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

GSyC

Noviembre de 2019

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. La programación asíncrona mediante el uso de los manejadores o *handlers* de Lower_Layer_UDP para recibir mensajes permitirá que el programa cliente pueda realizar concurrentemente las funciones de lector y de escritor definidas en la anterior práctica, sin necesidad de arrancar un programa cliente en modo escritor para leer de la entrada estándar (teclado) y enviar mensajes al servidor, y otro programa cliente en modo lector para recibir del servidor y mostrarlos en la salida estándar (pantalla).

En una sesión de chat participará un programa servidor y varios programas cliente, uno por cada usuario:

- 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, utilizando recepción asíncrona mediante manejadores o *handlers*.
- 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 cual enviará un mensaje al cliente informándole de que ha sido rechazado.
- El mandato `.quit` ejecutado por un cliente provocará que se desconecte el cliente del servidor.
- El servidor tendrá establecido un número máximo de clientes activos. 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.
- El servidor guardará información sobre los clientes que ya no están activos, ya sea porque se han desconectado utilizando el comando `.quit`, o porque han sido expulsados.

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 enlazarse 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 de forma síncrona.
- `Client_EP_Handler`, para recibir los mensajes `Server` del 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 de manera asíncrona.

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` (ATENCIÓN: el formato del mensaje Writer ha cambiado respecto a la práctica anterior). 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 asíncrona 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 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.

El servidor mostrará la información de los clientes activos cuando se pulse la tecla `l` o la tecla `L`, mostrando para cada uno de ellos su *nick*, su *Client_EP_Handler* y la hora de última actualización.

El servidor mostrará la información de antiguos clientes que ya no están activos cuando se pulse la tecla `o` o la tecla `O`.

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
l
```

```
ACTIVE CLIENTS
==============
carlos (127.0.1.1:13311): 02-Nov-15 20:01:07.236
ana (127.0.1.1:10313): 02-Nov-15 20:00:10.134
pablo (127.0.1.1:23025): 02-Nov-15 20:00:01.268

LOGOUT received from carlos
o
OLD CLIENTS
===========
carlos: 02-Nov-15 20:05:00.211

l
ACTIVE CLIENTS
==============
ana (127.0.1.1:10313): 02-Nov-15 20:00:10.134
pablo (127.0.1.1:23025): 02-Nov-15 20:00:01.268

WRITER received from ana: hasta luego chico, ¡vaya modales! Yo también me voy.
LOGOUT received from ana
o
OLD CLIENTS
===========
carlos: 02-Nov-15 20:05:00.211
ana: 02-Nov-15 20:07:50.433
```

3.2. Implementación

El servidor deberá guardar en una tabla de símbolos la información relativa a los **clientes activos**. Se añadirán clientes cuando se reciban mensajes `Init`. De cada cliente activo el servidor deberá almacenar en la tabla de símbolos de clientes activos, al menos, su `Client_EP_Handler`, su *nickname* y la hora a la que envió su último mensaje `Writer`.

Un cliente deja de estar activo en el servidor de chat cuando se recibe un mensaje `Logout` de ese cliente, o cuando el servidor decide expulsarlo. El servidor deberá guardar en otra tabla de símbolos la información relativa a **clientes antiguos que ya no están activos**. De cada cliente no activo el servidor deberá almacenar solamente su nickname y la hora a la que abandonó el chat dicho cliente (porque hizo *logout* o porque fue expulsado).

El servidor debe enlazarse 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 el *nick* entre los de los clientes activos, y si lo encuentra, comprobará que el `Client_EP_Handler` coincide con el enviado por el cliente en el mensaje. Si es así, 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 buscará el *nick* entre los de los clientes activos, y si lo encuentra, comprobará que el `Client_EP_Handler` coincide con el enviado por el cliente en el mensaje. Si es así, 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.
- La función `Ada.Text_IO.Get_Immediate` puede utilizarse para leer de la entrada estándar un carácter.

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`	`Nick`	`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`.
- `Nick`: `Unbounded_String` con el *nick* del cliente que envía el comentario al servidor.
- `Comentario`: `Unbounded_String` con la cadena de caracteres introducida por el usuario

Nótese que el *nick* SÍ viaja en estos mensajes. El servidor, al recibir este mensaje, deberá comprobar que el *nick* y `Client_EP_Handle` que vienen en el mensaje se corresponden con los almacenados en la colección de clientes activos.

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`	`Nick`

en donde:

- `Logout`: `Message_Type` que identifica el tipo de mensaje.
- `Client_EP_Handler`: `End_Point` del cliente.
- `Nick`: `Unbounded_String` con el *nick* del cliente que envía el `Logout` al servidor.

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`, incluyendo la forma de mostrar en pantalla valores de ambos tipos.

