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

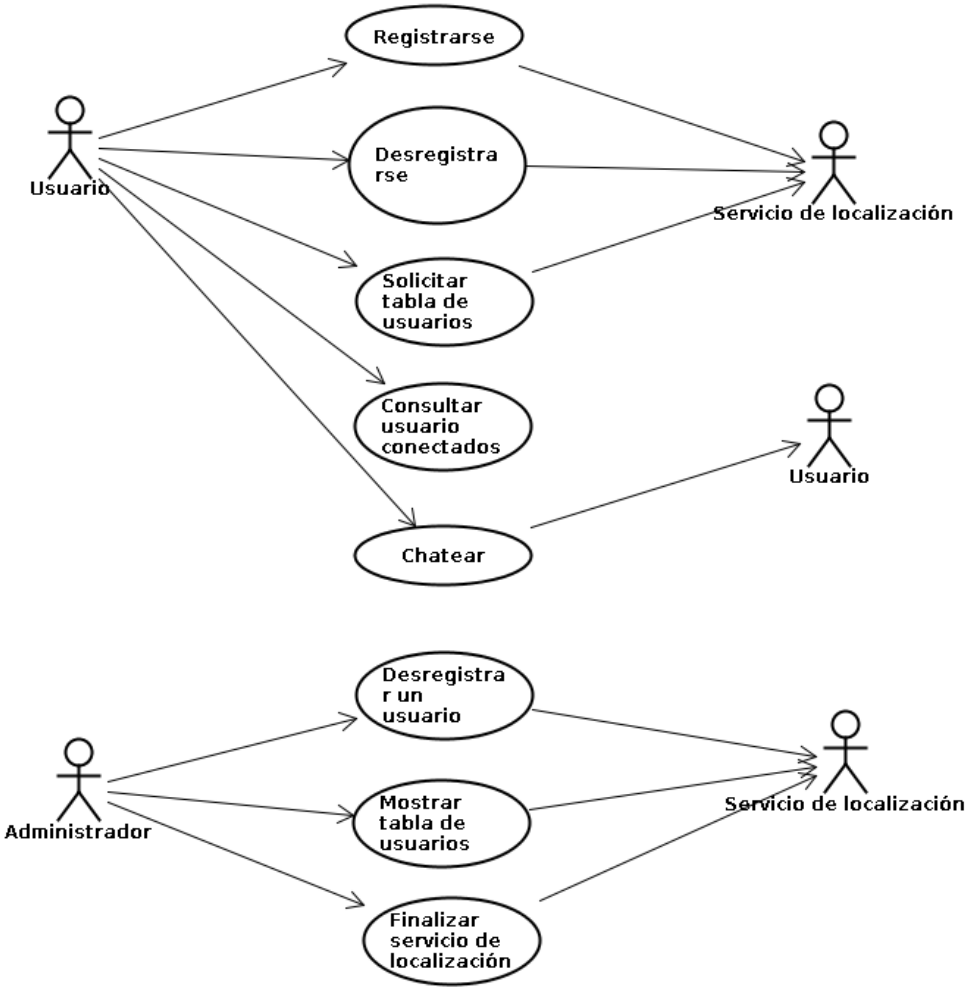
El trabajo práctico plantea el desarrollo de una sistema de chat llamado "ConcuChat" el cual debe permitir la comunicación bidireccional de mensajes entre dos partes.

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



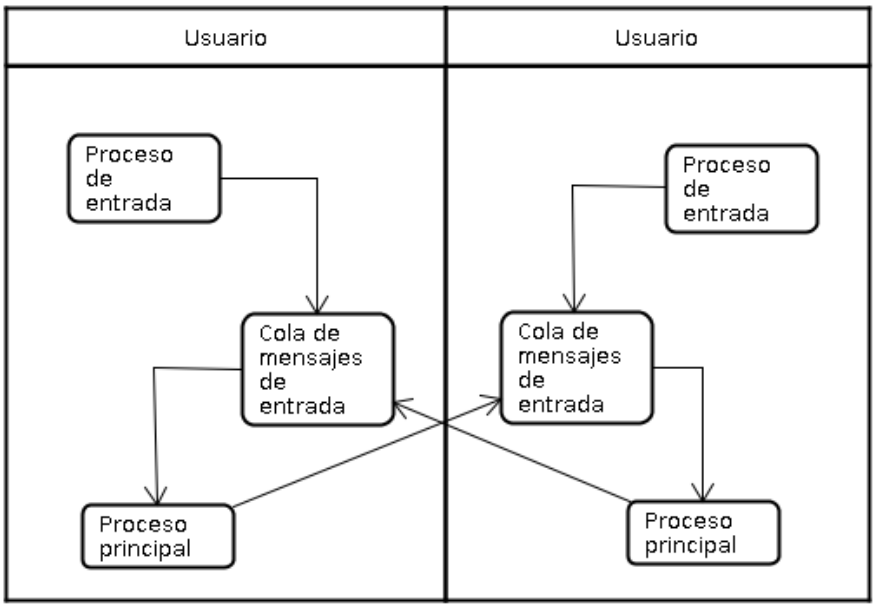
3. Diseño de la arquitectura

3.1. Cola de mensajes

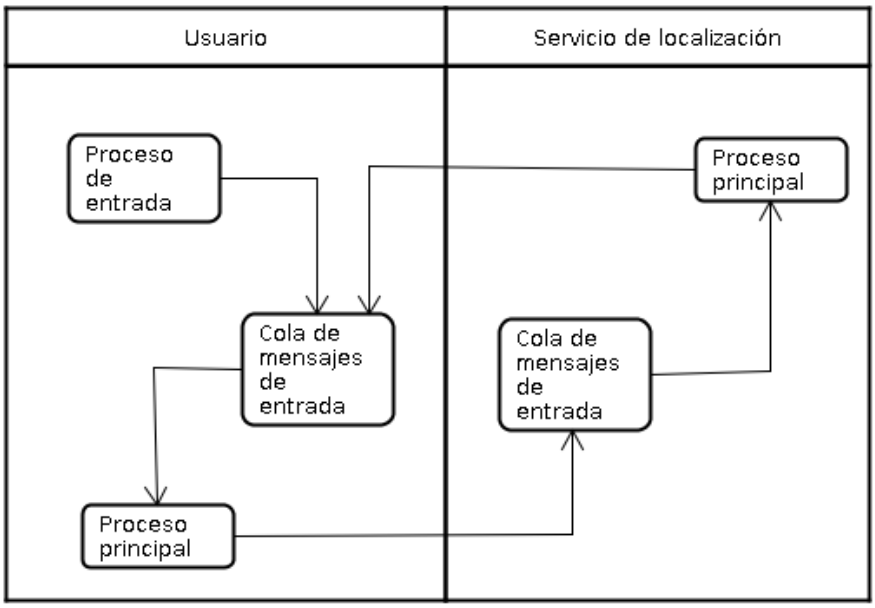
El mecanismo de comunicación que se utilizó para el pasaje de información entre los distintos procesos dentro del trabajo práctico, fue la cola de mensages. La principal razón por la cual se optó por este mecanismo se debió a que el sincronismo entre los mensajes escritos y leídos está a cargo del sistema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

3.2. Arquitectura

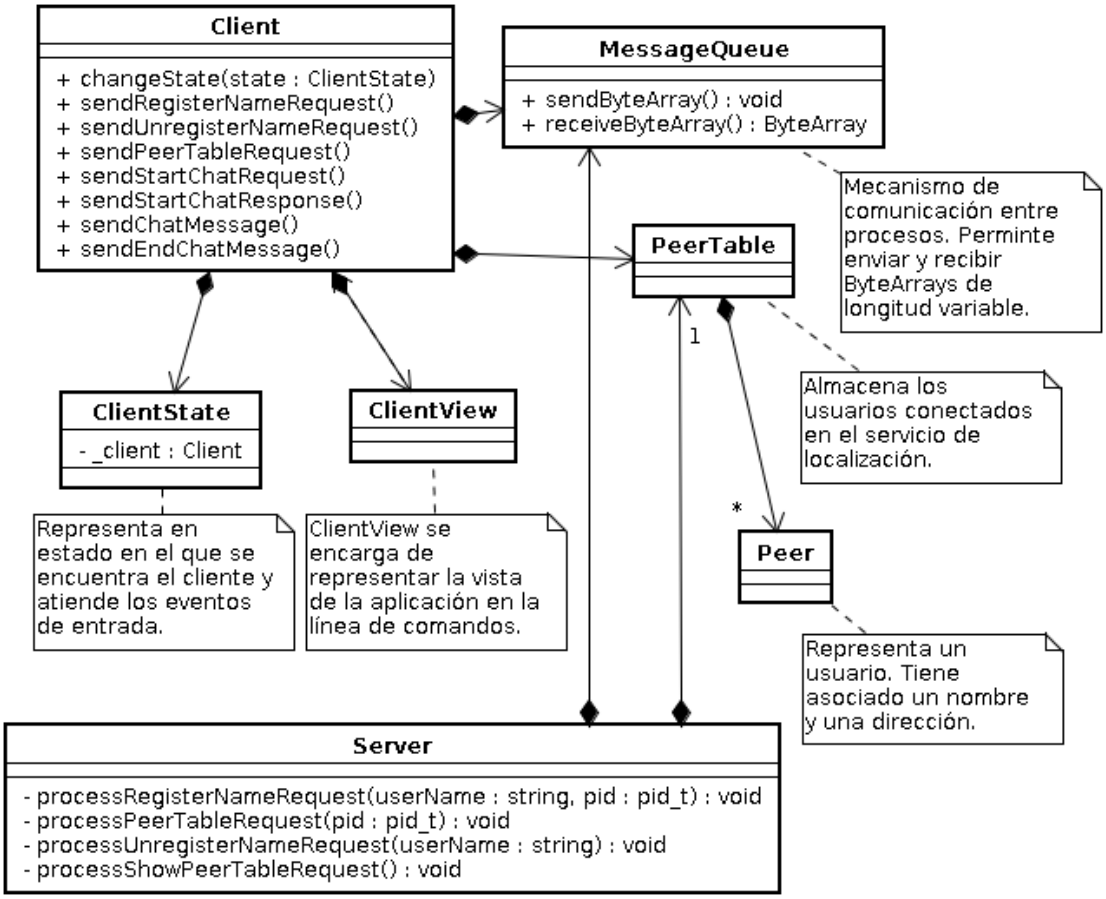
3.2.1. Cliente-Cliente



3.2.2. Cliente-Servicio de localización



3.3. Diagrama de clases



3.4. 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.

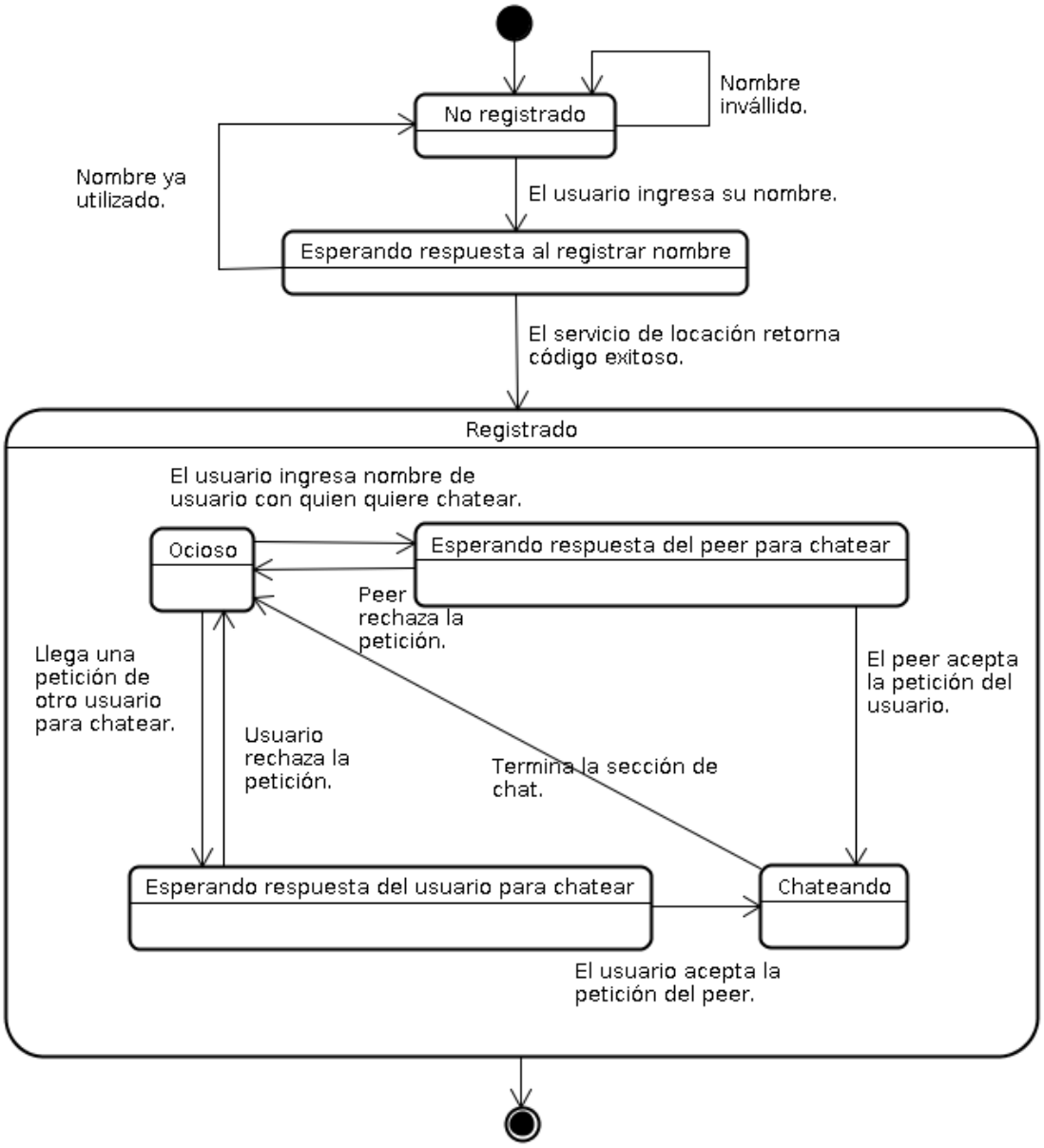
3.4.1. Tipos de mensajes

- **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 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.

