Una introducción a sistemas Cliente-Servidor (Versión 3)

Ing. Eduardo A. Martínez
eam@itba.edu.ar

1. Objetivos

Es usual que cuando se deba enfrentar el estudio de Sistemas Distribuídos, el primer paradigma (y uno de los más utilizados) que se aprende es el de Cliente-Servidor.

Si bien es bastante fácil entender los conceptos subyacentes y existen implementaciones automatizadas[1], es realmente útil para aquel que recién se acerca al tema realizar una ejercitación totalmente programada a *mano*.

El objetivo, pues, de este pequeño trabajo es guiar al probable alumno en esta ejercitación, dejando además una serie de ejercicios para que el mismo pueda seguir en su aprendizaje.

Para mostrar que no se trata de un tema inabordable, acostumbro comenzar planteando la ejercitación en el viejo MSDOS utilizando como compilador el Turbo C 2.0[2]. Antes de pasar a una plataforma más adecuada[3], se realiza una simulación en MSDOS, que además muestra las características de sincronismo utilizada en la mayor parte de los sistemas Cliente-Servidor.

2. El paradigma Cliente-Servidor

El problema fundamental es hacer que dos procesos (quizás corriendo en dos máquinas distintas) puedan colaborar entre sí para lograr un determinado fin.

Una de las formas más comunes es pensar que un proceso implementa un servicio necesario (el servidor) sobre la base de claras reglas (la interfase de servicio) y uno o más procesos requieren dicho servicio (los clientes) basados en la interfase de servicio.

[1] Rpcgen de Sun Microsystems es la más conocida
[2] Se puede bajar sin costo desde el site de Borland http://www.borland.com
[3] Por ejemplo, Linux

No haremos, en principio, hincapié sobre los métodos de comunicación utilizados y, para comprender los conceptos, supondremos que la comunicación es perfecta.

Esta interacción entre cliente y servidor es de tipo sincrónica, como se observa en la figura siguiente, es decir el cliente solicita un servicio y espera hasta que el servidor responde con el mismo; de manera análoga, el servidor normalmente está esperando a que algún cliente solicite servicio y, cuando así lo detecta, lo realiza, lo responde al cliente y vuelve a esperar otro requerimiento.

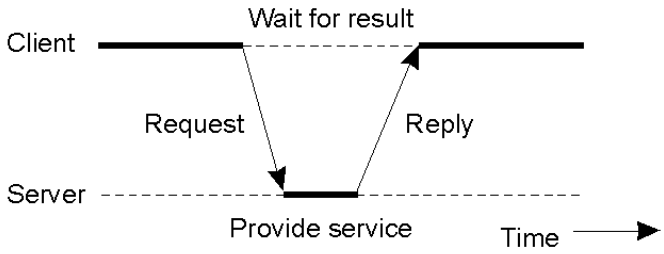

Figura 1 – Secuencia en un sistema Cliente-Servidor

En los códigos siguientes, se muestran esqueletos posibles de cliente y servidor, donde se supone que la llamada a *receive* es bloqueante, es decir, el proceso que la realiza espera (en principio indefinidamente) hasta que se recibe una comunicación desde el otro extremo.

```
struct message m1, m2;

.....
form_request( &m1 );
send( SERVER, &m1 );
receive( SERVER, &m2 );
process_service( &m2 );
.....
```

Código 1 – Estructura del Cliente

```
struct message m1, m2;
long client;
init_server;
forever
{
     client = receive( ANY, &m1);
     provide_service( &m1, &m2 );
     send( client, &m2 );
}
```

Código 2 – Estructura del Servidor

Se puede observar claramente de los dos códigos que el servidor está *bloqueado* continuamente en la recepción, mientras que el cliente está realizando otras tareas; cuando el cliente arma un pedido de servicio y lo envía, es ahora el cliente quien se bloquea luego del envío en el *receive* y desbloquea el servidor, ejecutando el servicio[4].

De esta sencilla estructura,se puede observar también una dificultad: la falta de transparencia que posee este sistema; en efecto, el programador se encuentra frente a la dificultad de deber programar para comunicaciones, entendiendo como son las tramas de comunicaciones (la estructura *message*) y refiriéndose continuamente a primitivas de comunicaciones; también está involucrado en la numeración de clientes y servidores y ni que decir cuando las cosas se compliquen y en realidad, no se trata de un sistema de comunicaciones confiabley deba lidiar con los respectivos errores.

A esta altura nos preguntamos si no existirá una forma más sencilla de trabajar con estos conceptos, específicamente una forma

[4] Sería conveniente que numere las sentencias de ambos códigos y coloque dichos números sobre la Figura 1.

que sea más cercana a la programación convencional y que oculte los problemas de comunicaciones.

En efecto, dicha forma existe y fue planteada por Birrell y Nelson[5] y se denomina ***Remote Procedure Call (RPC)*** o llamada a procedimiento remoto, lo cual analizaremos en el próximo punto.

3. Remote Procedure Call (RPC).

Desde el punto de vista del programador, está acostumbrado que cuando desea efectuar alguna acción, sencillamente llama a un procedimiento previamente programado; sería entonces interesante disponer de un procedimiento[6] que no se sepa realmente donde se ejecuta.

El programador lo utiliza como si estuviese dentro de su mismo proceso, logrando una gran transparencia del sistema de comunicaciones subyacente[7] y utilizando técnicas de programación bien conocidas.

La idea detrás de la implementación es que un llamado a un procedimiento local al cliente[8] en realidad obligue al servidor ejecutar el real procedimiento en forma remota.

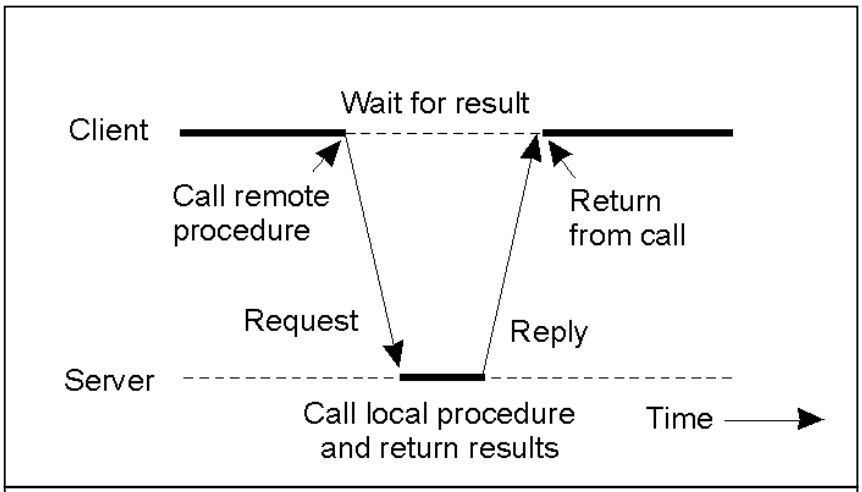

Figura 2 – Secuencia en un Cliente-Servidor mediante RPC

[5] **Implementing Remote Procedure Calls** – *ACM Transactions* – Vol. 2 No. 1 – Abril 1982
[6] En realidad, bajo el nombre de procedimiento se esconden también las funciones. Como se verá dentro de poco, son generalmente funciones.
[7] Si es que existe
[8] Como no puede ser de otra forma.

Obviamente, como se dijo en el párrafo anterior *debe existir un procedimiento local para que el programador pueda realmente invocarlo* y dicho *procedimiento local* actúa en representación de aquel que se encuentra en el servidor; la llamada a este procedimiento local desencadena una serie de eventos hasta que en el servidor se invoca al procedimiento que realmente produce el servicio; esta secuencia se muestra en la Figura 2, que no es otra cosa que la misma Figura 1 a la cual se le ha agregado los rótulos de las respectivas llamadas a procedimientos.

Obviamente entones, el código del procedimiento local debe esconder los detalles planteados como Código 1; esto obliga a que en el cliente exista un nivel inferior de codificación que denominaremos *stub*[9] (que en inglés recuerda que es solamente una parte [la de invocación] del procedimiento que existe en el servidor).

Si queremos además que el procedimiento respectivo en el servidor no posea código correspondiente a comunicaciones, se deberá agregar un nivel de programación en el servidor que permita *ocultar* parte del Código 2; este nivel en el servidor también se llama *stub*.

Obviamente, estos dos niveles forman parte de una capa específica de comunicaciones, que a nivel ISO OSI sería la capa de sesión y que más modernamente en Sistemas Distribuídos, se califica como perteneciente a *Middleware*.

Si ambos *stubs* pertenecen al sistema de comunicaciones, como es obvio, entonces posee un *protocolo* en común que dentro de poco veremos como se forma.

No debemos olvidar que, si bien los códigos de cliente y servidor son de aplicación, por lo tanto, también forman parte del sistema de comunicaciones y también poseen un *protocolo* en común, que no es otra cosa que *los prototipos de las funciones del servidor*.

Por lo tanto podríamos pensar la estructura de cliente y servidor dividida en tres capas, como se muestra en la Figura 3; allí se muestran los dos procesos (el proceso del lado cliente y el proceso del lado servidor), indicando los servicios y los protocolos.

[9] La traducción posee las siguientes acepciones: cabo, trozo, talón, resguardo.

Se puede observar que en el nivel más alto (el de aplicación) el protocolo es provisto por los prototipos de funciones del servidor.

El nivel intermedio (*middleware*) es el nivel de los *stubs*, que en definitiva es el nivel encargado de traducir la invocación local en el lado cliente a una trama de comunicaciones y en el lado servidor tomar la trama de comunicaciones y mediante ella, transformar la misma a una invocación del procedimiento en el lado servidor.

El nivel más bajo está constituído por la capa de *transporte* de la arquitectura elegida.

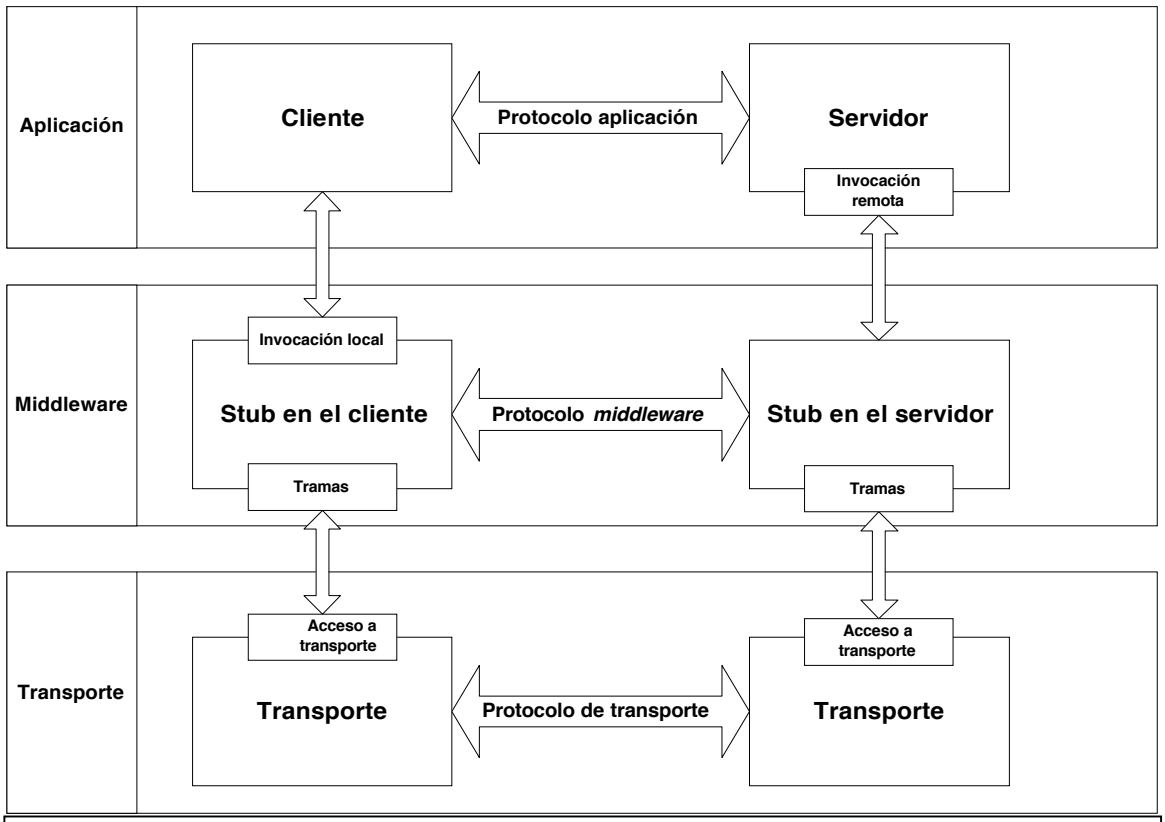

Figura 3: Arquitectura de comunicaciones del cliente servidor por RPC

Se observa que, de esta forma, los códigos de cliente y servidor *son independientes de las capas inferiores*, es decir, el código del cliente es *absolutamente independiente* de la capa de *middleware* y de las capas inferiores de comunicaciones e incluso que existan o no comunicaciones[10], así como el código del servidor es independiente de las capas inferiores.

