

SISTEMA DE MENSAJERÍA INSTANTÁNEA

MEMORIA DE LA PRÁCTICA

Alberto Miedes Garcés

Denys Sypko

PROGRAMACIÓN DE SISTEMAS DISTRIBUIDOS
UNIVERSIDAD COMPLUTENSE DE MADRID



Índice de contenidos

1. Introducción y aspectos generales	3
1.1. Funcionamiento general	3
1.2. Decisiones de diseño	4
2. Servicios disponibles	5
2.1. Estructuras de datos utilizadas	5
2.2. Gestión de usuarios y sesiones	6
2.3. Gestión de amigos y peticiones de amistad	7
2.4. Gestión de mensajes y confirmación de recepción	8
3. Descripción del cliente	9
3.1. Estructuras de datos utilizadas	9
3.2. Variables globales	9
3.3. Diseño de la interfaz de usuario	9
3.4. Comunicación con el servidor / servicios	10
4. Descripción del servidor	12
4.1. Estructuras de datos utilizadas	12
4.2. Constantes	12
4.3 Variables globales	12
4.4. Gestión de usuarios, amigos y peticiones de amistad	13
4.5. Gestión de mensajes	14
4.6. Gestión de ficheros y directorios	14
4.7. Caducidad de sesiones y SIGALARM	15
4.8. Concurrencia	15
4.9. Captura de SIGINT y persistencia	15
4.10. Gestión de bajas	16

1. Introducción y aspectos generales

1.1. Funcionamiento general

Esta práctica implementa un servicio de mensajería instantánea basado en una arquitectura cliente-servidor con servicios web implementados mediante la librería **gSOAP** de C.

Al compilar se generan dos ejecutables: client y server, que podrían ser ejecutados en la misma máquina o en máquinas diferentes.

- Instrucciones de compilación y ejecución para el servidor:

1. Generamos los stubs a partir de la interfaz remota (fichero ims.h):

```
> soapcpp2 -c -S ims.h
```

2. Compilamos

```
> make server
```

** Generará un ejecutable llamado 'servidor'*

3. Ejecución:

```
> ./server [puerto]
```

Por ejemplo:

```
> ./server 5000
```

- Instrucciones de compilación y ejecución para el cliente:

1. Generamos los stubs a partir de la interfaz remota (fichero ims.h):

```
> soapcpp2 -c -C ims.h
```

2. Compilamos

```
> make client
```

** Generará un ejecutable llamado 'cliente'*

3. Ejecución:

```
> ./client [URL + puerto]
```

Por ejemplo:

```
> ./client http:192.168.0.35:5000
```

El makefile original proporcionado con el esqueleto de la práctica fue modificado para que contemplase los nuevos archivos .c y .h que creamos.

1.2. Decisiones de diseño

Archivos de código

El código de nuestra aplicación se divide en los siguientes 12 archivos:

Archivo	Contenido y funciones
ims.h	Contiene las constantes utilizadas por cliente y servidor, las estructuras utilizadas como parámetros en los servicios gSOAP y las cabeceras de dichos servicios.
client.c	Contiene el código relativo al cliente.
server.c	Contiene el código que lanza el servidor y lo pone a la escucha de peticiones.
s_services.h	Contiene la implementación de los servicios gSOAP definidos en ims.h
s_usuarios.h s_usuarios.c	Contienen las estructuras de datos, funciones para su manipulación y persistencia en ficheros relativas a la gestión de los usuarios.
s_amigos.h s_amigos.c	Contienen las estructuras de datos, funciones para su manipulación y persistencia en ficheros relativas a la gestión de las listas de amigos de cada usuario, así como de las peticiones de amistad.
s_mensajes.h s_mensajes.c	Contienen las estructuras de datos, funciones para su manipulación y persistencia en ficheros relativas a la gestión de los mensajes enviados entre usuarios.
externo.h externo.c	Contienen varias funciones útiles utilizadas tanto por el cliente como el servidor.

Decisiones relativas a temporizadores y la captura de señales

La única señal que capturamos es el *SIGALARM* lanzado por los temporizadores encargados de manejar la caducidad de las sesiones de los usuarios.

Aunque en el enunciado de la práctica se habla de **‘capturar Ctrl+C’** tanto en el cliente como en el servidor, en el punto 4.9 se explica por qué hemos decidido hacerlo de otra forma.

