Universidad de Buenos Aires
Facultad De Ingeniería
Año 2010 - 1er 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 enguaje 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

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 la cola de mensages. 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 pensó en utilizar 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. Luego de analizar las diferentes alternativas, se optó por utilizar cola de mensajes ya que era el mecanismo de conumicación que mejor se adaptaba al problema.

A continuación, se prensenta la interfaz de la clase cola de mensajes:

```
/*
 * message_queue.h
 *
 *  Created on: 16/04/2010
 *      Author: nicolas
 */

#ifndef MESSAGE_QUEUE_H_
#define MESSAGE_QUEUE_H_

#include "utils.h"
#include "core/byte_array.h"
#include "resource.h"

#include <string>

#define QUEUE_INITAL_BUFFER_SIZE 32

using std::string;

class RawMessageQueue: public Resource {
public:

/**
 * Connect to a message queue, or create it if it doesn't exist
 *
 * @param pathName     A path to an existing file.
```

```
 * @param id           A char to identify the shared memory object.
 * @param ownResource  Specifies whether the resource owner.
 */
explicit RawMessageQueue(const string& pathName, char id, bool ownResource =
true);

/**
 * Sends size bytes from the buffer to the queue. The id of the message
 * is mtype. If another process wants to read this message, it has to use
 * this mtype value.
 *
 * @param buffer Message to be written on the queue.
 * @param size The size, in bytes, of the data to be read.
 * @param mtype The id of the message.
 */
void sendFixedSize(const void* buffer, size_t size, long mtype);

/**
 * Receives size bytes from the queue and stores them in the buffer.
 *
 * @param buffer Destination of the message.
 * @param size The size, in bytes, of the message to be read.
 * @param mtype Zero    : Retrieves the next message on the queue,
 *    regardless of its mtype.
 *  Positive: Gets the next message with an mtype equal to
 *    the specified mtype.
 *  Negative: Retrieve the first message on the queue whose
 *    mtype field is less than or equal to the
 *    absolute value of the mtype argument.
 */
void receiveFixedSize(void* buffer, size_t size, long mtype);

/**
 *  Immediately removes the message queue, awakening all waiting reader and
 *  writer processes (with an error return and errno set to EIDRM). The
 *  calling process must be the creator the message queue. Only one instance
 *  of RawMessageQueue (created by the same arguments) has the member
 *  _freeOnExit seted in true. If _freeOnExit's value is true, the message
 *  queue is removed.
 */
virtual ~RawMessageQueue() throw ();

protected:
int _queueId;

size_t tryReceive(void* buffer, size_t size, long mtype = 0);
```

```
virtual void doDispose() throw ();
virtual void print(ostream& stream) const;

// Non copiable.
DECLARE_NON_COPIABLE(RawMessageQueue)
};

/**
 * A class responsible for inter-process communication.
 */
class MessageQueue: public RawMessageQueue {
public:

static const long DEFAULT_SEND_MTYPE = 1;
static const long DEFAULT_RECEIVE_MTYPE = 0;

/**
 * Connect to a message queue, or create it if it doesn't exist
 *
 * @param pathName     A path to an existing file.
 * @param id           A char to identify the shared memory object.
 * @param ownResource  Specifies whether the resource owner.
 */
explicit MessageQueue(const string& pathName, char id, bool ownResource =
true) :
RawMessageQueue(pathName, id, ownResource) {
}

/**
 * Send an object through the queue. Note that the object must be of scalar
 * type (or it's internal structure be all scalar types), as it will be
 * written as bytes.
 *
 * @param obj The object to be send to the queue.
 * @param mtype The id of the message.
 */
template<typename T>
void send(const T& obj, long mtype = DEFAULT_SEND_MTYPE) {
RawMessageQueue::sendFixedSize(&obj, sizeof(T), mtype);
}

/**
 * Receives an object from the queue. Note that the object must be of
 * scalar type (or it's internal structure be all scalar types), as it
 * will be written as bytes.
```

```
 *
 * @param mtype The id of the message.
 * @return  The 'T' object read from the queue.
 */
template<typename T>
T receive(long mtype = DEFAULT_RECEIVE_MTYPE) {
T obj;
RawMessageQueue::receiveFixedSize(&obj, sizeof(T), mtype);
return obj;
}

/**
 * Send a byte array message through the queue.
 *
 * @param message The message to be send to the queue.
 * @param mtype The id of the message.
 */
void sendByteArray(const ByteArray& message, long mtype =
DEFAULT_SEND_MTYPE);

/**
 * Receives a byte array message from the queue.
 *
 * @param mtype  The id of the message.
 * @return   Byte array with the message.
 */
const ByteArray receiveByteArray(long mtype = DEFAULT_RECEIVE_MTYPE);

/*
 * Send a string message through the queue.
 *
 * @param message       The message to be send to the queue.
 * @param mtype         The id of the message.
 */
void
sendString(const std::string& message, long mtype = DEFAULT_SEND_MTYPE);

/**
 * Receives a string message from the queue.
 *
 * @param mtype The id of the message.
 * @return              The string message read from the queue.
 */
const std::string receiveString(long mtype = DEFAULT_RECEIVE_MTYPE);

/**
```

```
 * Removes the message queue. The calling process must be the creator the
 * message queue
 */
virtual ~MessageQueue() throw () {
}
};

#endif /* MESSAGE_QUEUE_H_ */
```

