Programación de Sistemas de Telecomunicación / Informática II
Práctica 5:
Chat-Peer v2.0
(Partes Opcionales)

Departamento de Sistemas Telemáticos y Computación
(GSyC)

Noviembre de 2013

Resumen

Este documento describe las partes opcionales de la Práctica P5: Chat-Peer v2.0. Junto al título del apartado en el que se describe cada parte opcional se da una orientación sobre el nivel de complejidad estimado.

En el apéndice A se describe cómo programar el acceso a ficheros que es necesario conocer para realizar la opción de transmisión de ficheros.

En el apéndice B se describen las normas y plazos de entrega de estas partes opcionales de la práctica P5.

1. Supernodo (Complejidad baja)

Hay dos formas mediante las que un nodo i puede conocer vecinos:

- Especificar en la línea de comandos del nodo i uno o dos nodos de contacto
- Que otros nodos consideren al nodo i como su nodo de contacto

En esta parte opcional implementarás una forma complementaria a estas dos para que un nodo i pueda conocer a otros nodos vecinos.

Tendrás que implementar un nuevo programa *supernodo* que almacene endpoints de otros nodos.

Cuando un nodo arranque se le podrá pasar en la línea de comandos un parámetro adicional con el nombre de host y puerto en los que está atado el *supernodo*. Tras arrancar, el nodo enviará un mensaje al *supernodo* para que éste almacene el `EP_H` del nodo que arranca. Como respuesta el *supernodo* le contestará enviándole unos cuantos `EP_H` de otros nodos que conozca el *supernodo*. Estos *endpoints* pasarán a ser considerados como vecinos por el nodo, añadiéndolos a su lista de vecinos.

Para implementar esta opción deberás definir todos los aspectos necesarios, como el protocolo, formato de mensajes, formato de entrada/salida, etc.

Para probar esta opción puedes desactivar la inyección de fallos y los retardos de propagación.

2. Topología (Complejidad media)

Implementa un protocolo para que desde un nodo se pueda descubrir cuál es la topología de la red de chat a la que está conectado el nodo. Con la información recuperada por el protocolo el nodo mostrará en la pantalla la relación de vecindad entre todos los nodos que forman parte de la red de chat.

Para implementar esta opción deberás definir todos los aspectos necesarios, como el protocolo, formato de mensajes, formato de entrada/salida, etc.

Para probar esta opción puedes desactivar la inyección de fallos y los retardos de propagación.

3. Implementación no recursiva del árbol de búsqueda binaria (Complejidad media)

La implementación de los subprogramas `Get, Put y Delete` de *Ordered_Maps_G* es recursiva.

Implementa estas operaciones sin utilizar la recursividad, realizando recorridos del árbol siguiendo los punteros `Left` y `Right`.

4. Implementación de tablas de símbolos con un array ordenado (Complejidad media)

Modifica la implementación de *Ordered_Maps_G* para que en lugar de utilizar un árbol de búsqueda binaria se utilice un array ordenado.

5. Implementación de tablas de símbolos con una tabla Hash (Complejidad media)

Modifica la implementación de *Ordered_Maps_G* para que en lugar de utilizar un árbol de búsqueda binaria se utilice una tabla hash.

6. Ejecución retardada de borrados de *Latest_Msgs* (Complejidad baja)

Utilizando manejadores temporizados programa el borrado retardado de los elementos de `latest_msgs` cuando se recibe un mensaje `Logout`. De esta forma podrás solucionar el problema de la reaparición de elementos en la lista `latest_msgs`.

6.1. Motivación

En Chat-Peer v2.0 puede ocurrir en ocasiones el siguiente escenario: llega a un nodo un mensaje *Logout* que provoca que se tenga que borrar la entrada correspondiente al creador de dicho mensaje de la lista de mensajes *latest_msgs*. Debido al protocolo de inundación y a las retransmisiones, posteriormente pueden llegar mensajes (*Writer, Confirm o Init*) procedentes del mismo nodo origen que envió el *Logout*. Estos casos provocan que aparezca de nuevo una entrada para dicho origen en *latest_msgs*, repitiéndose indefinidamente este ciclo de borrado/reaparición de elementos si más tarde vuelve a llegar un mensaje *Logout*.

Por esta razón en la parte básica de la práctica se indica que, como excepción al caso general, cuando llegue un *Logout* procedente de un EP que no esté en *latest_msgs*, debe ser ignorado. Esa solución tampoco es adecuada por completo pues también puede dar lugar a situaciones anómalas.

La solución que se propone en este apartado resuelve estos problemas.

6.2. Descripción de la solución

Cuando llegue un mensaje *Logout*, en lugar de borrar el elemento de la lista *latest_msgs* correspondiente al origen del mensaje en ese mismo instante, se actualizará la entrada con el número de secuencia del mensaje *Logout* y se programará un manejador temporizado para que cuando éste se ejecute en el futuro borre el elemento de *latest_msgs*.

