Redes de Computadores
Práctica 4: Implementación de protocolos y algoritmos de secuenciación (trabajo práctico)

Juan Segarra Flor

1. Objetivos

El objetivo de esta práctica es entender el trabajo práctico y empezar su implementación. El trabajo práctico consiste en implementar un protocolo de forma básica y extenderlo con los algoritmos de secuenciación de datos vistos en clase. Para ello hay que realizar un programa cliente que se comunique con el servidor proporcionado mediante el protocolo RCFTP.

2. Trabajo a realizar

Debes implementar *tres versiones* de un cliente que envíe su entrada estándar al servidor, que escribirá lo que reciba en el fichero `f_recibido`. El envío de datos debe cumplir las especificaciones del protocolo RCFTP, es decir, el cliente debe enviar los datos con el formato especificado en el protocolo y debe comprobar que el servidor confirma correctamente su recepción. El cliente debe estar preparado para trabajar tanto con IPv4 como con IPv6.

Invocado sin parámetros, el cliente muestra una ayuda con el formato que deben seguir sus parámetros:

```
Uso: ./rcftpclient [-v] -a[alg] [-t[Ttrans]] [-T[timeout]] [-w[tam]] -d<dirección>
-p<puerto>
  -v            Muestra detalles en salida estándar
  -a[alg]       Algoritmo de secuenciación a utilizar:
     1          Algoritmo básico
     2          Algoritmo Stop&Wait
     3          Algoritmo de ventana deslizante Go-Back-n
  -w[tam]       Tamaño (en bytes) de la ventana de emisión (sólo usado con -a3)
(por defecto: 2048)
  -t[Ttrans]    Tiempo de transmisión a simular, en microsegundos (por defecto: 200000)
  -T[timeout]   Tiempo de expiración a simular, en microsegundos (por defecto: 1000000)
  -d<dirección> Dirección del servidor
  -p<puerto>    Servicio o número de puerto del servidor
```

Las tres versiones del cliente son el algoritmo básico (sección 5), el algoritmo Stop&Wait (sección 6) y el algoritmo Go-back-n (sección 7). Para centrar la implementación en la programación del protocolo, puedes descargar de Moodle una versión incompleta del cliente a implementar. El cliente incompleto proporcionado contiene varias funciones ya implementadas (lectura de parámetros, lectura de entrada estándar, visualización de dirección remota, visualización de estadísticas una vez finalizada la transmisión), así como el `Makefile` correspondiente para poder compilarlo. Además, probablemente puedas reutilizar parte del código que ya has realizado anteriormente en el cliente-servidor UDP. Concretamente, en el fichero comprimido del cliente incompleto encontrarás los siguientes ficheros:

- rcftp.h/.c: Cabeceras y funciones del protocolo RCFTP
- multialarm.h/.c: Cabeceras y funciones de gestión de temporizadores
- rcftpclient.h/.c: Cabeceras, programa principal y varias funciones del cliente RCFTP

- Makefile: Dependencias y comandos para compilar mediante `make` (o `gmake`)
- misfunciones.h: Definiciones y cabeceras de varias de las funciones a implementar
- misfunciones.c: Espacio para implementar las funciones que necesites

3. Protocolo RCFTP

El programa de contar vocales implementado anteriormente realiza un intercambio de datos muy simple, donde los datos no necesitan ninguna organización particular. En aplicaciones más complejas los datos transmitidos necesitan una organización más estructurada, especificada por un protocolo. Para implementar el programa cliente de esta práctica es imprescindible conocer el protocolo de comunicaciones que va a utilizar: RCFTP. Dicho protocolo ha de ser usado tanto por el cliente como por el servidor.

RCFTP es un protocolo que, usado sobre UDP, añade fiabilidad y secuenciación de datos. En nuestro caso, el cliente enviará mensajes de acuerdo a las especificaciones correspondientes y el servidor le irá confirmando la recepción correcta de los mensajes.

3.1. Formato del mensaje

El protocolo RCFTP especifica el formato de las peticiones del cliente y respuestas del servidor (ambas de tamaño fijo) como se detalla a continuación:

1 B	1 B	2 B	4 B	4 B	2 B	512 B
Versión	Flags	Sum	NúmSeq	Next	Len	Buffer

```
struct rcftp_msg {
    uint8_t  version; /**< Versión RCFTP_VERSION_1; otro valor es inválido */
    uint8_t  flags;   /**< Flags. Máscara de bits de los defines F_X */
    uint16_t sum;     /**< Checksum en network format calculado con xsum */
    uint32_t numseq;  /**< Número de secuencia, medido en bytes */
    uint32_t next;    /**< Siguiente numseq esperado, medido en bytes */
    uint16_t len;     /**< Longitud de datos válidos en el campo buffer */
    uint8_t  buffer[RCFTP_BUFLEN]; /**< Datos de tamaño fijo 512 B */
};
```

