Gestor de archivos distribuido

Ana.Rubio@uclm.es

Proyecto de prácticas

El objetivo principal del proyecto es diseñar un sistema cliente-servidor que permita la subida y descarga de ficheros. La implementación de este proyecto permitirá al alumno trabajar, mediante ZeroC Ice, aspectos relacionados con el despliegue de servidores de forma dinámica, la gestión de aplicaciones con IceGrid, y la comunicación asíncrona mediante canales de eventos con IceStorm.

En los siguientes apartados se describirán los detalles de cada uno de los componentes que componen el sistema, así como las interfaces que determinarán los métodos que deberán implementar. También se darán pautas para facilitar el proceso de desarrollo, dividiendo el proyecto en varias fases. Finalmente, se describirá en qué consiste la evaluación del proyecto y cuáles son las condiciones de entrega.

1. Componentes e interacciones

El sistema estará formado por cinco tipos de componentes: los sirvientes de tipo `Uploader`, que funcionarán como manejadores del proceso de subida de un fichero; los de tipo `Downloader`, que se encargarán de la descarga de ficheros; los de tipo `FileManager`, que se encargarán de la gestión de los `Uploader` y `Downloader`, y del borrado de ficheros; y los de tipo `Frontend`, que harán de intermediarios entre las peticiones de los clientes y los `FileManager`, llevando un registro de los ficheros existents en el sistema. Por supuesto, también habrá clientes, que serán los encargados de solicitar la subida y descarga de ficheros.

1.1. Cliente

El cliente es el componente que se encarga de solicitar la subida, la descarga, o el borrado de ficheros. Para ello, se proporciona el nombre del fichero a subir, o el hash del fichero a borrar o descargar. Como se verá en la Sección 1.2, el `Frontend`, que es el tipo de sirviente con el que interacciona directamente el cliente, también permite listar todos los ficheros existentes en el sistema distribuido. La aplicación cliente debe implementar una forma cómoda de interactuar con el sistema, y debe permitir al usuario realizar las acciones mencionadas.

1.2. Frontend

El `Frontend` es el componente encargado de gestionar las peticiones de los clientes y de llevar un registro de los ficheros que existen en el sistema, que pueden estar distribuidos en diferentes nodos y ser accesibles por medio de distintos `FileManager`. Para ello, el `Frontend` debe ser capaz 1) de comunicarse con todos los `FileManager` existentes en el sistema, 2) de llevar un control de qué ficheros se añaden y se borran del sistema (solo cuando la tarea haya terminado), y 3) de sincronizarse con otros `Frontend` para mantener el registro de ficheros actualizado. Concretamente:

- El `Frontend` siempre está a la espera de recibir peticiones de subida, descarga o borrado de ficheros, o de listar todos los ficheros existentes en el sistema, por parte de aplicaciones cliente.

- Cuando se solicita el listado de ficheros disponibles, el `Frontend` retorna una lista de tipo `FileList` con todos los ficheros que figuran en su registro en estructuras `FileInfo`, que contienen el nombre del fichero y su hash.

```
struct FileInfo {
  string name;
  string hash;
};

sequence<FileInfo> FileList;
FileList getFileList();
```

- Cuando se solicita la subida de un fichero, indicando el nombre del fichero, el `Frontend` solicita a un `FileManager` (selecciona el que menos ficheros tenga) la creación de un `Uploader` para dicho fichero, y retorna la referencia del `Uploader` creado. Si el nombre del fichero ya figura en el registro, lanza una excepción.

```
Uploader* uploadFile(string filename) throws FileNameInUseError;
```

- Cuando se solicita la descarga de un fichero, indicando el hash del fichero, el `Frontend` solicita a un `FileManager` (selecciona el que contiene el fichero de interés) la creación de un `Downloader` para dicho fichero, y retorna la referencia del `Downloader` creado. Si el fichero no figura en el registro, lanza una excepción.

```
Downloader* downloadFile(string hash) throws FileNotFoundError;
```

- Cuando se solicita el borrado de un fichero, indicando el hash del fichero, el `Frontend` solicita a un `FileManager` (selecciona el que contiene el fichero de interés) el borrado de dicho fichero. Si el fichero no figura en el registro, lanza una excepción.

```
void removeFile(string hash) throws FileNotFoundError;
```

