Práctica sockets

Francisco Pinto Santos
Héctor Sánchez San Blas

Índice

1. Introducción

En este documento se propone una solución al ejercicio planteado en la asignatura Redes de Computadores I. El ejercicio solicita la realización un servidor y cliente, siguiendo el protocolo TFTP, capaces de establecer una comunicación en la que se produzca un intercambio de archivos ya sea de cliente a servidor o viceversa. La solución aquí presentada hace uso de los Sockets de Berkeley para llevarse a cabo.

1.1. Consideraciones de los scripts

Este apartado se incluye debido a que se sabe los scripts y el makefile pueden dar problemas. Esto se debe a que se ha desarrollado en diversas ocasiones en un entorno Windows con sublime text. Por ello en ocasiones da errores de sintaxis que no son ciertas debido a que el formato de fin de línea de Windows es distinto en unix.
Por eso, en el caso de que aparezca dicho problema se recomienda usar la orden dos2unix fichero.sh para solucionar estos problemas. Con esto se logrará que el fichero se convierta en formato unix y se pueda usar para el propósito para el cual fue escrito.

2. Solución propuesta

En este apartado se explicará los aspectos importantes relativos al trabajo desarrollado y apartados importantes del proyecto.

2.1. Funcionalidad

2.1.1. Consideraciones generales

- Se ha creado un fichero de cabecera llamado utils, donde se ha definido la mayoría de las constantes necesarias para la ejecución, como el numero de puerto, el tiempo asignado para los timeouts, el numero de intentos, tamaño de buffer, ….
  También se definen las carpetas donde se van a volcar los logs, ficheros, nombres estándar de ficheros para log, …
  En dicho fichero también se encuentran definidos algunos tipos de interés como bool, ProtocolMode (utilizado para indicar si se está operando en TCP o UDP), ….
  Respecto a este fichero cabe destacar que se utiliza para declarar algunas macros de interés como MODE_STR que traduce una variable del tipo ProtocolMode a un string (array de char) con el modo (utilizado para el log principalmente).
- Se ha encapsulado el introducir y extraer mensajes del buffer utilizado para el envío y recepción de datos en un fichero llamado msgUtils.c. En él se realizan tareas de copiado de memoria y establece el tipo de mensaje a bajo nivel (los 2 primeros bytes de la trama TFTP).
  Por último, esta fuente contiene una función que toma los 2 primeros bytes de la trama TFTP, indicando así el tipo de mensaje, para cuando es necesario consultarlo.
- En cuanto a la implementación del protocolo en sí, tanto en cliente como en servidor, se construyó una función para realizar el modo lectura y otra para el modo escritura, a la cuales se les indicaba si se necesitaba modo tcp o udp, para saber que funciones de comunicación utilizar.

- Para gestionar los logs, se han creado funciones de logConnection y logError tanto en cliente como en servidor, para realizar un mejor control de la forma de presentación de los datos en el log y variar dicha representación de una forma mas sencilla.
- Para realizar el tratamiento de señales, se bloquean todas tanto en cliente como en servidor, a excepción de SIGTERM y SIGARLM. En el caso del servidor tampoco se bloquea SIGCHLD.
  Dichas señales no se bloquean porque son utilizadas para los siguientes propósitos:
  - SIGALRM: gestionar que caduquen los timeouts en UDP.
  - SIGCHILD: matar a los procesos y liberar los recursos.
  - SIGCHILD: el servidor no la bloquea, pero la ignora, para permitir a los hijos morir sin quedar zombies.
- La gestión de los timeouts y retries, tanto en cliente como en servidor, se realiza principalmente en la manejadora de sigalrm junto a todos los datos relacionados con esto. Se distinguen fácilmente porque tienen el prefijo SIGALRM.
  En esta manejadora en el caso de que el número de intentos (nRetries) sea menor que el máximo de intentos establecidos (RETRIES en utils.h), se activa el flag timeOutPassed y se incrementa nRetries.
  En caso contrario de que se supere el número de intentos, se escribe en el log el error correspondiente y se cierran el socket correspondiente con el fichero abierto para el intercambio de información.

