Programación de Sistemas de Telecomunicación / Informática II
Práctica 4

GSyC
Departamento de Teoría de la Señal y Comunicaciones y Sistemas Telemáticos y Computación

30 de Noviembre de 2017

1. Parte I

La parte I de la práctica P4 consiste en modificar el código de Mini-Chat 2.0 desarrollado en la práctica P3 en los dos siguientes aspectos:

1.1. Tabla de símbolos de clientes activos como Tabla Hash

El servidor en la práctica P3 mantiene una tabla de símbolos[1] de clientes activos del chat. En esta nueva práctica P4, esta tabla de símbolos tiene que estar implementada como una tabla hash con **resolución de colisiones mediante encadenamiento**, tal y como se describe en el tema 6, Otras Estructuras de Datos.

El paquete con la implementación de la tabla hash deberá tener la siguiente especificación en su parte pública:

```
generic
   type Key_Type is private;
   type Value_Type is private;
   with function "=" (K1, K2: Key_Type) return Boolean;
   type Hash_Range is mod <>;
   with function Hash (K: Key_Type) return Hash_Range;
   Max: in Natural;

package Hash_Maps_G is

   type Map is limited private;

   Full_Map : exception;

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


   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);
```

[1] estructura de datos también conocida con otros nombres como *map*, *array asociativo*, *diccionario*,...

```
   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
   ...
end Hash_Maps_G;
```

La función `Hash` (parámetro de instanciación del paquete) debe convertir valores del tipo `Key_Type` en valores de un tipo modular `Hash_Range` que también es un parámetro de instanciación del paquete.

Los tipos modulares son tipos numéricos que se definen en Ada de la siguiente forma:

```
type My_Hash_Range is mod 27;

I: My_Hash_Range;
```

De esta forma, `I` podrá tomar valores del 0 al 26, con aritmética modular, lo que significa que al hacer:

```
I := 26;
I := I + 1;
```

el valor de `I` pasa a ser 0; y al hacer:

```
I := 0;
I := I - 1;
```

el valor de `I` pasa a ser 26.

Al efecturar operaciones con valores de tipos no modulares (como `Integer`) e intentar asignar el resultado a una variable de un tipo modular hay que tener cuidado de que no se obtenga un valor fuera del rango de valores permitidos (lo que produciría un `Constraint_Error`). Para que el resultado de esas asignaciones se asigne en forma modular puede utilizarse el atributo `'Mod`, en la forma:

```
type My_Hash_Range is mod 27;

H: My_Hash_Range;
I, J, K: Integer;

...

H := My_Hash_Range'Mod ( I + J * K );
```

Con respecto a la función `Hash`, si el tipo que vaya a utilizarse como `Key_Type` de la tabla de símbolos no es numérico, puede usarse la siguiente técnica para construir una función Hash que permita convertir la clave en un valor del tipo modular que devuelve la función Hash:

1. Pasar el tipo a un valor `Unbounded_String`
2. Sumar el valor numérico de cada carácter del `Unbounded_String`, que se obtiene usando `Character'Pos(C)`, siendo `C` de tipo `Character`.
3. Convertir el resultado de esa suma a un valor del tipo modular.

La función `Hash` de esta manera será capaz de calcular un valor del tipo modular `Hash_Range` para cada posible valor de clave. Este valor calculado será el índice de la posición del array en la que se almacerá el elemento con esa clave.

Cuando varios elementos con distinta clave resulten tener el mismo resultado de la función Hash, decimos que se produce una colisión. la implementación de la tabla hash con resolución de colisiones mediante encadenamiento establece que todos los elementos que colisionan estarán en una lista enlazada asociada a esa posición del array. Es decir, en cada posición del array de la tabla hash lo que hay es una lista enlazada de todos los elementos cuya clave produce el mismo valor al aplicarle la función hash.

A la hora de instanciar el paquete genérico de la tabla de símbolos, para determinar el tamaño del array ten en cuenta el valor máximo de elementos que querrás guardar en ella.

Deberías probar exhaustivamente la tabla hash antes de utilizarla en el chat. Te dejamos como ejemplo un programa de prueba (`hash_maps_test.adb` que instancia la tabla hash para claves y valores de tipo `Natural`. Estudia el código del programa de prueba, y en particular, mira cómo se declara la función Hash y cómo se instancia el paquete del mapa. Piensa cuál debería ser la salida correcta que debería mostrar este programa de prueba: Deberían aparecen los elementos ordenados por su resto de dividir la clave por 10, y dentro de los que tienen el mismo resto, por orden de inserción en el mapa.

Modifícalo para probar tu implementación del mapa también con claves que sean `Unbounded_String`.

1.2. Tabla de símbolos de clientes antiguos como array ordenado con búsqueda binaria

El servidor en la práctica P3 mantiene una tabla de símbolos de clientes antiguos del chat. En esta nueva práctica P4, esta tabla de símbolos tiene que estar implementada con un array ordenado con búsqueda binaria, tal y como se menciona en el tema 5, Tablas de Símbolos.