Por su parte, si bien el protocolo de *middleware* es dependiente de los servicios que provee el servidor (ya que debe colocar la información para lograr dichos servicios en las tramas de transporte), no depende de la forma que el cliente utilice los servicios del servidor ni de la forma particular que estén programados dichos servicios

Intentaremos, en lo que sigue, dar un ejemplo básico de aplicación donde desarrollaremos (en forma muy reducida) los dos niveles superiores (aplicación y *middleware*) en forma paulatina.

4. Ejemplo de aplicación

4.1. Protocolo del nivel de aplicación.

El trabajo comienza sobre la capa de aplicación; en efecto, lo primero que debe determinarse es el protocolo de la capa de aplicación que está fundamentalmente constituído por los prototipos de las funciones (procedimientos) que el servidor ofrece a sus clientes.

El ejemplo que expondremos es un manejador de archivos que, para simplificación del ejemplo, prácticamente establece las mismas llamadas que el sistema de archivos propio del sistema operativo.

En efecto, se han implementado cuatro funciones de acceso a archivo que son las típicas de apertura, lectura, escritura y cierre de archivos; a continuación se muestran los prototipos de dichas funciones:

```
typedef int RD;

RD ropen( const char *pathname, const char *mode );
int rread( RD rd, void *data, int qty );
```

[10] Vea el primer ejemplo de aplicación a continuación

```
int rwrite( RD rd, const void *data, int qty );
int rclose( RD rd );
```

Se ha colocado la letra 'r'[11] delante de los nombres para no tener colisión con las funciones existentes.

La función *ropen* recibe los argumentos a la usanza de *fopen* de la biblioteca *standard* y en vez de retornar un **FILE ***, retorna un entero que identifica en adelante el archivo abierto (denominado RD por *remote descriptor*) o un valor negativo ante error de apertura.

Este valor de remote descriptor es lo que se conoce como una *variable opaca*, es decir, una variable cuyo valor debe ser preservado por el programa del cliente, pero cuyo valor no tiene sentido para quien la preserva[12].

Es justamente este valor el que permite referirse, en las otras tres llamadas, al archivo ya abierto.

La llamada de lectura (rread) recibe el valor del remote descriptor, un puntero a una zona de memoria donde deben depositarse los datos leídos del archivo en cuestión y un entero (qty) que indica la cantidad de bytes a leer del archivo; la función retorna la cantidad de bytes efectivamente leídos o un valor negativo en caso de error.

La llamada de escritura (rwrite) tiene argumentos que significan prácticamente lo mismo que en el caso de rread, salvo en el caso del segundo[13] que indica la zona de memoria donde se encuentran los datos a escribir en el archivo; la función retorna la cantidad de datos efectivamente escritos o un valor negativo en caso de error.

La última llamada produce el cierre del archivo abierto bajo un determinado remote descriptor y retorna un 0 (cero) en caso de cierre correcto y un valor negativo en caso de error.

Es importante a esta altura una nota de aviso: si bien las capas inferiores de comunicaciones deben ser transparentes a la

[11] Por *remote* o remoto
[12] Lo mismo ocurre, normalmente, con el valor de **FILE *** que retorna la biblioteca *standard* o con el valor *fd* que retorna la llamada a *open*.
[13] Observe el adjetivo *const* en el prototipo.

implementación tanto de cliente y de servidor, la elección de los prototipos de funciones deben ser hechos considerando que van a ser utilizados en el caso de un servidor remoto y, por lo tanto, deben ser elegidos de tal manera que no generen un tráfico indeseado en los niveles inferiores. La elección correcta de la partición de funciones entre cliente y servidor tiene una importancia definitiva sobre la eficiencia de comunicaciones.

4.2. Cliente y servidor en un mismo proceso

Una vez establecido el protocolo del nivel de aplicación, lo primero que haremos será establecer el código del servidor y un código de cliente que servirá para la prueba del sistema[14]; una vez establecidos, se unirán mediante el *linker* ambos módulos, demostrando efectivamente que las llamadas del cliente están de acuerdo con los servicios que provee el servidor; en este caso,

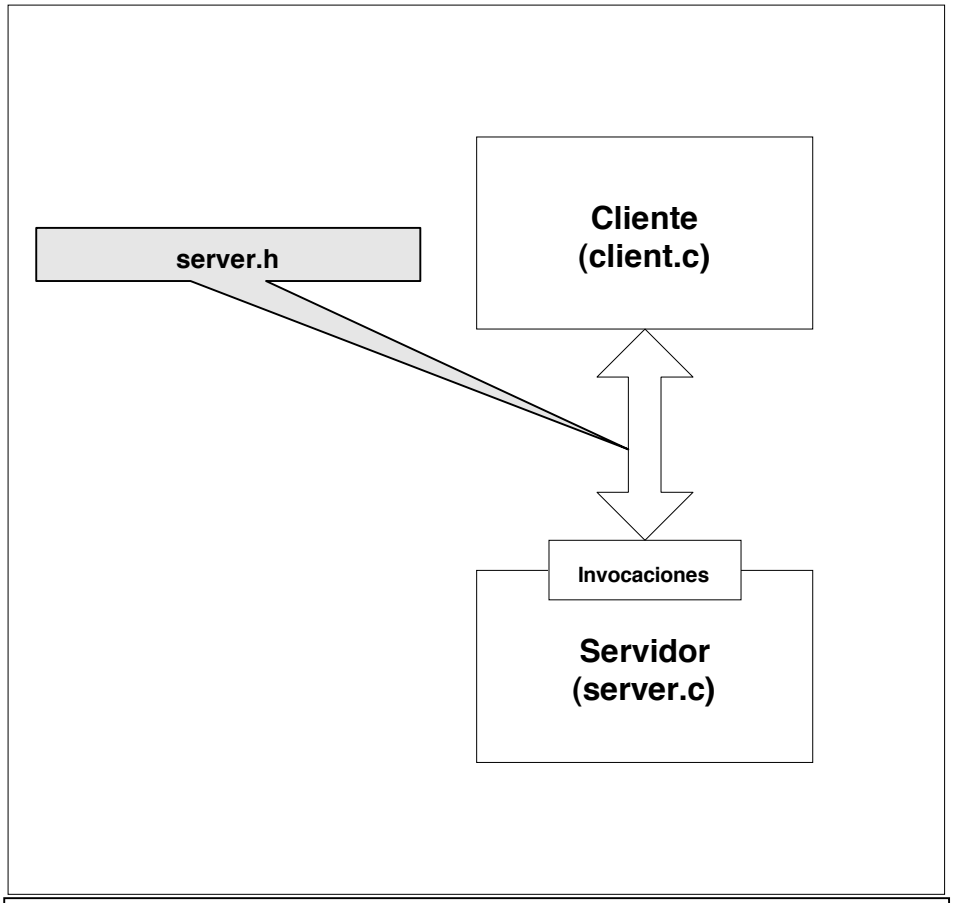

Figura 4: Cliente y servidor unidos en un único proceso

[14] Observe que mientras el servidor conforma un programa dedicado solamente a dicha finalidad, el código del cliente no es único y depende de la aplicación que se quiera realizar.

cliente y servidor estarán unidos en un mismo proceso *sin comunicaciones*.

En la figura 4 se muestra el sencillo esquema del caso: se ha realizado un módulo de compilación separada para el servidor (denominado *server.c*), otro módulo para el cliente (denominado *client.c*) encontrándose los prototipos en el archivo *server.h* que son incluídos en ambos módulos.

En el apéndice, figuran los códigos de cliente y servidor, de los cuales comentaremos someramente su funcionamiento.

4.2.1. Código del cliente

Si bien el código del cliente debería realizar una prueba exhaustiva de funcionamiento, se ha tratado de mantener lo más corto posible de forma de no perder comprensión del problema[15].

Se ve, siguiendo la función *main* que se trata de copiar archivos entre el servidor y el cliente en ambos sentidos; para ello, existen las funciones *put* y *get*.

La función *get* lee un archivo del lado servidor y lo copia al lado cliente; la función *put* realiza la operación contraria[16]. Los argumentos son los nombres de los archivos y, siguiendo la tradición de las funciones de biblioteca de C, la copia es desde el segundo argumento al primero.

La función *get* utiliza las funciones del servidor en cuestión para abrir, leer y cerrar un archivo en el lado del servidor, mientras que utiliza las funciones de la biblioteca *standard* para abrir, escribir y cerrar un archivo en el lado cliente; por su lado, la función *get* realiza lo contrario.

Obsérvese que el *buffer* de lectura y escritura se ha mantenido de un tamaño pequeño (32 bytes) de manera de poder observar claramente las múltiples transacciones entre cliente y servidor.

[15] En general, los programas de prueba exhaustiva no son muy agradables en su diagramación.
[16] Se ha tenido especial cuidado en la elección del nombre de los archivos, en el sentido que no estén superpuestos, ya que muchas de las pruebas se realizará bajo el mismo sistema de archivos (como en este caso).

4.2.2. Código del servidor

El código del servidor posee la definición de las cuatro funciones establecidas en *server.h*, más una función auxiliar privada y la estructura de datos que mantiene los datos de los archivos abiertos.

Como se verá, la estructura de datos aloja en un arreglo los valores retornados por la función *fopen*[17] de la biblioteca *standard* en una cantidad MAX_FILES.

En efecto, cada vez que se requiere abrir un archivo con *ropen*, se busca lugar disponible[18] en el arreglo para alojar el valor retornado por la llamada a *fopen*.

En caso que exista lugar, se realiza dicha llamada y se aloja el valor obtenido en la ranura disponible del arreglo, retornando el índice del arreglo como el valor de *remote descriptor*.

El código del resto de las funciones comienza de la misma manera, es decir validando el *remote descriptor* recibido tanto para verificar que el rango esté comprendido dentro de los límites del arreglo, como para determinar si pertenece a un archivo que esté abierto[19].

Una vez determinada la validez del valor de *rd*, solamente se toma el valor del *handle* de la biblioteca *standard* y se lo utiliza en una llamada equivalente de la misma y retornando de cada una de las funciones.

Al construir el código del servidor, han aparecido condiciones de excepción que se han codificado como valores de retorno negativos en todos los casos; dichos valores se han codificado con un *enum* en el archivo *server.h*, ya que el mismo contiene *todos los convenios de servicio que provee el servidor*, el código de este archivo de encabezamiento también se encuentra en el apéndice.

4.2.3. Prueba y conclusiones

[17] De hecho, es un arreglo de FILE *
[18] La convención es que el lugar está disponible siempre que el valor alojado sea NULL.
[19] Esta verificación se realiza mediante la función privada *not_verify_rd*.

Se deja al lector copiar el código con su editor preferido, compilarlo, generar el ejecutable y probarlo; lo que puede esperar queda claro del mismo código: la aparición, en este caso, de dos archivos en el mismo directorio llamados *client1.c* y *client2.c*[20].

Si tiene algún utilitario que permita determinar diferencias entre dos archivos, úselo para verificar que los contenidos de *client.c*, *client1.c* y *client2.c* son idénticos[21].

Una vez probados cliente y servidor, *tomaremos el compromiso de no cambiarlos hasta terminar esta ejercitación*, para mostrar que efectivamente la aparición de las capas de *middleware* y de comunicaciones inferiores (representada en el diagrama de niveles por la capa de transporte) no necesitan cambio de dichos módulos y, por lo tanto, son absolutamente transparentes a ellos.

4.3. *Middleware* y *stubs*.

Nos enfrentamos ahora con la construcción del nivel inferior.

Desde ya, como pasa en comunicaciones, deberíamos ponernos de acuerdo con el protocolo que manejará esta capa, ya que después de todo, forma la comunicación virtual en este nivel.

Para ello, deberíamos analizar primero que es lo que debe codificar este protocolo: a poco de observar, nos damos cuenta que en el sentido de cliente a servidor, debe codificar la función que debe ser llamada en el servidor, así como los valores de los parámetros de entrada de dicha función.

Una vez producido el servicio, en el sentido servidor a cliente deben codificarse los valores de salida de la función, uno de los cuales es el valor de retorno de dicha función.

[20] Tenga mucho cuidado: se está copiando uno de los archivos fuentes del proyecto (*client.c*) por lo cual, ante un error puede perderlo. Le sugiero que haga una copia de respaldo de los tres archivos antes de comenzar las pruebas.

[21] Si trabajó en MSDOS, verá que no son idénticos en cuanto a longitud, ya que muy probablemente su editor de texto le haya agregado un **^Z** al final del archivo, que es innecesario y que los programas de copia escritos en C no lo hacen.

Intentemos realizarlo para la más sencilla de las funciones de las cuatro planteadas, que evidentemente es la última: *rclose*.