2. Servicios disponibles

2.1. Estructuras de datos utilizadas

La estructura `Message2` representa cada uno de los mensajes enviados de un usuario a otro.

```
struct Message2 {
    char* receptor;
    char* emisor;
    char* msg;
};
```

La estructura `IMS_PeticionAmistad` representa la petición de amistad que un usuario envía a otro.

```
struct IMS_PeticionAmistad {
    char* emisor;
    char* receptor;
};
```

La estructura `RespuestaPeticionAmistad` representa la respuesta de un usuario a alguna de las peticiones de amistad que ha recibido.

```
struct RespuestaPeticionAmistad {
    char* emisor;        // Quien envió la petición de amistad original.
    char* receptor;      // Quien responde a ella.
    int aceptada;        // 1 si aceptada, 0 si denegada
};
```

La estructura `ResultMsg` se utiliza como tipo de retorno genérico para llamadas gSOAP que no deben devolver ningún dato en especial. En code se guarda un 0 si la llamada tuvo éxito o un número negativo en caso de error. msg guarda un mensaje asociado al error, si lo hubiese.

Como los servicios gSOAP sólo permiten un único parámetro de retorno, en los casos en los que era necesario devolver más datos a parte de un código y mensaje de error, hemos decidido encapsular los dos atributos de `ResultMsg` dentro de la estructura pertinente para cada caso (`ListaPeticiones`, `ListaAmigos`, `ListaMensajes`).

```
struct ResultMsg {
    int code;        // Código de error (0 si éxito, < 0 si error)
    xsd__string msg; // Mensaje asociado al código de error.
};
```

La estructura `ListaPeticiones` representa una lista con todas las peticiones de amistad nuevas, para un cliente concreto.

```
struct ListaPeticiones {
    int size;        // Número de peticiones en la lista
    xsd__string peticiones; // Nombres de las personas, separados por ' '
    int code;        // Código de error o éxito
    xsd__string msg; // Mensaje asociado al código
};
```

La estructura `ListaAmigos` contiene una lista con todos los amigos de un usuario.

```
struct ListaAmigos {
    int size;           // Número de amigos en la lista
    xsd_string amigos;  // Nombres de los amigos, separados por ' '
    int code;           // Código de error o éxito
    xsd_string msg;     // Mensaje asociado al código
};
```

La estructura `ListaMensajes` contiene todos los mensajes pendientes para un usuario.

```
struct ListaMensajes {
    int size;           // Número de mensajes
    xsd_string mensajes; // Mensajes pendientes separados por '\n'
    int code;           // Código de error o éxito
    xsd_string msg;     // Mensaje asociado al código
};
```

2.2. Gestión de usuarios y sesiones

Darse de alta en el sistema

Da de alta a un nuevo usuario en el sistema.

```
int ims__darAlta (char* username, struct ResultMsg *result);
```

- *username* → Nombre del usuario que se registra
- *result* → Resultado de la llamada

Darse de baja en el sistema

Da de baja en el sistema a un usuario.

```
int ims__darBaja(char* username, struct ResultMsg *result);
```

- *username* → Nombre del usuario que se da de baja
- *result* → Resultado de la llamada

Iniciar sesión

Hace login en el sistema.

```
int ims__login (char* username, struct ResultMsg *result);
```

- *username* → Nombre del usuario que hace login
- *result* → Resultado de la llamada

Cerrar sesión

Cierra la sesión de un usuario en el sistema.

```
int ims__logout (char* username, struct ResultMsg *result);
```

- *username* → Nombre del usuario que cierra la sesión.
- *result* → Resultado de la llamada

2.3. Gestión de amigos y peticiones de amistad

Enviar una petición de amistad

Envía una petición de amistad a un usuario. Pide el nombre del usuario destino al cliente y manda la petición al servidor.

result puede devolver un código de error si ya somos amigos de dicho usuario, existe una petición equivalente a esta en la dirección opuesta, o el usuario no existe.

```
int ims__sendFriendRequest (struct IMS_PeticionAmistad p, struct ResultMsg* result);
```

- *p* → Estructura con la petición de amistad (emisor y destinatario).
- *result* → Resultado de la llamada

Enviar una petición de amistad

Envía al servidor (y posteriormente se la pedirá el destinatario) la respuesta a una petición de amistad (aceptada o declinada).

```
int ims__answerFriendRequest (struct RespuestaPeticionAmistad rp, struct ResultMsg *result);
```

