Prácticas de Sistemas Telemáticos – I
Práctica 4:
Descarga de ficheros
Ingeniería de Telecomunicación

Grupo de Sistemas y Comunicaciones

Abril de 2008

Resumen

Siguiendo con el modelo P2P presentado en la práctica 3, en esta práctica se construirá un nodo o *peer* capaz de descargar un fichero de otro nodo, y de servir sus ficheros a nodos que se lo pidan.

La primera sección de este documento contiene la especificación de la funcionalidad que debe ofrecer cada nodo. A continuación se describe el protocolo de transferencia de ficheros que han de implementar los nodos. Después se describe el formato de los mensajes intercambiados por dicho protocolo. Posteriormente se describen unas pautas de implementación, concluyendo con una sección que ilustra la interfaz de programación que deberá utilizarse en esta práctica para leer y escribir ficheros.

1. Especificación

- El `peer` puede lanzarse con 2 o con 5 parámetros, es decir, en la forma:

  ```
  ./peer <puerto> <directorio>
  ```

  o bien en la forma:

  ```
  ./peer <puerto> <directorio> <máquina> <puerto> <fichero>
  ```

- Cuando el `peer` se lance con 2 parámetros, servirá cualquiera de los ficheros contenidos a partir del directorio dado (directorio de exportación). El puerto de servicio en el que el `peer` acepta las peticiones de ficheros será el primer parámetro que se le pase en la línea de mandatos al arrancarlo. El nombre del directorio (carpeta) cuyos ficheros se servirán será el segundo parámetro en la línea de mandatos. Al nodo se le podrá pasar como directorio uno especificado con un trayecto absoluto (por ejemplo, `/tmp/dir1`) o uno especificado con un trayecto relativo (por ejemplo `dir1`). En este último caso el directorio será relativo, lógicamente, al directorio desde el que se lanza el `peer`.

- Cuando el `peer` se lance con 5 parámetros, además de servir en el puerto del primer parámetro los ficheros del directorio especificado en el segundo parámetro, el programa pedirá un fichero a otro `peer`. El tercer y el cuarto parámetro serán el nombre de máquina y puerto del `peer` al que se le pide el fichero y el quinto parámetro será el nombre del fichero que se le pedirá. Dicho nombre de fichero **siempre se considerará que está por debajo (dentro) del directorio (carpeta) que sirve el otro peer**.

  El fichero descargado por un nodo se guardará en el directorio especificado en su segundo argumento, es decir, en el directorio que él mismo sirve.

  Ejemplo de invocación de un `peer` que sólo sirve ficheros y que no pide ninguno:

  ```
  ./peer 8444 dir1
  ```

  Ejemplo de invocación de un `peer` para que sí pida un fichero:

```
./peer 8222 dir2 zeta23 8444 fich1
```

- Cuando un `peer` pida un fichero a otro lo hará bloque a bloque, como se describe más adelante. El `peer` que tiene el fichero enviará los bloques solicitados al `peer` que se los pida.
- El fichero pedido por un `peer` se guardará **con el mismo nombre** en el mismo directorio que él exporta. Es decir, si se lanza un `peer` con:

  ```
  ./peer 8222 dir2 zeta23 8444 fich1
  ```

  éste hace lo siguiente:

  - Exporta a otros nodos los ficheros que tiene en la carpeta `dir2` (que en este ejemplo, por ser dir2 un directorio relativo que no comienza por `/`, estará debajo de la carpeta en la que está el programa ejecutable `peer`).
  - Pide al nodo que se ejecuta en la máquina `zeta23`, puerto `8444`, el fichero de nombre `fich1` que estará en el directorio de exportación de ese otro nodo. El fichero que recibe se guardará con nombre `fich1` en la carpeta `dir2`.
- Una vez terminada la descarga, **desde una ventana de terminal** debe comprobarse que el fichero transferido es idéntico al original. Para ello puede usarse el mandato de shell `cmp`:

  ```
  cmp fich_origen fich_destino
  ```

  Si los ficheros son idénticos, `cmp` no escribe nada. Si los ficheros se diferencian en algo, `cmp` muestra el primer byte en el que difieren.

