Universidad Nacional de Educación a Distancia
E.T.S.I. Informática
Dpto. de Sistemas de Comunicación y Control
PRÁCTICA
DE
Sistemas Distribuidos
(Grado en Ingeniería Informática)

Sistema básico de almacenamiento en la nube usando Java RMI
Curso 2020 – 2021

Alumno: Roberto José de la Fuente López
DNI: 11806707W
rfuente4@alumno.uned.es
robertofl@aconute.es

PÁGINA INTENCIONADAMENTE EN BLANCO

Sumario

PÁGINA INTENCIONADAMENTE EN BLANCO

1.- ANÁLISIS FUNCIONAL

El propósito de la práctica es el desarrollo de un software que implemente un sistema de almacenamiento de ficheros en la nube usando Java RMI.

En este sistema actuaran tres tipos de actores cuyas funciones se listan a continuación:

1. Servidor: La entidad Servidor se encarga de controlar el proceso de almacenamiento de ficheros y de gestionar los recursos de almacenaje, para ello hace uso de tres servicios:

   - Servicio Autenticación: Se encarga de registrar y de autenticar, cuando sea necesario, las otras entidades participantes en el sistema: clientes y repositorios. Ambas entidades se tienen que dar de alta en el sistema y recibir un identificador único para poder operar y realizar almacenaje de ficheros. El registro se lleva a cabo cuando este servicio devuelve a la entidad demandante (cliente o repositorio) el identificador único con el que tiene que autenticarse para cualquier operación que lo requiera en el sistema.
     El identificador único estará formado por un nombre de usuario que elegirá el operador de la entidad correspondiente, acompañada por un desafío que será una contraseña, que también elige el operador.
     Además de esto, el servidor creará un identificador de sesión único, transparente al usuario, que servirá para autentificar las transacciones que se produzcan en el sistema.

   - Servicio Gestor: Este servicio se encarga de gestionar las operaciones de los clientes en relación con sus ficheros en la nube (físicamente alojados en los repositorios). Usando este servicio, el cliente podrá:
     - Subir, bajar y borrar ficheros en la nube
     - Listar sus ficheros almacenados
     - y compartir ficheros con otros clientes que estén registrados en el sistema.
       Un cliente podrá compartir cada uno de sus ficheros (de uno en uno) con el resto de clientes autentificados en el sistema.
   - Datos: Este servicio hará las funciones de una base de datos que relacione Clientes-Ficheros-Metadatos-Repositorios. Es decir, mantendrá lista de clientes y repositorios conectados al sistema, junto con los ficheros; y los relacionarán permitiendo operaciones de consulta,

borrado y añadido. Los dos servicios anteriores (Servicio Autenticación y Servicio Gestor) harán uso de este servicio para realizar las operaciones sobre el estado de las entidades del sistema y sus atributos.
En este primer prototipo, para este servicio, se ha implementado una base de datos es estructuras de datos en memoria, por lo que no es permanente, utilizando la clase Vector que implementa el interface List de Java.

2. Repositorios: Estas entidades son las responsables de guardar en sus dispositivos de almacenamiento los ficheros que los clientes suben a la nube. Cuando un cliente solicita la subida/bajada/borrado de un fichero a través del servicio Gestor del servidor, este servicio Gestor selecciona el repositorio que le corresponde al cliente y le manda al cliente la información necesaria para completar la operación. Para hacer su función, los repositorios hacen públicas las interfaces de sus dos servicios:
   - Servicio Cliente-Operador: Este servicio se encarga de las operaciones de subida de ficheros al repositorio y borrado de los mismos. El servicio Gestor del servidor responde a la petición del cliente enviándole la URL de este servicio para que pueda completar su operación.
   - Servicio Servidor-Operador: Este servicio tiene un doble objetivo. Por un lado, suministra los métodos necesarios para que el servidor gestione los lugares de almacenamiento para cada cliente y, por otro lado, se encarga de la operación de bajada de ficheros desde el repositorio al cliente, es decir, cuando un cliente quiere bajar un fichero se lo pide al servidor mediante el servicio Gestor. Una vez que el servidor averigua que repositorio aloja el fichero requerido por el cliente, éste llama a un método del Servicio Servidor-Operador y le pasa la URL del cliente para que pueda llamar al método de descarga del servicio DiscoCliente que es el que realmente se encarga de la descarga. De esta manera, no cargamos al servidor con esta operación temporalmente costosa en términos de entrada/salida y conseguimos Escalabilidad.
3. Clientes: Son los propietarios de los ficheros. Se registran en el sistema a través del servidor para poder subir, gestionar y almacenar sus ficheros en un repositorio en la nube. El cliente publica la interfaz de un servicio cuyo nombre es DiscoCliente que será utilizado por el servicio Servidor-Operador del repositorio para descargar al disco duro local del cliente el fichero que este considere oportuno.

Tal y como hemos indicado, el servidor **nunca se encarga de subir/bajar los ficheros**, sólo de gestionar estas operaciones y llevar un registro mediante su Servicio Datos. Estas operaciones costosas se harán desde los servicios Servidor-Operador y DiscoCliente para evitar cargar al servidor y permitir la escalabilidad del sistema fácilmente.

En este primer prototipo, el sistema **sólo admitirá la gestión de ficheros** y no la creación de un árbol de carpetas por cada cuenta de cliente en el repositorio, es decir, el cliente podrá subir ficheros, pero no crear carpetas donde guardarlos. **Cada repositorio creará una carpeta por cada cliente** que alojará **todos los ficheros** del mismo. (Las carpetas se crearán en el path donde se encuentre el ejecutable del repositorio).

Para poder ejecutar varios clientes o repositorios en la misma máquina, será necesario crear previamente una carpeta para los ficheros de los clientes y otra para los ficheros de los repositorios.

1.1.- Operativa

Inicialmente la entidad Servidor levanta sus tres servicios: Autenticación, Gestor y Datos.

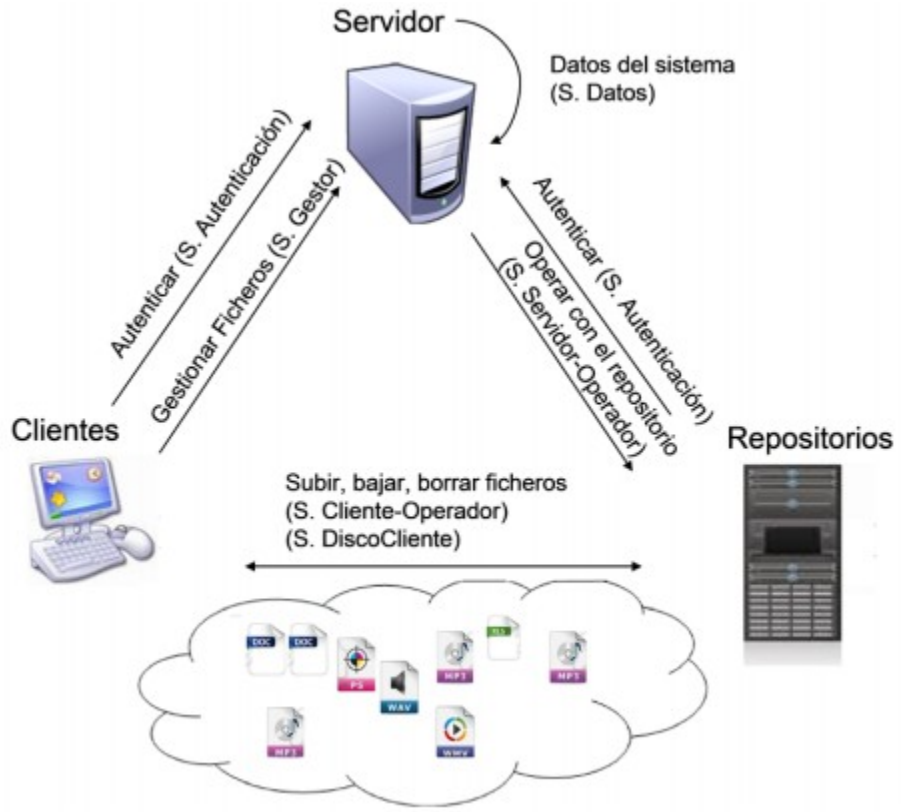
El/Los repositorio/s se autentican en el sistema mediante el servicio Autenticación del servidor, devolviéndole éste un identificador único. El repositorio a partir de este momento está listo para almacenar los ficheros de los clientes.

El/Los cliente/s se autentican en el sistema mediante el servicio Autenticación del servidor, devolviéndole éste un identificador único. En este momento, el servidor le asigna al cliente un repositorio y guarda esta relación a través del servicio Datos. Además, el servidor manda crear una carpeta en el dispositivo de almacenamiento del repositorio mediante el servicio Servidor-Operador. Esta carpeta tendrá por nombre el identificador único del cliente y dentro se alojarán todos sus ficheros en propiedad.

Una vez que el cliente está registrado en el sistema, podrá realizar las operaciones de subida/bajada/borrado de ficheros en la nube. Operaciones gestionadas por el servidor y ejecutadas en la carpeta que cada cliente tiene en el repositorio que le corresponde.

Además, el cliente puede pedir un listado de otros clientes conectados al sistema y permitirles el acceso para la descarga de sus ficheros “Compartir ficheros”.

En la figura siguiente podemos observar de forma esquemática la operativa del sistema:



1.2.-Interfaz

- El Servidor debe permitir mediante su interfaz las siguientes operaciones:

  1.- Listar Clientes.
  2.- Listar Repositorios.
  3.- Listar Parejas Repositorio-Cliente.
  4.- Salir.

- Los Repositorios deben permitir mediante su interfaz las siguientes operaciones:
  1.- Listar Clientes.
  2.- Listar ficheros del Cliente.
  3.- Salir.

- Los Clientes deben permitir mediante su interfaz las siguientes operaciones:

  1.- Subir fichero.
  2.- Bajar fichero.
  3.- Borrar fichero.

4.- Compartir fichero.
5.- Listar ficheros.
6.- Listar Clientes del sistema.
7.- Salir.

La operación "5.- Listar ficheros" debe mostrar tanto los ficheros en propiedad del cliente como los que puede obtener a través de la compartición que hacen otros clientes.