El paquete con la implementación de esta tabla de símbolos deberá tener la siguiente especificación en su parte pública:

```
generic
   type Key_Type is private;
   type Value_Type is private;
   with function "=" (K1, K2: Key_Type) return Boolean;
   with function "<" (K1, K2: Key_Type) return Boolean;
   Max: in Natural;

package Ordered_Maps_G is

   type Map is limited private;

   Full_Map : exception;

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


   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
   ...
end Ordered_Maps_G;
```

Los elementos del array deberán mantenerse en el orden que establezca la función `"<"` que se use en la instanciación del paquete. Así, dentro de la implementación del paquete, se usará esa función `"<"` (junto con `"="`) para recorrer el array cada vez que se necesite, tanto en la implementación de `Get`, como en `Put` y en `Delete`, a fin de realizar un proceso de búsqueda binaria.

Deberías probar exhaustivamente el array ordenado antes de utilizarlo en el chat. Realiza un programa de prueba para ello.

1.3. Condiciones de funcionamiento

Toda la funcionalidad descrita en el enunciado de la práctica P3 para Mini-Chat 2.0 deberá estar implementada en el código entregado, excepción hecha de la implementación de las tablas de símbolos de clientes activos y de clientes antiguos, que deberán implementarse según se ha descrito en las anteriores secciones:

- Con una tabla hash con resolución de colisiones mediante encadenamiento la tabla de símbolos de clientes activos.
- Con un array ordenado con búsqueda binaria la tabla de símbolos de clientes antiguos.

2. Parte II: Mini-Chat 3.0, un chat fiable ante pérdidas y desorden de mensajes

2.1. Resumen

`Lower_Layer_UDP` ofrece un servicio de entrega de mensajes no fiable y no ordenada. En esta parte opcional de la práctica P4 se extiende la funcionalidad de Mini-Chat 2.0 para que el servicio ofrecido por Mini-Chat 3.0 sea fiable y con entrega ordenada. Para conseguirlo, habrá que retransmitir los mensajes enviados hasta que éstos sean asentidos.

Cuando se ejecutan los clientes y el servidor de Mini-Chat 2.0 en una misma máquina o en dos máquinas conectadas al mismo *switch ethernet*, es difícil que ocurran fallos en la red que provoquen que los mensajes enviados por estos procesos se pierdan. Así mismo, es difícil que los mensajes lleguen desordenados al destino. Por ello, para poder verificar el correcto funcionamiento del código que desarrolles, utilizarás el servicio de inyección de fallos que proporciona el paquete `Lower_Layer_UDP`, que permite simular fallos en el envío de mensajes y retardos de propagación (ver sección 2.2). Al arrancar los clientes y el servidor se le pasarán 3 nuevos argumentos en la línea de comandos que permitan configurar la inyección de fallos y el retardo que sufrirán los mensajes (ver sección 2.3).

Para que el cliente se recupere de las pérdidas de mensajes `Init` se utilizará un protocolo sencillo de parada y espera. La recepción de un mensaje `Welcome` se considerará como señal de que el mensaje `Init` ha llegado al servidor, lo que hará que el mensaje `Init` deje de retransmitirse (ver sección 2.5.1).

Para que el cliente se recupere de las pérdidas de mensajes de tipo `Writer` y `Logout`, y para que el servidor se recupere de las pérdidas de mensajes de tipo `Server`, tanto cliente como servidor utilizarán un protocolo de envío continuo sin ventana. Estos tres mensajes serán asentidos por un nuevo tipo de mensaje `Ack`. Cuando se reciba el `Ack` correspondiente se dejará de retransmitir el mensaje asentido. Ver sección 2.5.2. Los mensajes `Writer`, `Server` y `Logout` ven modificado su formato. En la sección 2.5.3 se detalla el nuevo formato de estos mensajes, así como el del nuevo mensaje `Ack`.

En esta práctica, tanto el cliente como el servidor tienen que realizar tres actividades concurrentemente:

1. enviar mensajes cuando se lea algo de la entrada estándar (teclado)
2. procesar un mensaje cuando se reciba uno a través de `Lower_Layer_UDP`
3. retransmitir mensajes enviados aún no asentidos cuando llegue el momento en el que vence su plazo de retransmisión

Gracias a la recepción asíncrona de mensajes de `Lower_Layer_UDP`, en P3 hemos podido programar las dos primeras actividades concurrentes de la anterior lista.

En esta práctica P4 introducimos un nuevo mecanismo que nos permitirá programar la tercera actividad concurrente (la de las retransmisiones): el paquete `Protected_Ops` permite planificar la ejecución de un procedimiento a una hora determinada. Utilizarás este mecanismo para implementar el **procedimiento de gestión de retransmisiones** de los mensajes `Writer`, `Server` y `Logout`. Llegado el momento en el que se planificó la ejecución del procedimiento de gestión de retransmisiones el sistema lo ejecutará en un nuevo hilo de ejecución[2]. El paquete `Protected_Ops` también ofrece un servicio de control de concurrencia que permite ejecutar procedimientos que no pueden ser interrumpidos por otro hilos. En la práctica utilizarás este servicio para que los 3 hilos del programa (el que ejecuta el programa principal, el que ejecuta el manejador de recepción de mensajes y el que ejecuta el procedimiento de gestión de retransmisiones) puedan acceder concurrentemente a las estructuras de datos que comparten sin ser interrumpidos mientras que están consultándolas o modificándolas. Algunas de las estructuras de datos que tienen que ser accedidas concurrentemente por estos tres hilos son: la tabla de símbolos de clientes activos del servidor, la tabla de símbolos de clientes antiguos del servidor, y, tanto en el cliente como en el servidor, las nuevas colecciones de datos que se introducen en esta práctica que son necesarias para la gestión de las retransmisiones de mensajes. Los servicios ofrecidos por este paquete se explican en la sección 2.4.

