Programación de Sistemas de Telecomunicación / Informática II
Práctica 4:
Chat-Peer

Departamento de Sistemas Telemáticos y Computación
(GSyC)

Noviembre de 2014

Resumen

En esta práctica debes realizar un nuevo programa en Ada, **Chat-Peer**, que ofrezca un servicio de chat entre usuarios utilizando el modelo P2P descentralizado.

Todos los procesos que participan en Chat-Peer son iguales, por lo que sólo deberá implementarse un programa, `chat_peer`, compuesto de un procedimiento principal y varios paquetes. A cada proceso lo llamaremos nodo, par o *peer* indistintamente.

Para entrar en el chat un nodo debe seguir el **protocolo de admisión** antes de poder empezar a enviar mensajes con cadenas de texto a los demás pares, o de mostrar en pantalla las cadenas de texto que envíen los demás pares. Durante el protocolo de admisión un nodo que intenta entrar en Chat-Peer debe ser rechazado por cualquier otro nodo que ya esté usando el mismo apodo (*nickname*).

Cada nodo sólo conocerá a unos cuantos pares, a los que llamaremos **vecinos** (*neighbors*). Cuando un nodo cree un nuevo mensaje que quiere que se reciba en todos los nodos, utilizará un **protocolo de inundación controlada**: el nodo que crea el mensaje lo envía sólo a sus nodos vecinos; cada vez que un nodo recibe un nuevo mensaje que aún no había visto, lo reenvía a todos sus vecinos, salvo al vecino que se lo ha enviado a él. De esta forma, el mensaje llega finalmente a todos los nodos al menos una vez.

Es conveniente que antes de seguir leyendo estudies con detenimiento el tema de teoría 5: El modelo Peer-to-Peer (P2P).

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

El programa de cada nodo se lanzará pasándole 2, 4 o 6 argumentos en la línea de comandos:

```
./chat_peer port nickname [[neighbor_host neighbor_port] [neighbor_host neighbor_port]]
```

Los dos 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.

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

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

El nodo que arranca utilizará un **protocolo de admisión** que puede conducir a que se rechace al nodo que entra o a que se le acepte. Si el nodo es arrancado sin especificar vecinos iniciales en la línea de comandos, no efectuará el protocolo de admisión.

En caso de ser aceptado, el nodo pasará a formar parte de Chat-Peer. Entonces irá pidiendo al usuario cadenas de caracteres por la entrada estándar y se las enviará por inundación al resto de pares. Si la cadena de caracteres leída de la entrada estándar es `.salir`, el nodo terminará su ejecución.

Al mismo tiempo el nodo irá recibiendo las cadenas de texto enviadas por otros pares y las mostrará en pantalla.

En la figura 1 se muestra un ejemplo de ejecución de Chat-Peer en el que hay 5 nodos.