Identificamos en este caso los siguientes valores:

De cliente a servidor:

- Función a ejecutar: *rclose*
- Valor del *remote descriptor*

De servidor a cliente:

- Valor de retorno de la función o *status*.

Primero determinaremos como debe codificarse la función a ejecutar; a poco de razonar encontraremos que mediante un número entero, pueden codificarse una de cuatro funciones en cada instancia, por lo cual podremos hacer la siguiente tabla[22]:

Función	Código
ropen	0
rread	1
rwrite	2
rclose	3

Tabla 1 – Asignación de códigos de funciones

Una vez resuelto este problema, se nos ocurre tomar una hoja de papel cuadriculado y dibujar las tramas de ida y vuelta para esta función en particular, como se muestra en la figura 5.

Como se podrá observar, los protocolos están realizados colocando el código de operación de la función a realizar en el servidor en el protocolo de cliente a servidor; aunque parece innecesario, se ha repetido en el protocolo de respuesta, ya que sirve, cuando menos, para verificación que la respuesta del servidor corresponde al requerimiento enviado por el cliente.

También, se han copiado el valor de *rd* que recibe la función *rclose* del *stub* del cliente directamente sobre el campo correspondiente, así como en el retorno al cliente, el *stub* del servidor ha copiado el valor de retorno de la función *rclose* en el servidor sobre la trama de respuesta.

[22] Los valores elegidos son arbitrarios. Por comodidad de programación, conviene que empiecen de 0 (ya que programamos en C) y sean correlativos.

Podríamos continuar de esta forma resolviendo el problema en una forma un tanto gráfica, para después hacer una codificación *ad hoc* de dichos resultados.

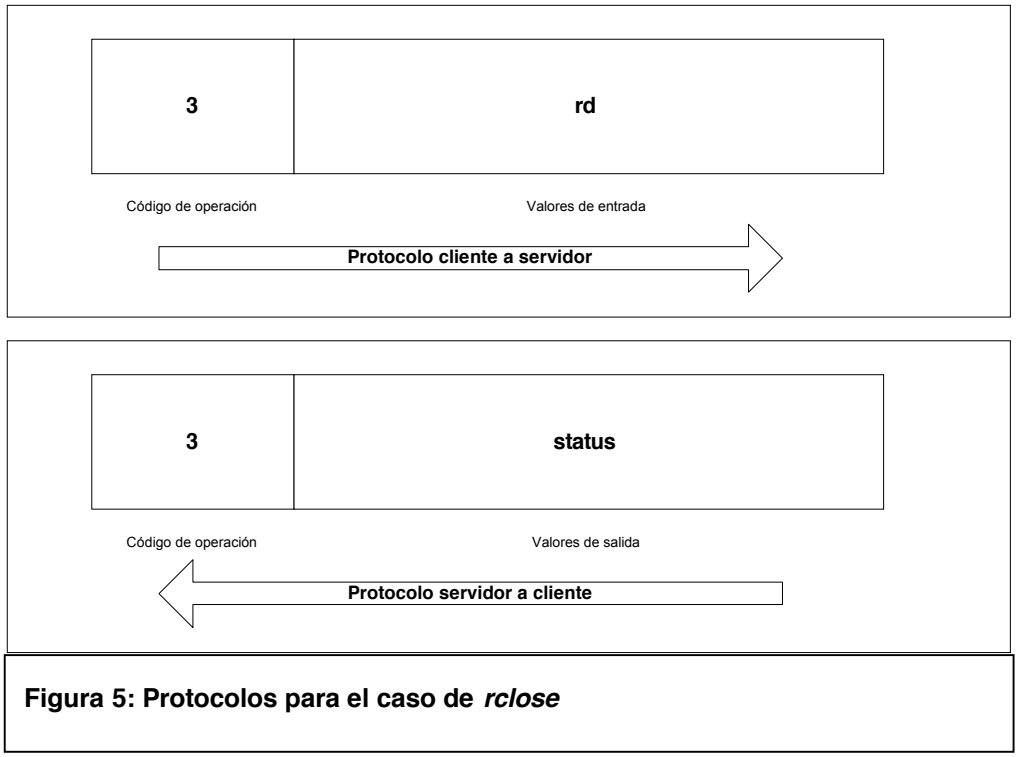

Figura 5: Protocolos para el caso de *rclose*

Sin embargo, nos damos cuenta que en un caso real el problema sería absolutamente tedioso, muy propenso a errores y de muy difícil mantenimiento.

Vamos, entonces, a razonar por otro camino.

La función del *stub* del cliente[23] es tomar la llamada a función por la cual ha sido activado y codificar una estructura de datos en memoria que represente la función remota a ejecutar así como copie los valores de entrada a dicha función.

Una vez realizada esta acción[24], se presenta a la capa de transporte solicitando su comunicación hacia el otro extremo.

Este mismo tipo de acción la realiza también el *stub* del servidor cuando quiere comunicar al cliente los resultados del servicio: coloca en una estructura de datos en memoria el código

[23] Veremos inmediatamente que algo similar ocurre en el *stub* del servidor.
[24] que se denomina en inglés *data marshalling*.

de la función ejecutada así como una copia de los valores de salida de dicha función.

A la inversa, cuando la capa de transporte del lado servidor recibe una trama, la misma es presentada al *stub* de dicho lado mediante una estructura de datos en memoria que dicho *stub* reconoce de acuerdo a las reglas del protocolo establecido y, mediante el código de la función y la copia de los argumentos de entrada a la misma, organiza el llamado a función del servidor.

Del lado del cliente, cuando la capa de transporte recibe una trama es presentada al *stub* del cliente como una estructura de datos en memoria, donde reconoce como válido el código de la función efectuada en el servidor, levanta la copia de los valores de salida de dicha función y retorna al código del cliente.

Reconocemos, entonces, que dicha representación en memoria

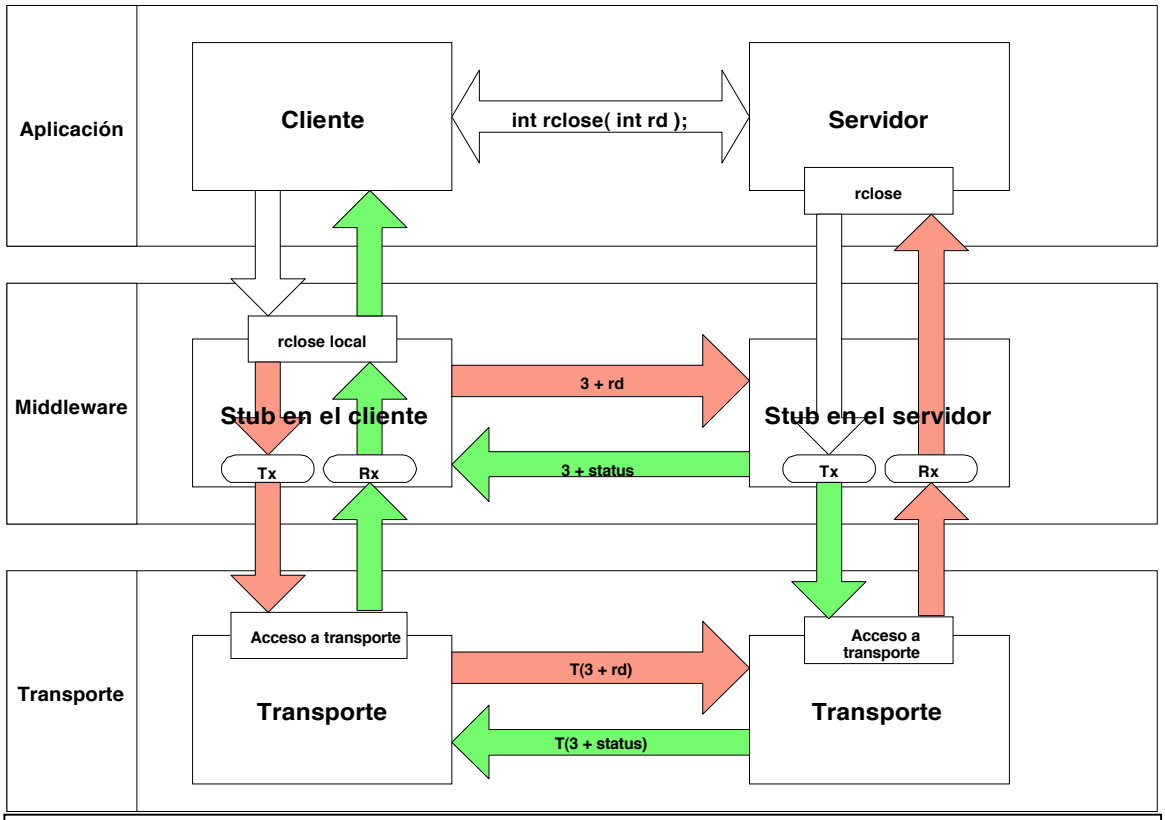

Figura 6: Arquitectura, servicios y protocolos para el caso de *rclose*.

debe ser alojada en *buffers* de comunicaciones a nivel de *stubs*, buffer que se encuentran en la interfase entre *stubs* y capa de transporte.

Podemos, entonces, alojar cuatro *buffers*, dos en cada *stub*, correspondiendo uno a transmisión y el otro a recepción.

En el diagrama de la Figura 6 se explotan con detalle las acciones y los buffers involucrados.

Por lo tanto, lo que debemos establecer clara y formalmente es la estructura de todas las tramas de middleware que pueden presentarse en los buffers de transmisión y recepción de cualquiera de ambos lados y si es posible, la determinación automática del dimensionamiento de los mismos.

Realizaremos una simplificación adicional en la estructura de buffers; si el funcionamiento de cliente-servidor es el que hemos establecido hasta ahora, es decir sincrónico, significa que las siguientes aseveraciones son correctas:

Un cliente no formulará un pedido al servidor si no ha obtenido la respuesta del servicio anterior.
El servidor no originará una comunicación a un cliente, si no es que el cliente ha pedido previamente un servicio.

Por lo tanto, se puede notar que cuando el cliente usa el buffer de transmisión, no está esperando recibir y viceversa, cuando está esperando recibir, no va a usar el buffer de transmisión.

Algo similar ocurre con el lado servidor: con referencia a un cliente en particular, el servidor no transmitirá si primero no recibe y una vez que recibió, su buffer de recepción no será utilizado.

Por lo tanto, podemos en este caso colapsar ambos buffers (el de Tx y Rx de cada lado) en uno solo[25].

4.3.1. Generación del protocolo del nivel de *stubs.*

Estamos en condiciones ahora de establecer una metodología de dimensionamiento de dichos *buffers* así como de establecimiento de los protocolos a utilizar por el nivel de *middleware*.

[25] Obsérvese que en el *stub* del servidor deberá existir de todas maneras un *buffer* por cliente.

En el apéndice se encuentra el archivo *clserv.h* que pretende resolver ambos problemas.

Como se observará, es un archivo que sólo contiene definiciones de tipos simples y estructurados, con lo cual puede ser convenientemente incluido en archivos fuentes, con la seguridad que no producirá doble código en el momento de compilación.

Para comprender como se arma, debemos comenzar por el final del archivo, donde se declara el tipo CLSVBUFF[26] como un tipo que proviene de una estructura, la cual está formada por un miembro **opc** de tipo **OPC** y un miembro **data** de tipo **DATA**; lo que indica esta definición es algo así como *el protocolo está formado por un código de operación + un área de datos*.

La pregunta es ¿cómo está formada el área de datos?. A poco de pensar, nos daremos cuenta que dicha área de datos depende del código de operación y, para el mismo código de operación, si se trata de tramas de cliente a servidor o de servidor a cliente.

Como existen cuatro códigos de operación, determinamos entonces que pueden existir ocho tipos de campos de datos en la comunicación; como en un determinado momento, un *buffer* de comunicación estará ocupado por uno solo de ellos, entonces la definición del tipo **DATA** deberá ser dado por una unión.

Si miramos en el archivo *clserv.h* por encima de la estructura que define CLSVBUFF, veremos que está definido el tipo DATA mediante una unión, la cual tiene claramente ocho miembros, cuatro para las tramas del protocolo correspondientes a la comunicación del cliente alservidor y cuatro para las correspondientes del servidor al cliente.

Para conveniencia de notación los tipos de cada uno de los miembros han sido nominados por los prefijos CLSV (para las tramas del cliente al servidor) y por SVCL (para las tramas del servidor al cliente) y los sufijos corresponden a la función involucrada, así por ejemplo:

[26] Como se dará cuenta, significa Client SerVer BUFFer

- SVCL_RCLOSE corresponde al tipo de DATA para las comunicaciones del servidor al cliente en el caso de la respuesta de *rclose*.
- CLSV_ROPEN corresponde al tipo de DATA para las comunicaciones del cliente al servidor en el caso de la invocación a *ropen*.

Para mantener la comodidad de notación, los miembros de la unión poseen el mismo nombre de su tipo pero con minúsculas.

Estos ocho tipos corresponden, entonces, a las definiciones de cada una de las tramas y se encuentran como estructuras de C; tomemos el caso de *rclose* ya analizado en forma gráfica.