- *rp* → Estructura con la respuesta a la petición de amistad (emisor, destinatario, aceptada (1) o rechazada (0)).
- *result* → Resultado de la llamada

Recibir todas las peticiones de amistad pendientes

Obtiene todas las peticiones de amistad pendientes de contestar.

```
int ims__getAllFriendRequests (char* username, struct ListaPeticiones *result);
```

- *username* → Nombre del usuario que pide las peticiones
- *result* → Resultado de la llamada

Recibir la lista de amigos

Obtiene la última versión de la lista de amigos de un usuario.

```
int ims__getFriendList(char* username, struct ListaAmigos* result);
```

- *username* → Usuario que pide su lista de amigos
- *result* → Resultado de la llamada

2.4. Gestión de mensajes y confirmación de recepción

Enviar un mensaje a un usuario

Envía un mensaje a un usuario. Éste se almacena (en un fichero) en el servidor hasta que el receptor lo pide.

```
int ims__sendMessage (struct Message2 myMessage, struct ResultMsg *result);
```

- *myMessage* → Mensaje a enviar (contiene emisor, receptor, mensaje).
- *result* → Resultado de la llamada

Recibir todos los mensajes pendientes

Obtiene todos los mensajes pendientes de un usuario.

```
int ims__receiveMessage (char* username, struct ListaMensajes* result);
```

- *username* → Nombre del usuario cuyos mensajes queremos obtener.
- *result* → Resultado de la llamada.

Consultar el estado de la entrega de los mensajes enviados

Permite obtener la información relativa a cuáles de nuestros mensajes enviados han llegado ya correctamente al receptor.

```
int ims__consultarEntrega(char* username, struct ListaMensajes* result);
```

- *username* → Nombre del usuario que solicita el *double-check* de sus mensajes enviados.
- *result* → Resultado de la llamada

3. Descripción del cliente

3.1. Estructuras de datos utilizadas

La única estructura necesaria ha sido una que almacene una lista con los amigos del usuario, para que a la hora de enviar un mensaje podamos comprobar si el destinatario se encuentra dentro de nuestras amistades (aunque también se comprueba en el servidor).

```
struct MisAmigos {  
    int nElems;  
    char amigos[IMS_MAX_AMIGOS][IMS_MAX_NAME_SIZE];  
};
```

3.2. Variables globales

Existen algunas variables en el cliente, que por estar utilizándose continuamente y evitarnos pasar todo el rato como parámetros, hemos decidido declararlas como variables globales:

- Contexto gsoap:

```
struct soap soap;
```

- Nombre del usuario que utiliza el servicio. Se establece su valor al hacer login:

```
char username_global[IMS_MAX_NAME_SIZE];
```

- Lista de amigos del usuario:

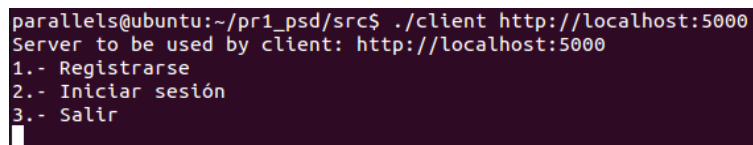
```
struct MisAmigos mis_amigos;
```

- URL del servidor con el que nos estamos comunicando:

```
char* serverURL;
```

3.3. Diseño de la interfaz de usuario

La interfaz de usuario está hecha a través de la línea de comandos. Existen varios menús que permiten al usuario acceder al funcionamiento de la aplicación.



```
parallels@ubuntu:~/pr1_psd/src$ ./client http://localhost:5000  
Server to be used by client: http://localhost:5000  
1.- Registrarse  
2.- Iniciar sesión  
3.- Salir  
█
```

El primer menú que aparece al ejecutar el cliente nos permite registrarnos, hacer login o salir.

