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

Departamento de Sistemas Telemáticos y Computación
(GSyC)

Noviembre de 2013

Resumen

En esta práctica debes realizar un nuevo programa en Ada, **Chat-Peer v2.0**, que mejore la versión 1.0 del mismo para que tolere las pérdidas y desorden de los mensajes que intercambian los nodos.

Para recuperarse de las pérdidas de los mensajes que un nodo envíe por inundación a sus vecinos, cada mensaje de inundación deberá ser asentido por cada uno de los vecinos que lo reciban.

Para recuperarse del desorden de los mensajes, no se asentirán los mensajes que lleguen fuera de secuencia, aunque sí se reenviarán por inundación.

Para implementar el sistema de reenvíos y asentimientos se utilizará una tabla de símbolos implementada mediante un árbol de búsqueda binaria que almacenará los mensajes pendientes de asentimiento. La implementación de esta tabla de símbolos utilizará un paquete genérico que se proporciona, por lo que el alumno sólo tendrá que realizar la instanciación.

En este enunciado se describe en primer lugar en la sección 1 la interfaz que tendrá el programa. A continuación en la sección 2 se explica cómo provocar fallos y retardos de propagación simulados cuando se envían mensajes con Lower Layer. Después se describen los *manejadores temporizados* en la sección 3, nuevo tipo de procedimientos manejadores que se utilizarán para programar la ejecución futura de procedimientos. En la sección 4 se dan pautas sobre cómo se debe llevar a cabo la implementación de la transmisión fiable de mensajes, objetivo principal de esta práctica. Este enunciado concluye exponiendo las condiciones de funcionamiento en la sección 6 y finalmente las normas de entrega de la práctica, en la sección 7.

1. Interfaz de usuario del programa `chat_peer_2.adb`

El programa de cada nodo se lanzará pasándole 5, 7 o 9 argumentos en la línea de comandos:

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

Los 5 primeros argumentos son obligatorios:

- `port`: Número del puerto en el que el nodo recibirá los mensajes enviados por inundación, utilizando un *handler* de Lower_Layer_UDP (LLU).
- `nickname`: Apodo del nodo. Cualquier cadena de caracteres será un apodo válido.
- `min_delay`: Retardo mínimo de propagación que sufrirán los envíos que haga este nodo (ver apartado 2), expresado como un número natural de **milisegundos**.
- `max_delay`: Retardo máximo de propagación que sufrirán los envíos que haga este nodo (ver apartado 2), expresado como un número natural de **milisegundos**. Debe ser mayor o igual que `min_delay`.
- `fault_pct`: Porcentaje de mensajes que se perderán de entre los envíos que haga este nodo (ver apartado 2), expresado como un número natural entre 0 y 100.

Opcionalmente, el nodo puede ser lanzado pasándole uno o dos nodos vecinos de contacto en la línea de argumentos. Cada uno de ellos se especifica mediante:

- `nb_host`: Nombre de la máquina en la que está el nodo vecino de contacto
- `nb_port`: Número del puerto en el que escucha el nodo vecino los mensajes enviados por inundación mediante un *handler* de LLU.

El nodo ofrecerá la misma interfaz que la especificada en el enunciado de la práctica anterior, permitiendo al usuario introducir cadenas de caracteres para enviar a otros nodos, y mostrando las cadenas enviadas desde ellos.

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 nodos 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, para poder comprobar que Chat-Peer v2.0 tolera pérdidas de mensajes, `Lower_Layer_UDP` puede simular dichas pérdidas.

2.1. Simulación de las pérdidas de paquetes

El procedimiento `Set_Faults_Percent`, invocado al principio del programa, provocará que se pierda un porcentaje de todos los envíos que se realicen. Dicho porcentaje se especifica en la llamada al subprograma, expresado como un argumento de tipo `Integer` con un valor 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);
```

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 al principio del programa principal, y afecta a todos los envíos que hace el mismo, incluidos los envíos realizados desde el manejador.

2.2. Simulación de los retardos de propagación

Chat-Peer v2.0 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.

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

2.3. Plazo de retransmisión

La presencia de pérdidas y desorden de mensajes requiere el envío de mensajes *Ack* para asentir los mensajes recibidos.

El nodo que envía un mensaje a un destino tendrá que **retransmitir** dicho envío si no recibe su *Ack* 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 nodo 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 conocer el retardo máximo de propagación de los mensajes. Pero dado que en esta práctica se conoce el retardo máximo de propagación por estar especificado en el argumento de la línea de comandos `max_delay`, 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;
```

3. Manejadores Temporizados

Para programar el envío futuro de las retransmisiones de mensajes en el instante en el que venza su plazo de retransmisión se utilizará el paquete `Timed_Handlers`, que permite planificar la ejecución futura de un determinado procedimiento a una hora concreta.

La especificación del paquete `Timed_Handlers` es la siguiente:

```
with Ada.Calendar;

package Timed_Handlers is

   type Timed_Handler_A is access procedure (Time: Ada.Calendar.Time);
   procedure Set_Timed_Handler (T : Ada.Calendar.Time; H : Timed_Handler_A);

   procedure Finalize;

end Timed_Handlers;
```

Mediante la llamada al procedimiento `Timed_Handlers.Set_Timed_Handler` se puede programar la ejecución de un *procedimiento manejador temporizado* para que se ejecute en un instante futuro. El manejador temporizado y el instante en el que se ha de ejecutar se le pasan como argumentos `H` y `T` respectivamente al procedimiento `Timed_Handlers.Set_Timed_Handler`.

El sistema llamará a este procedimiento manejador temporizado cuando se cumpla la hora para la que fue programado. El procedimiento manejador, cuando llegue la hora, se ejecutará por tanto de manera concurrente al resto de hilos de ejecución del programa.

Cuando llegue su hora, el procedimiento manejador será llamado por el sistema, pasándole en la llamada como parámetro el valor del instante de tiempo en el que debía ejecutarse.

En esta práctica, cada vez que un nodo cree o reenvíe un mensaje por inundación a sus vecinos, programará para el futuro (a una hora igual a la hora de envío + el plazo de retransmisión) la ejecución de un procedimiento manejador temporizado que se encargará de retransmitir ese mensaje a los vecinos de los que aún no haya recibido el *Ack*.