- Cuando arrancan las aplicaciones de los Clientes y los Repositorios debe aparecer inicialmente un menú con las siguientes opciones, antes de los menús anteriores. Éste debe permitir el registro de un nuevo usuario (cliente o repositorio según corresponda) en el sistema y/o autenticarse:

  1.- Registrar un nuevo usuario.
  2.- Autenticarse en el sistema (hacer login).
  3.- Salir

2.- ESPECIFICACIONES GENERALES

- Se debe utilizar código 100% Java JDK. No se admiten librerías de terceros.

- La sintaxis de llamada desde la línea de comandos tiene que ser:

  - servidor (para iniciar el programa Servidor y sus servicios).
  - repositorio (para iniciar un programa Repositorio y sus servicios).
  - cliente (para iniciar un programa Cliente y su servicio).

- Para cumplir con esta sintaxis de llamada hay que crear ficheros en batería (.bat o .sh) para arrancar las aplicaciones. Dentro de estos ficheros haya sólo una línea:
  java -jar nombre_fichero.jar.
  No se incluye dentro de los ficheros en batería órdenes para cambiar el path del sistema operativo, ni otra opciones de configuración específica de la máquina virtual.
- Por motivos de claridad, cada clase Java se almacena en un fichero .java
- Todos los ficheros están debidamente comentados, incluyendo al inicio de los mismos, todos los datos del autor: nombre, apellidos y correo electrónico.
- La aplicación final se presenta con el código encapsulado dentro de ficheros .jar.

3.- CONVENIOS DE DISEÑO

Con el objetivo de tener cierto orden, unificación y coherencia en el código que se entrega y para facilitar su posterior corrección, se nombran obligatoriamente las clases/interfaces principales del programa de la siguiente manera:

- Servidor: Clase que contiene el main de la entidad Servidor.
- Repositorio: Clase que contiene el main de la entidad Repositorio.
- Cliente: Clase que contiene el main de la entidad Cliente.
- ServicioAutenticacionInterface: contiene la interfaz remota del servicio de autenticación que depende de la entidad Servidor.
- ServicioAutenticacionImpl: clase que implementa la interfaz remota anterior.
- ServicioGestorInterface: contiene la interfaz remota del servicio Gestor que depende de la entidad Servidor.
- ServicioGestorImpl: clase que implementa la interfaz remota anterior.
- ServicioDatosInterface: contiene la interfaz remota del servicio Datos que depende de la entidad Servidor.
- ServicioDatosImpl: clase que implementa la interfaz remota anterior.
- ServicioSrOperadorInterface: contiene la interfaz remota del servicio Servidor-Operador que depende de la entidad Repositorio.
- ServicioSrOperadorImpl: clase que implementa la interfaz remota anterior.
- ServicioClOperadorInterface: contiene la interfaz remota del servicio Cliente-Operador que depende de la entidad Repositorio.
- ServicioClOperadorImpl: clase que implementa la interfaz remota anterior.
- ServicioDiscoClienteInterface: contiene la interfaz remota del servicio DiscoCliente que depende de la entidad Cliente.
- ServicioDiscoClienteImpl: clase que implementa la interfaz remota anterior.
- Fichero: clase Serializable que implementa las operaciones de lectura/escritura de ficheros en disco, junto con un sistema de comprobación por Checksum (suministrada por el equipo docente).

- Pueden tomarse las decisiones de diseño que se consideren oportunas, siempre y cuando vayan en consonancia con el enunciado y queden perfectamente comentadas.
- En cada entrada de datos no se hacen verificaciones del tipo ¿Está seguro (s/n)?

- Aparte de éstas clases, se pueden utilizar todas las clases que sean necesarias, pero no olvidar describir detalladamente su función, así como, su lugar en el diagrama de clases.

- Nombrar los objetos remotos usando una URL que recoja toda la información sobre su dirección-puerto, servicio que presta e identificador de su proveedor; como se hace en el libro de M. L. Liu:

  - URL_nombre="rmi://"+ip+":"+rmiport+"/"+nombre_servicio+"/"+identificador_unico;
  - Naming.rebind(URL_nombre, objExportado);

- Crear una clase metadatos e instanciar un nuevo objeto de este tipo por cada fichero nuevo que se suba al sistema. Esta clase debe tener varios campos que identifique al fichero como su nombre, identificador único de su propietario, etc.

- No cargar rmiregistry desde la línea de comandos. Crear una instancia del mismo mediante:

  - LocateRegistry.createRegistry(port)

4.- ENTORNO DE DESARROLLO

Plataforma: Se ha utilizado tanto Windows 10 Educational como Linux Debian Buster

IDE: Se ha utilizado Eclipse versión 2020-09, tanto en Windows como en Linux. También se ha utilizado para pruebas Notepad++.

JVM: Para compilación y ejecución se ha utilizado OpenJDK-11.0.2 (tanto el Windows como en Linux).

Confección de la memoria: LibreOffice 7. Para la captura del diagrama E-R se ha utilizado LibreBase y para la captura del diagrama de clases se ha utilizado Bluej 4.22 (los plugin de Eclipse no me han funcionado adecuadamente).

El motivo de utilizar los dos entornos ha sido por movilidad, no teniendo una máquina fija en la que desarrollar la práctica.

5.- DISEÑO

5.1.- GENERAL

Para el desarrollo de la aplicación, se ha desarrollado dos paquetes: servidor y usuario.

El paquete servidor contiene todo los necesario para ejecutarlo tal como se establece en el enunciado de la práctica.

Para repositorio y cliente he decidido crear un solo paquete por mantenimiento de la aplicación: la mayor parte del código de las dos entidades es repetido, por lo que la parte común se ha implementado en la clase Usuario (acceso al registro RMI, menú de usuario y autenticación), y cada parte específica se ha implementado, respectivamente en las clases UsrRepositorio y UsrCliente.

Para la ejecución, dado que Usuario necesita parámetros de entrada y en el enunciado se pide un archivo de lotes sin ellos, se han creado dos clases cada una con método main, Cliente y Repositorio, que configuran adecuadamente el acceso a la clase Usuario.

La práctica se entrega compilada para la utilización de todos los usuarios y servidor en el mismo host. Para ejecutar los clientes y repositorios desde otro host, es necesario modificar, en la clase Usuario, en las constantes de clase, las líneas correspondientes a la IP del registro y el puerto de escucha

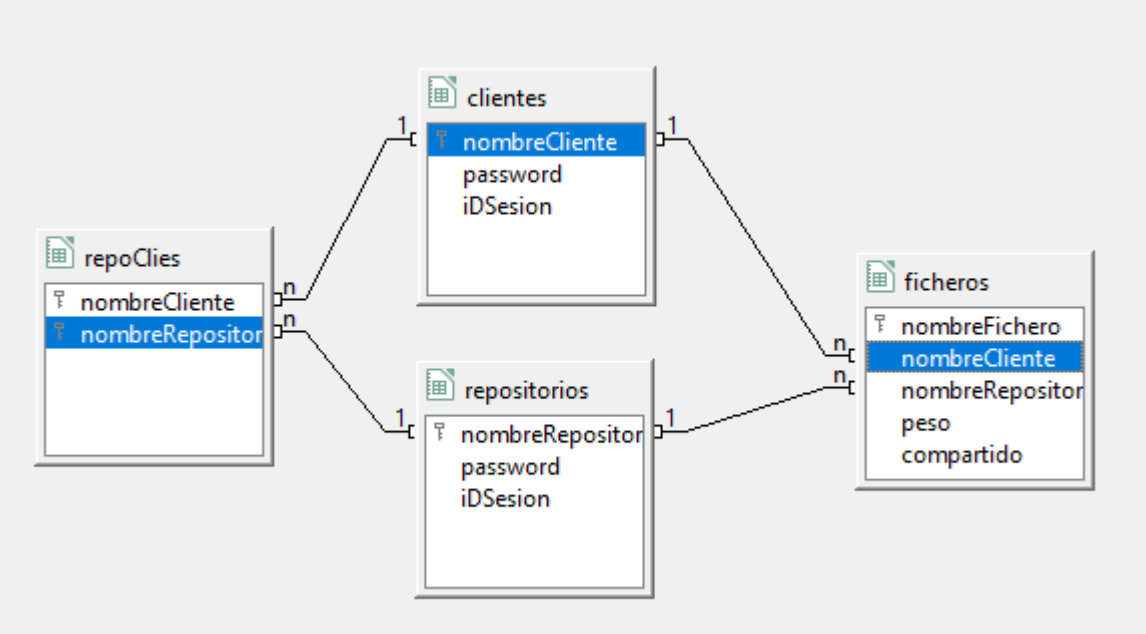
```
private static String iPHostRegistroRMI=NubeRMI.getIPServidorRMI();
private static Integer puertoServidorRMI=NubeRMI.getPuertoServidorRMI();
```

sustituyendo los datos obtenidos de NubeRMI por los datos reales donde se vaya a ubicar el servidor (en entorno de pruebas así se ha hecho).

Todas las clases están debidamente comentadas y documentadas en la carpeta doc

5.2.- MODELO DE DATOS

En el diagrama E-R a continuación podemos ver las tablas y relaciones que se establecen entre los distintos campos.



Donde:

- Clientes.- Es la tabla que almacenará todos los clientes registrados en el sistema. El nombreCliente y la contraseña será el identificador único que necesita el sistema para que los clientes accedan al sistema. Con el campo iDSesion se ha implementado un sistema de seguridad para la sesión de cada usuario. Cuando este está autentificado en el sistema, contendrá una cadena alfanumérica aleatoria única que autenficará todas las operaciones del usuario es una cadena alfanumérica que será distinta en cada sesión o la cadena vacía si no está autentificado.

- Repositorios.- Es la tabla que almacenará todos los repositorios dados de alta en el sistema. Como en el caso anterior, nombreRepositorio y contraseña son el identificador único que permite autentificar a un usuario en el sistema y el campo iDSesion es una cadena alfanumérica de autentificación de la sesión.

