

Sistemas Distribuidos

Práctica Final



Sistemas Distribuidos

Alejandro Ladrón de Guevara Jiménez, NIA = 100499541

Fernando Mendizábal Santiago, NIA = 100495773

Índice

1. Introducción	3
2. Funciones client.py	3
3. Funciones server.c	4
4. Servicio Web y RPC	5
5. Makefile y ejecutables	6
6. Batería de Pruebas	7
7. Conclusiones	7

1. Introducción

En esta memoria, explicaremos el proceso llevado a cabo en la creación de un sistema distribuído que simula el comportamiento de una red de ordenadores tipo peer-to-peer. Este consiste en la comunicación basada en sockets de diferentes dispositivos “cliente” con otros dispositivos gracias a las funcionalidades ofrecidas por un “servidor”. Incluiremos las debidas explicaciones de los diferentes archivos de código empleados, la forma de compilación y ejecución de dicho sistema, su funcionamiento secuencial en forma de diagramas y las pruebas realizadas que confirman el correcto funcionamiento de nuestro código.

2. Funciones client.py

El código fuente de los clientes está en código Python. Para la inicialización de un cliente se deberá escribir `python client.py -s <ip_server> -p <port_server>` en la terminal, en donde se introducirán la IP y el puerto servidor. Estos se podrán averiguar en la interfaz del servidor del sistema, explicado más adelante en esta memoria. En el caso de ejecutarse en un sistema Unix, se cambiará `python` por `python3`.

El intercambio de mensajes por sockets empleado se basa en una codificación de la información a través del formato XDR. Cabe destacar que para la versión de python encontrada en los sistemas Unix, este formato es considerado obsoleto, por lo que aparecerá un disclaimer advirtiendo de ello. Sin embargo, esto no indica que exista un error en su funcionamiento, por lo que el programa será ejecutado correctamente.

Tras ejecutarse el código, se podrá acceder a los diferentes comandos del sistema. En la mayoría de los casos, las operaciones de la parte cliente consisten en el intercambio de información. El cliente se dedica a codificar cierta información en el formato XDR para pasársela al servidor a través de sockets, conectándose a él con la dirección IP y el puerto pasados en el comando de inicialización. En las diferentes funciones se realizan además los diferentes pasos que requieran para su correcto desempeño:

- REGISTER: Empacará en el contenedor XDR “REGISTER” y el usuario pasado como argumento, y se lo pasará por sockets al servidor, todos en formato string. Como respuesta se recibirá en todas las funciones un entero que indica el resultado de la operación. Recibirá un 0 si todo ha ocurrido correctamente, un 1 si ya existe el usuario que se intenta registrar o un 2 para cualquier otro tipo de error.
- UNREGISTER: Empacará en el XDR “UNREGISTER” y el usuario pasado como argumento, y se lo pasará al servidor, todos en formato string. Como respuesta recibirá un 0 si todo ha ocurrido correctamente, un 1 si no existe el usuario que se intenta eliminar o un 2 para cualquier otro tipo de error.
- CONNECT: Buscará con la ayuda de un socket auxiliar un puerto libre en el dispositivo. Posteriormente, empacará en el XDR “CONNECT”, el usuario pasado como argumento y el puerto obtenido con anterioridad. Antes de enviar esta

información al servidor, se creará un thread que abrirá un socket para la dirección IP del dispositivo y el puerto obtenido anteriormente. Tras recibir la respuesta, si se recibe un 0 de que todo ha ocurrido correctamente, se guardará en una variable protegida llamada `_user` el nombre del usuario conectado. Si se recibe 1, el usuario que se intenta conectar no ha sido registrado. Si se recibe 2, dicho usuario ya está conectado. También se recibirá 3 para cualquier tipo de error. En cualquier caso de error, el thread que se había creado finalizará.

Como aclaración, se ha creado una variable `_user` para que el código detecte cuando un usuario está realizando las diferentes acciones del sistema estando o no conectado, para dar los errores pertinentes. También se ha creado `_thread_run` para controlar el cierre de los threads y también se ha creado `_thread` como contenedor del hilo creado, para las diferentes acciones entre funciones.