  Para lanzar los timeouts: se pone nRetries a 0, tras lo cual se entra a un bucle do while, en el cual dentro se encuentra enviar un mensaje (normalmente datos o ack), tras lo cual se pone el flag timeOutPassed a FALSE y se realiza un alarm(TIMEOUT), para que se lance la señal sigalrm dentro de TIMEOUT segundos. Por último se procede a entrar a recvfrom.
  El bucle tiene de condición de salida que timeOutPassed sea falso (que no se haya pasado el timeout), y tras la salida de este se realiza un alarm(0), para desactivar el alarm(TIMEOUT) que se realizo antes.

- En cuanto a los valores establecidos para las constantes del programa:
  - PORT: son las 4 ultimas cifras del DNI de Francisco.
  - TAM_BUFFER: es el tamaño de buffer para enviar o recibir. Se estableció 1024 de tamaño porque era una cantidad de bytes muy superior al tamaño de mensaje máximo que puede llegar, permitiendo así enviar o recuperar cualquier mensaje durante una ejecución normal.
  - RETRIES y TIMEOUT: estas dos constantes se comentan juntas, porque están íntimamente relacionadas.
    En cuanto al TIMEOUT se ha establecido con 10, un valor del que se tiene conocimiento que es excesivamente alto para este propósito, pero se tuvo que incrementar hasta esta cantidad pues es de dominio publico que olivo se satura cuando todos los alumnos están probando sus practicas.
    En cuanto a esto, también se hubiera podido incrementar el numero de intentos, pero se optó por el TIMEOUT.
    Sin embargo, al realizar pruebas en entornos locales, y en el aula hp, entre dos ordenadores del aula, el socket funcionaba perfectamente con 3 de timeout y 3 intentos.

2.1.1. Implementación cliente

El cliente, tanto de tcp como de udp, ha sido implementado teniendo en cuenta las plantillas de Nines. Lo único que ha variado es el tratamiento de argumentos, y la implementación del protocolo en si.

Respecto a la implementación del protocolo, en ambos modos (lectura y escritura), se realiza la inicialización que planteó Nines en su plantilla de los sockets, tras lo cual se procede a mandar una request al servidor del tipo necesario.

En cuanto a la implementación de la lectura, primero se manda la request de lectura y se espera por el primer bloque. Tras ello se abre el fichero donde se guardarán los datos recibidos y se entra en un bucle en el cual se actúa según el dato recibido. Tras esto se envía un ack y se espera el siguiente bloque, a excepción de que el ultimo bloque se haya recibido en la iteración anterior, en cuyo caso se sale del bucle, se envía el ultimo ack y se liberan los recursos que se han utilizado.

La implementación del modo escritura varia con la estructura de lectura, pero no mucho. Primero se manda el request de escritura y se espera el ack 0 que indica confirmación del servidor para empezar la recepción de bloques. Posteriormente, se entra en un bucle en el cual se envía un bloque y se recibe ack hasta terminar la lectura del fichero, lo cual indica terminación de la comunicación. Finalmente, se sale del bucle y se liberan los recursos utilizados.

2.1.2. Implementación servidor

En cuanto a la inicialización, en el main se ha utilizado la plantilla proporcionada por Nines, en el tcp igual pero el udp varía.
En el udp cuando llega una petición, se crea un hijo, el cual crea un nuevo socket y a partir de ahí la comunicación se realizará a través de este nuevo socket, proceso que será transparente para el cliente por como funciona el udp.

Tras esto la implementación del protocolo se ha realizado de forma muy parecida a la del cliente, ya que se han hecho dos funciones, una de lectura y otra de escritura, las cuales distinguen si estamos tratando con TCP o UDP mediante un parámetro.

En cuanto a la lectura, primero se recibe la request de lectura, posteriormente se comprueba si el fichero existe y en caso de que no exista, se manda el error correspondiente. Tras esto se procede al protocolo en si, que consiste en un bucle en el cual se va leyendo el fichero, se envía un bloque y se espera por el ack. Cuando se termina de leer el fichero se sale del bucle y se liberan los recursos utilizados.

En lo que se refiere a la escritura se recibe la request de escritura, tras lo cual se comprueba si el archivo existe, en el caso de que así sea, se envía un mensaje con el error correspondiente. Posteriormente, se abre el archivo para escritura y se envía la confirmación para empezar a recibir datos del cliente. Tras esto, se procede al protocolo que consiste en un bucle en que se recibe los bloques de datos del cliente, se comprueba si la información es correcta, en caso de serlo se escribe en el archivo abierto para escritura y se envía el ack. Cuando se recibe el último bloque de datos se envía el último ack y se liberan los recursos utilizados.

