se lo envía al cliente.

Arquitectura del software del sistema

Antes de describir cómo están organizadas las clases del proyecto, hay que resaltar que la práctica está diseñada para no permitir la definición de nuevas clases, 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 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 cuatro directorios: uno por cada uno de los tres tipos de nodos que existen en este sistema (maestro, nodo de datos y cliente) y un cuarto que contiene las interfaces de los distintos servicios que estarán disponibles en todos los nodos.

Directorio `master_node`

Contiene la funcionalidad del nodo maestro que se ha dividido en dos partes: la gestión de la metainformación de los ficheros, que se ha integrado en el paquete `master`, y la administración de los nodos de almacenamiento, que se ha incluido en el paquete `manager`. A continuación se revisan las clases de ambos paquetes:

- `master/MasterSrv`: Clase ya completada (**no se debe modificar**) que instancia y registra los servicios del maestro. Incluye la función `main` que recibe como argumentos el puerto del *registry* y el número de réplicas que usará el sistema (factor de replicación).
- `master/MasterImpl`: Clase que implementa la interfaz remota `Master` que ofrece el método `lookup`: devuelve un objeto que implementa la interfaz remota `File` que permite acceder a la metainformación de ese fichero.
- `master/FileImpl`: Clase que implementa la interfaz remota `File` que ofrece métodos para obtener el número de *chunks* que ocupa un fichero (`getNumberOfChunks`) y la información de ubicación de los *chunks* del fichero especificados (`getChunkDescriptors`), así como un método para reservar información de ubicación para los *chunks* especificados (`allocateChunkDescriptors`).
- `master/ChunkImpl`: Clase ya completada (**no se debe modificar**) que implementa la interfaz no remota `Chunk` que mantiene la información de ubicación de un *chunk*: su UUID y la lista de nodos de datos asignados para ese *chunk*, siendo el primario el primero de la lista.
- `manager/ManagerImpl`: Clase que implementa la interfaz remota `Manager` que ofrece el método `addDataNode` que permite darse de alta a un nodo de datos. Proporciona también el método remoto `getDataNodes` que devuelve todos los nodos de datos dados de alta en el sistema para facilitar la depuración durante el desarrollo de la práctica. En cuanto al método local `selectDataNode`, selecciona un nodo de datos para almacenar una copia de un *chunk*.

Directorio `data_node`

Contiene la funcionalidad de un nodo datos que se ha incluido en el paquete `datanode`. Incluye las siguientes clases:

- `datanode/DataNodeSrv`: Clase ya completada (**no se debe modificar**) que instancia el servicio de un nodo de datos y realiza la búsqueda en el *registry* del servicio `Manager` del maestro. Incluye la función `main` que recibe como argumentos la máquina y el puerto del *registry*, así como el nombre del nodo de datos. Nótese que se crea un directorio con el nombre del nodo de datos en el directorio `data_node/bin` para almacenar los ficheros que contienen los *chunks* guardados en ese nodo de datos.
- `datanode/DataNodeImpl`: Clase que implementa la interfaz remota `DataNode` que ofrece métodos para obtener el nombre del nodo de datos (`getName`), para leer un fichero que contiene un *chunk* almacenado en ese nodo (`readChunk`) y para escribirlo (`writeChunk`). Este último método propaga la escritura a otros nodos de datos si recibe como argumento una lista de nodos de datos que no esté vacía.
- `datanode/LockManager`: Clase ya completada (**no se debe modificar**) que implementa cerrojos para secuenciar las escrituras en la copia primaria. Ofrece un método `openLock` para obtener acceso a un cerrojo asociado a un nombre (clase interna `LockManager/Lock`) que proporciona operaciones de `lock` y `unlock`.