2. Protocolo de Transferencia de Ficheros

El nodo que pide el fichero irá enviando mensajes solicitando en cada uno de ellos un bloque de bytes del fichero. El bloque de bytes se especifica indicando la posición del primer byte del bloque solicitado, y la cantidad máxima de bytes que se espera recibir para ese bloque. El primer byte de un fichero es el que ocupa la posición número 1.

El nodo que tiene el fichero irá recibiendo las peticiones de bloques, leerá del disco los bytes solicitados, y los enviará como respuesta al nodo que le hizo la petición. En función del tamaño del fichero, puede ocurrir que algunos de los bytes solicitados no existan. En este caso el nodo que tiene el fichero completará el mensaje con tantos bytes como pueda. Por ejemplo, si el tamaño del fichero es de 2250 bytes, y llega una solicitud de 300 bytes a partir del byte número 2001, el nodo enviará los últimos 250 bytes del fichero (del 2001 al 2250), a pesar de que se le han solicitado 300.

Para pedir bloques del fichero se utilizarán mensajes **DATAREQ** con el formato que se describe más adelante. Para enviar los bloques del fichero se utilizarán mensajes **DATA** con el formato que se describe más adelante. En ciertas ocasiones un nodo puede responder con un mensaje **DATAERR** con el formato que se describe más adelante, en lugar de responder con un mensaje DATA. Los mensajes DATAERR se utilizan para indicar al nodo peticionario situaciones como la inexistencia de un fichero pedido o la inexistencia de un bloque pedido.

El nodo que pide el fichero desconoce inicialmente el tamaño de ese fichero. Los mensajes DATAREQ pueden incorporar campos opcionales para pedir al nodo que tiene el fichero que el mensaje DATA correspondiente incluya también el tamaño del fichero, medido en bytes.

El protocolo de transferencia de ficheros entre nodos no utilizará más tipos de mensajes que los tres anteriormente mencionados.

Para esta práctica se considerará que no se producen pérdidas ni desorden de mensajes en las comunicaciones entre nodos.

3. Formato de los mensajes

Todos los mensajes tienen unos **campos fijos**. Opcionalmente, **detrás** de los campos fijos, puede haber más campos, a los que denominaremos **Opciones**.

3.1. Campos fijos

Todos los mensajes tienen un primer campo que indica el **Tipo** del mismo. Dicho campo debe ser un valor del siguiente tipo enumerado de Ada (tipo que debe definirse de esta manera en el código del programa):

```
type Message_Type is (DATAREQ, DATA, DATAERR);
```

Todos los mensajes tienen un segundo campo que indica el **Número de Opciones** que tiene el mensaje. Este campo debe ser un valor de tipo `Natural`. Si el valor es 0, el mensaje no tiene opciones.

El resto de campos fijos dependen del tipo de mensaje. Más abajo se describen los campos de cada tipo de mensaje.

3.2. Opciones

Cada opción se compone de un campo que indica el **tipo de opción**, y cero o más campos adicionales, en función del tipo de opción. El tipo de opción debe ser un valor del siguiente tipo enumerado de Ada (tipo que debe definirse de esta manera en el código del programa):

```
type Option_Type is (SIZEREQ, SIZE, FILE_NOT_FOUND, BLOCK_NOT_FOUND);
```

3.3. Mensajes

A continuación se describen todos los tipos de mensaje con sus campos fijos y sus opciones:

- Mensaje `DATAREQ`:

  Tras enviarse un mensaje `DATAREQ` puede recibirse como respuesta un mensaje `DATA` o un mensaje `DATAERR`.

  Campos fijos:

DATAREQ	OPTIONS	EP_RES	FILE_NAME	BLOCK_POS	BLOCK_SIZE

  - `DATAREQ`: El tipo del mensaje (valor del tipo Message_Type).
  - `OPTIONS`: El número de opciones (de tipo `Natural`).
  - `EP_RES`: `End_Point` en el que el nodo que pide un bloque de fichero espera la respuesta conteniendo el bloque pedido.
  - `FILE_NAME`: Nombre del fichero cuyo bloque se pide en el siguiente campo (de tipo `Unbounded_String`).
  - `BLOCK_POS`: Posición del primer byte del bloque del fichero que se pide (de tipo `Positive`). El primer byte de un fichero es el byte número 1.
  - `BLOCK_SIZE`: Tamaño del bloque pedido, medido en bytes (de tipo `Positive`).