Cada campo almacena la siguiente información:

- Versión (1 byte): contiene el número 1, que identifica la versión del protocolo. Nos permitirá cambiar nuestro protocolo y seguir identificando paquetes de implementaciones más viejas.
- Flags (1 byte): contiene la máscara de bits de los defines `F_X`. Ver la sección 3.2
- Checksum (2 bytes): contiene la negación (inversión de bits) de la suma *complemento a uno* de la petición/respuesta. Esta operación ya está implementada en la función *xsum()* que tienes en el código proporcionado. Como el checksum siempre se calcula/verifica con respecto al mensaje en formato de red, no hay que aplicar ninguna función de reordenación de bytes sobre él.
- Número de secuencia (4 bytes en la ordenación de bytes de la red): contiene el número de secuencia del primer byte de datos que contiene el mensaje.
- NEXT o Siguiente número de secuencia esperado (4 bytes en la ordenación de bytes de la red): contiene el número de secuencia del siguiente byte de datos que esperamos recibir. Ten en cuenta que el cliente no espera recibir datos.
- Longitud (2 bytes en la ordenación de bytes de la red): contiene el número de bytes de datos válidos que hay en el campo *buffer*.
- Datos o buffer: contiene los datos y es siempre de longitud RCFTP_BUFLEN.

3.2. Flags

Los flags del protocolo RCFTP especifican el tipo de petición del cliente o estado del servidor como se detalla a continuación:

- `F_NOFLAGS`: Flag por defecto
- `F_BUSY`: Flag de servidor ocupado atendiendo a otro cliente
- `F_FIN`: Flag de intención/confirmación de finalizar transmisión
- `F_ABORT`: Flag de aviso de finalización forzosa

Los mensajes generados por el cliente siempre deben tener el valor `F_NOFLAGS`, excepto cuando se transmita el último mensaje de datos, en cuyo caso se usará el valor `F_FIN` en los flags. El cliente deberá terminar cuando reciba la confirmación de ese último mensaje (también marcado con el flag `F_FIN`) por parte del servidor.

4. Servidor RCFTPd

El código del servidor *rcftpd* (RCFTP dæmon) está disponible en Moodle en un fichero comprimido, con los ficheros fuente estructurados igual que en el cliente. Se usa de la forma siguiente:

```
Uso: ./rcftpd -p<puerto> [-v] [-a[alg]] [-e[frec]] [-t[Ttrans]] [-r[Tprop]]
  -p<puerto>    Especifica el servicio o número de puerto
  -v            Muestra detalles en salida estándar
  -a[alg]       Ajusta el comportamiento al algoritmo del cliente (por defecto: 0):
      0:        Sin mensajes incorrectos
      1:        Fuerza mensajes incorrectos hasta su corrección
      2:        Fuerza mensajes incorrectos/pérdidas/duplicados hasta su corrección
      3:        Fuerza mensajes incorrectos/pérdidas/duplicados
  -e[frec]      Fuerza en media un mensaje incorrecto de cada [frec] (por defecto: 5)
  -t[Ttrans]    Tiempo de transmisión a simular, en microsegundos (por defecto: 200000)
  -r[Tprop]     Tiempo de propagación a simular, en microsegundos (por defecto: 250000)
Nota: Los algoritmos 1-3 generan errores aleatoriamente. Además, los algoritmos 1 y 2
mantienen el error generado hasta que el cliente responda correctamente.
```

Ten en cuenta que gran parte de la implementación del cliente ya está hecha en el servidor, con lo que es muy conveniente estudiar el servidor antes de programar el cliente.

Abre el fichero `rcftpd.h` y observa que ahí figuran todas las funciones, con una breve descripción de su funcionamiento. Si lo consideras oportuno, puedes copiarlas en tu cliente y adaptarlas como necesites.

Abre el fichero `rcftpd.c` y localiza el código de la función «process_requests». Dicha función implementa el funcionamiento general del servidor, es decir, recibe mensajes, los procesa, construye mensajes de respuesta, planifica su envío y finalmente los envía. Hay varios puntos especialmente interesantes en el código:

- Localiza el comentario «construir el mensaje válido» y observa que un mensaje se construye simplemente rellenando la estructura *rcftp_msg*, que se enviará posteriormente.
- Observa que el parámetro de entrada en la función que calcula checksum (*xsum*) es la propia estructura *rcftp_msg*, es decir, el checksum se calcula sobre el contenido de esa estructura. Eso implica que la estructura debe rellenarse *antes* de calcular su checksum.
- Ten en cuenta también que el campo *sum* forma parte de la propia estructura, y que la función *xsum* tendrá en cuenta su valor. Observa el código para ver cómo se consigue que el contenido de ese campo no afecte al resultado.