Directorio `client_node`

Corresponde a un nodo cliente, donde se ejecutan las aplicaciones que usan el sistema de ficheros implementado. En el paquete `tests` se incluye un conjunto de programas que sirven para ir probando la funcionalidad del sistema de ficheros y se irán describiendo según se vayan presentando las distintas fases en el desarrollo de la práctica. En cuanto al paquete `client`, contiene la clase que proporciona a las aplicaciones la funcionalidad para acceder al sistema de ficheros desarrollado:

- `client/GFSFile`: Clase de cliente que proporciona los métodos para acceder a los ficheros. Corresponde al API ofrecida a las aplicaciones, que es similar a la proporcionada por la clase estándar `RandomAccessFile`. Esta clase debe contactar con el *registry* para poder acceder a los servicios del maestro y requiere que se definan las siguientes variables de entorno:
  - en qué máquina y por qué puerto está dando servicio el proceso `rmiregistry` en las variables `REGISTRY_HOST` y `REGISTRY_PORT`, respectivamente.
  - el tamaño del *chunk*: `CHUNKSIZE`.

A continuación, se muestra el API ofrecida a las aplicaciones a través de la clase `GFSFile`.

```
// constructor: habilita acceso a un fichero
GFSFile(String fileName);

// establece la posición de acceso al fichero (método local)
public void seek(int off);

// obtiene la posición de acceso al fichero (método local)
public int getFilePointer();

// obtiene la longitud del fichero en bytes
public int length() throws RemoteException;

// lee de la posición actual del fichero la cantidad de datos pedida;
// devuelve cantidad realmente leída, 0 si EOF o error;
// actualiza la posición de acceso al fichero
public int read(byte [] buf) throws RemoteException;

// escribe en la posición actual del fichero los datos especificados;
// devuelve falso si se ha producido un error;
// actualiza la posición de acceso al fichero
public boolean write(byte [] buf) throws RemoteException;
```

Y un programa de ejemplo que muestra el uso de este API:

```
// Ejemplo que copia un fichero local al miniGFS.
// Por las restricciones del miniGFS, el fichero copiado
// tendrá un tamaño múltiplo del chunkSize
package tests;
import java.io.FileInputStream;
import java.io.IOException;
import java.rmi.*;
import client.GFSFile;
class CopyFileToGFS {
    static public void main(String [] args) {
        if (args.length!=2) {
            System.err.println("Usage: CopyFileToGFS file chunkSize");
            return;
        }
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());
        try {
            int chunkSize = Integer.parseInt(args[1]);
            FileInputStream f1 = new FileInputStream(args[0]);
            GFSFile f2 = new GFSFile(args[0]);
            byte [] buf = new byte[chunkSize];
            while (f1.read(buf)>0)
                f2.write(buf);
        } catch (RemoteException e) {
```

```
            System.err.println("error accediendo a fichero destino");
        } catch (IOException e) {
            System.err.println("error accediendo a fichero origen");
        }
    }
}
```

Directorio `interfaces`

Contiene en un paquete del mismo nombre las interfaces existentes en el sistema tal como se han ido explicando en los apartados anteriores:

- `interfaces/Master`.
- `interfaces/Manager`.
- `interfaces/DataNode`.
- `interfaces/File`.
- `interfaces/Chunk` (es la única interfaz que no es remota).

Estas interfaces **no se pueden modificar**, ya están compiladas y empaquetadas en un JAR en el material de apoyo y deben estar disponibles en todos los nodos. Para reducir el espacio ocupado por los ficheros de la práctica, en vez de una copia del JAR, se han establecido enlaces simbólicos al JAR en todos los nodos.

Compilación y 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*.

Como primer paso, se debería descargar y desempaquetar el código de la práctica:

```
wget https://laurel.datsi.fi.upm.es/~ssoo/SD.dir/practicas/miniGFS.tgz
tar xvfz miniGFS.tgz
cd DATSI/SD/miniGFS.2022
```

Para compilar la práctica, existe un *script* denominado `./compila.sh` en cada uno de los 4 directorios de la práctica. También se dispone del *script* `./compila_todo.sh` en el directorio raíz de la práctica que va invocando los *scripts* de compilación de cada directorio.

En cuanto a la ejecución, se plantea un ejemplo con tres nodos de datos que se ejecuta en cinco máquinas.