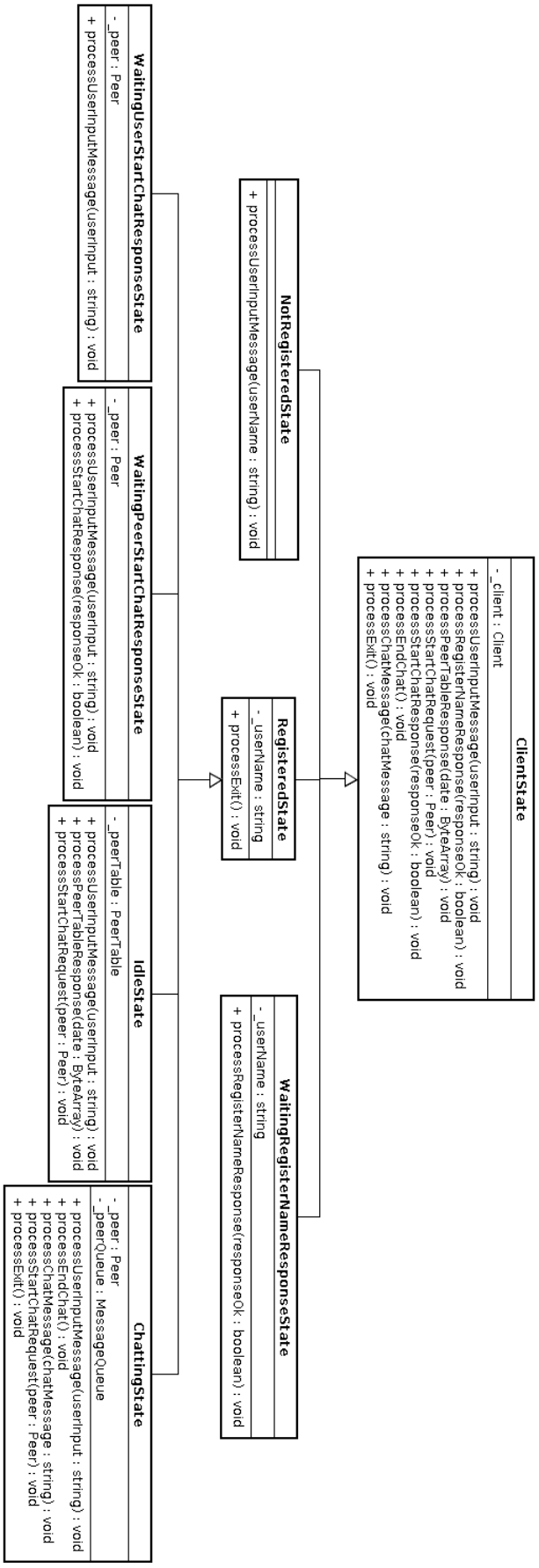
3.5. Estados del cliente

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. CódigoFuente

```
/*
 * server.h
 *
 *  Created on: 26/06/2010
 *      Author: nicolas
 */

#include "model/message.h"
#include "model/peer_table.h"

#include <string>

using std::string;

#ifndef SERVER_H_
#define SERVER_H_

class Server
{
public:
Server();

~Server();

int run();

private:
PeerTable _peerTable;

void processMessage(const Message& message, bool& exit);

static void createQueueFile();
static void destroyQueueFile();

void processRegisterNameRequest(const string&,pid_t);
void processPeerTableRequest(pid_t);
void processUnregisterNameRequest(const string&);
void processShowPeerTableRequest();
};


#endif /* SERVER_H_ */

/*
 * client.h
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#ifndef CLIENT_H_
#define CLIENT_H_

#include "ipc/message_queue.h"
#include "model/message.h"
#include "client_view.h"

#include <string>

class ClientState;
class Message;

using std::string;

class Client
{
public:
```

```
Client();

~Client();

int run();

/**
 * Change the current state of the client.
 */
void changeState(ClientState* newState);

/**
 * Sends a request to register a name to the server.
 * @param userName The name to be registered.
 * @return Whether the message could be sent or not.
 */
bool sendRegisterNameRequest(const string& userName);

/**
 * Sends a request to unregister a name to the server.
 * @param userName The name to be unregistered.
 * @return Whether the message could be sent or not.
 */
bool sendUnregisterNameRequest(const string& userName);

/**
 * Sends a request to the server to get the peer table.
 * @return Whether the message could be sent or not.
 */
bool sendPeerTableRequest();

/**
 * Sends a request to start a chat session to a peer.
 * @param peer The peer whom the user wants to chat.
 * @param userName The name of the user (this client's user).
 * @return Whether the message could be sent or not.
 */
bool sendStartChatRequest(const Peer& peer, const string& userName);

/**
 * Sends a response of a start chat request to a peer.
 * @param peer The peer that originally sent the start chat request.
 * @param responseOk True if the user accepted to chat with the peer. False
 * otherwise.
 * @return Whether the message could be sent or not.
 */
bool sendStartChatResponse(const Peer& peer, bool responseOk);

/**
 * Sends a chat message to a peer.
 * @return Whether the message could be sent or not.
 */
bool sendChatMessage(MessageQueue& peerQueue, const string& chatMessage);

/**
 * Sends an end chat message to a peer.
 * @return Whether the message could be sent or not.
 */
bool sendEndChatMessage(MessageQueue& peerQueue);

ClientView& getView();
private:

typedef Message::MessageType MessageType;

string _queueFileName;
ClientState* _state;
ClientView _view;

void runUserInputProcess();
```

```
void runMainProcess();

void createQueueFile();
void destroyQueueFile();
void processMessage(const Message& message, bool& exitNow);

bool sendMessageToQueue(MessageQueue& queue, MessageType type,
const ByteArray& data = ByteArray());
bool sendMessage(const string& queueFileName, MessageType type,
const ByteArray& data = ByteArray());
bool sendMessageToServer(MessageType type, const ByteArray& data =
ByteArray());
bool sendMessageToPeer(const Peer& peer, MessageType type,
const ByteArray& data = ByteArray());
};

#endif /* CLIENT_H_ */

/*
 * client_view.h
 *
 *  Created on: Jun 26, 2010
 *      Author: demian
 */

#ifndef CLIENT_VIEW_H_
#define CLIENT_VIEW_H_

#include "model/peer_table.h"
#include "utils.h"

#include <string>
using std::string;

class ClientView
{
private:
static const string LOWERCASE_YES;
static const string LOWERCASE_NO;

public:
static const string EXIT_COMMAND;
static const string PEER_TABLE_COMMAND;
static const string START_CHAT_COMMAND;
static const string END_CHAT_COMMAND;

ClientView();

void showWelcomeMessage();
void askUserName();
void showCouldNotContactServer();
void showCouldNotContactPeer(const string& peerName);
void showInvalidName(const string& userName);
void showAlreadyUsedName(const string& userName);
void showPeerTable(const PeerTable& peerTable);
void showIdleStateCommands();
void showCannotChatWithYourself();
void showInvalidPeerName(const string& peerName);
void showInvalidCommand(const string& command);
void showWaitingPeerResponse(const string& peerName);
void askUserStartChatWith(const string& peerName);
void showPeerCanceledChat(const string& peerName);
void showPeerAcceptedChat(const string& peerName);
void showStartChatSession(const string& peerName);
void showEndChatSession(const string& peerName);
void showPeerLeftChat(const string& peerName);
void showChatMessage(const string& peerName, const string& chatMessage);

bool isYesString(const string& str);
bool isNoString(const string& str);
```

```
DECLARE_NON_COPIABLE(ClientView)
};

#endif /* CLIENT_VIEW_H_ */

/*
 * client_state.h
 *
 *  Created on: Jun 26, 2010
 *      Author: demian
 */

#ifndef CLIENT_STATE_H_
#define CLIENT_STATE_H_

#include "model/peer_table.h"
#include "core/byte_array.h"
#include "ipc/message_queue.h"

#include <string>

using std::string;

// Forward declaration.
class Client;

/**
 * The abstract base class of the different client states. Consists of virtual
 * methods that control the client actions to different events that must
 * be overriden by the state subclasses.
 */
class ClientState
{
public:
virtual ~ClientState();

virtual void processUserInputMessage(const string& userInput);
virtual void processRegisterNameResponse(bool responseOk);
virtual void processPeerTableResponse(const ByteArray& data);
virtual void processStartChatRequest(const Peer& peer);
virtual void processStartChatResponse(bool responseOk);
virtual void processEndChat();
virtual void processChatMessage(const string& chatMessage);
virtual void processExit();

protected:
ClientState(Client& client);

Client& _client;
};

/**
 * First state of the client. Used when the user has not registered it's name
 * yet.
 * Only responds to user input events, awaiting for the user to type a
 * valid name.
 */
class NotRegisteredState: public ClientState
{
public:
NotRegisteredState(Client& client);
~NotRegisteredState();

void processUserInputMessage(const string& userName);
};

/**
 * Intermediate state in between the client has sent a register name response
 * and it's arrival.
 * Only responds to register name responses from the server, and change to
 * IdleState as soon as one arrives.
```

```
 */
class WaitingRegisterNameResponseState: public ClientState
{
public:
WaitingRegisterNameResponseState(Client& client, const string& userName);
~WaitingRegisterNameResponseState();

virtual void processRegisterNameResponse(bool responseOk);

private:
string _userName;
};


/**
 * Abstract superclass of all registered states. That is: all states after the
 * user name has been registered in the server.
 * Responds to the exit event unregistering the name from the server.
 */
class RegisteredState: public ClientState
{
public:
RegisteredState(Client& client, const string& userName);

virtual void processExit();

protected:
/** The user name already registered in the server. */
string _userName;
};

/**
 * Idle state. That is: when the user is not chatting with a peer, nor waiting a
 * start chat response from a peer or from the user.
 * Responds to multiple events:
 *  - User input: The user may enter a command to start chatting with a peer or
 *    to get the peer table.
 *  - Peer table response: the server sent the peer table in response to the
 *    user.
 *  - Start chat request: a peer wants to start a chat session with the user.
 */
class IdleState: public RegisteredState
{
public:

IdleState(Client& client, const string& userName);

virtual void processUserInputMessage(const string& userInput);
virtual void processPeerTableResponse(const ByteArray& data);
virtual void processStartChatRequest(const Peer& peer);

private:

void processStartChatCommand(const string& peerName);
PeerTable _peerTable;
};

/**
 * State when the user has send a start chat request to a peer and the client
 * must wait for it's response.
 */
class WaitingPeerStartChatResponseState: public RegisteredState
{
public:
WaitingPeerStartChatResponseState(Client& client, const string& userName,
const Peer& peer);

virtual void processUserInputMessage(const string& userInput);
virtual void processStartChatResponse(bool responseOk);
```

```
private:
Peer _peer;
};

/**
 * State when a request from a peer to start chatting has arrived and the client
 * must wait for the user to decide whether or not to start chatting with the
 * peer.
 */
class WaitingUserStartChatResponseState: public RegisteredState
{
public:
WaitingUserStartChatResponseState(Client& client, const string& userName,
const Peer& peer);

virtual void processUserInputMessage(const string& userInput);

private:
Peer _peer;
};

/**
 * State when the user and a peer are chatting.
 * Responds to multuiple events:
 *  - Input message: the user has entered some text. It's interpreted as a chat
 *    message and sent to the peer.
 *  - End chat: the peer left the chat session.
 *  - Start chat request by other peer: the client responds that it is busy.
 *  - CLient exists: Not only the user name has to be unregistered from the
 *    server (as with every other RegisteredState) but also a message must be
 *    sent to the peer to inform him that the user has left.
 */
class ChattingState: public RegisteredState
{
public:
ChattingState(Client& client, const string& userName, const Peer& peer);
virtual ~ChattingState();

virtual void processUserInputMessage(const string& userInput);
virtual void processEndChat();
virtual void processChatMessage(const string& chatMessage);
virtual void processStartChatRequest(const Peer& peer);
virtual void processExit();


private:
Peer _peer;
MessageQueue _peerQueue;
};

#endif /* CLIENT_STATE_H_ */

/*
 * exception.h
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */

#ifndef EXCEPTION_H_
#define EXCEPTION_H_

#include <exception>
#include <string>

using std::string;

/**
 * An exception with a stack trace!
 *
 * NOTE: Needs to be linked with the option -rdynamic under gcc in order to
```

```
 * have a legible stack trace.
 */
class Exception: public std::exception
{
public:
explicit Exception(const string& msj = "");
explicit Exception(const std::exception& e);
virtual ~Exception() throw () { }

virtual const char* what() const throw();

private:
string _what;
};

#endif /* EXCEPTION_H_ */

/*
 * random.h
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */

#ifndef RANDOM_H_
#define RANDOM_H_

/**
 * Generates a random integer in the range [0, n) (i.e: has n possible values)
 */
int randomInt(int n);

/**
 * Generates a random integer in the range [min, max]
 */
int randomInt(int min, int max);

#endif /* RANDOM_H_ */

/*
 * logger.h
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#ifndef LOGGER_H_
#define LOGGER_H_

#include "utils.h"


class Logger
{
public:

void log(const string& message);

void setLogging(bool value);

/** Singleton instance */
static Logger& instance();

private:
bool _logging;

/**
 * Log-file file descriptor.
 * Note: raw file descriptors are used instead of FILE* or, preferably,
 * std::ofstream because the FileLock needs a file descriptor in order to
```

```
 * work.
 */
int _fd;

Logger();

~Logger();

DECLARE_NON_COPIABLE(Logger)
};

#endif /* LOGGER_H_ */

/*
 * fifo.h
 *
 *  Created on: 10/04/2010
 *      Author: nicolas
 */

#ifndef IPC_FIFO_H_
#define IPC_FIFO_H_

#include "resource.h"
#include "utils.h"
#include "exception.h"

#include <string>

using std::string;


/**
 * A class responsible for inter-process communication. Fifo (named pipe) is
 * system-persistent and exists beyond the life of the process and must be
 * deleted once it is no longer being used.
 */

class Fifo: public Resource
{
public:
/**
 * Creates a named-pipe.
 *
 * @param pathName     A path to a non-existing file.
 */
explicit Fifo(const string& pathName, bool ownResource = true);

/**
 * Unlinks the associated file to the fifo from the file-system.
 */
virtual ~Fifo() throw ();

/**
 * Returns the pathName of the fifo.
 */
const string getPathName() const { return this->_pathName; }

virtual void print(ostream& stream) const;

protected:
virtual void doDispose() throw ();

private:
const string _pathName;

DECLARE_NON_COPIABLE(Fifo)
};

/**
 * A class responsible for writing on the fifo shared between two processes.
```

```
 */
class FifoWriter
{
public:
/*
 * Opens a named-pipe in writing mode.
 *
 * @param Fifo     A named-pipe that is use to get the pathName to open it.
 */
explicit FifoWriter(const Fifo& fifo);

/**
 * Closes the file of the fifo.
 */
virtual ~FifoWriter();

/**
 * Writes to the fifo.
 *
 * @param buffer Data to be written on the fifo.
 * @param size   The size, in bytes, of the data.
 * @return       The number of bytes written to the fifo. Might be less than
 *               size.
 */
size_t write(const void* buffer, size_t size);

/**
 * Writes a given number of bytes to the fifo.
 *
 * @param buffer Data to be written on the fifo.
 * @param size   The size, in bytes, of the data.
 */
void writeFixedSize(const void* buffer, size_t size);

/**
 * Writes an object to the pipe. 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 written to the pipe.
 */
template <typename T> void write(const T& obj)
{
writeFixedSize(&obj, sizeof(T));
}

private:
int _fileDescriptor;

DECLARE_NON_COPIABLE(FifoWriter)
};

/**
 * A class responsible for reading on the fifo shared between two processes.
 */
class FifoReader
{
public:
/*
 * Opens a named-pipe in reading mode.
 *
 * @param Fifo     A named-pipe that is use to get the pathName to open it.
 */
explicit FifoReader(const Fifo& fifo);

/**
 * Closes the file of the fifo.
 */
virtual ~FifoReader();
```

```
/**
 * Reads from the fifo.
 *
 * @param buffer Destination of the data.
 * @param size   The size, in bytes, of the data to be read.
 * @return       The number of bytes read and stored into the buffer. Might
 *               be less than size. Zero indicates end of file.
 */
size_t read(void* buffer, size_t size);

/**
 * Reads a given number of bytes from the fifo. In case less bytes could be
 * read, an exception is thrown.
 *
 * @param buffer Destination of the data.
 * @param size   The size, in bytes, of the data to be read.
 */
void readFixedSize(void* buffer, size_t size);

/**
 * Reads an object from the pipe. 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 read from the pipe.
 */
template <typename T> void read(T& obj)
{
readFixedSize(&obj, sizeof(T));
}

private:
int _fileDescriptor;

DECLARE_NON_COPIABLE(FifoReader)
};

#endif /* FIFO_H_ */

/*
 * file_lock.h
 *
 *  Created on: 07/07/2010
 *      Author: nicolas
 */

#ifndef FILE_LOCK_H_
#define FILE_LOCK_H_

#include <fcntl.h>

/**
 * A class responsible for locking a file so only one process can access to it
 * at the same time. FileLock is advisory, so it's the programmer responsibility
 * to use then when a file has one. FileLock locks the entire file for reading
 * and writing.
 */
class FileLock
{
public:
/**
 * Locks a file.
 *
 * @param fd  File descriptor of the file to be locked.
 */
explicit FileLock(int fd);

/**
 * Unlocks the file.
 */
virtual ~FileLock();
```

```
private:
int _fd;
struct flock _lockInfo;
};

#endif /* FILE_LOCK_H_ */

/*
 * mutex_set.h
 *
 *  Created on: Apr 19, 2010
 *      Author: demian
 */

#ifndef MUTEX_SET_H_
#define MUTEX_SET_H_

#include "semaphore_set.h"

class MutexProxy;

class MutexSet: public Resource
{
public:
MutexSet(const string& pathName, char id, size_t nMutex,
bool ownResources):
Resource(ownResources),
_semSet(pathName, id, SemaphoreSet::InitValues(nMutex, 1),
ownResources)
{ }

void lock   (size_t mutexIndex) { _semSet.wait          (mutexIndex); }
void unlock (size_t mutexIndex) { _semSet.signal        (mutexIndex); }
bool tryLock(size_t mutexIndex) { return _semSet.tryWait(mutexIndex); }


/**
 * @return A proxy to the mutexIndex'th mutex in the set.
 */
MutexProxy getMutex(size_t mutexIndex);

virtual bool ownResources() const
{ return _semSet.ownResources(); }

virtual void setOwnResources(bool value)
{ _semSet.setOwnResources(value); }

virtual void print(ostream& stream) const
{