3.2. Mensajes

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

3.2.1. Tipos de mensajes

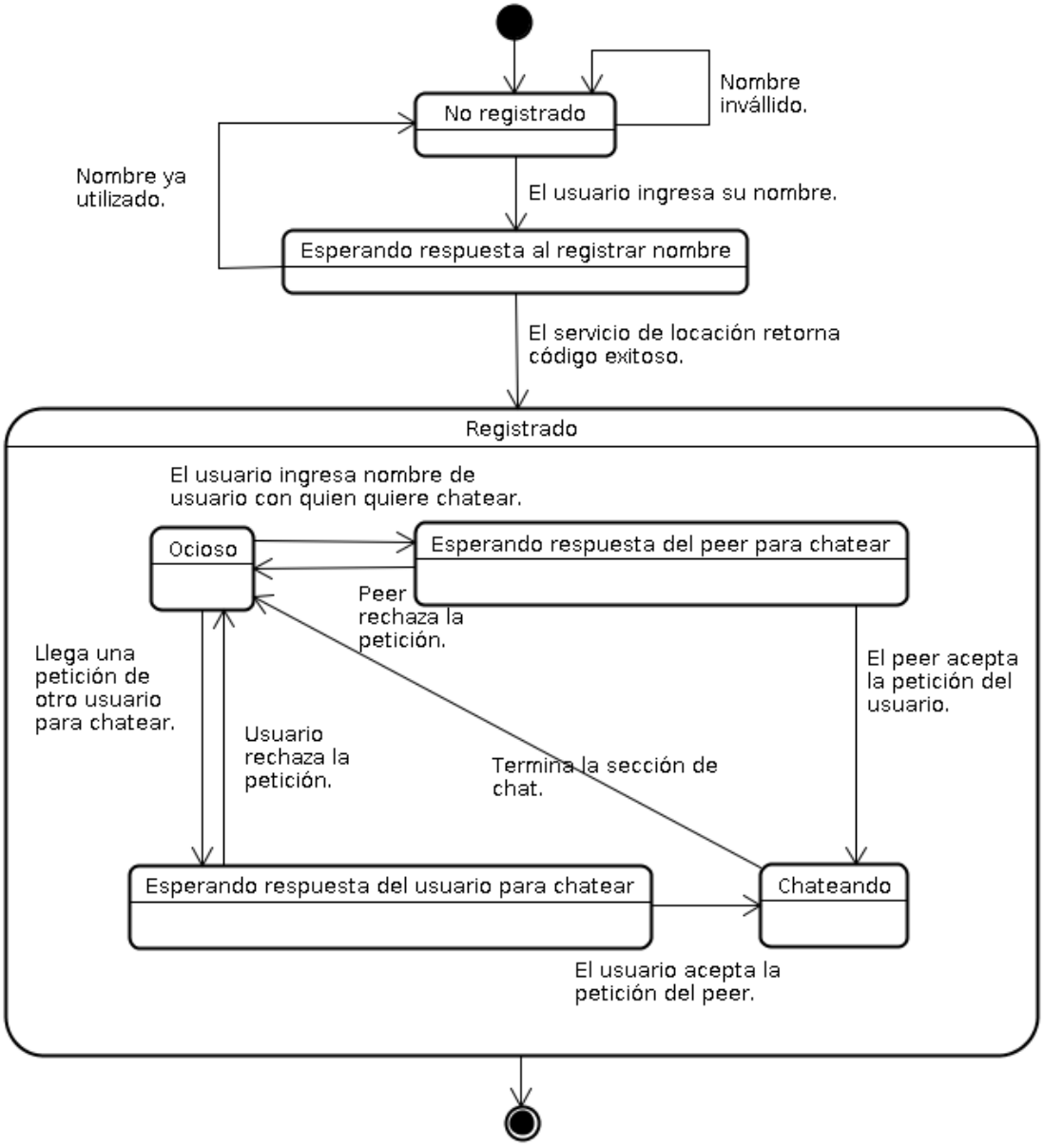
- **TYPE NONE:** tipo de mensaje inválido.
- **TYPE USER INPUT:** enviado desde el proceso que recibe los eventos de teclado hacia el proceso principal cuando el usuario ingresa un comando.
- **TYPE USER EXIT:** enviado desde el proceso que recibe los eventos de teclado hacia el proceso principal cuando 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 seción de chat.