En primer lugar, arrancamos el *registry* y el maestro:

```
triqui1: cd master_node
triqui1: ./arranca_rmiregistry 34567 &
triqui1: ./ejecuta_master.sh 34567 2 # 2 réplicas
```

A continuación, arrancamos los tres nodos de datos (normalmente, el número de nodos de datos sería mucho mayor que el factor de replicación):

```
triqui2: cd data_node
triqui2: ./ejecuta_datanode.sh triqui1 34567 nodo1

triqui3: cd data_node
triqui3: ./ejecuta_datanode.sh triqui1 34567 nodo2

triqui4: cd data_node
triqui4: ./ejecuta_datanode.sh triqui1 34567 nodo3
```

Por último, arrancamos un programa de prueba:

```
triqui5: cd client_node
triqui5: export BROKER_PORT=34567
```

```
triqui5: export BROKER_HOST=triqui1
triqui5: export CHUNKSIZE=16 # pequeño para las pruebas
triqui5: ./ejecuta.sh Test
```

Fases de la práctica

Para afrontar el trabajo de manera progresiva, se propone un desarrollo incremental en diez fases, valoradas con un punto cada una. 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. Evidentemente, cada pareja puede realizar las fases que considere oportuno, obteniendo la nota correspondiente a aquellas desarrolladas que superen las pruebas pertinentes.

Las 10 fases se pueden dividir en tres grupos dependiendo de a qué parte del sistema involucren:

- Las fases 1, 5, 6 y 7 están relacionadas con los nodos de datos y la funcionalidad `manager` del maestro.
- Las fases 2, 3 y 4 están vinculadas con la funcionalidad `master` del maestro.
- Las fases 8, 9 y 10 corresponden a la parte cliente.

Los dos primeros grupos son prácticamente independientes (el único método común es `selectDataNode`) pudiendo desarrollarse de forma separada por los miembros del grupo de prácticas si lo consideran oportuno. El tercer grupo requiere estar completada la funcionalidad de esos dos primeros grupos.

Fase 1: alta de los nodos de datos

En esta fase, que concierne a los nodos de datos y a la funcionalidad del maestro como `manager`, hay que hacer que cuando se activa un nodo de datos se dé de alta en el maestro:

- En el constructor de `DataNodeImpl` hay que dar de alta ese nodo en `Manager` e implementar el método `getName`.
- En `Manager` hay que habilitar una estructura de datos (lo más razonable sería usar alguna variedad de lista) para guardar los nodos de datos dados de alta e ir añadiéndolos (método `addDataNode`, que tiene que ser sincronizado para evitar condiciones de carrera) según vayan apareciendo. Para facilitar la depuración, también se implementará el método `getDataNodes` que devuelve directamente la lista con los nodos de datos dados de alta en el sistema.
- Asimismo, hay que implementar el método local `selectDataNode`, sincronizado para evitar condiciones de carrera, que, para simplificar, va a devolver como nodo de datos seleccionado el siguiente en orden de alta de forma cíclica (así, si hay 4 nodos de datos, sucesivas llamadas a este método, sean para el fichero que sean, devolverían los nodos 0, 1, 2, 3, 0, 1...). Nótese que, si añade un nuevo de datos, inmediatamente será tenido en cuenta en la próxima llamada a este método (es decir, se seguiría con la asignación cíclica pero ahora el ciclo tendría un nodo más).

Pruebas de la primera fase

Para probar esta fase, puede usar el programa `GetDataNodes` (la validación del método `selectDataNode` se realizará en la tercera fase):

```
triqui5: cd client_node
triqui5: ./ejecuta.sh GetDataNodes triqui1 34567
Lista de nodos de datos
nodo1
nodo2
nodo3
```

Fase 2: `lookup` de un fichero

Esta fase afecta solo a la clase `MasterImpl` y va a implementar el método remoto `lookup` que retorna una referencia remota a un objeto de tipo `File`. Como se verá en la siguiente fase, la clase `FileImpl` almacena la metainformación de un fichero. La clase `MasterImpl` deberá gestionar una estructura de datos (lo más

razonable es que sea algún tipo de mapa) que asocie el nombre de un fichero con el `FileImpl` vinculado con el mismo. La idea es que dos llamadas a `lookup` sobre el mismo fichero por parte de dos clientes deben recibir una referencia remota al mismo objeto `File`. De esta forma, el comportamiento del método `lookup` será el siguiente:

En el constructor meter la inicialización del mapa y en lookup

- Si el fichero no existe (no se ha hecho nunca un `lookup` del mismo y, por tanto, no aparece en la estructura de datos), se habilitará un objeto de tipo `FileImpl`, se le asociará al nombre del fichero y se retornará una referencia al mismo. Nótese la necesidad de que este método sea sincronizado para evitar condiciones de carrera. Recuerde que las peticiones de los clientes se ejecutan de forma concurrente en Java RMI.
- En caso de que ya exista, simplemente se retornará la referencia al objeto `FileImpl` asociado.