- RepoClies.- Esta tabla contiene la asociación de los Clientes a los repositorios que utilizan. La práctica que se presenta tiene restringido el número de repositorios de un cliente a uno solo (siempre el mismo). En este caso la tabla no es necesaria, pues podría haberse sustituido con un campo en la tabla Clientes, pero se presenta de cara al mantenimiento: al estar así implementado, la supresión de la restricción antes indicada solo requerirá pequeños cambios en las consultas y operaciones.

- Ficheros.- Esta tabla es la información útil de todos los ficheros subidos por los clientes a sus repositorios. Contiene el nombre del mismo, el propietario, su tamaño y si este está compartido con el resto de usuarios. Es una tabla de metadatos que evita tener que acceder al sistema de archivos cada ver que el sistema quiera conocer los ficheros de un repositorio o de un cliente, que es donde realmente están los ficheros.

Tal como se recomienda en el enunciado, la base de datos se implementa como una estructura de datos en memoria.

Las clases **ClienteBD, RepositorioBD, RepoClieBD y FicheroBD** representan cada uno de los registros de las correspondientes tablas antes descritas.

Se ha determinado implementar las tablas en la clase **Datos, así como todas las consultas relativas a las mismas**, y así no mezclar el acceso a datos con el Servicio de Datos que implementa el servidor. De esta forma, aunque complicamos un poco el esquema de interacción al añadir una llamada más, es más fácil de mantener: en el momento que se determine migrar a un SGBD conectado, por ejemplo, con JDBC, solo hay que tocar la lógica de acceso a datos, no siendo necesario tocar el servicio de datos.

Para implementar las tablas de la base de datos, he elegido la clase Vector, que implementa la interface List, por ser una clase sincronizada, cosa que no ocurre con hashMap. Esto ha sido así para proteger la integridad de los datos: la base de datos se va actualizar concurrentemente, luego necesitamos una estructura que sea thread safe. Además, para ello se ha utilizado el mecanismo simple de Java para la exclusión mutua, marcando todos los métodos como synchronized.

Para obtener el resultado de las distintas consultas con listados de registros, se ha implementado de forma simple con String. Aquí es válido por ser listados muy cortos,pero lo ideal sería almacenarlos en ficheros temporales y estos tratarlos en la visualización de forma adecuada.

5.3.- RMI

Para la implementación de esta aplicación distribuida se ha decidido crear una clase estática MiRegistroRMI, que abstrae toda la lógica relativa a la ejecución del registro RMI, registro y desregistro, así como referencias, de los objetos remotos (servicio).

La motivación principal de esta clase es descargar al resto del programa del trato con RMI. Incluye dos métodos para registrar y desregistrar objetos remotos. Hay que tener presente que estos solo funcionarán si se ejecutan en el mismo host en el que se ejecuta el servicio de registro, pues por seguridad, el propio middleware no lo permite.

Para ejecutar/obtener el registro RMI se ejecutará el método

refAMiRegRMI(IPHostRegistro,numPuertoRMI)

donde IP y puerto son opcionales (tiene los constructores necesarios para poder indicar solo IP, solo puerto, los dos o ninguno).

Si se ejecuta en el host que hará de servidor (sin parámetros o con la misma dirección IP que el host en el que se ejecuta), permite la creación de un servicio de registro RMI en el puerto por defecto (o el indicado). Si ya se encuentra en ejecución en este host, simplemente crea la referencia al registro y la guarda.

Ejecutada desde otros hosts es necesario indicarle la dirección IP y el puerto de escucha para que pueda referenciar el servicio de registro en ese otro host (así se utiliza en la clase Usuario). Si no hay servicio en la dirección:puerto indicadas, devolverá null, es decir, si registrRMI.isNull()==true, no hay servicio levantado.

También construye la URL valida de acceso tanto a él mismo como a los objetos remotos en él registrados (de la forma rmi://host:puerto/...)

El registro y desregistro de objetos remotos se realiza desde la referencia al registro, no con la URL. Esto facilita el mantenimiento del programa, dado que el usuario de esta clase solo tiene que facilitar el bind del objeto, no teniendo por qué saber el resto de la ruta al registro RMI, abstraída en esta clase.

Por defecto utiliza la IP de localhost (en esta versión tiene la de la primera interfaz de red que tenga registrada el sistema operativo) y el puerto predeterminado del servicio RMI.

Para complementar a esta clase reutilizable de servicio RMI se añade la clase NubeRMI, que contiene la configuración de nuestra aplicación distribuida: la IP y el puerto de nuestro registro RMI, que se ejecutará en el mismo host que "servidor" y todos los bind de cada uno de los servicios que se van a implementar en cada una de las entidades.

En esta clase se ha implementado un log de interacción con el registro, pero está en estado OFF, sólo activable en tiempo de compilación.

5.4.- GUI

Para cada una de las entidades se ha creado una interface con todos los métodos que necesita la aplicación para visualizar los mensajes y listados.

Esta interface nos facilita el mantenimiento de la aplicación: para esta práctica se ha implementado el entorno de usuario en modo texto (por consola), pero la implementación del entorno de ventanas es inmediata por medio de esta interfaz.

Esto hace que no hayamos implementado la clase como estática. En en caso de Cliente y Repositorio, además, se hace necesario para obtener los menús adecuados.

Para la visualización de los distintos listados se obtiene un String desde la base de datos. Tal como se ha indicado antes, esto solo es válido en el caso de esta práctica, en que hay pocos datos.

Para las pantallas de la entidad Cliente y Repositorio, se ha implementado la misma estructura que para las clases del programa: GUIUsuarioInterface que contienen el menú principal, mensajes de autenticación y de error generales. Los menus específicos de cada una de las entidades se desarrollan en GUIClienteInterface y en GUIRepositorioInterface.

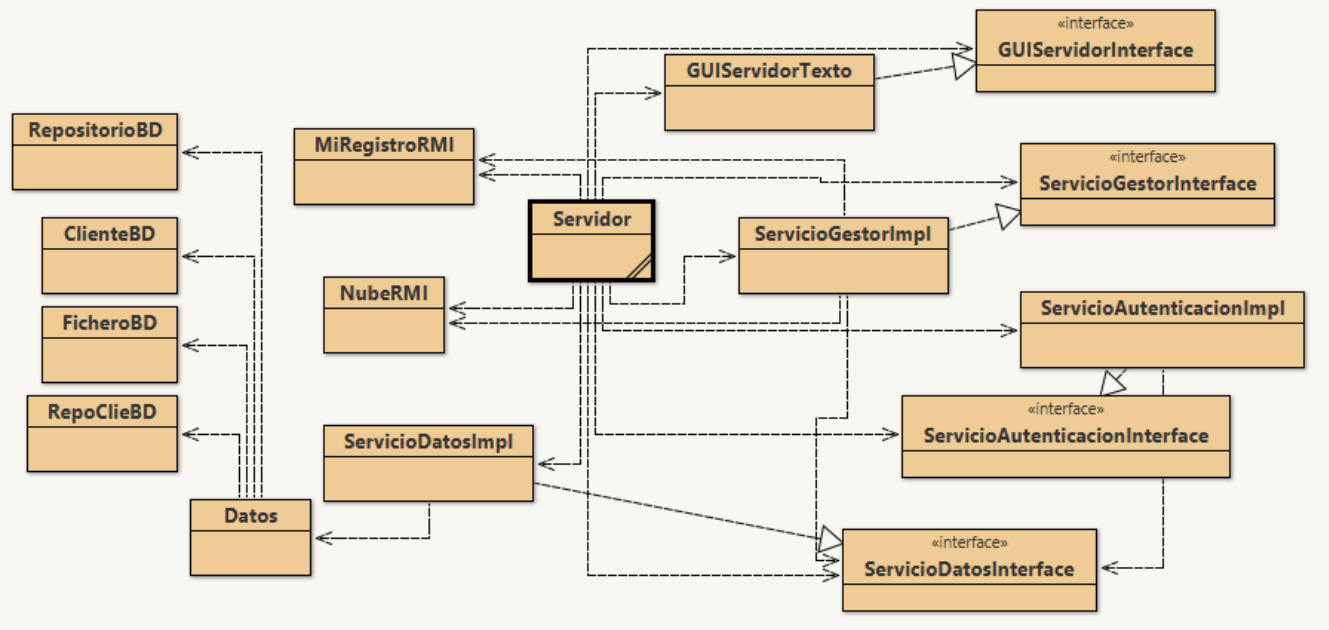
5.5.- ENTIDAD SERVIDOR

El servidor inicia sesión en un host, comprueba si hay un registro activo en el puerto indicado. Si es así obtiene su referencia así como su URL; en caso contrario ejecuta un servicio de registro en el puerto por defecto.

Una vez establecido el servicio de registro, publica los servicios de datos (este solo lo llaman los otros dos), autenticación y gestor en ese mismo host (no permite obtener datos registrados de otro host).

Es importante tener en cuenta que el servicio gestor se utilizará tanto para acceder al servicio de datos como al registro RMI: los usuarios (repositorios y clientes) no pueden hacer bind desde un host distinto, por lo que se implementan métodos para hacerlo desde este servicio.

Queda permanente el menú de interacción del servidor, hasta que se cierra el programa, momento en el que se desregistran los servicios en el registro.



5.6.- ENTIDAD REPOSITORIO

Lo primero que hace es comprobar que hay un registro RMI activo en la IP establecida. Si no es así, no inicia el programa.

Una vez establecida la comunicación con el registro, obtiene las referencias a los servicios de servidor autenticación y gestor.

Inicia el menú de usuario para el alta o autentificación de repositorio. Para estas operaciones utiliza el servicio autenticación del servidor.

