

RMI CLOUD

Nombre: Cumpián Díaz, Pablo

Asignatura: Sistemas Distribuidos

17 de enero de 2017

Centro Asociado María Zambrano (Málaga)

PRESENTACIÓN

Introducción

En el presente documento voy a documentar la práctica realizada para la asignatura Sistemas Distribuidos del curso 2016/17. Se hará un breve recorrido por las funciones que presta la misma, su funcionamiento, comentario del código, los requisitos cumplidos y no cumplidos por un sistema de ficheros en red de tipo canónico, y un breve listado de las posibles mejoras que es posible realizar.

Objetivo

El objetivo de nuestro programa será por un lado realizar un prototipo de servicio de archivos en la nube simplificado, y por otro ejemplificar el uso de la API Java RMI, de manera que conforme se vayan resolviendo las funciones del mismo se vaya complementando los fundamentos teóricos presentados en la asignatura.

Breve resumen del proyecto

EL paradigma usado para esta aplicación distribuida será el de la Programación Orientada a Objetos (POO), realizada en el lenguaje Java. El esquema usado será el de asignación de roles cliente-servidor a las distintas máquinas que intervengan.

Esquema del proyecto

Por una parte tendremos un servidor de autenticación (encargado de la gestión de usuarios y su login), servidores de archivos o repositorios (encargados de la manipulación de los archivos de los usuarios y su almacenamiento), y los propios clientes. La principal razón de este esquema, amén del reparto de trabajo, es la posibilidad de escalar los recursos según vayan siendo necesarios. Así, por ejemplo, a medida que crezcan los usuarios y sus necesidades de espacios, se pueden ir añadiendo más repositorios que atiendan las peticiones de almacenamiento.

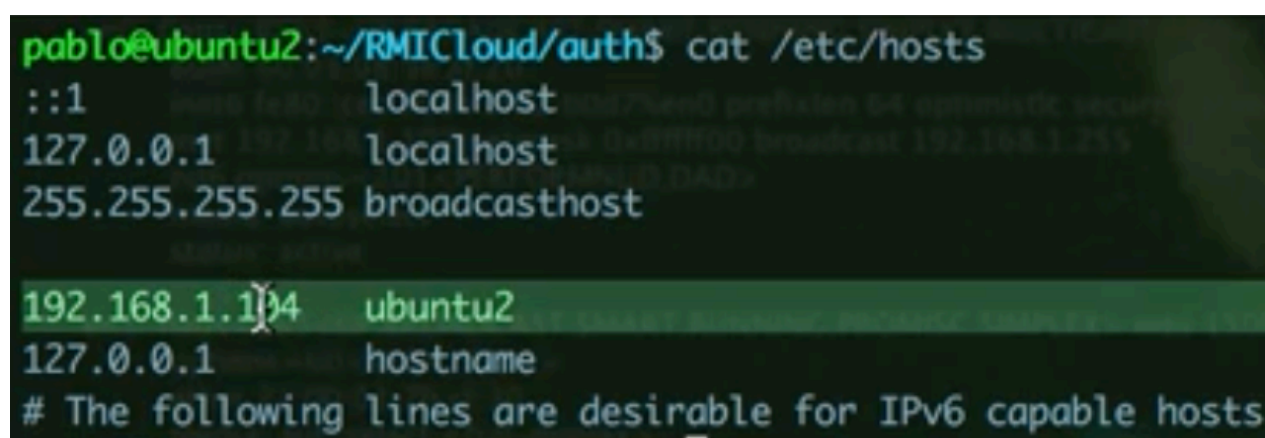
Vídeo

Con objeto de hacer lo más clara posible la explicación del funcionamiento he realizado un vídeo explicando por encima el funcionamiento y las configuraciones que han de realizarse antes de ejecutar los sistemas: <https://www.youtube.com/watch?v=qfc4S6VsPyo&feature=youtu.be>.