2.2. Implementación de mensajes

La estructura de los mensajes utilizados en el proyecto presentado se ha hecho basados en la documentación proporcionada con el enunciado del problema. La implementación de estos casos se encuentra en el archivo ‘msgUtils.c’ de la carpeta ‘src’. Puesto que se da el caso de enviar mensajes y recibir mensajes, se tienen implementaciones para cada tipo de mensaje utilizado en el sistema y para dos situaciones, recuperar la información del buffer recibido o para llenar el buffer con la información a enviar. A su vez, cuando se llena el buffer, se incluye la cabecera indicando el tipo de mensaje que se envía, dicha cabecera se guarda en formato ASCII segun lo requerido por Nines (también se adjunta en la carpeta ‘binaryHeader’ otro msgUtils.c para mensajes con cabecera en binario, como alternativa posible al ASCII). Para todas las funciones presentada el atributo ‘buffer’ es el lugar donde se vuelca (si es envío) o recoge (si se recibe) la información correspondiente al mensaje.
De esta forma se nos presentan los siguientes casos:

Envío de mensaje:

-Petición de lectura/escritura: int fillBufferWithReadMsg(bool isRead,char * fileName, char * buffer). Con este método obtenemos el buffer con la información necesaria para el envío de una petición de lectura o escritura. El atributo 'isRead' es TRUE si se require lectura, en caso contrario escritura. El atributo 'fileName' contiene el nombre del archive solicitado en la petición.
-Datos: int fillBufferWithDataMsg(int blockNumber, char * data, size_t dataSize,char * buffer). Con este método obtenemos en el buffer la información necesaria para el envío de datos de un archive. El atributo 'blockNumber' contiene el número de bloque se envía en dicho mensaje. El atributo 'data' contiene la información a enviar en ese mensaje. Finalmente, el atributo 'dataSize' contiene el tamaño de los datos a enviar.
-Ack: int fillBufferWithAckMsg(int blockNumber, char * buffer). Con este método obtenemos en el buffer la información necesaria para el envío de un asentimiento (ack) de un determinado bloque de datos recibido. El atributo 'blockNumber' se corresponde con el número de bloque recibido.
-Error: int fillBufferWithErrMsg(errorMsgCodes errorcode, char * errorMsg, char * buffer). Con este método obtenemos en el buffer la información necesaria para el envío de un error ocurrido. El atributo 'errorCode' contiene el código del error producido (su valor viene determinado por los errores proporcionados en el enunciado). El atributo 'errorMsg' contiene la información más específica del error producido.

Admisión de mensaje:

-Petición de lectura/escritura: rwMsg fillReadMsgWithBuffer(char * buffer). Con este método recuperamos la información mensaje de forma ordenada desde el buffer de información recibido correspondiente a un mensaje de petición de lectura o escritura.
-Datos: dataMsg fillDataWithBuffer(size_t dataSize, char * buffer). Con este método recuperamos la información mensaje de forma ordenada desde el buffer de información recibido correspondiente a un mensaje de datos. El atributo 'dataSize' contiene el tamaño de la información recibida.
-Ack: ackMsg fillAckWithBuffer(char * buffer). Con este método recuperamos la información mensaje de forma ordenada desde el buffer de información recibido correspondiente a un mensaje de asentimeinto.
-Error: errMsg fillErrWithBuffer(char * buffer). Con este método recuperamos la información mensaje de forma ordenada desde el buffer de información recibido correspondiente a un mensaje de error.

Cabe destacar que en la recuperación de mensajes se hacen las comprobaciones necesarias para saber si el contenido del mensaje es el correcto o no.