Pruebas de la segunda fase

Para probar esta fase, puede usar el programa `CheckFilesLength` que hace dos operaciones `lookup` de los ficheros especificados llamando, a continuación, al método que devuelve la longitud del fichero (que devolverá 0 ya que todavía no está implementado). Lógicamente, las dos operaciones `lookup` del mismo fichero deben obtener la misma referencia remota a `File`.

```
triqui5: cd client_node
triqui5: ./ejecuta.sh CheckFilesLength triqui1 34567 F1 F2 F3
lookup fichero 1ª vez F1
lookup fichero 1ª vez F2
lookup fichero 1ª vez F3
lookup fichero 2ª vez F1
lookup fichero 2ª vez F2
lookup fichero 2ª vez F3
longitud de fichero F1 = 0 chunks
longitud de fichero F2 = 0 chunks
longitud de fichero F3 = 0 chunks
```

Fase 3: Asignación de nodos de datos a los *chunks* de un fichero

Esta fase está centrada, principalmente, en la implementación del método remoto `allocateChunkDescriptors` de la clase `FileImpl`, que recibe como parámetros el rango de *chunks* del fichero para los que hay que asignar nodos de datos. Ese rango viene especificado como el primer *chunk* del rango (se numeran partiendo de 0) y el número de *chunks* que abarca dicho rango.

Esta clase debe usar una estructura de datos que mantenga la lista que contiene la metainformación de cada *chunk* del fichero (`ChunkImpl`). Aunque intuitivamente podría parecer que la solución más razonable es usar algún tipo de lista (una lista de objetos de tipo `ChunkImpl`), la implementación se simplifica significativamente si se utiliza algún tipo de mapa que asocie el número de *chunk* con el objeto `ChunkImpl` vinculado. Para entender el motivo de esta sugerencia, hay que tener en cuenta que, como veremos más adelante con la llamada `seek`, un fichero puede tener huecos y la gestión de los mismos puede ser más laboriosa si se usa una lista.

En esta fase, hay que programar la siguiente funcionalidad:

- En el método `allocateChunkDescriptors` de la clase `FileImpl`, hay que ir comprobando por cada *chunk* involucrado si ya tiene metainformación asociada (es decir, se está sobrescribiendo). Si no la tiene (no está en el mapa), hay que seleccionar un nodo de datos para cada réplica llamando repetidamente al método `selectDataNode` (que se implementó en la primera fase) de la clase `ManagerImpl` y crear un objeto `ChunkImpl` con esa información, añadiéndole a la estructura de datos de tipo mapa que mantiene los *chunks* de este fichero. Tanto si el *chunk* tenía ya metainformación asociada como si no y ha habido que asignársela, se añadirá a la lista de *chunks* que retorna este método. Nótese que, como parte de esta operación, puede crecer el tamaño del fichero, que, lógicamente, nunca puede decrecer.

Pruebas de la tercera fase

Para probar esta fase, puede usar el programa `TestAllocateChunks` que llama al método `allocateChunkDescriptors` con los argumentos que recibe. En primer lugar, puede ejecutar la siguiente prueba que crea un fichero con 7 *chunks* con un hueco de 4 al principio:

```
triqui5: cd client_node
triqui5: ./ejecuta.sh TestAllocateChunks triqui1 34567 F1 4 3
chunk cd2c30a1-2a2b-4f1c-8966-63f6131b7c40
datanode nodo1
datanode nodo2
chunk d5960dd3-3d72-497f-9339-5745a47a3380
datanode nodo3
datanode nodo1
chunk badcf7f3-c2cd-44a5-8133-ef3e847a4120
datanode nodo2
datanode nodo3

triqui5: ./ejecuta.sh CheckFilesLength triqui1 34567 F1 F2 F3
lookup fichero 1ª vez F1
lookup fichero 2ª vez F1
longitud de fichero F1 = 7 chunks
```