```
with Ada.Text_IO;
with Ada.Calendar;
with Gnat.Calendar.Time_IO;


procedure Plazo is

   use type Ada.Calendar.Time;


   function Time_Image (T: Ada.Calendar.Time) return String is
   begin
      return Gnat.Calendar.Time_IO.Image(T, "%d-%b-%y %T.%i");
   end Time_Image;


   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;

   Ada.Text_IO.Put_Line("Hora de inicio: " & Time_Image(Hora_Inicio));
   Ada.Text_IO.Put_Line("Hora de fin:    " & Time_Image(Hora_Fin));


   while Hora_Actual < Hora_Fin loop
       if Hora_Actual - Hora_Anterior > Intervalo then
          Ada.Text_IO.Put ("Hora actual:    " & Time_Image(Hora_Actual));
          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("Saliendo:       " & Time_Image(Hora_Actual));

end Plazo;
```

6. Condiciones de Funcionamiento

1. El chat debe funcionar limitando el número de clientes activos 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 de ellos antes de aceptar al nuevo que intenta ser acogido.

2. El número de elementos máximo que puede almacenarse en la estructura de datos en la que se almacenan los clientes antiguos que ya no están activos será igual a 150.

3. Tanto la estructura de datos en la que se almacenen los datos de los usuarios activos como la que almacena los usuarios antiguos que ya no están activos en el servidor deberán implementarse mediante la instanciación de una tabla de símbolos genérica. Se debe utilizar el paquete genérico `Maps_G` estudiado en clase de teoría. Se podrán añadir nuevos argumentos del paquete genérico (por ejemplo para especificar el número máximo de elementos de forma que `Put` eleve la excepción `Full_Map` si se ha alcanzado el máximo número de elementos), modificar la parte privada y el cuerpo (*body*). Pero no podrá modificarse la interfaz de la tabla de símbolos que ofrece el paquete, que forzosamente ha de ser la que aparece en `maps_g.ads`. A continuación se muestra en rojo la parte de la especificación que NO PUEDE MODIFICARSE:

```
with Ada.Text_IO;
generic
   type Key_Type is private;
   type Value_Type is private;
   with function "=" (K1, K2: Key_Type) return Boolean;
package Maps_G is

   type Map is limited private;

   procedure Get (M       : Map;
                  Key     : in  Key_Type;
                  Value   : out Value_Type;
                  Success : out Boolean);


   Full_Map : exception;
   procedure Put (M     : in out Map;
                  Key   : Key_Type;
                  Value : Value_Type);

   procedure Delete (M       : in out Map;
                     Key     : in  Key_Type;
                     Success : out Boolean);

   function Map_Length (M : Map) return Natural;

   --
   -- Cursor Interface for iterating over Map elements
   --
   type Cursor is limited private;
   function First (M: Map) return Cursor;
   procedure Next (C: in out Cursor);
   function Has_Element (C: Cursor) return Boolean;
   type Element_Type is record
     Key:   Key_Type;
     Value: Value_Type;
   end record;
   No_Element: exception;

   -- Raises No_Element if Has_Element(C) = False;
   function Element (C: Cursor) return Element_Type;

private

   type Cell;
   type Cell_A is access Cell;
   type Cell is record
      Key   : Key_Type;
      Value : Value_Type;
      Next  : Cell_A;
```

```
   end record;

   type Map is record
      P_First : Cell_A;
      Length  : Natural := 0;
   end record;

   type Cursor is record
      M         : Map;
      Element_A : Cell_A;
   end record;

end Maps_G;
```

4. Diferencias respecto al enunciado de la práctica anterior:

   - El *nick* `reader` ya no tiene un significado especial.
   - Para enviar a todos los clientes un mensaje (salvo al que lo ha generado) hay que iterar sobre todos los elementos de la colección utilizando un Cursor.
   - El formato del mensaje `Writer` ha cambiado: ahora incluye el *nick*.

5. Además de la implementación del paquete `Maps_G` estudiado en la clase de teoría, deberá realizarse una implementación adicional de dicho paquete que en lugar de utilizar punteros utilice un array no ordenado. La especificación de este paquete deberá ser idéntica a la de la versión implementada con punteros, salvo en su parte *private*. Dicha parte *private* deberá contener, al menos, las siguientes declaraciones:

```
  private
      type Cell is record
         Key   : Key_Type;
         Value : Value_Type;
         Full  : Boolean := False;
      end record;

      type Cell_Array is array (...) of Cell;

      type Cell_Array_A is access Cell_Array;

      type Map is record
         P_Array: Cell_Array_A;
         Length  : Natural := 0;
      end record;

      type Cursor is ...
```

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

   a) carpeta `maps_g_array`, que contendrá la especificación y el cuerpo del paquete `Maps_G` implementado utilizando un array no ordenado.

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

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

   Para probar la implementación basada en un array:

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

6. Se supondrá que nunca se perderán los mensajes enviados por el servidor ni por los clientes.
7. Deberán ser compatibles las implementaciones de clientes y servidores de los alumnos, para lo cuál es imprescindible respetar el formato de los mensajes.
8. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.
9. Se recomienda crear nuevos paquetes para organizar el código.

7. Entrega

El límite para la entrega de esta práctica es: **Miércoles, 4 de Diciembre a las 12:50h**

La **Prueba de Laboratorio 3** correspondiente a esta práctica se realizará en el aula de prácticas habitual el **Miércoles, 4 de Diciembre a las 13:00h**.