En la sección 2.6 se proporcionan algunos detalles sobre las nuevas estructuras de datos y los algoritmos que tendrás que implementar para que tanto el cliente como el servidor puedan recuperarse de situaciones en las que se han perdido mensajes, y para que puedan entregar los mensajes recibidos en el mismo orden en el que se enviaron.

2.2. Inyección de fallos, desorden y retardos de propagación

`Lower_Layer_UDP` ofrece un servicio de transmisión de mensajes **no fiable**. Esto significa que no se garantiza que los mensajes enviados con `Send` lleguen a su destino, ni que los mensajes que lleguen al destino lo hagan en el mismo orden en que se enviaron.

Hay que tener en cuenta que aunque el servicio sea no fiable, si los mensajes se envían entre programas que se ejecutan en máquinas de una misma subred resulta prácticamente imposible que se produzcan pérdidas, retardos de propagación apreciables o desorden en la llegada de mensajes. Por ello no se observan problemas cuando se ejecuta Mini-Chat 2.0. Si se ejecutasen los clientes y el servidor en máquinas situadas en diferentes subredes IP separadas por varios encaminadores, sí se podrían observar fallos en la red y desorden en la entrega de mensajes, lo que haría que la aplicación no ofreciese un servicio fiable a sus usuarios.

Por ello, para poder comprobar que el código de Mini-Chat v3.0 que tienes que desarrollar tolera pérdidas de mensajes y entrega desordenada de mensajes, en esta práctica utilizarás nueva funcionalidad de `Lower_Layer_UDP` que permite inyectar retardos artificiales a propósito, y pérdidas de paquetes.

2.2.1. Simulación de las pérdidas de paquetes

El procedimiento `Set_Faults_Percent` del paquete `Lower_Layer_UDP` provoca que se pierda un porcentaje de todos los envíos que se realicen a partir del instante en que este procedimiento es llamado. Normalmente se incluye una llamada a este procedimiento nada más arrancar la aplicación. El porcentaje de pérdidas se especifica como argumento en la llamada al subprograma, expresado como un valor comprendido entre 0 y 100.

Así, al principio del programa principal puede especificarse un porcentaje de pérdidas de, por ejemplo, el 25 %, invocando dicho procedimiento en la forma:

```
LLU.Set_Faults_Percent (25);
```

[2] también conocido como *thread* o *tarea*

A partir de que se ejecute esta llamada, cada invocación de `Send` cuenta con un 25 % de probabilidades de que dicho envío se pierda.

La llamada a `Set_Faults_Percent` debe realizarse una sola vez al principio del programa principal, y afecta a todos los envíos que hace el mismo, incluidos los envíos realizados desde el manejador.

2.2.2. Simulación de los retardos de propagación

Mini-Chat v3.0 debe tolerar retardos de propagación elevados y variables que puedan provocar que los mensajes lleguen desordenados a sus destinos.

`Lower_Layer_UDP` permite introducir retardos de propagación simulados. Para ello incluye el procedimiento `Set_Random_Propagation_Delay` que permite especificar los retardos mínimo y máximo que pueden sufrir los mensajes enviados con `Send`. Los valores de dichos retardos se expresan como argumentos de tipo `Integer` que representan un valor en **milisegundos**.

Así, al principio de un programa puede especificarse que se simulen retardos variables entre, por ejemplo, 0 y 500 milisegundos, invocando este procedimiento en la forma:

```
LLU.Set_Random_Propagation_Delay (0, 500)
```

A partir de que se ejecute esta llamada, cada envío realizado en el programa con `Send` se verá afectado por un retardo de propagación simulado de un número de milisegundos elegido aleatoriamente entre 0 y 500.

Nótese que, al poder experimentar distintos envíos retardos diferentes, el utilizar `Set_Random_Propagation_Delay` implica que pueden llegar desordenados los mensajes a su destino, siendo éste precisamente el efecto buscado.

La llamada a `Set_Random_Propagation_Delay` debe realizarse una sola vez al principio del programa principal, y afecta a todos los envíos que hace el mismo, incluidos los envíos realizados desde el manejador.

2.2.3. Plazo de retransmisión

El mensaje `Init` debe ser contestado mediante un mensaje `Welcome`, que aparte de informar sobre si el cliente ha sido aceptado o no, tiene la función de actuar como un asentimiento del mensaje `Init`. Los mensajes `Writer`, `Server` y `Logout` tienen que ser asentidos mendiante mensajes `Ack`.

El programa que envía un mensaje `Init`, `Writer`, `Server` o `Logout` tiene que **retransmitir** el mensaje si no recibe su `Ack` (mensaje `Welcome` en el caso de `Init`) pasado un plazo de tiempo denominado **plazo de retransmisión**.

El *plazo de retransmisión* debe establecerse en relación con el retardo máximo de propagación. Si desde que un proceso envió un mensaje ha pasado ya un tiempo igual al doble del retardo máximo de propagación sin que se haya recibido el asentimiento del destinatario, es prácticamente seguro que este asentimiento ya no va a llegar, y por lo tanto es razonable retransmitir el mensaje sin esperar más.

