jun 25, 13 13:44 **server_sincronizador.h** Page 1/1

```
1  //
2  //  server_sincronizador.h
3  //  CLASE SINCRONIZADOR
4  //
5
6
7  #ifndef SINCRONIZADOR_H
8  #define SINCRONIZADOR_H
9
10
11 #include "common_logger.h"
12 #include "common_thread.h"
13 #include "server_emisor.h"
14 #include "server_receptor.h"
15 #include "common_manejador_de_archivos.h"
16
17
18
19
20
21 /* ******************************************************************************
22  * DECLARACIÓN DE LA CLASE
23  * ******************************************************************************/
24
25
26 class Sincronizador : public Thread {
27 private:
28
29   Receptor *receptor;              // Receptor
30   Emisor *emisor;                 // Emisor
31   ManejadorDeArchivos *manejadorDeArchivos;   // Manejador
32   Logger *logger;                 // Logger de eventos
33
34 public:
35
36   // Constructor
37   Sincronizador(Receptor *receptor, Emisor *emisor,
38     ManejadorDeArchivos *manejadorDeArchivos, Logger *logger);
39
40   // Destructor
41   ~Sincronizador();
42
43   // Inicia el sincronizador.
44   void iniciar();
45
46   // Detiene al sincronizador.
47   void detener();
48
49   // Define tareas a ejecutar en el hilo.
50   // Toma los mensajes que van llegando, los procesa y responde a clientes.
51   virtual void run();
52 };
53
54 #endif
```

jun 25, 13 13:44 **server_sincronizador.cpp** Page 1/6

```
1  //
2  //  server_sincronizador.h
3  //  CLASE SINCRONIZADOR
4  //
5
6
7  #include <sstream>
8  #include <utility>
9  #include "common_protocolo.h"
10 #include "common_parser.h"
11 #include "common_convertir.h"
12 #include "server_sincronizador.h"
13
14
15
16
17
18 /* ******************************************************************************
19  * DEFINICIÓN DE LA CLASE
20  * ******************************************************************************/
21
22
23 // Constructor
24 Sincronizador::Sincronizador(Receptor *receptor, Emisor *emisor,
25   ManejadorDeArchivos *manejadorDeArchivos, Logger *logger) :
26   receptor(receptor), emisor(emisor),
27   manejadorDeArchivos(manejadorDeArchivos), logger(logger) { }
28
29
30 // Destructor
31 Sincronizador::~Sincronizador() { }
32
33
34 // Inicia el sincronizador.
35 void Sincronizador::iniciar() {
36   this→start();
37 }
38
39
40 // Detiene al sincronizador.
41 void Sincronizador::detener() {
42   // Detenemos thread
43   this→stop();
44
45   // Insertamos mensaje fantasma para poder destrabar la recepción.
46   this→receptor→ingresarMensajeDeEntrada(0, "");
47 }
48
49
50 // Define tareas a ejecutar en el hilo.
51 // Toma los mensajes que van llegando, los procesa y responde a clientes.
52 void Sincronizador::run() {
53   // Procesamos mensajes entrantes
54   while (this→isActive()) {
55     // Solicitamos un mensaje de entrada al receptor
56     std::pair < int, std::string > mensaje;
57     mensaje = this→receptor→obtenerMensajeDeEntrada();
58
59     if(¬this→isActive()) break;
60
61     // Tomamos instrucción y sus argumentos
62     std::string instruccion, args;
63     Parser::parserInstruccion(mensaje.second, instruccion, args);
64
65
66     // Caso en que el cliente solicita la lista de archivos del servidor
```

```
67      if (instruccion ≡ C_GET_FILES_LIST) {
68        // Mensaje de log
69        std::string e = "SINCRONIZADOR: Solicitud de lista de archivos.";
70        this→logger→emitirLog(e);
71
72        // Se crea el mensaje de respuesta
73        std::string respuesta;
74        respuesta.append(S_FILES_LIST);
75        respuesta.append(" ");
76
77        // Pide la lista de archivos que tiene el server
78        Lista<std::string>* lista = new Lista<std::string>;
79        this→manejadorDeArchivos→obtenerArchivosDeDirectorio(lista);
80
81        // Insertamos como cabecera de los argumentos la cantidad de
82        // archivos
83        int cantArchivos = lista→tamanio();
84        respuesta.append(Convertir::itos(cantArchivos));
85        if(cantArchivos > 0) respuesta.append(COMMON_DELIMITER);
86
87        // Se guarda la lista en un string
88        while (¬lista→estaVacia()) {
89          std::string nombreArchivo = lista→verPrimero();
90          std::string hashArchivo;
91
92          int cantBloques = this→manejadorDeArchivos→obtenerHash(
93            nombreArchivo, hashArchivo);
94
95          respuesta.append(nombreArchivo);
96          respuesta.append(COMMON_DELIMITER);
97          respuesta.append(hashArchivo);