El procedimiento `Timed_Handlers.Finalize` permite desactivar los hilos de ejecución internos al paquete `Timed_Handlers`, lo que es necesario hacer antes de que el programa principal pueda terminar la ejecución. Así, para poder terminar completamente su ejecución, el programa principal de Chat-Peer v2.0 deberá llamar tanto a `Lower_Layer_UDP.Finalize` como a `Timed_Handlers.Finalize`.

No hay que estudiar el código del paquete `Timed_Handlers`, pero sí el código de ejemplo `timed_handlers_test.adb` que te proporcionamos para ver cómo se utiliza este paquete para instalar procedimientos manejadores temporizados.

En el resto de este enunciado utilizaremos el término *manejador de recepción* para referirnos al procedimiento manejador de `Lower_Layer_UDP` que se encarga del procesamiento de mensajes recibidos en un `End_Point`, y utilizaremos el término *manejador temporizado* para referirnos a este nuevo tipo de procedimiento manejador que se ha introducido en esta sección.

4. Implementación

4.1. Tipos de mensaje utilizados en esta práctica

El formato de los mensajes *Init*, *Reject*, *Confirm*, *Writer* y *Logout* no cambia con respecto al definido en la práctica anterior. En esta práctica se utilizará un nuevo tipo de mensaje adicional a los especificados en la práctica anterior: **Mensaje de tipo *Ack***. Por ello ahora el enumerado para indicar el tipo de mensaje se redefinirá de la siguiente forma:

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

Mensaje *Ack*

Es el que envía un nodo para asentir un mensaje al vecino que se lo reenvía. El mensaje *Ack* NO se envía por inundación. El destinatario de este mensaje es el `EP_Hdl_Rsnd` del mensaje recibido que se asiente.
Formato:

Ack	EP_H_ACKer	EP_H_Creat	Seq_N

en donde:

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

4.2. Fases

Cada nodo de Chat-Peer v2.0 deberá ejecutar en orden las mismas fases que se detallan en el enunciado de la práctica anterior: Inicialización, protocolo de admisión, protocolo de envío/recepción de mensajes *Writer* y protocolo de salida. Todo lo especificado en el enunciado de la práctica anterior en relación a las fases se mantiene igual, salvo los cambios que aparezcen mencionados a continuación en este apartado.

4.3. Protocolo de admisión

El protocolo de admisión se mantiene igual que en la práctica anterior, teniendo en cuenta lo siguiente:

- Los mensajes *Init*, *Confirm* y *Logout* se deben transmitir ahora de forma fiable: pasado el *plazo de retransmisión* después del envío o reenvío de uno de estos mensajes, se debe retransmitir a los vecinos de los que no se haya recibido aún el *Ack*.
- El mensaje *Reject* sigue transmitiéndose sólo a un nodo, no por inundación. El mensaje *Reject* NO hay que transmitirlo de manera fiable en esta parte básica de la práctica: si se pierde no será retransmitido. Esto hace que la detección de apodos (*nicknames*) duplicados no funcione demasiado bien en la Parte Básica de Chat-Peer v2.0 ya que los mensajes *Reject*, al igual que el resto, pueden no llegar a su destino. Por tanto en ocasiones podrán estar dentro del chat varios nodos con el mismo apodo[1].

4.4. Protocolo de envío y recepción de mensajes *Writer*

Este protocolo se mantiene igual que en la práctica anterior, teniendo en cuenta que ahora los mensajes *Writer* se transmiten de forma fiable, es decir, pasado el *plazo de retransmisión* después del envío o reenvío de uno de estos mensajes, se debe retransmitir a los vecinos de los que no se haya recibido aún el *Ack*.

[1] Uno de los Módulos Opcionales de esta práctica consistirá en realizar la transmisión de los mensajes *Reject* de manera fiable para arreglar este problema

4.5. Protocolo de salida

Este protocolo se mantiene igual que en la práctica anterior, teniendo en cuenta que ahora los mensajes *Logout* se transmiten de forma fiable, es decir, pasado el *plazo de retransmisión* después del envío o reenvío de uno de estos mensajes, se debe retransmitir a los vecinos de los que no se haya recibido aún el *Ack*.

Hay que tener en cuenta que el nodo no debería terminar su ejecución hasta asegurarse de que ha recibido el asentimento de su mensaje *Logout* por parte de todos sus vecinos. Para ello puede esperar con un `delay` en el programa principal un tiempo igual a 10 veces el plazo de retransmisión, que sería lo que tardaría en retransmitir el *Logout* a todos sus vecinos en el caso peor[2].

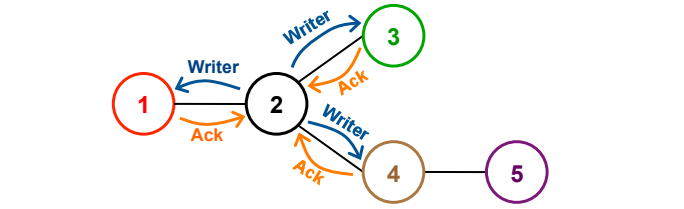
4.6. Protocolo de inundación controlada

El protocolo de envío de los mensajes *Init*, *Confirm*, *Writer* y *Logout* mediante inundación controlada se mantiene como en la práctica anterior con las modificaciones que aquí se detallan para protegerse de las pérdidas y el desorden de los mensajes.

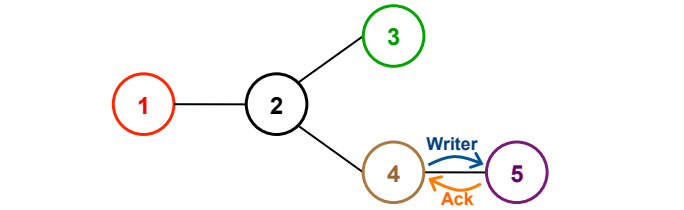
4.6.1. Mensajes *Ack*

Cada vez que un nodo **cree y envíe por inundación** o **reenvíe por inundación** un mensaje *Init*, *Confirm*, *Writer* o *Logout* a su lista de vecinos deberá recibir un *Ack* de cada uno de ellos.

A modo de ejemplo, la figura 1 muestra la progresión de un mensaje *Writer* creado por el nodo 2 de una red y enviado por inundación a todos sus vecinos. Los nodos que van recibiendo el *Writer* van contestando con un *Ack* al nodo que se lo reenvía.



