Estructura del proyecto NanoFiles

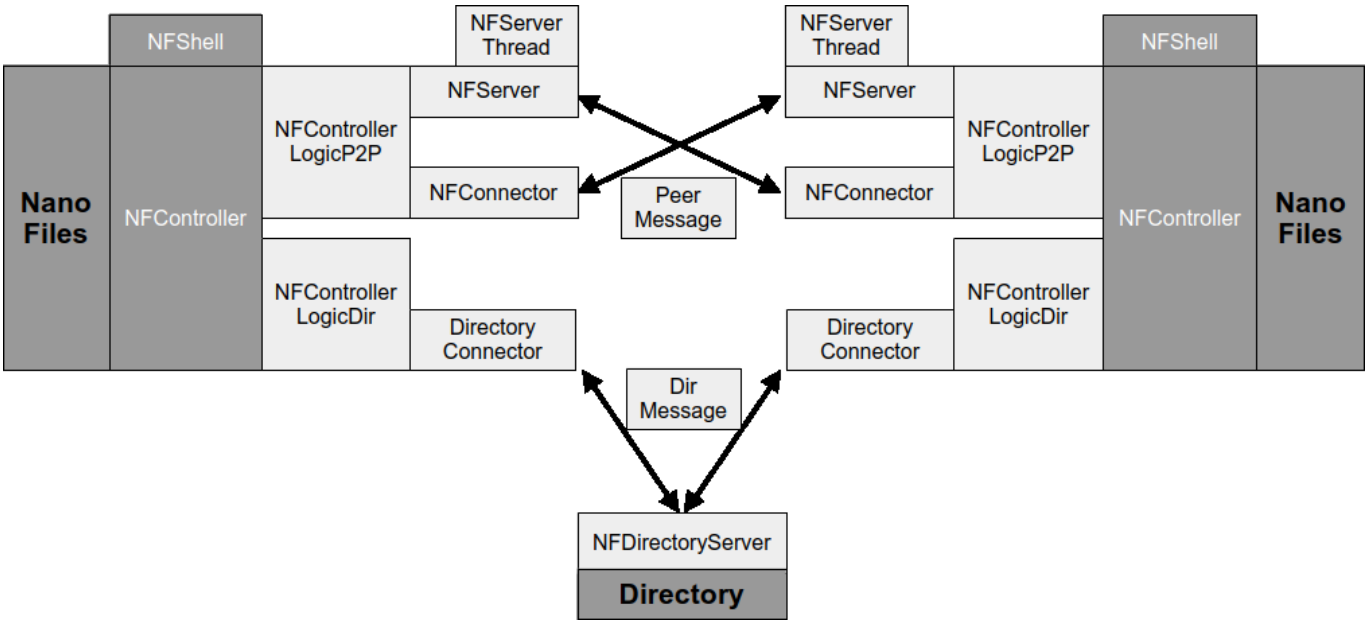
Redes de Comunicaciones - Curso 2024/25

Introducción

El objetivo de este boletín de prácticas es describir los componentes software que conforman el proyecto *nanoFilesP2P* que se proporciona a los estudiantes como punto de partida para el desarrollo de la práctica. Se persigue que los alumnos se familiaricen con la estructura de clases y sus principales métodos, de forma que adquieran soltura a la hora de modificar el código con el fin de implementar la funcionalidad requerida según la especificación del proyecto de prácticas.

Estructura de clases y relaciones entre ellas

La siguiente figura ilustra cómo se organiza el código del proyecto, qué clases se relacionan entre sí y cuáles llevan a cabo la comunicación entre las diferentes entidades de *NanoFiles*.



- En negrita se muestran las clases que contienen el método *main* de cada una de las dos aplicaciones Java que conforman el sistema *NanoFiles*.
- En gris oscuro se muestran las clases cuya implementación está en gran parte o completamente acabada.
- Las clases que tienen una relación de clientela comparten una arista común entre sus bloques. A continuación, se describen brevemente las relaciones entre clases:
  - `NanoFiles` usa una variable `NFController` para ir procesando comandos.
  - `NFController` tiene un atributo `NFShell` para leer comandos del shell introducidos por el usuario.
  - `NFController` tiene un atributo `NFControllerLogicDir` para procesar comandos que requieren de interacción con el directorio.