Como habrás observado, las funciones asociadas al protocolo RCFTP han sido separadas explícitamente de las del servidor RCFTPd para poder usarlas directamente desde el cliente. Abre el fichero `rcftp.h`. Observa que la declaración del *struct rcftp_msg* es ligeramente distinta de la del enunciado de

la práctica. A través de una directiva de preprocesador, el código determina si está siendo compilado por GCC o no, y dependiendo de esa condición se decide qué línea de código hay que usar. En la línea de código específica para GCC, el código tiene todos los campos del struct con atributo «packed» para especificar al compilador GCC que los ponga todos contiguos en memoria. Dicho atributo no pertenece al lenguaje C, sino al «dialecto de C» que entiende GCC, con lo que el resto de los compiladores generarían un error en ese punto. Aún sin poner ese atributo, el struct está organizado de forma que un compilador no se vea tentado a poner huecos entre campos. Aún así, podría darse el caso de que, si el compilador intentara optimizar mucho, pusiera cada campo en una dirección de memoria múltiplo del tamaño en bytes de sus registros (por ejemplo alineando todo a 64 bits). Eso podría ser más rápido en ejecución, porque el procesador podría cargar/guardar cada campo en memoria sin tener que alinear, pero sería desastroso para el programa, puesto que estaríamos enviando estructuras que no coincidirían con las especificadas en el protocolo. Es importante saber que los compiladores toman muchas decisiones que van más allá del lenguaje de programación. Al no estar presentes en el lenguaje, si un programador quiere especificar cómo tomar alguna de estas decisiones, debe recurrir a atributos, opciones de compilación, o directivas específicas del compilador de que se trate.

Observa las cabeceras y comentarios del fichero `rcftp.h`.

1. ¿Qué funciones tienes ya implementadas en el fichero `rcftp.c`?
2. ¿Hay que aplicar *htons* al usar *xsum*?

El código del servidor incluye un fichero `Makefile`, de forma que para compilar, basta con invocar el comando `make` (`gmake` en *Hendrix*). Este comando contiene las instrucciones para compilar y enlazar cada uno de los ficheros fuente con las bibliotecas necesarias, tanto en GNU/Linux-x86_64 (L1.02) como en SunOS-sparc (Hendrix). Visualiza el fichero `Makefile` para ver su funcionamiento. Observa que puede contener otros comandos además de los de compilación. Por ejemplo, ejecutando `make clean` se borrarán todos objetos y ejecutables generados por compilaciones anteriores.

3. ¿Cuáles son los ficheros necesarios para el objetivo *rcftpd*?
4. ¿Cuál es el comando que lanza ese objetivo?
5. ¿Qué hace el objetivo *rcftpd.tar.gz*?

4.1. Traza del servidor

A continuación puedes ver un ejemplo de lo que muestra por pantalla el servidor, con la opción `-v`, al comunicarse con un cliente que funciona. Tras implementar tu cliente, el servidor debería mostrar una salida similar.

```
Fichero "f_recibido" abierto para escritura
rcftpd $Revision$
Servidor RCFTP en puerto 23456
Comunicación con el puerto 55496 del equipo 127.0.0.1 usando IPv4

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 0
      Len: 512
      Checksum (net): 0x8a43 (ok)
Planificando envío 0 (todo correcto)

Realizando envio 0 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 512
```

```
      Len: 0
      Checksum (net): 0xc1cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 512
      Next: 0
      Len: 512
      Checksum (net): 0x5589 (ok)
Planificando envío 1 (todo correcto)

Realizando envio 1 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 1024
      Len: 0
      Checksum (net): 0xbfcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1024
      Next: 0
      Len: 512
      Checksum (net): 0x214 (ok)
Planificando envío 2 (todo correcto)

Realizando envio 2 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 1536
      Len: 0
      Checksum (net): 0xbdcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1536
      Next: 0
      Len: 512
      Checksum (net): 0x3745 (ok)
Planificando envío 3 (checksum con bytes desordenados y recepción incorrecta)

Realizando envio 3 (checksum con bytes desordenados y recepción incorrecta)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 1536
      Len: 0
      Checksum (net): 0xcdbb (error)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1536
      Next: 0
      Len: 512
      Checksum (net): 0x3745 (ok)
Planificando envío 4 (todo correcto)

Realizando envio 4 (todo correcto)
Mensaje RCFTP enviado:
```

```
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2048
      Len: 0
      Checksum (net): 0xbbcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2048
      Next: 0
      Len: 512
      Checksum (net): 0xa39c (ok)
Planificando envío 5 (checksum incorrecto y recepción incorrecta)

Realizando envio 5 (checksum incorrecto y recepción incorrecta)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2048
      Len: 0
      Checksum (net): 0xbacd (error)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2048
      Next: 0
      Len: 512
      Checksum (net): 0xa39c (ok)
Planificando envío 6 (todo correcto)

Realizando envio 6 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2560
      Len: 0
      Checksum (net): 0xb9cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2560
      Next: 0
      Len: 512
      Checksum (net): 0x5aa3 (ok)
Planificando envío 7 (todo correcto)

Realizando envio 7 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 3072
      Len: 0
      Checksum (net): 0xb7cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 2
      Núm. secuencia: 3072
      Next: 0
      Len: 306
      Checksum (net): 0xe60b (ok)
```

```
Planificando envío 8 (todo correcto)

Realizando envio 8 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 2
      Núm. secuencia: 0
      Next: 3378
      Len: 0
      Checksum (net): 0xb699 (ok)
Flag F_FIN recibido y confirmado
Recepción finalizada. Compara los ficheros para verificar los datos recibidos.
```