En un escenario real es difícil predecir el retardo máximo de propagación de los mensajes, calculándose normalmente una estimación en función de los retardos observados para pasados envíos. En esta práctica no calcularemos los retardos de propagación en función del tiempo de ida y vuelta (*round trip time*) de los mensajes que se van enviando y recibiendo, sino que usaremos un valor fijado de retardo máximo de propagación (`max_delay`), que le pasaremos como argumento a los programas cliente y servidor en la línea de comandos, expresado en milisegundos. El plazo de retransmisión siempre debe ser mayor o igual al retardo máximo de propagación. En esta práctica utilizaremos un plazo de retransión igual a dos veces el retardo máximo de propagación. Por ello tiene sentido fijar el *plazo de retransmisión* (en segundos) en función del *retardo máximo de propagación* (en milisegundos), de la siguiente manera:

```
Plazo_Retransmision: Duration;
...
Plazo_Retransmision := 2 * Duration(Max_Delay) / 1000;
```

2.3. Interfaz de usuario de los programas cliente y servidor

En la práctica anterior el cliente debía recibir 3 argumentos en la línea de comandos: máquina y puerto a los que está atado el servidor, y *nick*. El servidor debía recibir 2 argumentos: número del puerto al que se debe atar el servidor, y número máximo de clientes.

Además de estos argumentos, ambos programas deberán ahora recibir los siguientes 3 argumentos adicionales (situados al final de los argumentos de la práctica anterior):

- `min_delay`: Retardo mínimo de propagación que sufrirán los envíos que haga el programa (ver apartado 2.2), expresado como un número natural de **milisegundos**.
- `max_delay`: Retardo máximo de propagación que sufrirán los envíos que haga el programa (ver apartado 2.2), expresado como un número natural de **milisegundos**. Debe ser mayor o igual que `min_delay`.
- `fault_pct`: Porcentaje de envíos realizados por el programa que se perderán (ver apartado 2.2), expresado como un número natural entre 0 y 100.

2.4. Paquete Protected_Ops

Los servicios ofrecidos por este paquete se utilizarán en la práctica para satisfacer dos objetivos:

- Para que en ciertos instantes de tiempo, cuando venza un plazo de retransmisión, se ejecute el procedimiento de gestión de retransmisiones. Este procedimiento deberá comprobar si hay algún mensaje pendiente de ser asentido, cuya hora de retransmisión ya haya pasado, y por tanto haya que retransmitirlo.
- Para garantizar que el código que esté manipulando alguna estructura de datos a la que pueda accederse desde varios hilos de ejecución distintos (programa principal, manejador de recepción de mensajes o procedimiento de gestión de retransmisiones ejecutado por el sistema a una hora determinada) no es interrumpido hasta haber dejado la estructura de datos en un estado consistente.

La especificación del paquete `Protected_Ops` es la siguiente:

```
with Ada.Real_Time;

--
--  Calls to any Procedure_A, either those programmed through Program_Timer_Procedure
--   to be executed by the system in the future, or those executed through a call to
--   Protected_Call, are executed in mutual exclusion.
--

package Protected_Ops is
   type Procedure_A is access procedure;

   procedure Program_Timer_Procedure (H: Procedure_A; T: Ada.Real_Time.Time);
   -- Schedules H to be executed at time T by the system. When H.all is called, it
   --  will be executed in a new thread, in mutual exclusion with calls executed
   --  through Protected_Call.

   procedure Protected_Call (H: Procedure_A);
   -- The calling thread executes H.all, in mutual exclusion with other calls made
   --  through Protected_Call, and with calls to procedures scheduled to be executed
   --  by the system through calls to Program_Timer_Procedure

end Protected_Ops;
```

2.4.1. Ejecución de procedimientos a una hora determinada

Mediante la llamada al procedimiento `Protected_Ops.Program_Timer_Procedure` se encarga al sistema la ejecución en un instante del futuro de un procedimiento. El procedimiento que se quiere ejecutar en el futuro, y el instante en el que debe ejecutarse, se le pasan como argumentos `H` y `T` a `Program_Timer_Procedure`. El procedimiento que se le pase como argumento no puede tener argumentos.

El sistema llamará al procedimiento `H` cuando llegue la hora `T` a la que fue encargada su ejecución, creando para ello un nuevo hilo de ejecución. Cuando llegue esa hora el procedimiento se ejecutará de manera concurrente al resto de hilos de ejecución del programa (hilo del programa principal e hilo de recepción de mensajes mediante manejador). Llegado el momento, mientras que se esté ejecutando el procedimiento pasado como argumento a `Program_Timer_Procedure`, éste no será interrumpido por procedimientos que se ejecuten mediante llamadas a `Protected_Call` desde otros hilos de la aplicación.

Sólo puede haber un procedimiento planificado para ser ejecutado a cierta hora: si cuando se ejecuta una llamada a

`Program_Timer_Procedure` ya había un procedimiento planificado para ser ejecutado a otra hora, la nueva llamada a `Program_Timer_Procedure` cancela la anterior, quedando planificada sólo la ejecución del nuevo procedimiento a la nueva hora.

2.4.2. Ejecución de procedimientos en exclusión mútua

`Protected_Call` ejecuta el procedimiento `H` que se le pasa como parámetro en exclusión mútua con otros procedimientos que se estén ejecutando a través de `Protected_Call`, o con el procedimiento que el sistema esté ejecutando tras haber sido programada su ejecución futura a través de `Program_Timer_Procedure`.

El procedimiento `H` se ejecuta en el mismo hilo que llama a `Protected_Call`.