Es importante tener en cuenta que las tareas de adición y borrado de ficheros que se han listado anteriormente no añaden o eliminan los ficheros del registro del `Frontend`, solo del almacenamiento (en el lado del `FileManager`). El registro del `Frontend` se actualiza cuando el `FileManager` le notifica que la tarea, de adición o de borrado, ha terminado por medio de un canal de eventos denominado `FileUpdates`. Este canal de eventos, que se implementa por medio de sirvientes con el mismo nombre, `FileUpdates`, permite al `Frontend` mantener su registro de ficheros actualizado. Concretamente, esta actualización consiste en:

1. Cuando se termina la tarea de subida o de borrado de un fichero, el `Uploader` o el `FileManager`, respectivamente, notifica que la tarea ha terminado enviando un evento con una estructura

`FileData` al canal de eventos `FileUpdates` con la información del archivo y la referencia del `FileManager` que lo gestiona.

```
struct FileData {
  FileInfo fileInfo;
  FileManager* fileManager;
};

interface FileUpdates {
  void new(FileData file);
  void removed(FileData file);
};
```

2. Cuando los objetos suscriptores `FileUpdates` existentes en cada servidor reciben el evento, se lo hacen llegar a los `Frontend`.
3. Cuando al `Frontend` le llega la información, actualiza su registro de ficheros.

De la misma forma, para actualizar los registros de ficheros de los `Frontend` que se lancen una vez el sistema ya esté en marcha, se implementa un canal de eventos denominado `FrontendUpdates`. Este canal de eventos, que se implementa por medio de sirvientes con el mismo nombre, `FrontendUpdates`, permite al `Frontend` nuevo poner su registro de ficheros al día. Concretamente, esta actualización consiste en:

1. Cuando se lanza un nuevo `Frontend`, este publica un evento con su proxy directo en el canal de eventos `FrontendUpdates`. Es decir, invocando el método `newFrontend` del publisher `FrontendUpdates`.

```
interface FrontendUpdates {
  void newFrontend(Frontend* newFrontend);
};
```

2. Cuando los objetos suscriptores `FrontendUpdates` existentes en cada servidor reciben el evento, se lo hacen llegar a los `Frontend`, incluido el nuevo (tendrá que filtrar su propio evento), y estos responden al nuevo `Frontend` con su proxy directo por medio de una invocación remota.

```
void replyNewFrontend(Frontend* oldFrontend);
```

3. Una vez los `Frontend` responden al nuevo objeto, envían su registro de ficheros al nuevo `Frontend` por medio de eventos publicados en el canal de eventos `FileUpdates`.

Esta forma de actualizar el registro de ficheros, aunque eficaz, no es eficiente, ya que yodos los sirvientes `Frontend` existentes envían su registro completo de ficheros cada vez que se lanza un nuevo `Frontend`. Se propondrá la implementación de una mejora más adelante.

1.3. FileManager

El `FileManager` es el componente localizado en el lugar donde se almacenarán los archivos, y es el encargado de crear sirvientes `Downloader` y `Uploader` a demanda. Cada vez que se solicita la subida o la descarga de un fichero, se crea un sirviente de estos tipos, que se encargará de gestionar la tarea y será destruido al terminar. El `FileManager` también gestiona el borrado de ficheros, y solamente interacciona de forma directa con los `Frontend`. Concretamente:

- Está siempre a la espera de recibir peticiones de creación de sirvientes `Uploader` y `Downloader` por parte de los `Frontend`.
- Cuando se solicita la subida de un fichero, indicando el nombre del fichero, crea un sirviente `Uploader` y retorna su referencia.

  ```
  Uploader* createUploader(string filename);
  ```

- Cuando se solicita la descarga de un fichero, indicando el hash del fichero, crea un sirviente `Downloader` y retorna su referencia. Si el fichero no existe, no crea el objeto y lanza una excepción.

  ```
  Downloader* createDownloader(string hash) throws FileNotFoundError;
  ```