stream << "mutex set wrapping ";
_semSet.print(stream);
}

protected:

virtual void doDispose() throw ()
{ /* Nothing. _semSet is the real resource here. */ }

private:
SemaphoreSet _semSet;

DECLARE_NON_COPIABLE(MutexSet)
};


/**
 * A class with two X's in it's name, must be a very sexy class!
 */
class MutexProxy
{
```

```
public:
/** Default copy constructor and assignment operator */

/** @see MutexSet::lock() */
void lock() { _proxy.wait(); }

/** @see MutexSet::unlock() */
void unlock() { _proxy.signal(); }

/** @see MutexSet::tryLock() */
bool tryLock() { return _proxy.tryWait(); }

private:
SemaphoreProxy _proxy;

friend class MutexSet;
MutexProxy(SemaphoreProxy& proxy): _proxy(proxy) { }
};

inline MutexProxy MutexSet::getMutex(size_t mutexIndex)
{
SemaphoreProxy sem = _semSet.getSemaphore(mutexIndex);
return MutexProxy(sem);
}


#endif /* MUTEX_SET_H_ */

/*
 * resource.h
 *
 *  Created on: Apr 19, 2010
 *      Author: demian
 */

#ifndef IPC_RESOURCE_H_
#define IPC_RESOURCE_H_

#include "utils.h"

#include <iostream>

using std::ostream;

class Resource
{
public:
virtual ~Resource() throw ();

virtual bool ownResources() const;
virtual void setOwnResources(bool value);

virtual void print(ostream& stream) const = 0;

protected:
Resource(bool ownResources);

virtual void doDispose() throw () = 0;

private:
bool _ownResources;
bool _disposed;

// ResourceCollector may call dispose(), but no one else! Not even Resource
// subclasses!
friend class ResourceCollector;

void dispose() throw ();

DECLARE_NON_COPIABLE(Resource)
};
```

```
inline ostream& operator << (ostream& stream, const Resource& res)
{
res.print(stream);
return stream;
}

#endif /* RESOURCE_H_ */

/*
 * shared_memory.h
 *
 *  Created on: Apr 8, 2010
 *      Author: demian
 */

#ifndef IPC_SHARED_MEMORY_H_
#define IPC_SHARED_MEMORY_H_

#include "ipc/resource.h"
#include "utils.h"

#include <string>

using std::string;


/**
 * A class responsible for a segment of shared-between-processes memory.
 */
class RawSharedMemory: public Resource
{
public:
/**
 * Creates a zero-initialized shared memory segment of a given size.
 * The shared memory is attached to the calling process.
 *
 * @param size     The size, in bytes, of the shared memory segment.
 * @param pathName A path to an existing file.
 * @param id       A char to identify the shared memory object.
 * @param ownResource Whether the physical shared memory segment
 *                    should be deallocated when the SharedMemory is
 *                    destroyed.
 */
RawSharedMemory(size_t size, const string& pathName, char id,
bool ownResource = true);

/**
 * Detaches the shared memory from the calling process and, in case
 * ownResource is true, the physical shared memory is destroyed.
 */
~RawSharedMemory() throw ();

const void* get() const { return _data; }
  void* get()       { return _data; }

virtual void print(ostream& stream) const;

protected:
virtual void doDispose() throw ();

private:
void* _data;
int   _shmId;

// Non copiable.
DECLARE_NON_COPIABLE(RawSharedMemory)
};
```

```
/**
 * A class responsible for an object of a given type allocated in a
 * shared-between-processes memory.
 * @param T The type of the shared object. The type must be primitive or a
 *          struct of primitives types.
 */
template <typename T>
class SharedMemory: public Resource
{
public:

/**
 * Creates a shared memory segment containing an object of type T.
 * The memory of the object is zero-initialized and it's constructor is not
 * called.
 * The shared memory is attached to the calling process.
 *
 * @param pathName A path to an existing file
 * @param id       A char to identify the shared memory object.
 * @param ownResource Whether the physical shared memory segment
 *                    should be deallocated when the SharedMemory is
 *                    destroyed.
 */
SharedMemory(const string& pathName, char id, bool ownResource):
Resource(ownResource),
_sharedMem(sizeof(T), pathName, id, ownResource)
{ }


/**
 * Detaches the shared memory from the calling process and, in case
 * ownResource is true, the physical shared memory is deallocated.
 * The destructor of the shared object is never called, and it shouldn't
 * be necessary, as it should be a primitive object with no resources to
 * deallocate.
 */
~SharedMemory() throw () { }


/** Returns a reference to the shared object. */
const T& get() const { return * (T*) _sharedMem.get(); }
      T& get()       { return * (T*) _sharedMem.get(); }


virtual bool ownResources() const
{ return _sharedMem.ownResources(); }

virtual void setOwnResources(bool value)
{ _sharedMem.setOwnResources(value); }

virtual void print(ostream& stream) const
{
stream << "shared memory wrapping ";
_sharedMem.print(stream);
}

protected:
virtual void doDispose() throw ()
{ /* Nothing. _sharedMem is the real resource here. */ }

private:
RawSharedMemory _sharedMem;

// Non copiable.
DECLARE_NON_COPIABLE(SharedMemory)
};


#endif /* SHARED_MEMORY_H_ */

/*
```

```
 * semaphore.h
 *
 *  Created on: Apr 16, 2010
 *      Author: demian
 */

#ifndef SEMAPHORE_SET_H_
#define SEMAPHORE_SET_H_

#include "resource.h"
#include "utils.h"
#include <sys/sem.h>

#include <vector>
#include <string>

using std::string;
using std::vector;

class SemaphoreProxy;

/**
 * A set of semaphores.
 */
class SemaphoreSet: public Resource
{
public:

/** Structure used to indicate the initial values of the semaphore set. */
typedef vector<unsigned short> InitValues;

/**
 * Creates a semaphore set.
 * If the process is the first one to create a semaphore set with the given
 * path name and id, a new low-level semaphore set will be created and
 * initialized with the given values.
 * If another process has already created a semaphore set with the same
 * path name and id, this semaphore set will be associated with the same
 * low-level resources, and those will not be initialized (i.e: the initVals
 * vector will be ignored.
 *
 * @param pathName A path to an existing file.
 * @param id       An identifier for the semaphore set.
 * @param nSems    The number of semaphores in the set.
 * @param initVals The initial values for each semaphore in the set.
 * @param ownResources Whether this SemaphoreSet "owns" the external
 *                 system resources associated with it. If true, the system
 *                 resources will be eliminated on SemaphoreSet's
 *                 destruction.
 */
SemaphoreSet(const string& pathName, char id, size_t nSems, int initVals[],
bool ownResources);
SemaphoreSet(const string& pathName, char id, const InitValues& initVals,
bool ownResources);


/**
 * If ownWxternalResources was set to true in construction, destroys the
 * associated external resources associated with the semaphore set.
 */
~SemaphoreSet() throw ();


/**
 * If semIndex-th semaphore's value is greater than zero, decrements the
 * value by 1 and returns immediately.
 * If semIndex-th semaphore's value is zero, the calling process enters the
 * semaphore's waiting queue and waits until the semaphore is signal()ed
 * enough times.
 */
```

```
void wait(size_t semIndex);


/**
 * If there are none processes waiting for the semIndex-th semaphore on the
 * set, it's value is incremented by 1. Otherwise, the first process in the
 * waiting queue is waken up and the value of the semaphore remains zero.
 * Either way, this operation returns immediately.
 */
void signal(size_t semIndex);


/**
 * Tries to perform a wait() operation on the semIndex-th semaphore of the
 * set.
 * The wait() is performed only if it can be done immediately (i.e: the
 * semaphore's value is greater than zero). Otherwise, no wait() is
 * performed, and the method returns immediately too.
 *
 * @return Whether the wait() was performed (i.e: the semaphore count of the
 * semIndex-th semaphore could be decremented by one) or not
 */
bool tryWait(size_t semIndex);


/**
 * @return A proxy to the semIndex-th semaphore in the set.
 */
SemaphoreProxy getSemaphore(size_t semIndex);

virtual void print(ostream& stream) const;

protected:
virtual void doDispose() throw ();

private:
int    _semId;
size_t _nSems;

void validateIndex(size_t semIndex);

DECLARE_NON_COPIABLE(SemaphoreSet)
};


/** A simple proxy for individual semaphores. */
class SemaphoreProxy
{
public:
/** Default copy constructor and assignment operator */

/** @see SemaphoreSet::wait() */
void wait() { _set.wait(_index); }

/** @see SemaphoreSet::signal() */
void signal() { _set.signal(_index); }

/** @see SemaphoreSet::tryWait() */
bool tryWait() { return _set.tryWait(_index); }

private:
SemaphoreSet& _set;
int           _index;

friend class SemaphoreSet;
SemaphoreProxy(SemaphoreSet& set, int index): _set(set), _index(index) { }
};


#endif /* SEMAPHORE_H_ */
```

```
/*
 * resource_collector.h
 *
 *  Created on: Apr 23, 2010
 *      Author: demian
 */

#ifndef IPC_RESOURCE_COLLECTOR_H_
#define IPC_RESOURCE_COLLECTOR_H_

#include "utils.h"
#include <list>

using std::list;

class Resource;

class ResourceCollector
{
public:
/** Singleton instance. */
static ResourceCollector& instance();

void registerResource(Resource* res);
void unregisterResource(Resource* res);

private:
typedef list<Resource*> ResourceList;
ResourceList _resources;

/** Private constructor and destructor. */
ResourceCollector();
~ResourceCollector();

bool alreadyRegistered(Resource* res) const;
void disposeAllResources();

DECLARE_NON_COPIABLE(ResourceCollector)
};


#endif /* RESOURCE_COLLECTOR_H_ */

/*
 * 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_ */

/*
 * ipc_error.h
 *
 *  Created on: Apr 16, 2010
 *      Author: demian
 */

#ifndef IPC_ERROR_H_
#define IPC_ERROR_H_

#include "exception.h"

class IpcError: public Exception
{
public:
IpcError(const string& msj, int errorCode = 0);

int getErrorCode();

private:
int _errorCode;
};


#endif /* IPC_ERROR_H_ */

/*
 * byte_array.h
 *
 *  Created on: 23/05/2010
 *      Author: nicolas
 */

#ifndef BYTE_ARRAY_H_
#define BYTE_ARRAY_H_

#include <vector>
#include <string>

typedef std::vector<char> ByteArray;

ByteArray toByteArray(const void* arrayData, size_t size);

void addToByteArray(ByteArray& byteArray, const void* arrayData, size_t size);
bool getFromByteArray(const ByteArray& byteArray, size_t startIndex,
void* arrayData, size_t size);

void addStringToByteArray(ByteArray& byteArray, const std::string arrayData);
const std::string getStringFromByteArray(const ByteArray& byteArray,
size_t startIndex, size_t size);

ByteArray stringToByteArray(const std::string& str);
const std::string byteArrayToString(const ByteArray& bytes);

class ByteArrayWriter
{
public:
ByteArrayWriter();
```

```
template<typename T>
void write(T t)
{
addToByteArray(_bytes, &t, sizeof(T));
}

void writeString(const std::string&);
void writeByteArray(const ByteArray&);

ByteArray getByteArray();

private:
ByteArray _bytes;
};

class ByteArrayReader
{
public:
ByteArrayReader(const ByteArray&);

template<typename T>
T read()
{
T t;
size_t size = sizeof(T);

getFromByteArray(_bytes, _index, &t, size);
_index += size;

return t;
}

std::string readString();
ByteArray readAll();

private:
const ByteArray& _bytes;
int _index;
};

#endif /* BYTE_ARRAY_H_ */

/*
 * Serializable.h
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#ifndef SERIALIZABLE_H_
#define SERIALIZABLE_H_

#include "byte_array.h"

class Serializable
{
public:
virtual ByteArray serialize() = 0;
virtual void deserialize(const ByteArray& bytes) = 0;
void deserializeFromIndex(const ByteArray& bytes, size_t startIndex,
size_t size);
};

#endif /* SERIALIZABLE_H_ */

/*
 * utils.h
 *
 *  Created on: Apr 12, 2010
```

```
 *      Author: demian
 */

#ifndef UTILS_H_
#define UTILS_H_

#include <stdexcept>
#include <string>
#include <sstream>

using std::string;

/**
 * Nice macro to get the size of a static array.
 */
#define ARR_SIZE(arr) (sizeof(arr) / sizeof(arr[0]))

/**
 * An other nice macro to declare a class non copiable. i.e: declares a private
 * copy constructor and assignment operator. Users of this macro should not
 * define the copy constructor not the assignment operator.
 */
#define DECLARE_NON_COPIABLE(className) private: \
className(className&); \
void operator = (className&);

/**
 * Transforms a string to an integer. Equivalent to atoi function, except this
 * one throws a exception in case of a bad format.
 */
int strToInt(const char* str) throw (std::invalid_argument);

/**
 * Transforms any object of type T to a string. The operator << (ostring&, T)
 * must be defined.
 */
template<typename T> string toStr(T obj)
{
std::ostringstream oss;
oss << obj;
return oss.str();
}

/**
 * Generates a copy of a given string with no leading whitespace.
 * e.g: trimLeft("  hello world ") returns "hello world "
 */
string trimLeft(const string& str);

/**
 * Generates a copy of a given string with no trailing whitespace.
 * e.g: trimRight(" hello world  ") returns " hello world"
 */
string trimRight(const string& str);

/**
 * Generates a copy of a given string with no leading nor trailing whitespace.
 * e.g: trim("  hello world  ") returns "hello world"
 */
string trim(const string& str);

/**
 * Generates a lowercase version of a string.
 * e.g: toLowerCase("AbCd") returns "abcd"
 */
string toLowerCase(const string& str);


#endif /* UTILS_H_ */

/*
```

```
 * global_config.h
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#ifndef GLOBAL_CONFIG_H_
#define GLOBAL_CONFIG_H_

#include "config.h"
#include "utils.h"

#include <memory>

class GlobalConfig
{
public:

template <typename T>
static T get(const string& key)
{
if (!_config.get())
throw Exception("No configuration has been set");
return _config->get<T>(key);
}

typedef std::auto_ptr<Config> ConfigPtr;

static void setConfig(ConfigPtr config);

private:
static ConfigPtr _config;

DECLARE_NON_COPIABLE(GlobalConfig)
};

#endif /* GLOBAL_CONFIG_H_ */

/*
 * person.h
 *
 *  Created on: Apr 9, 2010
 *      Author: demian
 */

#ifndef PERSON_H_
#define PERSON_H_

#include <cstring>

class Person
{
public:
explicit Person(const char* name = "", int age = 0)
{
setName(name);
_age = age;
}

const char* name() { return _name; }
int         age()  { return _age; }

void becomeOlder() { _age++; }

private:

static const size_t NAME_SIZE = 256;

char _name[NAME_SIZE];
int  _age;
```

```
void setName(const char* name)
{
strncpy(_name, name, NAME_SIZE);
}

};

#endif /* PERSON_H_ */

/*
 * config.h
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#ifndef CONFIG_H_
#define CONFIG_H_

#include "exception.h"

#include <string>
#include <sstream>

using std::string;
using std::istringstream;

/**
 * An abstract configuration that maps string keys with values.
 */
class Config
{
public:
virtual ~Config();

/**
 * Retrieves the value of type T associated to a given key.
 */
template<typename T>
T get(const string& key)
{
string value = getString(key);
istringstream iss(value);
T ret;
iss >> ret;

if (iss.fail())
throw Exception("Error reading value. Key = " + key + " value = "
+ value);

return ret;
}

protected:
Config();

virtual string getString(const string& key) const = 0;
};

#endif /* CONFIG_H_ */

/*
 * queue_utils.h
 *
 *  Created on: 29/06/2010
 *      Author: nicolas
 */

#ifndef QUEUE_UTILS_H_
#define QUEUE_UTILS_H_
```

```
#include <string>

#include <sys/types.h>

using std::string;

string getClientQueueFileName(pid_t pid = getpid());
string getServerQueueFileName();

#endif /* QUEUE_UTILS_H_ */

/*
 * peers_table.h
 *
 *  Created on: May 11, 2010
 *      Author: ???
 */

#ifndef PEER_TABLE_H_
#define PEER_TABLE_H_

#include <vector>
#include <string>
#include <iosfwd>

#include "core/serializable.h"

using std::vector;
using std::string;

class Peer
{
public:
Peer(const string& name, pid_t id):
_name(name), _id(id)
{ }

const string& getName() const { return _name; }
const pid_t& getId()   const { return _id; }

bool operator == (const Peer& p)
{
return _name == p._name && _id == p._id;
}

friend std::ostream& operator << (std::ostream& os, const Peer& peer);

private:
string _name;
pid_t _id;
};

class PeerTable : public Serializable
{
public:
PeerTable()
{ }

void add(const Peer& peer);
void remove(const string& peerName);

bool containsId(pid_t peerId) const;
bool containsName(const string& peerName) const;

const Peer* getById(pid_t peerId) const;
const Peer* getByName(const string& peerName) const;