- **DISCONNECT**: Empacará “DISCONNECT” y el usuario pasado como argumento, ambos en string. Envía la información al servidor y recibe un 0 si se ha realizado todo sin errores, 1 si no existe el usuario, 2 si no está conectado y 3 por cualquier otro error. Si el resultado es 0, se cierra el thread iniciado por el cliente cuando se conectó. Además, `_user` pasará a ser NULL, para indicar que el usuario no está conectado.
- **PUBLISH**: Devuelve 2 si detecta que no hay un usuario conectado, estando `_user` vacío. Se empacan en string “PUBLISH”, el usuario de la variable protegida, la ruta absoluta del archivo y una descripción. Se envía la información, y posteriormente se recibirá 0 si se ha podido publicar, 1 si el usuario no existe y 2 si no aparece conectado. En ambos casos se deberá a que dicho usuario no existirá o no estará conectado en base a la información que tiene el servidor. Si ya existe un archivo con esta dirección, se devuelve 3. En caso de cualquier otro error, se devuelve 4.
- **DELETE**: Devuelve 2 si detecta que no hay un usuario conectado, estando `_user` vacío. Se empacan en string “DELETE”, el usuario de la variable protegida y la ruta absoluta del archivo. Se envía la información, y posteriormente se recibirá 0 si se ha podido eliminar, 1 si el usuario no existe y 2 si no aparece conectado. En ambos casos los errores se darán en base a la información que tiene o no en su base de datos el servidor. Si no existe un archivo con esta dirección, se devuelve 3. En caso de cualquier otro error, se devuelve 4.
- **LIST_USERS**: Devuelve 2 si el usuario no está conectado. Se empacan en string “LIST_USERS” y el nombre del usuario conectado. Se envía la información y posteriormente se obtiene el resultado. Si es 1, el usuario no existe, si es 2, el usuario no está conectado. En caso de que haya otro error, será devuelto 3. En el caso de que sea devuelto 0, se irán recibiendo conjuntos de valores independientes siguiendo el patrón de nombre, ip y puerto. Cada conjunto de estos corresponde a la información de los usuarios conectados. Estos serán impresos línea a línea tabulados, después de imprimirse el mensaje de “LIST_USERS OK”.

- **LIST_CONTENT**: Devuelve 2 si el usuario no está conectado. Se empacan en string “LIST_USERS”, el nombre del usuario conectado y el del usuario a cuya información se quiere acceder. Se envía la información y posteriormente se obtiene el resultado. Si es 1, el usuario no existe, si es 2, el usuario no está conectado, y si es 3, el usuario a cuya información se quiere acceder no existe. En caso de que haya otro error, será devuelto 4. En el caso de que sea devuelto 0, serán devueltos de forma independiente los nombres de las rutas absolutas que tiene guardadas el usuario. Estos serán impresos línea a línea tabulados, después de imprimirse el mensaje de “LIST_CONTENT OK”.
- **GET_FILE**: Devuelve 2 si el usuario no está conectado. Se empacan en string “LIST_USERS”, el nombre del usuario conectado y el del usuario a cuya información se quiere acceder. Se conecta por sockets al servidor y se le envía la información, para recibir el mensaje del resultado, 0 si todo correcto y 2 si ha habido algún error. Si se ha dado 0, se recibirá a su vez el valor de la ip y el puerto del usuario al que nos queremos conectar. Creamos una conexión vía sockets para esta ip y puerto. Empacaremos la ruta absoluta del archivo que queremos obtener del usuario remoto. Se recibe como respuesta un 0 si todo correcto, un 1 si el archivo no se ha podido encontrar en el otro dispositivo y un 2 en caso de cualquier otro error. En el caso de 0, se abrirá/creará un archivo con la ruta absoluta pasada como argumento, y se irá recibiendo del usuario remoto la información del archivo en formato de bytes, hasta que no se reciba más información por el socket.
- **QUIT**: Saldrá del sistema, imprimiendo “+++ FINISHED +++” por pantalla para indicar al usuario esta acción. Si el usuario estuviese conectado, se desconectaría llamando a la función “DISCONNECT”.

Cuando se realiza “CONNECT”, el usuario crea un thread. La función de este thread consiste en iniciar una conexión por sockets y escuchar cualquier mensaje enviado de cualquier dispositivo que se conecte al puerto especificado. Este thread escuchará en intervalos de tiempo las peticiones de los usuarios. Si no recibe nada, vuelve a escuchar. Si recibe algo, mientras que no sea desconectado su respectivo usuario del sistema, recibirá la ruta absoluta de un archivo de su dispositivo. Busca si existe dicho archivo. Si no existe, simplemente devuelve un 1 al otro usuario. Si existe, devuelve 0 y le va pasando paquete a paquete la información referente a dicho archivo al usuario que se ha conectado.