CONFIGURACIÓN PREVIA

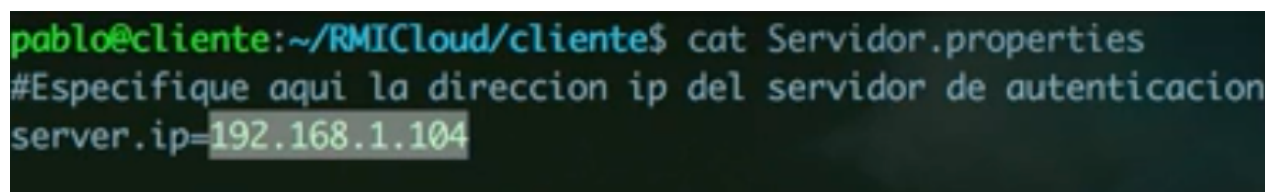
Para poder ejecutar la aplicación es necesario que se realicen cambios en algunos archivos de texto:

- En servidores de autenticación y repositorios se ha de añadir al archivo /etc/hosts una línea con la dirección IP local y el nombre del host.



```
pablo@ubuntu2:~/RMICloud/auth$ cat /etc/hosts
::1 localhost
127.0.0.1 localhost
255.255.255.255 broadcasthost
192.168.1.104 ubuntu2
127.0.0.1 hostname
# The following lines are desirable for IPv6 capable hosts
```

- En los clientes hemos de introducir en el archivo Servidor.properties la IP de nuestro servidor de autenticación, para que así el cliente pueda conectarse automáticamente a este al iniciar.



```
pablo@cliente:~/RMICloud/cliente$ cat Servidor.properties
#Especifique aqui la direccion ip del servidor de autenticacion
server.ip=192.168.1.104
```

De no introducirse estas direcciones en los archivos correspondientes la práctica no funcionará correctamente, por lo que este paso es indispensable. Queda pendiente pues que la aplicación automatice a través del script de inicio el añadir la dirección local al archivo /etc/hosts (aunque esto ocasionaría conflictos con terminales con más de una interfaz de red, por ejemplo).

EJECUCIÓN

Una vez se hayan hecho las configuraciones señaladas anteriormente, procederemos a ejecutar los diferentes programas en cada uno de los sistemas.

La práctica se compone de tres carpetas:

- auth
- repo
- cliente

Cada una de ellas contiene el código necesario para correr de forma autónoma en los sistemas que se quiera.

El arranque de cada rol es tan sencillo como introducir los siguientes comandos para compilar y ejecutar:

- `javac *.java && java Auth`
- `javac *.java && java Repositorio`
- `javac *.java && java Cliente`

Ademas cada uno de los directorios contienen ficheros .sh con los que arrancar los programas

- `sh auth.sh`
 - `sh repo.sh`
 - `sh cliente.sh`
-

En cada uno de ellos se desplegará un menú que mostrará las diferentes opciones disponibles.

```
pablo@ubuntu2:~/RMICloud/auth$ java Auth
Objetos maps cargados
Elige una de las siguientes opciones:
1) Listar usuarios
2) Listar repositorios
3) Listar usuarios y su repositorio
4) Registrar usuario
5) Borrar usuario
6) Consultar usuario
7) Iniciar servidor
8) Salir
Introduzca su opción:
7
Servidor arrancando
```

```
Pablo@Mac-Pablo:~/RMICloud/repolmaster ⚡
⇒ java Repositorio
Elige una de las siguientes opciones:
1) listar ficheros
2) listar clientes
3) iniciar repositorio
4) salir

Introduzca su opción:
3
Servidor activo en el puerto: 1099
```

```
pablo@cliente:~/RMICloud/cliente$ java Cliente
Arrancando el cliente...
Proxy[Servidor,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192.168.1.104:40526](remote),objID:[-58c6db8:159aeb1a1fa:-7fff, -6566943271081244705]]]]]
BIENVENIDO A MYCLOUD
Elige una de las siguientes opciones:
1) Log in
2) Registrar usuario
3) Salir
```

Al arrancar el cliente, este leerá el archivo Servidor.properties, del cual sacará la dirección IP del servidor de autenticación. A continuación mostrará el objeto en formato cadena que ha importado (el servicio de autenticación), y estará listo para hacer login o registrar un nuevo usuario.