Como puedes observar, el servidor muestra todos los mensajes recibidos y enviados, indicando además si son correctos o no.

Con el objetivo de simular el retardo de red, el servidor no realiza los envíos inmediatamente, sino que espera cierto tiempo antes de enviar (dependiendo de los tiempos de transmisión y propagación introducidos como parámetros). Es decir, para cada mensaje recibido, el servidor *planifica* un envío, *identificado con un número*. Después de los retardos correspondientes, ese envío planificado será realmente enviado (y recibido por el cliente inmediatamente por estar muy cercanos cliente y servidor).

Como ayuda para la depuración, la salida está coloreada utilizando códigos ANSI. En caso de que vuelques la salida en un fichero para analizarlo posteriormente, para que los códigos de color se interpreten correctamente puedes volcar ese fichero por pantalla usando `cat` o puedes usar el comando `less` con la opción `-R`.

5. Algoritmo básico

Se utilizará el parámetro `-a1` tanto en el cliente como en el servidor. El programa cliente debe asumir que todos los mensajes enviados tendrán respuesta. El cliente deberá verificar que la respuesta sea válida y sea la esperada (la confirmación de recepción correcta de lo enviado por el cliente). Si es así, el cliente deberá enviar un nuevo mensaje con el siguiente bloque de datos. En caso contrario, el cliente deberá reenviar el mensaje anterior.

Puedes ver el pseudocódigo de la descripción anterior en el algoritmo 1. ¡Es esencial que entiendas el pseudocódigo antes de empezar a programarlo!

5.1. Ayuda a la implementación del algoritmo básico

Como ayuda a la implementación, ten en cuenta lo siguiente:

- El código proporcionado como plantilla para el cliente está organizado en archivos de forma similar al servidor. En principio sólo necesitas editar los archivos `misfunciones.c` y `misfunciones.h`, aunque puedes editar también el resto del código
- Ya tienes implementada la lectura de parámetros de la línea de comandos, la lectura de datos desde la entrada estándar y una función que muestra información sobre la comunicación tras finalizarla correctamente
- El establecimiento del socket UDP (*initsocket()*) y las funciones de manejo de estructuras de direcciones (*obtener_struct_direccion()*, *printsockaddr()*) son las que ya has implementado en prácticas anteriores; copia/pega/ajusta lo necesario
- El código proporcionado compila sin errores (aunque sí con *warnings*, por estar incompleto). Una forma de trabajar es compilar después de cada cambio introducido y verificar que no se ha introducido nada que genere errores de compilación
- No uses funciones de *strings* para procesar los datos a transmitir, porque esos datos no tienen porqué ser cadenas de caracteres

Algorithm 1 Algoritmo básico RCFTPClient

```
ultimoMensaje ← false
ultimoMensajeConfirmado ← false
datos ← leerDeEntradaEstandar(RCFTP_BUFLEN)
if finDeFicheroAlcanzado then
  ultimoMensaje ← true
end if
mensaje ← construirMensajeRCFTP(datos)
while ultimoMensajeConfirmado = false do
  enviar(mensaje)
  recibir(respuesta)
  if esMensajeValido(respuesta) and esLaRespuestaEsperada(respuesta) then
    if ultimoMensaje then
      ultimoMensajeConfirmado ← true
    else
      datos ← leerDeEntradaEstandar(RCFTP_BUFLEN)
      if finDeFicheroAlcanzado then
        ultimoMensaje ← true
      end if
      mensaje ← construirMensajeRCFTP(datos)
    end if
  end if
end while
/*** La función construirMensajeRCFTP() debe rellenar cada uno de los campos del mensaje (struct
rcftp_msg) a enviar ***/
/*** La función esMensajeValido() debe comprobar la versión y el checksum del mensaje ***/
/*** La función esLaRespuestaEsperada() debe comprobar que respuesta.next sea mensa-
je.numseq+mensaje.len, que no haya flags de «ocupado/abortar» en respuesta, y que si mensaje.flags
tiene marcado «fin» tambin lo tenga respuesta.flags ***/
```