2.3. Realización de conexiones

Dos tipos de protocolos para la conexión son los llevados a cabo en este trabajo:
-TCP: este protocolo está orientado a la conexión. Para la realización de la conexión con el servidor se creó un socket que representa el punto de acceso para lograr acceder a los servicios que proporcionados. Tras la obtención del socket se hizo 'bind' del socket a un puerto a través del cual se comunicarían con él. Tras esto se preparó la cola de escucha con el uso de `listen`. Con esta cola de mensajes se logra almacenar las peticiones de clientes que se le soliciten al servidor TCP. Posteriormente el servidor empieza a aceptar peticiones de los distintos clientes que se encuentren en la cola de espera, esto se realiza con la función 'accept' que en caso de no tener peticiones, se bloquea hasta que estas lleguen. Tras la aceptación de una petición con un cliente empieza el proceso de comunicación con el mismo mediante los métodos 'send' para enviar mensajes y 'recv' para la recepción de los mismos. Una vez finalizada la comunicación con el cliente, vuelve a la parte de aceptar peticiones de la cola de espera.
-UDP: este protocolo no está orientado a la conexión. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión. Para la realización de la conexión con el servidor se creó un socket que representa el punto de acceso para lograr acceder a los servicios que proporcionados. Tras la obtención del socket se hizo 'bind' del socket a un puerto a través del cual se comunicarían con él.Tras eso el proceso espera a la recepción de una petición mediante la función 'recvfrom'. Cuando una petición le llega, este crea un hijo mediante la función 'fork' de manera que este atienda la petición que le ha llegado, mientras que el padre vuelve a la espera de más peticiones para atenderlas mediante su asignación a los hijos

que va creando. Posteriormente, la comunicación de los hijos con los clientes correspondiente se realiza mediante las funciones ‘sendto’ para el envío de mensajes y ‘recvfrom’ para recibirlos.

2.4. Diferencias de implementación

TCP y UDP son distintos protocolos dentro del nivel de transporte. En lo que se refiere a TCP es un protocolo orientado a la conexión. De esta forma, existe un único proceso que atiende cada petición que le llega y con el que se realiza la comunicación. Por lo tanto, como se ha explicado en el apartado anterior, el servidor TCP debe de crear una cola de escucha para el almacenamiento de peticiones que atiende de una por una. Debido a que es un protocolo orientado a la conexión, cuando envía y recibe mensajes con sus métodos ‘send’ y ‘recv’ no necesita conocer la dirección del cliente cada vez que utiliza dichas funciones.
Sin embargo, UDP es un protocolo que no está orientado a la conexión. De esta forma, existe un proceso padre que espera a la llegada de peticiones y, cuando recibe una, crea un hijo que la atiende de forma que el puedo seguir esperando por más peticiones, atendiendo así varias a la vez. Este protocolo hace uso de de las funciones ‘sendto’ y ‘recvfrom’ que cada vez que son llamadas necesitan conocer la dirección del cliente al que envían o del que esperan un mensaje respectivamente. El servidor UDP cada vez que va a enviar un nuevo mensaje resetea su tiempo de espera de respuesta, envía y espera a recibir respuesta. En caso de que pase el tiempo de espera, salta a la manejadora donde incrementa un contador y vuelve a repetir el proceso para el mensaje que había enviado. Si dicho contador alcanza el número de intentos máximo establecido acaba dicha comunicación. En caso de que reciba respuesta antes de que pase el tiempo de espera establecido, resetea el contador de intentos y repite el proceso para un nuevo mensaje.

3. Confirmación de errores

Para la comprobación de los errores contemplado en los protocolos ha sido necesario la modificación de algunas líneas del código o inexistencia de archivos (así como duplicación) para provocar dichos errores. Todos los resultados correspondientes a la ejecución de cada error se encuentra en la carpeta adjuntada ‘resultadosEjecuciones’.

3.1. Unknown

Para la realización de este error en el código del cliente se ha cambiado el incremento del numero de bloque a enviar en 3, de forma que cuando el primer mensaje de asentimiento por parte del servidor llegase al cliente, el primer bloque que envíe el cliente fuese el 4 en lugar del 1. De esta forma, cuando al cliente le llegue el asentimiento enviará el bloque de datos 4 por lo que cuando llegue al servidor no corresponderá con el que esperaba, generando así el error explicado.

El código cambiado:

```
blockNumber = 0;
endSesion = FALSE;
while(!endSesion){
    switch(getMessageTypeWithBuffer(buffer)){
        case ACK_TYPE:
            ackmsg = fillAckWithBuffer(buffer);
            //check if ack its correct; if not an error is send
            if( blockNumber != ackmsg.blockNumber ){
                close(s);
                fclose(f);
                logError(hostName, port, file, MODE_STR(mode), -1 , "Recived ACK for block which
                exit(EXIT_FAILURE);
            }
            blockNumber += 3;
        break;
        case ERR_TYPE:
            errmsg = fillErrWithBuffer(buffer);
            close(s);
            fclose(f);
            logError(hostName, port, file, MODE_STR(mode), errmsg.errorCode , errmsg.errorMsg);
            exit(EXIT_FAILURE);
```