(a) En primer lugar el nodo 2 **crea** un mensaje Writer y lo envía por inundación a sus vecinos, que lo asienten



(b) Posteriormente el nodo 4 **reenvía** el mensaje Writer a sus vecinos (menos al nodo 2), que lo asienten

Figura 1: Envío de un mensaje por inundación y de sus asentimientos

Si pasado el *plazo de retransmisión* desde que un nodo envío o reenvió un mensaje, aún no ha recibido el *Ack* de alguno de sus vecinos, les retransmitirá a éstos dicho mensaje. Pasado de nuevo el *plazo de retransmisión* se volverá a retransmitir otra vez el mensaje a los vecinos de los que siga sin recibirse el *Ack*. Y así sucesivamente hasta un máximo de 10 retransmisiones del mismo mensaje al mismo vecino. Pasadas las 10 retransmisiones a un mismo vecino se desistirá de volver a reenviar ese mensaje a ese vecino.

En la figura 2 puedes ver un ejemplo de cómo se relacionan los campos de un mensaje *Writer* enviado por inundación con los campos de sus asentimientos.

4.6.2. Estructuras de datos

El protocolo de inundación necesita almacenar los mensajes que ha enviado y que aún no ha asentido para poder retransmitirlos cuando se cumpla su plazo de retransmisión. Por ello, además de utilizar las tablas de símbolos *neighbors* y *latest_msgs* de la práctica anterior, se deberán utilizar dos nuevas tablas de símbolos: *Sender_Buffering* y *Sender_Dests*.

[2]Uno de los Módulos Opcionales consistirá en mejorar el protocolo de salida para solventar los problemas que presenta en la Parte Básica de esta práctica

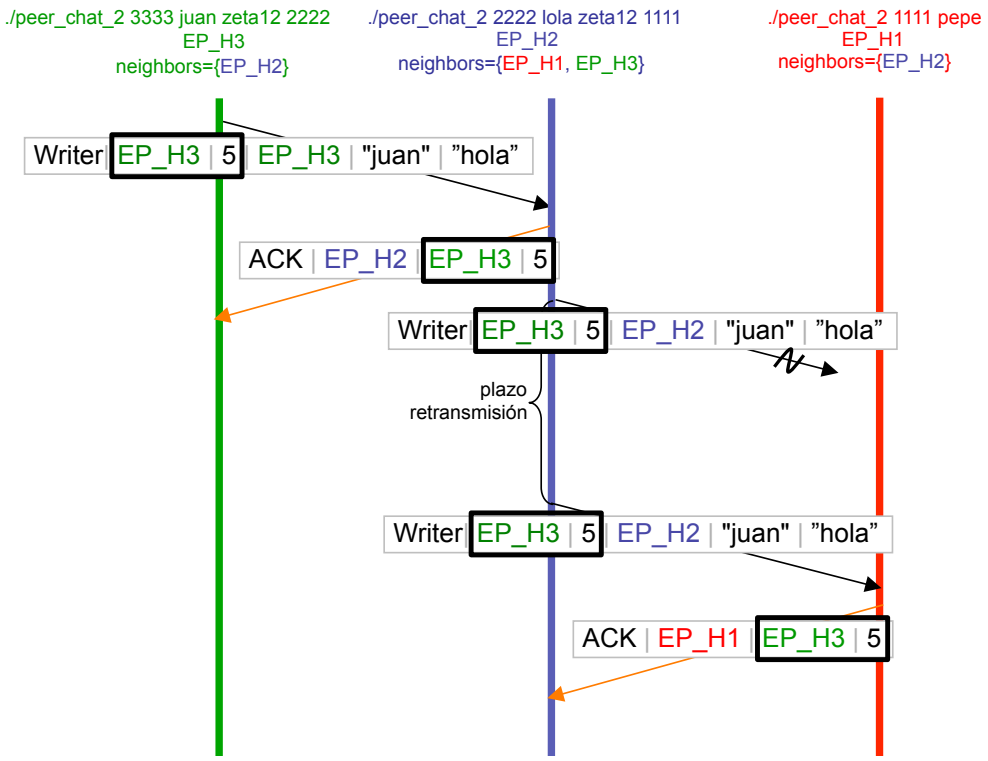


Figura 2: Relación entre los campos de los mensajes y sus asentimientos

Sender_Buffering

Se necesita una tabla de símbolos para almacenar los mensajes pendientes de ser asentidos. Los mensajes se guardan en esta tabla para ser retransmitidos en un instante del futuro. Llegado el momento de su retransmisión, habrá que buscar el mensaje en esta tabla. Por ello, se utilizará la hora de retransmisión de cada mensaje como su clave en la tabla.

Habrá que acceder a esta tabla con frecuencia, cada vez que venza el plazo de retransmisión de un mensaje no asentido aún. Por ello se utilizará una implementación basada en un Árbol de Búsqueda Binaria (ABB) que haga más rápidas las búsquedas.

Cada elemento de *Sender_Buffering* almacena una pareja `(Clave, Valor)` donde:

- `Clave`: Hora de retransmisión del mensaje.
- `Valor`: Registro con los siguientes campos, relativos a un mensaje enviado y pendiente de ser asentido:
  - `EP_H_Creat`: `EP_H` del nodo que creó el mensaje
  - `Seq_N`: número de secuencia del mensaje
  - `P_Buffer`: Puntero al Buffer relleno con el mensaje enviado y pendiente de ser asentido. Este es el buffer que se reenviará en caso de que venza el plazo de retransmisión.

En la figura 3 puede observarse un ejemplo de *Sender_Buffering* para el nodo 2 de una cierta red de nodos en ejecución:

- Cada elemento del árbol es un mensaje creado o reenviado por inundación por el nodo 2, y que aún está pendiente de ser asentido por algún vecino.
- En cada elemento está guardado un puntero al *buffer* que contiene el mensaje que se ha enviado a los vecinos por inundación, para poder reenviar dicho mensaje si fuera necesario.
- El campo clave (con fondo azul) de cada elemento del árbol contiene la hora a la que debe retransmitirse el *buffer* a los vecinos que aún no hayan enviado su *Ack*. Como se trata de un ABB, los elementos están ordenados en el árbol, quedando a la izquierda de un elemento otros elementos con hora de retransmisión anterior a la suya, y a la derecha otros elementos con hora de retransmisión posterior a la suya.

- En cada elemento se almacena el `EP_H_Creat` y el `Seq_N` que identifican al mensaje enviado o reenviado por el nodo 2.