COMENTARIO DEL CÓDIGO FUENTE

Interfaces

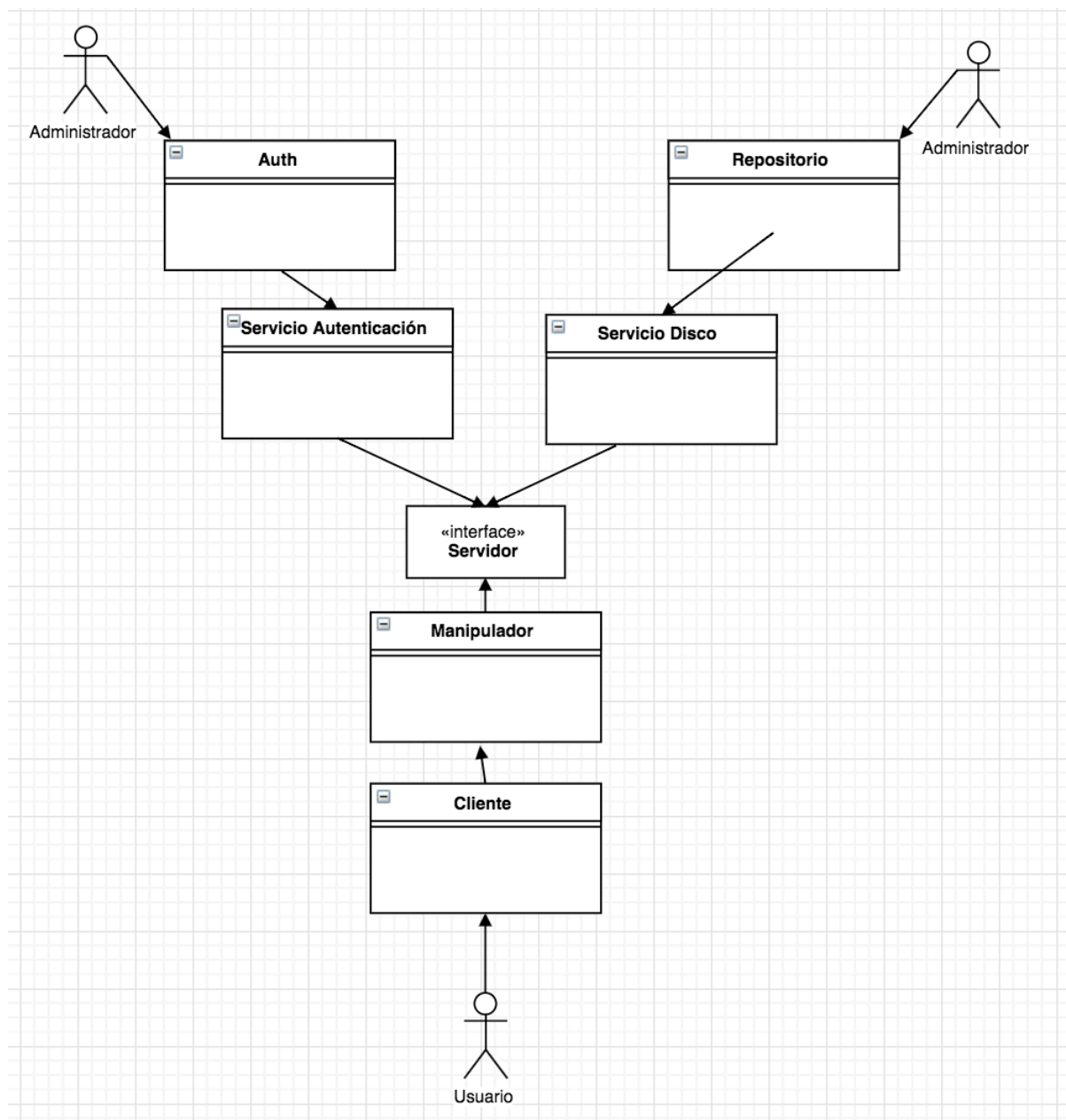
En la práctica se usa la interfaz servidor en todos los roles como forma de intermediario para poder acceder a los métodos remotos. Estas interfaces nos sirven además para visualizar de forma esquemática las funciones del sistemas:

```
1 > import java.util.*;~
4 ~
5 public interface Servidor extends java.rmi.Remote {~
6 >> String loginUsuario(String usuario, String password) throws RemoteException;~
7 ~
8 >> Map devolverUsersIP() throws RemoteException;~
9 ~
10 >> void crearCarpeta(String usuario) throws IOException;~
11 ~
12 >> void registrarUsuario(String user, String pass, String ip) throws RemoteException;~
13 ~
14 >> String listarClientes() throws RemoteException;~
15 ~
16 >> String listarFicheros(String usuario) throws RemoteException;~
17 ~
18 >> void borrar(String file, String usuario) throws IOException;~
19 ~
20 >> void mover(String archivo, String usuario) throws IOException;~
21 ~
22 >> OutputStream getOutputStream(File f) throws IOException;~
23 ~
24 >> InputStream getInputStream(File f) throws IOException;~
25 ~
26 }
```