3. Funciones server.c

Para inicializar el ejecutable del servidor, se deberá poner por la terminal ./servidor -p <port>, en donde puerto, será un puerto que especifiquemos por el que el servidor recibirá la información por sockets de los clientes. Tras ello, aparecerá un mensaje “init server <ip>|<port>”, en donde ip es la IP y port es el puerto que el servidor empleará. Para la inicialización de un cliente, deberemos poner en los campos debidos esta información ofrecida por el servidor, de la forma explicada anteriormente. Una vez inicializado el servidor, este obtendrá el puerto introducido por nosotros y creará el socket para la comunicación con los clientes, después creará la estructura server_addr y finalmente usará el comando bind para vincularse al socket creado y se pondrá en modo de escucha, a la

espera de posibles peticiones de clientes. En cuanto le llega una petición de un cliente, crea un thread de la función tratar_mensaje, la cual se encargará de procesar la petición independientemente de cual sea. Lo primero que hace el servidor en todas las funciones es imprimir el string de fecha y hora que le envía el cliente obtenido por el servicio web. Los diferentes tipos de peticiones son:

- **REGISTER**: Abre el fichero de usuarios y busca al usuario, por si ya se encuentra dentro del fichero. Si no lo encuentra, lo añade al fichero y lo cierra. En cuanto al envío de resultados, si el usuario no estaba registrado, se registra y devuelve 0, si el usuario si estaba devuelve 1, y si hay cualquier error con el fichero devuelve 2.
- **UNREGISTER**: Abre el fichero original y crea uno nuevo llamado “temp_users.txt”, donde copia todos los usuarios excepto el que debe eliminar. Si encuentra al usuario, reemplaza el fichero original por el nuevo; si no, borra “temp_users.txt”. Devuelve 0 si elimina correctamente, 1 si el usuario no se encuentra, y 2 si ocurre un error al abrir los ficheros o cualquier otro problema.
- **CONNECT**: Obtiene el puerto del cliente y abre el fichero de usuarios y uno temporal. Recorre los usuarios registrados: si encuentra al que quiere conectarse, añade su IP y puerto al fichero temporal, marcándolo como conectado; los demás usuarios se copian tal cual. Al finalizar, cierra ambos ficheros. Si el usuario fue encontrado y no estaba conectado, reemplaza el fichero original con el temporal y devuelve 0. Si no se encuentra, elimina el temporal y devuelve 1. Si ya estaba conectado, descarta el temporal y devuelve 2. Si ocurre un error al abrir ficheros, devuelve 3.
- **DISCONNECT**: Obtiene el puerto del cliente y abre los ficheros de usuarios y uno temporal. Recorre el fichero original: si encuentra al usuario que quiere desconectarse, elimina su IP y puerto; los demás se copian sin cambios. Luego cierra ambos ficheros. Si el usuario fue encontrado, reemplaza el fichero original con el temporal y devuelve 0. Si no se encuentra, borra el temporal y devuelve 1. Si hay errores al abrir archivos, devuelve 2.
- **PUBLISH**: Recibe del cliente el nombre del fichero a publicar y su descripción. Comprueba si el usuario está conectado; si no, devuelve 2. Si lo está, crea el archivo files/user.txt y, si puede abrirlo, añade la línea “Path: <fichero>” seguida de la descripción debajo. Si no puede abrirlo, devuelve 4. Si todo va bien, devuelve 0.
- **DELETE**: Recibe del cliente el nombre del fichero a eliminar y verifica si el usuario está conectado; si no, devuelve 2. Si lo está, intenta abrir su archivo en files/; si falla, devuelve 4. Si puede abrirlo, crea un archivo temporal files/temp.txt y copia todo menos el fichero a eliminar. Luego cierra ambos archivos. Si el fichero se elimina, reemplaza el original por el temporal y devuelve 0; si no se encuentra, devuelve 3.
- **LIST_USERS**: Abre el fichero de usuarios y lo lee entero, guardando en una variable el numero de usuarios y en otra si el usuario que hace la petición existe. Si el usuario no está conectado, se devuelve 1 y termina. Si hay un error al abrir el

fichero, se devuelve 3 y termina. Si todo está correcto, se imprime el número total de usuarios conectados y para cada uno se imprime su nombres, IP y puerto. Finalmente, cierra ambos ficheros. Si todo ha ido correctamente, se devuelve 0 y termina.