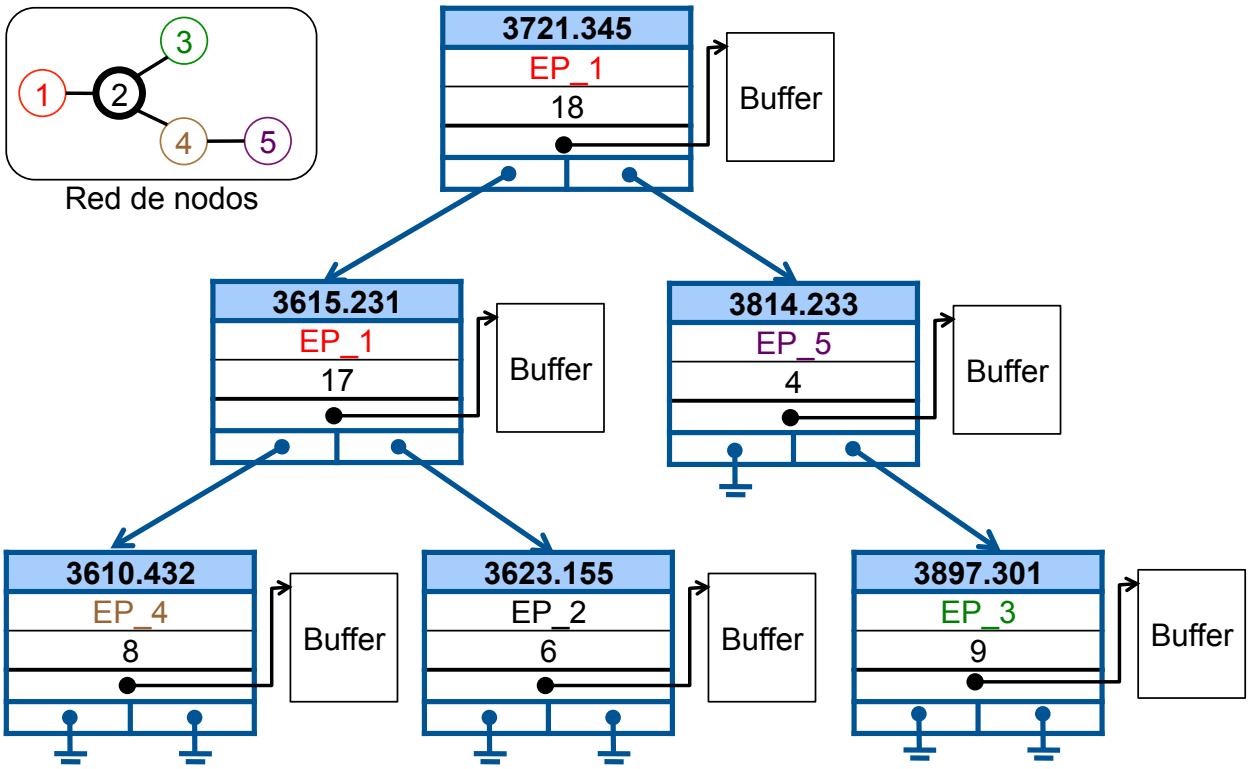


Figura 3: Ejemplo de un posible *Sender_Buffering* para el nodo 2 de la red de nodos de la esquina superior izquierda

Sender_Dests

Para ir marcando qué vecinos van asintiendo cada mensaje que se envía por inundación se necesita otra tabla de símbolos en la que se almacene qué vecinos van asintiendo cada mensaje. Cuando llegue un *Ack* de un vecino, se buscará en esta tabla el mensaje al que asiente para marcar que dicho vecino ya lo ha asentido. Cuando ya hayan llegado los asentimientos de todos los vecinos para un determinado mensaje (o cuando se haya alcanzado el máximo de retransmisiones para aquellos que aún están pendientes) se eliminará tanto de *Sender_Dests* como de *Sender_Buffering* la entrada que representa al mensaje.

Habrá que acceder a esta tabla con frecuencia, cada vez que llegue un mensaje de *Ack*. Por ello se utilizará también para esta tabla de símbolos una implementación basada en un Árbol de Búsqueda Binaria (ABB) que haga más rápidas las búsquedas.

Cada elemento de *Sender_Dests* almacena una pareja `(Clave, Valor)` donde:

- `Clave`: Registro con los siguientes campos que identifican un mensaje enviado:
  - `EP_H_Creat`: `EP_H` del nodo que creó el mensaje
  - `Seq_N`: número de secuencia del mensaje
- `Valor`: `Destinations`: Array de registros con la información de los vecinos destinatarios del mensaje. Este array tendrá 10 posiciones, el número máximo de vecinos que puede tener un nodo. Cada registro del array contendrá los siguientes campos:
  - `EP`: `EP_H` del vecino del que se espera *Ack*.
  - `Retries`: Número de reintentos de retransmisión ya realizados a ese vecino.

  En el array `Destinations` las posiciones que estén vacías tendrán el valor `null` en el campo `EP`.

Así, cada vez que se envíe o reenvíe un mensaje por inundación, se almacenará una entrada en *Sender_Dests* y otra entrada en *Sender_Buffering*. Igualmente cuando se elimine un mensaje de *Sender_Dests* por haber sido asentido por todos sus destinatarios, también habrá que borrar la entrada asociada en *Sender_Buffering*.

Cuando llegue un ACK asintiendo el mensaje identificado por (`EP_H_Creat`, `Seq_N`), se buscará por esa clave en *Sender_Dests* para obtener su array `Destinations` en el que anotar el nodo que ha asentido.

En la figura 4 se muestra el ABB *Sender_Dests* asociado al *Sender_Buffering* de la figura 3. Puede observarse como en el array `Destinations` de cada elemento aparecen con un `EP /= null` los `EP_H` de los vecinos de 2 que aún no han enviado el *Ack* de ese mensaje.