ByteArray serialize();
void deserialize(const ByteArray& bytes);
```

```
friend std::ostream& operator << (std::ostream& os, const PeerTable& peers);

private:
typedef vector<Peer> PeerVector;
PeerVector _peers;
};

#endif /* PEER_TABLE_H_ */

/*
 * model_error.h
 *
 *  Created on: May 11, 2010
 *      Author: demian
 */

#ifndef MODEL_ERROR_H_
#define MODEL_ERROR_H_

#include "exception.h"

class ModelError: public Exception
{
public:
ModelError(const string& msj): Exception(msj)
{ }
};


#endif /* MODEL_ERROR_H_ */

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

#ifndef MESSAGE_H_
#define MESSAGE_H_

#include "core/byte_array.h"
#include "core/serializable.h"
#include <string>

using std::string;

class Message: public Serializable
{
public:

/**
 * The different types of messages.
 */
enum MessageType
{
/** Invalid message type */
TYPE_NONE = 0,

/**
 * Sent from input process to main client process when user enters
 * some text on the screen.
 */
TYPE_USER_INPUT,

/**
 * Sent from input process to main client process when user enters the
 * exit command.
 */
```

```
TYPE_USER_EXIT,

/** Sent to server to register a new name. */
TYPE_REGISTER_NAME_REQUEST,

/**
 * Sent from server to client to inform whether the name was registered
 * or not.
 */
TYPE_REGISTER_NAME_RESPONSE,

/** Sent to server to unregister a name. */
TYPE_UNREGISTER_NAME_REQUEST,

/** Sent to server to make it show the peer table. */
TYPE_SHOW_PEER_TABLE_REQUEST,

/** Sent from client to server to request the current peer table. */
TYPE_PEER_TABLE_REQUEST,

/**
 * Sent from server to client in response of the peer table request.
 * Contains the peer table data.
 */
TYPE_PEER_TABLE_RESPONSE,

/**
 * Sent from peer to peer client to let the second know that the first
 * wants to start a chat session.
 */
TYPE_START_CHAT_REQUEST,

/**
 * Sent from peer to peer in response of a start chat request by the
 * second  informing whether the first wants to chat or not.
 */
TYPE_START_CHAT_RESPONSE,

/**
 * Sent from peer to peer to inform the second that the first has left
 * the chat session.
 */
TYPE_END_CHAT,

/**
 * Sent from peer to peer. A chat message.
 */
TYPE_CHAT_MESSAGE,

/**
 * Sent to server to make it quit remotely.
 */
TYPE_SERVER_EXIT
};

Message() :
_type(TYPE_NONE), _messengerPid(0)
{
}

Message(MessageType type, pid_t messengerPid, const ByteArray& data =
ByteArray()) :
_type(type), _messengerPid(messengerPid), _data(data)
{
}

MessageType getType() const
{
return _type;
}
```

```
pid_t getMessengerPid() const
{
return _messengerPid;
}

const ByteArray& getData() const
{
return _data;
}

virtual ByteArray serialize()
{
ByteArrayWriter writer;
writer.write(_type);
writer.write(_messengerPid);
writer.writeByteArray(_data);
return writer.getByteArray();
}

virtual void deserialize(const ByteArray& bytes)
{
ByteArrayReader reader(bytes);
_type = reader.read<MessageType> ();
_messengerPid = reader.read<pid_t> ();
_data = reader.readAll();
}

private:
MessageType _type;
pid_t _messengerPid;
ByteArray _data;
};

#endif /* MESSAGE_H_ */

/*
 * arg_parse.h
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#ifndef ARG_PARSE_H_
#define ARG_PARSE_H_

/**
 * Parses the command-line arguments for both client and server.
 */
void parseArguments(int argc, char* argv[]);

/**
 * Loads the configuration file for both client and server and sets it as the
 * gloabl config.
 */
void loadConfigFile();

#endif /* ARG_PARSE_H_ */

/*
 * config_file.h
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#ifndef CONFIG_FILE_H_
#define CONFIG_FILE_H_

#include "config.h"
```

```
#include <map>

using std::map;

/**
 * A Unix-style configuration file.
 * Each non blank line of the file can be either a comment (if it starts with
 * '#') or a key-value pair o the form:
 * key = value
 */
class ConfigFile: public Config
{
public:
ConfigFile(const string& fileName);
virtual ~ConfigFile();

protected:
virtual string getString(const string& key) const;

private:
map<string, string> _data;
};

#endif /* CONFIG_FILE_H_ */

/*
 * constants.h
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#ifndef CONSTANTS_H_
#define CONSTANTS_H_

class CommonConstants
{
public:
/**
 * TODO: move to global config.
 */
static const char QUEUE_ID = 'Q';
};

#endif /* CONSTANTS_H_ */

/*
 * main.cpp
 *
 *  Created on: 29/06/2010
 *      Author: nicolas
 */

#include "exception.h"
#include "ipc/message_queue.h"
#include "model/queue_utils.h"
#include "model/peer_table.h"
#include "model/message.h"
#include "constants.h"
#include "utils.h"
#include "core/byte_array.h"

#include <iostream>
#include <string>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <cstdlib>
```

```
#include <getopt.h>
#include <cstring>

using std::cout;
using std::cerr;
using std::string;

void unregisterUser(MessageQueue& serverQueue, string userName);
void showPeerTable(MessageQueue& serverQueue);
void quit(MessageQueue& serverQueue);
void showHelp(const string& commandName);

int main(int argc, char **argv)
{
string commandName = basename(argv[0]);

if (argc == 1)
{
showHelp(commandName);
return EXIT_SUCCESS;
}

MessageQueue serverQueue(getServerQueueFileName(),
CommonConstants::QUEUE_ID, false);

int c;
opterr = 0;
struct option longOptions[] =
{
{ "help",       no_argument,       0, 'h' },
{ "unregister", required_argument, 0, 'u' },
{ "show-peers", no_argument,       0, 's' },
{ "quit",       no_argument,       0, 'q' },
{ 0, 0, 0, 0 }
};

while ((c = getopt_long(argc, argv, ":hu:sq", longOptions, NULL)) != -1)
{
switch (c)
{
case 'h':
showHelp(commandName);
break;
case 'u':
unregisterUser(serverQueue, optarg);
break;
case 'm':
showPeerTable(serverQueue);
break;
case 'q':
quit(serverQueue);
break;
case ':':
cerr << "Missing argument for option " << char(optopt) << "\n";
showHelp(commandName);
return EXIT_FAILURE;
case '?':
cerr << "Invalid option: ";
if (optopt != 0)
cerr << char(optopt) << "\n";
else
cerr << argv[optind - 1] << "\n";
showHelp(commandName);
return EXIT_FAILURE;
default:
cerr << "Error parsing arguments\n";
return EXIT_FAILURE;
}
}
```

```
return EXIT_SUCCESS;
}

void showHelp(const string& commandName)
{
cout << "Usage: " << commandName << " [option]\n";
cout << "\n";
cout << "Server administration utility.\n";
cout << "\n";
cout << "Options:\n";
cout << "    -h, --help                  Show this help message and exit\n";
cout << "    -u, --unregister <USERNAME> Unregister USERNAME user\n";
cout << "    -s, --show-peers            Show peer table at server\n";
}

void unregisterUser(MessageQueue& serverQueue, string userName)
{
ByteArrayWriter writer;
writer.writeString(userName);

Message messageSent(Message::TYPE_UNREGISTER_NAME_REQUEST, getpid(),
writer.getByteArray());
serverQueue.sendByteArray(messageSent.serialize());
}

void showPeerTable(MessageQueue& serverQueue)
{
Message messageSent(Message::TYPE_SHOW_PEER_TABLE_REQUEST, getpid());
serverQueue.sendByteArray(messageSent.serialize());
}

void quit(MessageQueue& serverQueue)
{
Message messageSent(Message::TYPE_SERVER_EXIT, getpid());
serverQueue.sendByteArray(messageSent.serialize());
}

/*
 * main.cpp
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#include "server.h"
#include "common.h"

#include <iostream>

int main(int argc, char* argv[])
{
loadConfigFile();
parseArguments(argc, argv);

try
{
Server server;
return server.run();

} catch (std::exception& e)
{
std::cerr << "Exception thrown " << e.what() << "\n";
} catch (...)
{
std::cerr << "Unknown error\n";
}
}


/*
```

```
 * server.cpp
 *
 *  Created on: 26/06/2010
 *      Author: nicolas
 */

#include "server.h"
#include "exception.h"
#include "ipc/message_queue.h"
#include "constants.h"
#include "core/byte_array.h"
#include "model/queue_utils.h"
#include "model/model_error.h"
#include "logger.h"

#include <iostream>
#include <sstream>

#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>

using std::cout;
using std::cerr;
using std::ostringstream;

namespace
{

void log(const string& message)
{
ostringstream oss;
oss << "Server " << getpid() << ": " << message;
Logger::instance().log(oss.str());
}

void showServerStartMessage()
{
cout << "Servicio de Localización iniciado. Usar el comando "
<< "'server-admin --quit' o una señal SIGINT (crtl + C) para salir\n";
log("Up and running...");
}

void showServerExitMessage()
{
cout << "Cerrando Servicio de Localización...\n";
log("Closing down");
}

void signalHandler(int)
{
showServerExitMessage();
exit(EXIT_SUCCESS);
}

} // end namespace


Server::Server()
{
}

Server::~Server()
{
}

int Server::run()
```

```
{
createQueueFile();
atexit(destroyQueueFile);

signal(SIGINT, signalHandler);
signal(SIGTERM, signalHandler);

MessageQueue queue(getServerQueueFileName(), CommonConstants::QUEUE_ID,
true);
bool exit = false;

showServerStartMessage();

while (!exit)
{
Message message;
message.deserialize(queue.receiveByteArray());
processMessage(message, exit);
}

showServerExitMessage();

return EXIT_SUCCESS;
}

void Server::createQueueFile()
{
if (mknod(getServerQueueFileName().c_str(), 0666, 0) == -1)
throw Exception("could not create file " + getServerQueueFileName());
}

void Server::destroyQueueFile()
{
bool unlinkError = unlink(getServerQueueFileName().c_str()) == -1;

if (unlinkError && errno != ENOENT) // ENOENT = No such file.
throw Exception("could not destroy file " + getServerQueueFileName());
}

void Server::processMessage(const Message& message, bool& exit)
{
ByteArrayReader reader(message.getData());

switch (message.getType())
{
case Message::TYPE_PEER_TABLE_REQUEST:
{
pid_t userPid = message.getMessengerPid();

processPeerTableRequest(userPid);
break;
}
case Message::TYPE_REGISTER_NAME_REQUEST:
{
string userName = reader.readString();
pid_t userPid = message.getMessengerPid();

processRegisterNameRequest(userName, userPid);
break;
}
case Message::TYPE_UNREGISTER_NAME_REQUEST:
{
string userName = reader.readString();

processUnregisterNameRequest(userName);
break;
}
case Message::TYPE_SHOW_PEER_TABLE_REQUEST:
{
processShowPeerTableRequest();
```

```
break;
}
case Message::TYPE_SERVER_EXIT:
{
exit = true;
break;
}
default:
throw Exception("Invalid message type " + toStr(message.getType()));
}
}

void Server::processRegisterNameRequest(const string& userName, pid_t userPid)
{
bool registerOk = !_peerTable.containsName(userName);
if (registerOk)
{
Peer peer(userName, userPid);
_peerTable.add(peer);
}

ostringstream logMsg;
logMsg << "Received register name request from " << userPid << ", name = "
<< userName << ", register OK = " << std::boolalpha << registerOk;
log(logMsg.str());

string userQueueFileName = getClientQueueFileName(userPid);
MessageQueue queue(userQueueFileName, CommonConstants::QUEUE_ID, false);

ByteArrayWriter writer;
writer.write(registerOk);

Message message(Message::TYPE_REGISTER_NAME_RESPONSE, getpid(),
writer.getByteArray());
queue.sendByteArray(message.serialize());
}

void Server::processPeerTableRequest(pid_t userPid)
{
ostringstream logMsg;
logMsg << "Received peer table request from " << userPid;
log(logMsg.str());

string userQueueFileName = getClientQueueFileName(userPid);
MessageQueue queue(userQueueFileName, CommonConstants::QUEUE_ID, false);

Message message(Message::TYPE_PEER_TABLE_RESPONSE, getpid(),
_peerTable.serialize());
queue.sendByteArray(message.serialize());
}

void Server::processUnregisterNameRequest(const string& userName)
{
try
{
_peerTable.remove(userName);
ostringstream logMsg;
logMsg << "Received unregister name request, name = " << userName;
log(logMsg.str());
} catch (ModelError& e)
{
cerr << "Couldn't remove. The user doesn't exit\n";
}
}

void Server::processShowPeerTableRequest()
{
ostringstream logMsg;
logMsg << "Received show peer table request. " << _peerTable;
log(logMsg.str());
```

```
cout << _peerTable;
}

/*
 * client.cpp
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#include "client.h"
#include "exception.h"
#include "ipc/message_queue.h"
#include "model/queue_utils.h"
#include "constants.h"
#include "core/byte_array.h"
#include "client_state.h"
#include "ipc/ipc_error.h"
#include "logger.h"

#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>

#include <string>
#include <sstream>
#include <iostream>
#include <cstring>

using std::ostringstream;
using std::string;

namespace
{

void log(const string& message)
{
ostringstream oss;
oss << "Client " << getpid() << ": " << message;
Logger::instance().log(oss.str());
}

} // end namespace


Client::Client() :
_state(0)
{
_queueFileName = getClientQueueFileName();
}

Client::~Client()
{
changeState(0);
}

int Client::run()
{
signal(SIGINT, SIG_IGN);
signal(SIGTERM, SIG_IGN);

createQueueFile();

getView().showWelcomeMessage();

pid_t pid = fork();
switch (pid)
```

```
{
case -1:
throw Exception("fork()");
break;
case 0: // Child.
runUserInputProcess();
break;
default: // Parent.
changeState(new NotRegisteredState(*this));
runMainProcess();
destroyQueueFile();
break;
}
return EXIT_SUCCESS;
}

void Client::changeState(ClientState* newState)
{
if (_state)
delete _state;
_state = newState;
}

bool Client::sendRegisterNameRequest(const string& userName)
{
ByteArrayWriter writer;
writer.writeString(userName);
ByteArray data = writer.getByteArray();

return sendMessageToServer(Message::TYPE_REGISTER_NAME_REQUEST, data);
}

bool Client::sendUnregisterNameRequest(const string& userName)
{
ByteArrayWriter writer;
writer.writeString(userName);
ByteArray data = writer.getByteArray();

return sendMessageToServer(Message::TYPE_UNREGISTER_NAME_REQUEST, data);
}

bool Client::sendPeerTableRequest()
{
sendMessageToServer(Message::TYPE_PEER_TABLE_REQUEST);
}

bool Client::sendStartChatRequest(const Peer& peer, const string& userName)
{
ByteArrayWriter writer;
writer.writeString(userName);
ByteArray data = writer.getByteArray();

return sendMessageToPeer(peer, Message::TYPE_START_CHAT_REQUEST, data);
}

bool Client::sendStartChatResponse(const Peer& peer, bool responseOk)
{
ByteArrayWriter writer;
writer.write(responseOk);
ByteArray data = writer.getByteArray();

return sendMessageToPeer(peer, Message::TYPE_START_CHAT_RESPONSE, data);
}

bool Client::sendChatMessage(MessageQueue& peerQueue, const string& chatMessage)
{
ByteArrayWriter writer;
writer.writeString(chatMessage);
ByteArray data = writer.getByteArray();
```

```
return sendMessageToQueue(peerQueue, Message::TYPE_CHAT_MESSAGE, data);
}

bool Client::sendEndChatMessage(MessageQueue& peerQueue)
{
return sendMessageToQueue(peerQueue, Message::TYPE_END_CHAT);
}

ClientView& Client::getView()
{
return _view;
}

void Client::runUserInputProcess()
{
MessageQueue queue(_queueFileName, CommonConstants::QUEUE_ID, false);

string line;
bool exit = false;

while (!exit && bool(std::getline(std::cin, line)))
{
if (trim(line) == ClientView::EXIT_COMMAND)
{
exit = true;
}
else
{
ByteArrayWriter writer;
writer.writeString(line);
ByteArray data = writer.getByteArray();
sendMessageToQueue(queue, Message::TYPE_USER_INPUT, data);
}
}
sendMessageToQueue(queue, Message::TYPE_USER_EXIT);
}

void Client::runMainProcess()
{
log("Up and running...");

MessageQueue queue(_queueFileName, CommonConstants::QUEUE_ID, true);
bool exit = false;

while (!exit)
{
Message message;
message.deserialize(queue.receiveByteArray());
processMessage(message, exit);
}

log("Closing down");
}

void Client::createQueueFile()
{
if (mknod(_queueFileName.c_str(), 0666, 0) == -1)
throw Exception("could not create file " + _queueFileName);
}

void Client::destroyQueueFile()
{
bool unlinkError = unlink(_queueFileName.c_str()) == -1;

if (unlinkError && errno != ENOENT) // ENOENT = no such file.
throw Exception("destroyQueueFile(): could not unlink the queue file "
+ _queueFileName);
}

void Client::processMessage(const Message& message, bool& exitNow)
```

```
{
ByteArrayReader reader(message.getData());

switch (message.getType())
{
case Message::TYPE_USER_INPUT:
{
string userInput = reader.readString();
_state->processUserInputMessage(userInput);
break;
}
case Message::TYPE_USER_EXIT:
{
exitNow = true;
_state->processExit();
break;
}
case Message::TYPE_REGISTER_NAME_RESPONSE:
{
bool responseOk = reader.read<bool> ();
_state->processRegisterNameResponse(responseOk);
break;
}
case Message::TYPE_PEER_TABLE_RESPONSE:
{
_state->processPeerTableResponse(message.getData());
break;
}
case Message::TYPE_START_CHAT_REQUEST:
{
string peerName = reader.readString();
Peer peer(peerName, message.getMessengerPid());
