Programación de Sistemas de Telecomunicación / Informática II
Práctica 3:
Mini-Chat v2.0 en modo cliente/servidor

GSyC

Octubre de 2014

1. Introducción

En esta práctica debes realizar de nuevo dos programas en Ada siguiendo el modelo cliente/servidor que ofrezcan un servicio de chat entre usuarios.

Mini-Chat v2.0 mejora la implementación y la interfaz de uso de la anterior versión de Mini-Chat. El uso de los manejadores o *handlers* de Lower_Layer_UDP para recibir mensajes permitirá que un mismo cliente pueda realizar simultáneamente las funciones de lector y de escritor definidas en la anterior práctica.

En una sesión de chat participará un programa servidor y varios programas cliente:

- Los clientes leen de la entrada estándar cadenas de caracteres y las envían al servidor. Simultáneamente, los clientes van recibiendo las cadenas de caracteres que les envía el servidor procedentes de otros clientes.
- El servidor recibe los mensajes con cadenas de caracteres que le envían los clientes y las reenvía al resto de los clientes.

Mini-Chat v2.0 tendrá además la siguiente funcionalidad:

- El servidor se asegurará de que los apodos (*nicknames*) de los clientes no se repiten, rechazando a un cliente que pretenda utilizar un apodo que ya esté siendo usado por otro cliente, para lo cuál enviará un mensaje al cliente informándole de que ha sido rechazado.
- El mandato `.quit` ejecutado por un cliente provocará que se elimine en el servidor la información asociada a dicho cliente, para lo cuál el cliente enviará un mensaje al servidor.
- El servidor tendrá establecido un número máximo de clientes. Si en un momento dado quisiera entrar en el chat un cliente cuando ya se ha alcanzado el máximo de clientes, el servidor elegirá al cliente que lleva más tiempo sin escribir en el chat, y lo expulsará para dejar sitio al nuevo cliente.

2. Descripción del programa cliente `chat_client_2.adb`

2.1. Interfaz de usuario

El programa cliente se lanzará pasándole 3 argumentos en la línea de comandos:

- Nombre de la máquina en la que está el servidor
- Número del puerto en el que escucha el servidor
- *Nickname* (apodo) del cliente del chat. Cualquier cadena de caracteres distinta de la cadena `server` será un apodo válido siempre que no exista ya otro cliente conocido por el servidor que haya usado el mismo apodo. El apodo `server` lo utiliza el servidor para enviar cadenas de caracteres a los clientes informando de la entrada de un nuevo cliente en el chat, de la salida voluntaria del chat de un cliente, o de la expulsión de un cliente.

El cliente mostrará en pantalla una primera línea en la que informará de si ha sido aceptado o no por el servidor. Si no es aceptado el programa cliente termina.

En caso de ser aceptado el cliente irá pidiendo al usuario cadenas de caracteres por la entrada estándar, y se las enviará al servidor. Si la cadena de caracteres leída de la entrada estándar es `.quit`, el cliente terminará su ejecución tras enviar un mensaje al servidor informándole de su salida.

El cliente irá recibiendo del servidor las cadenas de caracteres enviadas por otros clientes del Mini-Chat v2.0, y las mostrará en pantalla.

En los siguientes cinco recuadros se muestra la salida de clientes que se arrancan en terminales distintos en el orden en el que aparecen los recuadros en el texto. Se supondrá que el servidor de Mini-Chat v2.0 que están usando los clientes está atado al puerto 9001 en la máquina zeta12, y que no hay atado ningún proceso al puerto 10001 en la máquina zeta20:

Primer cliente que se arranca:

```
$ ./chat_client_2 zeta12 9001 carlos
Mini-Chat v2.0: Welcome carlos
>>
server: ana joins the chat
>>
server: pablo joins the chat
>>
ana: entro
>> Hola
>> quién está ahí?
>>
ana: estoy yo, soy ana
>> ana dime algo
>>
ana: hola carlos
>> adios
>> .quit
$
```

Segundo cliente que se arranca:

```
$ ./chat_client_2 zeta12 9001 ana
Mini-Chat v2.0: Welcome ana
>>
server: pablo joins the chat
>> entro
>>
carlos: Hola
>>
carlos: quién está ahí?
>> estoy yo, soy ana
>>
carlos: ana dime algo
>> hola carlos
>>
carlos: adios
>>
server: carlos leaves the chat
>> hasta luego chico, ¡vaya modales! Yo también me voy.
>> .quit
$
```

