



UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
Año 2010 - 1<sup>er</sup> Cuatrimestre

## TÉCNICAS DE PROGRAMACIÓN CONCURRENTE (75.59)

### TRABAJO PRÁCTICO: CONCU-CHAT

#### Integrantes

Apellido, Nombre	Nro. Padrón	E-mail
Ferreiro, Demian	88443	epidemian@gmail.com
Mouso, Nicolás Gastón	88528	nicolasgnr@gmail.com

# Índice

# 1. Enunciado

## Objetivo

El objetivo de este trabajo es el desarrollo de un sistema de chat conocido como "ConcuChat". Este es un sistema que permitirá a dos partes poder chatear mediante el intercambio de mensajes instantáneos. Esto quiere decir que ambas partes podrán comunicarse únicamente si ambas están "en línea".

## Arquitectura

El sistema deberá estar formado por los siguientes módulos:

1. Servicio de Localización
2. Programa de Chat

## Servicio de Localización

Es un módulo que se encarga de almacenar el nombre y la dirección de los distintos programas de chat que estén en ejecución, así como de realizar la resolución entre nombre y dirección. Para ello contará con dos funciones:

1. Registrar un nuevo programa de chat
2. Resolver la dirección de un programa de chat dado su nombre

## Programa de Chat

Es el módulo que utilizarán los usuarios para comunicarse entre sí. Cuando el módulo de chat se inicia, lo primero que hace es registrarse en el Servicio de Localización para que pueda ser localizado por otros módulos de chat. Si el usuario quiere chatear con otro usuario, debe primero localizar a su contraparte utilizando el Servicio de Localización. Una vez localizada a la contraparte se podrá comenzar el chat.

## Requerimientos funcionales

### Servicio de Localización

1. Es el primer módulo del sistema que se ejecuta. Si el Servicio de Localización no está activo, el sistema no funciona.
2. Permite registrar un par nombre – dirección. Los nombres son únicos, por lo tanto debe rechazar una petición de registro si el nombre ya se encuentra registrado.
3. Permite consultar la dirección correspondiente a un nombre.
4. Permite desregistrar un par nombre – dirección.

5. No se requiere mantener un tiempo de vida de los registros, pero sí se debe poder eliminar un registro manualmente por línea de comandos. Si un programa de chat no puede desregistrarse por algún motivo, entonces el operador deberá poder eliminar el registro mediante la ejecución de un comando.

### **Programa de Chat**

1. Al iniciarse, el módulo de chat debe registrarse en el Servicio de Localización.
2. Para conectarse con el Servicio de Localización, el programa de chat debe conocer la dirección de éste.
3. Cuando el usuario finaliza la aplicación, debe desregistrarse del Servicio de Localización.
4. El usuario debe poder indicar al programa que desea comunicarse con otra persona. Para ello debe consultar al Servicio de Localización cuál es la dirección correspondiente al nombre de la otra parte.
5. Debe poder recibir mensajes de otros programas de chat.
6. Debe poder enviar mensajes a otros programas de chat.
7. El envío y la recepción de mensajes deben ser en forma simultánea.
8. No es necesario implementar el corte de la comunicación. Para terminar el chat actual y chatear con otra persona se puede cerrar y volver a abrir el programa de chat.

### **Requerimientos no funcionales**

1. El trabajo práctico deberá ser desarrollado en lenguaje C o C++, siendo este último el lenguaje de preferencia.
2. Los módulos pueden no tener interfaz gráfica y ejecutarse en una o varias consolas de línea de comandos.
3. El trabajo práctico deberá funcionar en ambiente Unix / Linux.
4. La aplicación deberá funcionar en una única computadora.
5. Cada módulo deberá poder ejecutarse en “modo debug”, lo cual dejará registro de la actividad que realiza en uno o más archivos de texto para su revisión posterior.

### **Esquema de direcciones**

Cada módulo del sistema se identifica unívocamente por una dirección, es decir, cada vez que se requiere referenciar a un módulo determinado para (por ejemplo) enviarle un mensaje, se lo debe hacer mediante su dirección. Sin embargo, las direcciones no suelen ser prácticas para las personas, por lo cual se definen nombres para cada módulo, y un esquema de resolución nombre dirección, el cual se implementa en el Servicio de Localización.

## Tareas a realizar

A continuación se listan las tareas a realizar para completar el desarrollo del trabajo práctico:

1. Dividir cada módulo en procesos. El objetivo es lograr que cada módulo esté conformado por un conjunto de procesos que sean lo más sencillos posible.
2. Una vez obtenida la división en procesos, establecer un esquema de comunicación entre ellos teniendo en cuenta los requerimientos de la aplicación. ¿Qué procesos se comunican entre sí? ¿Qué datos necesitan compartir para poder trabajar?
3. Tratar de mapear la comunicación entre los procesos a los problemas conocidos de concurrencia.
4. Determinar los mecanismos de concurrencia a utilizar para cada una de las comunicaciones entre procesos que fueron detectados en el ítem 2. No se requiere la utilización de algún mecanismo específico, la elección en cada caso queda a cargo del grupo y debe estar debidamente justificada.
5. Definir el esquema de direcciones que se va a utilizar para comunicar a las partes entre sí. Teniendo en cuenta que todos los componentes se ejecutan en la misma computadora, ¿cuál es el mejor modo de identificar unívocamente a las partes que intervienen?
6. Realizar la codificación de la aplicación. El código fuente debe estar documentado.

## Informe

El informe a entregar junto con el trabajo práctico debe contener los siguientes ítems:

1. Breve análisis de problema, incluyendo una especificación de los casos de uso de la aplicación.
2. Detalle de resolución de la lista de tareas anterior. Prestar atención especial al ítem 4 de la lista de tareas, ya que se deberá justificar cada uno de los mecanismos de concurrencia elegidos.
3. Diagramas de clases de cada módulo.
4. Diagrama de transición de estados de un módulo de chat genérico.