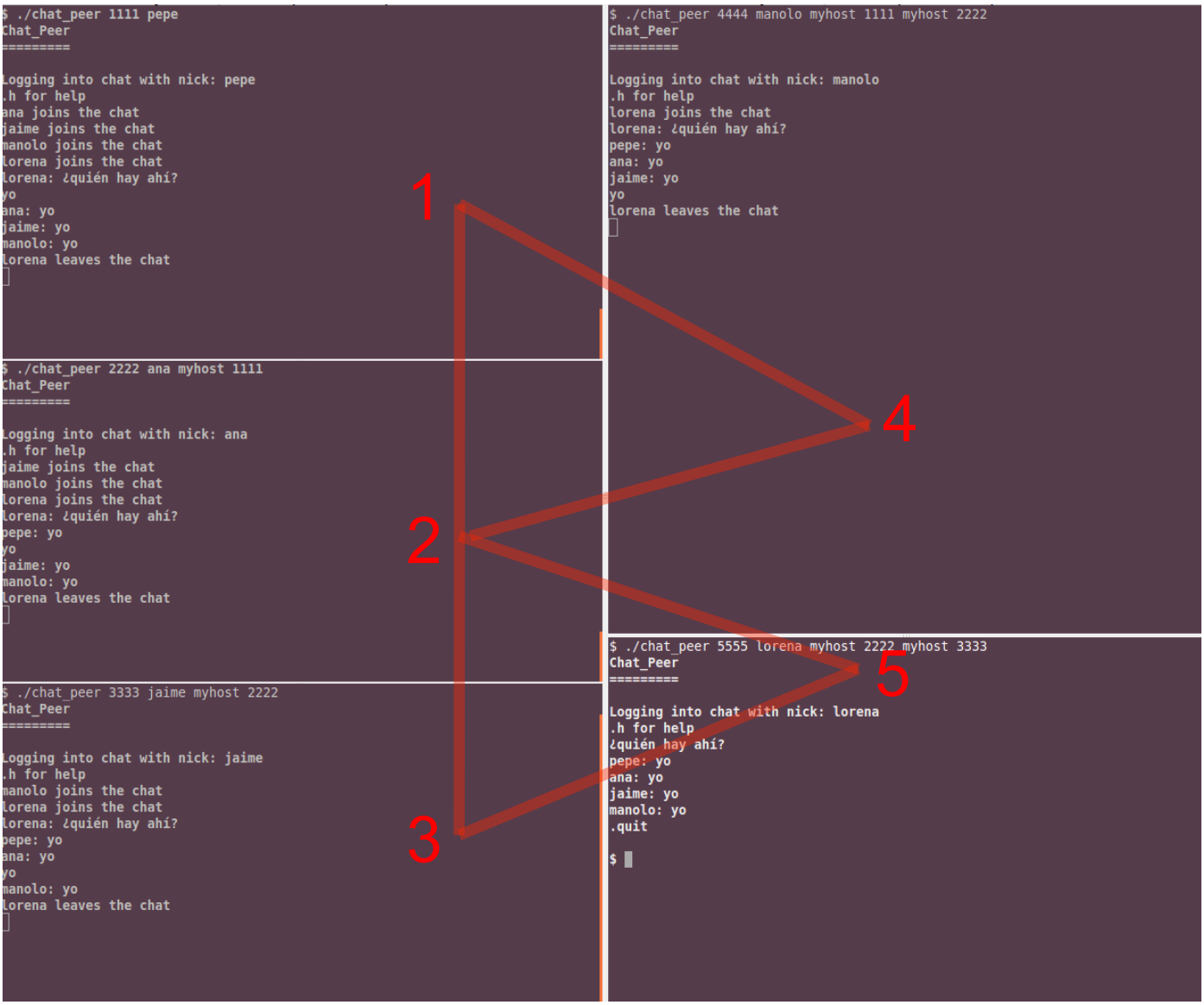

Figura 1: Han entrado 5 nodos en Chat-Peer en el orden indicado por los números. Las líneas muestran la relación de vecindad entre los vecinos.

2. Implementación

2.1. Tipos de mensaje utilizados en esta práctica

En esta práctica se utilizarán 5 tipos distintos de mensajes que se resumen a continuación. En el apartado 3 se explica el formato de cada uno de estos mensajes:

- **Mensaje *Init***: Es el que envía por inundación un nodo al arrancar como parte del protocolo de admisión.
- **Mensaje *Reject***: Es el que envía un nodo como parte del protocolo de admisión cuando al recibir un mensaje *Init* detecta que el nodo que creó el mensaje *Init* pretende utilizar su mismo apodo. Este tipo de mensaje es el único que no se envía por inundación.
- **Mensaje *Confirm***: Es el que envía por inundación un nodo al arrancar como parte del protocolo de admisión cuando pasan 2 segundos tras haber enviado un mensaje *Init* sin que se haya recibido un mensaje *Reject*.
- **Mensaje *Writer***: Es el que envía por inundación un nodo conteniendo una cadena de texto leída del teclado.
- **Mensaje *Logout***: Es el que envía por inundación un nodo para informar de que abandona el Chat-Peer. Este mensaje puede enviarse en dos circunstancias: como parte del protocolo de admisión si el nodo que arranca no ha sido admitido en el Chat-Peer por querer utilizar un apodo duplicado, y como parte del protocolo de salida si el nodo abandona el Chat-Peer tras haber participado durante un tiempo en él.

2.2. Fases

Cada nodo de Chat-Peer deberá ejecutar en orden las fases que se detallan a continuación: Inicialización, protocolo de admisión, protocolo de envío/recepción de mensajes *Writer* y protocolo de salida.

1. Nada más arrancar, antes de intercambiar mensajes, un nodo deberá realizar las siguientes operaciones de **Inicialización**:
   - Cada nodo debe construir los *endpoints* de sus nodos vecinos iniciales a partir de los argumentos que le pasen en la línea de argumentos, si los hubiera, y añadirlos a su tabla de símbolos de vecinos (*neighbors*). Un nodo se puede lanzar con 0, 1 o 2 **nodos vecinos iniciales**. Sólo el primer nodo de la red se lanzará con 0 vecinos iniciales.
   - Cada nodo utilizará 2 *endpoints* para recibir mensajes, a los que deberá atarse al comienzo del programa principal, nada más ser lanzado:
     - `EP_R`: para recibir mensajes *Reject* con la llamada bloqueante a `LLU.Receive`.
     - `EP_H`: para recibir el resto de mensajes mediante un *handler* de LLU. Este *endpoint* servirá para identificar al nodo en la red.
2. Para entrar en el chat un nodo debe seguir el **protocolo de admisión** antes de poder empezar a enviar mensajes con cadenas de texto a los demás pares, o de mostrar en pantalla las cadenas de texto que envíen los demás pares. Durante el protocolo de admisión un nodo que intenta entrar en Chat-Peer podrá ser rechazado por cualquier otro par que ya esté usando el mismo apodo (*nickname*).

   El protocolo de admisión utiliza los mensajes *Init, Reject, Confirm* y *Logout*. En el apartado 2.4 se detalla el protocolo de admisión.
3. En caso de no ser rechazado al ejecutar el protocolo de admisión el nodo pasará a ejecutar el **protocolo de envío y recepción de mensajes *Writer***, mediante el cuál se le ofrece al usuario el servicio habitual de un chat: enviar a los demás usuarios las cadenas de texto leídas del teclado y mostrar en pantalla las cadenas enviadas por el resto de usuarios.

   El protocolo de envío y recepción de mensajes *Writer* utiliza los mensajes *Writer*. En el apartado 2.5 se detalla este protocolo.
4. Cuando un usuario quiera abandonar Chat-Peer ejecutará el comando `.salir`. En este momento el nodo realizará el **protocolo de salida** y a continuación terminará el programa.

   Este protocolo utiliza los mensajes *Logout*. En el apartado 2.6 se detalla este protocolo.

Todos los mensajes, salvo los *Reject*, se envían por inundación, por lo que cada nodo deberá contribuir a realizar el **protocolo de inundación controlada**. Para ello, cada nodo tiene que realizar las siguientes funciones:

- cuando un nodo cree un mensaje nuevo para enviar, le pondrá un nuevo número de secuencia y se lo enviará a sus vecinos (que, a su vez, lo reenviarán a los suyos, y así sucesivamente) excepto al vecino que se lo envió a él.
- cuando un nodo reciba un mensaje de inundación de un vecino, comprobará si es nuevo para él o no:
  - si es nuevo para él, debe reenviarlo a sus vecinos (que, a su vez, lo reenviarán a los suyos, y así sucesivamente)
  - si no es nuevo para él, es decir, ya lo había visto antes, NO lo reenviará a nadie.