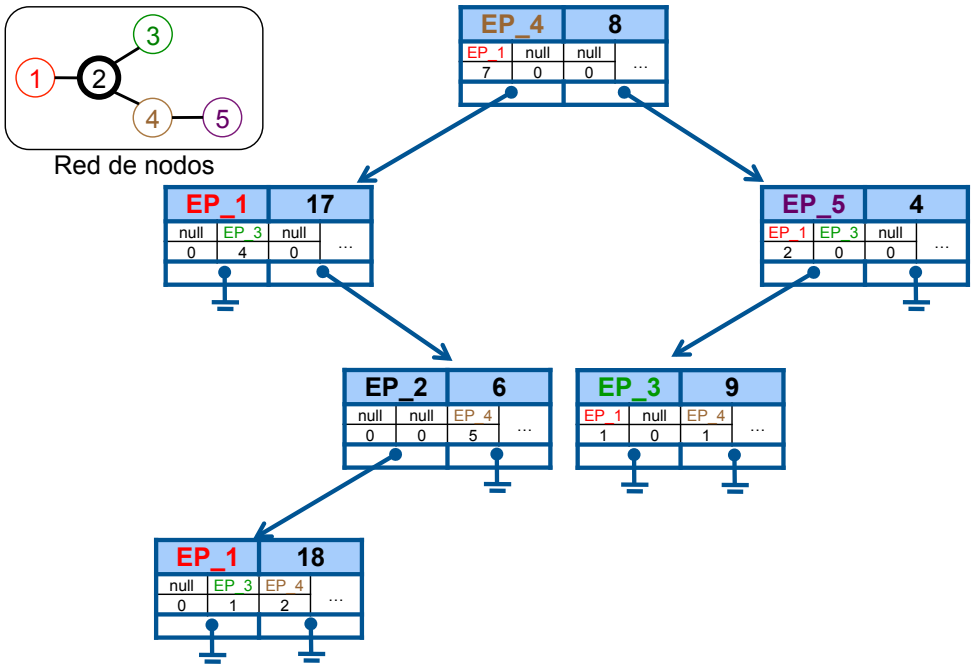


Figura 4: Ejemplo del *Sender_Dests* del nodo 2 que se corresponde con el *Sender_Buffering* de la figura 3

4.6.3. Descripción del protocolo de inundación

El protocolo de inundación descrito en la práctica anterior queda modificado en los siguientes aspectos:

- **Gestión de los mensajes *Ack***

  Los mensajes *Ack* serán recibidos en el *manejador de recepción*. Cada vez que se reciba un *Ack* se buscará en la tabla de símbolos *Sender_Dests* el elemento correspondiente al mensaje que asiente el *Ack*. Se localizará en el array `Destinations` el *endpoint* que envía el asentimiento, asignándole `null` a esa posición para indicar que ya se ha recibido el *Ack* de ese vecino.

  Si después de procesar un *Ack* aún queda en el array `Destinations` del mensaje asentido algún *endpoint* no nulo (es decir, del que aún no se ha recibido el Ack y aún no se ha llegado al número máximo de retransmisiones para él), se actualizará *Sender_Dests* haciendo un `Put` del nodo correspondiente al mensaje asentido para que se quede guardado el nuevo valor de `Destinations`.

  Si por el contrario ya no se espera recibir *Ack* de ningún vecino (porque ya han asentido todos o porque se ha alcanzado el número máximo de reintentos), se borrará la entrada de *Sender_Dests*[3].

- **Gestión de las retransmisiones**

  Al enviar o reenviar por inundación un mensaje, se creará un nuevo elemento que se insertará en *Sender_Dests* y un nuevo elemento en *Sender_Buffering*, y se instalará un manejador temporizado para que se ejecute cuando llegue esa hora de retransmisión, llamando para ello a `Timed_Handlers.Set_Timed_Handler`.

  Cuando llegue la hora de retransmisión de un mensaje el sistema llamará automáticamente al manejador temporizado que se instaló, pasándole como parámetro la hora a la que se programó su ejecución.

  El código del manejador temporizado deberá realizar las siguientes operaciones cuando sea llamado:

[3]Nótese que no puede borrarse también la entrada correspondiente de *Sender_Buffering*, porque no se tiene la clave (hora de retransmisión) para acceder a ella.

1. Buscar el elemento de *Sender_Buffering* correspondiente al mensaje a retransmitir. Para buscar el elemento se utilizará como clave la hora de retrasmisión que el sistema pasa como parámetro al procedimiento manejador temporizado cuando lo llama.
2. Borrar el elemento de *Sender_Buffering*, pues la hora de la clave con la que está almacenado ya no es válida. En caso de que se retransmita el mensaje a algún nodo, habrá que almacenar de nuevo el mensaje, pero usando como clave la nueva hora de retransmisión futura.
3. Con los campos `EP_H_Creat` y `Seq_N` obtenidos de *Sender_Buffering* se buscará en *Sender_Dests* para obtener el array `Destinations` del mensaje.

   Nótese que el elemento podría no estar ya en *Sender_Dests* si ya se hubieran recibido los *Ack* de todos los vecinos a los que se envió el mensaje (y, en ese caso, al recibirse el último *Ack* fue cuando se borró el elemento). En este caso no se continuaría con los pasos siguientes.
4. Si aún quedan *endpoints* de los que se espera recibir un *Ack* (porque aún no se ha recibido de ellos el *Ack* y no se ha alcanzado el número máximo de reintentos):
   - Se reenviará el mensaje a esos *endpoints*
   - Se actualizará la entrada en *Sender_Dests* para que se almacene el nuevo valor de `Destinations` (al aumentar el número de reintentos).
   - Se almacenará de nuevo un elemento en *Sender_Buffering*, con una nueva clave igual a la nueva hora de retransmisión (hora actual + plazo de retransmisión).
   - Se volverá a programar el manejador temporizado para que se vuelva a ejecutar a la nueva hora de retransmisión.

   Si no quedaran *endpoints* de los que se espera recibir un *Ack*, se borrará la entrada del mensaje de *Sender_Dests*.

Nótese que se programa la ejecución del manejador temporizado (que es un único procedimiento) una vez para cada mensaje que se almacena en *Sender_Buffering* y *Sender_Dests*.