Encontraremos que CLSV_RCLOSE es el tipo de una estructura que tiene como único miembro a **rd** de tipo **RD**, justamente el único argumento que posee la función.

A su vez, la estructura SVCL_RCLOSE también tiene un único miembro que es denominado **status** y que no representa otra cosa que el retorno de la función ejecutada en el servidor.

4.3.2. Pasaje por valor y por referencia

No hemos elegido por casualidad como primera función a analizar *rclose*; en efecto, de las cuatro funciones que hemos propuesto, es la única que recibe parámetros por valor y también retorna resultados por valor (de hecho, uno en cada caso).

Si analizamos las otras funciones encontraremos las siguientes condiciones de contorno:

Función	Por valor	Por referencia
ropen	[retorno]	filename, mode
rread	rd, qty,[retorno]	data
rwrite	rd, qty,[retorno]	data
rclose	rd,[retorno]	

Si analizamos ahora cuales parámetros son de entrada a la función y cuales son de salida, nos encontraremos con:

Función	Entrada	Salida
ropen	filename, mode	[retorno]
rread	rd, qty	data, [retorno]
rwrite	rd, qty, data	[retorno]
rclose	rd	[retorno]

Vemos, entonces, que un parámetro pasado por referencia puede servir tanto para entrada de la función (caso de **data** en *rwrite*) o de salida (caso de **data** en *rread*)[27].

Los parámetros pasados por valor *siempre* son *exclusivamente* de entrada y sólo existe un parámetro de salida por valor que es el retorno de la función.

Como se sabe, la única forma de pasaje por referencia en el lenguaje C es mediante el pasaje por valor de un puntero a la referencia y, por lo tanto, nos podemos hacer la pregunta ¿tiene sentido pasar en este caso el valor del puntero del cliente al servidor?. ¡Obviamente, no!

Fácilmente se puede observar que la dirección de memoria a la cual apunta el puntero en uno de los procesos no tiene sentido en el otro proceso (que posiblemente posee área de datos separada del primero) y, por lo tanto, no puede apuntar a ningún dato relevante.

Obviamente, entonces, dicho puntero debe ser reemplazado *por una copia de los datos válidos a los cuales apunta* y colocado en la trama correspondiente.

Observemos, entonces, la estructura denominada CLSV_RWRITE, donde el cliente debe pasar los datos a escribir en el archivo remoto; como se observa de la misma, se encuentran los dos parámetros recibidos por valor (*rd* y *qty*) y un arreglo de caracteres denominado *data* en el cual se van a copiar los datos válidos (en cantidad *qty*) a transferir al lado servidor[28].

[27] También podría servir de entrada y salida, pero no tenemos ejemplo para este caso tan sencillo.
[28] Dicho arreglo debe dársele una dimensión máxima, la cual debe verificarse que no sea excedida (MAX_DATA en este caso).

Véase la ubicación de este arreglo en la estructura: es el último de los miembros, ya que si bien se trata de un campo de longitud fijada en el momento de compilación, en el momento de ejecución puede no estar totalmente completo y al estar ubicado al fondo de la trama, puede realizarse un truncado en el momento de ejecución para evitar transferir bytes indeseados al lado servidor.

Esta discusión es equivalente para el lado servidor en el caso ce SVCL_RREAD con el arreglo llamado también *data*.

El último caso a analizar es el de la estructura CLSV_ROPEN; en este caso, las variables recibidas por referencia son dos (*filename* y *mode*) y, por lo tanto, la discusión anterior nos llevaría a que existen dos campos, en el momento de ejecución, de tamaño variable.

Ni bien nos detengamos a razonar, nos daremos cuenta que uno de ellos debe ser transmitido en su total longitud independientemente de la longitud válida en el momento de ejecución; la solución es buscar cual de los dos produce menos daño a la hora de razonar sobre eficiencia de comunicaciones.

Obviamente, el campo más pequeño es el de *mode* ya que el valor del mismo, a lo sumo, puede ser “r+b” o “w+b”, lo cual implica tres caracteres.

Otra pregunta es ¿cómo saber cual es la longitud real de los campos variables?

En el caso de CLSV_RWRITE, un miembro de la misma estructura (*qty*) indica cuantos bytes son válidos en el arreglo *data*; en el caso de SVCL_RREAD, también un miembro de la misma estructura (*status* que no es otra cosa que la copia del valor retornado por la función *rclose* ejecutada en el servidor) indica cuantos bytes son válidos sobre el arreglo *data*[29].

En el caso de CLSV_ROPEN, el único que conoce la longitud real de los *strings* es el *stub* del cliente, ya que los recibe como tal (*strings*) pero no existe otro miembro de la misma estructura que lleve dicha información al servidor, con lo cual quedan dos soluciones posibles:

[29] Desde ya, solo en el caso que *status* no posea valor negativo.

- Agregar dos miembros más a la misma estructura que indiquen los valores válidos de ambos campos variables.
- Colocar un terminador sobre cada arreglo que indique cual es el último byte válido en cada caso.

Como podrá observar, se ha utilizado este último método para la codificación de CLSV_ROPEN.

Por último, vemos que por la forma de construcción de la estructura CLSVBUFF, su dimensión en bytes será la que corresponde a la mayor de todas las tramas intercambiadas por los *stubs* y, por lo tanto, servirá para crear los *buffers* de cada uno de los *stubs.*

4.3.3. Una digresión sobre estructuras

Seguramente si Ud. ha trabajado en programación en C se estará preguntando si esta forma de especificar las tramas realmente es correcta.

En efecto, hemos dado por supuesto *que los miembros de una estructura son absolutamente adyacentes entre sí* para poder armar una trama que no tenga *agujeros* entre sus campos.

Esto no es absolutamente correcto, ya que un compilador de C trata de ser eficiente en su armado de datos en memoria, eficiencia que la mide por la cantidad de accesos a memoria que un dado procesador debe hacer para leer o escribir completamente uno de los miembros de la estructura.

Salvo que el procesador sea de 8 bits en su *bus* de datos[30], el alineamiento del miembro de la estructura en cuestión puede incidir desfavorablemente en la eficiencia de direccionamiento del procesador.

Por lo tanto, un compilador de buena calidad calculará que espacio *libre* debe dejar entre miembros para que el

[30] Lo cual sería impensable a esta altura de los acontecimientos, salvo que Ud. esté haciendo programación *embedded* para aplicaciones de pequeña dimensión.

direccionamiento del procesador a todos y cada uno de los miembros de la estructura no necesite más ciclos de máquina que los necesarios.

Por tratarse este punto de una optimización a nivel del compilador, seguramente existen opciones en la línea de comando[31] que permiten deshabilitar esta optimización para lograr que el compilador *empaquete* los miembros de la estructura sin *agujeros* intermedios; sin embargo, para procesadores de 32 o 64 bits de datos, varios compiladores imponen un límite mínimo de alineamiento de 16 bits.

Por esta razón, podrá observar que en la definición de *clserv.h* no existen miembros de estructuras que sean inferiores a *short*[32] (véase el caso de tipo OPC).

4.3.4. Una digresión sobre tipos.

Detengámonos un momento y recapacitemos sobre que hemos construído, para observar la validez más general de la solución.

Tomemos la llamada más sencilla de todas(*rclose*) y veamos que intentamos hacer y que hemos logrado en realidad.

Lo que intentamos hacer es que una llamada a *rclose* en el *stub* reciba el valor de un entero (pongamos por caso 4) y que a través del mecanismo de RPC, la función remota *rclose* sea llamada con el mismo valor del entero (4 en nuestro caso).

Diremos, entonces, que semánticamente las llamadas expresan lo mismo.

¿Qué hemos hecho en realidad? Hemos tomado la representación del número 4 en el cliente, hemos colocado dicha representación sobre una trama, la hemos transferido al servidor y supusimos que esa representación transferida al servidor e interpretada por el *stub* del servidor será transferida

[31] O sentencias del preprocesador de tipo *#pragma*
[32] Naturalmente, lo dicho no es válido para arreglos: los miembros que son arreglos son siempre adyacentes si no se quiere violar la definición de arreglo.

como el número 4 en la llamada a la función *rclose* del servidor.

¡Nunca más alejado de la realidad! Supongamos que la máquina del cliente es una PC corriendo MSDOS, donde la representación de un entero es de 16 bits y se comunica con una máquina (quizás igual a la del cliente) pero corriendo Linux, donde la representación de un entero es de 32 bits.

El resultado es que el cliente colocará dos bytes sobre la trama con la representación del número 4 (para el cliente) y el servidor intentará leerlo como la representación de un número entero de cuatro bytes, dando resultados erróneos.

Peor aún, en otro escenario podríamos considerar que una de las máquinas es una PC con entero representado en 32 bits y la otra es una MAC donde el entero es también de 32 bits, pero la representaciónen memoria, aún siendo en complemento a 2, tiene los bytes en distinto orden; nuevamente los resultados serán erróneos.

¿Cuál es la solución definitiva? No transferir solamente representaciones internas de cada una de las máquinas en la trama sino transferir también información de interpretación semántica.

Como Ud. puede ver, este tema[33] escapa a una presentación sencilla como la que estamos haciendo.

Por lo tanto, supondremos por ahora dos hechos para mantener el objetivo de este trabajo dentro de sus límites:

- La arquitectura de las máquinas involucradas será la misma, de manera de suponer que el orden de los bytes también es el mismo.
- Con referencia al tamaño de cada uno de los tipos involucrados, trabajaremos con definiciones

[33] El nivel de la arquitectura de comunicaciones de acuerdo a ISO OSI que se encarga de estos temas se denomina Capa de Presentación y se encuentra dentro de lo que modernamente se denomina *Middleware*. Si Ud. analiza las herramientas de generación automática de stubs (como rpcgen) verá que ellas se encargan de colocar la información semántica en las tramas a partir de la definición semántica que el usuario especifica a través de IDL (Interface Definition Language).

reconfigurables de acuerdo a la arquitectura de la máquina y los supuestos del sistema operativo.

Empezaremos, por lo tanto, generando un archivo de inclusión que es llamad *mytypes.h* y que, como es usual, lo encontrará en el apéndice.

En este archivo se define un tipo RINT_T que será utilizado como entero a los fines de los *stubs*. Obsérvese que la definición lo hace equivalente a un *short int* de manera de ocupar 16 bits.

En el caso (no muy común, por cierto, y por mí nunca visto) que en alguna instalación un *short int* no sea de 16 bits, *sólo deberá cambiarse esta definición en este único archivo* para obtener los resultados deseados.

Concordantemente, se ha cambiado *server.h* de manera de reflejar este cambio (muy especialmente por la definición de RD); el archivo *server.h* modificado se incluye en el apéndice.

4.4. Implementación de stubs.

Emprenderemos ahora la implementación de los *stubs* teniendo como base su protocolo que, de forma indirecta, está expresado por el archivo *clserv.h*.

Debemos avisarle que se ha mantenido el manejo de errores y excepciones del programa en un mínimo para mantener legibilidad; sin embargo, si Ud. piensa usar estos esqueletos en un proyecto, le recomiendo que le agregue el código necesario para hacerlo seguro.

4.4.1. Stub del cliente.

En el apéndice podrá encontrar el módulo denominado *clstub.c* que corresponde al código del *stub* en el cliente.

Obsérvese, como primera medida, los archivos de inclusión: se trata de *server.h* (ya que tiene los prototipos del servidor que serán representados en el cliente por este

stub[34]), *clserv.h* ya que define el protocolo del nivel de *stubs* y *physic.h* que representa las llamadas a la parte baja de la arquitectura de comunicaciones (representada en la figura de arquitectura por la capa de transporte).

Existen en el módulo sólo cuatro funciones públicas que corresponden a las cuatro funciones de servicio implementadas en el servidor y una función privada.

Se ha definido a un nivel privado el único *buffer* de comunicaciones con la capa de transporte en función del tipo CLSVBUFF que se ha denominado *clsvbuff*.

Tomemos como primer ejemplo la función más sencilla: *rclose*; en ella se copia sobre el miembro *rd* de la estructura *clsv_rclose* de la unión *data* del *buffer* el valor de *rd* que es argumento de la función; como ya se ha discutido ampliamente, es el único argumento que necesita la función *rclose* para poder proceder.

Acto seguido, se debe enviar la trama al servidor y esperar que retorne la respuesta *sobre el mismo buffer*, acción que realiza la función *send_rcv* que es privada de este módulo.

Una vez recibida la información del servidor, se extrae del *buffer* el miembro *status* de la estructura ahora presente en dicho *buffer* la cual es retornada por *rclose* del *stub*.

Como podemos ver, la adecuada diagramación del protocolo en términos programáticos dado por el archivo clserv.h ha resultado en un código compacto, simple, claro y muy fácil de mantener.