Tercer cliente que se arranca:

```
$ ./chat_client_2 zeta12 9001 pablo
Mini-Chat v2.0: Welcome pablo
>>
ana: entro
>>
carlos: Hola
>>
carlos: quién está ahí?
>>
ana: estoy yo, soy ana
>>
carlos: ana dime algo
>>
ana: hola carlos
>>
carlos: adios
>>
server: carlos leaves the chat
>>
ana: hasta luego chico, ¡vaya modales! Yo también me voy.
>>
server: ana leaves the chat
>>
```

Cuarto cliente que se arranca:

```
$ ./chat_client_2 zeta12 9001 pablo
Mini-Chat v2.0: IGNORED new user pablo, nick already used
$
```

Quinto cliente que se arranca:

```
$ ./chat_client_2 zeta20 10001 pepe
Server unreachable
$
```

2.2. Implementación

Al arrancar el cliente deberá enviar un mensaje `Init` al servidor de Mini-Chat v2.0 con su *nickname* para solicitar ser aceptado como nuevo cliente.

El servidor le enviará como respuesta un mensaje `Welcome`, informándole de si ha sido aceptado o no. Además, en caso de que lo acepte, el servidor enviará un mensaje `Server` al resto de clientes informándoles de la entrada de un nuevo usuario.

Un cliente será rechazado por pretender utilizar un apodo que ya está usándose por otro cliente del chat.

El cliente deberá atarse a dos `End_Point` distintos:

- `Client_EP_Receive`, en el que se recibirán los mensajes `Welcome`. Hasta que no reciba el mensaje `Welcome` no puede continuar, por lo que se deberá utilizar `LLU.Receive` para recibir en este End_Point.
- `Client_EP_Handler`, para recibir los mensajes de servidor. Estos mensajes hay que recibirlos a la vez que se van leyendo del teclado cadenas de caracteres y se van enviando al servidor, por lo que se deberá utilizar un *handler* de `Lower_Layer_UDP` para recibir en este End_Point.

Tras enviar un cliente su mensaje `Init`, pueden presentarse las siguientes situaciones:

- Si transcurridos 10 segundos el cliente no ha recibido el mensaje `Welcome` del servidor, el cliente terminará, mostrando en pantalla el mensaje `Server unreachable`. Puede verse un ejemplo en el último recuadro de la sección anterior.
- Si el cliente recibe el mensaje `Welcome` indicándole que es rechazado terminará su ejecución, mostrando en pantalla un mensaje explicativo. Puede verse un ejemplo en el penúltimo recuadro de la sección anterior.
- Si el cliente recibe el mensaje `Welcome` indicándole que es aceptado:
  - El cliente entrará en un bucle en el que pedirá al usuario cadenas de caracteres que le irá enviando al servidor mediante mensajes `Writer`. Si la cadena de caracteres leída es `.quit` el cliente envía un mensaje `Logout` al servidor y a continuación termina su ejecución.
  - Simultáneamente, el cliente irá recibiendo mensajes `Server` en el `Client_EP_Handler`, utilizando la recepción mediante *handler* (manejador) de `Lower_Layer_UDP`, debiendo el cliente mostrar su contenido en pantalla.

    El mensaje `Server` recibido en un cliente puede contener mensajes de texto de otros clientes que han sido recibidos por el servidor, pero también puede contener mensajes generados por el propio servidor para informar de que un cliente nuevo ha entrado, de que un cliente ha abandonado el chat, o de que el cliente que lo recibe que ha sido expulsado.

    Cuando un cliente es expulsado es responsabilidad del usuario del cliente el salir del programa cliente. En cualquier caso, salga o no el usuario después de haber recibido el mensaje de expulsión del servidor, a partir de ese instante el servidor no le reenviará ningún mensaje, ni reenviará los mensajes que le envíe el cliente expulsado.

En el apartado 4 se explica el formato de los mensajes mencionados.

3. Descripción del programa servidor `chat_server_2.adb`

3.1. Interfaz de usuario

El programa servidor se lanzará pasándole 2 argumentos en la línea de comandos:

- Número del puerto en el que escucha el servidor
- Número máximo de clientes que acepta el servidor, que será siempre un número comprendido entre 2 y 50.

