Programación de Sistemas de Telecomunicación / Informática II
Práctica 2:
Mini-Chat en modo cliente/servidor

GSyC

Octubre de 2017

1. Introducción

En esta práctica debes realizar dos programas en Ada que permitan implementar un sencillo sistema de chat entre usuarios, siguiendo el modelo cliente/servidor.

El programa cliente podrá comportarse de dos maneras: en modo escritor, o en modo lector. Si un cliente es escritor podrá enviar mensajes al servidor del chat. Si un cliente es lector podrá recibir los mensajes que envíe el servidor.

El servidor se encargará de recibir los mensajes procedentes de los clientes escritores, y reenviárselos a los clientes lectores.

En una sesión de chat participará un programa servidor y varios programas clientes (lectores y escritores). Cada usuario del chat tiene que arrancar 2 programas cliente: uno escritor, que lee del teclado y envía mensajes, y uno lector, que recibe mensajes del servidor y los muestra en la pantalla.

Como extensión de la práctica podrás realizar un tercer programa en Ada que opere como administrador del chat.

2. Descripción del programa cliente `chat_client.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. Si el *nick* es `reader`, el cliente funcionará en modo lector. Con cualquier otro *nick* distinto a `reader` el cliente funcionará en modo escritor.

Una vez lanzado:

- Si el cliente se lanza en modo **escritor**, pedirá al usuario cadenas de caracteres, y se las irá enviando al servidor. Los clientes lanzados en modo escritor no reciben mensajes del servidor. El programa terminará cuando el usuario introduzca la cadena `.quit`
- Si el cliente se lanza en modo **lector**, esperará a recibir mensajes del servidor (conteniendo las cadenas enviadas por los clientes escritores del chat), y los mostrará en pantalla. En este modo el cliente nunca terminará su ejecución.

MUY IMPORTANTE: Si un cliente escritor intenta entrar al chat utilizando un *nick* que ya está siendo usado por otro cliente escritor que entró anteriormente, todos sus mensajes serán ignorados por el servidor.

Ejemplo de ejecución: Suponiendo que se ha lanzado previamente un programa servidor, se muestra a continuación cómo se lanza un cliente lector y varios clientes escritores, cada cliente en una ventana de terminal diferente. El lector es el primer cliente en lanzarse.

```
$ ./chat_client zeta12 9001 reader
server: ana joins the chat
ana: entro
server: carlos joins the chat
carlos: Hola
```

```
carlos: quién está ahí?
ana: estoy yo, soy ana
carlos: ana dime algo
ana: hola carlos
carlos: adios
ana: hasta luego
server: carlos leaves the chat
```

```
$ ./chat_client zeta12 9001 carlos
Message: Hola
Message: quién está ahí?
Message: ana dime algo
Message: adios
Message: .quit
$
```

```
$ ./chat_client zeta12 9001 ana
Message: entro
Message: estoy yo, soy ana
Message: hola carlos
Message: hasta luego
Message:
```

```
$ ./chat_client zeta12 9001 ana
Message: he entrado?
Message: .quit
$
```

En el ejemplo puede verse cómo intenta entrar en el chat un segundo cliente escritor con *nick* `ana`, mientras un primer escritor con *nick* `ana` sigue en el chat. Sus mensajes son ignorados por el servidor, por lo cual en el lector no aparece ni su intento de entrada ni su mensaje `he entrado?`, ni notificación de su salida.

En el primer cliente, lanzado como lector, pueden verse todos los mensajes que van enviando los clientes escritores `carlos` y `ana`.

2.2. Implementación

El programa cliente tendrá dos comportamientos diferentes según sea lanzado como lector (con el *nick* `reader`) o escritor (con cualquier otro *nick*).

Cuando es lanzado **como escritor**, el cliente enviará un mensaje `Init` al servidor para indicarle su *nick*. Tras enviar ese mensaje, el programa entrará en un bucle que pida al usuario cadenas de texto, que le irá enviando al servidor mediante mensajes `Writer` (el servidor reenviará estos mensajes a todos los clientes lectores). El cliente escritor termina cuando el usuario introduce la cadena de texto `.quit` en el teclado. En ese momento el cliente escritor enviará un mensaje `Logout` al servidor y finalizará su ejecución.