2.4.3. Ejemplos

Para aprender a utilizar este paquete NO hay que estudiar el código del cuerpo del paquete `Protected_Ops`, pero sí el código de su especificación, y el código de los ejemplos que aparecen en las carpetas `test_protected_ops_1`, `test_protected_ops_2` y `test_protected_ops_3`.

- `test_protected_ops_1` es un ejemplo sencillo que muestra cómo un procedimiento ejecutado mediante `Protected_Call` no es interrumpido por otros procedimientos cuya ejecución se encargó al sistema mediante una llamada a `Program_Timer_Procedure`.
- El ejemplo de la carpeta `test_protected_ops_2` ilustra el problema que puede ocurrir cuando varios hilos están ejecutando concurrentemente código que accede a una misma estructura de datos, ya sea para consultarla o para modificarla. En el ejemplo el programa principal comienza encargando al sistema que 5000ms después se ejecute el procedimiento `Procedures.Timed_Procedure`. Llegado ese momento el sistema creará un nuevo hilo de ejecución para llamar a `Procedures.Timed_Procedure`, cuyo código borrará un elemento del Map.

  Pero antes de que llegue ese momento, el programa principal prosigue su ejecución, insertando 4 elementos en un Map.

  Por último el programa principal llama a `Print_Map`, y mientras que dentro de este procedimiento el cursor está situado sobre el nodo "www.urjc.es" (su puntero apunta a la celda de ese nodo dentro de Map), llega el instante en el que el sistema tiene que ejecutar `Procedures.Timed_Procedure`. Este procedimiento borra precisamente esa celda apuntada por el cursor[3].

  El hilo que ha ejecutado el procedimiento `Procedures.Timed_Procedure` se ha ejecutado mientras que el hilo del programa principal estaba ejecutando `Print_Map`. La operación de borrar provoca que el cursor que está usando `Print_Map` para recorrer el Map quede inconsistente, pues el puntero del cursor apunta a un elemento que ha sido borrado. Por ello el programa falla con una excepción.
- El ejemplo de la carpeta `test_protected_ops_3` ilustra cómo simplemente protegiendo la ejecución de `Print_Map` haciendo que se llame a través de `Protected_Ops.Protected_Call` se consigue que el hilo del programa principal que está ejecutando `Print_Map` no pueda ser interrumpido mientras que está recorriendo el Map por el hilo del programa principal que intenta borrar un elemento. Por ello en este caso el programa termina sin elevar una excepción, ya que no se borra el elemento del Map hasta que ha concluído su ejecución `Print_Map`.

2.5. Protocolos para la recuperación de mensajes perdidos y entrega ordenada

2.5.1. Recuperación de mensajes perdidos Init y Welcome

Para recuperarse de las pérdidas del mensaje `Init` el cliente utilizará un protocolo de parada y espera: el mensaje `Init` se retransmite hasta que se recibe su correspondiente mensaje `Welcome`, que de este modo actúa como si fuera un asentimiento del mensaje `Init`.

[3] Para provocar el problema se fuerza mediante delay el que el hilo del programa principal que está ejecutando Print_Map tarde en realizar el recorrido, para así dar tiempo a que se ejecute el hilo que borra un elemento del Map.

2.5.2. Recuperación de mensajes perdidos Writer, Server y Logout

Para recuperarse de las pérdidas de los mensajes `Writer` y `Logout` el cliente utilizará un protocolo de envío continuo sin ventana. También el servidor, para recuperarse de la pérdida de mensajes `Server`, utilizará un protocolo de envío contínuo sin ventana.

Estos mensajes se retransmitirán hasta que sean asentidos por sus destinatarios. Cada uno de estos mensajes llevará un número de secuencia, debiendo ser asentido por sus receptores mediante un nuevo tipo de mensaje `Ack` que identifica el mensaje que asiente.

Para que los receptores puedan entregar los mensajes recibidos en el orden en el que fueron enviados, los mensajes que no se reciban en orden no se asentirán. En el cliente, hasta no haberse recibido el mensaje `Welcome` se ignorarán los mensajes `Server` (ni se enviará su asentimiento ni se procesará el mensaje, ignorándolo como si no se hubiera recibido). Los mensajes `Writer`, `Server` y `Logout` sólo se procesarán una vez, en el orden de envío marcado por su número de secuencia.

Para la recuperación de pérdidas de estos mensajes se utilizará un protocolo de envío continuo sin ventana, con mensajes de asentimiento.

Los mensajes `Writer` y `Logout` enviados por el cliente, y los mensajes `Server` enviados por el servidor, han de ser asentidos por sus receptores mediante un nuevo tipo de mensajes de asentimiento. Por ello en esta práctica se utilizará un nuevo tipo de mensaje adicional a los especificados en la práctica anterior: **Mensaje de tipo `Ack`**. El tipo enumerado para indicar el tipo de mensaje se redefinirá de la siguiente forma:

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

Los mensajes `Ack` se reciben en el manejador de recepción de mensajes. Hasta que no se reciba el correspondiente asentimiento hay que retransmitir los mensajes `Writer`, `Server` y `Logout`, una vez haya vencido su plazo de retransmisión.

Para poder reconocer qué mensaje asiente un mensaje `Ack`, el formato de los mensajes `Writer`, `Logout` y `Server` cambia en esta práctica respecto a la anterior: se añade detrás del *End_Point* del emisor de un mensaje `Writer` o `Logout` un número de secuencia, y en los mensajes `Server` se añade el *End_Point* del servidor y un número de secuencia.