A continuación, para el mismo fichero, se pide asignar nodos de datos para 6 *chunks* empezando por el tercero (*chunk* 2). Como resultado de la operación, el fichero tiene 8 *chunks*: dos huecos al principio, dos nuevos a continuación, los tres ya asignados y uno nuevo al final:

```
triqui5: ./ejecuta.sh TestAllocateChunks triqui1 34567 F1 2 6
chunk a2f48b23-d324-4474-8860-75da739dbf29
datanode nodo1
datanode nodo2
chunk 694c9cd3-0ce3-4ff2-b437-8de2441d6565
datanode nodo3
datanode nodo1
chunk cd2c30a1-2a2b-4f1c-8966-63f6131b7c40
datanode nodo1
datanode nodo2
chunk d5960dd3-3d72-497f-9339-5745a47a3380
datanode nodo3
datanode nodo1
chunk badcf7f3-c2cd-44a5-8133-ef3e847a4120
datanode nodo2
datanode nodo3
chunk 315533ce-2323-4a70-a040-4caae0d6e69f
datanode nodo2
datanode nodo3

triqui5: ./ejecuta.sh CheckFilesLength triqui1 34567 F1
lookup fichero 1ª vez F1
lookup fichero 2ª vez F1
longitud de fichero F1 = 8 chunks
```

Fase 4: Obtención de los nodos de datos de los *chunks* de un fichero

Esta fase está centrada en la implementación del método remoto `getChunkDescriptors` de la clase `FileImpl`, que recibe como parámetros el rango de *chunks* del fichero para los que hay que recuperar su metainformación. Ese rango viene especificado como el primer *chunk* del rango (se numeran partiendo de 0) y el número de *chunks* que abarca dicho rango.

La funcionalidad a desarrollar es similar a la de la fase previa, pero, en este caso, no hay que asignar metainformación sino simplemente recuperarla (es lo mismo que hacía el método de la fase anterior en caso de sobrescritura). Por cada *chunk* de la petición, hay que tener en cuenta tres casos:

- si tiene asignados ya identificador y nodos de datos (está en el mapa), se añade a la lista que se retorna.
- si no los tiene (no está en el mapa) pero está dentro del tamaño del fichero (es un hueco), se añade un `null` en esa posición de la lista que se retorna.
- en caso contrario, no se añade nada a la lista ya que hemos llegado al final del fichero.

Pruebas de la cuarta fase

Para probar esta fase, puede usar el programa `GetAllocatedChunks` que llama al método `getChunkDescriptors` con los argumentos que recibe. Siguiendo con la prueba previa, puede solicitarse la metainformación de 8 *chunks* comenzando por el segundo (*chunk* 1). Debe devolver solo 7 entradas porque la última está fuera del fichero, siendo la primera un hueco.

```
triqui5: ./ejecuta.sh GetAllocatedChunks triqui1 34567 F1 1 8
chunk no asignado
chunk a2f48b23-d324-4474-8860-75da739dbf29
datanode nodo1
datanode nodo2
chunk 694c9cd3-0ce3-4ff2-b437-8de2441d6565
datanode nodo3
datanode nodo1
chunk cd2c30a1-2a2b-4f1c-8966-63f6131b7c40
datanode nodo1
datanode nodo2
chunk d5960dd3-3d72-497f-9339-5745a47a3380
datanode nodo3
datanode nodo1
chunk badcf7f3-c2cd-44a5-8133-ef3e847a4120
datanode nodo2
datanode nodo3
chunk 315533ce-2323-4a70-a040-4caae0d6e69f
datanode nodo2
datanode nodo3
```

Si se prueba un rango completamente más allá del fin de fichero, no debe devolver nada:

```
triqui5: ./ejecuta.sh GetAllocatedChunks triqui1 34567 F1 8 4
```

Fase 5: Escritura de un *chunk* en un nodo de datos secundario

Para entender esta fase hay que tener en cuenta que el método remoto `writeChunk` se va a usar en dos escenarios:

- Si el primer parámetro es una lista vacía, se trata de una escritura en un nodo secundario que solo requiere escribir el *chunk* en un fichero local con el mismo nombre del *chunk*.
- En caso contrario, se trata de una escritura en un nodo primario que conlleva, además de la escritura local, propagar la operación a los nodos secundarios (es decir, llamar al mismo método en los nodos secundarios pero con la lista vacía). Este escenario se recoge en la séptima fase.