Una vez autentificado, se registran los servicios Sr y Cl del repositorio

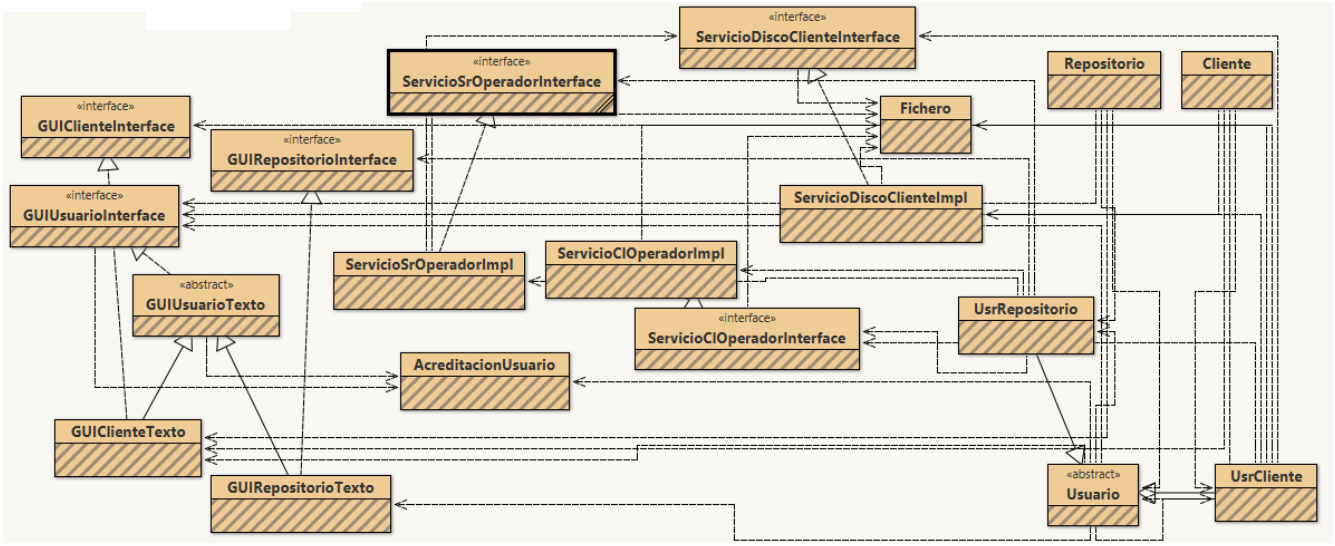
Todo este trabajo lo realiza la clase Usuario, común a repositorio y cliente.

Una vez autentificado en el sistema llama al método abstracto menuUsuarioAcreditado, que implementa UsrRepositorio, donde está toda la operativa correspondiente a los repositorios (listados). Para ello utiliza el servicio gestor, que es el que se encarga de comunicar.

La comunicación con la base de datos se hace con el servicio gestor. El servidor se comunica con el repositorio con el servicio Sr.

Queda visualizando el menú de administración del repositorio, hasta que se cierra la sesión, momento en el que se quitan del registro los servicios Sr y Cl de este repositorio.

Cuando se cierra el programa simplemente emite un mensaje de fin del mismo.



5.7. CLIENTE

El alta y autentificación de clientes es común con repositorio, como ya se ha comentado, con la clase Usuario.

Una vez autentificado se registra el servicio DiscoCliente de esta y se visualiza el menú de interacción del usuario (se describe su uso en el modelo de interacción).

La comunicación con la base de datos se realiza a través del servicio gestor y la comuncación con el repositorio asignado con el servicio Cl del repositorio.

Este servicio utilizará el servicio DiscoCliente para realizar la subida/bajada de ficheros (entre repositorio y cliente, solo informando del hecho a servidor).

Una vez finalizada la sesión del cliente, se da de baja del registro el servicio DiscoCliente de este usuario.

5.8.- MODELO DE INTERACCIÓN

El modelo de interacción está determinado por el enunciado de la práctica (punto 1 de este documento, con sus añadidos correspondientes).

No he cursado Diseño del Software, por lo que no sé poner los diagramas de interacción correctamente.

Detalles a tener en cuenta:

- Solo se permite la ejecución de un servidor en el mismo host
- No se permite iniciar sesión a un usuario (cliente o repositorio) si la sesión ya estaba activa (determinada por el ID de sesión).
- Los clientes se asocian siempre con el mismo repositorio, (inicialmente asignado aleatoriamente de entre los activos en el momento de autentificarse en el sistema). Si este repositorio no está en ejecución en el sistema (autentificado en servidor), el cliente no puede abrir sesión.
- En las operaciones con ficheros, no se permiten rutas.
- Los ficheros a subir a los repositorios deben estar en la misma carpeta donde se ejecuta el programa. En las bajadas, se da por supuesta la carpeta correspondiente al cliente.
- Los ficheros se comparten de uno en uno por el propietario del mismo.
- A la hora de bajar los ficheros, nos va a pedir siempre si es propio o compartido por otros. A efectos del usuario es transparente, pero en el programa nos permite establecer el repositorio donde está el fichero a bajar, si es que está autentificado.

- Los ficheros que se suben, se guardan en la ruta relativa (a partir de la de ejecución del programa)

  reposFiles/nombreRepositorio/nombreCliente

  Esta carpeta es creada cuando el usuario se autentifica por primera vez

- Los ficheros que se bajan, se guardan en la ruta relativa

  clientesFiles/nombreCliente.

- Las carpetas reposFiles y clientesFiles tienen como misión evitar que se mezclen ficheros de varios usuarios cuando los programas se ejecutan de forma concurrente desde la misma ubicación (como es el caso de esta práctica).

5.9.- MODELO DE SEGURIDAD

El modelo de errores implementado es muy simple: solo se han implementado controles para los inicios de sesión:

- El servidor comprueba que no hay registro RMI, cuyo caso lo levanta, pero no permite dos instancias del mismo en el mismo host.

- El repositorio y cliente comprueba que el registro está levantando, sino no inicia ejecución.

- El repositorio y el cliente no inician sesión si ya está ya una iniciada (no permite hacer bind dos veces).

- El cliente comprueba que el repositorio asignado tiene los servicios registrados.

Una vez establecidas las sesiones, si se produce un error de acceso a objetos remotos (por ejemplo, que se apague un repositorio y la sesión de sus clientes se quede abierta), se captura el error, se informa al usuario del error y se finaliza la ejecución del cliente.

Obviamente, deberíamos establecer controles en cualquier operación que requiera acceder a los objetos remotos, para así recuperarnos del error:

- Hacer varios intentos de conexión en caso de que sean por timeout.

- Dejar la aplicación en un estado que pueda cerrar correctamente la sesión, dependiendo del error, claro

5.10.- SEGURIDAD JAVA

No se ha implementado Security Manager (lo he intentado pero no he podido implementarlo de forma que funcione; se ha quedado como comentarios en el código).

Para una aplicación real, sería conveniente establecer unos permisos adecuados en fichero policy para prevenir accesos no deseados, así como la descarga dinámica de clases.

6.- EJECUCIÓN DE LA APLICACIÓN

Se presentan tres archivos jar: servidor.jar , repositorio.jar y cliente.jar

Se requerirá una JVM de la versión 11 o superior para su ejecución. Suponiendo que se ha establecido el path al JDK (JVM), se ejecutarán:

java -jar servidor.jar (una sola instancia)
java -jar repositorio.jar (los que se necesiten de forma concurrente)
java -jar cliente.jar (los que se necesiten de forma concurrente)

No obstante, se suministra los archivos por lotes (.bat) para su ejecución enWindows.

Como se suministra una carpeta con los fuentes, javadoc y los ejecutables, también se podría ejecutardirectamente :

java -cp enube2\bin servidor.Servidor
java -cp enube2\bin usuario.Repositorio
java -cp enube2\bin usuario.Cliente

Tener en cuenta que aquí los ficheros a subir debería estar dentro de la carpeta enube2

7.- ENTORNO DE PRUEBAS

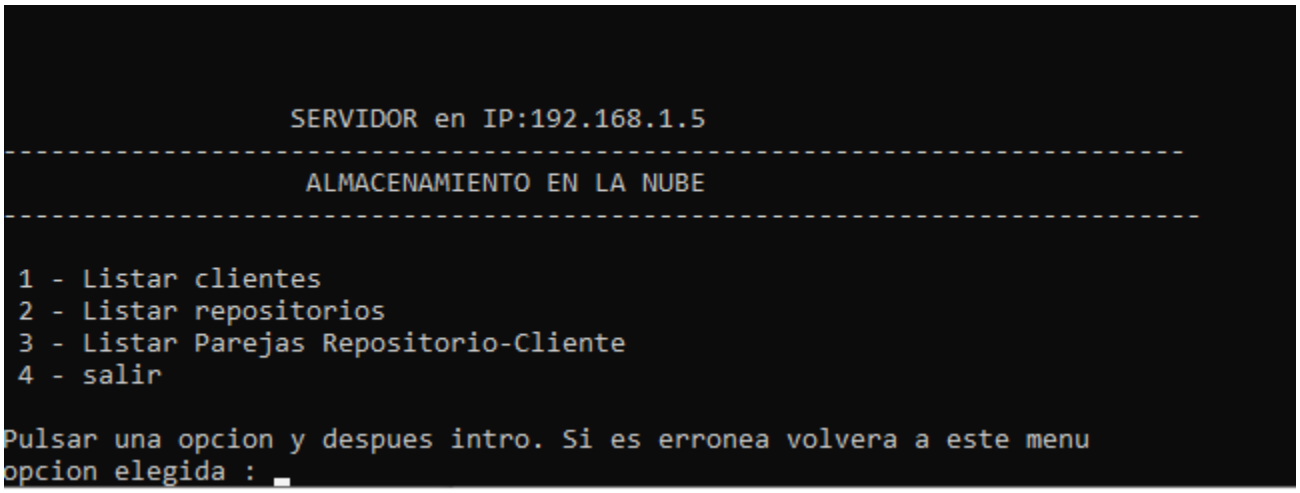
Se realizan las pruebas con dos ordenadores (máquinas físicas, pues las virtuales daban problemas a la hora de obtener el resultado de INetAddress) con Windows 10 Pro. El cortafuegos es mejor que esté deshabilitado para evitar problemas (no obstante, durante la primera ejecución, java lanza un mensaje para configurarlo y no debería dar ningún problema).

IP del servidor : 192.168.1.5
IP del otro ordenador : 192.168.1.34 (por DHCP).