## 2. Análisis del problema

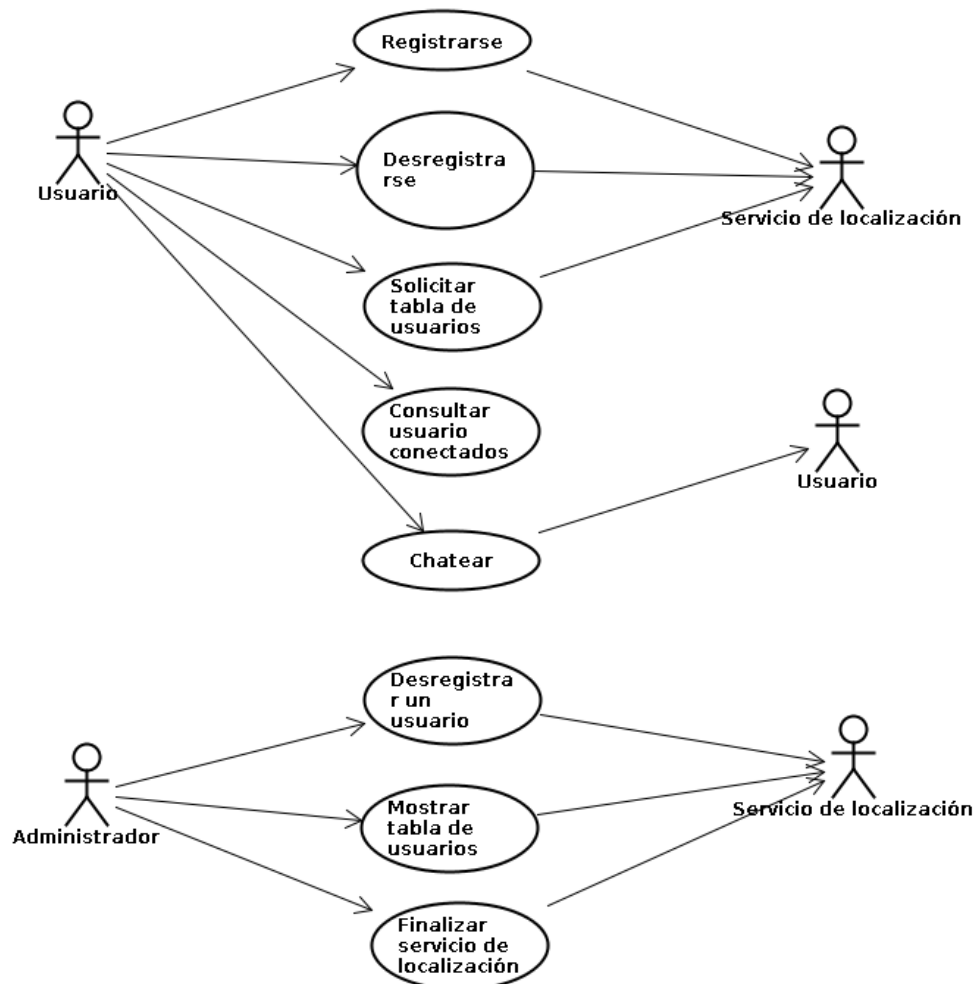
El trabajo práctico plantea el desarrollo de una sistema de chat llamado “ConcuChat” el cual debe permitir la comunicación bidireccional de mensajes entre dos partes.

El sistema esta formado por tres módulos:

1. Servicio de localización
2. Administrador del servicio de localización
3. Programa de chat

A continuación se detallan las funcionalidades de los módulos mencionados utilizando un diagrama de casos de uso:

### 2.1. Diagrama de casos de uso



## 3. Diseño de la arquitectura

### 3.1. Cola de mensajes

El mecanismo de comunicación que se utilizó para el pasaje de información entre los distintos procesos dentro del trabajo práctico fue, únicamente, la cola de mensajes. La principal razón por la cual se optó por este mecanismo se debió a que el sincronismo entre los mensajes escritos y leídos está a cargo del sistema operativo.

En un principio se consideró el uso de memoria compartida, sincronizando el acceso a la misma con semáforos. Esta solución era un poco limitada, ya que acotaba el tamaño de los datos compartidos a un máximo pre-establecido. En el caso de la tabla de contactos se limitaría la cantidad de usuarios registrados y también cuando dos usuarios estuvieran chateando no podrían enviarse mensajes más extensos que el tamaño de la memoria compartida.

Por otro lado, se analizó la posibilidad de utilizar tuberías con nombre ya que permitían enviar mensajes de cualquier longitud. El problema en el uso de fifos estaba relacionado con el sincronismo, ya que es inaceptable que más de un proceso escriba en forma simultánea en la tubería. Implementar un protocolo sincronizado para las fifos hubiese sido en definitiva hacer *a mano* la cola de mensajes.

Luego de analizar las diferentes alternativas, se optó por utilizar cola de mensajes ya que era el mecanismo de comunicación que mejor se adaptaba al problema.

#### Mensajes de tamaño variable

Si bien las colas de mensajes permiten el envío de mensajes de tamaño variable, la implementación de esto no es presisamente directa.

La función del sistema `msgsnd` recibe un puntero a un buffer y un tamaño, en bytes, de los datos a enviar. Esto permite, fácilmente, tener control sobre la cantidad de bytes enviados. Sin embargo, para recibir un mensaje es necesario conocer previamente la cantidad de bytes a leer. Por ende, no es posible saber de qué tamaño crear el buffer de lectura que se le pasa a `msgrcv`.

Esto se podría solucionar enviando dos mensajes: primero el tamaño en bytes y luego los datos del mensaje en sí. De esa forma, al leer un mensaje primero se recibiría el tamaño (que es un entero que ocupa una cantidad de bytes fijos, por ejemplo 4), y luego, con ese tamaño, leer los datos. Pero esto crea un problema de sincronismo: no se puede asegurar que entre que se envía el primero y el segundo mensajes otro proceso no mande un mensaje a la cola, rompiendo el protocolo.

Por fortuna, `msgrcv` informa cuando un mensaje a leer sobrepasa el tamaño del buffer de lectura: retornando -1 y seteando la variable global `errno` a un simpático valor `E2BIG`. De esta forma, se puede hacer un sencillo algoritmo para recibir mensajes de tamaño variable:

```
void* mensaje;
```

```

int tamano_mensaje;

int tamano_buffer = TAMANIO_BUFFER_INICIAL;
bool mensaje_recibido = false;
while (!mensaje_recibido)
{
    void* buffer = new char[tamano_buffer];
    int bytes_leidos = msgrcv(idCola, buffer, tamano_buffer, mtype);
    if (bytes_leidos == -1 && errno == E2BIG)
    {
        // No alcanzó el tamaño del buffer, re redimensiona.
        delete [] buffer;
        tamano_buffer *= 2;
    }
    else if (bytes_leidos != -1)
    {
        // El mensaje entró en el buffer.
        mensaje = buffer;
        tamano_mensaje = bytes_leidos;
        mensaje_leido = true;
    }
    else {
        // Ocurrió algún error...
    }
}

```

En esencia, lo que se hace es intentar leer el mensaje con un buffer de un tamaño inicial fijo y, si falla porque el buffer era muy chico, volver a intentar con un buffer del doble de tamaño, y así iterativamente hasta poder leer todo el mensaje. Al final del algoritmo, el puntero `mensaje` queda apuntando al mensaje leído, y `tamano_mensaje` es el tamaño del mensaje leído. El código utilizado en la aplicación, si bien análogo, no es idéntico a este, ya que se usaron clases y funciones auxiliares, así como tratamiento de los errores con excepciones.

## 3.2. Arquitectura General

### 3.2.1. Esquema de direcciones

Cada módulo en ejecución, ya sea el programa de chat (cliente) o el servicio de localización (servidor) tiene asociada una dirección, que se utiliza para que otros módulos en ejecución puedan contactarlo. La dirección de un módulo en ejecución es simplemente su identificador de proceso (pid).