El método loginUsuario realiza un intercambio de claves entre cliente y servidor, de forma que el primero le entrega su nombre y su contraseña y el servidor, previa validación de estos datos, devuelve la dirección ip del repositorio en el que se encuentran almacenados los ficheros del usuario. Para ello se utilizan estructuras de datos HashMap (provistas por el estándar Java) y su serialización para su persistencia en disco.

De la misma forma registroUsuario recoge los datos del usuario y los almacena en estas estructuras en el servidor. Este método usa además los métodos crearCarpeta() para crear una carpeta para el nuevo usuario registrado inmediatamente después de que sean guardados sus datos.

Asimismo, son funciones compartidas tanto por el cliente como por el servidor el listado de usuarios, listado de ficheros y consulta de los mismos. Con el objetivo de cumplir con esto se utilizan los métodos `devolverUsersIP()`, `listarClientes()`, `listarFicheros()`. Para el manejo de archivos se usa `crearCarpeta`, `borrar`, `mover`, y la descarga y subida de archivos con `getOutputStream` y `getInputStream`. Cualquier subida realizada al repositorio invoca el método `mover()` para asegurar que el archivo subido es trasladado a la carpeta del usuario. Debido a la trivialidad de algunas implementaciones, se mostrarán sólo aquí el código de algunos de los métodos más interesantes.



Como puede observarse, he realizado algunos cambios en el diseño facilitado por el equipo docente para simplificar y unificar los servicios. Así, cada servicio propio de los distintos roles de sistema será realizado por un objeto distinto, a saber Manipulador para el cliente, ServicioAutenticación para Auth, y ServicioDisco para el repositorio.

La conexión entre sistemas se hace con un esquema general parecido siempre al siguiente.

```
try {
    boolean useSecurityManager = false;
    try {
        Boolean.valueOf(properties.getString("useSecurityManager"));
    } catch (Exception e) {}
    if (useSecurityManager && System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }

    Registry registry = LocateRegistry.createRegistry(2099);

    ServicioAutenticacion obj = new ServicioAutenticacion();
    /* Bind this object instance to the name "SimpleServer" */
    // Naming.rebind("SimpleServer", obj);
    registry.rebind("Server", obj);

    System.out.println("Servidor arrancando");
} catch (Exception e){
    System.out.println("Error en servidor de autenticación: " + e.getMessage());
    e.printStackTrace();
}
```

En primer lugar se establece una política de seguridad en RMI, en el que se leerá el valor, si existe, useSecurityManager de nuestro ya conocido archivo Servidor.properties. En esta práctica he incluido el esquema general para incluir una política de seguridad, pero he prescindido de implementarla con objeto de mantener lo más simple posible la ejecución.

Acto seguido añadimos un registro RMI en el puerto 2009, y creamos un nuevo objeto/servicio de autenticación para que esté disponible mediante el rebind en nuestra red local. Este servicio se publicará bajo el nombre Server.

Por parte del cliente, la búsqueda de este servicio se realiza mediante la ejecución siguiente:

```
auth = (Servidor) Naming.lookup( "//" + serverIP + ":" + 2099 + "/ServerAuth");  
System.out.println(auth.toString());
```

Se castea como Servidor (interfaz que es intermediaria entre todos los servicios) el objeto que hemos descrito antes en el servidor, buscándolo con Naming.lookup, e indicándole su dirección IP (previamente establecida en el archivo Simple.properties) y el mismo puerto que hemos establecido antes, con el nombre al final, ServerAuth. Esta notación se aproxima un poco a la de las URLs convencionales (// host + puerto + /nombre).