El log correspondiente al servidor:

[2018-12-15 9:38:54][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][PORT: 62165][FILE: test1.gif][STARTED READ]
[2018-12-15 9:38:54][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][MODE: WRITE][PORT: 62165][FILE: test1.gif][ERROR : INTERNAL_SERVER_ERRROR][MSG: Recived data block which doesnt matches with current]
El log correspondiente al cliente:

[2018-12-15 9:38:54][HOST: 192.168.2.246][PROTOCOL: TCP][FILE: test1.gif][STARTED WRITE]
[2018-12-15 9:38:54][HOST: 192.168.2.246][PROTOCOL: TCP][FILE: test1.gif][SEND: 3]
[2018-12-15 9:38:54][HOST: 192.168.2.246][PROTOCOL: TCP][PORT: 62165][FILE: test1.gif][ERROR: UNKNOWN][MSG: Recived data block which doesnt matches with current]

3.2. File not found

Para la realización de este error se ha eliminado de los archivos de la carpeta “ficherosTFTPserver” el archivo test1.gif. Posteriormente, cliente ha solicitado la lectura de dicho archivo al servidor, puesto que el archivo no se encontraba entre los que el servidor poseía devolvió al cliente un mensaje de ‘FILE_NOT_FOUND’, indicándole así que el servidor no tiene dicho archivo en su información.

El log correspondiente al servidor:

[2018-12-15 9:21:4][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][PORT: 60708][FILE: test6.gif][STARTED WRITE]
[2018-12-15 9:21:4][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][MODE: READ][PORT: 60708][FILE: test6.gif][ERROR : FILE_NOT_FOUND][MSG: server coundn't found the requested file]
El log correspondiente al cliente:

[2018-12-15 9:21:4][HOST: 192.168.2.246][PROTOCOL: TCP][FILE: test6.gif][STARTED READ]
[2018-12-15 9:21:4][HOST: 192.168.2.246][PROTOCOL: TCP][PORT: 60708][FILE: test6.gif][ERROR: FILE_NOT_FOUND][MSG: server coundn't found the requested file]

3.3. Disk full

La comprobación de este error no ha sido posible realizarla puesto que para ello era necesario la realización de una partición con un tamaño lo suficientemente pequeño en la cual ejecutar el proyecto. Posteriormente, al enviar un archivo que ocupase una gran cantidad de espacio el cliente no dispondría de espacio suficiente para su almacenamiento.

3.4. Ilegal operation

Para la realización de este error se ha enviado desde cliente una petición con un tipo de mensaje no permitido en el contexto que nos concierne (resaltar que el cliente se encontraba en modo lectura). Posteriormente, cuando el servidor recibió dicha petición, al leer la cabecera de el mensaje y comprobar que no contemplaba la solicitud enviada, respondió con mensaje indicando el error 'ILEGAL_OPERATION' al cliente.

El código cambiado:

```
}

int fillBufferWithReadMsg(bool isRead,char * fileName, char * buffer){
    memset(buffer,0,TAM_BUFFER);

    char header[2];
    char mode[] = OCTET_MODE;
    strcpy(header, ((isRead) ? ("11") : ("12")));

    memcpy(buffer,header,2);
    memcpy(&(buffer[2]),fileName,strlen(fileName));
    memcpy(&(buffer[2 + strlen(fileName) + 1]),mode,strlen(mode));

    return (2 + strlen(fileName) + 1 + strlen(mode) + 1);
}
int fillBufferWithDataMsg(int blockNumber, char * data, size_t dataSize,char * buffer){
```

El log correspondiente al servidor:
[2018-12-15 9:34:48][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][MODE: WRITE][PORT: 61861][FILE: test3.gif][ERROR : ILEGAL_OPERATION][MSG: error on client request header: isn't read or write]

El log correspondiente al cliente:

[2018-12-15 9:34:48][HOST: 192.168.2.246][PROTOCOL: TCP][FILE: test3.gif][STARTED READ]
[2018-12-15 9:34:48][HOST: 192.168.2.246][PROTOCOL: TCP][PORT: 61861][FILE: test3.gif][ERROR: ILEGAL_OPERATION][MSG: error on client request header: isn't read or write]

3.5. File already exists

Para la realización de este error se ha incluido en las carpetas de archivos de servidor y cliente el archivo test2.gif. Posteriormente, cliente ha solicitado una petición de escritura de dicho archivo, puesto que el archivo ya se encontraba entre la información que poseía el servidor este envió un mensaje de 'FILE_ALREADY_EXISTS' al cliente indicándole la incapacidad escribir un archivo que ya posee.