### 3.2.2. Comunicación entre módulos

Para recibir mensajes, cada módulo en ejecución controla una cola de mensajes, esto es: sólo él lee mensajes de la cola (pero más de uno puede escribir mensajes en ella). En

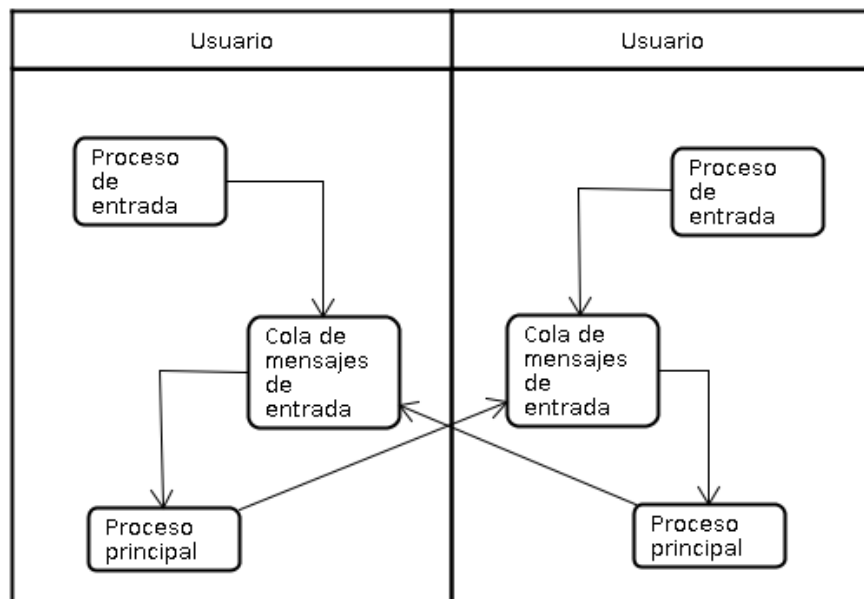


el caso del servicio de localización, al poder haber a lo sumo uno en ejecución en cualquier momento, la cola de mensajes se encuentra en un archivo de dirección fija (configurable a través de un archivo de configuración). En el caso de los clientes que, obviamente, pueden haber muchos en ejecución simultáneamente, su cola está asociada a un archivo con nombre tipo *cliente<direccion>.queue*, de manera que sea única y sólo un cliente lea de ella.

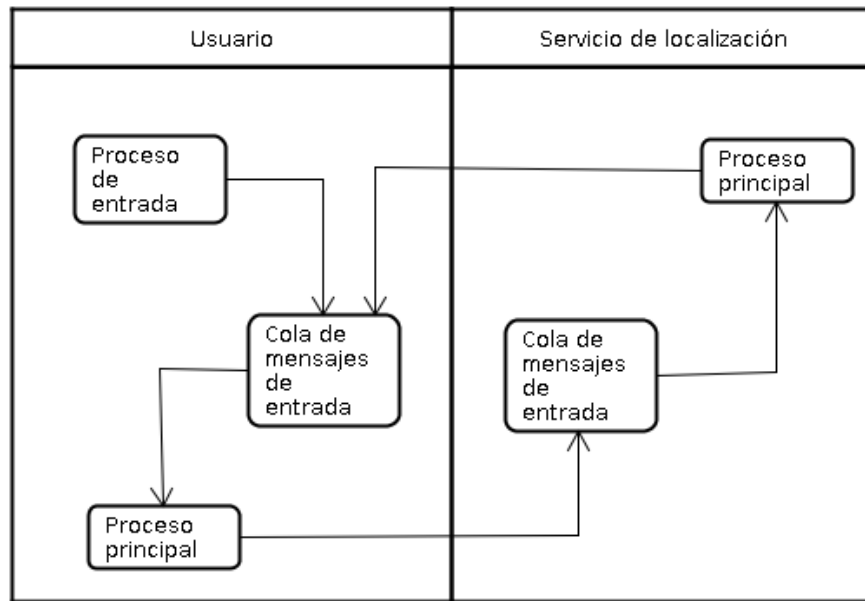
### 3.2.3. División en procesos

Para el servicio de localización se optó por la solución más sencilla: un único proceso. Dicho proceso es un ciclo infinito que en cada iteración se bloquea hasta recibir un mensaje de su cola de entrada y luego lo procesa.

El módulo cliente está dividido en dos procesos: uno muy sencillo encargado de la lectura de teclado y otro que maneja la lógica del cliente (el proceso principal). Al leer algún input del usuario por teclado, el proceso de entrada, para informarme al proceso principal de ese evento, le envía un mensaje con contenido del input a través de la cola de mensajes asociada a ese cliente. El proceso principal, análogamente al proceso del servidor, lee mensajes de la cola de entrada y los procesa. Estos mensajes pueden llegar tanto del servidor (como una respuesta de registro de nombre), de un peer (como un mensaje de chat), o del mismo módulo cliente (los mensajes de input desde el proceso de entrada).

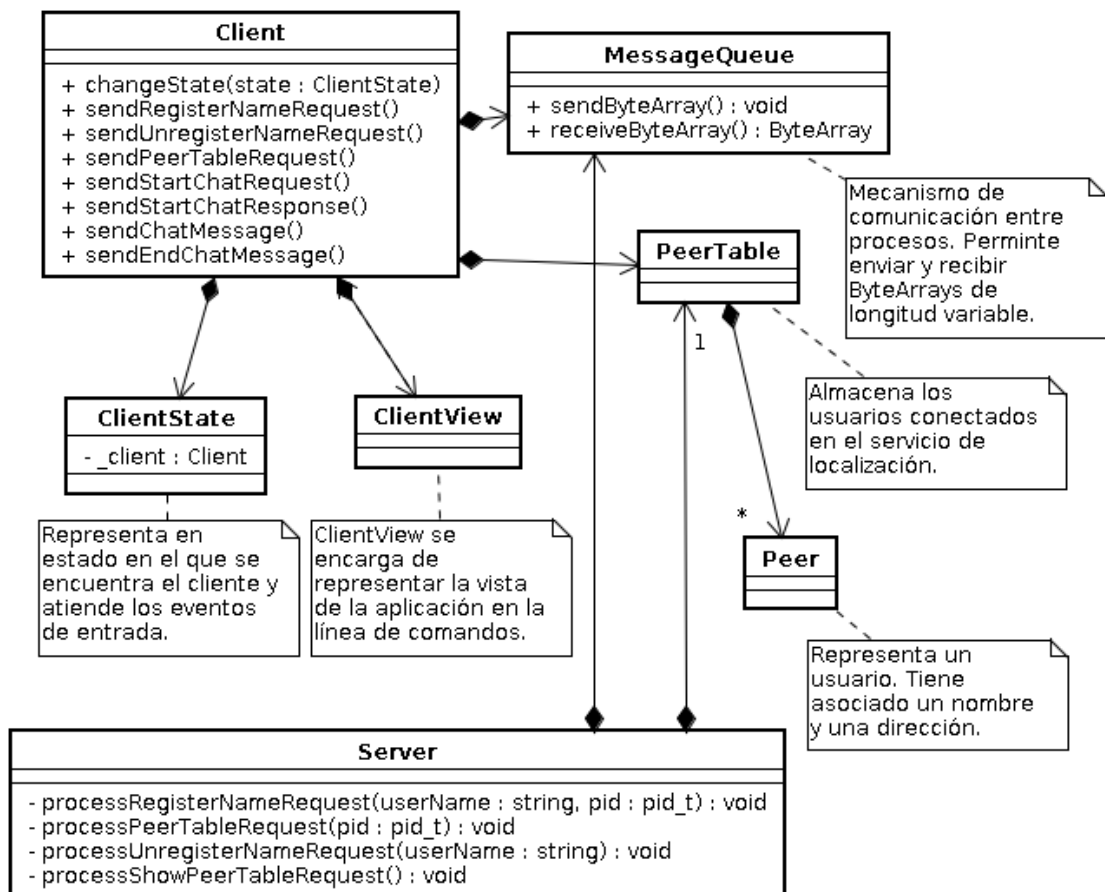


Equema de comunicación Cliente-Cliente (peer to peer). Las flechas representan entrada y salida de datos.