- En la compilación verás que se compilan también los fichero `multialarm.c` y `vemision.c`. De momento puedes ignorarlo, ya que en el algoritmo básico no se utilizan
- Empieza la implementación del cliente asumiendo que no habrá errores en la comunicación (puedes lanzar el servidor con `-a0` para ello), centrándote así en aspectos de la comunicación
- Cuando el cliente funcione correctamente asumiendo una comunicación sin errores, pasa a probarlo con errores (`-a1` en el servidor) y verifica caso por caso todos los errores que puedan presentarse. ¡No asumas que los campos del mensaje recibido son correctos! ¡Comprueba que contienen lo esperado!

5.2. Ayuda para probar el algoritmo básico

Para probar el funcionamiento del algoritmo básico, ten en cuenta lo siguiente:

- Para enviar un fichero, utiliza una tubería en la entrada estándar: `cat fichero | ./rcftpclient -v -a1 -d<servidor> -p<puerto>`
- Se recomienda transmitir un fichero con datos aleatorios. Puedes generar un fichero de 20 kB con contenido aleatorio mediante el comando:
  `dd if=/dev/urandom of=mificheroaleatorio bs=1000 count=20`
- Puedes comprobar que el fichero enviado coincide con el fichero `f_recibido` generado por el servidor usando el comando `cmp mificheroaleatorio f_recibido`
- Al principio se recomienda lanzar el servidor en el equipo del laboratorio (que es donde se evaluará la entrega), en el puerto 32002 (otros puertos pueden estar filtrados por cortafuegos), y el cliente en el mismo equipo. Cuando funcione se recomienda probar a lanzar el cliente en Hendrix. Recuerda

que los equipos del laboratorio tienen procesadores muy distintos a los de Hendrix, con lo que no sirven los mismos ejecutables ni los mismos ficheros objeto generados durante la compilación. Haz `make clean` (o `gmake clean`) antes de compilar en otro equipo

- Puedes utilizar *Wireshark* para monitorizar los paquetes que viajan por la red

5.3. Preguntas sobre el funcionamiento

6. Dado que el servidor nunca envía datos, ¿qué habrá en el campo *next* del mensaje enviado por el cliente?

7. Dados los tiempos de transmisión y propagación que simula el servidor, asumiendo que no hay errores, ¿cuánto debería costar la transferencia de un fichero de 20 kB con el algoritmo básico?

8. Lanza el servidor sin errores (`-a0`) y anota cuánto tiempo tarda tu cliente en transferir un fichero de 20 kB. ¿Se parece al tiempo anterior?

9. Lanza ahora el servidor con errores (`-a1`) y anota cuánto tiempo tarda tu cliente (`-a1`) en transferir el mismo fichero. Ten en cuenta que, si no has especificado lo contrario, el servidor genera un mensaje incorrecto de cada cinco. ¿Cuánto ha tardado la transferencia en este caso? ¿qué relación tiene con respecto al tiempo sin errores?

6. Algoritmo *Stop&Wait*

Esta versión se lanzará mediante el parámetro `-a2`, tanto en el cliente como en el servidor RCFTP. Para esta versión hay que añadir el tratamiento de pérdidas de mensajes al algoritmo básico.

Para implementar este algoritmo es necesario utilizar tiempos de expiración o *timeouts*. Una opción es utilizar la señal de alarma, tal y como se ha visto en la asignatura *Sistemas Operativos*. No obstante, la opción recomendada es utilizar el código *multialarm* proporcionado. La siguiente sección explica cómo usarlo.

6.1. Gestión de timeouts mediante *multialarm*

El código *multialarm* proporciona una interfaz sencilla para el manejo de múltiples alarmas simultáneamente, aunque para el algoritmo *Stop&Wait* sólo se necesita una. Además, al añadir un timeout bloquea momentáneamente al proceso, simulando el *tiempo de transmisión* que requeriría el envío asociado al timeout.

- Antes de programar timeouts hay que especificar (una sola vez) su duración y la duración del tiempo de transmisión, en microsegundos, mediante la función *settimeoutduration()*. Esto ya está hecho en la plantilla proporcionada a partir de los parámetros introducidos por línea de comandos.

- Antes de usar los temporizadores de *multialarm* hay que asociar la rutina de servicio de la alarma a la función *int handle_sigalrm(int signal)* mediante:

```
signal(SIGALRM, handle_sigalrm);
```

- La función *void addtimeout()* programa un timeout que vencerá después de los microsegundos indicados en la función anterior.