Para poder realizar las funciones anteriores, cada nodo además, tendrá que mantener actualizadas dos estructuras de datos:

- la tabla de símbolos de vecinos (***neighbors***)
- la tabla de símbolos de últimos mensajes recibidos (***latest_msgs***)

En el apartado 2.3 se detalla el protocolo de inundación controlada y cómo se utilizan estas dos estructuras de datos.

2.3. Protocolo de inundación controlada

Campos de los mensajes de inundación

Los siguientes tipos de mensaje son enviados por inundación: *Init, Confirm, Writer, Logout*.
Todos los tipos de mensaje que se envían por inundación tienen al menos los siguientes campos:

- `EP_H_Creat`: `EP_H` del nodo que creó el mensaje
- `Seq_N`: Número de secuencia asignado por `EP_H_Creat` a este mensaje. Cada vez que un nodo crea un nuevo mensaje que tiene que enviar por inundación debe incrementar en 1 su número de secuencia.
- `EP_H_Rsnd`: `EP_H` del nodo que reenvía el mensaje. Cada vez que un nodo reenvía un mensaje pone su propio `EP_H` en este campo antes de reenviarlo.

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`

El primer mensaje de inundación que crea un nodo tiene el número de secuencia 1.

Estructuras de datos

Cada nodo, para poder gestionar el protocolo de inundación controlada, utiliza dos estructuras de datos:

- Tabla de símbolos ***neighbors***: almacena como claves los `EP_H` de sus nodos vecinos y como valores las horas a las que se ha añadido cada vecino a la tabla.
  - A la tabla de símbolos *neighbors* de un nodo i se añaden elementos en las siguientes situaciones:
    - al arrancar, se añaden los nodos vecinos iniciales que se le han pasado en la línea de comandos al nodo i
    - más adelante, cuando otro nodo j arranque teniendo a i como vecino inicial, j enviará un Init. Cuando i lo reciba, deberá añadir al nodo j en su tabla *neighbors*, ya que j es un nuevo vecino para i.
  - De la tabla *neighbors* de un nodo i se elimina el `EP_H` de un nodo j (vecino de i) cuando i recibe un *Logout* creado por j.
- Tabla de símbolos ***latest_msgs***: almacena los últimos mensajes recibidos de cada nodo, siendo la clave `EP_H_Creat` y el valor `Seq_N`):

- Se añade un elemento nuevo a la tabla *latest_msgs* del nodo i cuando i recibe un mensaje creado por el nodo j cuyo `EP_H` aún no está en la tabla (por no haber enviado antes ese nodo ningún mensaje).
- Se añade también un elemento nuevo a la tabla *latest_msgs* del nodo i cuando i crea su primer mensaje (el *Init*), ya que el último mensaje creado por i también está en su propia tabla *latest_msgs*.
- Se modifica un elemento de la tabla *latest_msgs* del nodo i cuando i recibe un mensaje creado por el nodo j cuyo `EP_H` ya estaba en la tabla, pero cuyo `Seq_N` es **mayor** que el último que había en la tabla para ese nodo j.
- Se modifica también en la tabla *latest_msgs* del nodo i la entrada correspondiente al propio nodo i cada vez que i crea un mensaje nuevo.
- Se elimina de la tabla *latest_msgs* del nodo i la entrada correspondiente a un nodo j cuando i recibe un *Logout* creado por j.

Descripción del protocolo de inundación

- Al crear un mensaje:

  El nodo que crea un nuevo mensaje que quiere enviar por inundación tiene que realizar las siguientes acciones:

  1. Incrementar en 1 el `Seq_N` de mensaje (con respecto al que utilizó en el último mensaje creado por él).
  2. Actualizar su propia entrada en la tabla *latest_msgs*
  3. Enviar el mensaje a cada uno de los nodos de su tabla *neighbors*.

- Al recibir un mensaje:

  Sea i el nodo creador de un mensaje, j el nodo que reenvía el mensaje recibido de i, y k el nodo que recibe el mensaje creado por i y que ha sido reenviado por j:

  $$\textcircled{i} \rightarrow \textcircled{j} \rightarrow \textcircled{k}$$

  El nodo k que recibe un mensaje (`EP_H_i`, `Seq_N`) reenviado por inundación por `EP_H_j` tiene que realizar las siguientes acciones:

  1. Si `EP_H_i` no está en *latest_msgs*:
     - añadir una entrada en *latest_msgs* con clave `EP_H_i` y valor `Seq_N`
     - reenviar el mensaje a todos sus vecinos excepto a `EP_H_j` (que es el que se lo ha reenviado a él), poniendo k su `EP_H_k` en el campo `EP_H_Rsnd`
     - realizar el procesamiento específico del mensaje según su tipo (por ejemplo, si es un *Writer* tiene que escribir en pantalla el texto del mensaje...)
  2. Si `EP_H_i` está en *latest_msgs*, pero el `Seq_N` recibido es **mayor** que el que hay su entrada:
     - actualizar la entrada en *latest_msgs* para el nodo i con el nuevo `Seq_N`
     - reenviar el mensaje a todos sus vecinos excepto a `EP_H_j` (que es el que se lo ha reenviado a él), poniendo k su `EP_H_k` en el campo `EP_H_Rsnd`
     - realizar el procesamiento específico el mensaje según su tipo (por ejemplo, si es un *Writer*, escribir en pantalla el texto del mensaje...)
  3. Si `EP_H_i` está en *latest_msgs*, y el `Seq_N` recibido es **menor o igual** que el que hay su entrada:
     - ignorar el mensaje, pues ya se recibió, reenvió y procesó anteriormente (cuando este mensaje llegó por primera vez)

  * EXCEPCIÓN: Cuando el mensaje recibido es un *Logout* y el `EP_H_j` no está en *latest_msgs*, el mensaje debe ignorarse. Más información en el apartado 2.6.