- LIST_CONTENT: Recibe el nombre del cliente cuyas publicaciones se quieren ver. Comprueba que tanto el solicitante como el dueño están conectados. Si lo están, abre files/publisher_user.txt, cuenta las publicaciones y muestra sólo sus títulos. Finalmente cierra el archivo. Devuelve 0 si todo va bien, 1 si el solicitante no existe, 2 si no está conectado, y 3 si el dueño no existe ni está conectado.
- GET_FILE: Recibe el nombre del cliente del que se quiere obtener un fichero. Comprueba que tanto el solicitante como el dueño están registrados y conectados, y guarda la IP y puerto del publicador. Si todo está correcto, los envía y devuelve 0. Si alguno no existe, devuelve 2. Luego cierra el fichero.

4. Servicio web y RPC

Nuestro servicio web en Python realiza una simple función de obtención de la fecha y hora exactas con el uso de cada petición efectuada por cliente y se la envía al cliente para que este pueda enviarla al servidor y así mostrarla por pantalla. Para ello en otro fichero de Python llamado web_service.py hemos definido nuestro servicio web. Este utiliza SOAP (Simple Object Access Protocol) el cual permite la comunicación entre aplicaciones a través de la red usando HTTP y mensajes XML estandarizados.

En nuestro servicio web definimos nuestro servicio llamado DateTimeService el cual tiene una única función, get_datetime, que devuelve únicamente un string con la fecha y la hora actuales. Luego con el uso del protocolo WSGI (Web Server Gateway Interface) definimos la aplicación del servicio web usando nuestro servicio DateTimeService, como tns (Target Namespace) usamos 'spyne.examples.datetime', para identificarlo de forma única en nuestro servidor, y cerramos la definición de la aplicación con el protocolo de entrada y salida de mensajes de Soap11 para que no haya problemas con los datos. Finalmente, usamos la función make_server(<ip>, <puerto>, <aplicaciónWSGI>) para crear el servicio web y la función serve_forever() para dejarlo en escucha constante.

Ahora desde [client.py](#) usamos nuestra función get_datetime_from_service() que será la que se comunique con el servicio web. Esta, define la dirección IP a la que se conectará de la forma wsdl = "<http://localhost:8000/?wsdl>", crea un cliente SOAP usando la biblioteca zeep y la conecta al servicio web con la ip wsdl. Finalmente, llama al servicio web de la forma client.service.get_datetime() y guarda el resultado en la variable response. Esta variable después es enviada al servidor en cada una de las peticiones del cliente para que el servidor las imprima luego por pantalla. En el caso de que ocurra un error con el servicio web saltará el error: "SOAP Fault ocurrido", si ocurre un error de conexión inesperado saltará el error: "Error inesperado al obtener la fecha y hora del servicio web". En ambos escenarios, la función devolverá: "00/00/0000 00:00:00", debido al mal funcionamiento.

Para el RPC, comenzamos creando un archivo de definición en lenguaje XDR (.x) que describe el programa RPC llamado IMPRIMIR_PROG con la función IMPRIMIR_STRINGS, que recibe los parámetros nombre, op, file y fecha. Al usar rpcgen, se generan automáticamente los archivos rpc.h, rpc_clnt.c, rpc_svc.c, rpc_xdr.c y rpc_server.c. El archivo rpc_clnt.c contiene el stub del cliente, que gestiona las llamadas RPC desde el cliente, mientras que rpc_svc.c es el stub del servidor, que define el esqueleto para recibir y manejar las solicitudes del cliente. rpc_xdr.c se encarga de la serialización y deserialización de los datos entre cliente y servidor, y rpc.h declara las funciones, variables y estructuras necesarias para la comunicación. Finalmente, rpc_server.c es el archivo modificable donde se implementa la lógica del servidor para manejar la función IMPRIMIR_STRINGS, imprimiendo los datos recibidos. La impresión se realiza en la pantalla del servidor RPC, no en la del servidor principal, y debido a posibles firewalls, el servidor RPC solo ha funcionado en un entorno local.

5. Makefile y ejecutables

El archivo .x define la interfaz del sistema RPC y, junto con el Makefile, permite la correcta compilación del sistema distribuido. El Makefile se encarga de compilar dos ejecutables: el cliente (servidor), basado en server.c, y el servidor(servidor_rpc), que usa los archivos generados automáticamente a partir de rpc.x (rpc.h, rpc_svc.c, rpc_clnt.c, rpc_xdr.c) además de rpc_server.c, donde se implementan las funciones remotas. El cliente Python (client.py), al estar escrito en un lenguaje interpretado, no necesita compilación. Además, el Makefile gestiona correctamente las bibliotecas y opciones necesarias para compilar en un entorno compatible con RPC. Para compilar el sistema, se deberá ejecutar el comando “make -f Makefile.rpc”.