El servidor nunca terminará su ejecución, e irá mostrando en pantalla los mensajes que vaya recibiendo para permitir comprobar su funcionamiento. Puede verse un ejemplo de la salida que muestra el servidor en el siguiente recuadro:

```
$ ./chat_server_2 9001 5
INIT received from carlos: ACCEPTED
INIT received from ana: ACCEPTED
INIT received form pablo: ACCEPTED
INIT received form pablo: IGNORED. nick already used
WRITER received from ana: entro
WRITER received from carlos: Hola
WRITER received from carlos: quién está ahí?
WRITER received from ana: estoy yo, soy ana
WRITER received from unknown client. IGNORED
WRITER received from carlos: ana dime algo
WRITER received from ana: hola carlos
WRITER received from carlos: adios
LOGOUT received from carlos
WRITER received from ana: hasta luego chico, ¡vaya modales! Yo también me voy.
LOGOUT received from ana
```

3.2. Implementación

El servidor deberá guardar en una estructura de datos la información relativa a los clientes. Se añadirán clientes cuando se reciban mensajes `Init`. Se eliminarán clientes cuando se reciban mensajes `Logout` o cuando el servidor decida expulsar a un cliente. De cada cliente el servidor deberá almacenar, al menos, su `Client_EP_Handler`, su *nickname* y la hora a la que envió su último mensaje `Writer`.

El servidor debe atarse a un `End_Point` formado con la dirección IP de la máquina en la que se ejecuta, y el puerto que le pasan como argumento.

Una vez atado, entrará en un bucle infinito recibiendo mensajes de clientes:

- Cuando el servidor reciba un mensaje `Init` de un cliente, lo añadirá si el *nick* no está siendo ya utilizado por otro cliente, guardando la hora en la que se recibió el mensaje `Init`. A continuación le enviará un mensaje `Welcome` al `Client_EP_Receive` del cliente, informándole de si ha sido aceptado o rechazado.

  Si no hubiera huecos libres para almacenar la información del nuevo cliente, el servidor generará un hueco eliminando la entrada correspondiente al cliente que hace más tiempo que envió su último mensaje `Writer`. Si un cliente no ha enviado ningún mensaje `Writer` se considerará la hora a la que envió su mensaje de inicio para decidir si se le expulsa. El servidor enviará un mensaje `Server` a todos los clientes, incluyendo al que se expulsa, informándoles de que el cliente ha sido expulsado del chat. Este mensaje llevará en el campo *nickname* la cadena `server`, y en el campo `comentario` la cadena `<nickname> banned for being idle too long`, siendo `<nickname>` el apodo del cliente expulsado.

  Si el cliente es aceptado, el servidor enviará un **mensaje de servidor** a todos los clientes, salvo al que ha sido aceptado, informándoles de que un nuevo cliente ha sido aceptado en el chat. Este mensaje llevará en el campo *nickname* la cadena `server`, y en el campo `comentario` la cadena `<nickname> joins the chat`, siendo `<nickname>` el apodo del cliente que ha sido aceptado.

- Cuando el servidor reciba un mensaje `Writer` de un cliente, buscará su *nick* entre los clientes que conoce, y si lo encuentra enviará un mensaje `Server` al `Client_EP_Handler` de todos los clientes conocidos, salvo al que le ha enviado el mensaje.

- Cuando el servidor reciba un mensaje `Logout` de un cliente eliminará la información que guardaba de ese cliente y enviará un mensaje `Server` al resto de los clientes, informándoles de que el cliente ha abandonado el chat. Este mensaje llevará en el campo *nickname* la cadena `server` y en el campo comentario la cadena `<nickname> leaves the chat`, siendo `<nickname>` el apodo del cliente que abandona el chat.

En el apartado 4 se explica el formato de los mensajes.