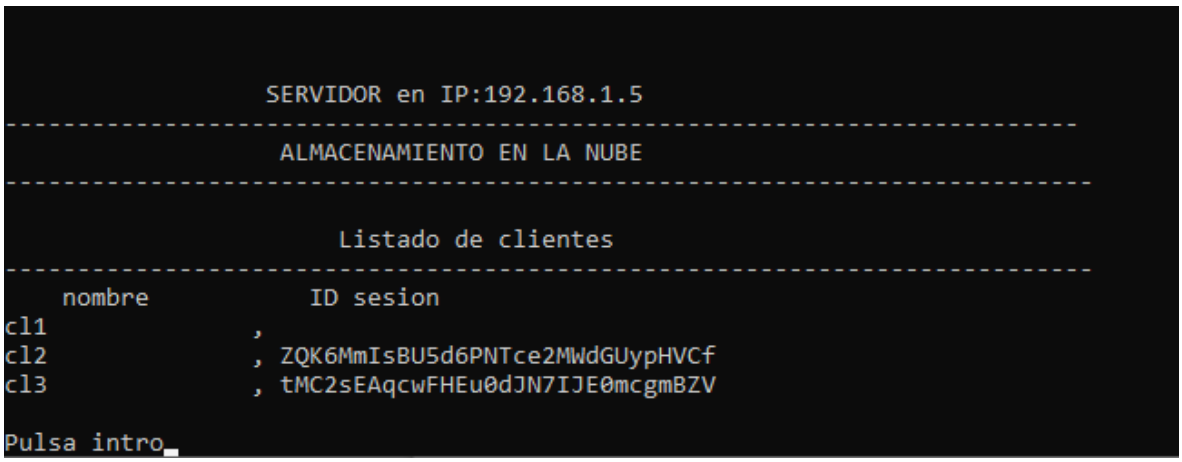
Todas las capturas de pantalla se realizan en una misma sesión, dado que si se cierra el servidor, se pierden todas las interacciones ocurridas.

Conexión inalámbrica

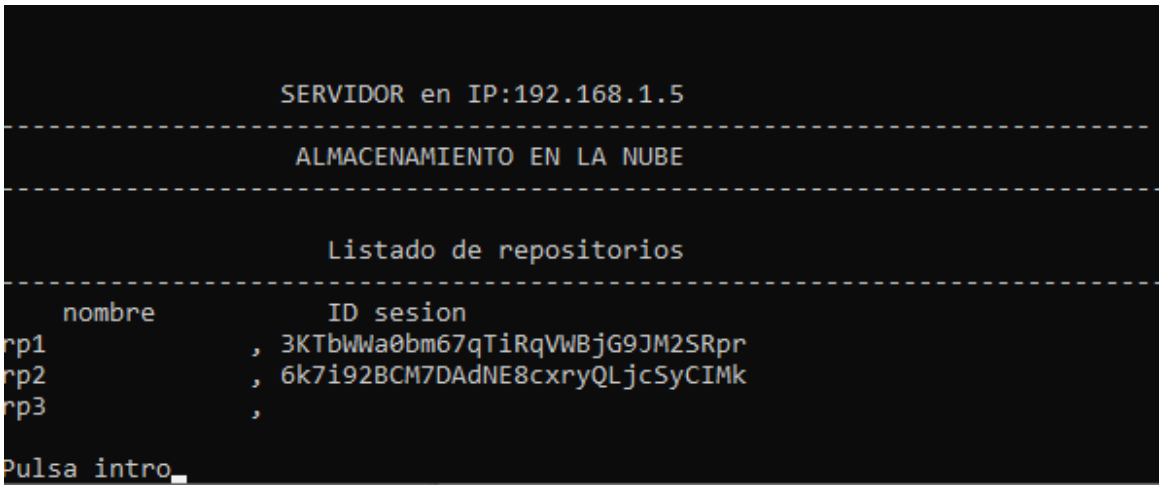
7.1. Servidor



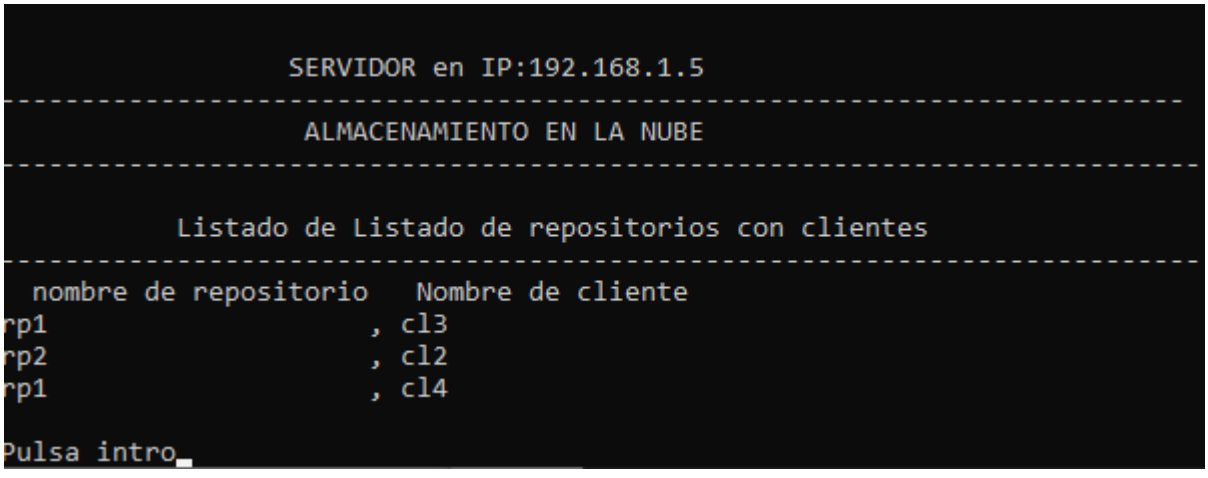
Listado de clientes después de varias interacciones



Listado de repositorios después de varias interacciones

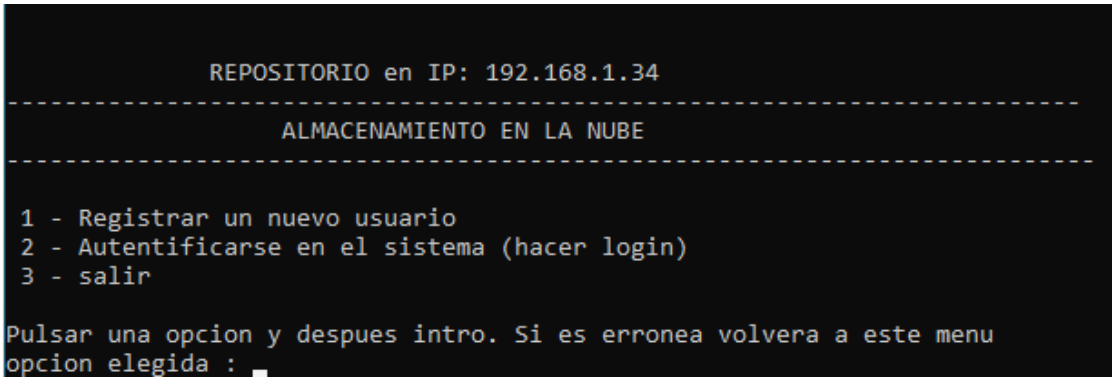


Listado de clientes y los repositorios asignados después de varias interacciones

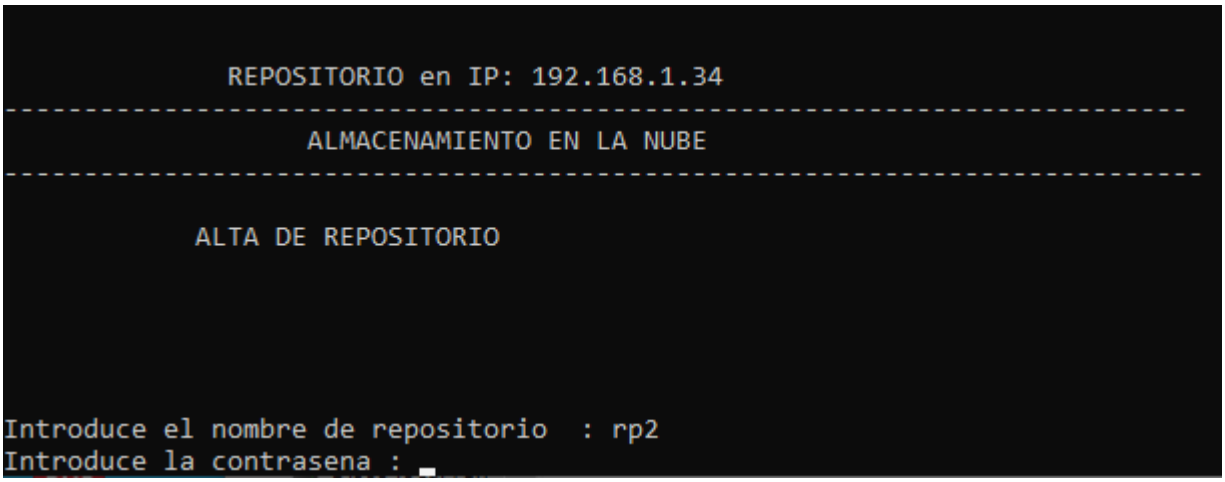


7.2.- Repositorio

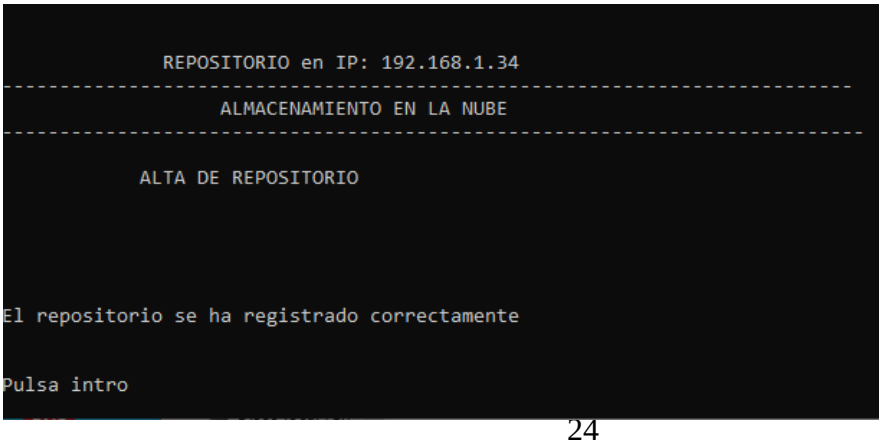
Pantalla inicial de repositorio.