Esquema de comunicación Cliente-Servicio de localización.

### 3.3. Diagrama de clases



### 3.4. Mensajes

Los procesos se comunican entre sí enviándose mensajes. Se creó una entidad llamada “Message” la cual encapsula un tipo de mensaje, el id del proceso creador y los datos asociados al mensaje. Dicho mensaje puede ser serializado y deserializado para permitir su envío y recepción a través de la cola de mensajes.

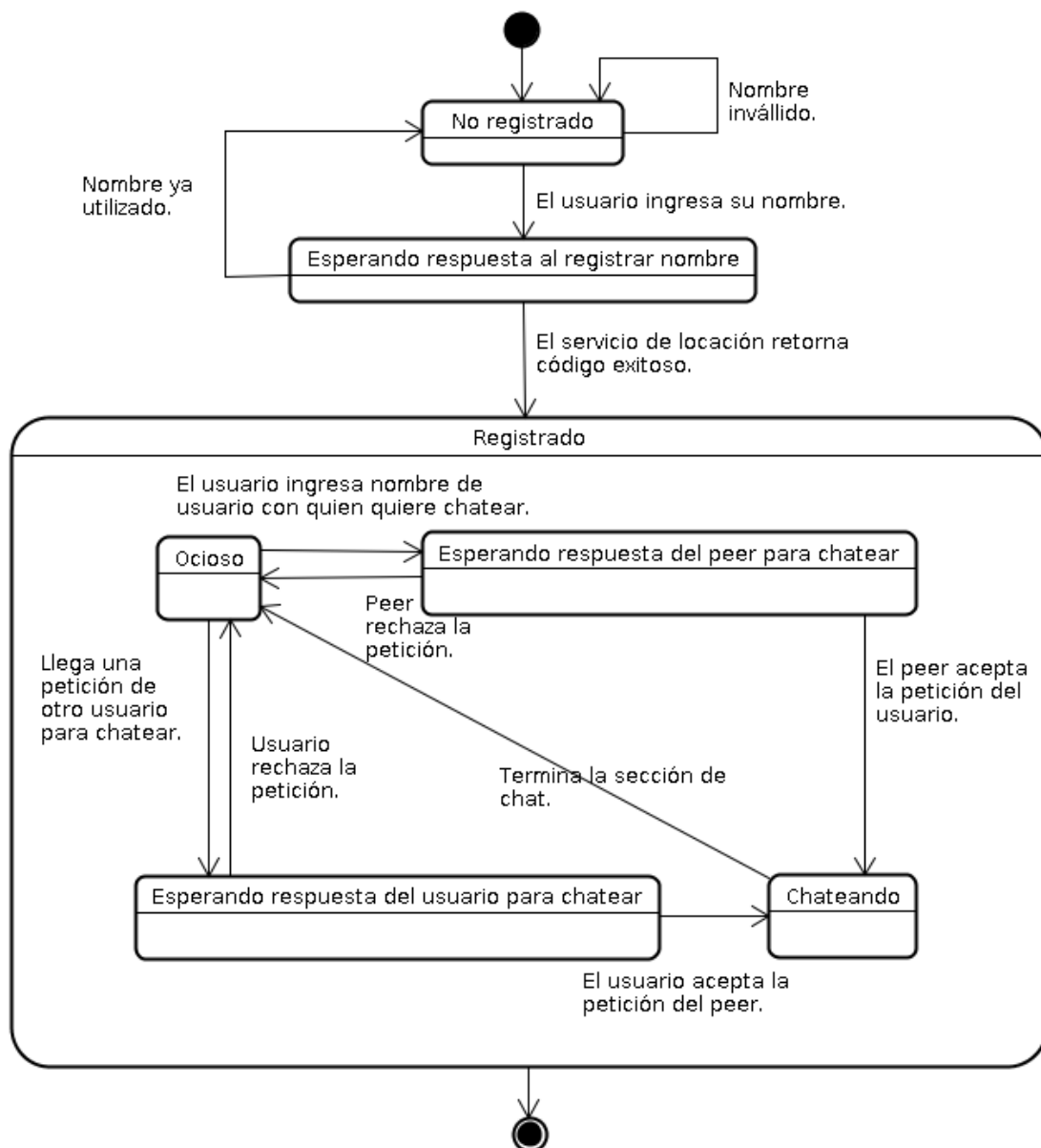
#### 3.4.1. Tipos de mensajes

- **TYPE NONE:** tipo de mensaje inválido.
- **TYPE USER INPUT:** enviado desde el proceso que recibe los eventos de teclado hacia el proceso principal cuando el usuario ingresa un comando.
- **TYPE USER EXIT:** enviado desde el proceso que recibe los eventos de teclado hacia el proceso principal cuando el usuario ingresa el comando de salida.
- **TYPE REGISTER NAME REQUEST:** enviado al servicio de localización para registrar un nuevo usuario.
- **TYPE REGISTER NAME RESPONSE:** enviado desde el servicio de localización al usuario para informar si el nombre pudo ser registrado con éxito o no.
- **TYPE UNREGISTER NAME REQUEST:** enviado al servicio de localización para desregistrar un usuario.
- **TYPE SHOW PEER TABLE REQUEST:** enviado al servicio de localización para hacer que muestre la tabla de contactos.
- **TYPE PEER TABLE REQUEST:** enviado al servicio de localización por parte de un usuario para solicitar la tabla de contactos actual.
- **TYPE PEER TABLE RESPONSE:** enviado desde el servicio de localización al usuario como respuesta de la solicitud de la tabla de contactos.
- **TYPE START CHAT REQUEST:** enviado entre usuarios para solicitar el comienzo de una sesión de chat.
- **TYPE START CHAT RESPONSE:** enviado entre usuarios para confirmar o rechazar el inicio de la sesión de chat.
- **TYPE END CHAT:** enviado entre usuarios para informar el fin de la sesión de chat.
- **TYPE CHAT MESSAGE:** enviado entre usuarios. Mensaje de chat.
- **TYPE SERVER EXIT:** enviado al servicio de localización para cerrarlo remotamente.