_state->processStartChatRequest(peer);
break;
}
case Message::TYPE_START_CHAT_RESPONSE:
{
bool responseOk = reader.read<bool> ();
_state->processStartChatResponse(responseOk);
break;
}
case Message::TYPE_CHAT_MESSAGE:
{
string chatMessage = reader.readString();
_state->processChatMessage(chatMessage);
break;
}
case Message::TYPE_END_CHAT:
{
_state->processEndChat();
break;
}
default:
throw Exception("Invalid message type " + toStr(message.getType()));
}
}

bool Client::sendMessageToQueue(MessageQueue& queue, MessageType type,
const ByteArray& data)
{
try
{
Message message(type, getpid(), data);
queue.sendByteArray(message.serialize());
return true;
} catch (IpcError& e)
{
// TODO log
std::cout
<< "Could not send message to queue. IpcError thrown. Error code: "
```

```
<< e.getErrorCode() << " - " << strerror(e.getErrorCode())
<< "\n";
return false;
}
}

bool Client::sendMessage(const string& queueFileName, MessageType type,
const ByteArray& data)
{
try
{
MessageQueue queue(queueFileName, CommonConstants::QUEUE_ID, false);
return sendMessageToQueue(queue, type, data);
} catch (IpcError& e)
{
// TODO log
std::cout
<< "Could not create message queue. IpcError thrown. Error code: "
<< e.getErrorCode() << " - " << strerror(e.getErrorCode())
<< "\n";
return false;
}
}

bool Client::sendMessageToServer(MessageType type, const ByteArray& data)
{
bool couldSend = sendMessage(getServerQueueFileName(), type, data);
if (!couldSend)
getView().showCouldNotContactServer();
return couldSend;
}

bool Client::sendMessageToPeer(const Peer& peer, MessageType type,
const ByteArray& data)
{
string peerQueueFileName = getClientQueueFileName(peer.getId());
bool couldSend = sendMessage(peerQueueFileName, type, data);
if (!couldSend)
getView().showCouldNotContactPeer(peer.getName());
return couldSend;
}


/*
 * main.cpp
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#include "client.h"
#include "common.h"

#include <iostream>

int main(int argc, char* argv[])
{
loadConfigFile();
parseArguments(argc, argv);

try
{
Client client;
return client.run();

} catch (std::exception& e)
{
std::cerr << "Exception thrown " << e.what() << "\n";
} catch (...)
{
std::cerr << "Unknown error\n";
```

```
}
}


/*
 * client_state.cpp
 *
 *  Created on: Jun 26, 2010
 *      Author: demian
 */

#include "client_state.h"
#include "client.h"
#include "exception.h"
#include "core/byte_array.h"
#include "utils.h"
#include "model/queue_utils.h"
#include "constants.h"

#include <string>
#include <algorithm>
#include <iostream>

using std::cout;
using std::cerr;
using std::string;

class NotSupportedMethodException: public Exception
{
public:
NotSupportedMethodException() :
Exception("Not supported method")
{
}
};

ClientState::ClientState(Client& client) :
_client(client)
{

}

ClientState::~ClientState()
{
}

void ClientState::processUserInputMessage(const string& userInput)
{
cerr << "Invalid processUserInputMessage() call. userInput= " << userInput
<< "\n";
}

void ClientState::processRegisterNameResponse(bool responseOk)
{
cerr << "Invalid processRegisterNameResponse() call. responseOk="
<< std::boolalpha << responseOk << "\n";
}

void ClientState::processPeerTableResponse(const ByteArray&)
{
cerr << "Invalid processPeerTableResponse() call\n";
}

void ClientState::processStartChatRequest(const Peer& peer)
{
cerr << "Invalid processStartChatRequest() call. peerName=" << peer << "\n";
}

void ClientState::processStartChatResponse(bool responseOk)
{
cerr << "Invalid processStartChatResponse() call. responseOk="
```

```
<< responseOk << "\n";
}

void ClientState::processEndChat()
{
cerr << "Invalid processEndChat() call.\n";
}

void ClientState::processChatMessage(const string& chatMessage)
{
cerr << "Invalid processChatMessage() call. chatMessage=" << chatMessage
<< "\n";
}

void ClientState::processExit()
{
}

NotRegisteredState::NotRegisteredState(Client& client) :
ClientState(client)
{
_client.getView().askUserName();
}

NotRegisteredState::~NotRegisteredState()
{
}

void NotRegisteredState::processUserInputMessage(const string& userInput)
{
string userName = trim(userInput);
bool valid = std::count_if(userName.begin(), userName.end(), isalnum) > 0;
if (!valid)
{
_client.getView().showInvalidName(userName);
}
else if (_client.sendRegisterNameRequest(userName))
{
_client.changeState(new WaitingRegisterNameResponseState(_client,
userName));
}
}

WaitingRegisterNameResponseState::WaitingRegisterNameResponseState(
Client& client, const string& userName) :
ClientState(client), _userName(userName)
{
}

WaitingRegisterNameResponseState::~WaitingRegisterNameResponseState()
{
}

void WaitingRegisterNameResponseState::processRegisterNameResponse(
bool responseOk)
{
if (responseOk)
_client.changeState(new IdleState(_client, _userName));
else
{
_client.getView().showAlreadyUsedName(_userName);
_client.changeState(new NotRegisteredState(_client));
}
}

RegisteredState::RegisteredState(Client& client, const string& userName) :
ClientState(client), _userName(userName)
{
}
```

```
void RegisteredState::processExit()
{
_client.sendUnregisterNameRequest(_userName);
}

IdleState::IdleState(Client& client, const string& userName) :
RegisteredState(client, userName)
{
_client.getView().showIdleStateCommands();
_client.sendPeerTableRequest();
}

void IdleState::processUserInputMessage(const string& userInput)
{
string trimmedInput = trim(userInput);
if (trimmedInput == ClientView::PEER_TABLE_COMMAND)
{
_client.sendPeerTableRequest();
}
else if (trimmedInput.find(ClientView::START_CHAT_COMMAND) == 0)
{
string peerName = trim(trimmedInput.substr(
ClientView::START_CHAT_COMMAND.size()));
processStartChatCommand(peerName);
}
else
{
_client.getView().showInvalidCommand(trimmedInput);
}
}

void IdleState::processStartChatCommand(const string& peerName)
{
if (peerName == _userName)
{
_client.getView().showCannotChatWithYourself();
}
else
{
const Peer* peer = _peerTable.getByName(peerName);
if (peer && _client.sendStartChatRequest(*peer, _userName))
_client.changeState(new WaitingPeerStartChatResponseState(_client,
_userName, *peer));
else
_client.getView().showInvalidPeerName(peerName);
}
}

void IdleState::processPeerTableResponse(const ByteArray& data)
{
_peerTable.deserialize(data);
_client.getView().showPeerTable(_peerTable);
}

void IdleState::processStartChatRequest(const Peer& peer)
{
_client.changeState(new WaitingUserStartChatResponseState(_client,
_userName, peer));
}

WaitingPeerStartChatResponseState::WaitingPeerStartChatResponseState(
Client& client, const string& userName, const Peer& peer) :
RegisteredState(client, userName), _peer(peer)
{
_client.getView().showWaitingPeerResponse(peer.getName());
}

void WaitingPeerStartChatResponseState::processUserInputMessage(const string&)
{
// TODO: tratar el cancelar (?)
```

```
_client.getView().showWaitingPeerResponse(_peer.getName());
}

void WaitingPeerStartChatResponseState::processStartChatResponse(
bool responseOk)
{
if (responseOk)
{
_client.getView().showPeerAcceptedChat(_peer.getName());
_client.changeState(new ChattingState(_client, _userName, _peer));
}
else
{
_client.getView().showPeerCanceledChat(_peer.getName());
_client.changeState(new IdleState(_client, _userName));
}
}

WaitingUserStartChatResponseState::WaitingUserStartChatResponseState(
Client& client, const string& userName, const Peer& peer) :
RegisteredState(client, userName), _peer(peer)
{
_client.getView().askUserStartChatWith(_peer.getName());
}

void WaitingUserStartChatResponseState::processUserInputMessage(
const string& userInput)
{
string trimmedInput = trim(userInput);
bool userSayYes = _client.getView().isYesString(trimmedInput);
bool userSayNo = _client.getView().isNoString(trimmedInput);

if (userSayYes || userSayNo)
{
bool startChatting = userSayYes;
bool couldSendResponse = _client.sendStartChatResponse(_peer,
startChatting);

if (startChatting && couldSendResponse)
_client.changeState(new ChattingState(_client, _userName, _peer));
else
_client.changeState(new IdleState(_client, _userName));
}
else
{
_client.getView().askUserStartChatWith(_peer.getName());
}
}

ChattingState::ChattingState(Client& client, const string& userName,
const Peer& peer) :
RegisteredState(client, userName), _peer(peer), _peerQueue(
getClientQueueFileName(peer.getId()), CommonConstants::QUEUE_ID,
false)
{
_client.getView().showStartChatSession(_peer.getName());
}

ChattingState::~ChattingState()
{
_client.getView().showEndChatSession(_peer.getName());
}

void ChattingState::processUserInputMessage(const string& userInput)
{
bool endChat = false;
if (trim(userInput) == ClientView::END_CHAT_COMMAND)
{
_client.sendEndChatMessage(_peerQueue);
endChat = true;
```

```
}
else
{
endChat = !_client.sendChatMessage(_peerQueue, userInput);
}

if (endChat)
_client.changeState(new IdleState(_client, _userName));
}

void ChattingState::processEndChat()
{
_client.getView().showPeerLeftChat(_peer.getName());
_client.changeState(new IdleState(_client, _userName));
}

void ChattingState::processChatMessage(const string& chatMessage)
{
_client.getView().showChatMessage(_peer.getName(), chatMessage);
}

void ChattingState::processStartChatRequest(const Peer& peer)
{
_client.sendStartChatResponse(peer, false);
}

void ChattingState::processExit()
{
RegisteredState::processExit();
_client.sendEndChatMessage(_peerQueue);
}


/*
 * client_view.cpp
 *
 *  Created on: Jun 26, 2010
 *      Author: demian
 */

#include "client_view.h"
#include "client_state.h"
#include "utils.h"

#include <iostream>
#include <sstream>

namespace
{

template<typename T> void print(const T& t)
{
std::cout << t << std::flush;
}

template<typename T> void println(const T& t)
{
std::cout << t << std::endl;
}

void println()
{
println("");
}

void printLineSeparator()
{
println("========================================"
"========================================");
}
```

```
} // end namespace

const string ClientView::EXIT_COMMAND = "-salir";
const string ClientView::PEER_TABLE_COMMAND = "-usuarios";
const string ClientView::START_CHAT_COMMAND = "-hola";
const string ClientView::END_CHAT_COMMAND = "-adios";
const string ClientView::LOWERCASE_YES = "si";
const string ClientView::LOWERCASE_NO = "no";

ClientView::ClientView()
{
}

void ClientView::showWelcomeMessage()
{
printLineSeparator();
println("Bienvenido a Concu-Chat 2010!");
println(
"Escribí " + EXIT_COMMAND
+ " o un caracter e fin de archivo (ctrl + D) para salir cuando quieras.");
printLineSeparator();
}

void ClientView::askUserName()
{
print("Tu nombre?: ");
}

void ClientView::showCouldNotContactServer()
{
println("No se pudo contactar al servicio de localización =(");
}

void ClientView::showCouldNotContactPeer(const string& peerName)
{
println("No se pudo contactar a " + peerName + " =(");
}

void ClientView::showInvalidName(const string& userName)
{
println("El nombre \"" + userName + "\" no es válido");
}

void ClientView::showAlreadyUsedName(const string& userName)
{
println("El nombre \"" + userName
+ "\" ya está siendo usado. Intentá con otro nombre =P");
}

void ClientView::showPeerTable(const PeerTable& peerTable)
{
print(peerTable);
}

void ClientView::showIdleStateCommands()
{
println("Comandos:");
println(PEER_TABLE_COMMAND + " para mostrar los usuarios conectados");
println(START_CHAT_COMMAND + " <nombre-usuario> para chatear con alguien");
}

void ClientView::showCannotChatWithYourself()
{
println("No podés chatear con vos mismo! (lo siento...)");
}

void ClientView::showInvalidPeerName(const string& peerName)
{
println("El usuario " + peerName
+ " no existe. Intentá actualizar la tabla de usuarios");
```

```
}

void ClientView::showInvalidCommand(const string& command)
{
println("Comando invlálido: " + command);
}

void ClientView::showWaitingPeerResponse(const string& peerName)
{
println("Esperando la respuesta de " + peerName);
}

void ClientView::askUserStartChatWith(const string& peerName)
{
println(peerName + " quiere chatear con vos");
print("Querés? (" + LOWERCASE_YES + "/" + LOWERCASE_NO + "): ");
}

void ClientView::showPeerCanceledChat(const string& peerName)
{
println(peerName + " está ocupado. Lo siento mucho =(");
}

void ClientView::showPeerAcceptedChat(const string& peerName)
{
println(peerName + " aceptó!");
}

void ClientView::showStartChatSession(const string& peerName)
{
printLineSeparator();
println("Sesión de chat con " + peerName + " iniciada");
println("Escribí " + END_CHAT_COMMAND + " para terminar la sesión");
printLineSeparator();
}

void ClientView::showEndChatSession(const string& peerName)
{
println("Sesión de chat con " + peerName + " terminada");
printLineSeparator();
}

void ClientView::showPeerLeftChat(const string& peerName)
{
println(peerName + " se ha ido =(");
}

void ClientView::showChatMessage(const string& peerName,
const string& chatMessage)
{
println(peerName + " dice: " + chatMessage);
}

bool ClientView::isYesString(const string& str)
{
return toLowerCase(str) == LOWERCASE_YES;
}

bool ClientView::isNoString(const string& str)
{
return toLowerCase(str) == LOWERCASE_NO;
}


/*
 * exception.cpp
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */
```

```
#include "exception.h"

#include <sstream>
#include <cstdlib>
#include <iostream>

#include <execinfo.h>
#include <cxxabi.h>

using std::ostringstream;


static const size_t MAX_STACK_DEPTH = 100;

string demangleStackString(const string& str)
{
// Find the parentheses and address offset surrounding the mangled name.
size_t begin = str.find('(');
size_t end   = str.find('+');

string ret;
if (begin != str.npos &&
    end != str.npos &&
    end > begin &&
    begin != str.size() - 1)
{
// Found our mangled name.
string mangledName = str.substr(begin + 1, end - begin - 1);
string fileName = str.substr(0, begin);

// Calls C++ ABI for demangling (quite hacky... but very nice!).
int status;
char* demangledName =
abi::__cxa_demangle(mangledName.c_str(), NULL, NULL, &status);

ret = demangledName ? fileName + ":" + demangledName
                    : fileName + ":" + mangledName + "()";

free(demangledName);
}
else {
// Didn't find the mangled name. Returns the whole line.
ret = str;
}

return ret;
}

string generateStackTraceMessage(const string& msj)
{
// Acquire the stack trace.
void* stackAddrs[MAX_STACK_DEPTH];
size_t stackDepth = backtrace(stackAddrs, MAX_STACK_DEPTH);
char** stackStrings = backtrace_symbols(stackAddrs, stackDepth);

// Probable out of memory condition. Doesn't matter =P
if (stackStrings == 0)
stackDepth = 0;

ostringstream stackTraceBuf;
// Starts from 2 because 1st and 2nd lines are generateStackTraceMessage()
// and Exception::Exception() respectively.
for (size_t i = 2; i < stackDepth; ++i)
{
string demangledString = demangleStackString(stackStrings[i]);
stackTraceBuf << "\tat " << demangledString << "\n";
}

// Free up the allocated memory.
free(stackStrings);
```

```
return msj + "\n" + stackTraceBuf.str();
}


Exception::Exception(const string& msj): _what(generateStackTraceMessage(msj))
{ }


Exception::Exception(const std::exception& e):
_what(generateStackTraceMessage(e.what()))
{ }

const char* Exception::what() const throw()
{
return _what.c_str();
}


/*
 * ipc_error.cpp
 *
 *  Created on: Apr 16, 2010
 *      Author: demian
 */

#include "ipc/ipc_error.h"

#include <cstring>

IpcError::IpcError(const string& msj, int errorCode) :
Exception(msj + " - Error: " + (errorCode ? strerror(errorCode) : "")),
_errorCode(errorCode)
{
}

int IpcError::getErrorCode()
{
return _errorCode;
}


/*
 * shared_memoty.cpp
 *
 *  Created on: Apr 9, 2010
 *      Author: demian
 */

#include "ipc/shared_memory.h"
#include "ipc/ipc_error.h"

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

#include <cstdio>

RawSharedMemory::RawSharedMemory(size_t size, const string& pathName, char id,
bool ownResources):
Resource(ownResources)
{
// Creates key.
key_t key = ftok(pathName.c_str(), id);
if (key == (key_t)-1)
throw IpcError("SharedMemory(): Could not create key", errno);

// Allocates a shared memory segment.
_shmId = shmget(key, size, 0644 | IPC_CREAT);
if (_shmId == -1)
throw IpcError("SharedMemory(): "
```

```
"Could not allocate shared memory", errno);

// Attaches the shared memory to some address.
_data = shmat(_shmId, NULL, 0);
if (_data == (void*)-1)
throw IpcError("SharedMemory(): "
"Could not attach shared memory", errno);
}

RawSharedMemory::~RawSharedMemory() throw ()
{
doDispose();
}