Esta fase se centra en el primer escenario (lista vacía) por lo que solo requiere la escritura del contenido del *chunk* (`buffer`) en un fichero con el mismo nombre que el del *chunk* (**nota importante**: el fichero se debe almacenar en un directorio, que ya está creado, que tiene el mismo nombre que el nodo de datos: `dataNodeName/chunkName`). Se recomienda usar la clase `FileOutputStream` para realizar la operación. El método devolverá falso si se produce una excepción en la operación sobre el fichero (por ejemplo, falla la escritura porque la partición está llena).

Pruebas de la quinta fase

Esta prueba crea un fichero denominado `CHUNK1` en el primer nodo de datos del sistema cuyo contenido es HOLA:

```
triqui5: ./ejecuta.sh WriteDataNode triqui1 34567 CHUNK1 HOLA

# puede comprobar en triqui2 que el contenido es correcto
triqui2: cat data_node/bin/nodo1/CHUNK1
HOLA
```

Fase 6: Lectura de un *chunk* de un nodo de datos

Esta fase se centra en la implementación del método remoto `readChunk` que lee el fichero que contiene un *chunk* (recuerde que el fichero estará almacenado en `dataNodeName/chunkName`). Se recomienda el uso de la clase `FileInputStream` para realizar la operación. El método devolverá `null` si se produce una excepción en la operación sobre el fichero.

Pruebas de la sexta fase

Esta prueba lee el fichero denominado `CHUNK1` creado en la prueba previa en el primer nodo de datos del sistema:

```
triqui5: ./ejecuta.sh ReadDataNode triqui1 34567 CHUNK1
Leído: HOLA
```

Fase 7: Escritura de un *chunk* en un nodo de datos primario

Retomamos el método `writeChunk` de la quinta fase, pero teniendo en cuenta el segundo escenario: si un nodo de datos recibe una llamada a este método en la que la lista recibida como parámetro no está vacía, se trata de una escritura que debe propagarse a los nodos de datos de la lista (que son los secundarios para ese *chunk*) llamando a este mismo método para esos nodos, pero con un primer parámetro que corresponda a una lista vacía. Acabada la propagación, se procede como en la quinta fase escribiendo el contenido del *chunk* en un fichero local.

Téngase en cuenta que dos clientes pueden intentar escribir simultáneamente el mismo *chunk* lo que podría causar que quedaran valores diferentes en las copias. Para evitar este problema, se deben secuencializar las escrituras sobre un mismo *chunk*: hasta que no se complete una operación de escritura en un *chunk* con la propagación a todos los nodos involucrados, no podrá iniciarse otra operación de escritura sobre ese mismo *chunk*.

Dado que el uso de un método sincronizado afectaría innecesariamente a escrituras sobre distintos *chunks*, se proporciona la clase `LockManager`, ya implementada, que gestiona una clase interna de tipo `Lock` asociada a un determinado nombre (en este caso, sería el nombre del *chunk*). A continuación, se muestra un ejemplo de su uso:

```
LockManager lm; // gestor de Locks (factoría de cerrojos asociados a nombres)
..............
lm = new LockManager(); // hay que crear una instancia para usarlo
..............
LockManager.Lock lck; // declaro un lock
.................
// pido un lock asociado a un nombre y lo bloqueo
(lck = lm.openLock(chunkName)).lock();
// Inicio de la sección critica asociada a ese nombre
.................
lck.unlock(); // desbloqueo el lock
// Fin de la sección critica asociada a ese nombre
```