98          respuesta.append(COMMON_DELIMITER);
99          respuesta.append(Convertir::itos(cantBloques));
100
101         // Eliminamos de la lista
102         lista→eliminarPrimero();
103
104         // Separamos del próximo archivo que se liste
105         if(¬lista→estaVacia())
106           respuesta.append(COMMON_DELIMITER);
107       }
108
109       delete(lista);
110
111       // Se envia la respuesta al cliente
112       this→emisor→ingresarMensajeDeSalida(mensaje.first, respuesta, 0);
113
114       // Mensaje de log
115       std::string ee = "SINCRONIZADOR: Lista de archivos enviada.";
116       this→logger→emitirLog(ee);
117     }
118     // Caso en que un cliente solicita bloques de un archivo
119     else if(instruccion ≡ C_FILE_PARTS_REQUEST) {
120       // Mensaje de log
121       std::string e = "SINCRONIZADOR: Solicitud de partes de archivo.";
122       this→logger→emitirLog(e);
123
124       // Parseamos argumentos
125       Lista< std::string > listaArgumentos;
126       Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
127
128       // Tomamos nombre de archivo
129       std::string nombreArchivo = listaArgumentos.verPrimero();
130       listaArgumentos.eliminarPrimero();
131
132       std::string respuesta;
```



```
133
134       // Caso en que no existe el archivo en el servidor
135       if(¬this→manejadorDeArchivos→existeArchivo(nombreArchivo)) {
136         respuesta.append(S_NO_SUCH_FILE);
137         respuesta.append(" ");
138         respuesta.append(nombreArchivo);
139       }
140       // Caso en que existe el archivo en el servidor
141       else {
142         // Armamos el mensaje de respuesta
143         respuesta.append(COMMON_FILE_PARTS);
144         respuesta.append(" ");
145         respuesta.append(nombreArchivo);
146         respuesta.append(COMMON_DELIMITER);
147         respuesta.append(Convertir::uitos(
148           this→manejadorDeArchivos→obtenerCantBytes(
149             nombreArchivo)));
150
151         // Iteramos sobre los bloques solicitados
152         while(¬listaArgumentos.estaVacia()) {
153           // Tomamos un número de bloque
154           std::string sNumBloque = listaArgumentos.verPrimero();
155           listaArgumentos.eliminarPrimero();
156           int numBloque = Convertir::stoi(sNumBloque);
157
158           // Insertamos número de bloque y su contenido
159           respuesta.append(COMMON_DELIMITER);
160           respuesta.append(sNumBloque);
161           respuesta.append(COMMON_DELIMITER);
162           respuesta.append(
163             this→manejadorDeArchivos→obtenerContenido(
164             nombreArchivo, numBloque));
165         }
166       }
167
168       // Se envia la respuesta al cliente
169       this→emisor→ingresarMensajeDeSalida(mensaje.first, respuesta, 0);
170
171       // Mensaje de log
172       std::string ee = "SINCRONIZADOR: Partes de archivo enviadas.";
173       this→logger→emitirLog(ee);
174     }
175     // Caso en que un cliente solicita un archivo
176     else if(instruccion ≡ C_FILE_REQUEST) {
177       // Mensaje de log
178       std::string e = "SINCRONIZADOR: Solicitud de archivo.";
179       this→logger→emitirLog(e);
180
181       // Armamos mensaje de respuesta
182       std::string respuesta;
183
184       // Caso en que no existe el archivo en el servidor
185       if(¬this→manejadorDeArchivos→existeArchivo(args)) {
186         respuesta.append(S_NO_SUCH_FILE);
187         respuesta.append(" ");
188         respuesta.append(args);
189       }
190       // Caso en que existe el archivo en el servidor
191       else {
192         // Armamos mensaje con contenido del archivo
193         respuesta.append(COMMON_SEND_FILE);
194         respuesta.append(" ");
195         respuesta.append(args);
196         respuesta.append(COMMON_DELIMITER);
197         respuesta.append(this→manejadorDeArchivos→obtenerContenido(
198           args, 0));
```

```
199         }
200
201         // Enviamos mensaje al cliente que realizó solicitud
202         this→emisor→ingresarMensajeDeSalida(mensaje.first, respuesta, 0);
203
204         // Mensaje de log
205         std::string ee = "SINCRONIZADOR: Archivo enviado.";
206         this→logger→emitirLog(ee);
207       }
208       else if (instruccion ≡ COMMON_SEND_FILE) {
209         // Mensaje de log
210         std::string e = "SINCRONIZADOR: Recepción de archivo nuevo.";
211         this→logger→emitirLog(e);
212
213         // Parseamos argumentos