Ahora ya pueden evitarse las situaciones anómalas:

- Hasta que el manejador temporizado no borre la entrada, si siguen llegando mensajes previos procedentes del mismo origen éstos serán ignorados por tener números de secuencia menores o iguales que el mensaje *Logout*, por lo que no se procesarán. Si se programa el manejador temporizado para que se ejecute en un instante futuro lo suficientemente lejano en el tiempo, dejarán de llegar mensajes previos procedentes del mismo origen una vez que el resto de los nodos deje de inundar dichos mensajes.

- Si se deja transcurrir el tiempo suficiente, cuando se ejecute el manejador es seguro realizar el borrado de la lista *latest_msgs* pues no se corre el riesgo de que reaparezca el elemento borrado al no recibirse ya más mensajes previos enviados por el mismo origen.

El instante de tiempo en el que se debería ejecutar el manejador temporizado para un determinado mensaje *Logout* debería dar suficiente margen para que no quedara ya circulando por la red de nodos ninguna copia de ese mensaje *Logout*. Debes pensar un valor razonable para calcular este tiempo, que debería estar en función del retardo máximo de propagación, del número de nodos actualmente en el sistema (aproximado por el número de entradas actualmente presentes en *latest_msgs*), y del porcentaje de fallos.

Para realizar esta implementación deberías utilizar una nueva tabla de símbolos ordenada de una forma similar al uso que se hace de *sender_buffering* para las retransmisiones.

7. Interrupción con Ctrl-C (Complejidad baja)

Si se termina un nodo con Ctrl-C no se ejecuta el protocolo de salida del nodo.

Utiliza el siguiente ejemplo para implementar la captura de la interrupción que se envía cuando se pulsa Ctrl-C en el teclado de forma que el usuario sea interrogado sobre si desea abandonar el chat o continuar en él, y en su caso, se realice el protocolo de salida antes de terminar el programa.

```
with Gnat.Ctrl_C;
with Manejador;
procedure Programa_Principal is
begin
   Gnat.Ctrl_C.Install_Handler (Manejador.Ctrl_C_Handler'Access);
   -- A partir de aquí si pulsa el usuario CTRL-C se ejecuta el código
   --  del manejador
end Programa_Principal;
```

```
package Manejador is
   procedure Ctrl_C_Handler;
end Manejador;
```

```
with Ada.Text_IO;
with Lower_Layer_UDP;
package body Manejador is
   procedure Ctrl_C_Handler is
   begin
      Ada.Text_IO.Put_Line ("Han pulsado CTRL-C... terminamos");
      Lower_Layer_UDP.Finalize;
      raise Program_Error;
   end Ctrl_C_Handler;
end Manejador;
```

8. Detección de duplicados tras el protocolo de admisión (Complejidad baja)

Supongamos que se arrancan dos grupos de nodos distintos, entre los cuales no hay comunicación. En cada uno de los grupos, al no estar conectados entre sí, pueden haberse lanzado nodos que estén utilizando *nicknames* iguales a los utilizados por nodos del otro grupo.

Si posteriormente se lanza un nodo que tiene como vecinos de contacto a un nodo de cada uno de los dos grupos, todos los nodos pasarían ahora a estar en el mismo chat al unir el nodo que se arranca a ambos grupo. Pero habría *nicksnames* duplicados. El protocolo de admisión no permite detectar esta situación.

Implementa una solución para este problema de forma que se puedan detectar apodos duplicados y se haga algo para que sólo pueda acabar habiendo un nodo con cada *nickname*.

9. Entrega fiable de los mensajes de unienvío como Reject (Complejidad alta)

En la práctica 5 has tenido que implementar el envío de mensajes de *Ack* y la retransmisión de mensajes de multienvío para que la aplicación se pueda recuperar de las pérdidas de mensajes.

En este apartado se describe cómo se debe implementar la entrega fiable de los mensajes que no se envían por inundación, como por ejemplo los mensajes *Reject*.

Lo descrito en esta sección es válido también para implementar la fiabilidad en el envío de otros mensajes nuevos de unienvío que se describen más adelante en este enunciado, como los de la transferencia de ficheros o los de los chats privados. Llamaremos a todos estos mensajes que no se envían por inundación **mensajes de unienvío**.