void RawSharedMemory::print(ostream& stream) const
{
stream << "raw shared memory (id = " << _shmId << ")";
}

void RawSharedMemory::doDispose() throw ()
{
int errorCode = shmdt(_data);
if (errorCode == -1)
// Throwing exceptions on destruction is not recommended.
perror("~SharedMemory(): Could not detach shared memory");

shmid_ds state;
errorCode = ::shmctl(_shmId, IPC_STAT, &state);
if (errorCode == -1)
perror("~SharedMemory() Could not get shared memory state");

if (state.shm_nattch == 0 && ownResources())
{
// TODO: qué pasa si justo acá el scheduler cambia a otro proceso y
// ese proceso hace un shmat de esta memoria?

// Destroys shared memory.
errorCode = shmctl(_shmId, IPC_RMID, NULL);
if (errorCode == -1)
perror("~SharedMemory(): Could not destroy shared memory");
}
}


/*
 * resource.cpp
 *
 *  Created on: Apr 23, 2010
 *      Author: demian
 */

#include "ipc/resource.h"
#include "ipc/resource_collector.h"

#include <iostream>

using std::cout;
using std::cerr;

Resource::Resource(bool ownResources) :
_ownResources(ownResources),
_disposed(false)
{
ResourceCollector::instance().registerResource(this);
}


Resource::~Resource() throw ()
{
ResourceCollector::instance().unregisterResource(this);
}
```

```
bool Resource::ownResources() const        { return _ownResources; }
void Resource::setOwnResources(bool value) { _ownResources = value; }

void Resource::dispose() throw ()
{
if (!_disposed)
{
doDispose();
_disposed = true;
}
else
cerr << "Resource::dispose(): resource already disposed";
}

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

#include "ipc/message_queue.h"
#include "ipc/ipc_error.h"

#include <sys/msg.h>
#include <errno.h>
#include <cstdio>
#include <stdlib.h>
#include <string.h>

RawMessageQueue::RawMessageQueue(const string& pathName, char id,
bool ownResource) :
Resource(ownResource)
{
// Creates key.
key_t key = ftok(pathName.c_str(), id);
if (key == (key_t) -1)
throw IpcError("RawMessageQueue(): Could not create key. path="
+ pathName, errno);

// Try to create/connect the message queue.
_queueId = msgget(key, 0644 | IPC_CREAT);
if (_queueId == -1)
{
throw IpcError("RawMessageQueue(): Could not create or connet"
" to the message queue", errno);
}
}

void RawMessageQueue::sendFixedSize(const void* buffer, size_t size, long mtype)
{
// Builds the package to send the message.
char *writeBuff = (char*) malloc(sizeof(mtype) + size);
if (!writeBuff)
throw IpcError("RawMessageQueue::send(): Could not build the package "
"to be send", errno);

*(long*) writeBuff = mtype;

memcpy(writeBuff + sizeof(mtype), buffer, size);

// Sends the message.
int errorCode = msgsnd(_queueId, writeBuff, size, 0);
free(writeBuff);

if (errorCode == -1)
throw IpcError("RawMessageQueue::send(): Could not write into the "
"queue", errno);
}
```

```
void RawMessageQueue::receiveFixedSize(void* buffer, size_t size, long mtype)
{
this->tryReceive(buffer, size, mtype);
}

size_t RawMessageQueue::tryReceive(void* buffer, size_t size, long mtype)
{
// Builds the package to receive the message.
char *readBuff = (char*) malloc(sizeof(long) + size);
if (!readBuff)
throw IpcError("RawMessageQueue::receive(): Could not build the "
"package to be received", errno);

// Receives the message.
ssize_t returnValue = msgrcv(_queueId, readBuff, size, mtype, 0);
// E2BIG: We don't receive everything that we were supposed to receive.
// Something wrong happened.
if (returnValue == -1 && errno != E2BIG)
{
free(readBuff);
throw IpcError("RawMessageQueue::receive(): Could not read from"
" the queue", errno);
}

memcpy(buffer, readBuff + sizeof(long), size);

free(readBuff);
return returnValue;
}

RawMessageQueue::~RawMessageQueue() throw ()
{
doDispose();
}

void RawMessageQueue::doDispose() throw ()
{
if (ownResources())
{
int errorCode = msgctl(_queueId, IPC_RMID, NULL);
if (errorCode == -1)
perror("~RawMessageQueue(): Could not destroy message queue");
}
}

void RawMessageQueue::print(ostream& stream) const
{
stream << "message queue";
}

void MessageQueue::sendByteArray(const ByteArray& message, long mtype)
{
RawMessageQueue::sendFixedSize(message.data(), message.size(), mtype);
}

const ByteArray MessageQueue::receiveByteArray(long mtype)
{
int bufferSize = QUEUE_INITAL_BUFFER_SIZE;
bool receiveOk = false;
ByteArray message;

while (!receiveOk)
{
char* readBuff = (char*) malloc(bufferSize);

ssize_t returnValue;

try
{
returnValue = tryReceive(readBuff, bufferSize, mtype);
```

```
} catch (...)
{
free(readBuff);
throw;
}

if (returnValue == -1)
{
bufferSize *= 2;
}
// Everything goes ok!
else
{
message = toByteArray(readBuff, returnValue);
receiveOk = true;
}

free(readBuff);
}

return message;
}

void MessageQueue::sendString(const std::string& message, long mtype)
{
/**
 * Sends the string.
 * The second parameter has a "+1" expression because  the size method
 * returns only the number of characters in the string, not including any
 * null-termination.
 */
RawMessageQueue::sendFixedSize(message.c_str(), message.size() + 1, mtype);
}

const std::string MessageQueue::receiveString(long mtype)
{
ByteArray data = this->receiveByteArray(mtype);
return getStringFromByteArray(data, 0, data.size());
}


/*
 * file_lock.cpp
 *
 *  Created on: 07/07/2010
 *      Author: nicolas
 */

#include "ipc/file_lock.h"
#include "ipc/ipc_error.h"

#include <stdio.h>
#include <unistd.h>
#include <errno.h>

FileLock::FileLock(int fd) : _fd(fd)
{
_lockInfo.l_type = F_RDLCK | F_WRLCK;
_lockInfo.l_whence = SEEK_SET;
_lockInfo.l_start = 0;
_lockInfo.l_len = 0;
_lockInfo.l_pid = getpid();

if (fcntl(_fd, F_SETLKW, &_lockInfo) == -1)
throw IpcError("FileLock(): Could not acquire the lock", errno);
}

FileLock::~FileLock()
{
_lockInfo.l_type = F_UNLCK;
if (fcntl(_fd, F_SETLKW, &_lockInfo) == -1)
```

```
perror("FileLock(): Could not release the lock");
}

#include "ipc/fifo.h"
#include "ipc/ipc_error.h"

#include <fcntl.h>
#include <cerrno>
#include <cstring>
#include <cstdio>
#include <iostream>

// Helper functions.

/**
 * Checks if a file is a fifo.
 * @param pathName A path of an existing file.
 * @return Whether the file is a fifo or not.
 */
bool isFifoFile(const string& pathName)
{
struct stat fileStat;
if (stat(pathName.c_str(), &fileStat) == -1)
throw IpcError("isFifoFile(): Could not get file info", errno);

return fileStat.st_mode & S_IFIFO;
}


Fifo::Fifo(const string& pathName, bool ownResource):
Resource(ownResource),
_pathName(pathName)
{
bool mknodEror = mknod(pathName.c_str(), S_IFIFO | 0666, 0) == -1;

if (mknodEror)
{
bool fileAlreadyExists = errno == EEXIST;
if (fileAlreadyExists)
{
if (!isFifoFile(pathName))
throw IpcError("Fifo(): File " + pathName + " already exists, "
               "but it's not a fifo");
}
else
throw IpcError("Fifo(): Could not create fifo file", errno);
}

}


Fifo::~Fifo() throw ()
{
doDispose();
}

void Fifo::print(ostream& stream) const
{
stream << "fifo (path " << getPathName() << ")";
}

void Fifo::doDispose() throw ()
{
if (ownResources())
{
bool unlinkError = unlink(_pathName.c_str()) == -1;

if (unlinkError && errno != ENOENT) // ENOENT = No such file.
perror("~Fifo(): Could not unlink the fifo's file");
}
}
```

```
FifoWriter::FifoWriter(const Fifo& fifo)
{
_fileDescriptor = open(fifo.getPathName().c_str(), O_WRONLY);
if (_fileDescriptor == -1)
throw IpcError("FifoWriter(): Could not open" + fifo.getPathName(),
               errno);
}


FifoWriter::~FifoWriter()
{
if (close(_fileDescriptor) == -1)
perror("~FifoWriter(): Could not close the fifo's file");
}


size_t FifoWriter::write(const void* buffer, size_t size)
{
int bytes = ::write(_fileDescriptor, buffer, size);
if (bytes == -1)
throw IpcError("FifoWriter::write(): Could not write to fifo", errno);

return bytes;
}


void FifoWriter::writeFixedSize(const void* buffer, size_t size)
{
size_t totalBytes = 0;
char* index = (char*) buffer;

while (totalBytes < size)
{
int bytes = write(index, size);

if (bytes == 0)
// Does not use errno.
throw IpcError("FifoWriter::writeFixedSize(): Could not write to "
               "fifo");

totalBytes += bytes;
index += bytes;
 }
}


FifoReader::FifoReader(const Fifo& fifo)
{
_fileDescriptor = open(fifo.getPathName().c_str(), O_RDONLY);
if (_fileDescriptor == -1)
throw IpcError("FifoReader(): Could not open" + fifo.getPathName(),
               errno);
}


FifoReader::~FifoReader()
{
if (close(_fileDescriptor) == -1)
perror("~FifoReader(): Could not close the fifo's file");
}


size_t FifoReader::read(void* buffer, size_t size)
{
int bytes = ::read(_fileDescriptor, buffer, size);
if (bytes == -1)
throw IpcError("FifoReader::read(): Could not read from fifo", errno);
```

```
return bytes;
}


void FifoReader::readFixedSize(void* buffer, size_t size)
{
size_t totalBytes = 0;
char* index = (char*) buffer;

while (totalBytes < size)
{
int bytes = read(index, size);

if (bytes == 0)
// Does not use errno.
throw IpcError("FifoReader::readFixedSize(): Unexpected EOF");

totalBytes += bytes;
index += bytes;
 }
}

/*
 * resource_collector.cpp
 *
 *  Created on: Apr 23, 2010
 *      Author: demian
 */

#include "ipc/resource_collector.h"
#include "ipc/resource.h"

#include <stdexcept>
#include <algorithm>
#include <iostream>
#include <cstdlib>



ResourceCollector::ResourceCollector()
{
}

ResourceCollector::~ResourceCollector()
{
disposeAllResources();
}

ResourceCollector& ResourceCollector::instance()
{
static ResourceCollector singleton;
return singleton;
}

void ResourceCollector::registerResource(Resource* res)
{
if (!alreadyRegistered(res))
_resources.push_back(res);
else
throw std::invalid_argument("ResourceCollector::registerResource(): "
"resource already registered");
}

void ResourceCollector::unregisterResource(Resource* res)
{
if (alreadyRegistered(res))
_resources.remove(res);
else
throw std::invalid_argument("ResourceCollector::unregisterResource(): "
"resource is not registered");
}
```

```
bool ResourceCollector::alreadyRegistered(Resource* res) const
{
ResourceList::const_iterator it =
std::find(_resources.begin(),_resources.end(), res);
return it != _resources.end();
}

void ResourceCollector::disposeAllResources()
{
while (!_resources.empty())
{
Resource* res = _resources.front();
res->dispose();
_resources.pop_front();
}
}

/*
 * semaphore.cpp
 *
 *  Created on: Apr 16, 2010
 *      Author: demian
 */

#include "ipc/semaphore_set.h"
#include "ipc/ipc_error.h"

#include <errno.h>
#include <sys/sem.h>

#include <ctime>
#include <cstdio>


static const int SEM_TIMEOUT = 5; // In seconds.

// Helper functions.

/**
 * Initializes a System V semaphore set, taking care of the race conditions
 * there might occur.
 *
 * If the semaphore set was already created by an other program, the initial
 * values vector is ignored.
 *
 * More-than-inspired by W. Richard Stevens' UNIX Network Programming 2nd
 * edition, volume 2, lockvsem.c, page 295.
*/
int initSemSet(const string& pathName, char id,
const SemaphoreSet::InitValues& initVals)
{

// Some helper functions declarations.
void initNewSemSet(int semId, const SemaphoreSet::InitValues& initVals);
int  waitForSemInitialization(key_t key, size_t nSems);

// Creates key.
key_t key = ftok(pathName.c_str(), id);
if (key == (key_t)-1)
throw IpcError("initSemSet(): Could not create key", errno);

const size_t nSems = initVals.size();

// Uses IPC_CREAT | IPC_EXCL in order to fail if the semaphore set already
// exists.
int semId = semget(key, nSems, IPC_CREAT | IPC_EXCL | 0666);

if (semId >= 0) // Great, we got it first.
{
initNewSemSet(semId, initVals);
```

```
}
else if (errno == EEXIST) // Someone got it first.
{
semId = waitForSemInitialization(key, nSems);
}
else
throw IpcError("initSemSet(): semget() failed.", errno);

return semId;
}


/** semun defined as man page. */
union semun
{
   int              val;    /* Vafirstlue for SETVAL */
   struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
   unsigned short  *array;  /* Array for GETALL, SETALL */
   struct seminfo  *__buf;  /* Buffer for IPC_INFO
   (Linux-specific) */
};


void initNewSemSet(int semId, const SemaphoreSet::InitValues& initVals)
{
const size_t nSems = initVals.size();

// Initializes all values to zero first (in Linux this is not needed, but,
// in general, it is).
union semun arg;
SemaphoreSet::InitValues zeros(nSems, 0);
arg.array = &zeros[0];

if (semctl(semId, 0, SETALL, arg) == -1)
{
int e = errno;
semctl(semId, 0, IPC_RMID); // Clean up.
throw IpcError("initNewSemSet(): semctl() failed", e);
}

// Now initializes values to the initial values using semop, wich "frees"
// other SemaphoreSets. This sets the sem_otime to something != 0, as needed
// in waitForSemInitialization().
for (size_t i = 0; i < nSems; ++i)
{
struct sembuf op;
// Moves the semaphore up initVals[i] times (if initVals[i] == 0,
// nothing is done, as the state of the i'th semaphore will be 0)
op.sem_op = initVals[i];
op.sem_num = i;
op.sem_flg = 0;

if (semop(semId, &op, 1) == -1)
{
int e = errno;
semctl(semId, 0, IPC_RMID); // Clean up.
throw IpcError("initNewSemSet(): semop() failed", e);
}
}
}


int waitForSemInitialization(key_t key, size_t nSems)
{
// Gets the semaphore set id without trying to create it.
int semId = semget(key, nSems, 0666);
if (semId == -1)
throw IpcError("waitForSemInitialization(): semget() failed.", errno);

// Waits until other process initializes semaphore set.
```

```
bool ready = false;
clock_t initClock = clock();
while (!ready)
{
struct semid_ds buf;
union semun arg;
arg.buf = &buf;
if (semctl(semId, nSems - 1, IPC_STAT, arg) == -1)
throw IpcError("waitForSemInitialization(): semctl() failed",
errno);

bool timeout = (clock() - initClock) > SEM_TIMEOUT * CLOCKS_PER_SEC;
if (timeout)
throw IpcError("waitForSemInitialization(): Timed out");

ready = arg.buf->sem_otime != 0;
}
return semId;
}


int semOperation(int semId, size_t semIndex, int op, int flags = 0)
{
struct sembuf operation;
operation.sem_num = semIndex;
operation.sem_op  = op;
operation.sem_flg = flags;

return semop(semId, &operation, 1);
}

// End helper functions.

SemaphoreSet::SemaphoreSet(const string& pathName, char id,
size_t nSems, int initVals[], bool ownResources) :
Resource(ownResources),
_semId(initSemSet(pathName, id, InitValues(initVals, initVals + nSems))),
_nSems(nSems)
{ }


SemaphoreSet::SemaphoreSet(const string& pathName, char id,
const InitValues& initVals, bool ownResources) :
Resource(ownResources),
_semId(initSemSet(pathName, id, initVals)),
_nSems(initVals.size())
{ }


SemaphoreSet::~SemaphoreSet() throw ()
{
doDispose();
}

void SemaphoreSet::wait(size_t semIndex)
{
validateIndex(semIndex);
if (semOperation(_semId, semIndex, -1) == -1)
throw IpcError("SemaphoreSet::wait(): semop() failed", errno);
}


void SemaphoreSet::signal(size_t semIndex)
{
validateIndex(semIndex);
if (semOperation(_semId, semIndex, 1) == -1)
throw IpcError("SemaphoreSet::signal(): semop() failed", errno);
}
```

```
bool SemaphoreSet::tryWait(size_t semIndex)
{
validateIndex(semIndex);
if (semOperation(_semId, semIndex, -1, IPC_NOWAIT) == -1)
{
if (errno == EAGAIN)
return false;
else
throw IpcError("SemaphoreSet::tryWait(): semop() failed", errno);
}
else
return true;
}


void SemaphoreSet::validateIndex(size_t semIndex)
{
if (semIndex >= _nSems)
throw IpcError("SemaphoreSet::validateIndex(): Index out of range");
}


SemaphoreProxy SemaphoreSet::getSemaphore(size_t semIndex)
{
validateIndex(semIndex);
return SemaphoreProxy(*this, semIndex);
}

void SemaphoreSet::print(ostream& stream) const
{
stream << "semaphore set (id = " << _semId << ", nsems = " << _nSems
       << ")";
}


void SemaphoreSet::doDispose() throw ()
{
if (ownResources() && semctl(_semId, 0, IPC_RMID) == -1)
perror("~SemaphoreSet(): Could not remove semaphore set");
}



/*
 * serializable.cpp
 *
 *  Created on: 25/05/2010
 *      Author: nicolas
 */

#include "core/serializable.h"

void Serializable::deserializeFromIndex(const ByteArray& bytes,
size_t startIndex, size_t size)
{
ByteArray::const_iterator from = bytes.begin() + startIndex;
ByteArray::const_iterator to = bytes.begin() + startIndex + size;
ByteArray subBytes = ByteArray(from, to);
this->deserialize(subBytes);
}

/*
 * byte_array.cpp
 *
 *  Created on: 23/05/2010
 *      Author: nicolas
 */

#include "core/byte_array.h"
```

```
#include <string.h>
#include "stdlib.h"

ByteArray toByteArray(const void* arrayData, size_t size)
{
ByteArray byteArray;
addToByteArray(byteArray, arrayData, size);