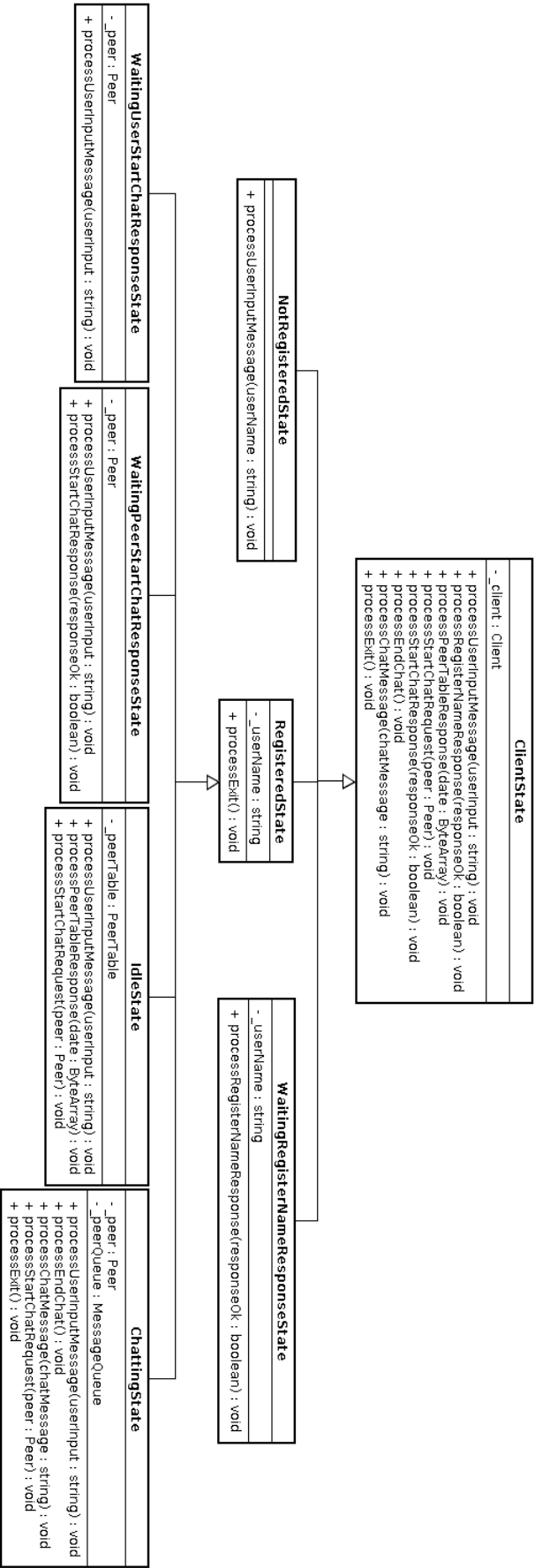
### 3.5. Estados del cliente

Cada vez que el usuario ingresa un comando, se procesa. El significado de los mensajes o comandos tipeados por el usuario dependen del estado en el que se encuentre la aplicación al momento de enviarlos. Por ejemplo, si dos usuarios se encuentran chateando, y uno usuario ingresa un comando “usuarios” (visualizar la tabla de contactos) se esperaría que el usuario con el que se encuentra chateando reciba un mensaje en texto plano con el contenido “usuarios” y no que visualice la tabla de contactos en su terminal. Por esta razón, se optó por el uso del patrón de diseño “State” (o estado) el cual nos permite que cada tipo de estado sea el encargado de procesar los mensajes que llegan al proceso principal de cada usuario.

#### 3.5.1. Diagrama de estados

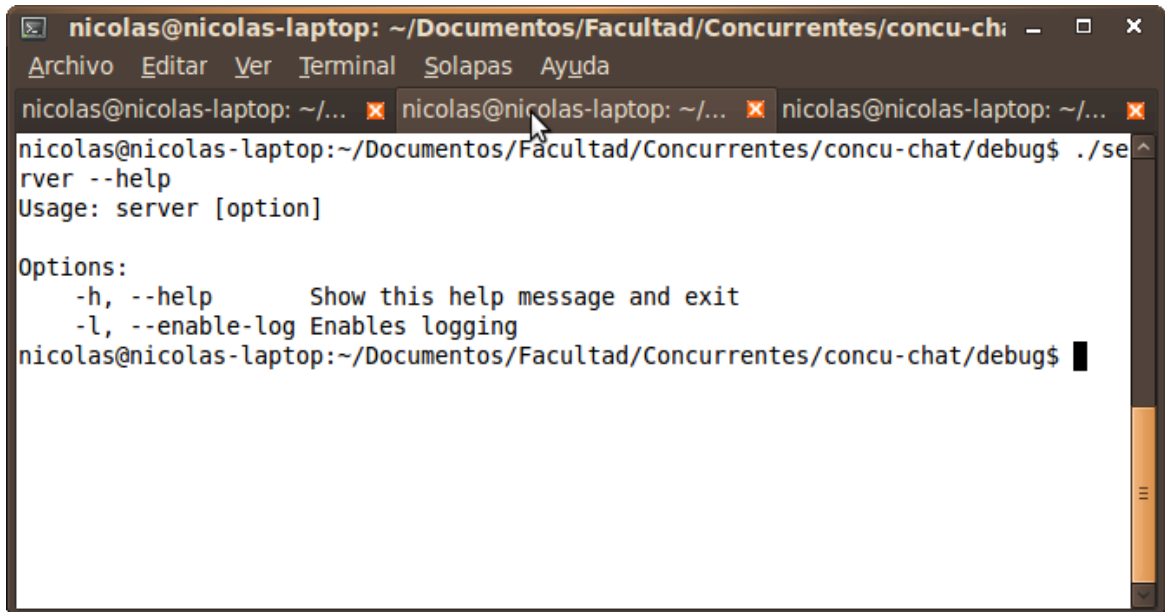


3.5.2. Diagrama de clases de los estados



## 4. Corrida de prueba

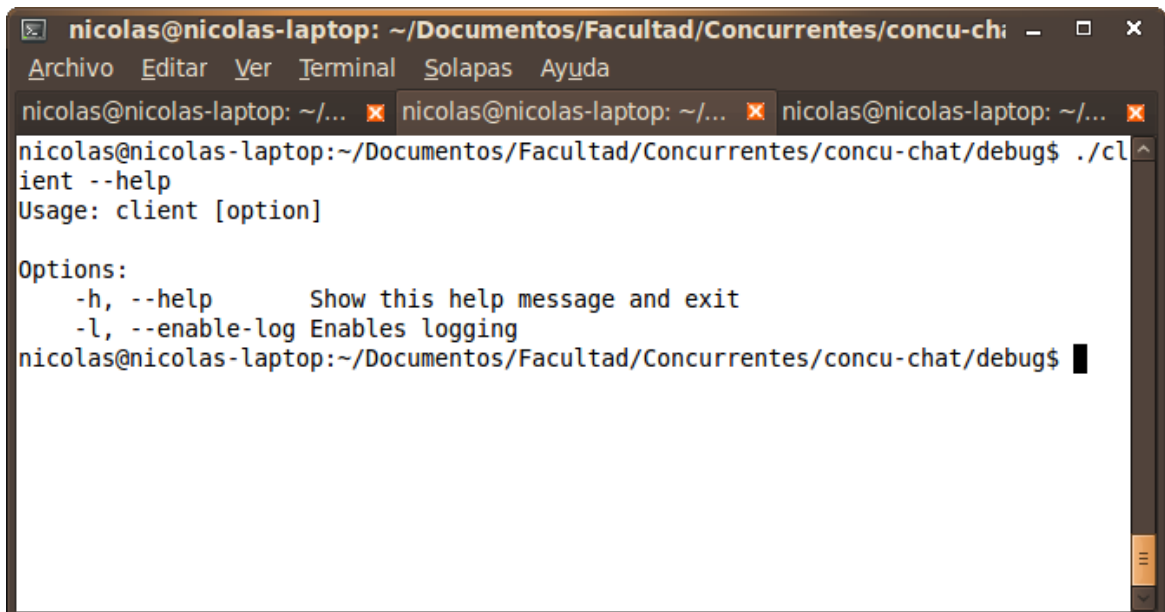
- Opciones al correr el servicio de localización.



```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./server --help
Usage: server [option]

Options:
  -h, --help          Show this help message and exit
  -l, --enable-log    Enables logging
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$
```