Tomemos, como caso extremo, la función *rread* que quizás es la más compleja; en este caso, el armado del *buffer* de transmisión implica depositar el valor de dos campos de la trama (o miembros de la estructura CLSV_RREAD) sobre el *buffer* en la zona *data* y recoger también dos campos (miembros de la estructura SVCL_RREAD).

[34] Aparentemente, es innecesario incluirlos, ya que las funciones se definirán en este mismo módulo. Sin embargo, forma parte de la *programación defensiva* ya que permite que el compilador verifique que las funciones definidas coinciden con las declaradas en forma externa.

Como es de esperar varias sentencias relacionadas con esta zona (*data*) con dos criterios distintos (el del cliente y el del servidor) se han definido dos punteros automáticos (*pc* y *ps*) y se han inicializado sobre *data* del *clsvbuff* pero con representación del cliente (para *pc*) y con representación del servidor (para *ps*)[35].

Se copian sobre la trama del cliente al servidor los valores de *rd* y *qty* recibidos como argumentos de la función, se envían al servidor y se espera la respuesta del mismo.

Se rescata el valor de *status* retornado por el servidor y, si el mismo es mayor que 0, se copia sobre el *buffer data* especificado en la llamada a *rread* desde el cliente los bytes que se encuentren en la zona de *data* de la trama recibida del servidor en una cantidad dada por *status*.

Lo último que queda por hacer es, sencillamente, retornar *status* como resultado de la función del *stub* del cliente.

Pude ahora Ud. investigar las otras dos funciones detalladamente hasta tener claro las acciones que se desarrollan visto desde el lado del cliente.

4.4.1.1. La función *send_rcv*.

Veamos ahora que realiza la función *send_rcv*.

Para ello, analicemos como se ha dividido el trabajo.

Las cuatro funciones analizadas en el *stub* del cliente se dedican, fundamentalmente a tratar el manejo del campo *data* del *buffer clsvbuff*, tanto en el armado para enviar al servidor, como en el desglose cuando se recibe del servidor.

El manejo del agregado del código de operación, el envío y la recepción es atributo de la función *send_rcv* lo cual permite su reusabilidad dentro del módulo.

[35] Sería interesante que, para mayor comprensión, en el momento de prueba imprima a esta altura el valor de los punteros para darse cuenta que la dirección a la cual apuntan *es exactamente la misma*. La razón de los dos punteros es una exclusiva razón de *semántica* respecto de la unión de tipo DATA.

Por ello, la función *send_rcv* recibe tres argumentos:

- Un puntero al *buffer*.
- El código de operación a colocar.
- La cantidad de bytes que la función llamante ha colocado sobre la unión *data*.

Obsérvese que, por lo tanto, la función *send_rcv* se hace independiente de la semántica de lo que se encuentra en *data* y sólo le interesa la cantidad de bytes útiles a transmitir en dicha zona.

Ingresemos, ahora, en el código de la función *send_rcv*; lo primero que observamos, es que dicha función coloca el código de operación como primer miembro del *buffer*.

A continuación, hace una llamada al módulo inferior (identificado como *physic* en el código y como *transporte* en el diagrama de arquitectura), pasando el puntero al *buffer* y la cantidad de bytes a transmitir.

Obsérvese que esta cantidad de bytes es la pasada por la función invocante más el tamaño ocupado por el código de operación copiado por *send_rcv*.

Inmediatamente, se pasa a la función *receive_packet* también alojada en el módulo de comunicación inferior, indicándole que lo que se reciba de comunicaciones debe ser colocado *en el mismo buffer recibido por send_rcv*, indicando el segundo argumento cual es la cantidad máxima de bytes que posee dicho *buffer* para que no sea sobrescrito en un caso de error de comunicaciones; esta función retorna, además, la cantidad total de bytes que ha colocado en el *buffer*.

La función *send_rcv* termina, sencillamente, retornando la cantidad de bytes efectivos que se encuentra en el miembro *data* del *buffer*.

Es oportuno ver cuales son los prototipos del módulo inferior de comunicaciones (archivo de inclusión *physic.h*).

Allí vemos los prototipos de *send_packet* y de r*eceive_packet*; puede observar que los tipos de los punteros son *void* indicando, definitivamente, que la función de comunicaciones inferior utilizada *es independiente del contexto superior* y su única finalidad es transferir una cantidad de bytes *en forma confiable*.

Si bien *send_packet* no necesitaría un valor de retorno, se ha agregado por completitud y, en condiciones normales, retorna la cantidad de bytes realmente transferidos[36] o la posibilidad de retornar error mediante un valor negativo.

4.4.2. Stub del servidor

Veamos ahora el código del *stub* en el servidor, que se encuentra en el apéndice.

Como ya se ha dicho anteriormente, la única finalidad del servidor es esperar solicitud de servicio de su cliente[37], por lo cual se trata de un proceso cuya finalidad está preestablecida.

Veremos que los archivos de inclusión son los mismos que en el *stub* del cliente y que se declara un *buffer* privado con el mismo nombre que en el cliente[38].

Busquemos, al final del archivo, la función *main*[39] que sencillamente, se sienta en un lazo infinito a ejecutar la función *do_server*.

La función *do_server* se encuentra inmediatamente arriba de ésta; la misma representa las acciones que se realizan *para un dado servicio del servidor*.

Básicamente, implementa lo realizado: comienza llamando a la función *receive_packet*, pasándole la dirección del*buffer* de recepción y la cantidad de bytes máxima a recibir.

[36] Que siempre debería ser la misma cantidad solicitada para transferir.
[37] O clientes, en un caso más general.
[38] Aunque, obviamente, no es el mismo.
[39] Que es la única pública del módulo.

Cuando se recibe algo del cliente, la función vuelve con la trama recibida en el *buffer* retornando la cantidad de bytes recibidos.

A continuación, se llama a la función *process_server*, que representa *todo* lo que debe hacer el *stub* del servidor y el servidor mismo (exceptuando las comunicaciones); hagamos un recuento de lo que esta función hará:

- Recibe la trama como vino del cliente
- En función del código de operación, genera los argumentos (de entrada) en forma local y llama a la función del servidor.
- Como resultado del retorno de la función del servidor, arma los valores de retorno (o salida) *sobre el mismo buffer*.
- Retorna la cantidad de bytes efectivos colocados en el *buffer* y que son necesarios que sean transmitidos al cliente.

Obviamente, muy poco le queda ya por hacer a *do_server*: sencillamente llamar a *send_packet* para que transfiera los contenidos del *buffer* al cliente.

Este ejemplo es uno de funcionalidad estricta: la función *do_server* no pretende hacer más que lo que le corresponde: ocuparse fundamentalmente de las comunicaciones contra la capa inferior y llamar a procesar *lo que se encuentre* sin necesidad de mayor interpretación.

Demos ahora un paso más hacia el interior: la función *process_server.*

Como se dijo en la enumeración anterior, debe realizarse un procesamiento de acuerdo al código de operación, por lo cual la responsabilidad de esta función está vinculada estrictamente con las decisiones basadas en el código de operación; por lo tanto, separa el código de operación y, en función de él, llama a una de cuatro funciones distintas de procesamiento.

Esto es más que adecuado: el código de operación no sólo define que función *en definitiva* del servidor debe invocarse,

sino que debe saber cómo interpretar los campos de la trama de recepción para armar los argumentos de llamada a la mencionada función y cómo debe representar los valores de retorno de la función para ponerlos en la trama de salida.

Si bien se podría haber usado un *switch*, nos pareció más elegante hacer una llamada indirecta a través de un arreglo de punteros a función llamado *proc* y que está inicializado con cuatro punteros a función:

- process_ropen
- process_rread
- process_rwrite
- process_rclose

Obsérvese detenidamente los tipos de los argumentos de todas estas funciones: el primero es de tipo DATA *, (es decir, apunta genéricamente al punto de la trama justamente después del código de operación) y el segundo es la cantidad de bytes que existe en dicha parte (excluyendo el código de operación); cada una de estas funciones retorna la cantidad de bytes colocados efectivamente en el *buffer* para retransmitir al cliente.

Una vez retornado de una de estas funciones, *process_server* coloca el código de operación de respuesta sobre la trama de salida y retorna la cantidad total de bytes a transmitir al cliente.

¿Qué significa SERVER_OFF que se agrega al código de operación de salida? Si bien el código de operación de respuesta al cliente está más por completitud que por otra razón, muchas veces observaremos tramas mediante un analizador de comunicaciones en alguna parte del circuito de comunicaciones físicas o lógicas[40] y sería interesante reconocer el sentido de la trama[41] lo cual puede reconocerse muy fácilmente por su encabezado, supuesto que no sea el mismo en los dos sentidos.

[40] Veremos más adelante un ejemplo.
[41] Cliente-Servidor o Servidor-Cliente

Si consulta nuevamente *clserv.h* verá que existe un *enum* que define los valores de los códigos de operación y, además, una definición llamada justamente SERVER_OFF[42] y que, en el listado que figura en el apéndice, figura como 0[43]; si necesita cambiarla, sencillamente puede recurrir a esta definición y darle un valor reconocible[44].

Por último, las cuatro funciones de procesamiento son directas y no creo que haga falta describirlas[45]

Las funciones de la capa inferior están representadas por lo prototipos en *physic.h,* ya analizado para el caso del lado servidor; se verá que estos dos niveles son los mismos en ambos lados.

4.5. Generando los procesos

Si bien nuestro primer proyecto está inconcluso, ya que no hemos decidido una forma de comunicación a nivel de transporte, sería útil, a esta altura, ver que estamos en condiciones de generar dos procesos independientes.

Para ello, sería necesario escribir, aunque más no sea, un nivel de comunicaciones que alojaremos en un archivo *physic.c* al solo efecto de poder completar el proyecto y ver que podemos generar dos ejecutables[46].

En el apéndice se ha agregado el módulo llamado *physic.c* que como se ve, contiene dos funciones *vacías* para *receive_packet* y para *send_packet*.

Detallaremos, ahora, que archivos conforman cada uno de los ejecutables:

[42] Por SERVER OFFset
[43] De acuerdo a lo establecido anteriormente en este trabajo.
[44] Como la mayoría de los analizadores de comunicaciones por lo menos trabajan en hexadecimal, es conveniente darle el valor 0x80 o 0x8000 ¿Por qué?
[45] Prácticamente, son *one-liners*.
[46] Aunque, obviamente, no podrán ser ejecutados.

Módulo fuente	Módulo de inclusión
client.c	server.h
clstub.c	clserv.h
physic.c	physic.h
	mytypes.h
	mydefs.h

Tabla 2: Módulos que conforman el ejecutable *client*

Módulo fuente	Módulo de inclusión
server.c	server.h
svstub.c	clserv.h
physic.c	physic.h
	mytypes.h
	mydefs.h

Tabla 3: Módulos que conforman el ejecutable *server*

Obsérvese que el módulo *physic.c* es el mismo para ambos procesos, indicando la simetría del sistema de comunicaciones debajo de la capa de *Middleware*.

Le invitamos, por último, que realice la compilación completa de ambos hasta obtener los ejecutables *client* y *server*, de manera de poder convencerse que sus fuentes son correctos (por lo menos desde un punto de vista sintáctico) y que se pueden generar los dos ejecutables.

Obviamente, no trate de ejecutarlos: sería inútil; el código de la parte inferior de comunicaciones aún está incompleto.

4.6. Una simulación.

Teniendo en cuenta que, de una arquitectura de comunicaciones, las partes superiores que se desarrollan[47], si bien tienen una directa vinculación física con las capas de comunicaciones inferiores,tiene por sí solo una entidad que trasciende a las mismas, sería interesante establecer los marcos de prueba en forma independiente a las capas más bajas.

[47] Como en nuestro caso la de *Middleware*.

Ello posibilita la verificación de las capas superiores en un marco independiente de las capas más bajas, pudiendo por un lado trabajar en forma independiente a las capas más bajas y, por el otro lado, no requerir pruebas utilizando costosas instalaciones de comunicaciones y tener que determinar qué partes de las capas superiores merecen cambios en función de mediciones realizadas con costosos y no siempre disponibles equipamientos de verificación de tramas inferiores.

4.6.1. Simulación de las capas inferiores

Intentaremos, ahora, realizar una simulación de manera de poder evaluar, sin necesidad de recurrir a un sistema de comunicaciones *real* la prueba de nuestros conceptos.

Para ello, deberemos establecer una simulación a nivel de las capas inferiores de comunicaciones (aquello que estaba hasta ahora comprendido en el archivo *physic.c*).

Adoptaremos una visión muy simplista para esta simulación, la cual supone pensar que existe un *cable* virtual que une ambos procesos[48].

En el apéndice, se muestra la segunda versión del módulo correspondiente a los prototipos de *physic.h* que se ha llamado (a fin de darse cuenta que es una simulación específica) *cable.c* donde se ha agregado el código y la estructura de datos para esta simulación.

Respecto de la estructura de datos, la misma está formada por un arreglo llamado **cable** que almacena los datos que se transmiten o que se reciben en esta capa y en el respectivo lado y un entero, **qty**, que almacena la cantidad de datos recibidos.