- **Gestión del desorden de los mensajes**

  Cuando en un nodo se recibe un mensaje *Init*, *Confirm*, *Writer* o *Logout* creado por un cierto `EP_H_Creat` hay que comprobar su `Seq_N` con el que esté almacenado en *latest_msgs* para ese mismo nodo, y según su resultado:

  - Si el mensaje recibido tiene un `Seq_N` **inmediantamente consecutivo** al almacenado en *latest_msgs*, o procede de un `EP_H_Creat` que no está aún en *latest_msgs*[4], **el mensaje es nuevo**,
    - Hay que enviar un `Ack` al `EP_Hdl_Rsnd` del mensaje.
    - Hay que reenviar el mensaje recibido por inundación, programando su posible retransmisión.
    - Hay que realizar el procesamiento específico del mensaje según su tipo (por ejemplo, mostrar en pantalla el contenido de un mensaje *Writer*).
  - Si el mensaje recibido tiene un `Seq_N` **menor o igual** que el almacenado en *latest_msgs*, **el mensaje ya ha sido visto anteriormente**, por lo que:
    - Sí hay que enviar un `Ack` al `EP_H_Rsnd` del mensaje, aunque ya se habrá enviado otro anteriormente. Esta situación puede darse en estos dos escenarios:
      - ◇ La primera vez que su vecino `EP_H_Rsnd` le envió este mismo mensaje no recibió el `Ack` y por eso dicho vecino lo retransmite. Por ello hay que enviarle de nuevo el `Ack`.
      - ◇ El nodo ya ha recibido este mismo mensaje a través de otro vecino. Por tanto tiene que enviar el `Ack` para que este segundo vecino que le envía ahora por primera vez el mismo mensaje no se lo vuelva a retransmitir de nuevo en el futuro.
    - No hay que reenviarlo por inundación, puesto que ya se hizo antes.
    - No hay que realizar el procesamiento específico del mensaje (por ejemplo, mostrar en pantalla el contenido de un mensaje *Writer*), puesto que ya se hizo antes.
  - Si el mensaje recibido tiene un `Seq_N` **mayor en 2 o más unidades** al almacenado en *latest_msgs*, **es un mensaje del futuro**, demasiado nuevo, por lo que:
    - No hay que asentirlo, puesto que no puede ser procesado ahora, sino que antes hay que recibir y procesar los mensajes creados por el mismo nodo que aún no se han recibido. Al no ser asentido por el nodo que lo ha recibido, este mensaje será retransmitido por el vecino que se lo ha enviado, por lo que se volverá a recibir más tarde.

[4] y no es un *Logout*, como se explica al final del apartado 2.6 de la práctica anterior

- Sí hay que reenviarlo por inundación, ya que puede ser que a los vecinos de este nodo no les falten los mensajes anteriores que a este nodo sí le faltan, y puedan ya procesar el mensaje
- No hay que realizar el procesamiento específico del menasaje, pues antes hay que recibir y procesar los mensajes que le preceden.

- **Ejemplo**

  La figura 5 muestra el ejemplo de una red a la que se está incorporando el nodo 5. El nodo 5 enviará el mensaje *Init* al nodo 4, que, tras asentirlo, se lo reenviará al nodo 2. En la figura se muestra cómo quedan las tablas `sender_buffering`, `sender_dests`, `latest_messages` y `neighbors` en el nodo 2 tras la recepción y procesamiento del mensaje *Init* que recibe del nodo 4.

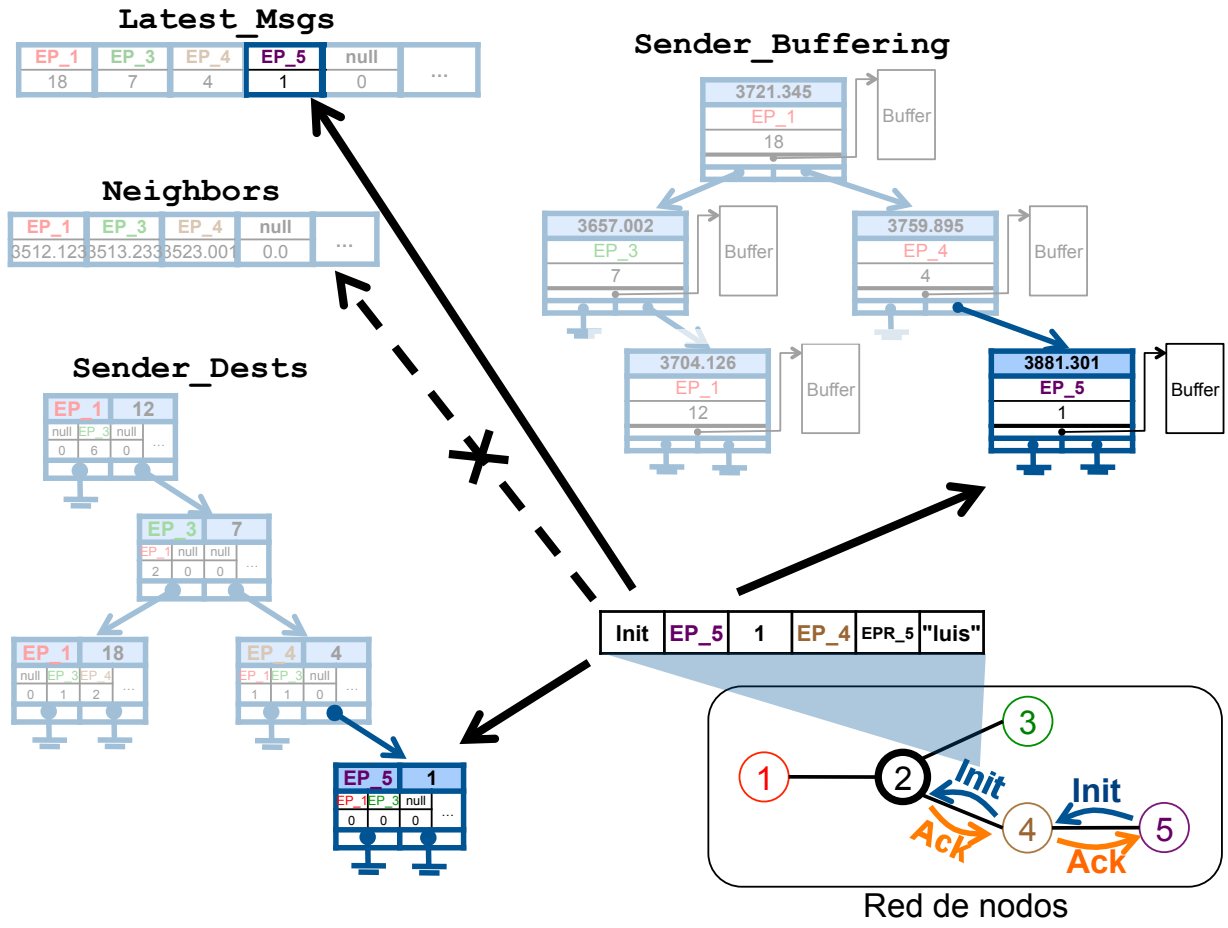


Figura 5: Ejemplo que muestra el estado de las tablas del nodo 2 tras recibir el mensaje *Init* que le envía el nodo 4 y que fue creado por el nodo 5

El nodo 2 analizará el mensaje *Init* y procederá como sigue:

- Consulta la tabla *latests_msgs*. Al ser el primer mensaje recibido de ese *endpoint* se asiente, y se añade el mensaje a la tabla de símbolos *latest_msgs*.
- Se comprueba si el *nick* del mensaje es el mismo que el del nodo 2, en cuyo caso se enviaría un *Reject*.
- Se añade un elemento a la tabla de símbolos *Sender_Dests* y a la tabla de símbolos *Sender_Buffering* para gestionar el posible reenvío del mensaje a los nodos vecinos 1 y 3, se reenvía ese mensaje *Init* a esos vecinos, y se programa el manejador temporizado para encargarse de su posible retransmisión a la hora adecuada.
- Al no coincidir los campos `EP_H_Creat` y `EP_H_Rsnd`, no hay que añadir un nuevo elemento a la tabla *neighbors*.