Pruebas de la séptima fase

Esta prueba crea un fichero denominado `CHUNK2` con el contenido ADIOS invocando el método remoto `writeChunk` del primer nodo de datos del sistema pasándole como primer parámetro una lista con los nodos de datos restantes:

```
triqui5: ./ejecuta.sh WriteDataNodes triqui1 34567 CHUNK2 ADIOS

# puede comprobar en triqui2 que el contenido es correcto
triqui2: cat data_node/bin/nodo1/CHUNK2
ADIOS

# puede comprobar en triqui3 que el contenido es correcto
triqui3: cat data_node/bin/nodo2/CHUNK2
ADIOS
```

```
# puede comprobar en triqui4 que el contenido es correcto
triqui4: cat data_node/bin/nodo3/CHUNK2
ADIOS
```

A partir de aqui todas las fases son pal cliente usando lo que hemos ido haciendo

Fase 8: Clase de cliente `GFSFile`: constructor y algunos métodos

Las tres últimas fases corresponden a la parte del cliente implementada en la clase `GFSFile` que completa la funcionalidad del sistema. Antes de entrar a especificar el objetivo concreto de esta fase, se van a realizar algunas consideraciones sobre estas tres fases finales:

- La clase `GFSFile` hace uso de los servicios desarrollados en las fases previas para integrar la funcionalidad de este sistema. Por tanto, a la hora de desarrollar esta clase se va a poder reutilizar parte del código de los programas de prueba de esas fases previas ya que hacían también uso de estos servicios.
- El programa de prueba para esta fase se denomina `Test` y ofrece una interfaz de texto que permite al usuario solicitar las distintas operaciones del sistema de ficheros.
- La clase `GFSFile` debe acceder a las variables de entorno especificadas al principio del documento y realizar la operación de búsqueda en el *registry* del servicio `master` del maestro (esa búsqueda es un ejemplo de código que se puede reutilizar de los programas de prueba de las fases anteriores).
- El puntero de posición en el acceso a un fichero (*offset*) se gestiona localmente (no requiere estado en el servidor): simplemente se guarda en un atributo local de la clase `GFSFile`.

A continuación, se describe la funcionalidad a implementar en esta fase:

- El constructor de la clase `GFSFile`. Debe acceder a las variables de entorno y realizar la operación de `lookup` del *registry* para obtener una referencia remota del servicio del maestro requerido.
- Método `length`. Debe devolver el tamaño del fichero en bytes invocando directamente el método remoto de `File` correspondiente.
- Método `seek`. Actualiza el atributo local para establecer el nuevo valor del puntero de posición.
- Método `getFilePointer`. Devuelve el valor actual del puntero de posición del fichero.

Pruebas de la octava fase

Como se ha comentado previamente, para probar estas tres últimas fases se usa el programa interactivo `Test`. A continuación, se muestra cómo se ejecuta este programa y cómo se realizan algunas pruebas:

```
triqui5: cd client_node
triqui5: export BROKER_PORT=34567
triqui5: export BROKER_HOST=triqui1
triqui5: export CHUNKSIZE=16 # pequeño para las pruebas
triqui5: ./ejecuta.sh Test
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
0
Introduzca nombre de fichero:
fichero
Fichero abierto ID 0
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
3
Introduzca ID de fichero y nueva posición (debe ser múltiplo de chunkSize)
0 32
Puntero colocado en posicion 32
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
4
Introduzca ID de fichero
0
Puntero colocado en posicion 32
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
```

```
5
Introduzca ID de fichero
0
tamaño del fichero 0
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
^D
```

Fase 9: Clase de cliente `GFSFile`: `write`

En esta fase se implementa el método `write` que debe llevar a cabo la siguiente labor:

- Calcular el rango de *chunks* afectados teniendo en cuenta la posición actual y el número de bytes pedidos. Recuerde que, para simplificar, ambos valores serán múltiplos del tamaño del *chunk*, por lo que el cálculo son dos simples divisiones.
- Llamar a `allocateChunkDescriptors` con ese rango.
- Iterar sobre la lista de *chunks* obtenida.
- Por cada *chunk*, obtener el nodo de datos primario (el primero de la lista) e invocar el método `writeChunk` de ese nodo pasándole como primer parámetro la lista de los nodos de datos secundarios (la lista original eliminando el primario), el nombre del *chunk* y la *rodaja* del *buffer* recibido en el `write` que corresponde a ese *chunk* (puede usar `Arrays.copyOfRange` para extraerla).
- El método avanzará el puntero de posición conforme a la cantidad escrita.
- El método retornará falso si `writeChunk` devuelve un error.