return byteArray;
}

void addToByteArray(ByteArray& byteArray, const void* arrayData, size_t size)
{
char* buffer = (char*) arrayData;
for (size_t i = 0; i < size; i++)
byteArray.push_back(buffer[i]);
}

bool getFromByteArray(const ByteArray& byteArray, size_t startIndex,
void* arrayData, size_t size)
{
if (byteArray.size() < size + startIndex - 1)
return false;

memcpy((char*) arrayData, byteArray.data() + startIndex, size);

return true;
}

void addStringToByteArray(ByteArray& byteArray, const std::string arrayData)
{
addToByteArray(byteArray, arrayData.c_str(), arrayData.size() + 1);
}

const std::string getStringFromByteArray(const ByteArray& byteArray,
size_t startIndex, size_t size)
{
return std::string(byteArray.begin() + startIndex, byteArray.begin()
+ startIndex + size);
}

const std::string byteArrayToString(const ByteArray& bytes)
{
return getStringFromByteArray(bytes, 0, bytes.size());
}

ByteArray stringToByteArray(const std::string& str)
{
ByteArray result;
addStringToByteArray(result, str);
return result;
}

ByteArrayWriter::ByteArrayWriter()
{
}

void ByteArrayWriter::writeString(const std::string& string)
{
addStringToByteArray(_bytes, string);
}

void ByteArrayWriter::writeByteArray(const ByteArray& bytes)
{
_bytes.insert(_bytes.end(), bytes.begin(), bytes.end());
}

ByteArray ByteArrayWriter::getByteArray()
{
return _bytes;
}
```

```
ByteArrayReader::ByteArrayReader(const ByteArray& bytes) :
_bytes(bytes)
{
_index = 0;
}

std::string ByteArrayReader::readString()
{
const char* data = &_bytes[0];
std::string string(data + _index);
_index += string.size() + 1;
return string;
}

ByteArray ByteArrayReader::readAll()
{
ByteArray bytes(_bytes.begin() + _index, _bytes.end());
_index += bytes.size();
return bytes;
}

/*
 * random.cpp
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */

#include "random.h"

#include <stdexcept>
#include <cstdlib>
#include <ctime>

// A little hackery...
class Randomizer
{
friend int randomInt(int n);

Randomizer()
{
srandom(time(NULL));
}
};

int randomInt(int n)
{
static Randomizer rand; // Created only once ;)
return random() % n;
}

int randomInt(int min, int max)
{
if (max < min)
throw std::invalid_argument("max must be greater than or equal to min");
return min + randomInt(max - min + 1);
}

/*
 * main.cpp
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */
/*
#include "tests.h"
#include "utils.h"
#include "random.h"
```

```
#include <iostream>

using std::cerr;

int main(int argc, char* argv[])
{
TestFunction tests[] = {
                         sharedMemoryTest,
                         fifoTest,
                         exceptionsTest,
                       };
size_t nTests = ARR_SIZE(tests);

size_t index = argc == 2 ? strToInt(argv[1])
                         : randomInt(nTests);

if (index >= nTests)
{
cerr << "Invalid index\n";
return 1;
}
else {
try {
return tests[index](argc, argv);
}
catch(...)
{
throw; // To force stack-unwinding.
}
}
return 0;
}
*/

/*
 * config.cpp
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#include "config.h"

Config::~Config()
{
}

Config::Config()
{
}

/*
 * logger.cpp
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#include "logger.h"
#include "ipc/ipc_error.h"
#include "ipc/file_lock.h"
#include "global_config.h"

#include <ctime>
#include <cstdio>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
```

```
static const string LOG_FILE_NAME_KEY = "log_file";

namespace
{

static const size_t BUFFER_SIZE = 256;

string getTimeString()
{
time_t rawTime = time(NULL);
struct tm* timeInfo = localtime(&rawTime);
char buffer[BUFFER_SIZE];

strftime(buffer, BUFFER_SIZE, "%c", timeInfo);

return string(buffer);
}

} // end namespace


Logger& Logger::instance()
{
static Logger instance;
return instance;
}

Logger::Logger() :
_logging(false)
{
string logFileName = GlobalConfig::get<string>(LOG_FILE_NAME_KEY);
_fd = open(logFileName.c_str(), O_RDWR | O_APPEND | O_CREAT, 0644);

if (_fd == -1)
throw IpcError("Could not open log file " + logFileName, errno);
}

Logger::~Logger()
{
if (close(_fd) == -1)
perror("~Logger(): Could not close log file");
}

void Logger::log(const string& message)
{
if (_logging)
{
//FileLock lock(_fd);
string logLine = getTimeString() + " - " + message + "\n";
if (write(_fd, logLine.c_str(), logLine.size()) == -1)
throw IpcError("Could not write log file ", errno);
}
}

void Logger::setLogging(bool value)
{
_logging = value;
}

/*
 * utils.cpp
 *
 *  Created on: Apr 12, 2010
 *      Author: demian
 */

#include "utils.h"

#include <sstream>
```

```
#include <algorithm>

int strToInt(const char* str) throw (std::invalid_argument)
{
int n;
std::istringstream iss(str);
iss >> n;
if (!iss)
throw std::invalid_argument("strToInt(): Bad format");
return n;
}

string trimLeft(const string& str)
{
size_t startPos = 0;
while (startPos < str.size() && isblank(str[startPos]))
startPos++;
return startPos == 0 ? str : str.substr(startPos);
}

string trimRight(const string& str)
{
size_t nChars = str.size();
while (nChars > 0 && isblank(str[nChars - 1]))
nChars--;
return nChars == str.size() ? str : str.substr(0, nChars);
}

string trim(const string& str)
{
return trimLeft(trimRight(str));
}

string toLowerCase(const string& str)
{
string lower = str;
for (size_t i = 0; i < str.size(); i++)
lower[i] = tolower(str[i]);
return lower;
}

/*
 * peer_table.cpp
 *
 *  Created on: May 11, 2010
 *      Author: demian
 */

#include "model/peer_table.h"
#include "model/model_error.h"

#include <algorithm>
#include <iostream>

std::ostream& operator << (std::ostream& os, const Peer& peer)
{
os << "Peer [name=" << peer.getName() << ", id=" << peer.getId() << "]";
return os;
}

std::ostream& operator << (std::ostream& os, const PeerTable& peers)
{
os << "Peer Table: Name - Adress\n";

vector<Peer>::const_iterator iterator = peers._peers.begin();
for (; iterator != peers._peers.end(); iterator++)
{
Peer peer = *iterator;
os << peer.getName() << " - " << peer.getId() << "\n";
}
return os;
```

```
}

void PeerTable::add(const Peer& peer)
{
if (containsName(peer.getName()))
throw ModelError("A peer with the same name already exists");
if (containsId(peer.getId()))
throw ModelError("A peer with the same id already exists");

_peers.push_back(peer);
}

void PeerTable::remove(const string& peerName)
{
PeerVector::iterator it = _peers.begin();

for (; it != _peers.end(); it++)
{
Peer peer = *it;
if (peer.getName() == peerName)
break;
}

if (it == _peers.end())
throw ModelError("Peer does not exist");
_peers.erase(it);
}

bool PeerTable::containsId(pid_t peerId) const
{
return bool(getById(peerId));
}

bool PeerTable::containsName(const string& peerName) const
{
return bool(getByName(peerName));
}

const Peer* PeerTable::getById(pid_t peerId) const
{
for (size_t i = 0; i < _peers.size(); i++)
if (_peers[i].getId() == peerId)
return &_peers[i];
return 0;
}

const Peer* PeerTable::getByName(const string& peerName) const
{
for (size_t i = 0; i < _peers.size(); i++)
if (_peers[i].getName() == peerName)
return &_peers[i];
return 0;
}

ByteArray PeerTable::serialize()
{
ByteArrayWriter writer;
// The first element in the byte array is the size of the peer table.
writer.write(_peers.size());
for (size_t i = 0; i < _peers.size(); i++)
{
Peer peer = _peers[i];
writer.writeString(peer.getName());
writer.write<pid_t>(peer.getId());
}

return writer.getByteArray();
}

void PeerTable::deserialize(const ByteArray& bytes)
```

```
{
_peers.clear();

ByteArrayReader reader(bytes);
int size = reader.read<int> ();

for (int i = 0; i < size; i++)
{
std::string name = reader.readString();
pid_t id = reader.read<pid_t>();

Peer peer(name, id);
_peers.push_back(peer);
}
}

/*
 * utils.cpp
 *
 *  Created on: 29/06/2010
 *      Author: nicolas
 */

#include "model/queue_utils.h"

#include <sstream>

using std::ostringstream;

string getClientQueueFileName(pid_t pid)
{
ostringstream oss;
oss << "queues/client" << pid << ".queue";
return oss.str();
}

string getServerQueueFileName()
{
return "queues/server.queue";
}

/*
 * global_config.cpp
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#include "global_config.h"
#include <iostream>

GlobalConfig::ConfigPtr GlobalConfig::_config;

void GlobalConfig::setConfig(ConfigPtr config)
{
std::cout << "setConfig! " << config.get() << "\n" << std::flush;

_config = config;
}

/*
 * arg_parse.cpp
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#include "common.h"
#include "logger.h"
#include "config_file.h"
```

```
#include "global_config.h"

#include <getopt.h>
#include <cstdlib>
#include <iostream>
#include <cstring>

using std::cout;
using std::cerr;

namespace
{

static const string CONFIG_FILE_NAME = "config.txt";

void showHelp(const string& commandName)
{
cout << "Usage: " << commandName << " [option]\n";
cout << "\n";
cout << "Options:\n";
cout << "    -h, --help       Show this help message and exit\n";
cout << "    -l, --enable-log Enables logging\n";
}

} // end namespace

void parseArguments(int argc, char* argv[])
{
string commandName = basename(argv[0]);

int c;
opterr = 0;
struct option longOptions[] =
{
{ "help", no_argument, 0, 'h' },
{ "enable-log", no_argument, 0, 'l' },
{ 0, 0, 0, 0 } };

while ((c = getopt_long(argc, argv, ":hl", longOptions, NULL)) != -1)
{
switch (c)
{
case 'h':
showHelp(commandName);
exit(EXIT_SUCCESS);
case 'l':
Logger::instance().setLogging(true);
break;
case '?':
cerr << "Invalid option: ";
if (optopt != 0)
cerr << char(optopt) << "\n";
else
cerr << argv[optind - 1] << "\n";
showHelp(commandName);
exit(EXIT_FAILURE);
default:
cerr << "Error parsing arguments\n";
exit(EXIT_FAILURE);
}
}
}


void loadConfigFile()
{
GlobalConfig::ConfigPtr configFile(new ConfigFile(CONFIG_FILE_NAME));
GlobalConfig::setConfig(configFile);
}

/*
```

```
 * config_file.cpp
 *
 *  Created on: Jul 12, 2010
 *      Author: demian
 */

#include "config_file.h"
#include "utils.h"

#include <fstream>

using std::ifstream;

namespace
{

void readConfigData(map<string, string>& data,
ifstream& configFile)
{
string line;

while (getline(configFile, line))
{
// Skip comments
if (trimLeft(line)[0] == '#')
continue;

size_t eqPos = line.find('=');
if (eqPos == line.npos)
continue;

std::string key = trim(line.substr(0, eqPos));
std::string value = trim(line.substr(eqPos + 1));

data[key] = value;
}
}

} // end namespace

ConfigFile::ConfigFile(const string& fileName)
{
std::ifstream configFile(fileName.c_str());

if (configFile.is_open())
readConfigData(_data, configFile);
else
throw Exception("Could not open config file " + fileName);
}

ConfigFile::~ConfigFile()
{
}

string ConfigFile::getString(const string& key) const
{
map<string, string>::const_iterator it = _data.find(key);
if (it == _data.end())
throw Exception("Invalid key: " + key);
else
return it->second;
}

/*
 * fifo_test.cpp
 *
 *  Created on: Apr 8, 2010
 *      Author: nicolas
 */

#include "ipc/fifo.h"
```

```
#include "person.h"

#include <unistd.h>
#include <sys/wait.h>

#include <iostream>

using std::string;
using std::cout;

int writerProcess(pid_t);
int readerProcess();

int main(int argc, char* argv[]) {

cout << "Fifo Test\n\n";

pid_t pid = fork();
return pid ? writerProcess(pid)
   : readerProcess();
return 0;
}

int writerProcess(pid_t childPid)
{
Fifo fifo("fifo.fif");

FifoWriter w_fifo(fifo);
Person person("Mario Bros",55);

/**
 * Mario Bros enters into the time-travel-pipe.
 */

cout << person.name() << " is traveling through the time-travel-pipe. He "
"is " << person.age() << " years old.\n";
sleep(4);

w_fifo.write(person);

waitpid(childPid, NULL, 0);

return 0;
}


int readerProcess()
{
Fifo fifo("fifo.fif");

FifoReader r_fifo(fifo);
Person person;

/**
 * Mario Bros arrives into the Bros's World.
 */

r_fifo.read(person);

/**
 * In the new world he becomeOlder ten times.
 */

for(int i=0; i<10; i++)
person.becomeOlder();

cout << "He arrives to the new world from the time-travel-pipe.\n";
cout << "The new age of " << person.name() << " is: " << person.age() << "\n";

return 0;
```

```
}


/*
 * byte_array_test.cpp
 *
 *  Created on: 26/06/2010
 *      Author: nicolas
 */

#include <iostream>
#include <string>

#include "core/byte_array.h"

void testInts();
void testStrings();
void testIntsAndStrings();

int main(int argc, char **argv)
{
testInts();
testStrings();
testIntsAndStrings();

return 0;
}

void testInts()
{

int int1 = 10;
int int2 = 15;
int int3 = 30;
int int4 = 0;

ByteArrayWriter writer;
writer.write<int>(int1);
writer.write<int>(int2);
writer.write<int>(int3);
writer.write<int>(int4);

ByteArray bytes = writer.getByteArray();

ByteArrayReader reader(bytes);

if (reader.read<int>() == int1 &&
reader.read<int>() == int2 &&
reader.read<int>() == int3 &&
reader.read<int>() == int4)
std::cout << "Green test" << std::endl;
else
std::cout << "Red test" << std::endl;
}

void testStrings()
{
std::string string1 = "how";
std::string string2 = "u";
std::string string3 = "doing";
std::string string4 = "?";

ByteArrayWriter writer;
writer.writeString(string1);
writer.writeString(string2);
writer.writeString(string3);
writer.writeString(string4);

ByteArray bytes = writer.getByteArray();

ByteArrayReader reader(bytes);
```

```
if (reader.readString() == string1 &&
reader.readString() == string2 &&
reader.readString() == string3 &&
reader.readString() == string4)
std::cout << "Green test" << std::endl;
else
std::cout << "Red test" << std::endl;
}

void testIntsAndStrings(){

std::string name = "Mario Bross";
int lifes = 3;
std::string pass = "Green pipe";

ByteArrayWriter writer;
writer.writeString(name);
writer.write(lifes);
writer.writeString(pass);

ByteArray bytes = writer.getByteArray();

ByteArrayReader reader(bytes);

if (reader.readString() == name &&
reader.read<int>() == lifes &&
reader.readString() == pass)
std::cout << "Green test" << std::endl;
else
std::cout << "Red test" << std::endl;



}

/*
 * semaphore-test.cpp
 *
 *  Created on: Apr 19, 2010
 *      Author: demian
 */

#include "ipc/shared_memory.h"
#include "ipc/semaphore_set.h"
#include "person.h"

#include <unistd.h>
#include <sys/wait.h>

#include <string>
#include <cstdlib>
#include <iostream>

using std::string;
using std::cout;

const char SEMAPHORE_ID  = 's';
const char SHARED_MEM_ID = 'm';

int writerProcess(pid_t childPid, const string& pathName);
int readerProcess(const string& pathName);

int main(int argc, char* argv[])
{
cout << "Semaphore Test\n\n";

string pathName = argv[0];

pid_t pid = fork();
return pid ? writerProcess(pid, pathName)
```

```
   : readerProcess(pathName);
}


int writerProcess(pid_t childPid, const string& pathName)
{
SharedMemory<Person> sharedPerson(pathName, SHARED_MEM_ID, true);
int initValue = 0;
SemaphoreSet semSet(pathName, SEMAPHORE_ID, 1, &initValue, true); // true =
// own resources.
SemaphoreProxy semaphore = semSet.getSemaphore(0);

Person& person = sharedPerson.get(); // A reference to the shared person.
person = Person("John Locke", 48);

cout << "Writer: created person with name = " << person.name()
     << " and age = " << person.age() << "\n";

cout << "Writer: now I'll wait 2 secs and the reader won't do anything.\n";
sleep(2);

semaphore.signal();

// Waits for reader.
waitpid(childPid, NULL, 0);

cout << "Writer: I'm finished\n";
return 0;
}


int readerProcess(const string& pathName)
{
SharedMemory<Person> sharedPerson(pathName, SHARED_MEM_ID, true);
Person& person = sharedPerson.get();

int initValue = 0;
SemaphoreSet semSet(pathName, SEMAPHORE_ID, 1, &initValue, false); // false
// = not own resources.
SemaphoreProxy semaphore = semSet.getSemaphore(0);

cout << "Reader: I want to read!\n";

semaphore.wait();

cout << "Reader: read shared person with name = " << person.name()
     << " and age = " << person.age() << "\n";

cout << "Reader: I'm finished\n";
return 0;
}

/*
 * producer_consumer_test.cpp
 *
 *  Created on: Apr 19, 2010
 *      Author: demian
 */

#include "ipc/mutex_set.h"
#include "ipc/semaphore_set.h"
#include "ipc/shared_memory.h"
#include "random.h"
#include "exception.h"

#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>

#include <string>
#include <cstdlib>
```

```
#include <cstring>
#include <iostream>
#include <stdexcept>
#include <sstream>

using std::string;
using std::cout;
using std::ostringstream;

const size_t MAX_BUFFER_SIZE = 10;

const char SEMAPHORE_ID  = 'G';
const char SHARED_MEM_ID = 'o';
const char MUTEX_ID      = 'd';