Las códigos de las funciones *receive_packet* y *transmit_packet* son directos de entender en relación a la estructura de datos especificada.

4.6.2. Simulación de los dos procesos.

[48] Nuevamente, se ha supuesto que al ser la comunicación *sincrónica*, el único cable existente se utiliza alternativamente para ambos sentidos de comunicación; de hecho, en un caso general, deberíamos colocar tantos cables virtuales como clientes existiesen.

Esta parte de la simulación permite trabajar con un solo proceso, involucrando los dos lados del Cliente-Servidor.

Obviamente, no es una forma general de realizarlo pero tiene varias ventajas: una de ellas es que puede ser realizado en el marco de un sistema operativo monousuario con MS-DOS, sin necesidad de crear dos procesos y utilizar alguna forma de comunicación entre ellos.

Por otro lado, muestra una forma de proceder para colocar en forma *lo más transparente posible* información de *debugging* en módulos, de forma que sea fácil su conmutación sin tener que reeditar dichos módulos.

Esta forma de simulación puede realizarse en este caso ya que, si se observa en todo lo que se ha planteado hasta el momento, se verá que cuando el servidor está bloqueado en un *receive_packet* el cliente se encuentra en ejecución y que, cuando el cliente envía una trama al servidor, lo cual desbloquea el servidor, el cliente se bloquea en un *receive_packet* hasta que el servidor lo desbloquea con la respuesta.

Por supuesto, desde la posición *se puede hacer* hasta *realmente hacerlo*, deberemos recorrer un camino que está *contaminado* de detalles que son propios de la implementación particular.

La idea detrás de esta simulación, está planteada en la figura siguiente:

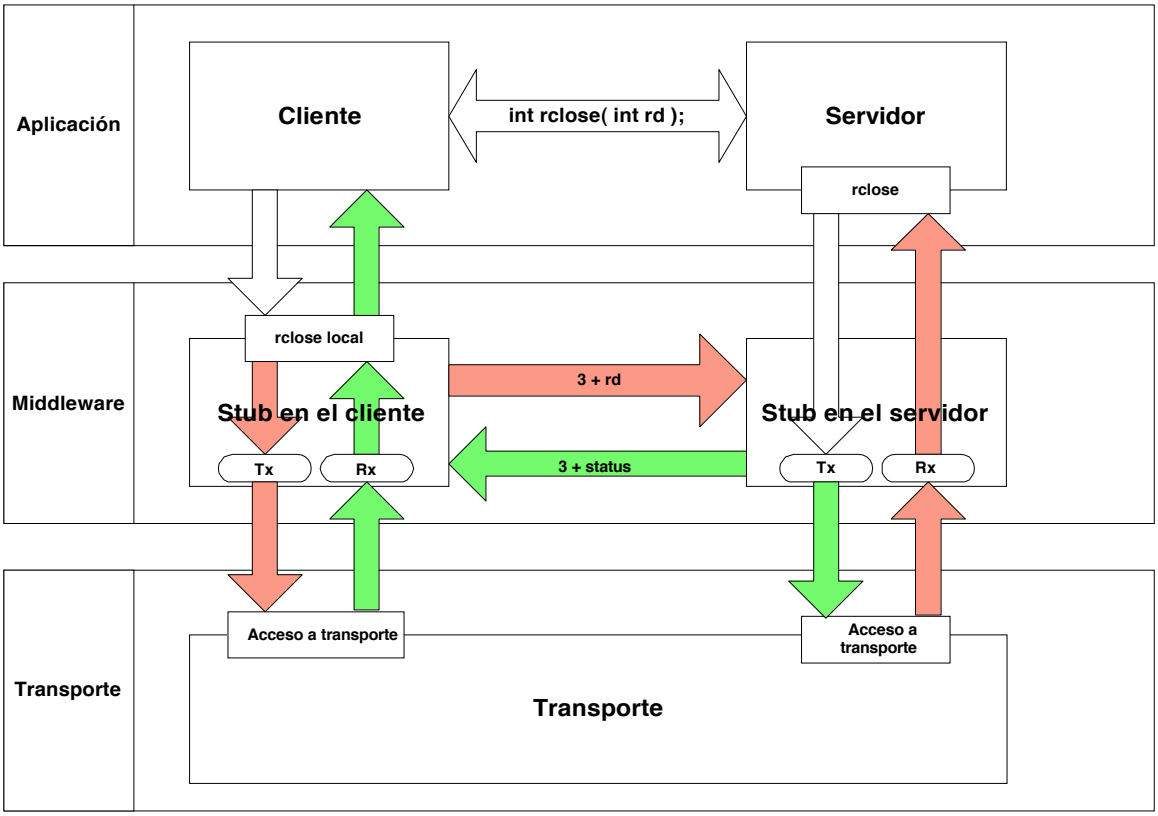

Figura 7 – Arquitectura en un solo proceso

Si se compara con una figura anterior, se verá que se ha cambiado la capa de transporte, de manera que sea *exactamente la misma* para ambos lados, por lo cual se habrá construído un solo proceso para la simulación cliente-servidor.

Conceptualmente, debería funcionar de la siguiente manera: cuando el cliente hace un requerimiento a través del *stub*, éste realiza el *data marshalling*, llama a *send_packet* para enviar la trama hacia el lado del servidor y luego realiza una llamado bloqueante a *receive_packet*.

La llamada a *send_packet*, coloca en el arreglo **cable** de *cable.c* la trama a enviar y la llamada a *receive_packet* (he aquí el cambio) llama a la entrada *do_server* en el *stub* del servidor de manera que el servidor recupere dicha trama la procese y la reenvíe hacia el cliente utilizando la función *send_packet*, hecho lo cual termina la llamada original del cliente a *receive_packet*.

Se ve que esto simula perfectamente el hecho que el cliente esté bloqueado esperando la terminación del servicio del servidor.

Sin embargo, las llamadas a *receive_packet* del lado cliente y del lado servidor no pueden ser las mismas pues sino generaría una recursión infinita.

En el apéndice, se ha colocado el archivo *cable.c* modificado para esta simulación, donde se ve una nueva función que se llama *client_receive_packet*.

No es todo lo que hay que realizar para esta simulación; existen otros cambios que deben ser efectuados en los *stubs*.

Sin embargo, aquí los cambios deben ser efectuados con más cuidado que en el caso de la capa de transporte.

En efecto, mientras que se sabe que el módulo *cable.c* está realizado especialmente para la simulación y, posteriormente, será *totalmente* reemplazado por el que corresponda a la capa de transporte adecuada, los *stubs* deberán permanecer incólumes luego de la simulación.

No tendría sentido cambiar el código de los *stubs* para realizar la simulación, para después de terminada, sacar el código de simulación con un editor de texto, pues cambiaríamos entonces aquello que hemos simulado y nunca estaríamos seguros que el código resultante corresponde al simulado.

Por otro lado, es buena política de desarrollo dejar el código de simulación o prueba para el caso que se quiera, en el futuro, volverse a realizar la prueba o simulación.

Recurriremos, por lo tanto, a la capacidad de *compilación condicional* que nos ofrece el lenguaje C a través de su preprocesador.

Para dilucidar si nos encontramos en simulación, haremos depender el código de una variable del preprocesador denominada **DEBUG_DOS**; si esta variable está definida, el código de los módulos de los *stubs* que se compilará incluirá

aquellos cambios que corresponden a la simulación y si no está definida, el código compilado coincidirá con el desarrollado hasta el momento.

Lo que se debe mantener incólume, de acuerdo a nuestra propuesta original, son los módulos *client.c* y *server.c*.

Debemos observar, además, que si vinculamos los cinco módulos nos encontraremos con problemas adicionales que provienen, fundamentalmente, del hecho que fueron pensados para generar dos procesos distintos y no estar compilados y vinculados para formar un único proceso.

En efecto, en el módulo *client.c* existe una función *main* pero también en el módulo *svstub.c* existe una función *main*.

Por otro lado, las funciones que implemente el servidor (los servicios del servidor) son funciones públicas tanto a nivel de *server.c* como de *clstub.c*.

Vamos a analizar por parte los cambios:

En el módulo *clstub.c* se han realizado pocos cambios (de hecho uno solo) y que es una sustitución de texto realizada por el preprocesador[49]: la llamada a *receive_packet* debe ser cambiada por una llamada a *client_receive_packet* siempre y cuando esté definida la variable del preprocesador **DEBUG_DOS**.

Veamos ahora los cambios en *svstub.c*: en este caso son varios: si vamos al final del módulo, veremos que toda la función *main* no es compilada en el caso que exista definida la variable **DEBUG_DOS**; esto evita una doble definición de *main*, quedando como única válida la que está en *client.c*.

Por otro lado, la función *do_server* (que era estática del módulo) debe ser pública para la simulación, ya que debe ser llamada desde el único módulo *cable.c*.

Este *truco* se logra anteponiendo el prefijo STATIC a la función, prefijo que en realidad es un macro definido en el

[49] Véase el módulo *clstub.c* modificado en el apéndice

archivo de inclusión *debugdos.h* y que está incluído en este módulo.

Si se observa el contenido, se verá que, en el caso de estar definida **DEBUG_DOS**, STATIC se reemplaza por el string vacío, permitiendo que entonces la función *do_server* sea pública; si **DEBUG_DOS** no está definida, entonces STATIC será reemplazada por static, permaneciendo el código original.

SI bien hemos intentado no tocar *server.c* lo hemos debido hacer pero no para hacer un cambio representativo ni de concepción; en efecto, en *server.c* se encuentran las cuatro funciones de servicio que necesariamente deben ser públicas; por lo dicho anteriormente, no podría quedar de esta forma pues sino habría doble definición con las que se encuentran en los *stubs*.

Obviamente, las funciones de *server.c* se deben transformar en privadas del módulo para no tener colisión con las de *clstub.c*, para lo cual hemos usado un *truco* parecido al utilizado en *do_server* pero contrario; se le han impuesto a las cuatro funciones el prefijo PUBLIC que en *debugdos.h* figura definida como el *string* vacío, en el caso que no esté definida **DEBUG_DOS** y como *static* si **DEBUG_DOS** está definida.

Sin embargo, ¿para qué sirven funciones privadas de un módulo si no son llamadas de dicho módulo?

La forma de solucionar este problema es poco ortodoxa pero eficiente: incluir el archivo *server.c* en el archivo *svstub.c* en lugar de incluir *server.h*; si se observa nuevamente *svstub.c* se verá que cerca del encabezamiento del archivo, existe un bloque de compilación condicional[50] en función de **DEBUG_DOS** que permite decidir si se incluye *server.h* o *server.c*.

A continuación, se muestra que archivos componen el proyecto de compilación:

[50] #ifdef - #else - #endif

Módulos	Archivos de inclusión
client.c	server.h
clstub.c	server.h
cable.c	physic.h
svstub.c	server.c
	clserv.h
	server.h
	mydefs.h
	mytypes.h
	debugdos.h

Es muy importante, aesta altura, que comprenda perfectamente cómo funciona esta simulación y que exprima al máximo sus observaciones: para ello, compílelo y genere un ejecutable (se sugiere el nombre *clserv*) y, si posee un *debugger*, trate de seguir paso a paso la secuencia de eventos para darse cuenta *operativamente* como funciona un sistema cliente servidor.

4.7. Una herramienta valiosa de visualización

Es común que los programadores de estos sistemas usen los *debuggers* que usualmente están integrados con compiladores para el seguimiento y la visualización de los programas.

Sin embargo y a criterio de este autor, no son muy útiles por más sofisticados que parezcan: generalmente, estos *debuggers* permiten visualizar más a nivel del lenguaje que a nivel de la aplicación.

En efecto, la observación que se desea realizar sobre un programa bajo inspección debe estar directamente vinculada al problema y expresada en términos del problema, más que del lenguaje que aporta la solución.

Por lo tanto, es mejor (aunque al principio no parezca más cómodo) establecer, conjuntamente con el programa, el desarrollo de los métodos de prueba y visualización y, en lo posible, dejarlos inmersos en el código para despertarlos en el momento que sean necesarios.

Para ello, se ha realizado un módulo denominado *analyzer.c* que simula la salida de un analizador de comunicaciones y que permite visualizar las tramas entre cliente y servidor; su código se incluye en el apéndice y la parte visible del mismo (es decir la función pública mediante la cual se lo invoca) se denomina *analyze*.

Esta función recibe un puntero al área de memoria que se pretende visualizar, la cantidad de bytes que se pretenden observar y un flag que indica el sentido de la comunicación (cliente a servidor o servidor a cliente).

No se hará una explicación de su forma de funcionamiento, ya que la misma es directa pero sí se observa que la salida sobre consola no es directa y que se realiza por las funciones de otro módulo de menor nivel que se denomina *terminal.c*.

En este módulo se encuentran funciones que son dependientes de la implementación de consola del sistema operativo y, por ende, este módulo está sujeto a cambio si se desea migrar de sistema operativo o si se desea cambiar la forma de presentar la información pero no el contenido de la misma.