    - `NFControllerLogicDir` tiene un atributo `DirectoryConnector` para llevar a cabo las interacciones con el directorio (código cliente UDP).
    - `DirectoryConnector` es el cliente UDP encargado de implementar la comunicación mediante datagramas con el servidor de directorio de *NanoFiles*.
    - `DirectoryConnector` y `NFDirectoryServer` son las dos únicas clases que utilizarán la clase `DirMessage`, la cual implementa los mensajes definidos en el protocolo de comunicación con el directorio diseñado por los alumnos.
    - `NFController` tiene un atributo `NFControllerLogicP2P` para procesar comandos que requieren de interacción con otros pares.
    - `NFControllerLogicP2P` utilizará un objeto `NFConnector` para llevar a cabo la descarga de ficheros de otros pares.
    - `NFConnector` es el cliente TCP encargado de implementar la comunicación con otros pares que están sirviendo ficheros, para solicitar y llevar a cabo descargas de ficheros.
    - `NFControllerLogicP2P` utilizará un objeto `NFServerSimple` para lanzar un servidor en primer plano.
    - `NFControllerLogicP2P` utilizará un objeto `NFServer` para lanzar un servidor de ficheros en segundo plano
    - `NFServer` posee los métodos para llevar a cabo la comunicación con un cliente, y deberá crear un hilo de la clase `NFServerThread` por cada cliente que se conecte al servidor.
    - `NFConnector` y `NFServer` son las dos únicas clases que utilizarán la clase `PeerMessage` que implementa los mensajes definidos en el protocolo de comunicación entre pares diseñado por los alumnos.

Descripción de paquetes y clases

El código proporcionado a los estudiantes está formado por los siguientes paquetes y clases:

Paquete `es.um.redes.nanoFiles.application`

- `NanoFiles.java`: Clase que contiene el método *main* del cliente. También incluye el atributo de clase público `db` que actúa como base de datos de ficheros que este *peer* puede compartir con el resto.
    - Esta clase ya está **completamente implementada**.
- `Directory.java`: Clase que contiene el método *main* del servidor de directorio. Se encarga de procesar los parámetros pasados al servidor de directorio por la línea de comandos, y se encarga de crear y lanzar el hilo de la clase `NFDirectoryServer` en el que se ejecuta el servidor de directorio.
    - Esta clase ya está **completamente implementada**.

Paquete `es.um.redes.nanoFiles.logic`

- `NFController.java`: Implementación del controlador que será encargado procesar los comandos introducidos por el usuario a través del *shell* y actuar consecuentemente en función del comando y del estado del autómata en el que nos encontremos. Para llevar a cabo las acciones necesarias por los

comandos que implican comunicarse con el directorio u otros *peers*, hace uso de la lógica implementada en las otras dos clases *controller* de este paquete.

  - Esta clase está **en gran parte implementada**, a falta de que los alumnos modelen los estados y transiciones según el autómata del protocolo diseñado.

- `NFControllerLogicDir.java`: En esta clase los alumnos deben implementar la lógica relacionada con las acciones que requieren comunicación con el directorio: *ping*, obtener y mostrar la lista de ficheros disponibles, etc. Para ello, los métodos de esta clase harán uso de un objeto de clase `DirectoryConnector`, que será quien finalmente se encargue de enviar y recibir datagramas al directorio.

  - La interfaz que esta clase expone a `NFController` ya está definida, mientras que la práctica totalidad de **sus métodos están sin implementar**. Como consecuencia de esto, en el código de partida, teclear cualquier comando en el *shell* que implique comunicación con el directorio no tiene ningún efecto.
  - Aunque no es estrictamente necesario, en función de la funcionalidad implementada por los alumnos (mejoras) puede ser necesario modificar la interfaz de la clase: añadir parámetros a los métodos definidos, definir métodos nuevos, etc.