const size_t ITEMS_READY_IND = 0;
const size_t EMPTY_PLACES_IND = 1;
const size_t MUTEX_IND = 0;


struct Item
{
static const size_t DESC_SIZE = 256;
char description[DESC_SIZE];
};

struct Buffer
{
Buffer(): _size(0) { }

void push(Item& item)
{
if (_size == MAX_BUFFER_SIZE)
throw std::logic_error("Buffer::push(): buffer full");
_items[_size++] = item;
}

Item pop()
{
if (_size == 0)
throw std::logic_error("Buffer::pop(): buffer full");
return _items[--_size];
}

private:
Item _items[MAX_BUFFER_SIZE];
size_t  _size;
};


void producerProcess(const string& pathName, int producerNo);
void consumerProcess(const string& pathName, int consumerNo);
void initializeSharedReources(const string& pathName);
void finalizeSharedReources(const string& pathName);
vector<pid_t> launchProcesses(const string& pathName, size_t nProducers,
size_t nConsumers);
void waitForChildren(vector<pid_t>& childPids);
void initializeParentSignalHandlers();
void initializeChildSignalHandlers();

int main(int argc, char* argv[])
{
initializeParentSignalHandlers();

cout << "Semaphore Test\n\n";

string pathName = argv[0];

size_t nProducers, nConsumers;
switch(argc)
```

```
{
case 3:
nProducers = strToInt(argv[1]);
nConsumers = strToInt(argv[2]);
break;
case 1 :
nProducers = randomInt(1, 10);
nConsumers = randomInt(1, 10);
break;
default :
cout << "Usage: " << argv[0] << " <n-producers> <n-consumers>\n";
return 1;
}

cout << "Using " << nProducers << " producers and "
     << nConsumers << " consumers\n"
     << "Parent pid: " << getpid() << "\n"
     << "Starting in 5 seconds (^C to start now and ^C to finish once it "
     << "started)\n";

sleep(5);

try {
initializeSharedReources(pathName);

vector<pid_t> childPids =
launchProcesses(pathName, nProducers, nConsumers);
waitForChildren(childPids);
finalizeSharedReources(pathName);
return 0;
}
catch(...)
{
throw;
}
}


void parentInterruptionHandler(int signum)
{
cout << "Parent process (pid " << getpid() << ") received signal "
     << signum << "! I will keep on waiting :)\n";
}


void initializeParentSignalHandlers()
{
struct sigaction sa;
sa.sa_handler = parentInterruptionHandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART; // waitpid() restarts.

sigaction(SIGINT, &sa, 0);
sigaction(SIGTERM, &sa, 0);
}


void childInterruptionHandler(int signum)
{
cout << "Child process (pid " << getpid() << ") received signal " << signum
 << "! Exiting now...\n";
// The resources will be collected by RecourceCollector =)
exit(EXIT_SUCCESS);
}


void initializeChildSignalHandlers()
{
struct sigaction sa;
// Child terminates it's execution on interruptions.
```

```
sa.sa_handler = childInterruptionHandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;

sigaction(SIGINT, &sa, 0);
sigaction(SIGTERM, &sa, 0);
}


SemaphoreSet::InitValues getSemaphoreInitValues()
{
SemaphoreSet::InitValues values(2);
values[ITEMS_READY_IND] = 0; // No items ready.
values[EMPTY_PLACES_IND] = MAX_BUFFER_SIZE;
return values;
}


struct ProducerConsumerResources
{
SharedMemory<Buffer> buffer;
SemaphoreSet semaphores;
MutexSet mutexes;


ProducerConsumerResources(const string& pathName, bool ownResources):
buffer(pathName, SHARED_MEM_ID, ownResources),
semaphores(pathName, SEMAPHORE_ID, getSemaphoreInitValues(),
ownResources),
mutexes(pathName, MUTEX_ID, 1, ownResources)
{ }

void setOwnResources(bool value)
{
buffer.setOwnResources(value);
semaphores.setOwnResources(value);
mutexes.setOwnResources(value);
}
};


void initializeSharedReources(const string& pathName)
{
// Creates shared memory, semaphores and mutex.
ProducerConsumerResources res(pathName, true);
res.buffer.get() = Buffer(); // The buffer gets constructed with 0 items.

// If a resource could not be created and raised an exception, the other
// resources already created would have been freed.
// But now change the ownership:
res.setOwnResources(false);

// Now when the objects destroy, they will not deallocate the resources.
}


void finalizeSharedReources(const string& pathName)
{
ProducerConsumerResources res(pathName, true);
}


vector<pid_t> launchProcesses(const string& pathName, size_t nProducers,
size_t nConsumers)
{
vector<pid_t> childrenPids;

for (size_t i = 0, n = nProducers + nConsumers; i < n; ++i)
{
pid_t pid = fork();
```

```
switch (pid)
{
case -1 :
throw Exception(string("fork():") + strerror(errno));
case 0  :
// Child.
initializeChildSignalHandlers();
if (i < nProducers) producerProcess(pathName, i);
else consumerProcess(pathName, i - nProducers);
// Children never reaches here, by just to be sure they end and
// don't continue with the loop...
throw
Exception("Child process continued unexpectedly");
default :
// Parent.
childrenPids.push_back(pid);
break;
}
}
// Only parent reaches here.
return childrenPids;
}


void waitForChildren(vector<pid_t>& childPids)
{
for (size_t i = 0; i < childPids.size(); ++i)
{
if (waitpid(childPids[i], NULL, 0) == -1)
throw Exception(string("waitpid(): ") + strerror(errno));
}
}

Item produce(int producerNo);
void consume(Item& item, int consumerNo);

void producerProcess(const string& pathName, int producerNo)
{
cout << "Producer " << producerNo << " started in process " << getpid()
     << "\n";

// Resource initialization.
ProducerConsumerResources res(pathName, false);

Buffer& buffer             = res.buffer.get();
SemaphoreProxy itemsReady  = res.semaphores.getSemaphore(ITEMS_READY_IND);
SemaphoreProxy emptyPlaces = res.semaphores.getSemaphore(EMPTY_PLACES_IND);
MutexProxy mutex           = res.mutexes.getMutex(0);

while (true)
{
Item item = produce(producerNo);
emptyPlaces.wait();

// Critical section.
mutex.lock();
buffer.push(item);
mutex.unlock();

itemsReady.signal();
}
}


void consumerProcess(const string& pathName, int consumerNo)
{
cout << "Consumer " << consumerNo << " started in process " << getpid()
     << "\n";

// Resource initialization.
```

```
ProducerConsumerResources res(pathName, false);

Buffer& buffer             = res.buffer.get();
SemaphoreProxy itemsReady  = res.semaphores.getSemaphore(ITEMS_READY_IND);
SemaphoreProxy emptyPlaces = res.semaphores.getSemaphore(EMPTY_PLACES_IND);
MutexProxy mutex           = res.mutexes.getMutex(0);

while (true)
{
itemsReady.wait();

// Critical section.
mutex.lock();
Item item = buffer.pop();
mutex.unlock();

emptyPlaces.signal();
consume(item, consumerNo);
}
}

Item produce(int producerNo)
{
static const char* types[] = { "Alfajores", "Bananas", "Melones",
"Choripanes", "Empanadas", "Pizzas", "Chocolates" };
static const int nTypes = ARR_SIZE(types);

int randNo = randomInt(1, 100);
const char* randType = types[randomInt(nTypes)];

ostringstream description;
description << randNo << " " << randType;
Item item;
strncpy(item.description, description.str().c_str(),
Item::DESC_SIZE);

cout << "Producer " << producerNo << " made " << item.description << "\n";
cout << std::flush;
return item;
}


void consume(Item& item, int consumerNo)
{
cout << "Consumer " << consumerNo << " uses " << item.description << "\n";
}

/*
 * exceptions_test.cpp
 *
 *  Created on: Apr 13, 2010
 *      Author: demian
 */

#include "exception.h"
#include "utils.h"

#include <iostream>
#include <vector>

using std::cout;
using std::cerr;

void test1();
void test2();
void test3();
void test4();

int main(int argc, char* argv[])
{
cout << "Exceptions Test\n\n";
```

```
void ( *tests[] )() = { test1, test2, test3, test4 };
size_t nTests = ARR_SIZE(tests);

for (size_t i = 0; i < nTests; ++i)
{
try {
tests[i]();
}
catch (std::exception& e)
{
cout << e.what() << "\n\n";
}
}

return 0;
}

int divide(int n, int d)
{
if (d == 0)
throw Exception("Division by zero");
else
return n / d;
}

void recursiveFunc(int n)
{
if (n > 0)
recursiveFunc(n -1);
else
divide(5, n);
}

string thirdFunction(const string& foobar = "default parameter")
{
recursiveFunc(3);
return foobar;
}

float secondFunc(int foo, double bar)
{
thirdFunction("lala");
return foo / bar;
}

void firstFunc()
{
secondFunc(45, -7.3);
}


void test1()
{
cout << "Test 1: Normal functions, long stack\n";
firstFunc();
}

class Explosion
{
public:
Explosion(int a, float lol) { method1(lol, a); }

private:
float method1(float m, int n) { return method2(n); }
float method2(int what, char the = 't', string fuck = "foo")
{ return explode(fuck); }
float explode(string lol)
{
throw Exception("Boooooooom!");
```

```
}

};

void test2()
{
cout << "Test 2: Class methods\n";
Explosion e(1, 2);
}


template <typename First, typename Second>
class SuperClass
{
public:
SuperClass(First f, Second s) { }

void explode() { method1(); }

private:
void method1() { method2(3); }
float method2(int a) { templateMethod<std::vector<int> >(); return 3; }
template <typename T>
T templateMethod() { doExplode();  return T(); }
void doExplode()
{
throw Exception("Something went very wrong");
}

};

void test3()
{
cout << "Test 3: Parameterized class\n";
SuperClass<double, std::string> superObject(4, "lalala");
superObject.explode();
}

static void hiddenExplosion()
{
throw Exception("ERROR: This will be hard to debug");
}

static void hiddenRecursiveFunc(int n)
{
if (n) hiddenRecursiveFunc(n - 1);
else hiddenExplosion();
}

static void hiddenFunc()
{
hiddenRecursiveFunc(3);
}

void test4()
{
cout << "Test 4: Static functions\n";
hiddenFunc();
}



/*
 * message_queue_test.cpp
 *
 *  Created on: 16/04/2010
 *      Author: nicolas
 */

#include <iostream>
#include <string>
```

```
#include <sys/types.h>
#include <sys/wait.h>

#include "ipc/message_queue.h"

using std::string;

const char ID = 'i';

int test1(int argc, char **argv);
int test2(int argc, char **argv);

int parent(pid_t firSonPid, pid_t secSonPid, const string& pathName);
int child1(const string& pathName);
int child2(pid_t brothersPid, const string& pathName);

int main(int argc, char **argv)
{
std::cout << "Message Queue Testing";

// std::cout << "\n\nMessage Queue Test 1 begins\n\n";
// test1(argc,argv);

std::cout << "\n\nMessage Queue Test 2 begins\n\n";
test2(argc, argv);
}

int test1(int argc, char **argv)
{
string pathName = argv[0];
MessageQueue queue(pathName, ID);

std::string m1("This is our first message!");
std::cout << "We send a message: " << m1 << "\n";
queue.sendString(m1, getpid());

sleep(2);

std::string m2 = queue.receiveString(getpid());
std::cout << "We receive a message: '" << m2 << "'\n";

return 0;
}

int test2(int argc, char **argv)
{
string pathName = argv[0];

pid_t pid = fork();

if (pid == 0)
{
sleep(2);
/**
 * First son's code.
 */
return child1(pathName);
}
else
{
/**
 * Father's code.
 */
pid_t brothersPid = pid;
pid = fork();

if (pid == 0)
{
sleep(2);
/**
```

```
 * Second son's code.
 */
return child2(brothersPid, pathName);
}
else
{
/**
 * Father's code.
 */
return parent(brothersPid, pid, pathName);
}
}
}

int parent(pid_t firSonPid, pid_t secSonPid, const string& pathName)
{
/*
 * std::cout << "I'm the father. My process id is: " << getpid() << "\n";
 * std::cout << "My first sun process id is: " << firSonPid << "\n";
 * std::cout << "My second sun process id is: " << secSonPid << "\n";
 */

MessageQueue queue(pathName, ID);

try
{
/**
 * Father sends a message saying hi to his first son.
 */
std::string m_son1("Hi first son!");
std::cout << "Father sends a message to his first son: " << m_son1
<< "\n";
queue.sendString(m_son1, firSonPid);

/**
 * Father sends a message saying hi to his second son.
 */
std::string m_son2("Hi second son!");
std::cout << "Father sends a message to his second son: " << m_son2
<< "\n";
queue.sendString(m_son2, secSonPid);

/**
 * We wait the sons.
 */
waitpid(firSonPid, NULL, 0);
waitpid(secSonPid, NULL, 0);
} catch (...)
{
}

return 0;
}

int child1(const string& pathName)
{
/*
 * std::cout << "I'm the first son. My process id is: " << getpid() << "\n";
 * std::cout << "My father process id is: " << getppid() << "\n";
 */

MessageQueue queue(pathName, ID, false);

try
{
/**
 * Receives a message from his father.
 */
std::string m_father = queue.receiveString(getpid());
std::cout << "Son1 received a message: " << m_father << "\n";
```

```
/**
 * Receives a message from his brother.
 */
std::string m_brother = queue.receiveString(getpid());
std::cout << "Son1 received a message: " << m_brother << "\n";
} catch (...)
{
}

return 0;
}

int child2(pid_t brothersPid, const string& pathName)
{
/*
 * std::cout << "I'm the second son. My process id is: " << getpid() << "\n";
 * std::cout << "My father process id is: " << getppid() << "\n";
 */

MessageQueue queue(pathName, ID, false);

try
{
/**
 * Receives a message from his father.
 */
std::string m_father = queue.receiveString(getpid());
std::cout << "Son2 receives a message: " << m_father << "\n";

/**
 * Son2 sends a message to his brother.
 */
std::string m_brother("Hi brothe!");
std::cout << "Son2 sends a message to his brother: " << m_brother
<< "\n";
queue.sendString(m_brother, long(brothersPid));
} catch (...)
{
}

return 0;
}


/*
 * shared_memory_test.cpp
 *
 *  Created on: Apr 8, 2010
 *      Author: demian
 */

#include "ipc/shared_memory.h"
#include "person.h"

#include <unistd.h>
#include <sys/wait.h>

#include <string>
#include <cstdlib>
#include <iostream>

using std::string;
using std::cout;

const char INT_ID = 'i';
const char PERSON_ID = 'p';

int writerProcess(pid_t childPid, const string& pathName);
int readerProcess(const string& pathName);
```

```
int main(int argc, char* argv[])
{
cout << "Shared Memory Test\n\n";

string pathName = argv[0];

pid_t pid = fork();
return pid ? writerProcess(pid, pathName)
   : readerProcess(pathName);
}


int writerProcess(pid_t childPid, const string& pathName)
{
SharedMemory<int>    sharedInt   (pathName, INT_ID, true);
SharedMemory<Person> sharedPerson(pathName, PERSON_ID, true);

sharedInt.get() = 481516;
Person& person = sharedPerson.get(); // A reference to the shared person.
person = Person("Jesus", 2010);

cout << "Writer: created shared int = " << sharedInt.get() << "\n";
cout << "Writer: created person with name = " << person.name()
     << " and age = " << person.age() << "\n";

// "Waits" for reader to read first values.
sleep(4);

person.becomeOlder();
cout << "Writer: Happy birthday " << person.name() << "!\n";

// Waits for reader.
waitpid(childPid, NULL, 0);

cout << "Writer: I'm finished\n";
return 0;
}


int readerProcess(const string& pathName)
{
SharedMemory<int>    sharedInt   (pathName, INT_ID, true);
SharedMemory<Person> sharedPerson(pathName, PERSON_ID, true);
Person& person = sharedPerson.get();

// "Waits" for writer to write first values.
sleep(2);

cout << "Reader: read shared int = " << sharedInt.get() << "\n";
cout << "Reader: read shared person with name = " << person.name()
     << " and age = " << person.age() << "\n";

// "Waits" for second value change.
sleep(6);

cout << "Reader: now " << person.name() << " is " << person.age() << "!\n";

cout << "Reader: I'm finished\n";
return 0;
}

/*
 * file_lock_test.cpp
 *
 *  Created on: 07/07/2010
 *      Author: nicolas
 */

#include "exception.h"
#include "ipc/file_lock.h"
```

```
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>
#include <string>
#include <errno.h>

using std::string;
using std::cout;

void destroyFile(string pathname);

int test1();
int child(int fd);
int father(int fd);

int main(int argc, char **argv)
{
return test1();
}

int test1()
{
string pathname = "filelock.txt";

int fd = open(pathname.c_str(), O_RDWR | O_CREAT);
if (fd == -1)
throw Exception("Error in open");

pid_t pid = fork();
switch (pid)
{
case -1:
throw Exception("Error in fork");
return 1;
case 0:
// Child code.
return child(fd);
default:
// Parent code.
father(fd);
}

wait(0);

close(fd);
destroyFile(pathname);
return 0;
}

int child(int fd)
{
// Child acquires the lock of the file.
{
FileLock lock(fd);
cout << "I'm the child and I've locked the file!\n"
"I'll wait 7 seconds.\n";
sleep(7);
}
cout << "I'm the child and I've released the lock of the file!\n";

return 0;
}

int father(int fd)
{
{
sleep(2);
```

```
cout << "I'm the father and I want to read the file!\n";
FileLock lock(fd);
cout << "I'm the father and I've locked the file!\n";
}
cout << "I'm the father and I've released the lock of the file!\n";

return 0;
}

void destroyFile(string pathname)
{
bool unlinkError = unlink(pathname.c_str()) == -1;

if (unlinkError && errno != ENOENT) // ENOENT = No such file.
throw Exception("Could not destroy file " + pathname);
}
```

5. Conclusiones