- **TYPE START CHAT RESPONSE:** enviado entre usuarios para confirmar o rechazar el inicio de la seción de chat.
- **TYPE END CHAT:** enviado entre usuarios para informar el fin de la seción de chat.
- **TYPE CHAT MESSAGE:** enviado entre usuarios. Mensaje de chat.
- **TYPE SERVER EXIT:** enviado al servicio de localización para cerrarlo remotamente.

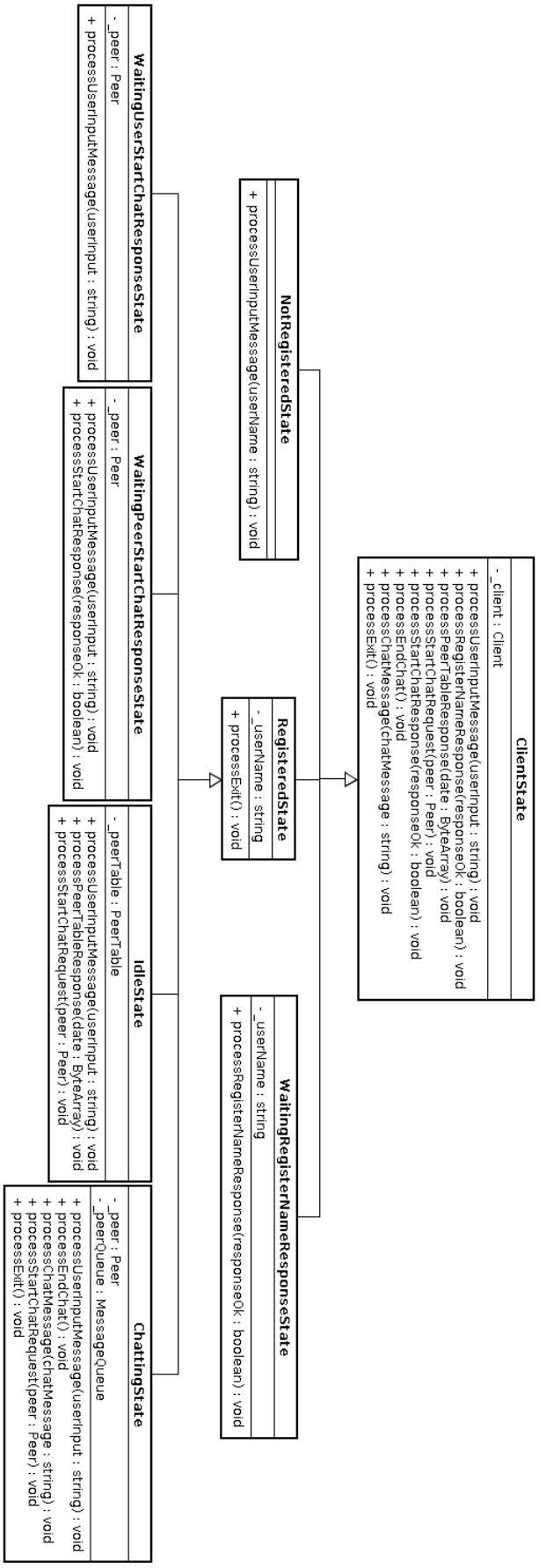
3.3. Estados del cliente

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.

3.3.1. Diagrama de estados



3.3.2. Diagrama de clases de los estados



4. Conclusiones