2.4. Protocolo de admisión

El protocolo de admisión le sirve al nodo que lo inicia para:

- Detectar si su *nickname* ya está siendo usado por otro nodo, en cuyo caso no puede entrar en el chat.
- Darse a conocer a sus nodos vecinos iniciales especificados en la línea de comandos, para que éstos lo añadan a su tabla de vecinos.
- Informar a todos los nodos de la red de su presencia como nuevo usuario (en caso de que el protocolo de admisión concluya con éxito).

Descripción del protocolo de admisión

- Al arrancar un nodo:

  1. El nodo enviará por inundación un **mensaje *Init***.
  2. Tras enviar el mensaje *Init* el nodo esperará a recibir mediante `LLU.Receive` en su `EP_R` un **mensaje *Reject*** con tiempo máximo de espera de 2 segundos.
     - Si transcurrido el tiempo máximo de espera de 2 segundos no se ha recibido ningún mensaje *Reject*, el nodo enviará por inundación un **mensaje *Confirm*** y dará por concluído con éxito el protocolo de admisión.
     - Si recibe el mensaje *Reject* antes de que venza el tiempo máximo de espera de 2 segundos, enviará por inundación un **mensaje *Logout***.
       A continuación el programa terminará por no haber sido admitido el nodo en Chat-Peer.

- Cada nodo que reciba un mensaje *Init* lo tratará como cualquier otro mensaje enviado por inundación, reenviándolo y procesándolo si es preciso. En el caso del *Init* su procesamiento específico consitirá en:
  - comprobar si el apodo que viene en el mensaje es el mismo que el suyo, en cuyo caso se enviará un **mensaje *Reject*** directamente al `EP_R` del nodo que creó el mensaje *Init* (campo `EP_R_Creat` del mensaje *Init*).
  - comprobar si el nodo que ha creado el mensaje *Init* le considera como vecino (es decir, el campo `EP_H_Creat` = `EP_H_Rsnd` en el mensaje *Init*), en cuyo caso añadirá a `EP_H_Creat` a su tabla *neighbors*.

- Cada nodo que reciba un mensaje *Confirm* lo tratará como cualquier otro mensaje enviado por inundación, reenviándolo y procesándolo si es preciso. En el caso del *Confirm* su procesamiento específico consitirá en:
  - mostrar en pantalla un texto indicando que el usuario cuyo nick viene en el mensaje *Confirm* ha entrado en Chat-Peer

- Cada nodo que reciba un mensaje *Logout* lo tratará como cualquier otro mensaje enviado por inundación, reenviándolo y procesándolo si es preciso. En el caso de estos mensajes *Logout* su procesamiento específico consitirá en:
  - comprobará si el nodo que ha creado el mensaje *Logout* es vecino suyo (campo `EP_H_Creat` = `EP_H_Rsnd`), en cuyo caso el nodo receptor del mensaje eliminará al emisor (`EP_H_Creat`) de su tabla *neighbors*.
  - borrará de su tabla *latest_msgs* la entrada correspondiente al nodo que creó el mensaje (`EP_H_Creat`).

Ejemplos

La figura 2 muestra un diagrama con un ejemplo de ejecución del protocolo de admisión que no incluye el envío de ningún *Reject*, y la figura 3 muestra otro ejemplo en el que sí se envía un *Reject*.