Suponiendo que ya tengamos el usuario creado, si hacemos login aparecerá un nuevo menú que ya nos permite acceder a las funciones principales de la aplicación (envío de mensajes, gestión de amistades...).

```

parallels@ubuntu:~/pr1_psd/src$ ./client http://localhost:5000
Server to be used by client: http://localhost:5000
1.- Registrarse
2.- Iniciar sesión
3.- Salir
2
Nombre de usuario:usuario1
Login correcto.
1.- Enviar mensaje a otro usuario
2.- Consultar mensajes
3.- Consultar la entrega
4.- Enviar petición de amistad
5.- Consultar peticiones de amistad
6.- Ver amigos
7.- Dar de baja
8.- Cerrar sesión

```

3.4. Comunicación con el servidor / servicios

En general, cada opción del menú del cliente corresponde con un servicio del servidor (aunque en los casos en los que es necesario hacer alguna comprobación previa, puede haber más de una invocación a gSOAP).

La mayoría de las funciones que invocan a los servicios gSOAP siguen el mismo procedimiento:

- 1º) Reservar memoria (dinámica y/o estática) para los datos y estructuras que se envían al servidor.
- 2º) Pedir al usuario los datos necesarios, e introducirlos dentro de las estructuras que se van a enviar.
- 3º) Invocar al servicio gSOAP, poniendo como parámetros los datos que queremos enviar y reservando el último de ellos para los datos de retorno.
- 4º) Liberar la memoria dinámica utilizada para los datos enviados.
- 5º) Comprobar el éxito o fracaso de la llamada.
- 6º) Comprobar el resultado de la llamada. Es distinto al punto anterior, pues un fallo en el 5 indicaría la imposibilidad de comunicarnos con el servidor mientras que lo que comprobamos en este punto es si los datos que enviamos/solicitamos pudieron ser procesados correctamente.

Como ejemplo representativo, éste es el código que se ejecuta cuando queremos enviar una petición de amistad a un usuario:

```

/**
 * Envía una petición de amistad al usuario que indiquemos.
 */
void sendFriendRequest() {

    struct IMS_PeticionAmistad pet;
    char receptor[IMS_MAX_NAME_SIZE];
    struct ResultMsg res;

    // 1. Poner el emisor
    pet.emisor = malloc (IMS_MAX_NAME_SIZE);
    strcpy (pet.emisor, username_global);

    // 2. Poner el receptor
    printf("Destinatario: ");

```

```

scanf("%255s", receptor);
receptor[strlen(receptor)] = '\0';
clean_stdin();
pet.receptor = malloc (IMS_MAX_NAME_SIZE);
strcpy(pet.receptor, receptor);

// 3. Llamada gSOAP
soap_call_ims__sendFriendRequest(&soap, serverURL, "", pet, &res);

// 4. Liberar memoria
free(pet.emisor);
free(pet.receptor);

// 5. Comprobar errores de gSOAP
if (soap.error) {
    soap_print_fault(&soap, stderr);
    exit(1);
}

// 6. Resultado de la llamada
printf("%s\n", res.msg);
}

```

4. Descripción del servidor

4.1. Estructuras de datos utilizadas

No existe ninguna estructura de datos declarada, propiamente dicho, en servidor.c No obstante, las estructuras de datos de las que hace uso se explicarán en las secciones concretas (4.4, 4.5 y 4.6).

4.2. Constantes

Existen tres constantes declaradas en servidor.c Las dos primeras se utilizan para alternar los dos tipos de temporizadores (SIGALARM) que utilizamos.

```
#define ALARMA_TIRAR_SESIONES 0 /* SIGALARM handler, type 1 */  
#define ALARMA_CHECK_SESIONES 1 /* SIGALARM handler, type 2 */
```

La tercera indica el tiempo máximo que permanece activa la sesión de un cliente desde que éste realizó su última petición al servidor.

```
#define SESSION_DURATION 500 /* Client session duration */
```

4.3 Variables globales

Existen algunas variables en el cliente, que por estar utilizándose continuamente y evitarnos pasar todo el rato como parámetros, hemos decidido declararlas como variables globales:

- Estructura con todos los usuarios dados de alta (o que lo estuvieron):
`struct ListaUsuarios lu;`
- Estructura con las peticiones de amistad pendientes de entregar a algún usuario:
`struct ListaAmistadesPend ap;`
- Estructura con la lista de amigos de cada uno de los usuarios dados activos en el sistema:
`struct ListasAmigos la;`
- Estructura que guarda todos los mensajes enviados (de todas las conversaciones):
`struct ListasMensajes lmsg;`
- Las siguientes variables se utilizan para llevar a cabo todo lo relacionado con la expiración de sesiones:

```
volatile int sesiones_expiradas [MAX_USERS];  
volatile sig_atomic_t check_sessions;  
volatile sig_atomic_t TIPO_ALARMA;  
volatile sig_atomic_t exit_thread;
```