La forma de insertar este analizador se encuentra en el archivo modificado *cable.c*, donde se ve que se ha colocado una llamada a *analyze* para mostrar lo que se encuentra en al arreglo **cable** tanto en *receive_packet* como en *client_receive_packet*.

5. Conclusiones

Se ha pretendido realizar un sencillo ejemplo de la estructura cliente servidor mediante pasos sucesivos de complejidad creciente.

Como se ha dicho ya en el texto, la idea más que presentar un proyecto completo y profesional, es una *aventura* de aprendizaje, donde, de todas maneras, se ha pretendido mantener reglas de estilo conformes al plan original.

Sin embargo, lo realizado aquí puede usarse como esqueleto de un proyecto real[51].

[51] De hecho, el autor en una emergencia ha modificado los programas utilizados en clases de demostración para formar parte de un sistema más grande.

Es importante que el alumno siga detalladamente los ejemplos propuestos y en el mismo orden creciente para tener una idea acabada del tema.

6. Ejercicios

Es importante recalcar que estos ejercicios sean atacados luego que se hayan recreado y comprendido los códigos que se han desarrollado.

6.1.

Utilizando cualquiera de las primitivas de comunicación entre procesos de UNIX[52], genere y pruebe el código resultante de usar dicha primitiva para realizar el nivel especificado en *physic.h*.

Si el módulo necesita inicialización, la misma debe ser realizada en forma transparente a los restantes módulos.

De manera de visualizar los paquetes de comunicaciones, coloque las llamadas a *analyze* en los lugares adecuados.

6.2.

Cambie el funcionamiento del lado servidor para que el mismo pueda servir hasta 128 clientes.

Establezca que cambios debe hacer en cada módulo, manteniendo la transparencia adecuada con respecto a los otros módulos.

6.3. .

Con el funcionamiento de los dos problemas anteriores, pruebe a cancelar un cliente cuando se está en el medio del funcionamiento[53] y vuelva a arrancar el cliente; observe que ocurre.

[52] UNIX es el nombre genérico de un tipo de Sistema Operativo; puede utilizarse cualquiera de su tipo (como Linux) salvo que se indique específicamente uno en particular.

[53] Esto es relativamente fácil de realizar ya que por el funcionamiento de *analyze* el programa se detiene cuando se muestra cada trama de comunicaciones.

Ahora, en el medio del funcionamiento de los clientes, haga caer el servidor y rearránquelo antes de continuar el funcionamiento de los clientes; observe que ocurre.

En función de lo observado ¿qué debería cambiar en el funcionamiento del cliente-servidor para evitar estos problemas?

Hágalo y verifique que los problemas observados han desaparecido.

6.4.

Realice un módulo denominado *noise.c* que pueda simular el ruido en un canal de comunicaciones

Dicho módulo debe poder vincularse directamente con el resto de los módulos o colocarse como un proceso que se comunica con el módulo implementado en el punto anterior.

Este módulo *noise* debe poder inicializarse con la inversa del *Bit Error Rate*, es decir, si se desea un ruido que modifique 1 bit de cada 10000 bits, debe inicializarse con el valor 10000.

Vincúlelo al resto del proyecto y pruébelo. Cambie el *Bit Error Rate* desde 10^{-10} hasta 10^{-3} y saque conclusiones de su comportamiento.

6.5. .

Para resolver el problema planteado en el punto anterior, se decide realizar una capa de *Data Link* cuyo protocolo esté formado de la siguiente manera:

- Encabezamiento: **STX**
- Fin: **ETX**
- Verificador: checksum realizado por el exclusive-or de todos los bytes incluyendo ETX pero excluyendo STX
- Campo de datos: los códigos que se confundan con los de control (**STX**, **ETX**, **DLE**) debe anteponérsele **DLE**.

En resumen, el formato es:

STX ----------Data--------- **ETX CHK**

En caso que la capa receptora de *Data Link* no verifique el checksum recibido, sencillamente no pasará a la capa de *stub* correspondiente la trama recibida.
Razone en qué capa o capas debe poner temporizadores para realizar una nueva solicitud de transmisión.

Implemente la capa de *Data Link*, únala al resto del proyecto conjuntamente con el módulo *noise* y verifique ahora para que *Bit Error Rate* la capa de *Data Link* deja de resolver el problema.

6.6. .

Es interesante, si dispone de dos computadoras, reemplazar la capa de comunicaciones (realizada hasta ahora con IPC) por una de manejo de comunicaciones en RS232.

Cambie el código de la capa de comunicaciones para manejar el puerto de comunicacionesserie, consiga un cable (o fabríquelo) que tenga cruzados Rx y Tx[54] y coloque los dos procesos, el de cliente y elde servidor, en distintas computadoras; retire el módulo *noise* de su proyecto.

Pruebe los programas y, una vez que funcione, verifique si extrayendo y colocando repetida y rápidamente uno de los conectores del cable, las comunicaciones se restituyen adecuadamente para el funcionamiento del cliente-servidor.

6.7. .

Si dispone de tres computadoras y adaptadores RS485, puede realizar el siguiente e interesante ejercicio, si ha realizado el problema 6.2.

Implemente la sub-capa de acceso al medio para RS485 y vincúlelo al resto del proyecto; observe el comportamiento y saque conclusiones.

6.8. .

A partir del problema 6.2, cambie la capa de comunicaciones para usar ya sea UDP o TCP entre tres o más computadoras.

Realice los programas de cliente y servidor y pruébelos.

[54] Un cable serie construído para comunicación por LapLink se puede conseguir armado en casas de computación.

Apéndices de Códigos

----------client.c----------

```
/*
 *  client.c
 */

/*
 *  Test of server
 */

#include <stdio.h>
#include <stdlib.h>

#include "server.h"

#define XFER_BUFF  32

static
void
get( const char *to, const char *from )
{
    RD rd;
    FILE *f;
    int qty;
    char buffer[ XFER_BUFF ];

    printf( "transferring %s from remote to %s local\n", from, to );
    rd = ropen( from, "r" );
    printf( "get: open remote: %d\n", rd );
    f = fopen( to, "w" );
    if( f == NULL )
    {
            perror( "get" );
            exit( 1 );
    }
    while( ( qty = rread( rd, buffer, sizeof( buffer ) ) ) > 0 )
            fwrite( buffer, 1, qty, f );
    printf( "get: end of xfer %d\n", qty );
    printf( "closing remote %d\n", rclose( rd ) );
    fclose( f );
}

static
void
put( const char *to, const char *from )
{
    RD rd;
    FILE *f;
    int qty;
    char buffer[ XFER_BUFF ];

    printf( "transferring %s from local to %s in remote\n", from, to );
    rd = ropen( to, "w" );
    printf( "get: open remote: %d\n", rd );
    f = fopen( from, "r" );
    if( f == NULL )
    {
            perror( "get" );
            exit( 1 );
    }
    while( ( qty = fread( buffer, 1, sizeof( buffer ), f ) ) > 0 )
            rwrite( rd, buffer, qty );
    printf( "closing remote %d\n", rclose( rd ) );
    fclose( f );
}

void
main( void )
{
    get( "client1.c", "client.c" );
    put( "client2.c", "client1.c");
}
```

----------server.c----------

```
/*
 *  server.c
 */

#include <stdio.h>

#include "server.h"

#define MAX_FILES  20

static FILE *openfiles[ MAX_FILES ];

static
int
not_verify_rd( RD rd )
{
    if( rd < 0 || rd >= MAX_FILES )
            return -S_BAD_RD;
    if( openfiles[ rd ] == NULL )
            return -S_NOT_OPENED;
    return S_OK;
}

RD
ropen( const char *pathname, const char *mode )
{
    FILE **pf;

    for( pf = openfiles ; pf < openfiles + MAX_FILES ; ++pf )
            if( *pf == NULL )
            {
                    *pf = fopen( pathname, mode );
                    if( *pf == NULL )
                            return -S_BAD_OPEN;
                    return pf - openfiles;
            }
    return -S_NO_ROOM;
}

int
rread( RD rd, void *data, int qty )
{
    int status;

    if( status = not_verify_rd( rd ), status )
            return status;
    return fread( data, 1, qty, openfiles[ rd ] );
}

int
rwrite( RD rd, const void *data, int qty )
{
    int status;

    if( status = not_verify_rd( rd ), status )
            return status;
    return fwrite( data, 1, qty, openfiles[ rd ] );
}

int
rclose( RD rd )
{
    int status;

    if( status = not_verify_rd( rd ), status )
            return status;
    status = fclose( openfiles[ rd ] );
    openfiles[ rd ] = NULL;
    return status;
}
```

----------server.h (primera versión)----------

```
/*
 *  server.h
 */

typedef int RD;

enum
{
    S_OK, S_BAD_OPEN, S_NO_ROOM,
    S_BAD_RD, S_NOT_OPENED,
    S_NUM_ERRORS
};

RD ropen( const char *pathname, const char *mode );
int rread( RD rd, void *data, int qty );
int rwrite( RD rd, const void *data, int qty );
int rclose( RD rd );
```

----------clserv.h----------

```
/*
 *      clserv.h
 */

#include "mytypes.h"

typedef RINT_T OPC;

#define MAX_MODE       4
#define MAX_NAME    50
#define MAX_DATA       512

enum
{
        ROPEN, RREAD, RWRITE, RCLOSE, MAX_OPCODES, SERVER_OFF = 0
};



                /*      Client -> server       */

                typedef struct
                {
                        char mode[ MAX_MODE ];
                        char pathname[ MAX_NAME ];
                } CLSV_ROPEN;

                typedef struct
                {
                        RD rd;
                        RINT_T qty;
                } CLSV_RREAD;

                typedef struct
                {
                        RD rd;
                        RINT_T qty;
                        char data[ MAX_DATA ];
                } CLSV_RWRITE;

                typedef struct
                {
                        RD rd;
                } CLSV_RCLOSE;

                /*      Server -> client       */

                typedef struct
                {
                        RD rd;
```

```
                } SVCL_ROPEN;

                typedef struct
                {
                        RINT_T status;
                        char data[ MAX_DATA ];
                } SVCL_RREAD;

                typedef struct
                {
                        RINT_T status;
                } SVCL_RWRITE;

                typedef struct
                {
                        RINT_T status;
                } SVCL_RCLOSE;


        typedef union
        {
                CLSV_ROPEN clsv_ropen;
                CLSV_RREAD clsv_rread;
                CLSV_RWRITE clsv_rwrite;
                CLSV_RCLOSE clsv_rclose;
                SVCL_ROPEN svcl_ropen;
                SVCL_RREAD svcl_rread;
                SVCL_RWRITE svcl_rwrite;
                SVCL_RCLOSE svcl_rclose;
        } DATA;

typedef struct
{
        OPC     opc;
        DATA data;
} CLSVBUFF;
```

----------mytypes.h----------

```
#ifndef __MYTYPES__
#define __MYTYPES__

/*
 *      mytypes.h
 */

typedef short RINT_T;

#endif
```

----------server.h (segunda versión)----------

```
/*
 *      server.h
 */

#include "mytypes.h"


typedef RINT_T RD;

enum
{
        S_OK, S_BAD_OPEN, S_NO_ROOM,
        S_BAD_RD, S_NOT_OPENED,
        S_NUM_ERRORS
};

RD ropen( const char *pathname, const char *mode );
int rread( RD rd, void *data, int qty );
int rwrite( RD rd, const void *data, int qty );
int rclose( RD rd );
```

----------clstub.c----------

```
/*
 *      clstub.c
 */

#include <string.h>

#include "server.h"
#include "clserv.h"
#include "physic.h"
static CLSVBUFF clsvbuff;
static
int
send_rcv( CLSVBUFF *p, int opcode, int qty )
{
        int qtyrec;

        p->opc = opcode;
        send_packet( p, qty + sizeof( OPC ) );
        qtyrec = receive_packet( p, sizeof( *p ) );
        return qtyrec - sizeof( OPC );
}


RD
ropen( const char *pathname, const char *mode )
{
        CLSV_ROPEN *pc;
        SVCL_ROPEN *ps;

        pc = &clsvbuff.data.clsv_ropen;
        ps = &clsvbuff.data.svcl_ropen;
        strcpy( pc->pathname, pathname );
        strcpy( pc->mode, mode );
        send_rcv( &clsvbuff, ROPEN, sizeof( CLSV_ROPEN ) );

        return ps->rd;
}

int
rread( RD rd, void *data, int qty )
{
        CLSV_RREAD *pc;
        SVCL_RREAD *ps;

        int status;

        pc = &clsvbuff.data.clsv_rread;
        ps = &clsvbuff.data.svcl_rread;

        pc->rd = rd;
        pc->qty = qty;
        send_rcv( &clsvbuff, RREAD, sizeof( CLSV_RREAD ) );

        status = ps->status;
        if( status > 0 )
                memcpy( data, ps->data, status );
        return status;
}

int
rwrite( RD rd, const void *data, int qty )
{
        CLSV_RWRITE *pc;
        SVCL_RWRITE *ps;

        pc = &clsvbuff.data.clsv_rwrite;
        ps = &clsvbuff.data.svcl_rwrite;

        pc->rd = rd;
        pc->qty = qty;
        if( qty > 0 )
```

```
                memcpy( pc->data, data, qty );
        send_rcv( &clsvbuff, RWRITE, sizeof( CLSV_RWRITE ) );

        return ps->status;
}

int
rclose( RD rd )
{
        clsvbuff.data.clsv_rclose.rd = rd;
        send_rcv( &clsvbuff, RCLOSE, sizeof( CLSV_RCLOSE ) );
        return clsvbuff.data.svcl_rclose.status;
}
```