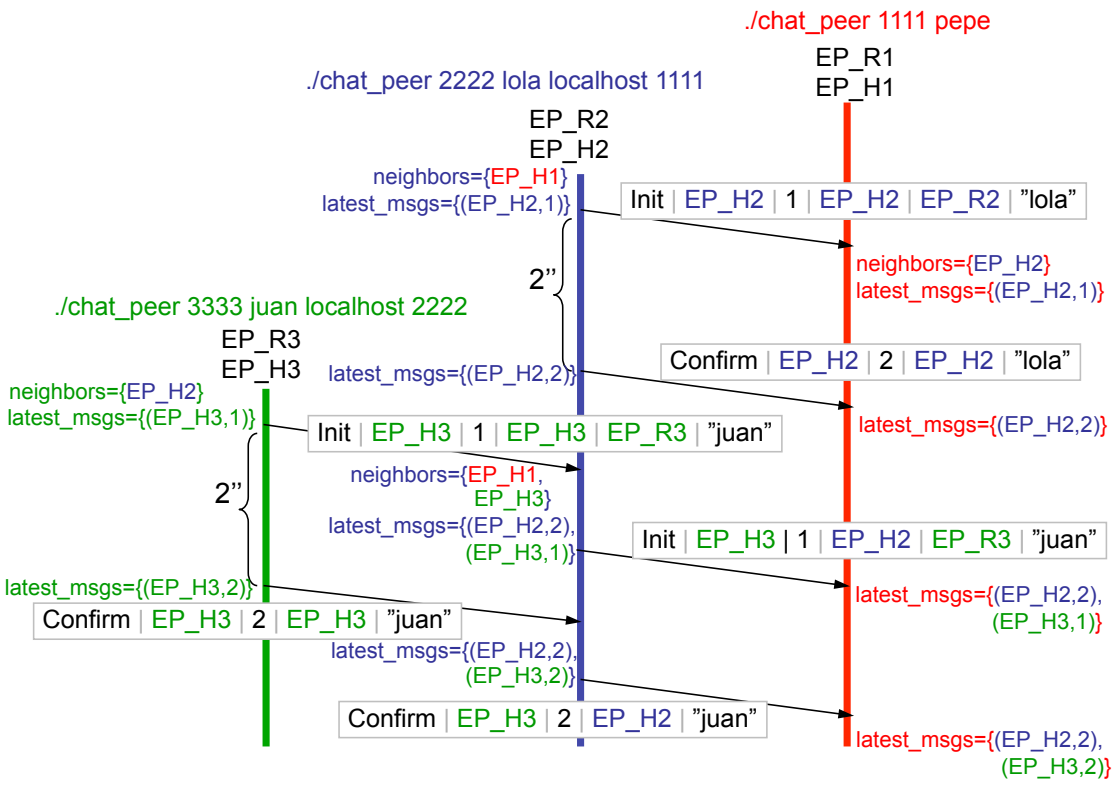

Figura 2: Ejemplo de protocolo de admisión en el que no hay rechazo

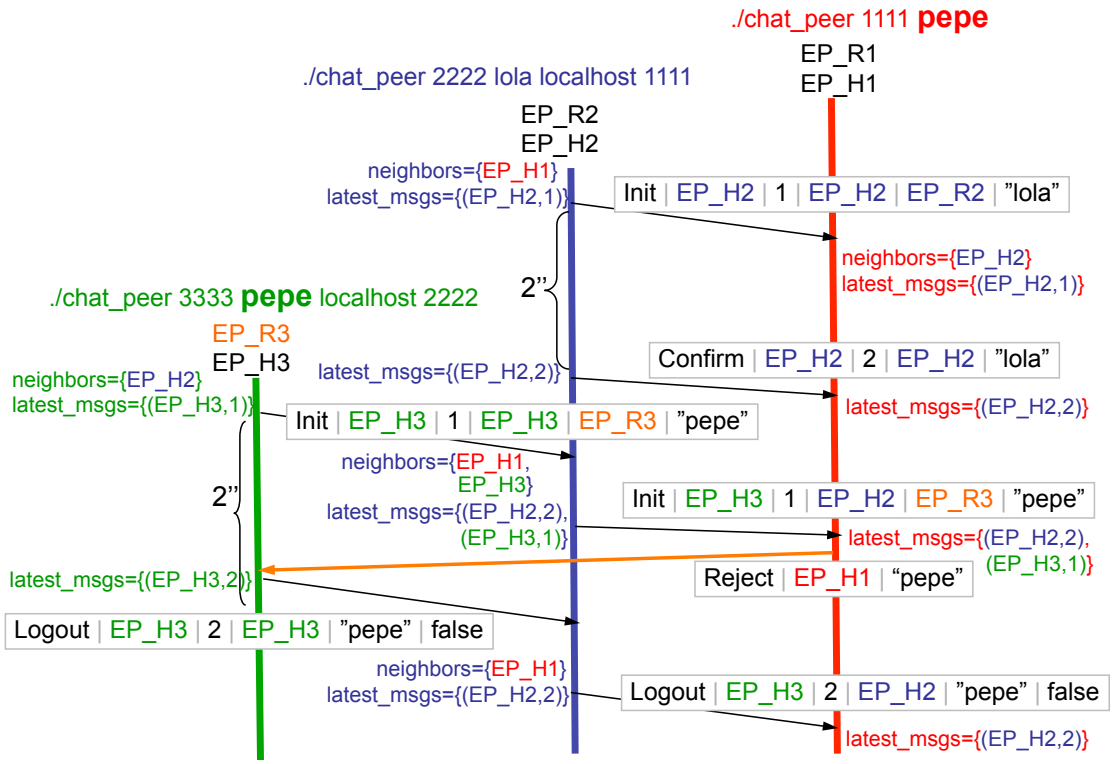

Figura 3: Ejemplo de protocolo de admisión en el que un nodo es rechazado por apodo duplicado

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

- Un nodo leerá de la entrada estándar cadenas de texto y las enviará al resto de nodos mediante el protocolo de inundación controlada en **mensajes *Writer***.
- Cada nodo que reciba un mensaje *Writer* lo tratará como cualquier otro mensaje enviado por inundación, reenviándolo y procesándolo si es preciso. En el caso del *Writer* su procesamiento específico consitirá en:
  - mostrar en pantalla las cadenas de texto recibidas

2.6. Protocolo de salida

- Cuando un usuario quiera abandonar Chat-Peer ejecutará el comando `.salir`. El nodo enviará por inundación un **mensaje *Logout***.
- Cada nodo que reciba un mensaje *Logout* lo tratará como cualquier otro mensaje enviado por inundación, reenviándolo y procesándolo si es preciso, con la excepción indicada en el punto siguiente. En el caso del *Logout* su procesamiento específico consitirá en:
  - comprobar si el nodo que ha creado el mensaje *Logout* le considera como vecino (campo `EP_H_Creat` = `EP_H_Rsnd`), en cuyo caso el nodo receptor del mensaje eliminará al emisor (`EP_H_Creat`) de su tabla *neighbors*.
  - se borrará de la tabla *latest_msgs* la entrada correspondiente al nodo que creó el mensaje *Logout* recibido (`EP_H_Creat`).
  - se mostrará en pantalla un mensaje informando de que el usuario cuyo nick aparece en el mensaje recibido ha abandonado Chat-Peer
- EXCEPCIÓN AL PROCESAMIENTO HABITUAL DE LA INUNDACIÓN PARA MENSAJES *LOGOUT*: Cuando un nodo reciba un *Logout* con un `EP_H_Creat` que no esté en su tabla *latest_msgs*, debe ignorarlo por completo. La razón es que dicho mensaje podría ser en realidad una repetición de un *Logout* ya visto anteriormente por ese nodo, pero que (como parte del procesamiento específico) terminó eliminando esa entrada de la tabla *latest_msgs*. Si el nodo volviera a aceptar este mensaje y a reenviarlo por inundación, podría crearse (en ciertos casos) una inundación infinita de un mensaje *Logout*: el mensaje nunca pararía de reenviarse porque a los nodos siempre les volviera a llegar y les pareciera un mensaje nuevo.