214         Lista< std::string > listaArgumentos;
215         Parser::dividirCadena(args, &listaArgumentos,
216           COMMON_DELIMITER[0]);
217
218         // Agregamos el archivo en el servidor
219         this→manejadorDeArchivos→agregarArchivo(listaArgumentos[0],
220           listaArgumentos[1]);
221
222         // Enviamos notificación a clientes de que se agregó archivo
223         std::string respuesta;
224         respuesta.append(S_NEW_FILE);
225         respuesta.append(" ");
226         respuesta.append(listaArgumentos[0]);
227         respuesta.append(COMMON_DELIMITER);
228         respuesta.append(listaArgumentos[2]);
229
230         // Se envia la notificación de nuevo archivo a los clientes
231         this→emisor→ingresarMensajeDeSalida(0, respuesta, mensaje.first);
232
233         // Mensaje de log
234         std::string ee = "SINCRONIZADOR: Se realizó notificación a ";
235         ee += "clientes de la existencia de un nuevo archivo.";
236         this→logger→emitirLog(ee);
237       }
238       // Caso en que se recibe la notificación de la modificación de archivo
239       else if (instruccion ≡ C_MODIFY_FILE) {
240         // Mensaje de log
241         std::string e = "SINCRONIZADOR: Recepción de modificaciones ";
242         e += "en archivo.";
243         this→logger→emitirLog(e);
244
245         // Parseamos argumentos
246         Lista< std::string > listaArgumentos;
247         Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
248
249         // Tomamos nombre de archivo
250         std::string nombreArchivo = listaArgumentos.verPrimero();
251         listaArgumentos.eliminarPrimero();
252
253         // Tomamos cantidad de bytes que debe tener el archivo ahora
254         std::string sCantBytesTotal = listaArgumentos.verPrimero();
255         unsigned int cantBytesTotal = Convertir::stoui(sCantBytesTotal);
256         listaArgumentos.eliminarPrimero();
257
258         // Lista de bloques a reemplazar
259         Lista< std::pair< int, std::string > > bloques;
260         Lista< int > numBloques;
261
262         // Armamos mensaje con notificación
263         std::string respuesta;
264         respuesta.append(S_FILE_CHANGED);
```



```
265         respuesta.append(" ");
266         respuesta.append(nombreArchivo);
267         respuesta.append(COMMON_DELIMITER);
268         respuesta.append(sCantBytesTotal);
269
270         // Tomamos los bloques y sus contenidos de los argumentos
271         while(¬listaArgumentos.estaVacia()) {
272           std::string sBloque = listaArgumentos.verPrimero();
273           int bloque = Convertir::stoui(sBloque);
274           listaArgumentos.eliminarPrimero();
275           std::string contenido = listaArgumentos.verPrimero();
276           listaArgumentos.eliminarPrimero();
277
278           bloques.insertarUltimo(std::make_pair(bloque, contenido));
279           numBloques.insertarUltimo(bloque);
280         }
281
282         // Resguardamos la estabilidad del sincronizador ante archivos
283         // fantasmas de backup
284         try {
285           // Enviamos a modificar el archivo
286           this→manejadorDeArchivos→modificarArchivo(nombreArchivo,
287             cantBytesTotal, bloques);
288         }
289         catch(char const * e) {
290           // Emitimos salida de log pero seguimos adelante
291           this→logger→emitirLog(e);
292         }
293
294
295         while(¬numBloques.estaVacia()) {
296           int b = numBloques.verPrimero();
297           numBloques.eliminarPrimero();
298           std::string sB = Convertir::itos(b);
299
300           respuesta.append(COMMON_DELIMITER);
301           respuesta.append(sB);
302           respuesta.append(COMMON_DELIMITER);
303           respuesta.append(
304             this→manejadorDeArchivos→obtenerHashDeBloque(
305               nombreArchivo, b));
306         }
307
308         // Se envia la notificación de modificación a los clientes
309         this→emisor→ingresarMensajeDeSalida(0, respuesta, mensaje.first);
310
311         // Mensaje de log
312         std::string ee = "SINCRONIZADOR: Se realizó notificación a ";
313         ee += "clientes de la modificación de un archivo.";
314         this→logger→emitirLog(ee);
315       }
316       // Caso en que se recibe la notificación de la eliminación de archivo
317       else if (instruccion ≡ COMMON_DELETE_FILE) {
318         // Mensaje de log