- `NFControllerLogicP2P.java`: En esta clase los alumnos deben implementar la lógica de control para la comunicación con otros *peers*, ya sea como cliente o como servidor: ejecutar un servidor de ficheros, descargar un fichero de otros servidores, etc.

  - La interfaz que esta clase expone a `NFController` ya está definida. Sin embargo, **sus métodos están sin implementar**. Por consiguiente, los comandos del *shell* que implican comunicación con otros *peers* tampoco tienen efecto alguno.
  - Aunque no es estrictamente necesario, en función de la funcionalidad implementada por los alumnos (mejoras) puede ser necesario modificar la interfaz de la clase: añadir parámetros a los métodos definidos, definir métodos nuevos, etc.

Paquete `es.um.redes.nanoFiles.udp.client`

- `DirectoryConnector.java`: En esta clase los alumnos deberán llevar a cabo la implementación de un cliente UDP que se comunicará con un servidor (el programa *Directory*).
  - La interfaz que esta clase expone a `NFControllerLogicDir` ya está definida, si bien todos los métodos que llevan a cabo la comunicación con el directorio **están sin implementar**.
  - Los alumnos deberán implementar los métodos que llevan a cabo el envío al directorio de uno u otro tipo de mensaje, en función de los diferentes servicios que ofrece el directorio (*ping*, consulta de ficheros disponibles, etc.), y la recepción y procesamiento de las correspondientes respuestas por parte del directorio (extracción de los datos contenidos en la respuesta, etc.).
  - Para programar esta clase, se debe hacer uso de las clases `DirMessage` y `DirMessageOps`, las cuales modelan los mensajes diseñados por los propios alumnos para sus protocolos de comunicación con el directorio.

Paquete `es.um.redes.nanoFiles.udp.message`

- `DirMessage.java`: Clase que modela los mensajes del protocolo de comunicación con el directorio.
    - Su **implementación está únicamente esbozada**, incluyendo parte de la interfaz que se expone a `DirectoryConnector` para convertir mensajes de su formato de codificación concreto (textual) a objetos Java, y viceversa. También se incluyen constantes y atributos que pueden servir de guía para modelar todos los mensajes.
- `DirMessageOps.java`: Clase de apoyo a `DirMessage` para facilitar la implementación de los mensajes para la comunicación con el directorio.
    - En esta clase se deben definir los tipos de mensajes existentes en el protocolo diseñado.

Paquete `es.um.redes.nanoFiles.udp.server`

- `NFDirectoryServer.java`: Implementación incompleta de un servidor UDP que actúa como directorio.
    - Los alumnos **deben completar su implementación** para ser capaz de recibir datagramas y responder a los clientes en cada caso con el tipo de mensaje y datos adecuados, en función de la petición recibida.

Paquete `es.um.redes.nanoFiles.tcp.client`

- `NFConnector.java`: Debe contener la funcionalidad necesaria en el lado del cliente para permitir descargar ficheros de un servidor lanzado en otro *peer*, utilizando para ello sockets TCP.
    - Esta clase está **prácticamente sin implementar**.

Paquete `es.um.redes.nanoFiles.tcp.message`

- `PeerMessage.java`: Implementación esbozada de la clase que modela los mensajes del protocolo de comunicación entre *peers*.
    - Esta clase está **prácticamente sin implementar**.
- `PeerMessageOps.java`: Clase de apoyo que deberá contener la definición de las constantes que representan tipos de mensajes en este protocolo.

Paquete `es.um.redes.nanoFiles.tcp.server`

- `NFServer.java`: En esta clase se deberá implementar un servidor de ficheros que se ejecute en segundo plano. Se encarga de recibir los mensajes de solicitud de los *peer* clientes conectados al servidor, procesarlos y responder adecuadamente con uno u otro mensaje en función del tipo de solicitud.Para implementar un servidor que pueda generar hilos para atender múltiples clientes simultáneamente, deberá hacer uso de la clase `NFServerThread`.
    - Esta clase está **sin implementar**.
- `NFServerThread.java`: Clase auxiliar necesaria para que el servidor sea multi-hilo y pueda atender a varios clientes simultáneamente.

    - Esta clase está **sin implementar**.