El cliente utiliza un único espacio de números de secuencia para los mensajes que envía, ya sean `Writer` o `Logout`. El número de secuencia del mensaje `Logout` será una unidad mayor que el del último de los mensajes `Writer` enviados por ese cliente.

El servidor utiliza un espacio de números de secuencia distinto para cada cliente, de forma que el mensaje con número de secuencia `3` enviado a un cliente es distinto al mensaje con número de secuencia `3` enviado a otro, aunque sus contenidos puedan ser los mismos. De esta forma el servidor podrá identificar correctamente qué clientes sí han asentido y cuáles no cada uno de los mensajes enviados.

Para implementar los **números de secuencia** de los mensajes se utilizará el siguiente **tipo modular de datos**:

```
type Seq_N_T is mod Integer'Last;
```

El primer valor del tipo `Seq_N_T` es `0` y el último `Integer'Last - 1`.

Una característica de estos tipos de datos modulares de Ada es que tienen aritmética modular:

- `Seq_N_T'Last + 1 = Seq_N_T'First`
- `Seq_N_T'First - 1 = Seq_N_T'Last`

2.5.3. Formatos de los mensajes

El formato de los mensajes `Init` y `Welcome` es el mismo que el utilizado en la práctica 3.

Sin embargo sí se modifica el formato de los mensajes `Writer`, `Server` y `Logout` respecto al que tenían en la práctica 3. A continuación se muestra el nuevo formato de estos tres mensajes (en rojo aparecen los cambios respecto a la práctica 3). Finalmente se muestra el formato del nuevo mensaje `Ack`:

Mensaje `Writer`

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

Writer	Client_EP_Handler	Seq_N	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` y `Ack` en este `End_Point`.

- Seq_N: Valor del tipo Seq_N_T. Número de secuencia de los mensajes enviados por el cliente (ya sean Writer o Logout).
- 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

Mensaje Server

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

Server	Server_EP_Handler	Seq_N	Nick	Comentario

en donde:

- Server: Message_Type que identifica el tipo de mensaje.
- Server_EP_Handler: End_Point del servidor, que es el que envía el mensaje. El servidor recibe mensajes Init, Writer, Logout y Ack en este End_Point.
- Seq_N: Valor del tipo Seq_N_T. Número de secuencia de los mensajes enviados por el servidor. El servidor utilizará espacios de números de secuencia distintos para cada cliente distinto, por lo que deberá guardar cuál es el último número de secuencia que ha enviado a cada cliente.
- 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	Seq_N	Nick

en donde:

- Logout: Message_Type que identifica el tipo de mensaje.
- Client_EP_Handler: End_Point del cliente.
- Seq_N: Valor del tipo Seq_N_T. Número de secuencia de los mensajes enviados por el cliente (ya sean Writer o Logout).
- Nick: Unbounded_String con el *nick* del cliente que envía el Logout al servidor.

Mensaje Ack

Estos mensajes los envía el programa cliente para asentir mensajes Server enviado por el servidor, y los envía el programa servidor para asentir mensajes Writer y Logout enviados por los clientes . El destinatario de un mensaje de tipo Ack es el programa que envió el mensaje que se desea asentir.
Formato:

Ack	EP_H_ACKer	Seq_N

en donde:

- Ack: Valor del tipo Message_Type que identifica el tipo de mensaje.
- EP_H_ACKer: EP_H del proceso que envia el asentimiento.
- Seq_N: Valor del tipo Seq_N_T. Número de secuencia que tenía el mensaje recibido que se asiente.

2.6. Implementación de los protocolos

En esta sección se proporcionan detalles sobre el modo en el que se deben implementar los protocolos de recuperación de mensajes perdidos y entrega ordenada.

Cuando el cliente envía un mensaje `Init` espera recibir un mensaje `Welcome` utilizando recepción bloqueante mediante `LLU.Receive`. Dado que en esta práctica se pueden perder los mensajes, si tras el plazo de espera especificado en `LLU.Receive` el servidor no contesta, el cliente reintentará el envío del mensaje `Init`. Si tras un máximo número de reintentos no se recibe respuesta[4], el cliente terminará mostrando un mensaje que describa la imposibilidad de contactar con el servidor. El mensaje `Welcome` enviado por el servidor no se asiente, del mismo modo que un mensaje de asentimiento `Ack` no se asiente. Si se pierde, el cliente retransmitirá el mensaje `Init`. Por ello siempre que el servidor reciba un mensaje `Init` debe contestar con el mensaje `Welcome` adecuado: si el mismo cliente ya está en la tabla de símbolos de clientes activos, con el mismo nick y el mismo *End_Point*, se le acepta. Si no está, se le añade y se le acepta. Si está con el mismo nick y distinto *End_Point*, se le rechaza.

Para implementar el sistema de reenvíos de mensajes `Writer`, `Server` y `Logout` hay que guardar los mensajes que se han enviado (o reenviado) y aún no han sido asentidos, y la información sobre a qué hora hay que retransmitir cada uno de esos mensajes.

La ejecución del código que comprueba si hay que retransmitir un mensaje una vez que ha pasado su plazo de retransmisión se realizará mediante el paquete `Protected_Ops` que permite programar la ejecución en un instante futuro de un **procedimiento de gestión de retransmisiones**. Llegado ese instante el procedimiento se ejecutará concurrentemente con el resto de hilos de la aplicación (programa principal y ejecución de manejadores de recepción asíncrona de mensajes). Este mismo paquete ofrece un servicio para poder controlar el acceso concurrente a las estructuras de datos compartidas por los hilos de la aplicación.