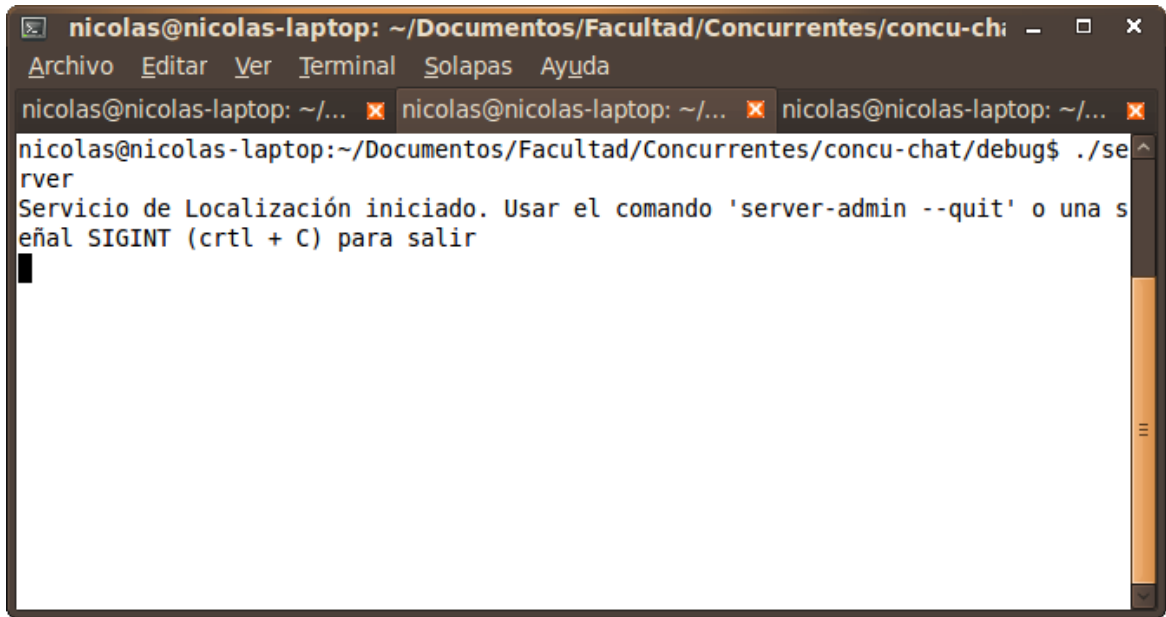
- Opciones al correr un cliente.



```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./client --help
Usage: client [option]

Options:
  -h, --help          Show this help message and exit
  -l, --enable-log    Enables logging
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$
```

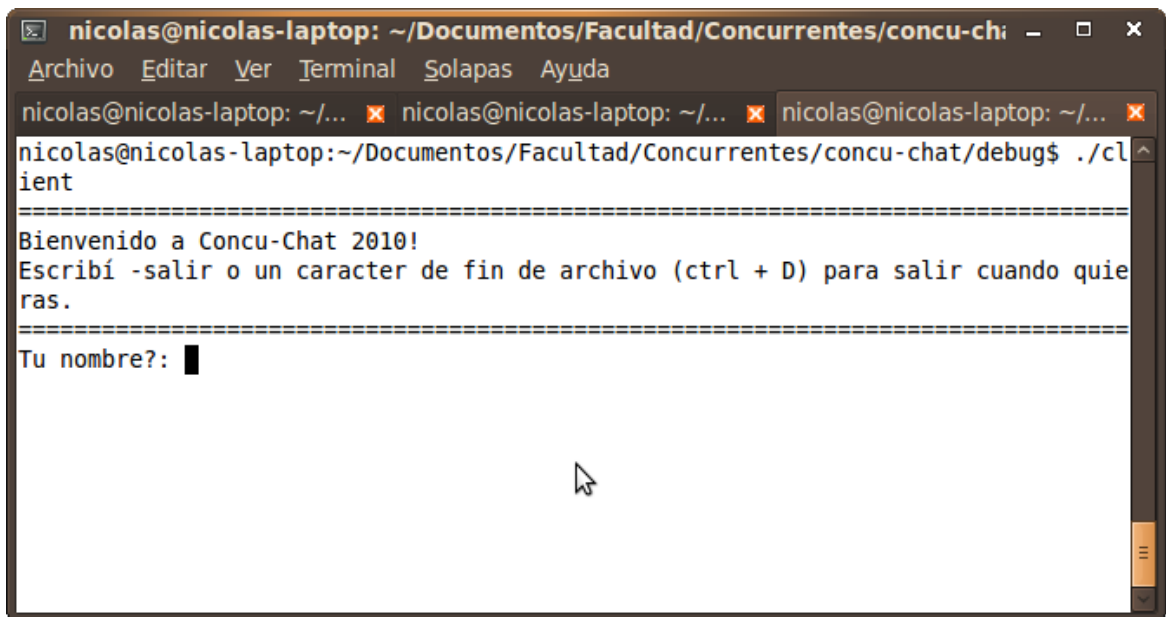
- Correr servicio de localización.



A terminal window titled "nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi" with a menu bar (Archivo, Editar, Ver, Terminal, Solapas, Ayuda). The prompt is "nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug\$". The command ".server" has been executed, resulting in the output: "Servicio de Localización iniciado. Usar el comando 'server-admin --quit' o una señal SIGINT (ctrl + C) para salir". A cursor is visible on the line following the output.

```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./server
Servicio de Localización iniciado. Usar el comando 'server-admin --quit' o una señal SIGINT (ctrl + C) para salir
```

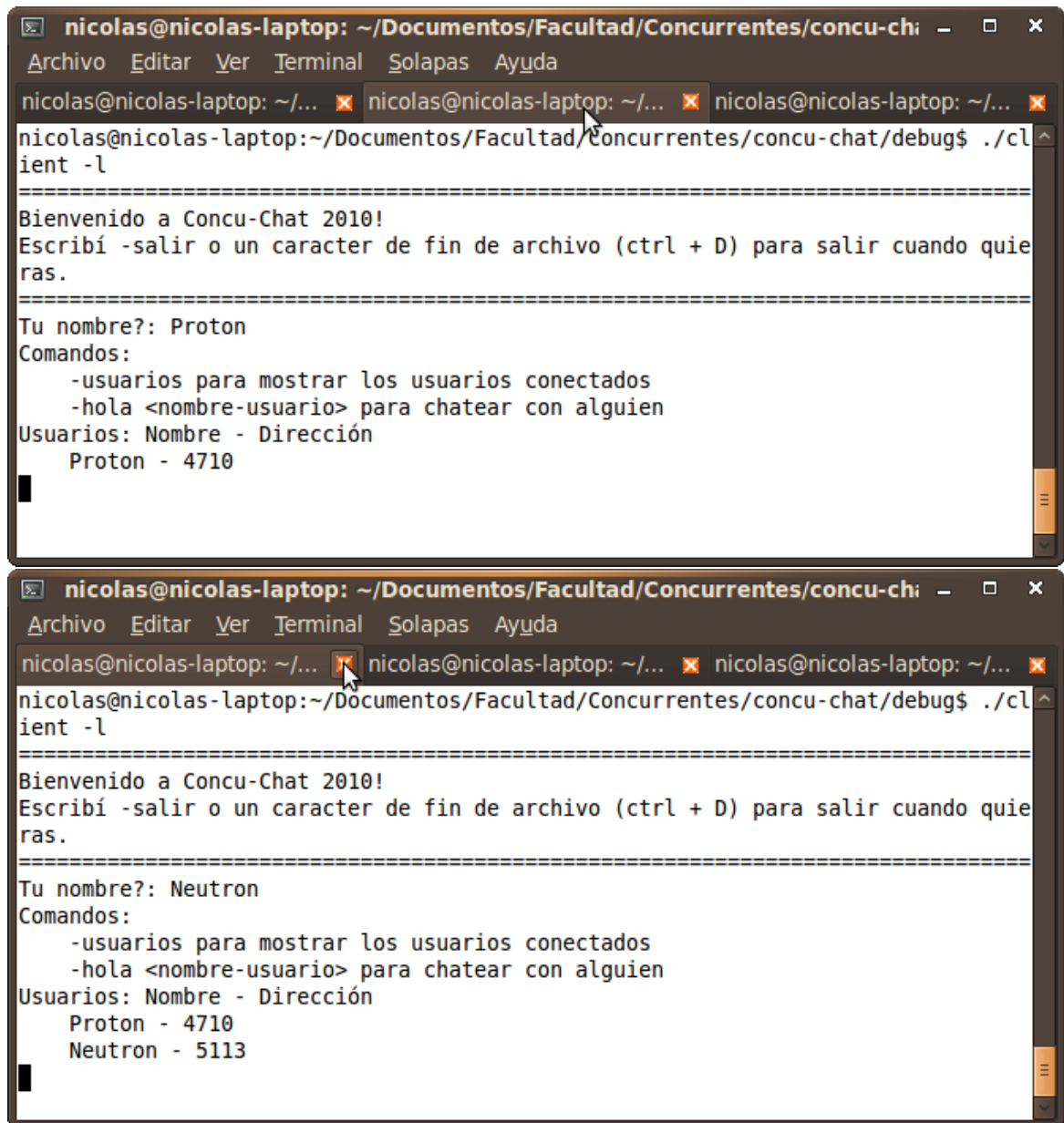
- Correr usuarios.



