# Estructura del proyecto NanoFiles

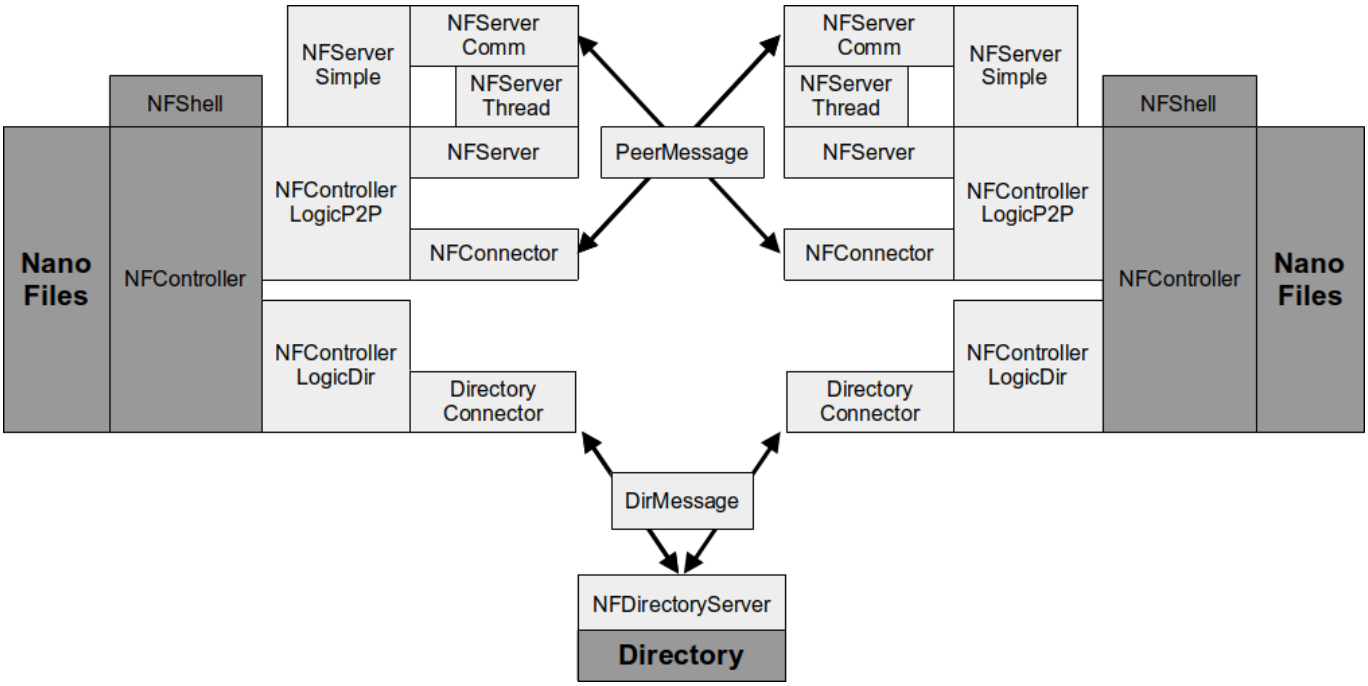
*Redes de Comunicaciones - Curso 2023/24*

## Introducción

El objetivo de este boletín de prácticas es describir los componentes que conforman el proyecto de Eclipse *nanoFilesP2P* que se proporciona a los estudiantes para desarrollar la práctica a partir de él. 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 interactuar mediante datagramas con del servidor de directorio de *NanoFiles*.
- `DirectoryConnector` y `NFDirectoryServer` son las dos únicas clases que utilizarán la clase `DirMessage` que 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 solicitar y descargar ficheros de otros pares que están sirviendo ficheros.
- `NFControllerLogicP2P` utilizará un objeto `NFServer` para lanzar un servidor de ficheros en segundo plano
- `NFControllerLogicP2P` utilizaŕa un objeto `NFServerSimple` para lanzar un servidor en primer plano.
- `NFServer` deberá crear un hilo de la clase `NFServerThread` por cada cliente que se conecte al servidor.
- `NFServerSimple` y `NFServerThread` deberán usar los métodos de `NFServerComm` para llevar a cabo la comunicación con un cliente, ya que son independientes el tipo de servidor implementado (primer o segundo plano, secuencial o paralelo).
- `NFConnector` y `NFServerComm` 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: iniciar sesión, cerrar sesión, obtener y mostrar la lista de usuarios o 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 en primer, crear un hilo servidor en segundo plano, descargar un fichero de uno o varios 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 de solicitud, en función de las diferentes funciones que ofrece el directorio (login, logout, consulta de ficheros disponibles, consulta de usuarios registrados, etc.) y la recepción de las correspondientes respuestas por parte del directorio, y su procesamiento (extracción de los datos contenidos en el mensajes, 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.

### 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`

- `NFServerSimple.java`: En esta clase se deberá implementar un servidor de ficheros minimalista que se ejecute **en primer plano** en cada *peer*.
    - Esta clase está **sin implementar**.
- `NFServer.java`: En esta clase se deberá implementar un servidor de ficheros que se ejecute en segundo plano. 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 que implementará el código a ejecutar por el hilo servidor encargado de atender la comunicación con un cliente ya conectado al servidor.
    - Esta clase está **sin implementar**.
- `NFServerComm.java`: Clase que 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. La funcionalidad que se proporcionará en esta clase (implementación del servidor de ficheros) es común e independiente del tipo de servidor implementado (primer o segundo plano, secuencial o paralelo).
    - 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.