Resumen:

- Cuando se envíe/reenvíe un mensaje de unienvío como *Reject*, también se deberá almacenar el mensaje en las estructuras de datos *Sender_Buffering* y *Sender_Dests*, hasta que sea asentido. En el array *destinations* del elemento que se añada a *Sender_Dests* se deberá añadir un único endpoint, el del nodo destinatario, pues es el único que se espera que asienta el mensaje.
- Cada vez que se envíe/reenvíe un mensaje de unienvío deberá programarse su retransmisión mediante un manejador temporizado, igual que cuando se programa la retransmisión de los mensajes que se envían por inundación. Nótese que el mismo manejador temporizado programado para retransmitir los mensajes enviados por inundación sirve para retransmitir los mensajes de unienvío.
- Los mensajes de unienvío como *Reject* también tienen que ser asentidos por su receptor mediante un mensaje *Ack*.

9.1. Números de secuencia

Los mensajes de unienvío como *Reject* se envían *fuera de banda* respecto a los mensajes enviados por inundación y respecto a otros tipos de mensajes de unienvío. Esto quiere decir que los mensajes de unienvío Reject que crea un nodo no guardan relación con los mensajes de inundación que ese mismo crea, ni con otros mensajes de unienvío *Reject* enviados a otros nodos, ni con otros mensajes de unienvío distintos de *Reject* que pudiera enviar ese nodo.

Los mensajes de unienvío, como p.ej. *Reject*, llevarán ahora un campo `Seq_N` para poderlos identificar correctamente, al igual que se hace con los mensajes de multienvío. Para asignar valores a este campo no se puede emplear la misma secuencia de números que la utilizada para el campo `Seq_N` de los mensajes enviados por inundación, ni se pueden emplear las mismas secuencias utilizadas para numerar otros tipos de mensajes de unienvío.

9.1.1. Motivación

Veamos un ejemplo de por qué es necesario utilizar diferentes espacios de números de secuencia.

Supongamos que un nodo j está utilizando el mismo espacio de números de secuencia para numerar los mensajes *Reject* que el utilizado para numerar los mensajes que crea y envía por inundación:

1. En un momento dado el nodo j ha creado y enviado por inundación 4 mensajes distintos (*Init, Confirm, Writer, Writer*), numerándolos de 1 a 4 respectivamente.
2. A continuación recibe un mensaje *Init* creado por otro nodo i con el mismo *nickname* que está utilizando ya el nodo j. Entonces j tendrá que enviar un mensaje *Reject* a i. Dado que el último número de secuencia utilizado por j para etiquetar los mensajes por él creados fue el 4, el *Reject* llevará el número 5.
3. Si a continuación j crea y envía un nuevo mensaje *Writer*, su número de secuencia deberá ser el 6. Cuando este mensaje sea recibido en procesos distintos de i éstos no podrán procesarlo por no haber recibido el mensaje 5 aún. ¡Pero jamás lo recibirán, pues j sólo le envió el mensaje *Reject* etiquetado con el número de secuencia 5 a i!

9.1.2. Solución: espacios de números de secuencia independientes

Así pues, deberá utilizarse un espacio de números de secuencia para numerar los mensajes *Reject* que sea distinto al utilizado para numerar los mensajes de inundación, y distinto de los utilizados para numerar otros tipos de mensajes definidos más adelante en este enunciado de partes opcionales. Además, cada nuevo mensaje *Reject* que se envíe utilizará su propio espacio de números de secuencia, por lo que todos los mensajes *Reject* enviados llevarán el número de secuencia 1.

9.2. Identificador de sesión

Los mensajes de inundación son asentidos por mensajes *Ack*. También el resto de tipos de mensaje, y en particular los mensajes *Reject*, deben ser ahora asentidos por mensajes de tipo *Ack*.

Sin embargo, queda un problema por resolver: teniendo en cuenta las consideraciones del apartado anterior respecto a los números de secuencia se puede concluir lo siguiente: la recepción de un mensaje *Ack* no permite saber qué mensaje está asintiendo. Veamos por qué.

9.2.1. Motivación

Supongamos que un nodo j ha enviado su mensaje *Init* con número de secuencia 1. En ese instante recibe un mensaje *Init* de un nodo i que tiene su mismo *nickname*, y envía un mensaje *Reject*. Dado que se utilizan números de secuencia distintos para los mensajes *Reject*, este mensaje también será etiquetado con el número de secuencia 1 y no el 2, pues se utiliza un espacio de números de secuencia para los mensajes *Reject* distinto del utilizado para los mensajes enviados por inundación. Por tanto no se puede distinguir del mensaje *Init*, por llevar ambos el número de secuencia 1 y provenir del mismo origen, ni de otros mensajes *Reject* que tenga que enviar el mismo nodo.