2.6.1. Acciones para la recuperación de mensajes perdidos y entrega ordenada

Hay varios momentos en los que un programa tiene que llevar a cabo acciones relacionadas con la recuperación de mensajes perdidos:

- Cuando se envía por primera vez o se retransmite un mensaje `Writer`, `Server` o `Logout`:
  1. Hay que guardar información relativa a este envío para que en caso de no recibirse el correspondiente `Ack` pueda retransmitirse el mensaje una vez haya pasado su plazo de retransmisión.
  2. Hay que comprobar si es necesario replanificar la ejecución del procedimiento de gestión de retransmisiones. En caso de que ya esté programada su ejecución para un instante anterior al momento en el que hay que retransmitir este mensaje, no hay que hacer nada, pues el propio procedimiento de gestión de retransmisiones se va reprogramando cada vez que se ejecuta.
- Cuando llega un mensaje `Ack` al manejador de recepción de mensajes: hay que eliminar la información relativa al mensaje que está siendo asentido pues ya no habrá que retransmitirlo.
- Cuando llega un mensaje que no es un `Ack` al manejador de recepción de mensajes:
  1. Si el servidor recibe un mensaje `Init` tiene que contestar con el mensaje `Welcome` adecuado[5].
  2. Cuando un proceso recibe un mensaje `Writer`, `Server` o `Logout`:
     - Si su número de secuencia no es igual o menor que el esperado para ese emisor: no debe procesarlo ni asentirlo.
     - Si el número de secuencia es igual al esperado para ese emisor, debe procesarlo normalmente, y debe enviar el correspondiente mensaje `Ack`.
     - Si el número de secuencia es menor que el esperado, se trata de una retransmisión de un mensaje que ya se recibió, procesó, y asintió. Si lo han retransmitido será porque el asentimiento no llegó antes de que venciera el plazo de retransmisión. Por tanto, en este caso no hay que procesar el mensaje, pero sí hay que enviar el `Ack` que corresponda.
- Cuando llega el momento en el que el sistema llama al procedimiento de gestión de retransmisiones, cuya ejecución se encargó mediante una llamada a `Protected_Ops.Program_Timer_Procedure`:
  1. El código del procedimiento tiene que retransmitir todos los mensajes cuya hora de retransmisión sea anterior al momento en el que se ejecuta el procedimiento.
  2. Lo último que tiene que hacer el procedimiento de gestión de retransmisiones es comprobar si hay mensajes pendientes de ser asentidos, y si es así, volver a llamar a `Protected_Ops.Program_Timer_Procedure` para encargar la ejecución de nuevo del procedimiento de gestión de retransmisiones a la primera de las horas a las que venza un plazo de retransmisión.

[4] Ver apartado 2.7 para ver cómo se establece el máximo número de retransmisiones

[5] Nótese que el mensaje `Welcome` se envía al `Client_EP_Receive`

2.6.2. Estructuras de datos para almacenar la información necesaria para realizar retransmisiones de mensajes Writer, Server y Logout no asentidos

***Pending_Msgs*: Colección de mensajes pendientes de asentimiento**

Cada vez que se envía un nuevo mensaje de tipo `Writer`, `Server` o `Logout` se debe guardar información sobre el mensaje en una colección de mensajes enviados que aún no han sido asentidos. Llamaremos a esta colección *Pending_Msgs*. El cliente mantendrá una colección para los mensajes `Writer` y `Logout` pendientes de ser asentidos y el servidor otra para los mensajes `Server` pendientes de ser asentidos.

Cuando se recibe un `Ack`, en caso de que sea el esperado se podrá eliminar la información asociada al mensaje asentido de *Pending_Msgs*.

Es aconsejable utilizar una tabla de símbolos o *Map* para implementar en el cliente y en el servidor las respectivas estructuras de datos *Pending_Msgs*.

Cuando se envíe un mensaje habrá que añadir una entrada a *Pending_Msgs*, y cuando llegue un `Ack` habrá que borrar el mensaje asentido. Tanto para añadir una entrada de *Pending_Msgs* como para borrarla parece lógico que la clave de esta tabla de símbolos esté compuesta por una 3-tupla con los siguientes valores: *End_Point* origen del mensaje asentido, *End_Point* destino del mensaje asentido, y su número de secuencia. Esta 3-tupla de valores permite identificar de manera única tanto los mensajes enviados por el cliente al servidor, como los mensajes enviados por el servidor a cada cliente. Téngase en cuenta que en este segundo caso, el servidor utiliza espacios de números de secuencia distintos para cada uno de los clientes.

Para cada entrada almacenada en *Pending_Msgs* hay que almacenar como valor el conjunto de campos del mensaje que permitan componer de nuevo el mensaje a reenviar. Como los mensajes `Writer` y `Server` llevan como campos un *nick* y un comentario, el valor puede almacenar un registro con el tipo de mensaje y dos unbounded string. En el caso del mensaje `Logout` se desaprovecharía el campo de comentario pues sólo tiene campo *nick*.

***Retransmission_Times*: Colección de horas de retransmisión y mensajes a retransmitir**

Cuando el procedimiento de gestión de retransmisiones es ejecutado por el sistema tiene que reenviar todos los mensajes cuya hora de retransmisión ya ha llegado.