Uno de los métodos que más interés tienen es el que permite hacer login a los usuarios:

```
>> public String loginUsuario(String usuario, String password) throws RemoteException {  
>> >> String ip = "";  
>> >> try {  
>> >> >> FileInputStream fileIn1 = new FileInputStream("user_pass.ser");  
>> >> >> ObjectInputStream in1 = new ObjectInputStream(fileIn1);  
>> >> >> user_pass = (HashMap) in1.readObject();  
>> >> >> in1.close();  
>> >> >> fileIn1.close();  
>> >> >> FileInputStream fileIn2 = new FileInputStream("user_ip.ser");  
>> >> >> ObjectInputStream in2 = new ObjectInputStream(fileIn2);  
>> >> >> user_ip = (HashMap) in2.readObject();  
>> >> >> in2.close();  
>> >> >> fileIn2.close();  
>> >> >> if(password.equals(user_pass.get(usuario))){  
>> >> >> ip = user_ip.get(usuario);  
>> >> >> }  
>> >> >> }catch(IOException i) {  
>> >> >> i.printStackTrace();  
>> >> >> }catch(ClassNotFoundException c) {  
>> >> >> c.printStackTrace();  
>> >> >> }  
>> >> return ip;  
>> }  
>> }
```

En el se hace una deserialización de los archivos user_ip.ser y user_pass.ser, los cuales contienen la relación de sus usuarios y sus IPs de repositorios y sus contraseñas. Así, el método recoge el usuario y la contraseña, y, de corresponder el uno con el otro, devuelve la IP de ese usuario. La serialización de HashMaps se usa aquí para que los datos sean persistentes en disco.

Por otra parte el manejo de los archivos suele resolverse con ejecución de comandos UNIX básicos, como vemos en el siguiente ejemplo:

```
public void crearCarpeta(String usuario) throws IOException {  
    >> Process p;  
  
    >> try {  
        >> >> p = Runtime.getRuntime().exec("mkdir " + "clientes" + "/" + usuario);  
  
        >> >> p.waitFor();  
        >> >> p.destroy();  
    >> } catch (Exception e) {}  
}
```

crearCarpeta es un método al que se llama cuando un usuario se da de alta en el sistema, de forma que es el propio servidor de autenticación quien se encarga de llamarlo al repositorio correspondiente. Para llevar a cabo el comando UNIX simplemente se crea un proceso de ejecución en el que el comando que queramos pasarle se hace con un String.

REQUERIMIENTOS CUMPLIDOS

En esta sección haremos un breve repaso de los requisitos que un sistema de archivos distribuido debe cumplir, evaluando si el presente proyecto los cumple o no:

- Transparencia: cumple con la transparencia de acceso, de ubicación y de movilidad, ya que la configuración que asegura la independencia de los roles se mantiene gracias al archivo de configuración `Servidor.properties`. En cuanto a la transparencia de prestaciones es un requisito difícil de calibrar: yo he probado los servicios con tres clientes funcionando, y funciona con normalidad. La transparencia de escala se cumple debido al diseño que separa repositorios y servidores de autenticación, pudiendo aumentarse los repositorios únicamente según se necesite más espacio para nuevos usuarios.
- La actualizaciones concurrentes de archivos no se cumple ya que no he implementado el servicio de compartir archivos, por lo que se mantiene de forma rígida la independencia de directorios entre clientes.
- Replicación de archivos: no se cumple
- Heterogeneidad de hardware y del sistema operativo: el proyecto funciona únicamente en sistemas UNIX y Linux, ya que algunos de los comandos que se ejecutan para el manejo de archivos pertenecen al estándar POSIX o similares.
- Tolerancia a fallos: se cumple, ya que el sistema no se cae al realizar una conexión fallida con otro sistema.
- La consistencia se mantiene ya que está restringida a una sola copia del archivo que sube un usuario.
- Seguridad: no se implementan ACLs ni políticas de seguridad en el programa original.
- Eficiencia: he realizado pruebas con archivos grandes (pdfs de 10 MB aprox.) y la velocidad oscila en una media de 15 segundos.

POSIBLES MEJORAS

A continuación se listan algunas de las mejoras que el autor considera que pueden realizarse al presente proyecto:

- Si bien es cierto que Java RMI obliga al programador a situar sus sentencias entre bloques de try/catch para el control de fallos, así como cada método de las interfaces tiene asignada una excepción, hay operaciones que no están comprobadas aún.
 - Por comodidad se debería desarrollar un script que automatice el añadido de la dirección IP al `/etc/hosts` de servidores y repositorios para que no hay que introducirlo manualmente.
 - Con previsión de depurar código, sería útil implementar un archivo de logs que registre las operaciones básicas realizadas (subir, bajar, logins, etc...)
-