5. Pautas de implementación

En esta sección se dan indicaciones sobre cómo abordar la programación de algunos aspectos importantes de esta práctica.

5.1. Implementación de *Sender_Buffering* y *Sender_Dests* mediante paquetes genéricos

Las tablas de símbolos *Sender_Buffering* y *Sender_Dests* deben implementarse **obligatoriamente** en la forma indicada en este apartado.

1. Debe partirse del paquete `Ordered_Maps_G` (disponible junto a este enunciado) que implementa una tabla de símbolos mediante un ABB, cuya especificación es la siguiente:

```
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;
   with function ">" (K1, K2: Key_Type) return Boolean;
   with function Key_To_String (K: Key_Type) return String;
   with function Value_To_String (K: Value_Type) return String;
package Ordered_Maps_G is

   type Map is limited private;

   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;

   procedure Print_Map (M : Map);


private
   ...
end Ordered_Maps_G;
```

2. Deberá instanciarse 2 veces el paquete `Ordered_Maps_G`, para obtener dos paquetes:

   - Paquete `NP_Sender_Dests`: Como `Key_Type` se usará el tipo `Mess_Id_T` y como `Value_Type` se usará el tipo `Destinations_T`, definidos como se muestra a continuación:

     ```
     type Mess_Id_T is record
        EP: LLU.End_Point_Type;
        Seq: Types.Seq_N_T;
     end record;

     type Destination_T is record
        Ep      : Llu.End_Point_Type := null;
        Retries : Natural := 0;
     end record;

     type Destinations_T is array (1..10) of Destination_T;
     ```

     En estas declaraciones debe sustituirse el nombre de paquete `Types` por el nombre del paquete en el que esté declarado el tipo `Seq_N_T`.

     Así, para cada mensaje que se envíe/reenvíe por inundación se guardará como clave el registro formado por (`EP_H_Creat`, `Seq_N`) y como valor array de destinatarios..

     Para poder hacer esta instanciación deberán definirse funciones de comparación entre `Mess_Id_T`: `"="`, `"<"` y `">"`. Para ello, puede establecerse la comparación como el resultado de comparar primero los campos `Ep` del registro, y si son iguales, comparar los campos `Seq`. Para comparar dos `End_Point_Type` puede usarse la comparación como `String` del `LLU.Image` de cada `End_Point`.

     También deberá definirse una función que para un `Mess_Id_T` devuelva un `String`, y una que para un `Destinations_T` devuelva un `String`.

   - Paquete `NP_Sender_Buffering`: Como `Key_Type` se usará el tipo `Ada.Calendar.Time` y como `Value_Type` se usará el tipo `Value_T` definido a continuación:

     ```
     type Buffer_A_T is access LLU.Buffer_Type;

     type Value_T is record
        EP_H_Creat    : LLU.End_Point_Type;
        Seq_N         : Types.Seq_N_T;
        P_Buffer      : Buffer_A_T;
     end record;
     ```

     Para poder hacer la instaciación habrá que usar los operadores de comparción entre dos `Time` del paquete `Ada.Calendar`.

     También deberá definirse una función que para un `Ada.Calendar.Time` devuelva un `String`, y una que para un `Value_T` devuelva un `String`.

3. Tanto el programa principal del nodo como su manejador deberán acceder concurrentemente a las variables que contengan las tablas *Sender_Dests* y *Sender_Buffering*. El acceso concurrente a variables compartidas requiere utilizar en Ada una estructura de datos llamada `protected type`, que no explicaremos. Proporcionamos junto a este enunciado el paquete genérico `Ordered_Maps_Protector_G`, que instanciado con los 2 paquetes del punto anterior, proporcionará `Maps` protegidos ante el acceso concurrente. Estas insanciaciones se harán de la siguiente forma:

   ```
   package Sender_Dests is new
      Ordered_Maps_Protector_G (NP_Sender_Dests);

   package Sender_Buffering is new
      Ordered_Maps_Protector_G (NP_Sender_Buffering);
   ```

   Estas instanciaciones pueden hacerse en el `.ads` de paquete `Chat_Handler` que contenga el *handler* del nodo.

   Los paquetes `Sender_Dests` y `Sender_Buffering` serán los que se usarán en todo código del nodo, tanto en el programa principal como en el paquete `Chat_Handler`.

4. Los paquetes `Sender_Dests` y `Sender_Buffering` que se han instanciado dentro del paquete `Chat_Handler` deben usarse desde el programa principal **sin volver a instanciarlos**.

   Así, en `chat_peer.adb` se utilizará el paquete `Sender_Dests` escribiendo `Chat_Handler.Sender_Dests`, y el paquete `Sender_Buffering` se utilizará escribiendo `Chat_Handler.Sender_Buffering`.

5. Las variables de las dos tablas de símbolos de tipo `Sender_Buffering.Prot_Map` y `Sender_Dests.Prot_Map` respectivamente también deberán ser declaradas en la especifición del paquete `Chat_Handler`, para poder ser usadas tanto desde el propio manejador como desde el programa principal.

5.2. Gestión de *Buffers*

Debido a que *Sender_Buffering* debe almacenar, como parte del `Value_T`, un puntero a un `LLU.Buffer_Type`, es necesario que cada vez que se componga un mensaje para enviar o reenviar por inundación (tanto en `chat_peer.adb` como en `chat_handler.adb`) se haga de la siguiente forma:

- En `chat_messages.ads` se definirán dos punteros a *buffers* de la siguiente forma:

```
P_Buffer_Main: Buffer_A_T;
P_Buffer_Handler: Buffer_A_T;
```

  Siendo el tipo `Buffer_A_T` el tipo puntero a `LLU.Buffer_Type` definido en el punto 5.1 apartado 2.