319         std::string e = "SINCRONIZADOR: Recepción de orden de eliminación";
320         e += "de un archivo.";
321         this→logger→emitirLog(e);
322
323         // Parseamos argumentos
324         Lista< std::string > listaArgumentos;
325         Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
326
327         // Eliminamos archivo en carpeta del servidor
328         this→manejadorDeArchivos→eliminarArchivo(listaArgumentos[0]);
329
330         // Enviamos notificación a clientes de que se eliminó archivo
```

jun 25, 13 13:44 **server_sincronizador.cpp** Page 6/6

```
331         std::string respuesta;
332         respuesta.append(COMMON_DELETE_FILE);
333         respuesta.append(" ");
334         respuesta.append(listaArgumentos[0]);
335 
336         // Se envia la notificación dela eliminación archivo
337         this→emisor→ingresarMensajeDeSalida(0, respuesta, mensaje.first);
338 
339         // Mensaje de log
340         std::string ee = "SINCRONIZADOR: Envío de orden de eliminación";
341         ee += "de un archivo a clientes.";
342         this→logger→emitirLog(ee);
343       }
344     }
345 }
```

jun 25, 13 13:44 **server_servidor.h** Page 1/2

```
1  //
2  //  server_servidor.h
3  //  CLASE SERVIDOR
4  //
5 
6 
7  #ifndef SERVIDOR_H
8  #define SERVIDOR_H
9 
10 
11 #include "common_thread.h"
12 #include "common_socket.h"
13 #include "common_lista.h"
14 #include "common_logger.h"
15 #include "server_conexion_cliente.h"
16 #include "server_administrador_de_clientes.h"
17 #include "server_administrador_de_cuentas.h"
18 
19 
20 
21 
22 
23 /* ************************************************************************
24  * DECLARACIÓN DE LA CLASE
25  * ************************************************************************/
26 
27 
28 class Servidor : public Thread {
29 private:
30 
31   int puerto;                // Puerto en el que se escucha.
32   Socket socket;               // Socket en el que escucha el
33                              // servidor.
34   bool activo;               // Estado del seridor
35   AdministradorDeClientes *admClientes;   // Administrador de clientes
36   AdministradorDeCuentas *admCuentas;     // Administra las cuentas
37                              // de los clientes
38   Logger *logger;              // Logger de eventos
39 
40 public:
41 
42   // Constructor
43   Servidor();
44 
45   // Destructor
46   ~Servidor();
47 
48   // Define tareas a ejecutar en el hilo.
49   // Mantiene a la escucha al servidor y acepta nuevos clientes.
50   virtual void run();
51 
52   // Inicia la ejecución del servidor. No debe utilizarse el método start()
53   // para iniciar.
54   // POST: si se inició correctamente el servidor devuelve true, y en caso
55   // contrario devuelve false
56   bool iniciar(int puerto);
57 
58   // Detiene la ejecución del servidor. No debe utilizarse el método stop()
59   // para detener.
60   void detener();
61 
62   // Comprueba si el servidor se encuentra activo.
63   // POST: devuelve true si el servidor se encuentra iniciado y en ejecución
64   // o false si se encuentra detenido.
65   bool estaActivo();
66 };
```

jun 25, 13 13:44 **server_servidor.h** Page 2/2

```
67
68  #endif
```

jun 25, 13 13:44 **server_servidor.cpp** Page 1/3

```
1   //
2   //  server_servidor.cpp
3   //  CLASE SERVIDOR
4   //
5
6
7   #include <iostream>
8   #include "common_lock.h"
9   #include "server_config.h"
10  #include "server_servidor.h"
11
12
13
14  // Constantes
15  namespace {
16    const int MAX_CONEXIONES = 10;
17  }
18
19
20
21  /* ******************************************************************************
22   * DEFINICIÓN DE LA CLASE
23   * ******************************************************************************/
24
25
26  // Constructor
27  Servidor::Servidor() : activo(false) {
28    // Creamos el logger
29    this→logger = new Logger(LOGGER_RUTA_LOG + LOGGER_NOMBRE_LOG);
30
31    // Creamos al administrador de clientes
32    this→admClientes = new AdministradorDeClientes(this→logger);
33    this→admClientes→iniciar();
34
35    // Se crea un administrador de usuario y contrasenia
36    this→admCuentas = new AdministradorDeCuentas;
37  }
38
39
40  // Destructor
41  Servidor::~Servidor() {
42    // Liberamos espacio utilizado por atributos
43    this→admClientes→detener();
44    this→admClientes→join();
45    delete this→admClientes;
46    delete this→admCuentas;
47    delete this→logger;
48  }
49
50
51  // Define tareas a ejecutar en el hilo.