El log correspondiente al servidor:

[2018-12-15 9:26:55][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][PORT: 61176][FILE: test1.gif][STARTED READ]
[2018-12-15 9:26:55][HOST: Mithrandir][IP: 192.168.2.246][PROTOCOL: TCP][MODE: WRITE][PORT: 61176][FILE: test1.gif][ERROR : FILE_ALREADY_EXISTS][MSG: The requested file already exists]

El log correspondiente al cliente:

[2018-12-15 9:26:55][HOST: 192.168.2.246][PROTOCOL: TCP][FILE: test1.gif][STARTED WRITE]
[2018-12-15 9:26:55][HOST: 192.168.2.246][PROTOCOL: TCP][PORT: 61176][FILE: test1.gif][ERROR: FILE_ALREADY_EXISTS][MSG: The requested file already exists]

4. Ejemplo de funcionamiento correcto

Para la ejecución solicitada para el proyecto se utilizó el siguiente script lanzarServidor.sh :

```
#!/bin/sh
#vars
SERVER_IP=olivo

FILE1=fichero1.txt
FILE2=fichero2.txt
FILE3=fichero3.txt
FILE4=fichero4.txt
FILE5=fichero5.txt
FILE6=fichero6.txt
#compile
make
#run
servidor
cliente $SERVER_IP TCP e $FILE1 &
cliente $SERVER_IP TCP l $FILE2 &
cliente $SERVER_IP TCP e $FILE3 &
cliente $SERVER_IP UDP e $FILE4 &
cliente $SERVER_IP UDP l $FILE5 &
cliente $SERVER_IP UDP e $FILE6 &
```

A parte de este script que contiene el lanzamiento de los clientes solicitados por el trabajo junto al servidor, se ha añadido otro script (clean.sh) con código para comprobar la existencia de las carpetas necesarias para el de la información y archivos necesarios (así como eliminación para una nueva ejecución). En caso de de no existir las crea. Esto se debe a que el servidor consulta si existe el archivo en su carpeta antes de ponerse a realizar la petición de lectura y el cliente antes de la petición de escritura. En el caso de que no exista el intento de acceso generará un core dumped.

Tras la realización de la comunicación de cada cliente con el servidor se obtiene un resultado exitoso tal y como se puede comprobar en el log del servidor:

[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41920][FILE: fichero1.txt][STARTED READ]
[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41922][FILE: fichero2.txt][STARTED WRITE]

[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59261][FILE: fichero4.txt][STARTED READ]
[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41923][FILE: fichero3.txt][STARTED READ]
[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59263][FILE: fichero6.txt][STARTED READ]
[2018-12-15 14:3:26][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59262][FILE: fichero5.txt][STARTED WRITE]
[2018-12-15 14:4:34][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59262][FILE: fichero5.txt][BLOCKS: 4294][READ SUCCED]
[2018-12-15 14:4:36][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41923][FILE: fichero3.txt][BLOCKS:4295][WRITE SUCCED]
[2018-12-15 14:4:36][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59261][FILE: fichero4.txt][BLOCKS:4295][WRITE SUCCED]
[2018-12-15 14:4:36][HOST: olivo][IP: 212.128.144.104][PROTOCOL: UDP][PORT: 59263][FILE: fichero6.txt][BLOCKS:4295][WRITE SUCCED]
[2018-12-15 14:4:36][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41920][FILE: fichero1.txt][BLOCKS:4295][WRITE SUCCED]
[2018-12-15 14:4:36][HOST: olivo][IP: 212.128.144.104][PROTOCOL: TCP][PORT: 41922][FILE: fichero2.txt][BLOCKS: 4294][READ SUCCED]

Esto se puede corroborar a su vez en los logs correspodientes a cada cliente que se encuentran en la carpeta adjuntada al informe ‘doc/excution results/correct execution’ correspondiendo cada .txt a:

- UDP escritura fichero5.txt : 59262.txt .
- UDP lectura fichero6.txt: 59263.txt .
- UDP lectura fichero4.txt: 59261.txt .
- TCP lectura fichero1.txt: 41920.txt .
- TCP escritura fichero2.txt: 41922.txt .
- TCP lectura fichero3.txt: 41923.txt .