- También hacemos uso de un mutex para controlar el acceso a las variables globales por parte de los distintos threads que se lanzan desde el servidor:
`pthread_mutex_t mtx;`

4.4. Gestión de usuarios, amigos y peticiones de amistad

Para la gestión de los usuarios el servidor hace uso de las siguientes estructuras:

```
struct Usuario {
    int baja;
    int connected;
    char username[IMS_MAX_NAME_SIZE];
};

struct ListaUsuarios {
    int size; // Nº de usuarios
    struct Usuario usuarios[MAX_USERS]; // Array de usuarios
};
```

La primera representa a un único usuario y la segunda a la totalidad de ellos.

Cuando arranca el servidor, el contenido de estas estructuras se lee desde el fichero *lista_usuarios.txt* y se carga en la memoria del servidor. A partir de este instante todas las manipulaciones que se hagan sobre estos datos se harán sobre la versión residente en memoria, no la del fichero.

Para guardar la lista de amigos de cada usuario se utilizan las siguientes estructuras:

```
/* Estructura que almacena la lista de amigos de un usuario */
struct AmigosUsuario {
    int size; // Nº amigos del usuario
    char usuario[IMS_MAX_NAME_SIZE]; // Nombre del usuario
    char amigos[IMS_MAX_AMIGOS][IMS_MAX_NAME_SIZE]; // Lista de amigos
};

/* Estructura que almacena la lista de amigos de cada usuario. */
struct ListasAmigos {
    int size; // Nº usuarios del sistema
    struct AmigosUsuario listas [MAX_USERS]; // Lista de amigos de cada usuario
};
```

Para guardar las peticiones de amistad pendientes de ser respondidas:

```
/* Estructura que representa una petición de amistad */
struct PeticionAmistad {
    char emisor[IMS_MAX_NAME_SIZE];
    char destinatario[IMS_MAX_NAME_SIZE];
};

/* Estructura del servidor que almacena todas las peticiones de amistades pendientes de procesar. */
struct ListaAmistadesPend {
    int size;
    struct PeticionAmistad peticiones[MAX_AMISTADES_PENDIENTES];
};
```

4.5. Gestión de mensajes

Para gestionar los mensajes en el servidor utilizamos dos estructuras una que contiene los campos del emisor, receptor y el mensaje y la otra es la lista de dichos mensajes.

Al enviar un mensaje el servidor lo guarda en la estructura y en el fichero de mensajes_pendientes.txt del usuario receptor.

Una vez recibido el mensaje por el receptor se borra de los mensajes pendientes y se le concatena un * para indicar al emisor que su amigo ha recibido el mensaje.

```
struct Mensaje {
    char emisor [IMS_MAX_NAME_SIZE];
    char receptor [IMS_MAX_NAME_SIZE];
    char msg [IMS_MAX_MSG_SIZE];
};

struct ListasMensajes{
    int size;
    struct Mensaje lista [MAX_MENSAJES];
};
```

4.6. Gestión de ficheros y directorios

El servidor hace uso de cuatro ficheros para almacenar de forma permanente toda la información que gestiona:

- usuarios.txt

Almacena el nombre de los usuarios del sistema, así como un flag para indicar si se han dado de baja o no (para impedir la reutilización de su nombre de usuario).

- listas_amigos.txt

Almacena para cada usuario, una lista con sus amigos.

- peticiones_pendientes.txt

Almacena todas las peticiones de amistad que todavía no han sido respondidas y por lo tanto no pueden ser eliminadas del servidor.

- mensajes_enviados.txt

Almacena todos los mensajes que se han enviado.

En el caso de que alguno de estos cuatro ficheros no exista en el momento de arrancar el servidor, se crea.

Estos cuatro ficheros están dentro del directorio '*Servidor*'.

Para cada usuario del sistema (que no se haya dado de baja) mantenemos un directorio dentro del servidor (que lleva su nombre) que contiene un fichero con todos los mensajes pendientes de serle entregados.