4. Formato de los mensajes

Los cinco tipos de mensajes que se necesitan para esta práctica se distinguen por el primer campo, que podrá adoptar los valores del siguiente tipo enumerado:

```
type Message_Type is (Init, Welcome, Writer, Server, Logout);
```

Dicho tipo deberá estar declarado en el fichero `chat_messages.ads` y desde ahí ser usado tanto por el cliente como por el servidor. Así el código de cliente o del servidor tendrá el siguiente aspecto:

```
with Chat_Messages;
...
procedure ... is
   package CM renames Chat_Messages;
   use type CM.Message_Type;
   ...

   Mess: CM.Message_Type;

begin
   ...
   Mess := CM.Init;
   ...
   if Mess = CM.Server then
      ...

end ...;
```

Si el paquete `Chat_Messages` no contiene ningún procedimiento no es necesario que tenga cuerpo (fichero `.adb`), sino que sólo tendrá especificación (fichero `.ads`).

Mensaje `Init`

Es el que envía un cliente al servidor al arrancar. Formato:

`Init`	`Client_EP_Receive`	`Client_EP_Handler`	`Nick`

en donde:

- `Init`: `Message_Type` que identifica el tipo de mensaje.
- `Client_EP_Receive`: `End_Point` del cliente que envía el mensaje. El cliente recibe mensajes `Welcome` en este `End_Point`.
- `Client_EP_Handler`: `End_Point` del cliente que envía el mensaje. El cliente recibe mensajes `Server` en este `End_Point`.
- `Nick`: `Unbounded_String` con el *nick* del cliente.

Mensaje `Welcome`

Es el que envía un servidor a un cliente tras recibir un mensaje `Init` para indicar al cliente si ha sido aceptado o rechazado en Mini-Chat v2.0. Formato:

`Welcome`	`Acogido`

en donde:

- `Welcome`: `Message_Type` que identifica el tipo de mensaje.
- `Acogido`: `Boolean` que adoptará el valor `False` si el cliente ha sido rechazado y `True` si el cliente ha sido aceptado.

Mensaje `Writer`

Es el que envía un cliente al servidor con una cadena de caracteres introducida por el usuario. Formato:

`Writer`	`Client_EP_Handler`	`Comentario`

en donde:

- `Writer`: `Message_Type` que identifica el tipo de mensaje.
- `Client_EP_Handler`: `End_Point` del cliente que envía el mensaje. El cliente recibe mensajes `Server` en este `End_Point`.
- `Comentario`: `Unbounded_String` con la cadena de caracteres introducida por el usuario

Nótese que el *nick* no viaja en estos mensajes, el cliente queda identificado por su `Client_EP_Handler`, y el *nick* deberá ser encontrado por el servidor entre la información que guarda de los clientes, para así poder componer el mensaje `Server` que reenviará a los clientes.

Mensaje `Server`

Es el que envía un servidor a un cliente con el comentario que le llegó en un mensaje `Writer`. Formato:

`Server`	`Nick`	`Comentario`

en donde:

- `Server`: `Message_Type` que identifica el tipo de mensaje.
- `Nick`: `Unbounded_String` con el *nick* del cliente que envió el comentario al servidor, o con el *nick* `server` si es un mensaje generado por el propio servidor para informar a los clientes de la entrada al chat de un nuevo cliente, del abandono de un cliente, o de la expulsión de un cliente.
- `Comentario`: `Unbounded_String` con la cadena de caracteres introducida por un usuario, o la cadena de caracteres generada por el servidor en el caso de los mensajes `Server` con *nickname* `server`.

Mensaje `Logout`

Es el que envía un cliente al servidor para informarle de que abandona el Mini-Chat v2.0. Formato:

`Logout`	`Client_EP_Handler`

en donde:

- `Logout`: `Message_Type` que identifica el tipo de mensaje.
- `Client_EP_Handler`: `End_Point` del cliente.

5. Utilización de plazos y tiempos en Ada

El paquete `Ada.Calendar` ofrece el tipo `Time` para trabajar con instantes de tiempo.

La función `Ada.Calendar.Clock`, sin parámetros, devuelve la fecha y la hora actual en un valor del tipo `Ada.Calendar.Time`.

A un valor de tipo `Ada.Calendar.Time` se le puede sumar o restar una cantidad de tiempo expresada como un valor en segundos (con decimales) de tipo `Duration`, devolviendo un nuevo valor de tipo `Time`.

También se pueden restar dos `Time`, obteniéndose un valor que representa la diferencia entre ambos instantes de tiempo como un valor de tipo `Duration`.