9.2.2. Solución: sesiones

Existen diversas soluciones para solventar este problema. En esta parte opcional se adoptará una solución basada en el concepto de **sesión**:

- Todos los mensajes que crea un mismo nodo j para ser enviados por inundación pertenecen a una misma sesión
- Todos los mensajes de unienvío creados por un nodo j que tengan a un mismo nodo como destinatario pertenecen a una misma sesión, que es distinta a la sesión de mensajes de multienvío de j y a otras sesiones de unienvío definidas por j para destinatarios distintos.
- Cada sesión de j estará identificada por un *identificador de sesión*
- El identificador de sesión pasa a ser un nuevo campo denominado `Session_Id`, de tipo Positive, en todos los mensajes de inundación y de unienvío creados por cada nodo.
- Los mensajes *Ack* también se ven modificados, pasando a tener un nuevo campo antes del campo `EP_H_Creat`, denominado `Session_Id`, de tipo Positive

El lector podrá comprobar que utilizando los identificadores de sesión, el escenario de ejemplo de la sección 9.2.1 deja de plantear problemas: la recepción de un *Ack* no conlleva ambigüedad ya que los campos `Session_Id`, `EP_H_Creat` y `Seq_N` del mensaje *Ack* recibido identifican ahora a un único mensaje de los almacenados por j en la estructura de datos *Sender_Dests* pues el mensaje *Init* y el mensaje *Reject* pertenecen a sesiones con identificadores distintos.

9.3. Cambios en el formato de los mensajes

La introducción de los identificadores de sesión hace que el formato de todos los mensajes utilizados hasta ahora haya de ser modificado:

9.3.1. Mensajes Ack

Formato:

Ack	EP_H_ACKer	Session_Id	EP_H_Creat	Seq_N

donde el nuevo campo `Session_Id` es de tipo Positive, y el resto de campos son los mismos definidos en el enunciado de la parte básica de la práctica 5.

El mensaje que asiente un *Ack* queda identificado por los 3 campos `Session_Id`, `EP_H_Creat` y `Seq_N`.

9.3.2. Mensajes enviados por inundación

En todos los mensajes enviados por inundación el nodo que lo crea inserta ahora un campo detrás del tipo de mensaje. Este nuevo campo es el identificador de la sesión a la que pertenece el mensaje enviado.

Por ejemplo, el mensaje *Writer* pasa a tener el siguiente formato:

Writer	Session_Id	EP_H_Creat	Seq_N	EP_H_Rsnd	Nick	Text

- `Writer`: Valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `EP_H_Creat`: `EP_H` del nodo que creó el mensaje. Dicho nodo recibe mensajes de inundación en este `End_Point`.
- `Session_Id`: Valor del tipo `Positive` que identifica la sesión a la que pertenece el mensaje
- `Seq_N`: Valor del tipo `Seq_N_T`. Es el número de secuencia asignado al mensaje por el nodo `EP_H_Creat` en la sesión `Session_Id`.

- `EP_H_Rsnd`: `EP_H` del nodo que ha reenviado el mensaje. Cuando un nodo recibe un mensaje por inundación, este campo identifica al vecino que le ha reenviado a él el mensaje recibido.
- `Nick`: `Unbounded_String` con el *nick* del nodo que creó el mensaje.
- `Text`: `Unbounded_String` con el texto del mensaje.

El resto de mensajes de multienvío pasan a tener también este nuevo campo `Session_Id` detrás del campo de tipo de mensaje.

9.3.3. Mensaje Reject

En los mensajes de univenvío se añaden dos nuevos campos, uno para el identificador de sesión y uno para el número de secuencia. Así, el formato de los mensajes *Reject* pasa a ser el siguiente:

Reject	Session_Id	EP_H_Creat	Seq_N	Nick

en donde:

- `Reject`: Valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `Session_Id`: Valor del tipo `Positive` que identifica la sesión a la que pertenece el mensaje
- `EP_H_Creat`: *endpoint* `EP_H` para recepción mediante *handlers* de LLU del nodo que envía el mensaje.
- `Seq_N`: Valor del tipo `Seq_N_T`. Es el número de secuencia asignado al mensaje por el nodo `EP_H` en la sesión `Session_id`.
- `Nick`: `Unbounded_String` con el *nick* del nodo.

9.4. Cambios en las estructuras de datos *Sender_Dests* y *Sender_Buffering*

Al haberse modificado el formato de todos los mensajes, también deberá modificarse el código de la estructura de datos *Sender_Dests*. Ahora el campo `Key` será un registro compuesto por los campos `Session_Id`, `EP_H_Creat` y `Seq_N`.

Así, cuando llegue un mensaje *Ack* deberá buscarse en *Sender_Dests* el mensaje que asiente basado en los 3 campos que forman la `Key`.

También hay que añadir el `Session_Id` al campo `Value` en la estructura de datos *Sender_Buffering*.

9.5. Uso de *latest_msgs* para garantizar la entrega ordenada de mensajes de unienvío

La estructura de datos *latest_msgs* puede utilizarse también para implementar la entrega ordenada de los mensajes de unienvío.