  Opciones:

SIZEREQ

  - `SIZEREQ`: El tipo de opción (valor del tipo `Option_Type`). Se utiliza para pedir al receptor del mensaje `DATAREQ` que incluya el tamaño del fichero (medido en bytes) en el mensaje `DATA` que enviará como respuesta a este mensaje `DATAREQ`. Este tipo de opción no lleva campos adicionales.

  Ejemplos:

  Así, si se envía un `DATAREQ` pidiendo los 1024 primeros bytes del fichero `f1` sin opciones, el mensaje será:

DATAREQ	0	EP_RES	f1	1	1024

  Y si se envía el mismo `DATAREQ` con la opción `SIZEREQ`, el mensaje será:

DATAREQ	1	EP_RES	f1	1	1024	SIZEREQ

- Mensaje `DATA`:

  Se envía un mensaje `DATA` cuando se ha recibido un mensaje `DATAREQ`, y el bloque pedido existe. Si la primera posición del bloque pedido es mayor que el tamaño del fichero, los bytes pedidos no existen, en cuyo caso no se enviará un mensaje `DATA`, sino un mensaje `DATAERR`.

  Campos fijos:

DATA	OPTIONS	FILE_NAME	BLOCK_POS	BLOCK_SIZE	BLOCK_DATA

  - `DATA`: El tipo del mensaje (valor del tipo Message_Type).
  - `OPTIONS`: El número de opciones (de tipo `Natural`).
  - `FILE_NAME`: Nombre del fichero cuyo bloque se envía (de tipo `Unbounded_String`).
  - `BLOCK_POS`: Posición del primer byte del bloque del fichero que se envía (de tipo `Positive`). El primer byte de un fichero es el byte número 1. El valor de este campo deberá coincidir siempre con el campo `BLOCK_POS` del mensaje `DATAREQ` en el que se pedía este bloque de datos.
  - `BLOCK_SIZE`: Longitud de bloque del fichero que se envía (de tipo `Positive`). Este campo tendrá siempre un valor menor o igual al campo `BLOCK_SIZE` del mensaje `DATAREQ` en el que se pedía este bloque de datos.
  - `BLOCK_DATA`: Contenidos del bloque `BLOCK_NUM` del fichero `FILE_NAME` (de tipo `Stream_Element_Array`).

  Opciones:

SIZE	BYTES

  - `SIZE`: El tipo de opción (valor del tipo Option_Type). Se utiliza para responder a un mensaje `DATAREQ` recibido con la opción `SIZEREQ`. Este tipo de opción lleva un campo adicional que se describe a continuación:
  - `BYTES`: Tamaño del fichero medido en número de bytes. De tipo `Natural`. Pueden existir ficheros de tamaño 0. Sin embargo para estos ficheros nunca se enviaría un mensaje `DATA`, ya que tras recibirse el mensaje `DATAREQ` en el que se pide el primer bloque, se deberá contestar con un mensaje `DATAERR`, indicando que el bloque no existe,

  Ejemplos:

  Así, si se envía un `DATA` con los 1024 primeros bytes del fichero `f1` sin opciones, el mensaje será:

DATA	0	f1	1	1024	BLOCK_DATA

  Y si se envía el mismo `DATA` con la opción `SIZE`, teniendo el fichero `f1` un tamaño de 21345 bytes, el mensaje será:

DATA	1	f1	1	1024	BLOCK_DATA	SIZE	21345

- Mensaje `DATAERR`:

  Se envía un mensaje `DATAERR` tras recibirse un mensaje `DATAREQ` en el que se solicitan un bloque de datos cuyo primer byte es mayor que el tamaño del fichero, o cuando se solicitan bytes de un fichero que no existente.