- En `chat_peer.adb`: Cada vez que se componga un mensaje `Init`, `Confirm`, `Writer` o `Logout` deberá usarse un *buffer* nuevo creado dinánicamente de la siguiente forma:

```
Chat_Messages.P_Buffer_Main := new LLU.Buffer_Type(1024);
```

  Y a continuación con el *buffer* apuntado por `P_Buffer_Main` se compondrá el mensaje, se almacerá en *Sender_Buffering*, y se enviará el mensaje.

- En `chat_handler.adb`: Cada vez que se vaya a reenviar por inundación un mensaje `Init`, `Confirm`, `Writer` o `Logout` deberá usarse un *buffer* nuevo creado dinánicamente de la siguiente forma:

```
Chat_Messages.P_Buffer_Handler := new LLU.Buffer_Type(1024);
```

  Y a continuación con el *buffer* apuntado por `P_Buffer_Handler` se compondrá el mensaje, se almacerá en *Sender_Buffering*, y se enviará el mensaje.

- Cada vez que se elimine definitivamente una entrada en *Sender_Buffering* (sin volver a reinsertarse porque el mensaje ya no debe ser reenviado más) , si se desea liberar la memoria apuntada por `Value.P_Buffer` deberá declararse el procedimiento `Free` en la forma:

```
procedure Free is new Ada.Unchecked_Deallocation (LLU.Buffer_Type, Buffer_A_T);
```

  Y para liberar la memoria debe usarse:

```
Free(Value.P_Buffer);
```

5.3. Mensajes de depuración

Es **imprescindible** que el programa escriba **mensajes de depuración de manera sistemática y ordenada** para ayudarte mientras que desarrollas el programa.

En particular deberán mostrarse los principales campos de los mensajes que se van recibiendo, de los mensajes que se crean, de los mensajes que se reenvían y de los mensajes que no se reenvían.

También deberás añadir comandos que te permitan mostrar los contenidos de las tablas *neighbors*, *latest_msgs*, *Sender_Dests* y *sender_buffering* en cualquier momento, interactivamente.

Cuando imprimas en pantalla mensajes de depuración aparecerá mucha información, por lo que tienes que utilizar diferentes colores que mejoren la legibilidad de los mensajes de depuración. El paquete `pantalla` que se proporciona junto a este enunciado permite utilizar diferentes colores en un teminal.

Junto a este enunciado se proporciona una implementación de Chat_Peer v2.0 para que la puedas utilizar como referencia. Utiliza el mismo formato para los mensajes de depuración que se utiliza en dicha paquete.

5.4. Orden de implementación

Recomendamos realizar la implementación de la práctica en el orden siguiente:

1. Cambiar el uso de los *buffers* para los mensajes de inundación como se explica en el apartado 5.2.
2. Estudiar el programa `timed_handlers_test.adb` para entender la forma de usar los manejadores temporizados.
3. Escribir las funciones de comaparación de `Mess_Id_T` y todas las funciones de conversión a `String` que se necesitan para poder instanciar los dos paquetes para las dos nuevas tablas de símbolos `Sender_Dests` y `Sender_Buffering`.
4. Escribir un pequeño programa de prueba aparte en que se realice las instanciaciones necesarias para obtener los paquetes `Sender_Dests` y `Sender_Buffering` (tal y como se explica en el apartado 5.1), y que compruebe el funcionamiento básico de dichas tablas de símbolos.
5. Copiar la instanciaciones de los paquetes a `chat_handler.ads`.
6. Escribir en `chat_peer.adb` las llamadas que fuerzan el desorden de mensajes y el porcentaje de pérdidas a partir de los nuevos parámetros de la línea de comandos.
7. Escribir en `chat_peer.adb` y `chat_handler.adb` el código necesario para almacenar en las dos nuevas tablas de símbolos cada mensaje que se envíe o reenvíe, y que programa un manejador temporizado para su hora de retransmisión.
8. Escribir el procedimiento manejador temporizado que se encarga de las retransmisiones.
9. Añadir el código con el que se envíe un mensaje de ACK para un mensaje recibido cuando proceda.
10. Escribir en el handler de recepción el procesamiento de los mensajes ACK que se reciban en el nodo.
11. Escribir el nuevo código de consulta a *latest_msgs* para determinar si un mensaje recibido es un mensaje ya visto, un mensaje inmediatamente consecutivo o un mensaje del futuro, y ajustar lo que se hace en cada uno de estos 3 casos.

6. Condiciones de funcionamiento

1. La tabla de símbolos *neighbors* debe poder almacenar hasta 10 nodos.
2. La tabla de símbolos *latest_msgs* debe poder almacenar las entradas de hasta 50 nodos.
3. Las tablas de símbolos *Sender_Dests* y *sender_buffering* no tendrán límite en cuanto al número de nodos que pueden almacenar.
4. Chat-Peer v2.0 deberá funcionar con normalidad aunque haya pérdidas y desorden de los mensajes que se envíen por la red.
5. En Chat-Peer v2.0 **se supondrá que han sido previamente arrancados los nodos vecinos de contacto que se especifican en la línea de comandos al lanzar un nodo, y que éstos han terminado su protocolo de admisión**.
6. En Chat-Peer v2.0 se supondrá que un nodo no terminará su ejecución si con ello dejara a vecinos suyos "desconectados" del chat al no quedarles ningún otro vecino arrancado.
7. Se supondrá que los nodos no pueden terminar con CTRL-C[5].
8. Deberán ser compatibles las implementaciones de los nodos desarrollados por diferentes alumnos, para lo cuál es imprescindible respetar los protocolos, y particularmente el formato de los mensajes.
9. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.

[5]Uno de los Módulos Opcionales de esta práctica consistirá en capturar la interrupción producida por CTRL-C para enviar el mensaje *Logout* antes de terminar el programa

7. Normas y plazos de entrega

Esta práctica se entregará presencialmente en las aulas de prácticas a un profesor de la asignatura, que probará el funcionamiento correcto del programa y realizará al alumno preguntas orales sobre su código fuente.

Habrá 2 fechas posibles para entregar la práctica, a elección de cada alumno:

- El día 9 de enero, después del examen escrito
- El día 10 de enero por la mañana.

Se anunciará en la página de la asignatura la forma de elegir turno de correción por parte de cada alumno.