Como ha quedado dicho antes, ahora el identificador de sesión forma parte de los campos que identifican un mensaje, junto al endpoint del creador y al número de secuencia. Por ello, tanto en el caso de los mensajes de unienvío como en los enviados por inundación, es necesario incluir el identificador de sesión en la clave de los elementos de *latest_msgs*. Así, en esta estructura de datos el campo `Key` pasará a ser el registro compuesto por el `Session_ID` y el `EP_H_Creat`.

Así, tanto para los mensajes enviados por inundación como para los de unienvío ahora hay que utilizar el identificador de sesión y el endpoint del creador de un mensaje que se reciba para buscar en *latest_msgs* el último número de secuencia visto para ese origen.

Si el mensaje recibido es de unienvío (por ejemplo los de tipo *Reject*), una vez encontrado el número de secuencia almacenado en *latest_msgs* para el origen se deberá comprobar si el mensaje recibido ha llegado o no en orden:

- Si el mensaje recibido tiene un `Seq_N` **menor o igual** que el almacenado en *Latest_Msgs* para ese origen, no hay que procesarlo, pero hay que enviar un mensaje *Ack* al `EP_H_Creat` del mensaje recibido. Esta situación puede ocurrir cuando se recibe una retransmisión de un mensaje previo.
- Si el mensaje recibido tiene un `Seq_N` **inmediantamente consecutivo** al almacenado en *Latest_Msgs* para ese origen, es un mensaje nuevo que llega en orden: hay que asentirlo y procesarlo.
- Si el mensaje recibido tiene un `Seq_N` **mayor en 2 o más unidades** al almacenado en *latest_msgs* para ese origen, es un mensaje que llega desordenado: NO hay que asentirlo NI procesarlo.

En el caso de los mensajes de unienvío no tiene sentido que el creador de un mensaje almacene en su tabla *latest_msgs* el identificador del mensaje creado, ya que dicho mensaje nunca le puede volver a llegar a él reenviado por otro nodo.

9.6. Implementación alternativa

En lugar de almacenar el campo *session_id* en *latest_msgs* puede crearse una nueva tabla de símbolos del tipo de *latest_msgs* para cada una de las sesiones que esté utilizando el nodo.

10. Envío fiable de ficheros (Complejidad alta)

Antes de leer esta parte opcional es preciso haber leído previamente la sección 9. En ella se explica cómo se debe implementar la fiabilidad de los mensajes de unienvío como los que se introducen en esta sección.

Esta parte opcional consiste en implementar el envío fiable de ficheros entre usuarios de Chat-Peer v2.0. En cualquier momento un usuario podrá enviar un fichero a otro utilizando el comando `.send_file`, debiendo para ello conocer el nombre de máquina y el puerto del nodo destino.

En el apéndice A se describe cómo programar el acceso a ficheros que es necesario conocer para realizar la opción de transmisión de ficheros.

En el enunciado de esta parte opcional llamaremos **nodo origen** al nodo del usuario que quiere enviar un fichero, y **nodo destino** al nodo del usuario destinatario del fichero.

10.1. Interfaz de usuario

Para poder recibir ficheros los nodos ahora se arrancarán especificando en la línea de comandos un argumento adicional al principio, por lo que ahora el programa de cada nodo se lanzará pasándole 6, 8 o 10 argumentos en la línea de comandos:

10.1.1. Línea de comandos

```
./chat_peer_2 files_dir port nickname min_delay max_delay fault_pct [[nb_host nb_port] [nb_host nb_port]]
```

Los 6 primeros argumentos son obligatorios. El primero es nuevo:

- `files_dir`: Nombre de la carpeta en la que residen los ficheros que se pueden enviar a otros nodos y en la que se guardarán los ficheros recibidos de otros nodos. Este nombre es relativo a la carpeta desde la que se ejecuta el programa. Ejemplo: si el programa se ejecuta desde la carpeta `/Users/pepe/P5`, y se arranca el programa poniendo en el primer argumento `mis_ficheros`, la carpeta para los ficheros a enviar o recibir será `/Users/pepe/P5/mis_ficheros`.

El resto de argumentos de la línea de comandos son los mismos que los utilizados en la parte básica de la práctica P5.

10.1.2. Interfaz de usuario para enviar un fichero y para aceptar su recepción

Cuando un usuario quiera enviar un fichero a otro escribirá el comando `.send_file` en el teclado. A continuación el nodo origen le preguntará a este usuario por el *EP_H* del nodo destino [1] y por el nombre del fichero que se le quiere enviar.

Ejemplo suponiendo que el usuario *pepe* quiere enviar el fichero `f1` al nodo destino con `EP_H = (localhost, 1111)`:

```
.send_file
   Host Name: localhost
   Port     : 1111
   File Name: f1
```

Al usuario destinatario se le pregunta si acepta recibir un fichero, debiendo responder si lo acepta o lo rechaza a través del teclado.