El siguiente programa muestra el uso de `Clock` y del tipo `Time`.

```
with Ada.Text_IO;
with Ada.Calendar;

procedure Plazo is

   use type Ada.Calendar.Time;

   Plazo: constant Duration := 3.0;
   Intervalo: constant Duration := 0.2;
   Hora_Inicio, Hora_Fin: Ada.Calendar.Time;
   Hora_Actual, Hora_Anterior: Ada.Calendar.Time;

begin

   Hora_Inicio := Ada.Calendar.Clock;
   Hora_Fin := Hora_Inicio + Plazo;

   Hora_Anterior := Ada.Calendar.Clock;
   Hora_Actual := Ada.Calendar.Clock;

   while Hora_Actual < Hora_Fin loop
       if Hora_Actual - Hora_Anterior > Intervalo then
          Ada.Text_IO.Put_Line ("Han pasado ya: " &
                                Duration'Image(Hora_Actual - Hora_Inicio) &
                                " segundos");
          Hora_Anterior := Hora_Actual;
       end if;
       Hora_Actual := Ada.Calendar.Clock;
   end loop;
   Ada.Text_Io.Put_Line("Fin.");

end Plazo;
```

6. Condiciones de Funcionamiento

1. El chat debe funcionar limitando el número de clientes al especificado en el segundo argumento de la línea de comandos con la que se arranca el servidor, que podrá adoptar un valor comprendido entre 2 y 50. Cuando se haya alcanzado el número máximo de clientes el servidor debe expulsar a uno antes de aceptar al nuevo que intenta ser acogido.

2. La estructura de datos en la que se almacenen los datos de los usuarios en el servidor deberá implementarse como un **Tipo Abstracto de Datos** en un paquete aparte de nombre `Client_Lists`.

   La especificación de este paquete se proporciona a continuación, siendo la semántica de cada procedimiento o función la misma que la de la práctica anterior. Esta especificación **sólo puede modificarse en su parte private** (y en la lista de paquetes incluidos en cláusulas `with`:

```
with Ada.Strings.Unbounded;
with Lower_Layer_UDP;

package Client_Lists is
   package ASU renames Ada.Strings.Unbounded;
   package LLU renames Lower_Layer_UDP;

   type Client_List_Type is private;

   Client_List_Error: exception;

   procedure Add_Client (List: in out Client_List_Type;
                         EP: in LLU.End_Point_Type;
                         Nick: in ASU.Unbounded_String);

   procedure Delete_Client (List: in out Client_List_Type;
                            Nick: in ASU.Unbounded_String);

   function Search_Client (List: in Client_List_Type;
                           EP: in LLU.End_Point_Type)
                          return ASU.Unbounded_String;

   procedure Send_To_All (List: in Client_List_Type;
                          P_Buffer: access LLU.Buffer_Type;
                          EP_Not_Send: in LLU.End_Point_Type);

   function List_Image (List: in Client_List_Type) return String;

   procedure Update_Client (List: in out Client_List_Type;
                            EP: in LLU.End_Point_Type);

   procedure Remove_Oldest (List: in out Client_List_Type);
                            EP: out LLU.End_Point_Type;
                            Nick: out ASU.Unbounded_String);

   function Count (List: in Client_List_Type) return Natural;


private
...
end Client_Lists;
```

El comportamiento esperado de los subprogramas es el mismo que el especificado en el enunciado de la práctica anterior, con las siguientes excepciones:

- El *nick* `reader` ya no es especial, sino un *nick* más, por lo que no puede estar repetido dentro de la lista.
- El antiguo subprograma `Send_To_Readers` ahora se llama `Send_To_All` y se encarga de enviar el mensaje contenido en el buffer apuntado por `P_Buffer` **a todos los clientes del chat excepto al que tiene como EP el pasado como tercer parámetro** (que será el que envió el mensaje `Writer`).
- El procedimiento `Update_Client` actualizará en la lista la hora almacenada para el cliente cuyo `EP` recibe como parámetro. Si no encuentra en la lista ese `EP`, elevará la excepción `Client_List_Error`.
- El procedimiento `Remove_Oldest` borrará del la lista al cliente que tenga almacenada la hora más antigua, y devolverá su EP y su *nick*. Si la lista está vacía, elevará la excepción `Client_List_Error`.
- La función `Count` devolverá el número de clientes actuales en la lista.