Cuando es lanzado **como lector**, el cliente enviará un mensaje `Init` al servidor para indicarle su *nick*, que siempre será `reader`. Tras enviar este mensaje, el programa entrará en un bucle infinito esperando a recibir los mensajes `Server` enviados por el servidor, cuyo contenido mostrará en pantalla. El cliente lector nunca termina su ejecución[1].

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

3. Descripción del programa servidor `chat_server.adb`

3.1. Interfaz de usuario

El programa servidor se lanzará pasándole 1 argumento en la línea de comandos:

[1] Para interrumpir el programa habrá que pulsar CTRL-C en la ventana de terminal en la que está lanzado

- Número del puerto en el que escucha el servidor

Una vez lanzado, el servidor recibirá mensajes procedentes de clientes:

- Si recibe un mensaje `Init`, añadirá el cliente a la colección de clientes lectores o a la de escritores, y, además, cuando el cliente sea un nuevo escritor, enviará **a todos los lectores** un mensaje `Server` notificando la entrada del nuevo usuario en el chat.
- Si recibe un mensaje `Writer`, comprobará si pertenece a un cliente conocido y, en caso afirmativo, enviará **a todos los lectores** un mensaje `Server` conteniendo el *nick* del cliente escritor y el texto que contenía el mensaje `Writer` recibido.
- Si recibe un mensaje `Logout`, comprobará si pertenece a un cliente conocido y, en caso afirmativo, eliminará al cliente de la colección de clientes escritores, y enviará **a todos los lectores** un mensaje `Server` notificando la salida del usuario del chat.

El servidor nunca terminará su ejecución[2].
El servidor irá mostrando en pantalla los mensajes que vaya recibiendo para permitir comprobar su funcionamiento.
Ejemplo de ejecución que se corresponde con los mensajes enviados por los clientes del ejemplo del apartado anterior:

```
$ ./chat_server 9001
INIT received from reader
INIT received from ana
INIT received form carlos
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
INIT received from ana. IGNORED, nick already used
WRITER received from carlos: ana dime algo
WRITER received from ana: hola carlos
WRITER received from unknown client. IGNORED
WRITER received from carlos: adios
WRITER received from ana: hasta luego
LOGOUT received from carlos
LOGOUT received from unknown client, IGNORED
```

En el ejemplo puede verse cómo es ignorado el mensaje inicial del segundo cliente escritor que quiere usar el *nick* `ana`, y como posteriormente sus mensajes `Writer` y `Logout` son también ignorados al venir de un cliente desconocido.

3.2. Implementación

El servidor debe atarse en 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, el servidor entrará en un bucle infinito recibiendo mensajes de clientes. El servidor deberá almacenar en **dos colecciones** los clientes conocidos. **Para implementar cada colección se debe escribir un paquete que exporte en su especificación un Tipo Abstracto de Datos implementado con memoria dinámica mediante una lista enlazada**. De cada cliente el servidor deberá almacenar su `End_Point` y su `nick`.

Cuando el servidor reciba un mensaje `Init` de un cliente, si es de un lector lo añadirá a la colección de clientes lectores conocidos. Si es de un escritor y su *nick* aún no está siendo usado, lo añadirá a su colección de clientes escritores conocidos y enviará un mensaje `Server` a todos los clientes lectores indicando el *nick* del nuevo usuario que entra en le chat. Si el *nick* recibido en un mensaje `Init` de un escrirtor ya está siendo utilizado por otro cliente, el servidor ignorará ese mensaje inicial, y no enviará ningún mensaje `Server` a los lectores.

Cuando el servidor reciba un mensaje `Writer` de un cliente, comprobará si el mensaje pertenece a un cliente que esté en su colección de clientes escritores conocidos. Si es así, enviará un mensaje `Server` a todos los clientes de la colección de lectores indicando el *nick* del cliente escritor y el texto contenido en el mensaje recibido.

Cuando el servidor reciba un mensaje `Logout` de un cliente, comprobará si el mensaje pertenece a un cliente que esté en su colección de clientes escritores conocidos. Si es así, eliminará a dicho cliente de esa colección y enviará un mensaje `Server` a todos los clientes de la colección de lectores indicando el *nick* del usuario que abandona el chat.