3. Formato de los mensajes

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

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

Dicho tipo deberá estar declarado en el `.ads` de algún paquete.

Salvo el tipo de mensaje `Reject`, todos los demás se envían por inundación, recibiéndolos cada par en su endpoint `EP_H` de recepción mediante *handler* de LLU.

El mensaje `Reject` NO se envía por inundación. Se envía directamente al *endpoint* `EP_R` de recepción bloqueante mediante LLU.Receive.

Mensaje *Init*

Es el que envía por inundación un nodo al arrancar como parte del protocolo de admisión.
Formato:

Init	EP_H_Creat	Seq_N	EP_H_Rsnd	EP_R_Creat	Nick

en donde:

- `Init`: Valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `EP_H_Creat`: `EP_H` del nodo que creó el mensaje. Dicho nodo recibe mensajes de inundación en este `End_Point`.
- `Seq_N`: Valor del tipo `Seq_N_T`. Es el número de secuencia asignado al mensaje por el nodo `EP_H_Creat`.

- EP_H_Rsnd: EP_H del nodo que ha reenviado el mensaje. Cuando un nodo recibe un mensaje por inundación, este campo identifica al vecino que le ha reenviado a él el mensaje recibido.
- EP_R_Creat: EP_R del nodo que creó el mensaje. Es a este endpoint al que hay que enviar mensajes *Reject* si es necesario.
- Nick: Unbounded_String con el *nick* del nodo que creó el mensaje.

Mensaje *Reject*

Es el que envía un nodo como parte del protocolo de admisión cuando al recibir un mensaje *Init* detecta que el creador del mensaje *Init* pretende utilizar su apodo. Este mensaje NO se envía por inundación. Se envía sólo al nodo creador del mensaje *Init*.
Formato:

Reject	EP_H	Nick

en donde:

- Reject: Valor del tipo Message_Type que identifica el tipo de mensaje.
- EP_H: *endpoint* EP_H para recepción mediante *handlers* de LLU del nodo que envía el mensaje.
- Nick: Unbounded_String con el *nick* del nodo.

Mensaje *Confirm*

Es el que envía por inundación un nodo al arrancar como parte del protocolo de admisión cuando no ha sido rechazado.
Formato:

Confirm	EP_H_Creat	Seq_N	EP_H_Rsnd	Nick

en donde:

- Confirm: Valor del tipo Message_Type que identifica el tipo de mensaje.
- EP_H_Creat: EP_H del nodo que creó el mensaje. Dicho nodo recibe mensajes de inundación en este End_Point.
- Seq_N: Valor del tipo Seq_N_T. Es el número de secuencia asignado al mensaje por el nodo EP_H_Creat.
- EP_H_Rsnd: EP_H del nodo que ha reenviado el mensaje. Cuando un nodo recibe un mensaje por inundación, este campo identifica al vecino que le ha reenviado a él el mensaje recibido.
- Nick: Unbounded_String con el *nick* del nodo que creó el mensaje.

Mensaje *Writer*

Es el que envía por inundación un nodo para comunicar a los demás nodos la cadena de texto leída del teclado.
Formato:

Writer	EP_H_Creat	Seq_N	EP_H_Rsnd	Nick	Text

en donde:

- Writer: Valor del tipo Message_Type que identifica el tipo de mensaje.
- EP_H_Creat: EP_H del nodo que creó el mensaje. Dicho nodo recibe mensajes de inundación en este End_Point.
- Seq_N: Valor del tipo Seq_N_T. Es el número de secuencia asignado al mensaje por el nodo EP_H_Creat.
- EP_H_Rsnd: EP_H del nodo que ha reenviado el mensaje. Cuando un nodo recibe un mensaje por inundación, este campo identifica al vecino que le ha reenviado a él el mensaje recibido.
- Nick: Unbounded_String con el *nick* del nodo que creó el mensaje.
- Text: Unbounded_String con el texto del mensaje.

Mensaje *Logout*

Es el que envía por inundación un nodo para comunicar a los demas nodos que abandona el Chat-Peer. Si lo envía por haber recibido un mensaje *Reject* durante el protocolo de admisión, pone el campo *Confirm_Sent* a `False`, ya que abandona sin haber llegado a enviar el mensaje *Confirm*. Si lo envía porque el usuario ha introducido el comando `.salir`, pone el campo *Confirm_Sent* a `True`.