A la hora de ejecutar todo el sistema, lo primero que haremos después del make, será ejecutar en una terminal el servidor con el comando `./servidor <puerto>`, el cual después de esto se pondrá en escucha constante mostrando por pantalla en qué IP y en qué puerto está escuchando. Después, en otra terminal ejecutaremos el servicio web con el comando `python3 web_service.py`, el cual nos mostrará en pantalla la IP y el puerto de escucha también. Luego, en otra terminal, lanzaremos el servidor rpc con el comando `./servidor_rpc` para que funcionen ambas partes.. Finalmente, en otra terminal, lanzaremos el cliente con el comando `python3 client.py -s localhost -p <puerto>`, usando obviamente el mismo puerto que usamos en el servidor. Una vez esté todo correctamente funcionando, solo queda realizar peticiones en el terminal de cliente, las cuales se mostrarán en el terminal de la forma “<operación> OK” junto con el string de fecha-hora que se le pide al servicio web en cada una de ellas, de la forma “Respuesta obtenida: <fecha> <hora>”. Por otro lado, en el terminal de servidor se mostrarán las peticiones realizadas de la forma “OPERATION REGISTER FROM <usuario>”.

6. Batería de pruebas

Para la batería de pruebas, al no tener que realizar un archivo de pruebas como era app-cliente.c en las anteriores prácticas, lo que haremos será describir todos y cada uno de los escenarios de uso de todas las peticiones que le puede enviar el cliente al servidor, y

sus impresiones por pantalla. Mostramos los escenarios para las diferentes peticiones:

- REGISTER: para la petición REGISTER usamos “REGISTER <nombre_usuario>”. Para la primera prueba introdujimos “REGISTER Carlos”, como no había ningun Carlos registrado la terminal mostró “REGISTER OK”. Después, en otro terminal intentamos usar de nuevo “REGISTER Carlos” y el terminal nos mostró el mensaje “USERNAME IN USE”, ya que Carlos era un nombre ya registrado.
- UNREGISTER: para la petición de UNREGISTER usamos “UNREGISTER <nombre_usuario>”. Para la primera prueba introdujimos “UNREGISTER Carlos”, como había un Carlos registrado la terminal mostró “UNREGISTER OK”. Después, en otro terminal intentamos usar de nuevo “UNREGISTER Carlos” y el terminal nos mostró el mensaje “USER DOES NOT EXIST”, ya que Carlos era un nombre que no estaba registrado.
- CONNECT: para la petición CONNECT usamos “CONNECT <nombre_usuario>”. Para la primera prueba introdujimos “CONNECT Luigi”, como Luigi estaba registrado la terminal mostró “CONNECT OK”. Después, intentamos usar de nuevo “CONNECT Luigi” y el terminal nos mostró el mensaje “USER ALREADY CONNECTED”, ya que Luigi ya estaba conectado. Luego usamos “CONNECT Carlos” y como Carlos ya no estaba registrado, el terminal mostró “CONNECT FAIL, USER DOES NOT EXIST”.
- DISCONNECT: para la petición DISCONNECT usamos “DISCONNECT <nombre_usuario>”. Para la primera prueba introdujimos “DISCONNECT Luigi”, como Luigi estaba registrado y conectado la terminal mostró “DISCONNECT OK”. Después, intentamos usar de nuevo “DISCONNECT Luigi” y el terminal nos mostró el mensaje “DISCONNECT FAIL, USER NOT CONNECTED”, ya que Luigi ya estaba desconectado. Luego usamos “CONNECT Carlos” y como Carlos ya no estaba registrado, el terminal mostró “DISCONNECT FAIL, USER DOES NOT EXIST”.
- PUBLISH: para la petición PUBLISH usamos “PUBLISH <nom_fichero> <descripción>”. Para la primera prueba introdujimos “PUBLISH Prueba hola”, y como Juan estaba registrado y conectado la terminal mostró “PUBLISH OK”, y creó la publicación de Juan en un fichero txt con su nombre. Después, intentamos usar de nuevo “PUBLISH Prueba hola” y el terminal nos mostró el mensaje “PUBLISH OK”, ya que esa publicación ya había sido creada, que no logramos implementar esa funcionalidad. Luego usamos “PUBLISH Prueba1 hola” y como Juan no estaba conectado, el terminal mostró “PUBLISH FAIL, USER NOT CONNECTED”. Finalmente, usamos “PUBLISH Prueba2 aaa” y como Juan no estaba conectado y encima ha sido borrado como usuario, el terminal ha mostrado “PUBLISH FAIL, USER DOES NOT EXIST”.
- DELETE: para la petición DELETE usamos “DELETE <nom_fichero>”. Para la primera prueba introdujimos “DELETE Prueba”, y como Juan estaba registrado, conectado y con ese archivo la terminal mostró “DELETE OK”, y borró la publicación. Después, intentamos usar de nuevo “DELETE Prueba” y el terminal nos mostró el mensaje “DELETE FAIL, CONTENT NOT PUBLISHED”, ya que esa publicación ya había sido borrada. Luego usamos “DELETE Prueba” y como Juan no estaba conectado, el terminal mostró “DELETE FAIL, USER NOT CONNECTED”. Finalmente, usamos “DELETE Prueba” y como Juan no estaba conectado y encima ha sido borrado como usuario, el terminal ha mostrado “DELETE FAIL, USER DOES NOT EXIST”.