- La función *void canceltimeout()* cancela el próximo timeout a vencer.

- La función *int getnumtimeouts()* devuelve el número de timeouts pendientes de vencer[1].

[1] En Stop&Wait nunca va a haber más de un timeout pendiente de vencer.

Algorithm 2 Pseudocódigo que sustituye a «recibir(respuesta)» en el algoritmo básico

```
addtimeout()
esperar ← true
while esperar do
  numDatosRecibidos ← recibir(respuesta) /*** No bloqueante: devuelve -1 si no hay datos ***/
  if numDatosRecibidos> 0 then
    canceltimeout() /*** Se puede realizar después de comprobar que es el mensaje esperado ***/
    esperar ← false
  end if
  if timeouts_procesados ≠ timeouts_vencidos then
    esperar ← false
    timeouts_procesados ← timeouts_procesados+1
  end if
end while
/*** comprobar respuesta SOLO si realmente se ha recibido; reenviar mensaje en caso contrario ***/
```

- La variable *volatile int timeouts_vencidos* almacena en todo momento el número de timeouts vencidos. Ten en cuenta que esta variable *volatile* se modifica desde la rutina de servicio, con lo que para evitar posibles inconsistencias *no hay que modificarla* desde el programa. La forma más sencilla para actuar ante el vencimiento de los timeouts es comparar esta variable con otra (e.g. *timeouts_procesados*) inicializada a 0 y, si son distintas, tratar el timeout (salir del bucle de espera) e incrementar en uno la otra variable (ver algoritmo 2).

6.2. Interrupciones de llamadas al sistema

Ten en cuenta que un timeout (señal de alarma) puede vencer mientras el programa está bloqueado en una llamada al sistema (e.g. *read*, *recvfrom*). Si esto sucede en *hendrix*, la llamada acabará con resultado -1 y error *EINTR*. GNU/Linux no presenta este comportamiento, y la llamada al sistema *no es interrumpida*. Para conseguir el mismo comportamiento en ambos sistemas, la opción recomendada es configurar el socket como «no bloqueante». Con esta configuración, si *recvfrom* (o *read*) no tiene datos que recibir, devolverá -1 y error *EAGAIN* en lugar de bloquearse esperando datos. Para poner el socket en modo «no bloqueante» se puede usar el siguiente código:

```
int sockflags;
sockflags = fcntl(socket,F_GETFL,0);      //obtiene el valor de los flags
fcntl(socket, F_SETFL, sockflags | O_NONBLOCK); //modifica el flag de bloqueo
```

6.3. Ayuda a la implementación del algoritmo *Stop&Wait*

Como ayuda a la implementación, ten en cuenta lo siguiente:

- El código cliente proporcionado ya incluye todo lo necesario para usar *multialarm*
- Si vence un timeout, seguramente no habrás recibido el mensaje esperado. ¡No verifiques la corrección de mensajes que no se han recibido!
- Funcionalmente, lo único que cambia respecto al algoritmo básico *al enviar* es que además de enviar hay que añadir un timeout.
- Funcionalmente, lo único que cambia respecto al algoritmo básico *al recibir* es que hay que esperar una respuesta *o el vencimiento de un timeout*. El algoritmo 2 detalla este comportamiento.

6.4. Preguntas sobre el funcionamiento

De forma similar a como habías hecho con el algoritmo básico, lanza el servidor con errores (`-a2`) y anota cuánto tiempo tarda tu cliente en transferir un fichero del mismo tamaño usando *Stop&Wait* (`-a2`).

10. Compáralo con los tiempos del algoritmo básico (`-a1` en cliente y servidor). ¿Con *Stop&Wait* y pérdidas tarda más o menos? ¿Por qué?

11. Lanza ahora tu cliente *Stop&Wait* (`-a2`) con errores básicos en el servidor (`-a1`) y compáralo con lo que tarda el algoritmo básico de tu cliente (`-a1`). ¿Qué puedes deducir?

12. Finalmente, compara cuánto se tarda en transferir el mismo fichero con tus dos versiones del cliente (`-a1`, `-a2`) con el servidor sin errores (`-a0`). ¿Tiene sentido?