Formato:

Logout	EP_H_Creat	Seq_N	EP_H_Rsnd	Nick	Confirm_Sent

en donde:

- `Init`: Valor del tipo `Message_Type` que identifica el tipo de mensaje.
- `EP_H_Creat`: `EP_H` del nodo que creó el mensaje. Dicho nodo recibe mensajes de inundación en este `End_Point`.
- `Seq_N`: Valor del tipo `Seq_N_T`. Es el número de secuencia asignado al mensaje por el nodo `EP_H_Creat`.
- `EP_H_Rsnd`: `EP_H` del nodo que ha reenviado el mensaje. Cuando un nodo recibe un mensaje por inundación, este campo identifica al vecino que le ha reenviado a él el mensaje recibido.
- `Nick`: `Unbounded_String` con el *nick* del nodo que creó el mensaje.
- `Confirm_Sent`: `Boolean` que indica si el nodo `EP_H_Creat` envió previamente un mensaje `Confirm` o no. Este campo permite distinguir si el *Logout* si se envía el *Logout* como parte del protocolo de admisión (valor `False`) o como parte del protocolo de salida (valor `True`).

4. Implementación

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

4.1. Implementación de *neighbors* y *latest_msgs* mediante paquetes genéricos

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

1. Partiendo del paquete `Maps_G` (disponible junto al tema de teoría 6: Tablas de Símbolos) que implementa una tabla de símbolos mediante una lista simplemente enlazada, deben realizarse en él los siguientes cambios:
   - Añadir a la especificación, como un nuevo parámetro genérico del paquete, el valor `Null_Key`, representando un valor nulo del tipo `Key_Type`.
   - Añadir a la especificación, como un nuevo parámetro genérico del paquete, el valor `Null_Value`, representando un valor nulo del tipo `Value_Type`.
   - Añadir a la especificación, como un nuevo parámetro genérico del paquete, el valor `Max_Length` representando el máximo número de elementos permitidos en el `Map`.
   - Añadir a la especificación del procedimiento `Put` un parámetro `Success` pasado en modo `out` que devuelva si pudo completarse con éxito el `Put`. Dicho parámetro devolverá `False` cuando no se pueda añadir un **nuevo** elemento a la tabla de símbolos por haber alcanzado ya el número de elementos `Max_Length`.
   - Añadir a la especificación una nueva función `Get_Keys`, que devolverá un array de `Max_Length` elementos con todas las `Keys` del `Map`. Si el `Map` tuviera menos de `Max_Length` elementos, se completaría el array con claves con valor `Null_Key`.
   - Añadir a la especificación la función `Get_Values`, que devolverá un array de `Max_Length` elementos con todos los `Values` del `Map`. Si el `Map` tuviera menos de `Max_Length` elementos, se completaría el array con claves con valor `Null_Value`.
   - Extender y modificar el cuerpo del paquete para que se adecúe a los cambios introducidos en la especificación.
   - Modificar la implementación del TAD `Map` para que se utilice una lista **doblemente enlazada** (cada celda contiene un puntero a la celda siguiente y otro puntero a la celda anterior).

Tras efectuar dichos cambios, la nueva especificación del paquete será la siguiente:

```
generic
   type Key_Type is private;
   type Value_Type is private;
   Null_Key: in Key_Type;
   Null_Value: in Value_Type;
   Max_Length: in Natural;
   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 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;
                  Success : out Boolean);

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

   type Keys_Array_Type is array (1..Max_Length) of Key_Type;

   function Get_Keys (M : Map) return Keys_Array_Type;

   type Values_Array_Type is array (1..Max_Length) of Value_Type;

   function Get_Values (M : Map) return Values_Array_Type;

   function Map_Length (M : Map) return Natural;

   procedure Print_Map (M : Map);

private

    ...

end Maps_G;
```

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

   - paquete `NP_Neighbors` para la tabla de vecinos. Como `Key_Type` se usará el tipo `LLU.End_Point_Type` y como `Value_Type` se usará el tipo `Ada.Calendar.Time`, de forma que para cada vecino de un nodo se almacene la hora a la que se le añadió a la tabla
   - paquete `NP_Latest_Msgs` para la tabla de últimos mensajes vistos. Como `Key_Type` se usará el tipo `LLU.End_Point_Type` y como `Value_Type` se usará el tipo `Seq_N_T`.

3. Tanto el programa principal del nodo como su manejador deberán acceder concurrentemente a los `Maps` que contienen las tablas *neighbors* y *latest_msgs*. 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 `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 Neighbors is new Maps_Protector_G (NP_Neighbors);