Paquete `es.um.redes.nanoFiles.shell`

- `NFShell.java`: Implementación del shell conforme a lo establecido en el documento de prácticas.
    - Esta clase ya está **completamente implementada**.
    - Si bien esta clase ya soporta diversos comandos correspondientes a mejoras propuestas, también puede ser modificada para incorporar otros comandos relacionados con mejoras que no estén contempladas en el enunciado de prácticas.
- `NFCommands.java`: Clase de apoyo para facilitar la implementación del shell. Contiene la definición de los tipos de comandos y de los parámetros aceptados por los mismos.
    - Esta clase ya está **completamente implementada**
    - Puede ser modificada para incorporar mejoras.

Paquete `es.um.redes.nanoFiles.util`

- Clases de apoyo para facilitar la creación de la *base de datos* de ficheros compartidos por cada *peer*, así como la representación y manejo de metadatos de ficheros o el cálculo del *hash* a partir de su contenido.

Ejercicio a realizar: implementación de prueba de `ping`.

En el contexto de *NanoFiles*, hacer un "ping" significa contactar con el directorio (servidor UDP) para enviarle nuestro identificador de protocolo (un simple String único para cada grupo de prácticas definido en la clase `NanoFiles`), y recibir una confirmación de que el directorio está a la escucha y utiliza un protocolo compatible. Si el identificador de protocolo recibido por el directorio en un mensaje de *ping* concuerda con el suyo propio, el directorio deberá responder con un mensaje de confirmación dando la bienvenida al *peer*.

Vamos a empezar implementando la funcionalidad requerida por el comando `ping`, pero de momento usando mensajes sin formatear (con los datos "en crudo" en una simple cadena de caracteres).

1. Pon un punto de ruptura en el método `testCommunicationWithDirectory` de `NFControllerLogicDir`. Verás que, utilizando el atributo `DirectoryConnector`, se debería invocar exitosamente al método de prueba `testSendAndReceive`. Tras ello, se llama a `pingDirectoryRaw`. Tu trabajo en este ejercicio es conseguir que este último método realice las acciones requeridas.

2. Sigue los `TODO`'s para programar el método `pingDirectoryRaw` de la clase `DirectoryConnector`. La implementación será similar a la de `testSendAndReceive`, salvo que en este caso, enviaremos como dato adicional en el mensaje el *protocolID*. Por ahora, enviaremos como mensaje una cadena de caracteres sin formatear ("en crudo"):

    1. Debe enviar un datagrama con la cadena "ping&protocolID", donde el identificador de protocolo es una cadena definida en la clase `NanoFiles`.
    2. Debe usar el método `sendAndReceiveDatagrams` programado en el boletín anterior para enviar y recibir datagramas.

3. Debe comprobar que la respuesta recibida consiste en la cadena de caracteres "welcome".
4. Finalmente, debe informar por pantalla del éxito/fracaso de la operación.
5. Si se recibe otra respuesta distinta de "welcome", debe avisar del error por pantalla y devolver falso.

3. En el directorio, sigue los `TODO`'s para amplíar el código del método `sendResponseTestMode` (clase `NFDirectoryServer`). Si se recibe exactamente la cadena "ping", se trata de la prueba básica con `testSendAndReceive` y se debe devolver "pingok" (boletín anterior). Si en vez de esto, se recibe un mensaje del tipo "ping&protocolID" (prueba mediante `pingDirectoryRaw`), se debe responder bien con la cadena "welcome" (si el identificador de protocolo que contiene el mensaje coincide con el valor definido en la clase `NanoFiles`), bien con la cadena "denied". Si el mensaje recibido no comienza por "ping", debe devolver como respuesta un mensaje con la cadena "invalid".

4. Una vez implementado lo anterior, ya deberías poder hacer `ping` al directorio. Una vez superado el test de `pingDirectoryRaw`, debes desactivar el modo prueba. Esto te permitirá empezar a implementar y probar el resto de métodos cuya implementación está pendiente tanto en *NanoFiles* como en *Directory*. Para ello, tan sólo establece el valor del atributo `NanoFiles.testMode` a falso.