## Familiarizarse con el código del controlador de *NanoFiles*

Tras completar los ejercicios del boletín anterior (Sockets UDP), deberías haber comprobado que la comunicación entre los programas *NanoFiles* y *Directory* se realiza correctamente. Una vez alcanzado este punto, vamos a continuar familiarizándonos con el código del proyecto:

1. Desactiva el modo de prueba, cambiando a falso el valor del *flag* `testMode` en la clase `NanoFiles`.
2. Establece un punto de ruptura al comienzo del método `main` de `NanoFiles.java` y otro al inicio del método `processCommand` de la clase `NFController` (paquete `es.um.redes.nanoFiles.logic`).
3. Depura el programa *NanoFiles* paso a paso con `Step over` (F6); observa el bucle principal de dicho programa e introduce el comando `login localhost alumno` en la consola. Observa, una vez detenido el programa en el segundo *breakpoint*, el valor que toma el atributo `currentCommand` y el valor de la constante `NFCommands.COM_LOGIN`. Recuerda que puedes ver el valor de atributos, constantes, etc. poniendo el puntero del ratón sobre su nombre en el código.

4. Verás que, una vez desactivado el *testMode*, la ejecución del programa llega a la línea donde se invoca el método `doLogin` de la clase `NFControllerDir`. Utiliza `Step into` (F5) para entrar dentro de dicho método.

5. Como ves, el método `doLogin` está sin implementar. Puedes continuar la ejecución (F8) e introducir los diferentes comandos soportados, empezando por `help`. Verás que, aunque los comandos se reconocen como correctos, no está implementada ninguna funcionalidad. Utiliza `Step into` para inspeccionar los diferentes métodos de `NFControllerDir` que se invocan según el comando tecleado, probando con todos los comandos disponibles.

## Ejercicio a realizar: implementación de `login`

En el contexto de *NanoFiles*, "iniciar sesión" significa contactar con el directorio (servidor UDP) para enviarle el nombre de usuario introducido por teclado en el comando `login`, y recibir una confirmación de que el registro ha sido exitoso.

- Si el *nickname* es un nombre válido (no duplicado), el directorio lo registrará en su lista de usuarios conectados, y nos deberá devolver un entero aleatorio entre 0 y 10000 que actuará como identificador de sesión (*sessionKey*).

- Este identificador de sesión será asimilable a una *contraseña* que el cliente UDP de *NanoFiles* deberá incluir en los sucesivos mensajes enviados al directorio como prueba de que ha iniciado sesión previamente.

- En todos los mensajes que el directorio recibe (excepto los de solicitud de *login*), siempre se deberá comprobar que la *sessionKey* aportada es válida y corresponde a un cliente registrado.

Una vez conocemos qué métodos de `NFControllerDir` se invocan en cada caso y las acciones que deben llevarse a cabo para iniciar sesión en el directorio, vamos a empezar implementando la funcionalidad requerida por el comando `login`.

1. Sigue el `TODO` del método `doLogin` de `NFControllerLogicDir`. Has de construir un objeto `DirectoryConnector` que será utilizado por otros métodos de `NFControllerLogicDir` en sucesivos comandos, ya que estos asumirán que se interacciona con el mismo directorio indicado por `login`. Debes capturar y tratar las excepciones que puedan ocurrir (informar del error, etc.). En particular, `doLogin` debe invocar el método `logIntoDirectory` de la clase `DirectoryConnector`, que es quien envía datagramas al directorio.

2. Sigue los `TODO`'s para programar el método `logIntoDirectory` de la clase `DirectoryConnector`. **NOTA**: No debes preocuparte de implementar la clase `DirMessage` (veremos como formatear los mensajes adecuadamente en el próximo boletín de prácticas), sino que por ahora enviaremos como mensaje una cadena de caracteres sin formatear ("en crudo").

    1. Debe enviar un datagrama con la cadena "login&nickname", donde *nickname* será el valor del parámetro pasado al comando `login` por el shell.
    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 "loginok&NUM", donde *NUM* será un número entero entre 0 y 1000 (es decir, la *sessionKey*).

    4. Si se recibe como respuesta "loginok&NUM", el método debe convertir *NUM* a entero, guardarlo en el atributo `sessionKey` (ya que este número debe enviarse al directorio en sucesivos mensajes. *Ayuda*: puedes usar los métodos `split` de la clase `String` y `parseInt` de `Integer`).
    5. Finalmente, debe informar por pantalla del éxito de la operación y la *sessionKey* obtenida.
    6. Si se recibe otra respuesta distinta de "loginok", debe avisar del error por pantalla y devolver falso.

3. Añade el código necesario en el directorio (clase `NFDirectoryServer`) para que, si recibe un mensaje con la cadena "login", envíe una respuesta con la cadena "loginok&NUM", donde NUM será un número aleatorio generado con `random.nextInt(10000)`. Cada vez que se reciba un mensaje, el directorio deberá informar por su consola (incluyendo si se trata de un mensaje con contenido inesperado).

4. Una vez implementado lo anterior, comprueba que puedes hacer `login localhost alumno` (con el mismo nombre de usuario) tantas veces como desees, ya que el directorio siempre responde afirmativamente, pero con un valor de *sessionKey* distinto.

5. Añade el código necesario en el directorio para comprobar que el *nickname* que se está registrando no existe previamente en la base de datos (`nicks`). Se enviará como respuesta un datagrama que contendrá la cadena "login_failed:-1" en caso de que el nombre de usuario esté duplicado.