7. Algoritmo *Go-Back-n*

Para esta versión, el programa cliente deberá usar el algoritmo de ventana deslizante *Go-Back-n*. Se utilizará el parámetro `-a3`, tanto en el cliente como en el servidor RCFTP, y el parámetro `-w[bytes]` indicará el tamaño de la ventana de emisión a utilizar, en bytes. Es importante destacar que ciertos detalles de *Go-Back-n* pueden implementarse de formas distintas, es decir, no hay una única implementación correcta. En la realidad, cada sistema operativo implementa su versión particular de TCP, que incorpora una versión adaptada de ventana deslizante. Por ejemplo, en tu implementación deberás tomar varias decisiones:

- Cuando el cliente reciba una respuesta, ¿debe comprobar si es un duplicado de la anteriormente recibida y en ese caso ignorar esa respuesta, o debe tratar cada mensaje de forma independiente?
- Cuando el cliente reciba una respuesta incorrecta, ¿debe asumir que el mensaje enviado se ha perdido y reenviarlo, o debe simplemente ignorar la respuesta incorrecta y no reenviar hasta que venza el timeout?
- Cuando el cliente reciba una respuesta incorrecta, ¿debe reenviar obligatoriamente los mensajes posteriores que haya en su ventana, o debe dar cierto margen por si son confirmados?
- etc.

Dependiendo de las decisiones que tomes, la transmisión será más o menos eficiente, más o menos rápida, le afectarán más o menos ciertos errores, etc. Independientemente de los detalles anteriores, la implementación debería tener un esquema general como el mostrado en el algoritmo 3.

7.1. Gestión de timeouts e interrupciones

El algoritmo *Go-Back-n* necesita poder manejar varios timeouts simultáneamente. Para ello, puedes utilizar el código *multialarm* (sección 6.1). Además, este algoritmo necesita que el cliente no se bloquee al realizar *recvfrom*s (o *read*s) cuando no haya datos que recibir. Este comportamiento se puede conseguir de varias formas (hilos de ejecución, funciones `select`, `poll`, etc.). La recomendación para esta práctica es configurar el socket como «no bloqueante» (sección 6.2), ya que te permitirá reutilizar en gran medida el código de los algoritmos anteriores.

7.2. Ayuda a la implementación del algoritmo *Go-Back-n*

- Se recomienda haber completado el algoritmo *Stop&Wait* antes de pasar al *Go-Back-n*.
- Es importante verificar que el cliente sigue funcionando si se especifica `-a1` y `-a2` en el servidor. ¡Si el cliente *Go-Back-n* funciona con `-a3` en el servidor, pero no con `-a2` en el servidor, es que el cliente no funciona correctamente!
- Ten en cuenta que el servidor podría aceptar parte de los datos de un mensaje. En ese caso el cliente no debería reenviar el mensaje anterior, sino construir uno nuevo conteniendo parte de los datos anteriores junto con datos nuevos.
- La ventana de emisión debe almacenar sólo datos, y no mensajes completos (*struct rcftp_msg*). Puedes usar el código proporcionado en `vemision.c/.h`, que implementa una cola circular estática en un vector (tal y como has visto en las asignaturas *Programación II* y *Estructuras de datos y algoritmos*) con un índice adicional para reenvíos (ver figura 1).

Algorithm 3 Esquema general del cliente con ventana deslizante

```
while ultimoMensajeConfirmado = false do
  /*** BLOQUE DE ENVIO: Enviar datos si hay espacio en ventana ***/
  if espacioLibreEnVentanaEmision and finDeFicheroNoAlcanzado then
    datos ← leerDeEntradaEstandar()
    mensaje ← construirMensajeRCFTP(datos)
    enviar(mensaje)
    addtimeout()
    añadirDatosAVentanaEmision(datos)
  end if
  /*** BLOQUE DE RECEPCION: Recibir respuesta y procesarla (si existe) ***/
  numDatosRecibidos ← recibir(respuesta) /*** No bloqueante: devuelve -1 si no hay datos ***/
  if numDatosRecibidos > 0 then
    if esMensajeValido(respuesta) and esLaRespuestaEsperada(respuesta) then
      canceltimeout()
      liberarVentanaEmisionHasta(respuesta.confirmado)
      if esConfirmacionDeUltimosDatos(respuesta) then
        ultimoMensajeConfirmado ← true
      end if
    end if
  end if
  /*** BLOQUE DE PROCESADO DE TIMEOUT ***/
  if timeouts_procesados ≠ timeouts_vencidos then
    mensaje ← construirMensajeMasViejoDeVentanaEmision()
    enviar(mensaje)
    addtimeout()
    timeouts_procesados ← timeouts_procesados+1
  end if
end while
```

7.3. Preguntas sobre el funcionamiento

13. Lanza el servidor con errores (`-a3`) y anota cuánto tiempo tarda tu cliente en transferir un fichero del mismo tamaño que en las pruebas anteriores usando *Stop&Wait* (`-a2`). Repite la operación utilizando el cliente con *Go-Back-n* (`-a3`) y compara el tiempo necesario para transferir el fichero. ¿Qué puedes deducir?

14. Lanza otra vez ambos con `-a3`, pero ahora especificando en el cliente un timeout de 0,5 s. ¿Qué sucede en la comunicación?