Ejemplo de recepción en el nodo (localhost, 1111) de la petición de envío del fichero f1 que hace el usuario pepe:

```
pepe quiere enviar el fichero f1. ¿Aceptas recibirlo? (S/N): S
```

Se informará al nodo origen si el usuario destinatario ha aceptado o no la transmisión del fichero:

```
El nodo (localhost, 1111) ha aceptado recibir el fichero f1
```

[1] El *EP_H* es el *End_Point* en el que el nodo destino recibe mensajes mediante *handler* de Lower_Layer_UDP

o bien:

```
El nodo (localhost, 1111) ha rechazado recibir el fichero f1
```

Cuando el fichero ha sido transferido en su totalidad se informa al usuario destinatario:

```
El fichero f1 enviado por pepe está disponible en la carpeta mis_ficheros
```

10.2. Protocolo

Para implementar este protocolo se añaden tres nuevos tipos de mensaje cuyo formato es descrito más adelante. Por ello ahora el enumerado para indicar el tipo de mensaje se redefinirá de la siguiente forma (aparecen en negrita los nuevos tipos de mensaje):

```
type Message_Type is (Init, Reject, Confirm, Writer, Logout, Ack,
                      File_Send_Request, File_Send_Reply, File_Data);
```

A continuación se describe el protocolo que regula la transmisión entre el nodo origen y el nodo destino:

- Cuando un nodo origen quiere transmitir un fichero a un nodo destino comienza enviándole un mensaje ***File_Send_Request*** al *EP_H* del destino, esperando a continuación a recibir mediante `LLU.Receive` en su `EP_R` el mensaje de tipo ***File_Send_Reply*** con el que el nodo destino acepta o rechaza la transmisión del fichero.
- Cuando el mensaje *File_Send_Request* es recibido en el *handler* del nodo destino se le pregunta al usuario destinatario si acepta recibir el fichero o no (ver apartado 10.1.2).
  - Si el usuario destinatario acepta la petición de envío de un fichero:
    1. El nodo destino creará un fichero en la carpeta especificada en el primer argumento de la línea de comandos, nombrándolo con el mismo nombre que el fichero que le quieren enviar.
    2. El nodo destino le enviará un mensaje de tipo *File_Send_Reply* al `EP_R` del nodo origen informándole de que acepta el envío del fichero.
  - Si por el contrario el usuario destinatario no acepta que se le envíe el fichero, el nodo destino enviará un mensaje de tipo *File_Send_Reply* al `EP_R` del nodo origen informándole de que se rechaza la recepción del fichero.
- Cuando el nodo origen recibe el mensaje *File_Send_Reply*:
  - Si el nodo destino acepta la transmisión:
    1. Se informará al usuario del nodo origen que el destinatario ha aceptado la transmisión del fichero
    2. El nodo origen comenzará a leer secuencialmente del disco los bloques de bytes, y los irá enviando al nodo destino en mensajes ***File_Data***. Todos los mensajes *File_Data* contendrán 512 bytes de datos, salvo quizá el último, que podrá contener menos de 512 bytes en el caso de que el tamaño del fichero no sea múltiplo de 512.
    3. El nodo destino escribirá en la posición que corresponda del fichero destino los bloques de bytes que vaya recibiendo en los mensajes de tipo *File_Data*.
  - Si por el contrario el nodo destino no acepta la transmisión del fichero, el nodo origen no enviará ningún mensaje de tipo *File_Data* e informará al usuario de que ha sido rechazada la transmisión del fichero.
- Cuando el nodo destino recibe el mensaje *File_Data* que contiene los últimos bytes de datos del fichero informa al usuario destinatario de que la transferencia del fichero ha concluido.

10.3. Tipos de mensaje

Mensaje *File_Send_Request*

Mensaje enviado por el nodo origen que quiere transmitir un fichero al `EP_H` del nodo destino para solicitarle permiso para transferirle el fichero.

Formato:

File_Send_Request	Session_Id	EP_H_Creat	Seq_N	EP_R	File_Name	Nick

en donde:

- File_Send_Request: Valor del tipo Message_Type que identifica el tipo de mensaje.
- Session_Id: Valor del tipo Positive que identifica la sesión a la que pertenece el mensaje
- EP_H_Creat: *End_Point* para recepción mediante *handlers* de LLU del nodo que crea el mensaje.
- Seq_N: Valor del tipo Seq_N_T con el número de secuencia con el que el emisor numera los mensajes de tipo *File_Send_Request* y *File_Data*.
- EP_R: *End_Point* del nodo que envía el mensaje, en el que espera recibir mediante LLU.Receive mensajes de tipo *File_Send_Reply*
- File_Name: Valor del tipo Unbounded_String con el nombre del fichero que se desea transmitir.
- Nick: Valor del tipo Unbounded_String con el *nick* del nodo que envía el mensaje.