Pruebas de la novena fase

Use el programa `Test` para realizar sucesivas escrituras en un fichero y compruebe que los ficheros almacenados en los nodos de datos son correctos. Intercale algún `seek` entre las escrituras para comprobar que se generan correctamente huecos.

Fase 10: Clase de cliente `GFSFile`: `read`

En esta fase se implementa el método `read` que debe llevar a cabo la siguiente labor:

- Calcular el rango de *chunks* afectados teniendo en cuenta la posición actual y el número de bytes pedidos. Recuerde que, para simplificar, ambos valores serán múltiplos del tamaño del *chunk*, por lo que el cálculo son dos simples divisiones.
- Llamar a `getChunkDescriptors` con ese rango.
- Iterar sobre la lista de *chunks* obtenida.
- Por cada *chunk*, debe comprobar, en primer lugar, si es nulo o no:
  - Si no es nulo, invocará el método `readChunk` de cualquiera de los nodos de datos de la lista, copiando (puede usar `System.arraycopy`) en la *rodaja* correspondiente del *buffer* del `read` los datos retornados por `readChunk`.
  - Si es nulo (se trata de un hueco), rellenará con ceros (el carácter nulo) la *rodaja* correspondiente del *buffer* del `read` (puede usar `Arrays.fill`).
- El método avanzará el puntero de posición conforme a la cantidad leída.
- El método retornará 0 si `readChunk` devuelve un error.

Pruebas de la décima fase

Use el programa `Test` para mezclar escrituras, operaciones `seek` y lecturas. Para comprobar que se leen correctamente los huecos (se leen caracteres nulos), dado que los caracteres nulos no aparecen en la pantalla, puede ejecutar el programa de `Test` redirigido a un fichero y luego revisar ese fichero con un mandato como `od`.

El siguiente ejemplo usa esa estrategia de dirección realizando las siguientes operaciones:

- `seek` a la posición 32.

- escritura de 16 bytes (1 *chunk*) que deja un hueco inicial de 2 *chunks*.
- `seek` a la posición 0.
- lectura de 64 bytes (4 *chunks*) que obtendrá solo 48 bytes (3 *chunks*): 32 a 0 al principio y los 16 bytes escritos por el `write` como se puede apreciar con el mandato `od`.

```
triqui5: cd client_node
triqui5: export BROKER_PORT=34567
triqui5: export BROKER_HOST=triqui1
triqui5: export CHUNKSIZE=16 # pequeño para las pruebas
triqui5: ./ejecuta.sh Test > /tmp/TRAZA
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
0
Introduzca nombre de fichero:
fich
Fichero abierto ID 0
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
3
Introduzca ID de fichero y nueva posición (debe ser múltiplo de chunkSize)
0 32
Puntero colocado en posicion 32
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
2
Introduzca ID de fichero y cantidad a escribir (debe ser múltiplo de chunkSize)
0 16
Escritos 16 bytes
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
3
Introduzca ID de fichero y nueva posición (debe ser múltiplo de chunkSize)
0 0
Puntero colocado en posicion 0
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
1
Introduzca ID de fichero y cantidad a leer (debe ser múltiplo de chunkSize)
0 64
Leidos 48 bytes
Seleccione operacion
        0 open|1 read|2 write|3 seek|4 getFilePointer|5 length (Ctrl-D para terminar)
^D

triqui5: od -cv /tmp/TRAZA
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P
```

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/miniGFS.2022/`.

Entrega de la práctica

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

```
entrega.sd miniGFS.2022
```

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.
- `master_node/src/master/MasterImpl.java`
- `master_node/src/master/FileImpl.java`
- `master_node/src/manager/ManagerImpl.java`
- `data_node/src/datanode/DataNodeImpl.java`
- `client_node/src/client/GFSFile.java`