[2]Para interrumpir el programa habrá que pulsar CTRL-C en la ventana de terminal en la que está lanzado

En el apartado 4 se explica el formato de los mensajes mencionados. Como puede verse en ese apartado, el mensaje `Writer` y el mensaje `Logout` que envían los clientes sólo contienen, para identicar al usuario del chat, el `End_Point` del cliente escritor, y no se incluye su *nick*. Por lo tanto el servidor deberá buscar en su colección de clientes escritores conocidos dicho `End_Point` y así podrá obtener el *nick* asociado a él. Lo necesita para poder componer el mensaje `Server` que enviará a todos los clientes de la colección de clientes lectores, pues este mensaje sí incluye el *nick* del cliente escritor al que corresponde la información. Si el `End_Point` buscado no aparece en la colección de escritores, es que el mensaje `Writer` o `Logout` que ha recibido procede de un cliente escritor desconocido, y debe ser ignorado, y por no lo tanto, no se enviará el mensaje `Server` informando a los clientes de la colección de lectores.

NOTA: Para poder comparar dos `End_Points` puede utilizarse directamente el operador de comparación de igualdad siempre que se incluya también una clásula `use type ...` de dicho tipo en la zona declarativa del procedimiento o del paquete:

```
use type LLU.End_Point_Type;

EP_1, EP_2: LLU.End_Point_Type;

...

if EP_1 = EP_2 then ...
```

4. Formato de los mensajes

Los 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, Writer, Server, Logout);
```

MUY IMPORTANTE: Es imprescindible mantener el orden mostrado en los valores de este tipo, a fin de que sean compatibles las implementaciones de clientes y servidores realizadas por distintos alumnos.

Dicho tipo deberá estar declarado **única y exclusivamente** en el fichero `chat_messages.ads` y desde ahí será 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 ...;
```

NOTA: De nuevo es necesario un `use type ...` del tipo para poder usar directamente el operador de comparación entre dos `Message_Type`.

Si este 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`	`Nick`

en donde:

- `Init`: valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `Client_EP`: `End_Point` del cliente que envía el mensaje.
- `Nick`: `Unbounded_String` con el *nick* del cliente. Si el *nick* es `reader`, el cliente estará en modo lector, en caso contrario estará en modo escritor.

Mensaje `Writer`

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

`Writer`	`Client_EP`	`Comentario`

en donde:

- `Writer`: valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `Client_EP`: `End_Point` del cliente que envía el mensaje.
- `Comentario`: `Unbounded_String` con la cadena de caracteres introducida por el usuario

Nótese que el *nick* no viaja en estos mensajes: sólo se puede identificar al cliente que envía un mensaje `Writer` a partir del campo `Client_EP`. El servidor, tras recibir este mensaje, deberá buscar en la colección de clientes el *nick* asociado a `Client_EP`. Y con él podrá componer el mensaje de servidor que reenviará a los clientes lectores.

Mensaje `Server`

Es el que envía un servidor a cada cliente lector, tras haber recibido un mensaje `Writer` procedente de un cliente escritor conteniendo un texto escrito por un usuario. El servidor envía el mensaje `Server` para comunicar a todos los lectores dicho texto. Formato:

`Server`	`Nick`	`Texto`

en donde:

- `Server`: valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `Nick`: `Unbounded_String` con el *nick* del cliente que escribió el texto. Si el mensaje es para informar de que un nuevo cliente ha entrado en el chat, este campo tendrá el valor `server`.
- `Texto`: `Unbounded_String` con la cadena de caracteres introducida por el usuario. Si el mensaje es para informar de que un nuevo cliente ha entrado en el chat, este campo tendrá el valor del *nick* del usuario concatenado con la cadena `joins the chat`.

Mensaje `Logout`

Es el que envía un cliente escritor al servidor para informarle de que abandona el chat. Formato:

`Logout`	`Client_EP`

en donde:

- `Logout`: valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `Client_EP`: `End_Point` del cliente que envía el mensaje.

Nótese que, igual que en el caso de los mensajes `Writer`, el *nick* no viaja en estos mensajes. El servidor, tras recibir este mensaje, deberá buscar en la colección de clientes el *nick* asociado a `Client_EP`. Y con él podrá componer el mensaje `Server` que reenviará a los clientes lectores.

5. Condiciones obligatorias de funcionamiento

1. Se supondrá que nunca se pierden los mensajes enviados ni por el servidor ni por los clientes.

2. Los programas deberán ser robustos, comportándose de manera adecuada cuando no se arranquen con las opciones adecuadas en línea de comandos.