A terminal window titled "nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi" with a menu bar (Archivo, Editar, Ver, Terminal, Solapas, Ayuda). The prompt is "nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug\$". The command ".client" has been executed, resulting in the output: "Bienvenido a Concu-Chat 2010! Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras." followed by a separator line of equals signs and the prompt "Tu nombre?: ". A cursor is visible after the prompt.

```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./client
=====
Bienvenido a Concu-Chat 2010!
Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras.
=====
Tu nombre?:
```

- Usuarios ingresan nombres.

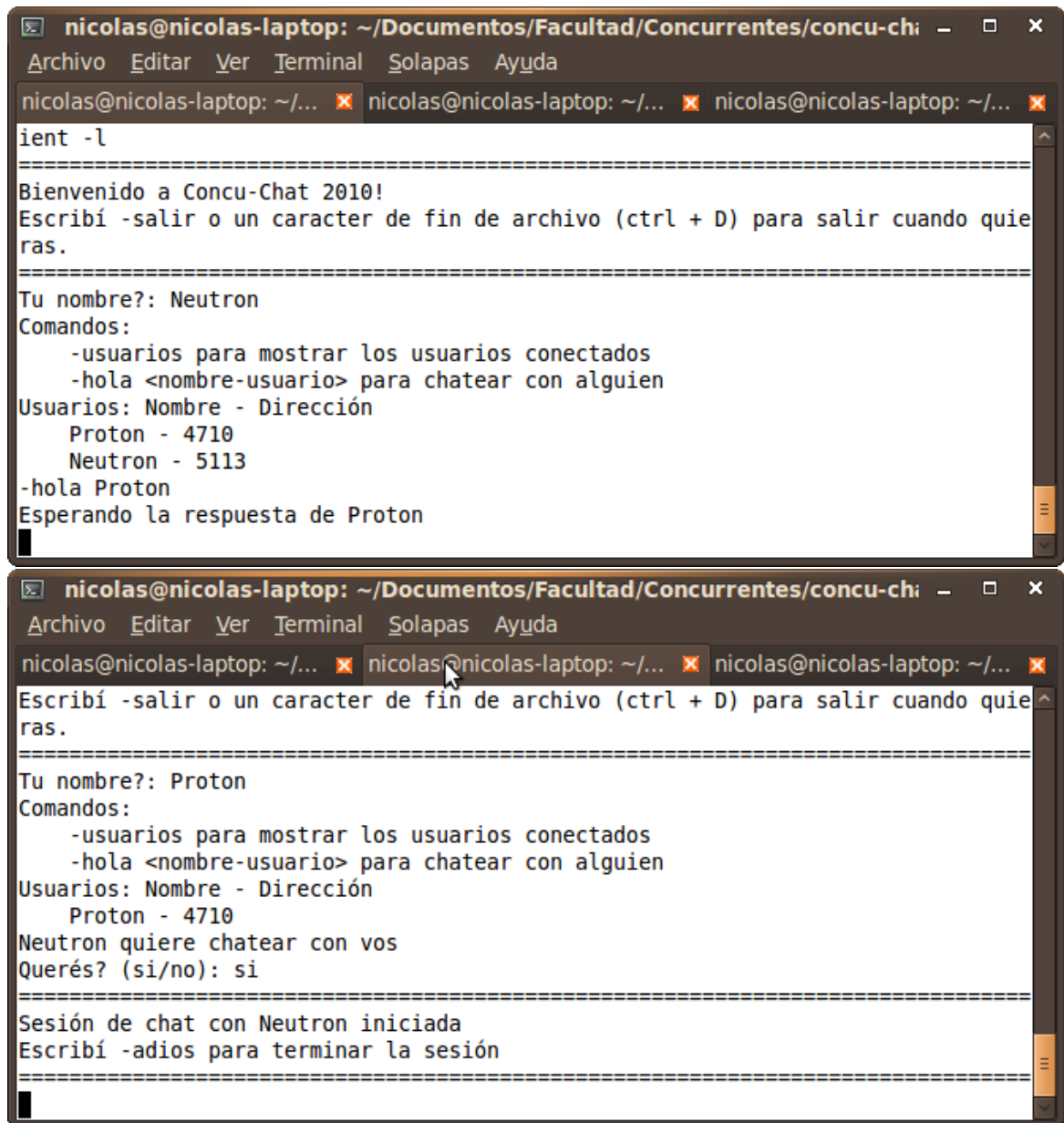


```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./client -l
=====
Bienvenido a Concu-Chat 2010!
Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras.
=====
Tu nombre?: Proton
Comandos:
  -usuarios para mostrar los usuarios conectados
  -hola <nombre-usuario> para chatear con alguien
Usuarios: Nombre - Dirección
          Proton - 4710
█

nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chat/debug$ ./client -l
=====
Bienvenido a Concu-Chat 2010!
Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras.
=====
Tu nombre?: Neutron
Comandos:
  -usuarios para mostrar los usuarios conectados
  -hola <nombre-usuario> para chatear con alguien
Usuarios: Nombre - Dirección
          Proton - 4710
          Neutron - 5113
█
```

- Un usuario envía una petición de chat, y el otro la acepta.