- Cuando se solicita el borrado de un fichero, indicando el hash del fichero, le indica al `FileManager` que lo contiene que borre el fichero. Si el fichero no existe, lanza una excepción.

  ```
  void removeFile(string hash) throws FileNotFoundError;
  ```

1.4. Uploader

El `Uploader` es el componente encargado de gestionar la subida de un fichero al sistema. Cuando estos objetos se crean, abren el fichero en el que se va a escribir, y se quedan a la espera de recibir los datos del fichero. Cuando se termina de escribir el fichero, se cierra y se notifica mediante un evento de que la tarea ha terminado. Concretamente:

- Cuando es creado, queda a la espera de recibir los datos del fichero que se desea almacenar.
- Cuando se recibe un bloque de datos, que se ha enviado directamente desde el cliente, se escribe en el fichero.

  ```
  void send(string data);
  ```

- Cuando no hay más datos que mandar, el cliente guarda el archivo. En este momento, el archivo se almacena con el hash del fichero como nombre. En caso de que ya exista un fichero con ese hash (es decir, el archivo ya ha sido subido, pero con otro nombre), se lanza una excepción que contiene el hash del fichero que ya existe. Si el fichero es almacenado sin problemas, se publica un evento en el canal de eventos `FileUpdates` para notificar que la

tarea ha terminado y para que los `Frontend` puedan actualizar sus registros, y se retorna al cliente una estructura `FileInfo` con el nombre del fichero y su hash.

```
FileInfo save() throws FileAlreadyExistsError;
```

- Finalmente, el `Uploader` se destruye bajo petición explícita del cliente.

```
void destroy();
```

1.5. Downloader

El `Downloader` es el componente encargado de gestionar la descarga de un fichero del sistema. Cuando estos objetos se crean, abren el fichero que se va a leer, y se quedan a la espera de recibir peticiones de lectura de datos. El cliente debe detectar cuándo ha terminado de leer el fichero, y destruir el objeto explícitamente. Concretamente:

- Cuando es creado, abre el fichero que se desea leer y queda a la espera de recibir peticiones de lectura de datos.
- Cuando se recibe una petición de lectura de datos, se lee un bloque de datos del fichero de un tamaño concreto y se retorna al cliente.

```
string recv(int size);
```

- Cuando el cliente ha terminado de leer el fichero, destruye el objeto explícitamente.

```
void destroy();
```

2. Fases de desarrollo

Con el objetivo de facilitar el desarrollo de la aplicación, el proceso de implementación se divide en tres fases, cada una de las cuales añade funcionalidades nuevas al sistema. En la primera fase se debe implementar el funcionamiento básico del sistema; en la segunda fase, una versión que soporte múltiples servidores con sirvientes de diverso tipo; y, finalmente, en la tercera fase se ha de configurar la gestión de la aplicación con IceGrid.

2.1. FASE 1: Funcionamiento básico

En esta fase se debe implementar el funcionamiento básico del sistema, que consiste en la subida, descarga y borrado de ficheros. Para ello, es necesario habilitar con *icegridregistry* transparencia de localización, y se debe implementar una primera versión del código del `Frontend`, del `FileManager`, del `Uploader` y del `Downloader`, así como el código de la aplicación cliente. En concreto, en esta fase se debe conseguir:

- Un cliente capaz de subir, descargar y borrar ficheros.
- Un `Frontend` capaz de gestionar las peticiones de los clientes, y de crear `Uploader` y `Downloader` en los `FileManager`.

- Un `FileManager` capaz de crear `Uploader` y `Downloader` a demanda, y de gestionar el borrado de ficheros.
- Un `Uploader` capaz de recibir los datos de un fichero y de almacenarlos en el sistema.
- Un `Downloader` capaz de leer los datos de un fichero y de enviarlos al cliente.

Sin embargo, en esta fase no se debe utilizar IceStorm, ni es necesaria la gestión de múltiples servidores. Es decir, que solamente deben ser funcionales un cliente, un `Frontend`, y un `FileManager`. Por tanto, el `Frontend` no tiene que sincronizarse con otros `Frontend` y, puesto que no se usan canales de eventos, no mantiene un registro de ficheros. En este contexto, el `Frontend` aún no implementa el listado de ficheros y, por ello, el cliente debe conocer los ficheros que ha subido para poder hacer peticiones de descarga y borrado.