- **LIST_USERS**: para la petición LIST_USERS usamos “LIST_USERS”. Para la primera prueba introdujimos “LIST_USERS”, y como Juan estaba registrado y conectado mostró la lista de usuarios conectados. Después, intentamos usar de nuevo, con Juan desconectado, “LIST_USERS” y el terminal nos mostró el mensaje “LIST USERS FAILED, USER NOT CONNECTED”. Finalmente, usamos “LIST_USERS” y como Juan no estaba conectado ni registrado el terminal mostró “LIST USERS FAILED, USER DOES NOT EXIST”.
- **LIST_CONTENT**: para la petición LIST_CONTENT usamos “LIST_CONTENT <usuario>”. Para la primera prueba introdujimos “LIST_CONTENT Juan”, y como Juan estaba registrado y conectado mostró “LIST_CONTENT OK” y la lista de sus publicaciones. Después, intentamos usar de nuevo, con Juan conectado, “LIST_CONTENT Carlos” y el terminal nos mostró el mensaje “LIST_CONTENT FAILED , REMOTE USER DOES NOT EXIST” ya que Carlos no estaba registrado. Luego, usamos “LIST_CONTENT Juan” y como Juan no estaba conectado el terminal mostró “LIST_CONTENT FAIL , USER NOT CONNECTED”. Finalmente, usamos “LIST_CONTENT Juan” y como Juan no estaba conectado ni registrado el terminal mostró “LIST_CONTENT FAIL , USER NOT CONNECTED”.
- **GET_FILE**: para la petición GET_FILE usamos “GET_FILE <user_remoto> <nom_fichero_remoto> <nom_fichero_local>”. Para la primera prueba introdujimos “GET_FILE Carlos Prueba Prueba1”, y como Juan estaba conectado y con la publicación Prueba1, y Carlos estaba conectado y con la publicación Prueba, mostró “GET_FILE OK”. Luego, introdujimos “GET_FILE Carlos Prueba2 Prueba1”, y como Juan estaba conectado y con la publicación Prueba1, y Carlos estaba conectado pero sin la publicación Prueba2, mostró “GET_FILE FAIL , FILE NOT EXIST”.

Para todas las pruebas, el servidor muestra por pantalla “OPERATION <operación> FROM <usuario>” y el cliente muestra lo que le manda el servicio web, que es “<fecha_actual> <hora_actual>”.

7. Conclusiones

Para finalizar, nuestro sistema distribuido de comunicación y publicación de ficheros cliente-servidor, cliente-cliente cliente-servicio web ha logrado implementar todas las funciones. Esto se debe al uso de un fichero de texto como almacenamiento de las publicaciones. El acceso y manipulación de este tipo de almacenamiento de datos es bastante simple, lo que supone una ventaja respecto a otros sistemas de bases de datos. Sin embargo, su acceso simple produce que pueda ser modificado de forma externa al sistema, corrompiendo o borrando los datos que contenga. Por ello, para un sistema pequeño como el que hemos creado esta forma de almacenamiento es idónea. En el caso de querer escalar este sistema, deberemos cambiar esta base de datos por otra más sofisticada.