Tal como se ha definido en la sección 2.6.2, *Pending_Msgs* es una tabla de símbolos cuya clave es la 3-tupla de *End_Point* origen, *End_Point* destino y número de secuencia. Podría almacenarse la hora de retransmisión en el valor de cada elemento de *Pending_Msgs*. Pero en ese caso, cuando el procedimiento de gestión de retransmisión se ejecutase tendría que recorrer todos los elementos de *Pending_Msgs* para ver si su hora de retransmisión ya ha llegado.

En lugar de hacerlo así en la práctica se utilizará una nueva colección denominada *Retransmission_Times*, cuyos elementos sean 2-tuplas (hora de retransmisión, identificador de mensaje).

Si al añadir elementos a *Retransmission_Times* se insertan en la posición que les corresponde según el orden marcado por las horas de retransmisión, el procedimiento de gestión de retransmisiones podrá acceder rápidamente a los elementos cuya hora de retransmisión haya llegado: bastará con que vaya consultando cada elemento desde el primero, hasta que encuentre uno cuya hora de retransmisión sea superior a la hora en la que se realiza la consulta. Los elementos con hora menor que la hora de la consulta se deberán eliminar de *Retransmission_Times*. A continuación se pueden buscar en *Pending_Msgs* los identificadores de los elementos eliminados de *Retransmission_Times*, para así poder enviar sus correspondientes mensajes allí almacenados.

Para insertar un elemento en *Retransmission_Times* habrá que buscar el lugar que le corresponde según el orden de las horas. Nótese que, a diferencia de una tabla de símbolos, en *Retransmission_Times* puede haber dos elementos con la misma hora.

Al no poder ser una tabla de símbolos, **el alumno tendrá que realizar el diseño de la interfaz de *Retransmission_Times***, que no tiene por qué ser genérica, aunque sí ha de ser un tipo abstracto de datos.

La estructura de cada elemento almacenado por *Retransmission_Times* está clara: 2-tuplas (hora de retransmisión, identificador de mensaje).

La interfaz de *Retransmission_Times* deberá permitir:

- Añadir un elemento, que deberá insertarse en la posición adecuada según su hora de retransmisión, pudiéndo haber varios con la misma hora de reenvío.
- Consultar el valor del primero de los elementos (el de hora de retransmisión menor).
- Eliminar el primero de los elementos (el de hora de retransmisión menor).

De la anterior explicación se deduce que la estructura de datos que hay que diseñar para implementar *Retransmission_Times* no es ni una tabla de símbolos, ni una pila, ni una cola.

Resumiendo: para implementar el sistema de reenvíos y asentimientos se utilizarán dos estructuras de datos que almacenen, por un lado, los mensajes que se han enviado (o reenviado) y aún no han sido asentidos, y por otro, la información sobre a qué hora

hay que retransmitir cada uno de esos mensajes. La primera de estas estructuras de datos, denominada *Pending_Msgs*, será una tabla de símbolos (*Map*). La otra estructura de datos, denominada *Retransmission_Times* deberá ser diseñada específicamente para esta práctica en función del uso que se hará de ella.

2.7. Condiciones de Funcionamiento

1. 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 según se explica en la «Parte I» de este enunciado.
2. Para implementar el sistema de reenvíos y asentimientos se utilizarán dos estructuras de datos que almacenen, por un lado, los mensajes que se han enviado (o reenviado) y aún no han sido asentidos, y por otro, la información sobre a qué hora hay que retransmitir cada uno de esos mensajes. La primera de estas estructuras de datos, denominada *Pending_Msgs*, será una tabla de símbolos (*Map*). La otra estructura de datos, denominada *Retransmission_Times* deberá ser diseñada específicamente para esta práctica, tanto su interfaz como su implementación, en función del uso que se hará de ella.
3. El cliente y el servidor deberán tener la interfaz en línea de comandos descrita en la sección 2.3, debiendo utilizarse los procedimientos para la inyección de fallos y retardos de propagación descrita en la sección 2.2.
4. Se supondrá que los mensajes enviados por el servidor y por los clientes se pueden perder. Por ello hay que implementar un protocolo de parada y espera para recuperar las pérdidas de los mensajes `Init`, y un protocolo de envío continuo sin ventana para recuperar las pérdidas y entregar en orden los mensajes `Writer`, `Server` y `Logout`.
5. Para realizar las retransmisiones de mensajes `Writer`, `Server` y `Logout` se utilizarán los servicios del paquete `Protected_Ops`.
6. La retransmisión de mensajes `Writer`, `Server` y `Logout` se realizará en un procedimiento de gestión de retransmisiones cuya ejecución se encargará al sistema mediante llamadas a `Protected_Ops.Program_Timer`.
7. El número `MAX` de retransmisiones de un mismo mensaje será función del porcentaje de pérdidas, según la siguiente fórmula: $\texttt{MAX} = 10 + (\%pérdidas/10)^2$. Así, el número máximo de retransmisiones resultará:

%pérdidas	`MAX`
10	11
20	14
30	19
...	...

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

2.8. Extensiones

El alumno que lo desee podrá implementar la tabla de símbolos que se utiliza para *Pending_Messages* mediante una tabla hash con resolución de colisiones mediante direccionamiento abierto, con borrado perezoso (alternativa III).

3. Entrega

La entrega de esta práctica se hará a través del aula virtual, con límite: **Miércoles 10 de enero a las 23:55h**.
La Prueba de Evaluación L4 de esta práctica se hará el **Jueves 11 a las 11h00**.