El mensaje de tipo File_Send_Request que envíe el nodo origen para transmitir un fichero f pertenecerá a la misma sesión que los mensajes de tipo File_Data que se envíen para transmitir el fichero f.

Mensaje *File_Data*

El nodo origen envía los bloques de bytes del fichero que quiere transmitir en mensajes de tipo *File_Data*.
Formato:

File_Data	Session_Id	EP_H_Creat	Seq_N	File_Name	Block_Pos	Block_Size	Block_Data

en donde:

- File_Data: Valor del tipo Message_Type que identifica el tipo de mensaje.
- Session_Id: Valor del tipo Positive que identifica la sesión a la que pertenece el mensaje
- EP_H_Creat: *End_Point* para recepción mediante *handlers* de LLU del nodo que envía el mensaje.
- Seq_N: Valor del tipo Seq_N_T con el número de secuencia con el que el emisor numera los mensajes de tipo *File_Send_Request* y *File_Data*
- File_Name: Valor del tipo Unbounded_String con el nombre del fichero al que pertenece el bloque de bytes que se envía en este mensaje.
- Block_Pos: Valor del tipo Positive con la posición que ocupa en el fichero el primer byte del bloque que se envía en este mensaje. El primer byte de un fichero ocupa la posición número 1.
- Block_Size: Valor del tipo Positive con la longitud del bloque del fichero que se envía en este mensaje. Este campo tendrá siempre un valor igual a 512, salvo quizá en el último bloque, en caso de que el tamaño del fichero no sea múltiplo de 512.
- Block_Data: Valor de tipo Stream_Element_Array que contiene los bytes del bloque que ocupa la posición Block_Pos en el fichero File_Name.

El mensaje de tipo File_Send_Request que envíe el nodo origen para transmitir un fichero f pertenecerá a la misma sesión que los mensajes de tipo File_Data que se envíen para transmitir el fichero f.

Mensaje *File_Send_Reply*

Mensaje enviado por un nodo destino tras haber recibido un mensaje *File_Send_Request*. Lo envía el nodo destino dirigido al EP_R del nodo origen para comunicarle si acepta o rechaza la transmisión de un fichero.
Formato:

File_Send_Reply	Session_Id	EP_H_Creat	Seq_N	File_Name	Accept

en donde:

- File_Send_Reply: Valor del tipo Message_Type que identifica el tipo de mensaje.
- Session_Id: Valor del tipo Positive que identifica la sesión a la que pertenece el mensaje

- `EP_H_Creat`: *End_Point* para recepción mediante *handlers* de LLU del nodo que envía el mensaje.
- `Seq_N`: Valor del tipo `Seq_N_T` con el número de secuencia con el que el emisor numera todos los mensajes de tipo *File_Send_Reply* que envía.
- `File_Name`: Valor del tipo `Unbounded_String` con el nombre del fichero cuya transmisión se acepta o se rechaza.
- `Accept`: Valor de tipo `Boolean`. El valor `True` significa que se acepta la transmisión del fichero `File_Name`, y el valor `False` que se rechaza.

El mensaje de tipo `File_Send_Reply` que envíe el nodo destino para responder a la petición de envío de un fichero f por parte de un nodo origen pertenecerá a una sesión distinta a todas las utilizadas hasta entonces por el nodo destino.

10.4. Pautas de implementación

Para depurar el código, 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.

Si se desea ver todos los bytes en que difieren los ficheros, puede incluirse la opción `-l`:

```
cmp -l fich_origen fich_destino
```

11. Mensajes privados (Complejidad alta)

Antes de leer esta parte opcional es preciso haber leído previamente la sección 9. En ella se explica cómo se debe implementar la fiabilidad de los mensajes de unienvío como los que se introducen en esta sección.

Cualquier usuario podrá enviar un mensaje privado a otro usuario del chat con tal de que haya recibido de él ya algún mensaje:

- Un usuario podrá enviar un mensaje privado a otro usuario que conozca del chat, dado su *nick*. Para ello utilizará en la interfaz de usuario la siguiente sintaxis:

  ```
  .priv <nick> <texto del mensaje>
  ```

  Ejemplo:

  ```
  .priv pepe ¿te apetece ir a cenar esta noche?
  ```

- Estas cadenas de texto no se enviarán por inundación mediante mensajes *Writer*, sino mediante mensajes de unienvío de un nuevo tipo: ***Writer_Uni***.
- Para poder implementar esta funcionalidad el nodo deberá utilizar una nueva tabla de símbolos que almacene la correspondencia entre el `nick` y el `EP_H_Creat` de cada nodo. Cada vez que se reciba un mensaje `Init` o `Writer` se añadirá una entradas a esta tabla (si no existiera ya), y cada vez que se reciba un `Logout` se eliminará la entrada del usuario en cuestión.