15. Lanza otra vez ambos con `-a3`, pero ahora especificando en el cliente un timeout de 5 s. ¿Qué sucede en la comunicación?

16. Compara ahora la comunicación (ambos con `-a3`) con un servidor con $T_t = 50$ ms (pon también este valor en el cliente) y $T_p = 300$ ms, frente a otra comunicación con el servidor con $T_t = 300$ ms (pon también este valor en el cliente) y $T_p = 50$ ms. ¿Cuál funciona mejor? ¿Por qué?

17. Calcula el valor óptimo de la ventana para los tiempos por defecto ($T_t = 200$ ms, $T_p = 250$ ms) y pásalo a unidades de byte según el tamaño del mensaje RCFTP. Lanza una comunicación con un valor de ventana (`-w`) la mitad del valor anterior (aproximadamente), y otra comunicación con un valor de ventana el doble del anterior (aproximadamente). ¿Qué puedes deducir?

8. Evaluación del trabajo

El trabajo deberá entregarse antes de la fecha *límite* especificada a través de Moodle. Antes de la fecha límite se pueden realizar entregas *recomendadas* del trabajo a través de Moodle (¡revisa las fechas

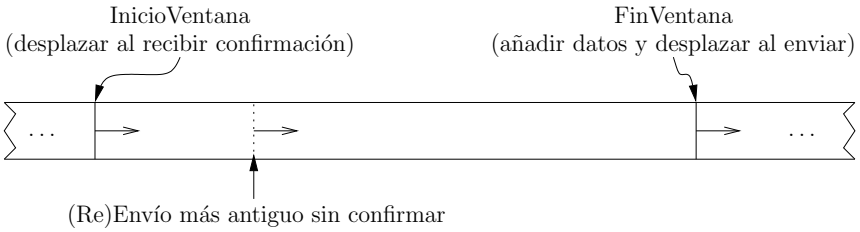

Figura 1: Esquema de funcionamiento de la ventana de emisión.

concretas!). Posteriormente a cada fecha de entrega recomendada, se evaluarán los trabajos entregados y se informará a los autores de los resultados. En caso de que se encuentren fallos, puedes aprovechar para corregirlos y entregar una nueva versión del trabajo antes de la fecha límite.

Recuerda que puedes acudir a tu profesor para resolver dudas del trabajo. No obstante, ten en cuenta que es un trabajo de redes de computadores y no de programación. Los profesores de la asignatura no vamos a evaluar aspectos de programación *ni a resolver dudas sobre programación*. Sólo se atenderán consultas relativas a la comprensión del trabajo a realizar y los algoritmos para realizarlo, es decir, *sin revisar el código*.

Al realizar la entrega habrá que tener en cuenta los siguientes puntos:

1. Hay que comprimir todos los fuentes necesarios para compilar el *cliente* (¡no el servidor!) en un único fichero. Los ficheros a comprimir deberán estar en un fichero *tar* comprimido mediante *gzip* (con extensión resultante `.tar.gz`). Recuerda que el fichero `Makefile` incluye un objetivo para generar este fichero.

2. Entre los ficheros comprimidos debe encontrarse un `Makefile`, a través del cual el programa debe *compilar sin errores* en el sistema GNU/Linux de los equipos del laboratorio y debe funcionar tal y como se indica en el enunciado.

3. El ejecutable debe imprimir por pantalla el nombre del autor o autores (máximo 2).

4. No es imprescindible que las entregas tengan implementados todos los algoritmos (básico, *Stop&Wait*, *Go-Back-n*).

5. Para cada fecha de entrega, se permiten *múltiples reenvíos* de la práctica, es decir, si después de enviarla se detecta algún error, se puede volver a enviar. Se evaluará únicamente la última versión enviada.

6. En caso de no realizar la entrega límite, se obtendrá la nota de la última entrega recomendada, si existe.

7. La evaluación del trabajo incluye un *control anticopia* que compara cada trabajo con otros trabajos entregados (incluyendo los de convocatorias y cursos anteriores). Si el porcentaje de similitud de dos trabajos supera cierto umbral se calificarán con 0.

Si no se cumple algún punto, se podrá reducir la nota obtenida.

9. ¿Sabías que...?

- Existen herramientas para generar automáticamente la documentación de código. Para ello, los comentarios introducidos en el código deben seguir cierto formato. En el código de la práctica, el código se ha comentado de forma que la herramienta *doxygen* pueda generar su documentación. Esta herramienta se puede usar sobre una gran variedad de lenguajes de programación (C, C++, Java, C#, PHP, Fortran, VHDL, etc.) y permite generar documentación en múltiples formatos, tales como HTML, XML, RTF, páginas de `man`, PostScript y PDF (vía LaTeX).