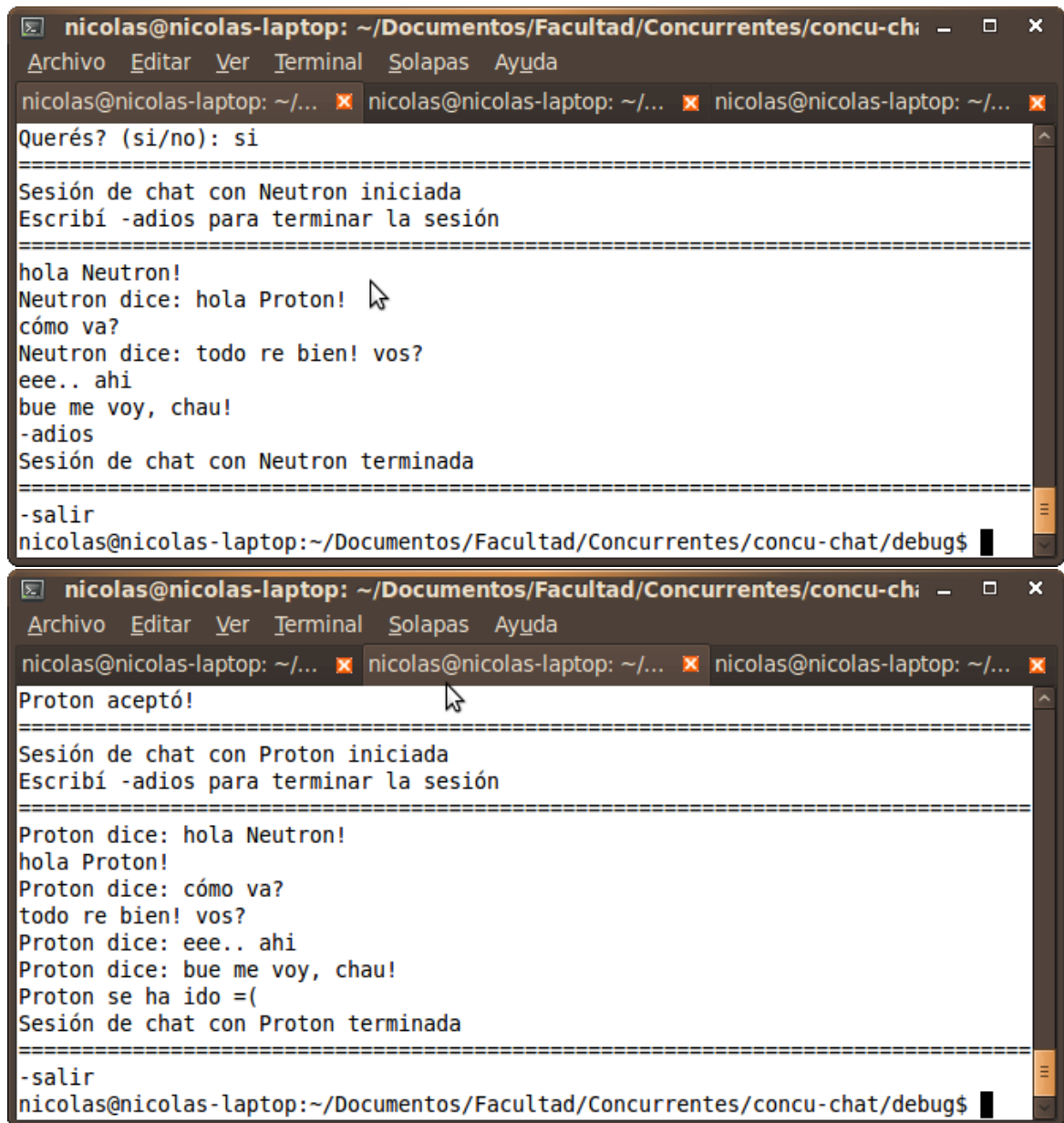




```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
ient -l
=====
Bienvenido a Concu-Chat 2010!
Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras.
=====
Tu nombre?: Neutron
Comandos:
  -usuarios para mostrar los usuarios conectados
  -hola <nombre-usuario> para chatear con alguien
Usuarios: Nombre - Dirección
  Proton - 4710
  Neutron - 5113
-hola Proton
Esperando la respuesta de Proton
█

nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
Escribí -salir o un caracter de fin de archivo (ctrl + D) para salir cuando quieras.
=====
Tu nombre?: Proton
Comandos:
  -usuarios para mostrar los usuarios conectados
  -hola <nombre-usuario> para chatear con alguien
Usuarios: Nombre - Dirección
  Proton - 4710
Neutron quiere chatear con vos
Querés? (si/no): si
=====
Sesión de chat con Neutron iniciada
Escribí -adios para terminar la sesión
=====
█
```

- Usuarios chateando.



The image displays two terminal windows from a Linux environment, showing the execution of a chat program. The top window shows a session with 'Neutron', and the bottom window shows a session with 'Proton'. Both sessions start with a confirmation to start the chat, followed by a series of messages and a final exit command.

```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
Querés? (si/no): si
=====
Sesión de chat con Neutron iniciada
Escribí -adios para terminar la sesión
=====
hola Neutron!
Neutron dice: hola Proton!
cómo va?
Neutron dice: todo re bien! vos?
eee.. ahi
bue me voy, chau!
-adios
Sesión de chat con Neutron terminada
=====
-salir
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$
```

```
nicolas@nicolas-laptop: ~/Documentos/Facultad/Concurrentes/concu-chi
Archivo Editar Ver Terminal Solapas Ayuda
nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x nicolas@nicolas-laptop: ~/... x
Proton aceptó!
=====
Sesión de chat con Proton iniciada
Escribí -adios para terminar la sesión
=====
Proton dice: hola Neutron!
hola Proton!
Proton dice: cómo va?
todo re bien! vos?
Proton dice: eee.. ahi
Proton dice: bue me voy, chau!
Proton se ha ido =(
Sesión de chat con Proton terminada
=====
-salir
nicolas@nicolas-laptop:~/Documentos/Facultad/Concurrentes/concu-chat/debug$
```

## 5. Conclusiones

A medida que se avanzó en la investigación sobre mecanismos de comunicación entre procesos en Unix se analizaron ventajas o mejores y desventajas que proveía cada uno. En un comienzo se utilizó memoria compartida y semaforos para sincronizar la misma. Como ya se dijo en la sección de diseño, esta solución traía problemas por el tamaño fijo de la memoria. Luego se pensó utilizando tuberías, pero en este caso el problema rondaba por la sincronización en el envío de los mensajes. Finalmente se terminó usando cola de mensajes ya que permitía enviar mensajes de tamaño variable sincronizados.

Por otro lado, se discutió cuál era la forma correcta de dividir en procesos a un programa concurrente. El principal problema estaba relacionado con que la memoria no era compartida como en el caso de los hilos livianos. En ese caso era muy simple pensar una solución ya que todos los recursos en común dentro de la aplicación se encontrarían en un mismo espacio de direcciones. Finalmente, como ya se analizó en el sección de diseño, se decidió separar al módulo cliente en dos procesos. Uno muy simple, el cual tenía como única función leer comandos de teclado y enviarlos a un procesos principal encargado de interpretar dichos comandos. En el caso del servidor, se optó por un sólo proceso encargado de recibir peticiones, atenderlas y responderlas a medida que iban arriando.

Otro tema de discusión que surgió fue, cómo realizar la arquitectura del sistema para que sea compatible con una interfaz gráfica. Como concecuencia del problema mencionado sobre la memoria no-compartida, el diseño actual no soportaría el uso de la misma. Dos procesos distintos no podrían estar usando la misma interfaz gráfica ya que los objetos utilizados por la vista no estarían en el espacio de direcciones de ambos procesos.