----------svstub.c----------

```
/*
 *      svstub.c
 */

/*
 *      System includes
 */

/*
 *      General includes
 */

#include "mydefs.h"

/*
 *      Project includes
 */

#include "server.h"
#include "clserv.h"
#include "physic.h"

/*
 *      Static variables
 */

static CLSVBUFF clsvbuff;

/*
 *      Static functions
 */
static
int
process_ropen( DATA *p, int qty )
{
        p->svcl_ropen.rd = ropen( p->clsv_ropen.pathname, p->clsv_ropen.mode );
        return sizeof( SVCL_ROPEN );
}

static
int
process_rread( DATA *p, int qty )
{
        p->svcl_rread.status =
                rread( p->clsv_rread.rd, p->svcl_rread.data, p->clsv_rread.qty );
        return sizeof( SVCL_RREAD );
}

static
int
process_rwrite( DATA *p, int qty )
{
        p->svcl_rwrite.status =
                rwrite( p->clsv_rwrite.rd, p->clsv_rwrite.data, p->clsv_rwrite.qty );
        return sizeof( SVCL_RWRITE );
```

```
}
static
int
process_rclose( DATA *p, int qty )
{
        p->svcl_rclose.status = rclose( p->clsv_rclose.rd );
        return sizeof( SVCL_RCLOSE );
}

static int (*proc[])( DATA *p, int qty ) =
{
        process_ropen, process_rread, process_rwrite, process_rclose
};



static
int
process_server( CLSVBUFF *p, int qty )
{
        int opcode;

        opcode = p->opc;
        qty = (*proc[opcode])( &p->data, qty - sizeof( OPC ) );
        p->opc = opcode + SERVER_OFF;
        return qty + sizeof( OPC );
}

static
void
do_server( void )
{
        int qty;

        qty = receive_packet( &clsvbuff, sizeof( clsvbuff ) );
        qty = process_server( &clsvbuff, qty );
        send_packet( &clsvbuff, qty );
}

/*
 *      Public functions
 */

void
main( void )
{
        forever
                do_server();
}
```

----------physic.h----------

```
/*
 *      physic.h
 */

int send_packet( const void *p, int qty );
int receive_packet( void *p, int lim );
```

----------mydefs.h----------

```
ifndef __MYDEFS__
#define __MYDEFS__

#define forever        for(;;)

#endif
```

----------physic.c (sin código útil)----------

```
/*
 *      physic.c
 */
```

```
int
send_packet( const void *p, int qty )
{
}

int
receive_packet( void *p, int lim )
{
}
```

----------mydefs.h (nueva versión)----------

```
#ifndef __MYDEFS__
#define __MYDEFS__

#define forever        for(;;)
#define K      1024

#endif
```

----------cable.c ----------

```
/*
 *      cable.c
 */

#include <string.h>

#include "mydefs.h"
#include "physic.h"

static unsigned char cable[ 2 * K ];
static int cable_qty;

int
send_packet( const void *p, int qty )
{
        memcpy( cable, p, qty );
        cable_qty = qty;
        return qty;
}

int
client_receive_packet( void *p, int lim )
{
        do_server();
        memcpy( p, cable, cable_qty );
        return cable_qty;
}

int
receive_packet( void *p, int lim )
{
        memcpy( p, cable, cable_qty );
        return cable_qty;
}
```

------- clstub.c (modificado para simulación)------

```
/*
 *      clstub.c
 */

#include <string.h>

#include "server.h"
#include "clserv.h"
#include "physic.h"

#ifdef DEBUG_DOS
#define receive_packet client_receive_packet
#endif


static CLSVBUFF clsvbuff;
```

```
static
int
send_rcv( CLSVBUFF *p, int opcode, int qty )
{
        int qtyrec;

        p->opc = opcode;
        send_packet( p, qty + sizeof( OPC ) );
        qtyrec = receive_packet( p, sizeof( *p ) );
        return qtyrec - sizeof( OPC );
}


RD
ropen( const char *pathname, const char *mode )
{
        CLSV_ROPEN *pc;
        SVCL_ROPEN *ps;

        pc = &clsvbuff.data.clsv_ropen;
        ps = &clsvbuff.data.svcl_ropen;
        strcpy( pc->pathname, pathname );
        strcpy( pc->mode, mode );
        send_rcv( &clsvbuff, ROPEN, sizeof( CLSV_ROPEN ) );

        return ps->rd;
}

int
rread( RD rd, void *data, int qty )
{
        CLSV_RREAD *pc;
        SVCL_RREAD *ps;

        int status;

        pc = &clsvbuff.data.clsv_rread;
        ps = &clsvbuff.data.svcl_rread;

        pc->rd = rd;
        pc->qty = qty;
        send_rcv( &clsvbuff, RREAD, sizeof( CLSV_RREAD ) );

        status = ps->status;
        if( status > 0 )
                memcpy( data, ps->data, status );
        return status;
}

int
rwrite( RD rd, const void *data, int qty )
{
        CLSV_RWRITE *pc;
        SVCL_RWRITE *ps;

        pc = &clsvbuff.data.clsv_rwrite;
        ps = &clsvbuff.data.svcl_rwrite;

        pc->rd = rd;
        pc->qty = qty;
        if( qty > 0 )
                memcpy( pc->data, data, qty );
        send_rcv( &clsvbuff, RWRITE, sizeof( CLSV_RWRITE ) );

        return ps->status;
}

int
rclose( RD rd )
{
        clsvbuff.data.clsv_rclose.rd = rd;
        send_rcv( &clsvbuff, RCLOSE, sizeof( CLSV_RCLOSE ) );
        return clsvbuff.data.svcl_rclose.status;
}
```

------- svstub.c (modificado para simulación)------

```
/*
 *      svstub.c
 */

/*
 *      System includes
 */

/*
 *      General includes
 */

#include "mydefs.h"
#include "debugdos.h"

/*
 *      Project includes
 */

#ifndef DEBUG_DOS
#include "server.h"
#else
#include "server.c"
#endif

#include "clserv.h"
#include "physic.h"

/*
 *      Static variables
 */

static CLSVBUFF clsvbuff;

/*
 *      Static functions
 */
static
int
process_ropen( DATA *p, int qty )
{
        p->svcl_ropen.rd = ropen( p->clsv_ropen.pathname, p->clsv_ropen.mode );
        return sizeof( SVCL_ROPEN );
}

static
int
process_rread( DATA *p, int qty )
{
        p->svcl_rread.status =
                rread( p->clsv_rread.rd, p->svcl_rread.data, p->clsv_rread.qty );
        return sizeof( SVCL_RREAD );
}

static
int
process_rwrite( DATA *p, int qty )
{
        p->svcl_rwrite.status =
                rwrite( p->clsv_rwrite.rd, p->clsv_rwrite.data, p->clsv_rwrite.qty );
        return sizeof( SVCL_RWRITE );
}

static
int
process_rclose( DATA *p, int qty )
{
        p->svcl_rclose.status = rclose( p->clsv_rclose.rd );
        return sizeof( SVCL_RCLOSE );
}

static int (*proc[])( DATA *p, int qty ) =
```

```
{
        process_ropen, process_rread, process_rwrite, process_rclose
};

static
int
process_server( CLSVBUFF *p, int qty )
{
        int opcode;

        opcode = p->opc;
        qty = (*proc[opcode])( &p->data, qty - sizeof( OPC ) );
        p->opc = opcode + SERVER_OFF;
        return qty + sizeof( OPC );
}

STATIC
void
do_server( void )
{
        int qty;

        qty = receive_packet( &clsvbuff, sizeof( clsvbuff ) );
        qty = process_server( &clsvbuff, qty );
        send_packet( &clsvbuff, qty );
}

/*
 *      Public functions
 */

#ifndef DEBUG_DOS

void
main( void )
{
        forever
                do_server();
}

#endif
```

---------- debugdos.h ----------

```
/*
 *      debugdos.h
 */

#ifdef DEBUG_DOS
#define STATIC
#define PUBLIC static
#else
#define STATIC static
#define PUBLIC
#endif
```

--------- server.c (modificado para simulación) ------

```
/*
 *      server.c
 */

#include <stdio.h>

#include "debugdos.h"

#include "server.h"

#define MAX_FILES      20

static FILE *openfiles[ MAX_FILES ];

static
int
not_verify_rd( RD rd )
```

```
{
        if( rd < 0 || rd >= MAX_FILES )
                return -S_BAD_RD;
        if( openfiles[ rd ] == NULL )
                return -S_NOT_OPENED;
        return S_OK;
}

PUBLIC
RD
ropen( const char *pathname, const char *mode )
{
        FILE **pf;

        for( pf = openfiles ; pf < openfiles + MAX_FILES ; ++pf )
                if( *pf == NULL )
                {
                        *pf = fopen( pathname, mode );
                        if( *pf == NULL )
                                return -S_BAD_OPEN;
                        return pf - openfiles;
                }
        return -S_NO_ROOM;
}

PUBLIC
int
rread( RD rd, void *data, int qty )
{
        int status;

        if( status = not_verify_rd( rd ), status )
                return status;
        return fread( data, 1, qty, openfiles[ rd ] );
}

PUBLIC
int
rwrite( RD rd, const void *data, int qty )
{
        int status;

        if( status = not_verify_rd( rd ), status )
                return status;
        return fwrite( data, 1, qty, openfiles[ rd ] );
}

PUBLIC
int
rclose( RD rd )
{
        int status;

        if( status = not_verify_rd( rd ), status )
                return status;
        status = fclose( openfiles[ rd ] );
        openfiles[ rd ] = NULL;
        return status;
}
```

---------- analyzer.h ----------

```
/*
 *      analyzer.h
 */

enum
{
        SVCL, CLSV
};

void analyze( const void *p, int qty, int cl2sv );
```

------------------ analyzer.c ------------------

```
/*
 *      analyzer.c
 */

#include <ctype.h>

#include "mydefs.h"
#include "analyzer.h"
#include "terminal.h"

#define BLOCK_DUMP     16

static int init;

static
void
test_init( void )
{
        if( !init )
        {
                init = 1;
                vfsend( "*************************\n" );
        }
}

static
void
dump_hex( const unsigned char *p, int qty )
{
        int i;
        const unsigned char *q;


        for( i = 0, q = p ; i < qty ; ++i )
                vfsend( "%02.2X ", *q++ );

        for(  ; i < BLOCK_DUMP ; ++i )
                vfsend( "   " );
}

static
void
dump_ascii( const unsigned char *p, int qty )
{
        for( ; qty-- ; ++p )
                vfsend( isprint( *p ) ? "%c" : "." , *p );
}

static
void
dump( const unsigned char *p, int qty )
{
        unsigned address;

        for( address = 0 ; qty > 0 ;
                qty -= BLOCK_DUMP, p += BLOCK_DUMP, address += BLOCK_DUMP )
        {
                vfsend( "%03.2X-%02d:", address, address );
                dump_hex( p, qty > BLOCK_DUMP ? BLOCK_DUMP: qty );
                vfsend( "   " );
                dump_ascii( p, qty > BLOCK_DUMP ? BLOCK_DUMP: qty );
                vfsend( "\n" );
        }
}

void
analyze( const void *p, int qty, int cl2sv )
{
        test_init();
        vfsend( cl2sv ? ">>>> Client request\n" : "<<<<< Server answer\n" );
        dump( p, qty );
        vfsend( "----------------------\n" );
        getkey();
}
```

------------------ terminal.h ------------------

```
/*
 *      terminal.h
 */

void vfsend( const char *fmt, ... );
int keyhit( void );
int getkey( void );
```

------------------ terminal.c------------------

```
/*
 *      terminal.c
 *              Written for MS-DOS
 *              This file is OS dependent
 */

#include <stdio.h>
#include <stdarg.h>
#include <conio.h>

#include "terminal.h"

static
void
send_char( int c )
{
        fputc( c, stderr );
}

static
void
send_string( const char *p )
{
        while( *p != '\0' )
                send_char( *p++ );
}

void
vfsend( const char *fmt, ... )
{
        va_list ap;
        char buffer[80];

        va_start( ap, fmt );
        vsprintf( buffer, fmt, ap );
        va_end( ap );
        send_string( buffer );
}

int
keyhit( void )
{
        return kbhit();
}

int
getkey( void )
{
        return getch();
}
```