1.- Registro de usuario

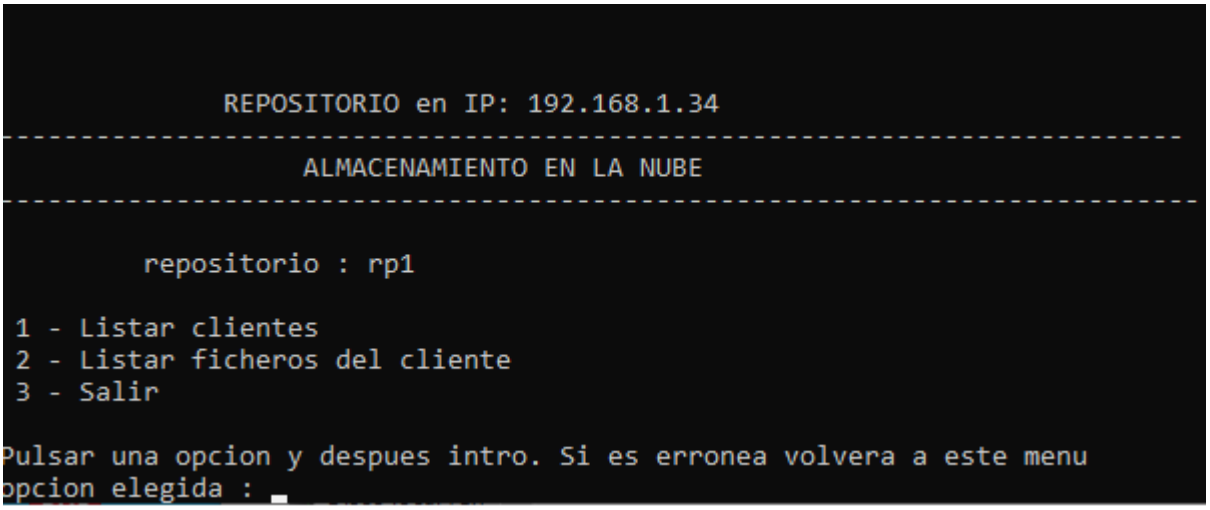


Alta de repositorio correcta

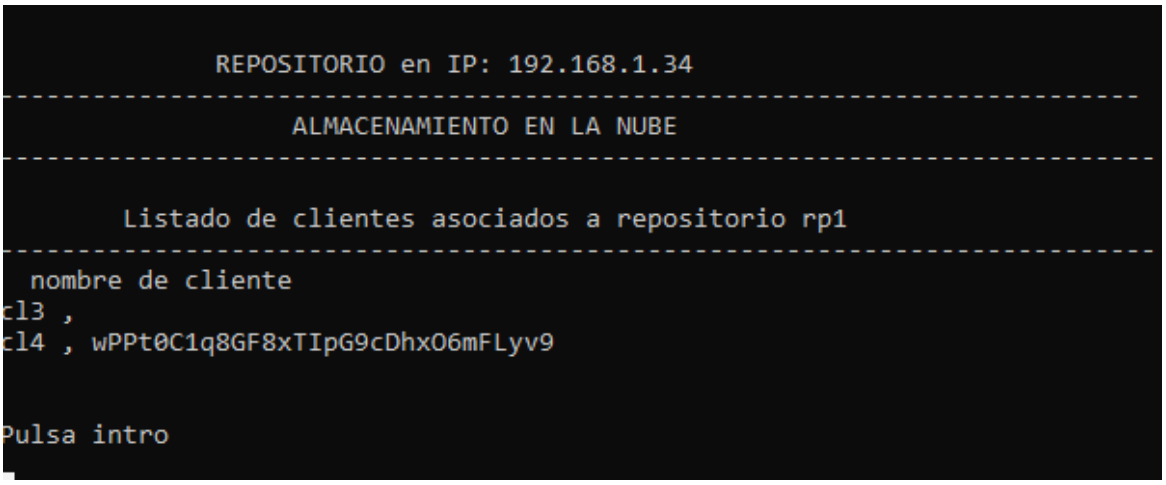


No se puede dar de alta otro usuario

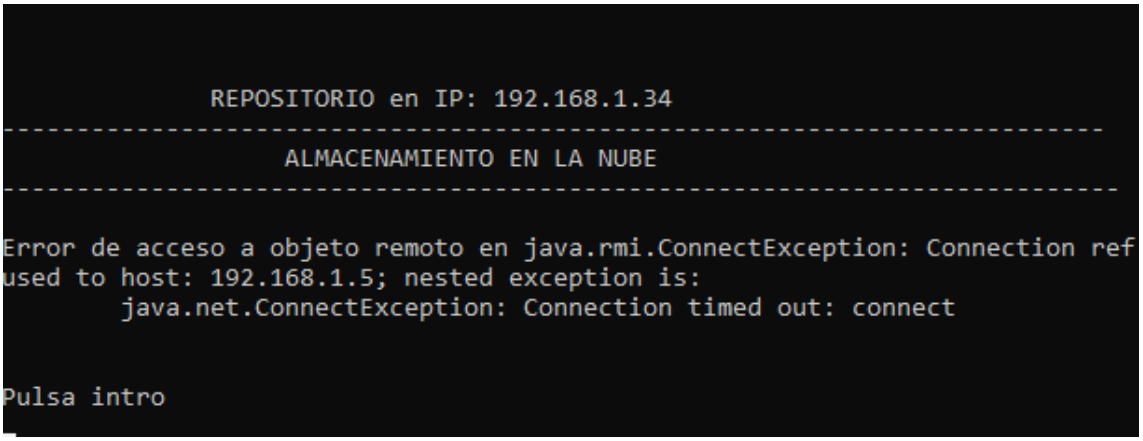
Menú de repositorio autentificado



1.- Listar clientes (del repositorio)



Error en repositorio por fallo de servidor (se prueba a cerrar el proceso)



Listado de ficheros de un cliente en un repositorio

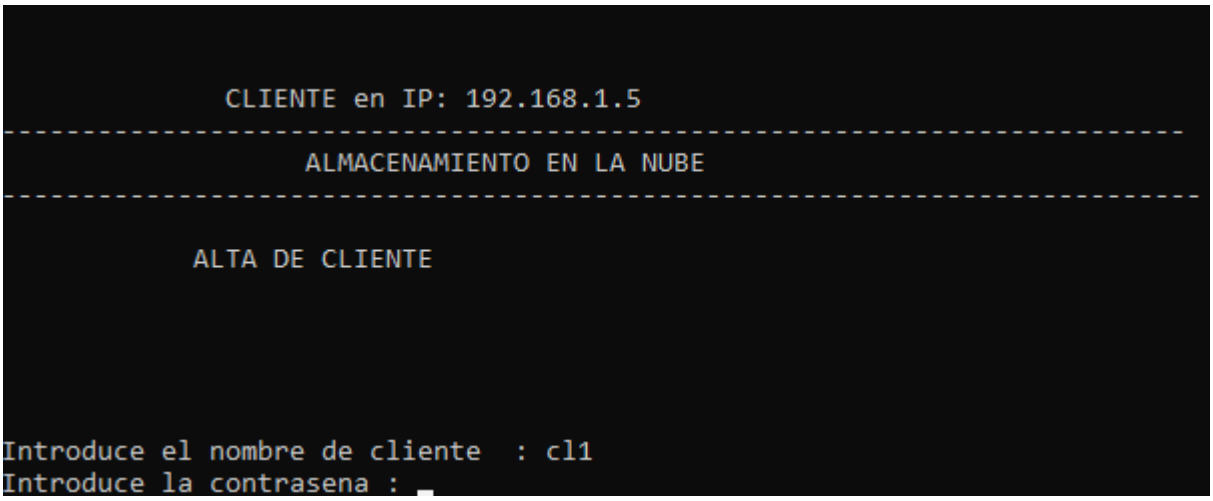
```
            REPOSITORIO en IP: 192.168.1.5
-------------------------------------------------------------------------
                  ALMACENAMIENTO EN LA NUBE
-------------------------------------------------------------------------


                Repositorio : rp2
          Listado de ficheros del cliente cl2
-------------------------------------------------------------------------

  nombre de fichero
Fichero : fotos.txt en  repositorio rp2 compartido por cl2


Pulsa intro
```

7.3.- Cliente

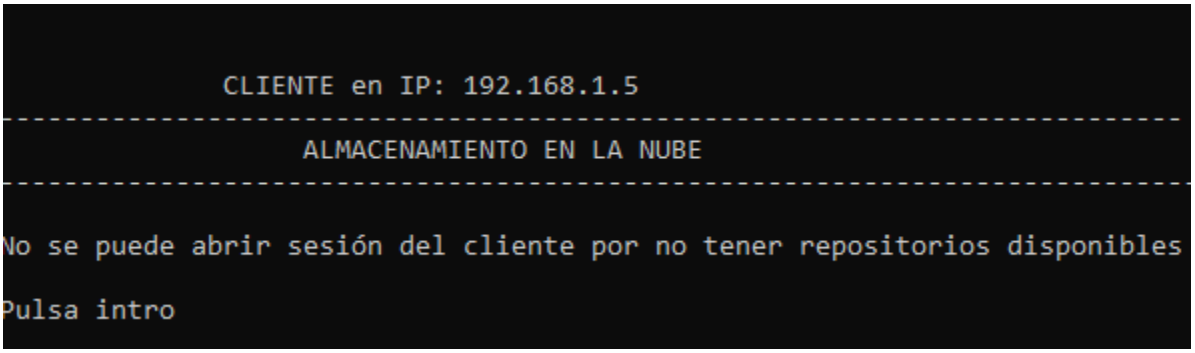
Pantalla inicial

```
            CLIENTE en IP: 192.168.1.5
-------------------------------------------------------------------------
                  ALMACENAMIENTO EN LA NUBE
-------------------------------------------------------------------------

 1 - Registrar un nuevo usuario
 2 - Autentificarse en el sistema (hacer login)
 3 - salir

Pulsar una opcion y despues intro. Si es erronea volvera a este menu
opcion elegida :
```

Las pantallas de alta de cliente, cliente dado de alta satisfactoriamente y error de alta son iguales a las de repositorio, salvo que indica que se trata de cliente.