2.2. FASE 2: IceStorm y Múltiples servidores

En esta fase se debe implementar la gestión de múltiples servidores, y se debe utilizar IceStorm para la comunicación asíncrona entre los `Frontend` y los `FileManager`. En concreto, en esta fase se debe conseguir:

- Un cliente capaz de subir, descargar, borrar y listar ficheros.
- Un `Frontend` capaz de gestionar las peticiones de los clientes, creando objetos `Uploader` y `Downloader` en los `FileManager`, y manteniendo un registro de ficheros actualizado (sincronización de `Frontends`).
- Un `FileManager` capaz de crear `Uploader` y `Downloader` a demanda, y de gestionar el borrado de ficheros. Debe poder notificar que la tarea de borrado ha terminado por medio de eventos.
- Un `Uploader` capaz de recibir los datos de un fichero y de almacenarlos en el sistema. Debe poder notificar que la tarea ha terminado por medio de eventos.
- Un `Downloader` capaz de leer los datos de un fichero y de enviarlos al cliente.
- Objetos `FrontendUpdates` y `FileUpdates` que permitan la sincronización de `Frontends` y la actualización de los registros de ficheros.

En esta fase debe ser posible ejecutar más de un `Frontend` y más de un `FileManager`. El cliente podrá utilizar cualquiera de los `Frontend` disponibles, y los `Frontend` deben conocer todos los `FileManager` en marcha para distribuir las tareas de subida de ficheros y para saber a quién asignar tareas de descarga o de borrado.

2.3. FASE 3: Gestión de la aplicación con IceGrid

En esta fase la aplicación debe implementarse de modo que sea posible llevar a cabo su despliegue con IceGrid. La aplicación se debe llamar *URFSApp*, y debe quedar configurada en tres nodos de la siguiente manera:

- **Nodo 1:** Aloja el binder, un servidor IcePatch2, y un servidor de IceStorm.
- **Nodo 2:** Aloja tres servidores `Frontend`, definidos mediante una plantilla. Estos servidores deberán pertenecer a un grupo de réplica alcanzable por medio del identificador *frontend*.
- **Nodo 3:** Aloja dos servidores `FileManager`, definidos mediante una plantilla. Sus proxies indirectos serán conocidos por los `Frontend`.

Configura los servidores `Frontend` para que arranquen solo bajo demanda, y configura los servidores `FileManager` para que arranquen automáticamente. El cliente debe conectar con los `Frontend` por medio del proxy del grupo de réplica *frontend*, no a través de los proxies directos/indirectos de cada sirviente.

3. Evaluación

La entrega del proyecto se realizará en grupos de dos personas y se hará un seguimiento de su proceso de desarrollo mediante un repositorio compartido generado con GitHub Classroom. Para poder crear dicho repositorio se deben seguir los siguientes pasos:

1. Conformad el equipo
2. Elegid cuál de los integrantes del equipo será el portavoz a lo largo de las prácticas
3. El portavoz debe entrar en el enlace *[se proporcionará más adelante]* y crear un equipo con el siguiente nombre: *ApellidosPortavoz-ApellidosCompañero*
4. Una vez el equipo haya sido creado por el portavoz, el otro integrante ha de entrar en el enlace del punto anterior y elegir su equipo

Si todo ha ido correctamente, deberías poder acceder a un repositorio localizado en *https://github.com/ssdd-classrooms/urfs-ApellidosPortavoz-ApellidosCompañero*.

La entrega final tiene como fecha límite el **12 de enero de 2024 a las 23h59**, y deberá hacerse a través de una tarea de Campus Virtual a la que el portavoz subirá un comprimido .zip llamado *ApellidosPortavoz-ApellidosCompañero.zip* incluyendo todos los ficheros necesarios para la compilación de los distintos componentes, configuración de la aplicación (fichero *XML*), y otros scripts y ficheros necesarios para el despliegue, ejecución y prueba de la aplicación.