  Campos fijos:

DATAERR	OPTIONS	FILE_NAME	BLOCK_POS

  - `DATAERR`: El tipo del mensaje (valor del tipo Message_Type).
  - `OPTIONS`: El número de opciones (de tipo `Natural`).
  - `FILE_NAME`: Nombre del fichero del que se informa (de tipo `Unbounded_String`).
  - `BLOCK_POS`: Posición del primer byte del bloque del fichero del que se informa (de tipo `Positive`).

  Opciones:

FILE_NOT_FOUND

- FILE_NOT_FOUND: El tipo de opción (valor del tipo Option_Type). Indica que el fichero pedido no existe. Este tipo de opción no lleva campos adicionales.

- BLOCK_NOT_FOUND
  - BLOCK_NOT_FOUND: El tipo de opción (valor del tipo Option_Type). Indica que el primer byte del bloque pedido no existe por ser mayor que el tamaño del fichero. Este tipo de opción no lleva campos adicionales. Cuando se recibe una solicitud de un bloque cuyo primer byte de datos es menor que el tamaño del fichero, pero no hay bytes de datos suficientes para enviar todos los bytes del bloque solicitado, NO se envía un mensaje DATAERR, sino un mensaje DATA con los todos los bytes que se puedan completar.

Ejemplos:

Así, si un nodo recibe una petición de la posición 1 del fichero f1, y no tiene en su directorio de exportación dicho fichero, enviará el siguiente DATAERR:

DATAERR	1	f1	1	FILE_NOT_FOUND

Y si un nodo recibe una petición de la posición 4097 del fichero f1, y no existe el byte de esa posición en dicho fichero, enviará el siguiente DATAERR:

DATAERR	1	f1	4097	BLOCK_NOT_FOUND

4. Sugerencias de Implementación

- En el manejador debe estar el código que atiende peticiones de bloques de ficheros realizadas desde otros nodos. Fundamentalmente debe encargarse de leer del disco un bloque del fichero pedido y enviarlo como mensaje de respuesta

- En el programa principal debe realizarse en primer lugar el Bind en el End_Point de servicio, y después, sólo si se ha lanzado el nodo con 5 parámetros, pedir en un bucle (bloque a bloque) el fichero especificado, empezando con el bloque que comienza en la posición 1. Con la opción SIZEREQ del primer mensaje DATAREQ se puede pedir información sobre el tamaño del fichero, y de esta forma saber cuántos bloques distintos hay que pedir. Para esta práctica se recomienda pedir los bloques utilizando siempre un tamaño de bloque (BLOCK_SIZE) de 1024 bytes.

- El nombre del directorio donde están los ficheros que se exportan y donde se salva el fichero que se descarga debe guardarse dentro del programa en una variable a la que debe tener acceso tanto el programa principal como el manejador. Por ello, el sitio adecuado para declarar dicha variable es la especificación (.ads) del paquete del manejador[1]. De esta forma el procedimiento manejador podrá acceder a dichas variables directamente, y el programa principal tendrá acceso a ellas, ya que hace with del paquete del manejador. El programa principal deberá asignar el nombre del directorio a esa variable antes de llamar a Bind con el manejador. De esta forma la variable tendrá el valor adecuado cuando se ejecute el manejador.

5. Acceso a ficheros en Ada

El paquete Ada.Streams.Stream_IO permite el acceso a ficheros tratándolos, básicamente, como un array de bytes.

El acceso a los ficheros para lectura o escritura se hace por bloques de bytes. Para almacenar cada bloque se usa el tipo predefinido de Ada Ada.Streams.Stream_Element_Array, que es un array irrestringido de elementos de tipo Ada.Streams.Stream_Element, que es esencialmente un byte.