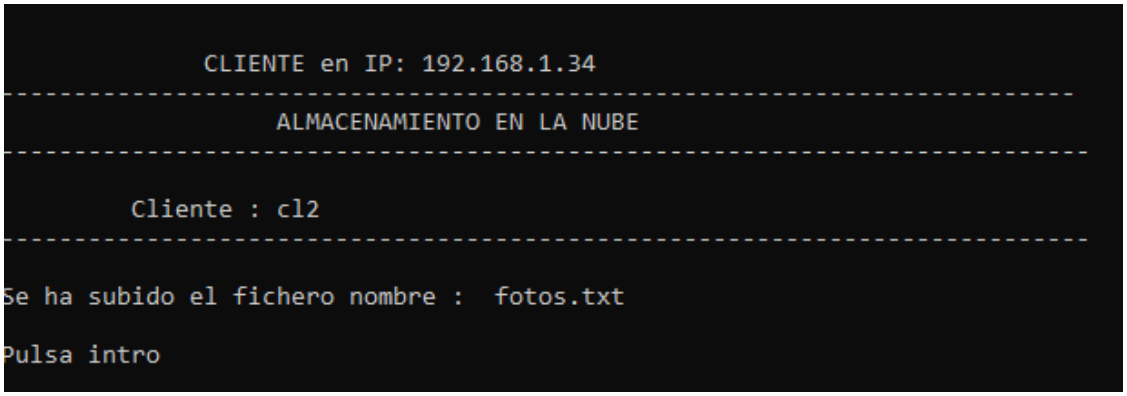
Si el repositorio asignado no está disponible, no abre sesión



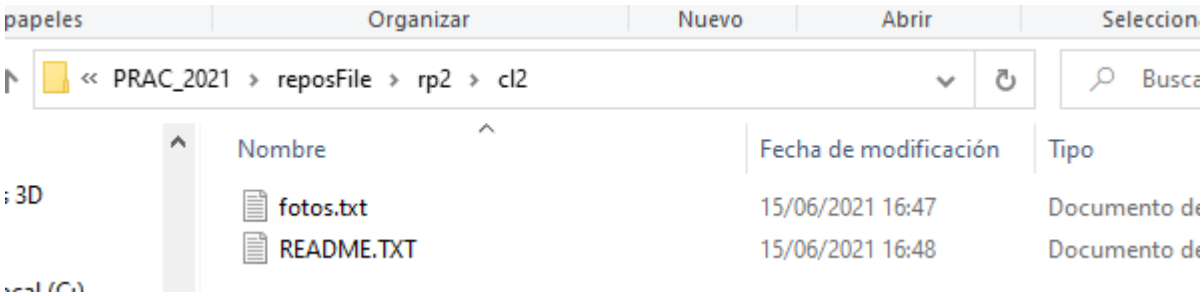
Para las pruebas de ficheros se ha realizado con la siguiente configuración

Repositorio rp1 en IP 192.168.1.34
repositorio rp2 en IP 192.168.1.5
cliente cl2 en IP 192.168.34
cliente cl4 en IP 192.168.1.5

1.- Subir fichero

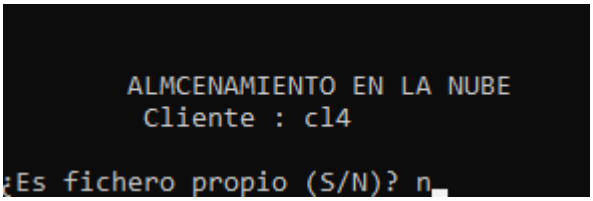


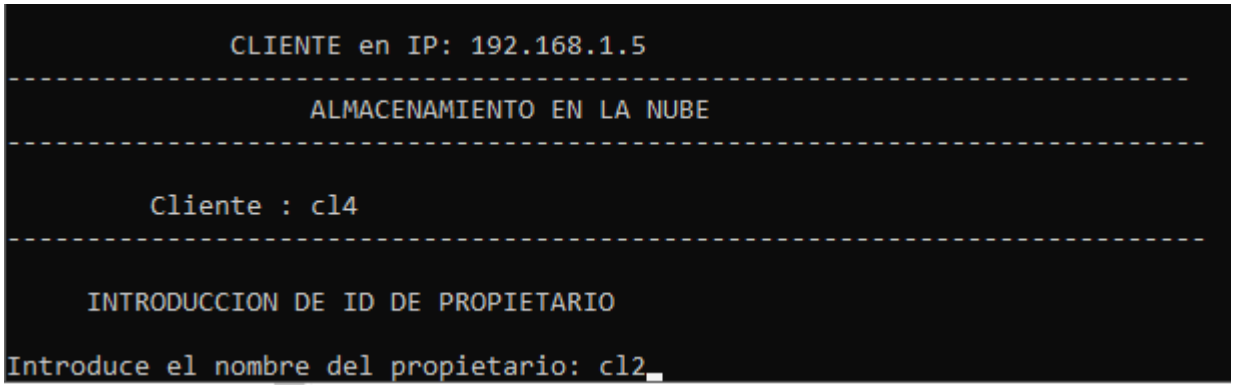
Sistema de archivos con los ficheros subidos



menu 2.- Bajar fichero

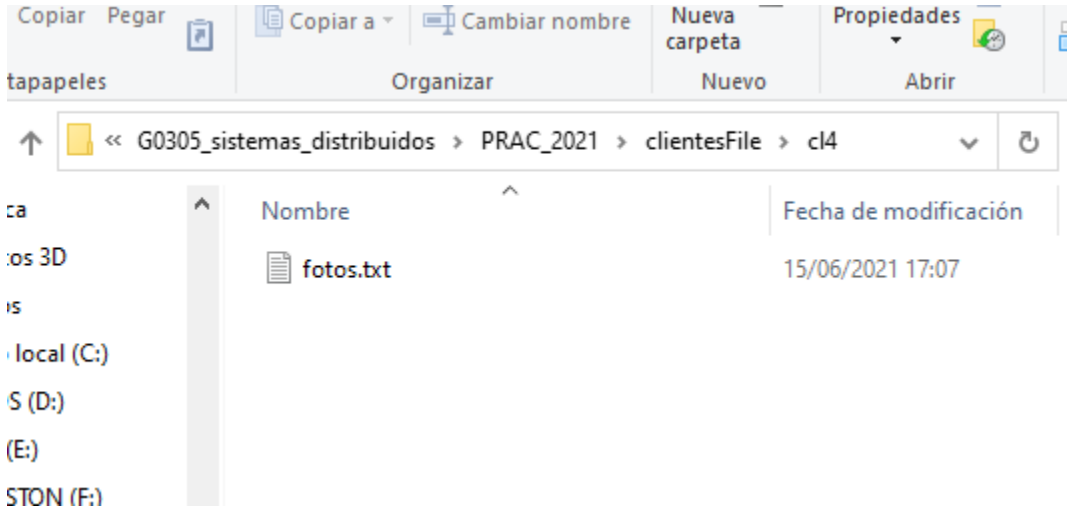
Primero nos pide si es fichero propio (vamos a bajar desde cl4 fotos.txt)





```
            CLIENTE en IP: 192.168.1.5
--------------------------------------------------------------------------
                  ALMACENAMIENTO EN LA NUBE
--------------------------------------------------------------------------

        Cliente : cl4
--------------------------------------------------------------------------

     INTRODUCCION DE NOMBRE DE FICHERO

Introduce el nombre del fichero : fotos.txt
```

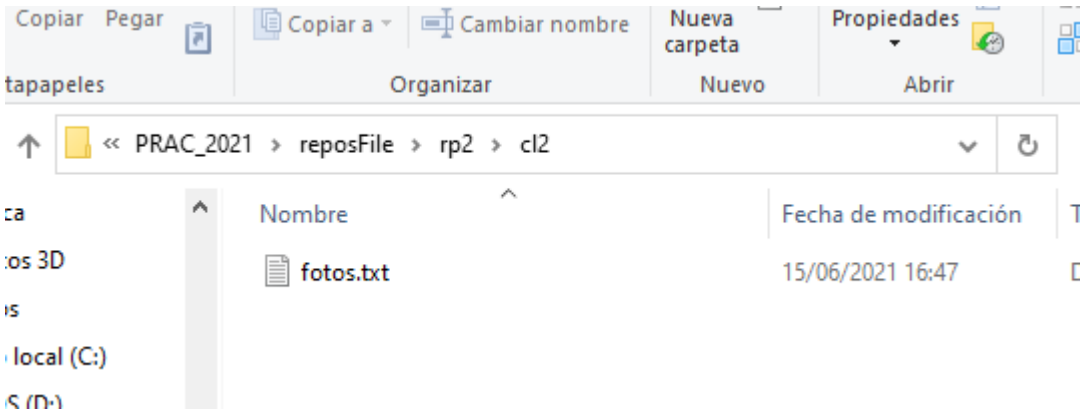
```
            CLIENTE en IP: 192.168.1.5
--------------------------------------------------------------------------
                  ALMACENAMIENTO EN LA NUBE
--------------------------------------------------------------------------

        Cliente : cl4
--------------------------------------------------------------------------

Se ha bajado un fichero fichero nombre fotos.txtdel propietario cl2

Pulsa intro
```