El TAD `Client_List_Type` deberá implementarse de dos maneras distintas: una de las implementaciones deberá utilizar un array y la otra una lista dinámica con punteros. En cada posición de la lista deberá almacenarse un campo con la hora de último acceso al chat de cada cliente.

El programa `chat_server_2` deberá poder compilarse indistintamente con una u otra implementación. Para ello deberán crearse dos subcarpetas dentro de la carpeta en que esté contenida la práctica:

a) carpeta `client_lists_array`, que contendrá la especificación y el cuerpo del paquete `Client_Lists` implementado utilizando un array

b) carpeta `client_lists_dyn`, que contendrá la especificación y el cuerpo del paquete `Client_Lists` implementado utilizando una lista dinámica.

Nótese que en ambos casos el nombre del paquete será el mismo: `Client_Lists`, por lo que en ambas carpetas deberá haber 2 ficheros: `client_lists.ads` y `client_lists.adb`.

Para probar la implementación basada en un array:

- se compilará `chat_server_2` con el siguiente comando:
  `gnatmake -I./client_lists_array -I/usr/local/ll/lib chat_server_2.adb`

Para probar la implementación basada en una lista dinámica:

- se compilará `chat_server_2` con el siguiente comando:
  `gnatmake -I./client_lists_dyn -I/usr/local/ll/lib chat_server_2.adb`

3. Se supondrá que nunca se perderán los mensajes enviados por el servidor ni por los clientes.
4. Deberán ser compatibles las implementaciones de clientes y servidores de los alumnos, para lo cuál es imprescindible respetar el formato de los mensajes.
5. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.

7. Parte opcional: tipo privado limitado

Se deberán extender las dos implementaciones del TAD `Client_List_Type` de forma que el tipo sea privado limitado, debiéndose proporcionar en la interfaz del tipo un procedimiento `Copy` y una función `Compare`.

`Copy` recibe dos argumentos de tipo `Client_List_Type`, copiando el primero en el segundo. `Compare` recibe dos argumentos de tipo `Client_List_Type`, devolviendo el booleano True si son iguales y False en caso contrario.

Evidentemente, para realizar esta parte opcional se permite modificar la parte pública de la especificación del paquete `Client_Lists` para poder exportar el procedimiento `Copy` y la función `Compare`.

Se deberá crear un programa principal `Limited_Private_Type_Test` en el fichero `limited_private_type_test.adb` que demuestre el funcionamiento correcto de `Copy` y de `Compare`.

8. Normas de entrega

8.1. Alumnos de Programación de Sistemas de Telecomunicación

Debes dejar el código fuente de tu programa dentro de tu cuenta en los laboratorios de prácticas. Directamente en el directorio de tu cuenta debes crear una carpeta de nombre `PST.2014`, y dentro de ella una carpeta de nombre `P3`. En ella deben estar todos los ficheros correspondientes a los dos programas (`chat_client_2.adb`, `chat_server_2.adb`) y las 2 subcarpetas `client_lists_array` y `client_lists_dyn` que contienen las dos implementaciones del TAD. Si hubieras implementado otros paquetes auxiliares deberás incluir sus ficheros, de forma que se puedan compilar los programas `chat_client_2` y `chat_server_2`.
Importante: Pon un comentario con tu nombre completo en la primera línea de cada fichero fuente.

8.2. Alumnos de Informática II

Debes dejar el código fuente de tu programa dentro de tu cuenta en los laboratorios de prácticas. Directamente en el directorio de tu cuenta debes crear una carpeta de nombre `I2.2014`, y dentro de ella una carpeta de nombre `P3`. En ella deben estar todos los ficheros correspondientes a los dos programas (`chat_client_2.adb`, `chat_server_2.adb`) y las 2 subcarpetas `client_lists_array` y `client_lists_dyn` que contienen las dos implementaciones del TAD. Si hubieras implementado otros paquetes auxiliares deberás incluir sus ficheros, de forma que se puedan compilar los programas `chat_client_2` y `chat_server_2`.
Importante: Pon un comentario con tu nombre completo en la primera línea de cada fichero fuente.

9. Plazo de entrega

Los ficheros deben estar en tu cuenta, en la carpeta especificada más arriba, antes de las 23:59h del **miércoles 5 de noviembre de 2014**.