Así, un ejemplo simple de acceso a ficheros sería:

```
with Ada.Streams;
with Ada.Streams.Stream_IO;
```

[1] Siguiendo el principio de la encapsulación/ocultación de información, alternativamente se puede declarar la variable en el cuerpo del paquete del manejador, y proporcionar en la especificación del paquete dos subprogramas: un procedimiento para modificar el valor de la variable y una función para consultar su valor

```
procedure Prueba_Ficheros is
   package S_IO renames Ada.Streams.Stream_IO;

   -- tamaño en bytes de los bloques a leer/escribir
   Tamaño_Bloque: constant := 1024;
   -- bloque de datos del fichero
   Bloque: Ada.Streams.Stream_Element_Array(1..Tamaño_Bloque);

   -- último byte leído (si no quedaba un bloque entero)
   Último: Ada.Streams.Stream_Element_Offset;

   Fichero_Origen: S_IO.File_Type;
   Fichero_Destino: S_IO.File_Type;

begin
   S_IO.Open(Fichero_Origen, S_IO.In_File, "/etc/hosts");
   S_IO.Create(Fichero_Destino, S_IO.Out_File, "/tmp/prueba");

   while not S_IO.End_Of_File(Fichero_Origen) loop
      S_IO.Read(Fichero_Origen, Bloque, Último);

      -- hay que escribir sólo los bytes hasta Último, pues el último
      -- bloque del fichero puede que no esté entero
      S_IO.Write(Fichero_Destino, Bloque (1..Último));
   end loop;

   S_IO.Close(Fichero_Origen);
   S_IO.Close(Fichero_Destino);
end Prueba_Ficheros;
```

El último parámetro de `Read`, `Último`, es un parámetro pasado en modo `out` y cuando termina el `Read` contiene el número de bytes que se han leído. Siempre se intenta leer tantos bytes como tamaño tenga el array `Bloque`, pero puede ser (si se está leyendo el último bloque del fichero), que no haya suficientes bytes por leer. En ese caso se almacenan al principio del array `Bloque` los que se han podido leer, pero el resto de posiciones del array contendrá valores obsoletos. Esta es la razón de que al escribir el `Bloque` con `Write` se escriba sólo la rodaja de los `Último` primeros bytes: de esta forma nos aseguramos de escribir sólo los bytes que se han leído.

Aparte de los subprogramas que aparecen en el ejemplo, existen más procedimientos y funciones en el paquete `Ada.Streams.Stream_IO` que pueden ser de interés para la realización de esta fase. En particular, pueden ser relevantes los siguientes:

- El procedimiento `Set_Index` ajusta la posición del fichero en la que se efectuará la siguiente operación de lectura o escritura. Si no se invoca nunca, las operaciones de lectura o escritura serán secuenciales, esto es, se harán en la posición siguiente a la anterior operación. En cambio, si se llama a `Set_Index` la posición de la próxima operación de lectura o escritura se efectuará a partir de la posición especificada en el parámetro `To`.

  ```
  procedure Set_Index (File : in File_Type; To : in Positive_Count);
  ```

- Además del `Read` del ejemplo (de 3 parámetros), existe otro procedimiento `Read` que incluye un cuarto parámetro que indica a partir de qué byte se efectuará la lectura:

  ```
  procedure Read
    (File : in  File_Type;
     Item : out Stream_Element_Array;
     Last : out Stream_Element_Offset;
     From : in  Positive_Count);
  ```

  Es equivalente invocar este `Read` de cuatro parámetros a llamar primero a `Set_Index` y luego al `Read` de tres parámetros.

- Además del `Write` del ejemplo (de 2 parámetros), existe otro procedimiento `Write` que incluye un tercer parámetro que indica a partir de qué byte se efectuará la escritura:

```
procedure Write
  (File : in File_Type;
   Item : in Stream_Element_Array;
   To   : in Positive_Count);
```

  Es equivalente invocar este `Write` de tres parámetros que llamar primero a `Set_Index` y luego al `Write` de dos parámetros.

- La función `Size` devuelve el tamaño en bytes de un fichero.

```
function Size  (File : in File_Type) return Count;
```

Por último, hay que resaltar que el tipo de los parámetros de posicionamiento es `Positive_Count`, tipo numérico que resulta incompatible con los `Integer`, `Natural` o `Positive`. Para mezclar estos tipos en la misma expresión es necesario usar la conversión explícita de tipos numéricos de Ada, como por ejemplo en:

```
   I: Integer;
   P: Ada.Streams.Stream_IO.Positive_Count;
...
   I := Integer(P);
   P := Ada.Streams.Stream_IO.Positive_Count(I);
```