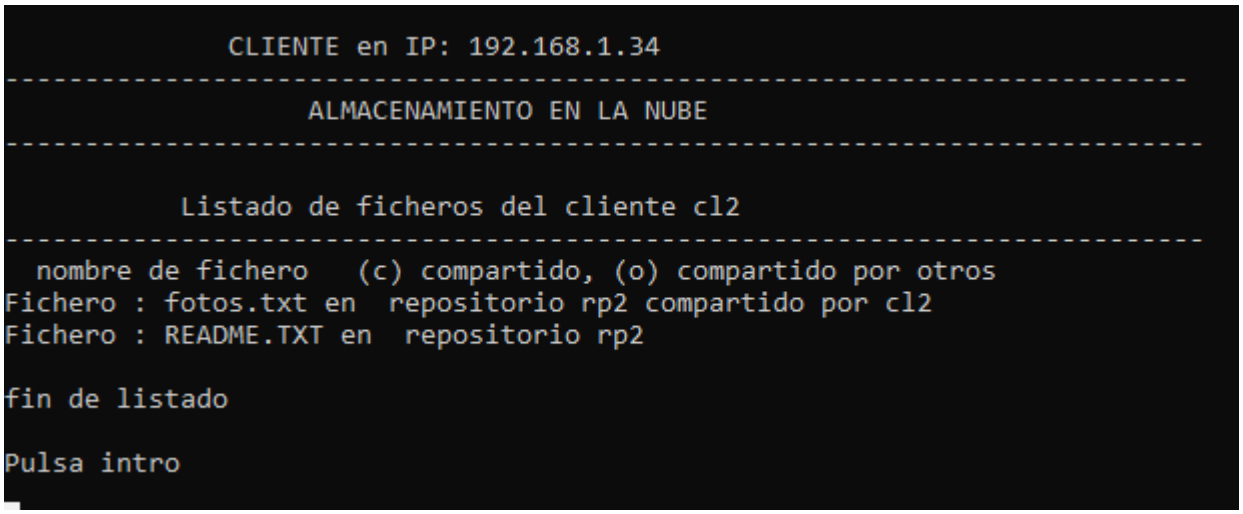
menú 3.- borrar fichero

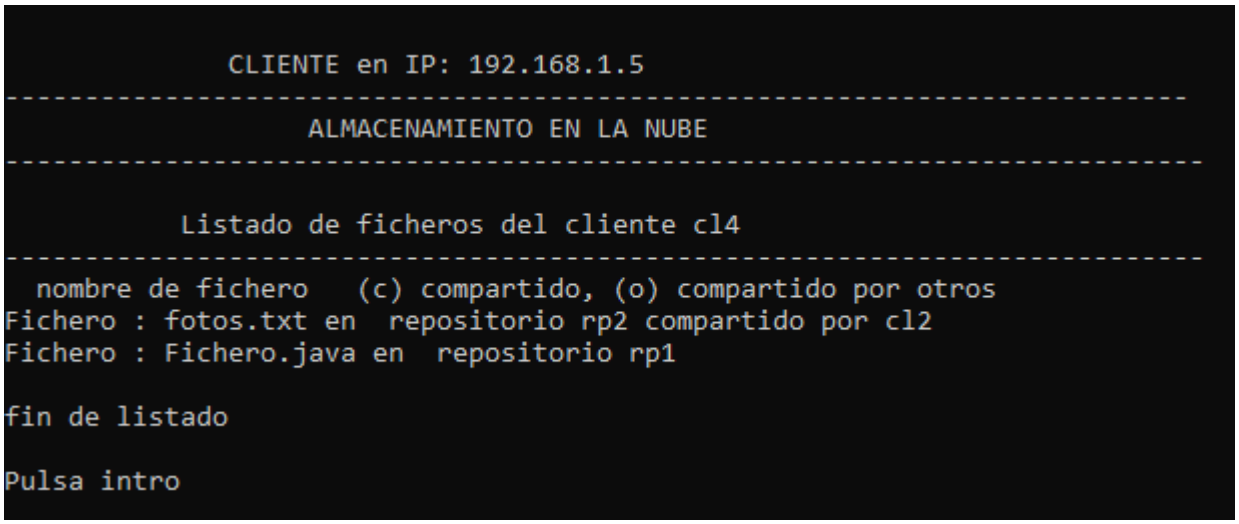
```
            CLIENTE en IP: 192.168.1.34
--------------------------------------------------------------------------
                  ALMACENAMIENTO EN LA NUBE
--------------------------------------------------------------------------

        Cliente : cl2
--------------------------------------------------------------------------

Se ha borrado el fichero fichero nombre README.TXT

Pulsa intro
```

Sistema de archivos después de borrar el fichero README.TXT

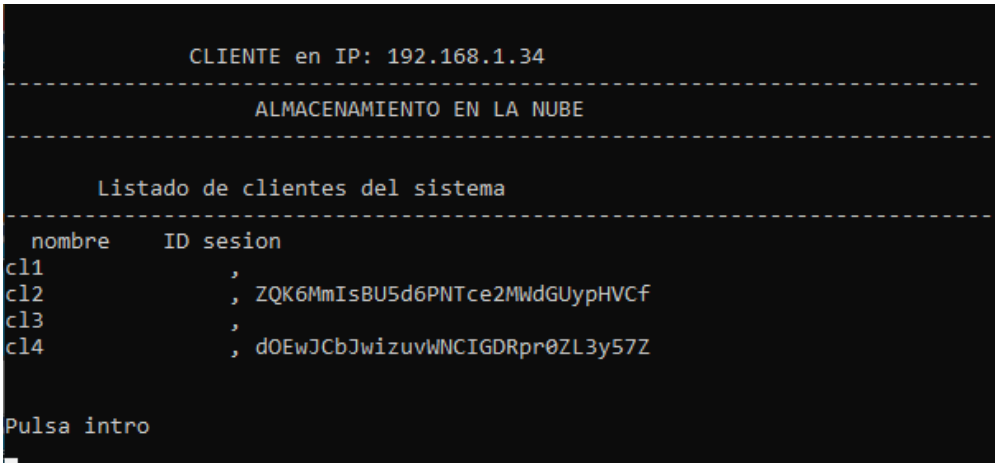


menú 5.- Listado de ficheros del cliente





Menú 6.- Listado de clientes del sistema desde un cliente



8.- MEJORAS QUE SE PROPONEN EN EL SISTEMA

Además de las ya reseñadas en los puntos anteriores:

- Mejorar la presentación.
- Con el diseño realizado, la primera mejora a realizar sería sacar el servicio de datos como una entidad de carácter propio.
- Permitir descargar ficheros desde cualquier sistema de almacenamiento
- Permitir a servidor desactivar una sesión (borrar el IDSesion) cuando un usuario tiene una desconexión inesperada (condición de error o cierre manual).
- Guardar en un fichero Properties la IP y el puerto de escucha del registro RMI, para una configuración fácil de los programas.
- En el momento de obtener la contraseña, guardarla cifrada.
- Permitir quitar la característica de compartido para un fichero (ahora mismo no se puede).

9.- CONCLUSIONES

El objetivo principal ha sido demostrar como funciona una aplicación cliente-servidor distribuida y los condicionantes que nos impone a la hora de desarrollarla.

Desde el punto de vista del desarrollo, se ha decidido hacerlo en paquetes y no por proyectos como en el video de Fermín. Esto es así porque el desarrollo era unipersonal y que la aplicación era pequeña.

Después de hacer 3 paquetes (servidor, cliente y repositorio), constaté que cliente y repositorio compartían muchas de sus funciones, motivo por el cual se quedó en 2 paquetes. No obstante, en el caso de que las entidades hubiesen sido baseDeDatos, Servidor y Usuario, sí habrían sido 3 paquetes (o proyectos, según la envergadura de la aplicación y el equipo de desarrollo).

Desde el punto de vista del middleWare de Java (RMI), una vez asimilada la forma de utilizar sus clases e interfaces (incluyendo la seguridad inherente a la misma), se constata que, a pesar de ser una aplicación cliente-servidor, el código no requiere prácticamente nuevas estructuras, es decir, se cumple que con RMI no cambiar la forma de tratar los mensajes entre clases, salvo a la hora de instanciar las clases remotas.

Por ello, consideré necesario retirar, más si cabe, las referencias a RMI, por lo que encapsulé toda esta operativa en una clase abstracta (como ya he comentado). Esto, si bien me llevó más tiempo del necesario, me permitió obtener unas clases reutilizables para toda esta aplicación (u otra que desarrollara en un futuro). Esto le ha dado mayor legibilidad al código y, si cabe, una mayor ocultación además de la que de por sí brinda el midelware (RMI).

Dicho lo anterior, durante el diseño de la aplicación, sí hay que tener en cuenta como propagar los posibles errores que se capturen durante la ejecución, ya que un error en servidor podría lanzar una excepción no deseada en el cliente (dicho de forma genérica y no circunscrito a esta práctica) . Es decir, hay que diseñar un modelo de seguridad, de forma que cada entidad tenga claras cuales son sus excepciones para no mezclarlas.

Para este tipo de aplicaciones hay que tener en cuenta la serialización de objetos. Si bien en la versión final de mi práctica no se envían objetos propios entre entidades (solo strings, que son Serializable por defecto), durante el

desarrollo hay que tener en cuenta este detalle, pues sino dará errores de ejecución de “marshalling”.

También he constatado que en una aplicación de este tipo no es conveniente utilizar “localhost” como parte de una URL, sino que es conveniente utilizar las herramientas que brinda Java para obtener las direcciones IP y construcción de URL. Uno de los problemas encontrados ha sido que en la ejecución con máquinas virtuales obtenía dirección del host anfitrión y no la de la MV, como era de esperar.

Por último, a la hora de poner en marcha la aplicación, es necesario tener en cuenta fatores externos, como son los cortafuegos y antivirus, que podrían dar problemas a la hora ejecutar los clientes o el servidor.

bibliografía

RMI se estudia en el libro de texto básico, aunque es muy recomendable mirar el contenido de los siguientes enlaces:

- http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html, página Web oficial de RMI, donde se proporciona una documentación muy amplia (en inglés) sobre la utilización de RMI y las clases asociadas.

- http://docs.oracle.com/javase/tutorial/rmi/index.html, tutorial que ayuda a entender y comprender la arquitectura de RMI. Es muy recomendable su lectura.

- Los capítulos sobre Java RMI del Libro de la bibliografía complementaria: ISBN: 978-0201796449, Título: Distributed Computing: Principles and Applications., Autor M.L. Liu.,Editorial Addison Wesley Higher Education. (Existe también versión en castellano).

- Sesiones 5, 6 ,7 y 8 de las tutorías Intercampus de este curso.

- Vídeo tutorial del alumno Fermín Silva sobre Java RMI en Eclipse. http://www.youtube.com/watch?v=vnWBrCSjb44

De esta manera, se obtienen los conocimientos mínimos necesarios para ejecutar una aplicación distribuida RMI de ejemplo que sirva de base para el desarrollo de esta práctica