3. El servidor debe almacenar los clientes lectores conocidos y los clientes escritores conocidos en dos variables distintas. Estas variables serán del mismo **tipo abstracto de datos al que en el enunciado nos referimos como colección, que debe estar implementado con una lista dinámica**. Dicha lista dinámica debe estar implementada en el paquete `Client_Collections` cuya especificación, **que no puede modificarse**, es la siguiente:

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

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

   type Collection_Type is limited private;
   Client_Collection_Error: exception;

   procedure Add_Client (Collection: in out Collection_Type;
                         EP: in LLU.End_Point_Type;
                         Nick: in ASU.Unbounded_String;
                         Unique: in Boolean);

   procedure Delete_Client (Collection: in out Collection_Type;
                            Nick: in ASU.Unbounded_String);

   function Search_Client (Collection: in Collection_Type;
                           EP: in LLU.End_Point_Type)
                           return ASU.Unbounded_String;

   procedure Send_To_All (Collection: in Collection_Type;
                          P_Buffer: access LLU.Buffer_Type);

   function Collection_Image (Collection: in Collection_Type) return String;

   function Total (Collection: in Collection_Type) return Natural;

private
   type Cell;
   type Cell_A is access Cell;
   type Cell is record
      Client_EP: LLU.End_Point_Type;
      Nick: ASU.Unbounded_String;
      Next: Cell_A;
   end record;

   type Collection_Type is record
      P_First: Cell_A;
      Total: Natural := 0;
   end record;
end Client_Collections;
```

4. A diferencia de la práctica anterior, el tipo que representa a toda la estructura dinámica (`Collection_Type`) no es simplemente un puntero al primer elemento de la misma, sino que ahora es un registro que contiene 2 campos: dicho puntero al primer elemento, y un campo con el total de elementos que hay en la colección.

5. Comportamiento esperado de los subprogramas:

   - `Add_Client`:
     - Si el parámetro `Nick` NO está en la lista, crea una nueva celda para el cliente, almacenando en ella su `EP` y su `Nick`. Aumenta en 1 el total de elementos de la lista.
     - Si el parámetro `Nick` ya está en la lista y el parámetro `Unique` es True, eleva la excepción `Client_Collection_Error`.
     - Si el parámetro `Nick` ya está en la lista y el parámetro `Unique` es False, crea una nueva celda para el cliente, almacenando en ella su `EP` y su `Nick`. Aumenta en 1 el total de elementos de la lista.
   - `Delete_Client`: Si `Nick` está en la lista, elimina su celda de la lista y libera la memoria ocupada por ella (llamando adecuadamente a `Free`), y disminuye en 1 el total de elementos de la lista. Si `Nick` no está en la lista, eleva la excepción `Client_Collection_Error`.
   - `Search_Client`: Si `EP` está en la lista, devuelve su `Nick`. Si `EP` no está en la lista, eleva la excepción `Client_Collection_Error`.
   - `Send_To_All`: Este subprograma lo usará el servidor pasando como primer parámetro la colección de clientes lectores. Envía a todos los clientes de la colección que se pasa como parámetro el mensaje que hay en el Buffer apuntado por `P_Buffer`. Si la colección está vacía, el subprograma no hace nada, pero no eleva ninguna excepción.
   - `Collection_Image`: Devuelve un `String` con la concatenación de los datos de todos los clientes de la colección que se pasa como parámetro, **en orden inverso al que se introdujeron en ella**. El formato deberá ser el siguiente:
     - Para cada elemento de la colección, se concatenan los datos del EP (IP y puertos separados por el carácter ":"), un espacio, y el nick. Ejemplo: `"193.147.49.72:1025 carlos"`
     - Los datos de los diversos elementos de la colección se concatenan poniendo entre ellos el carácter ASCII.LF (fin de línea). Ejemplo:
       `"193.147.49.72:1025 carlos" & ASCII.LF & "193.147.49.72:1026 ana"`
       Así, si la colección contiene sólo esos dos elementos, y se escribe en pantalla el contenido de dicho `String`, se mostrará:

       ```
       193.147.49.72:1025 carlos
       193.147.49.72:1026 ana
       ```

       Si la colección está vacía, esta función devuelve un `String` vacío.
       Para obtener la IP y el puerto contenidos en un `End_Point`, puede usarse la siguiente función del paquete `Lower_Layer_UDP`:

       ```
          function Image (EP: End_Point_Type) return String;
       ```

       Dicha función devuelve un String con el siguiente formato:

       ```
       LOWER_LAYER.INET.UDP.UNI.ADDRESS IP: 193.147.49.72, Port:  1025
       ```

       Por lo tanto, dicha cadena debe ser troceada convenientemente mediante las funciones `Index`, `Head` y `Tail` (que ya se utilizaron en la práctica anterior) para extraer la IP y el puerto y poder componer la salida de `Collection_Image` tal y como se pide.
   - `Total`: Devuelve un `Natural` con el número de elementos que hay actualmente en la colección.

6. Aunque en la parte básica de la práctica no se necesita utilizar el subprograma `Collection_Image`, el alumno debe implementarlo correctamente, mostrando la colección de todos los clientes **en orden inverso al que se introdujeron en ella**.

7. La salida del programa deberá ser **exactamente igual** a la que se muestra en los ejemplos de ejecución.

8. Un cliente programado por un alumno deberá poder funcionar con un servidor programado por cualquier otro alumno, y viceversa. Es conveniente que pruebes tus programas con los de otros alumnos. Por ello es imprescindible respetar el protocolo de comunicación entre cliente y servidor y, especialmente, el formato de los mensajes.

9. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.

6. Extensión: programa `chat_admin.adb`

El alumno debe extender el funcionamiento del chat desarrollado en la parte básica desarrollando un tercer programa (`chat_admin.adb`) que permita realizar labores de administración en el chat.

6.1. Interfaz de usuario de `chat_admin.adb`

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

- Nombre de la máquina en la que está el servidor
- Número de puerto en el que escucha el servidor
- *Password* (Contraseña) de administración

Una vez lanzado, el programa mostrará al usuario un menú interactivo permitiendo al usuario realizar las siguientes tareas:

- Mostrar la colección de clientes escritores conocidos por el servidor, incluyendo su `End_Point` y su `nick`
- Expulsar del chat a un cliente escritor dado su `nick`
- Terminar la ejecución del servidor
- Salir del programa `chat_admin`

La contraseña de administración debe coincidir con la que se le pase como parámetro adicional al servidor (ver apartado 6.3).
Ejemplo de ejecución:

```
$ ./chat_admin zeta12 9001 admin
Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 1

193.147.49.72:1025 carlos
193.147.49.72:1026 ana

Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 2
Nick to ban? carolina

Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 1

193.147.49.72:1025 carlos
193.147.49.72:1026 ana

Your option? 2
```

```
Nick to ban? carlos

Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 1

193.147.49.72:1026 ana

Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 3
Server shutdown sent

Options
1 Show writers collection
2 Ban writer
3 Shutdown server
4 Quit

Your option? 4
$
```

6.2. Implementación de `chat_admin.adb`

Para ver la colección de clientes escritores, el programa enviará un mensaje `Collection_Request`, y esperará como respuesta del servidor un mensaje `Collection_Data` que contiene la información de los clientes escritores a mostrar en pantalla.

Para expulsar a un cliente, el programa enviará un mensaje `Ban` indicando el *nick* del cliente escritor a expulsar. Este mensaje no es respondido por otro desde el servidor, por lo que para comprobar si ha tenido los efectos previstos el usuario administrador tendrá que elegir después la opción para mostrar la colección de clientes escritores.

Para provocar que termine la ejecución del servidor, el programa enviará un mensaje `Shutdown`. Este mensaje no tiene respuesta.

Los detalles de los nuevos tipos de mensajes se muestran en el apartado 6.4. Todos los mensajes que envía el programa `chat_admin` incluyen la contraseña de administración recibida por la línea de comandos. Si no fuera la correcta, sus mensajes serán ignorados por el servidor. Por tanto un mensaje `Collection_Request` con password incorrecta no obtendrá respuesta. Por ello, si tras enviar un mensaje `Collection_Data` con password incorrecta pasan **5 segundos** sin recibir respuesta, el programa `chat_admin` terminará su ejecución mostrando el mensaje `Incorrect password`.

6.3. Cambios en `chat_server.adb`

Para soportar la funcionalidad del programa `chat_admin`, el programa `chat_server` deberá sufrir las siguientes modificaciones con respecto a la parte básica:

- Deberá recibir un segundo argumento en la línea de comandos con la contraseña de adminsitración que debe recibir del programa `chat_admin`.
- El programa `chat_server` ya no será un bucle infinito recibiendo mensajes, sino que terminará su ejecución al recibir un mensaje `Shutdown`.
- Deberá estar preparado para atender los nuevos tipos de mensajes:

- Al recibir un mensaje `Collection_Request`, comprobará si la contraseña es correcta y enviará como respuesta un mensaje `Collection_Data`. Como puede verse en el apartado 6.4, todos los datos de la colección de clientes escritores deben enviarse en un único `Unbounded_String`. Se sugiere concatenar las IPs, puertos y *nicks* junto con caracteres fin de línea (`ASCII.LF`) para formar el `Unbounded_String` ya listo para ser mostrado en pantalla. Si la contraseña no fuera correcta, se ignorará el mensaje.
- Al recibir un mensaje `Ban`, comprobará si la contraseña es correcta y si el *nick* es de un escritor que está en la colección de clientes escritores conocidos. En ese caso, eliminará a ese cliente de la colección. Si la contraseña no fuera correcta, si el *nick* fuera `reader` o si el *nick* no estuviera en la colección de escritores, se ignorará el mensaje.
- Al recibir un mensaje `Shutdown`, comprobará si la contraseña es correcta. En ese caso, el cliente terminará su ejecución. Si la contraseña no fuera correcta, se ignorará el mensaje.

Ejemplo de ejecución:

```
$ ./chat_server 9001 admin
INIT received from ana
INIT received form carlos
WRITER received from ana: entro
WRITER received from carlos: Hola
...
LIST_REQUEST received
BAN received for carolina. IGNORED, nick not found
LIST_REQUEST received
BAN received for carlos
LIST_REQUEST received
LIST_REQUEST received. IGNORED, incorrect password
SHUTDOWN received
```

6.4. Formato de los nuevos mensajes

Los nuevos tipos de mensajes que se necesitan para la extensión de esta práctica serán valores adicionales para el tipo enumerado `Message_Type`, con lo que el tipo quedará definido ahora en la forma:

```
type Message_Type is (Init, Writer, Server, Logout, Collection_Request,
                      Collection_Data, Ban, Shutdown);
```

MUY IMPORTANTE: Es imprescindible mantener el orden mostrado en los valores de este tipo, manteniendo los mensajes de la parte básica en el mismo sitio, a fin de que un servidor de la extensión sea compatible también con clientes de la parte básica realizados por cualquier otro alumno.

Mensaje `Collection_Request`

Es el que envía `chat_admin` al `chat_server` para pedirle la colección de clientes. Formato:

`Collection_Request`	`Admin_EP`	`Password`

en donde:

- `Collection_Request`: `Message_Type` que identifica el tipo de mensaje.
- `Admin_EP`: `End_Point_Type` con el valor del *End_Point* en el que escucha el programa `client_admin` y en el que esperará la recepción del mensaje `Collection_Data` de respuesta.
- `Password`: `Unbounded_String` con la contraseña de administración.

Mensaje `Collection_Data`

Es el que envía `chat_server` al `chat_admin` como respuesta a un mensaje `Collection_Request`. Formato:

`Collection_Data`	`Data`

en donde:

- `Collection_Data`: `Message_Type` que identifica el tipo de mensaje.
- `Data`: `Unbounded_String` con los datos de todos los clientes escritores en el mismo formato en que los devuelve el subprograma `Collection_Image` pero convertido a `Unbounded_String`.

Mensaje `Ban`

Es el que envía `chat_admin` al `chat_server` para expulsar a un cliente. Formato:

`Ban`	`Password`	`Nick`

en donde:

- `Ban`: `Message_Type` que identifica el tipo de mensaje.
- `Password`: `Unbounded_String` con la contraseña de administración.
- `Nick`: `Unbounded_String` con el *nick* del cliente a expulsar.

Mensaje `Shutdown`

Es el que envía `chat_admin` al `chat_server` para que termine su ejecución. Formato:

`Shutdown`	`Password`

en donde:

- `Shutdown`: `Message_Type` que identifica el tipo de mensaje.
- `Password`: `Unbounded_String` con la contraseña de administración.

7. Entrega

El límite para la entrega de esta práctica P2 es: **Lunes 1 de noviembre a las 23:59h**

La entrega se realizará a través de la tarea dispuesta para ello en el aula virtual.

La **Prueba de Laboratorio 2 (L2)** correspondiente a esta práctica se realizará en el aula de prácticas habitual el jueves 2 de noviembre a las 11:00h.

A dicha prueba el alumno acudirá con el código de su práctica, y en el transcurso de la misma deberá ser capaz de realizar los ejercicios que se indiquen.