La calificación obtenida dependerá de las funcionalidades implementadas, teniendo en cuenta que la implementación de únicamente las fases 1 y 2 (ver Secciones 2.1 y 2.2), es decir, del nivel básico, permite obtener hasta un máximo de 19 puntos. La implementación de la fase 3 (ver Sección 2.3), del nivel avanzado, permite obtener hasta 6 puntos adicionales, de modo que la nota máxima posible es de 25 puntos. Además, se valorarán con un punto extra sobre la nota máxima (es decir, es posible obtener más de 25 puntos en el apartado de prácticas) las siguientes mejoras:

- Desarrollo de un frontend web mínimo para el cliente (+1p).
- Propuesta y desarrollo de una mejora del método de actualización/sincronización de registros de ficheros de los `Frontend` (+1p).

Para terminar, será necesario incluir en la entrega un fichero README donde se indiquen los nombres de los integrantes del grupo, si se ha implementado el nivel básico o el nivel avanzado, y si se han implementado alguna de las mejoras anteriormente indicadas. La defensa del proyecto se hará los días posteriores a la entrega y con cita previa, de modo que el alumno debe comprobar que todo funciona correctamente en su máquina y en el entorno de forma que pueda mostrar el código y poner la aplicación en ejecución.

3.1. Criterios

Se valorarán dos aspectos: funcionalidad y calidad. La **funcionalidad** valora que se resuelva el problema conforme a sus especificaciones y que sea consistente, es decir, funcione adecuadamente en todas y cada una de las ejecuciones. Si solo funciona a veces, se considera que no funciona. Si

existen casos no contemplados o faltan mecanismos de gestión de errores, la puntuación se verá afectada negativamente aunque el programa nunca falle.

En cuanto a la **calidad** del código, se valoran los siguientes aspectos:

- Estructura y organización del código, mediante clases, funciones o métodos.
- Elección de nombres significativos para funciones, variables, etc.
- Evitar uso de variables globales innecesarias.
- Evitar anidación injustificada de estructuras de control.
- Evitar uso incorrecto de tipos de datos, código redundante, código muerto, etc.
- Evitar bloques (funciones/métodos) demasiado largos.

La **valoración de la calidad es negativa**, es decir, restará sobre la nota final obtenida en funcionalidad. Esto implica que una práctica completamente funcional puede no obtener la máxima nota. Es importante destacar que **la defensa también puede afectar de forma negativa a la nota final**.

3.1.1. Penalizaciones

Se contemplan 3 motivos por los que la puntuación total de la actividad puede sufrir penalizaciones:

- Usar una versión de Python anterior a la 3.5: -3 puntos.
- Errores de sintaxis que impiden ejecutar el programa: -3 puntos.
- Incumplir otras condiciones del enunciado: hasta -10 puntos.

En el caso de errores de sintaxis, si supone cambios menores, se pedirá al alumno presentar una versión corregida.

3.2. Código de terceros

El alumno debe tener especial cuidado con la incorporación de código ajeno en su programa. Está PERMITIDO siempre que se cumplan las siguientes condiciones:

- Los fragmentos copiados y su autor están claramente identificados.
- El alumno comprende y es capaz de explicar con TODO DETALLE la funcionalidad del código copiado.
- El autor del código original permite la copia mediante la licencia correspondiente y esta aparece claramente junto al código original.
- El autor del original mantiene su código públicamente accesible (y el alumno proporciona el enlace en forma de comentario).
- El autor del original no es ni ha sido alumno de la UCLM.

Infringir **cualquiera** de estas normas al utilizar código ajeno se considerará plagio. Cometer plagio implica una calificación de 0 en la actividad «Realización de prácticas de laboratorio» según el artículo 9 de la «Guía de Evaluación del Estudiante»:

> Art. 9. Realización fraudulenta de pruebas de evaluación.
> 1. La constatación de la realización fraudulenta de una prueba de evaluación o el

incumplimiento de las instrucciones fijadas para la realización de la prueba dará lugar a la calificación de suspenso (con calificación numérica de 0) en dicha prueba. En el caso particular de las pruebas finales, el suspenso se extenderá a la convocatoria correspondiente.