El alumno deberá definir el formato del mensaje de unienvío *Writer_Uni*, así como el resto del protocolo, incluyendo la definición de nuevos mensajes si los considera necesarios.

Deberá tenerse en cuenta lo definido en la sección 9 para el envío fiable de mensajes de unienvío. En particular, los mensajes *Writer_Uni* que se envíen en cada chat privado han de enviarse en una sesión distinta.

Esta opción es diferente de la opción 12 y puede implementarse sin necesidad de implementarse la otra, o pueden implementarse las dos.

12. Chat privado (Complejidad alta)

Antes de leer esta parte opcional es preciso haber leído previamente la sección 9. En ella se explica cómo se debe implementar la fiabilidad de los mensajes de unienvío como los que se introducen en esta sección.

Cualquier pareja de usuarios deberá poder establecer sesiones de chat privado entre sí:

- Un usuario A deberá poder solicitar a otro usuario B el establecimiento de un chat privado.
- En caso de ser aceptada por el usuario B una petición de chat privado proveniente del usuario A, a partir de ese momento las cadenas de texto que escriba en el teclado el nodo A sólo se deberán recibir en el nodo B y viceversa.
- Estas cadenas de texto no se enviarán por inundación mediante mensajes *Writer*, sino mediante mensajes de unienvío de un nuevo tipo: ***Writer_Uni***.
- Cualquiera de los dos usuarios que están manteniendo un chat privado podrá decidir terminar la sesión de chat privado, debiendo informar al otro nodo de la finalización del chat privado. A partir de ese momento ambos nodos deberán dejar de utilizar los mensajes *Writer_Uni* para enviar el texto leído del teclado, volviendo a utilizar mensajes *Writer* enviados por inundación a todos los nodos.

El alumno deberá definir el formato del mensaje de unienvío *Writer_Uni*, así como el resto del protocolo, incluyendo la definición de nuevos mensajes que considere necesarios y la interfaz de usuario necesaria para implementar esta especificación.

Deberá tenerse en cuenta lo definido en la sección 9 para el envío fiable de mensajes de unienvío. En particular, los mensajes *Writer_Uni* que se envíen en cada chat privado han de enviarse en una sesión distinta.

Esta opción es diferente de la opción 11 y puede implementarse sin necesidad de implementarse la otra, o pueden implementarse las dos.

A. Acceso a ficheros de cualquier tipo en Ada

El paquete `Ada.Streams.Stream_IO` permite el acceso a ficheros independientemente de su contenido, tratándolos, básicamente, como un array de bytes. Este paquete se usará para leer y escribir el fichero que se descarga.

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;

procedure Prueba_Ficheros_1 is
   package S_IO renames Ada.Streams.Stream_IO;

   -- tamaño en bytes de los bloques a leer/escribir
   Tam_Bloque: constant := 512;
   -- bloque de datos del fichero
   Bloque: Ada.Streams.Stream_Element_Array(1..Tam_Bloque);

   -- último byte leído (si no quedaba un bloque entero)
   Ultimo: 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, Ultimo);

      -- 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..Ultimo));
   end loop;

   S_IO.Close(Fichero_Origen);
   S_IO.Close(Fichero_Destino);
end Prueba_Ficheros_1;
```

El procedimiento `Open` abre el fichero cuyo nombre se proporciona en el tercer parámetro y devuelve en el primer parámetro un manejador de fichero que se se utilizará para realizar cualquier operación sobre dicho fichero. El segundo parámetro de `Open` es el modo de acceso a ese fichero:

- `S_IO.In_File`: acceso en modo lectura.
- `S_IO.Out_File`: acceso en modo escritura, destruyendo datos que previamente hubiera en ese fichero.
- `S_IO.Append_File`: acceso en modo escritura, para añadir datos en un fichero que ya existe.

Para utilizar `Open` es necesario que el fichero exista. Para crear un fichero que no existe se utilizará el procedimiento `Create` cuyo último parámetro es el nombre de fichero que queremos crear y devolverá en el primer parámetro el manejador de dicho fichero. El modo de acceso que utilizaremos con la llamada `Create` será `S_IO.Out_File` porque lo que querremos hacer con ese fichero recién creado es introducir nuevos datos.

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);
```

B. Normas y plazos de entrega

El alumno que realice una o más de las partes opcionales de la práctica P5 deberá entregarlas presencialmente en las aulas de prácticas ante un profesor de la asignatura, que la probará y hara al alumno preguntas orales sobre su código.

Las fechas son las mismas que las fechas de entrega de la parte básica de P5.