package Latest_Msgs is new Maps_Protector_G (NP_Latest_Msgs);
```

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

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

4. Los paquetes `Neighbors` y `Latest_Msgs` 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 `Neighbors` escribiendo `Chat_Handler.Neighbors`, y el paquete `Latest_Msgs` se utilizará escribiendo `Chat_Handler.Latest_Msgs`.

5. Las variables compartidas con la lista de vecinos y la lista de últimos mensajes también deberán ser declaradas en la especificación del paquete `Chat_Handler`, para poder ser usadas tanto desde el propio manejador como desde el programa principal. También puede situarse en la especificación del paquete `Chat_Handler` una variable para almacenar el valor del *nick* del nodo, de forma que pueda ser también usada tanto desde el propio manejador como desde el programa principal.

4.2. 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* y *latest_msgs* 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. En las figuras 4 y 5 se muestra un ejemplo de mensajes de depuración (en color verde) en dos nodos que están en Chat-Peer.

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

4.3. Orden de implementación

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

1. Completar el paquete `Maps_G` del tema de teoría para que tenga la especificación que se pide, incluyendo los nuevos subprogramas, y la modificación del `Put`, **pero sin cambiar aún la especificación a una lista doblemente enlazada**.

2. Escribir un pequeño programa de prueba aparte en que se realice las instanciaciones necesarias para obtener los paquetes `Neighbors` y `Latest_Msgs` (tal y como se explica en el apartado 4.1), y que compruebe el funcionamiento básico de las tablas.

3. Cambiar en `Maps_G` la implementación de la lista simplemente enlazada a una lista doblemente enlazada.

4. Volver a comprobar con el pequeño programa de prueba que todo sigue funcionando.

5. Copiar la instanciaciones de los paquetes a `chat_handler.ads`.

6. Escribir en `chat_peer.adb` y `chat_handler.adb` el código necesario para implementar sólo el protocolo de admisión, incluyendo el envío y reenvío por inundación de los mensajes que intervienen en él.

7. Escribir el código del protocolo de envío y recepción de los mensajes `Writer`.
8. Escribir el código del protocolo de salida.

5. 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. En Chat-Peer se supondrá que nunca se perderán los mensajes enviados los nodos.
4. En Chat-Peer **se supondrá antes de lanzar un nodo han sido previamente arrancados los nodos vecinos iniciales que se especifican en su línea de comandos**.
5. Se supondrá que los nodos no pueden terminar con CTRL-C, sino que siempre lo harán con `.salir`.
6. Se supondrá que nunca terminará un nodo de forma que deje a vecinos suyos "desconectados" del chat al no quedarles ningún vecino arrancado.
7. 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.
8. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.

6. Normas de entrega

6.1. Alumnos de Programación de Sistemas de Telecomunicación

Debes dejar el código fuente de tu programa dentro de tu cuenta en los laboratorios de prácticas. Directamente en el directorio de tu cuenta debes crear una carpeta de nombre `PST.2014`, y dentro de ella una carpeta de nombre `P4`. En ella debe estar el fichero correspondiente al programa `chat_peer.adb` y los ficheros de los paquetes auxiliares.
Importante: Pon un comentario con tu nombre completo en la primera línea de cada fichero fuente.

6.2. Alumnos de Informática II

Debes dejar el código fuente de tu programa dentro de tu cuenta en los laboratorios de prácticas. Directamente en el directorio de tu cuenta debes crear una carpeta de nombre `I2.2014`, y dentro de ella una carpeta de nombre `P4`. En ella debe estar el fichero correspondiente al programa `chat_peer.adb` y los ficheros de los paquetes auxiliares.
Importante: Pon un comentario con tu nombre completo en la primera línea de cada fichero fuente.

7. Plazo de entrega

Los ficheros deben estar en tu cuenta, en la carpeta especificada más arriba, antes de las 23:59h del **miércoles 26 de noviembre de 2014**.

```
$ ./chat_peer 1111 pepe
NOT following admission protocol because we have no initial contacts ...
Chat_Peer
=========

Logging into chat with nick: pepe
.h for help
RCV Init 127.0.1.1:2222  1 127.0.1.1:2222 ... ana
    Adding to neighbors 127.0.1.1:2222
    Adding to latest_messages 127.0.1.1:2222 1
    FLOOD Init 127.0.1.1:2222  1 127.0.1.1:1111 ...  ana

RCV Confirm 127.0.1.1:2222  2 127.0.1.1:2222 ana
ana joins the chat
    Adding to latest_msgs 127.0.1.1:2222 2
    FLOOD Confirm 127.0.1.1:2222  2 127.0.1.1:1111 ana

RCV Init 127.0.1.1:4444  1 127.0.1.1:4444 ... manolo
    Adding to neighbors 127.0.1.1:4444
    Adding to latest_messages 127.0.1.1:4444 1
    FLOOD Init 127.0.1.1:4444  1 127.0.1.1:1111 ...  manolo
        send to: 127.0.1.1:2222

RCV Confirm 127.0.1.1:4444  2 127.0.1.1:4444 manolo
manolo joins the chat
    Adding to latest_msgs 127.0.1.1:4444 2
    FLOOD Confirm 127.0.1.1:4444  2 127.0.1.1:1111 manolo
        send to: 127.0.1.1:2222

RCV Confirm 127.0.1.1:4444  2 127.0.1.1:2222 manolo
    NOFLOOD Confirm 127.0.1.1:4444  2 127.0.1.1:2222 manolo

RCV Writer 127.0.1.1:4444  3 127.0.1.1:4444 manolo hola
manolo: hola
    Adding to latest_msgs 127.0.1.1:4444 3
    FLOOD Writer 127.0.1.1:4444  3 127.0.1.1:1111 manolo hola
        send to: 127.0.1.1:2222

RCV Writer 127.0.1.1:4444  3 127.0.1.1:2222 manolo hola
    NOFLOOD Writer 127.0.1.1:4444  3 127.0.1.1:2222 manolo
```

Figura 4: Peer con mensajes de depuración en color verde

```
$ ./chat_peer 4444 manolo myhost 1111 myhost 2222
Adding to neighbors 127.0.1.1:1111
Adding to neighbors 127.0.1.1:2222

Admission protocol started ...
Adding to latest_messages 127.0.1.1:4444 1
FLOOD Init 127.0.1.1:4444  1 127.0.1.1:4444 ...  manolo
        send to: 127.0.1.1:1111
        send to: 127.0.1.1:2222

Adding to latest_msgs 127.0.1.1:4444 2
FLOOD Confirm 127.0.1.1:4444  2 127.0.1.1:4444 manolo
        send to: 127.0.1.1:1111
        send to: 127.0.1.1:2222

Admission protocol finished.

Chat_Peer
=========

Logging into chat with nick: manolo
.h for help
hola
Adding to latest_msgs 127.0.1.1:4444 3
FLOOD Writer 127.0.1.1:4444  3 127.0.1.1:4444 manolo hola
        send to: 127.0.1.1:1111
        send to: 127.0.1.1:2222
```

Figura 5: Peer con mensajes de depuración en color verde