4.7. Caducidad de sesiones y SIGALARM

Para gestionar la caducidad de las sesiones de los clientes mantenemos una variable global (array de int) en el servidor que indica para cada usuario, si debemos forzar su cierre de sesión (poner su atributo *connected* a 0).

```
volatile int sesiones_expiradas [MAX_USERS];
```

El proceso que se sigue es el siguiente:

1. Un temporizador hace saltar de forma alternativa dos alarmas: *ALARMA_TIRAR_SESIONES* y *ALARMA_CHECK_SESIONES*.
 - En el handler de la primera, nos encargamos de poner a 1 todo el vector *sesiones_expiradas*.
 - En el handler de la segunda, recorremos dicho vector y cerramos la sesión de todos los usuarios que tengan su componente del vector correspondiente a 1.
2. En el servidor, con el objetivo de evitar que caduquen la sesiones de los clientes que están activos, cada vez que atendemos una petición nos encargamos de poner a 0 la componente de *sesiones_expiradas* del usuario que haya realizado dicha petición, de modo que cuando salte *ALARMA_CHECK_SESIONES* no expulsemos a este usuario.
3. Cuando el servidor recibe una petición de un cliente, lo primero que hace es comprobar si la sesión de dicho cliente está todavía activa, devolviéndole el mensaje de error correspondiente en el caso de que no lo esté. De este modo evitamos que clientes con una sesión no iniciada o expirada lleven a cabo acciones.

4.8. Concurrencia

Para hacer concurrente el servido hemos utilizado el ejemplo

https://www.cs.fsu.edu/~engelen/soapdoc2.html#tth_sEc7.2.4

que aparece en el manual de gSoap pero añadiendo un mutex para hacer la exclusión mutua para evitar el ingreso a secciones críticas por más de un proceso a la vez.

La secciones críticas son las Estructuras que tenemos como variables globales en el servidor.

4.9. Captura de SIGINT y persistencia

• Captura de SIGINT en el cliente

En este caso debemos asegurarnos de que antes de salir del programa, finalicen las llamadas al servidor que estén siendo procesadas, en el caso de que haya alguna.

Para lograr esto, en vez de capturar la señal de SIGINT lo que hemos hecho ha sido enmascarar dicha señal justo antes de realizar una llamada a los servicios remotos, y desenmascararla cuando éste acabe.

```
// Enmascarar SIGINT
sigprocmask(SIG_BLOCK, &grupo, NULL);

switch(opcion) {
    case '1':
```

```

        enviarMensaje();
        break;
    case 'n':
        ...
    default:
        break;
}

// Desenmascarar SIGINT
sigprocmask(SIG_UNBLOCK, &grupo, NULL);

```

• Captura de SIGINT en el servidor

En el servidor debemos asegurarnos de que antes de salir del programa guardamos el contenido de todas las estructuras de datos en sus correspondientes ficheros y que acabamos de atender todas las peticiones de clientes que estén todavía en curso.

- Para guardar los datos del servidor en ficheros antes de salir podríamos haber utilizado un *handler* que capturase la señal SIGINT y que se encargase de llevar a cabo esta tarea. El problema que presenta esta solución es que (*según la documentación acerca de los manejadores de señal*), todas las operaciones que se efectúen dentro de un *signal handler* deben ser **reentrant**, o lo que es lo mismo, que en el caso de que ser interrumpidas en mitad de su ejecución puedan ser llamadas de nuevo sin que se produzcan efectos adversos.
- Existe una lista de funciones que presentan esta cualidad, pero como las que necesitábamos utilizar nosotros no estaban entre ellas, lo hemos solucionado de otra forma: justo después de atender la petición de un cliente, volcamos todo el contenido de las estructuras del servidor en los ficheros.
- Para evitar interrumpir las peticiones de clientes se sigue la misma estrategia que en el caso del cliente, enmascaramos SIGINT justo antes de ponernos a atender una petición y la desenmascaramos cuando esta finaliza.

4.10. Gestión de bajas

El sistema contempla la posibilidad de que un usuario se de de baja. Si éste lo hace, se marca **baja = 1** en su correspondiente **struct Usuario**.

Al dar de baja a un usuario, se borra su lista de amigos y se le elimina de las listas de amigos del resto de usuarios.

Además, se borra el directorio del usuario y todo su contenido (los mensajes pendientes para él). También se elimina de la estructura **struct ListasMensajes lmsg;** del servidor todos los mensajes enviados por este usuario.

Aunque un usuario se de de baja, su nombre de usuario no es eliminado del servidor para evitar que nuevos usuarios se den de alta con el nombre de éste.