52  // Mantiene a la escucha al servidor y acepta nuevos clientes.
53  void Servidor::run() {
54    // Nos ponemos a la espera de clientes que se conecten
55    while(this→isActive()) {
56      Socket *socketCLI = 0;
57
58      // Aceptamos nuevo cliente
59      socketCLI = this→socket.aceptar();
60
61      // Salimos si el socket no esta activo o si se interrumpió
62      // la escucha de solicitudes de conexión
63      if(¬this→socket.estaActivo() ∨ ¬socketCLI) break;
64
65      // Mensaje de log
66      this→logger→emitirLog("Se ha conectado un cliente (IP = 0.0.0.0).");
```

```
67
68      // Generamos una nueva conexión para escuchate
69      ConexionCliente *conexionCLI = new ConexionCliente(socketCLI,
70          this→admClientes, this→admCuentas, this→logger);
71
72      // Damos la orden de que comience a ejecutarse el hilo del cliente.
73      conexionCLI→start();
74    }
75  }
76
77
78  // Inicia la ejecución del servidor. No debe utilizarse el método start()
79  // para iniciar.
80  // POST: si se inició correctamente el servidor devuelve true, y en caso
81  // contrario devuelve false
82  bool Servidor::iniciar(int puerto) {
83    // Guardamos el puerto
84    this→puerto = puerto;
85
86    // Mensaje de log
87    this→logger→emitirLog("Iniciando servidor AU...");
88
89    try {
90      // Iniciamos la escucha del servidor
91      this→socket.crear();
92      this→socket.escuchar(MAX_CONEXIONES, this→puerto);
93    }
94    catch(char const * e) {
95      // Creamos entrada en Log para informar error
96      this→logger→emitirLog(e);
97      std::string err = "ERROR: Falló el intento de conexión del servidor.";
98      this→logger→emitirLog(err);
99
100     // Detenemos servidor de inmediato
101     this→detener();
102
103     return false;
104   }
105
106   // Iniciamos hilo de ejecución
107   this→start();
108
109   // Cambiamos el estado del servidor
110   this→activo = true;
111
112   // Mensaje de log
113   this→logger→emitirLog("Se ha iniciado el servidor.");
114
115   return true;
116 }
117
118
119 // Detiene la ejecución del servidor. No debe utilizarse el método stop()
120 // para detener.
121 void Servidor::detener() {
122   // Cambiamos el estado del servidor
123   this→activo = false;
124
125   // Detenemos hilo
126   this→stop();
127
128   // Forzamos el cierre del socket para evitar nuevas conexiones entrantes
129   try {
130     this→socket.cerrar();
131   }
132   // Ante una eventual detención abrupta, previa a la inicialización del
```



```
133   // socket, lanzará un error que daremos por obviado.
134   catch(...) { }
135
136   // Mensaje de log
137   this→logger→emitirLog("Se ha detenido el servidor.");
138 }
139
140
141 // Comprueba si el servidor se encuentra activo.
142 // POST: devuelve true si el servidor se encuentra iniciado y en ejecución
143 // o false si se encuentra detenido.
144 bool Servidor::estaActivo() {
145   return this→activo;
146 }
```

jun 25, 13 13:44 **server_recolector_de_informacion.h** Page 1/1

```
1  // Devuelve informacion sobre cantidad de bytes almacenados en servidor
2
3  #ifndef SERVER_RECOLECTOR_DE_INFORMACION_H
4  #define SERVER_RECOLECTOR_DE_INFORMACION_H
5
6  #include <string>
7  #include <stack>
8  #include "dirent.h"
9  #include <sys/stat.h>
10 #include "server_configuracion.h"
11
12 class Recolector {
13 public:
14
15   // Devuelve la cantidad de bytes almacenados en el servidor
16   // recorriendo todos los directorios que se encuentran a partir
17   // del raiz. Soporta directorios anidados
18   static int cantidadBytesAlmacenados();
19 };
20
21 #endif /* SERVER_RECOLECTOR_DE_INFORMACION_H */
```

jun 25, 13 13:44 **server_recolector_de_informacion.cpp** Page 1/1

```
1  #include "server_recolector_de_informacion.h"
2
3  // Devuelve la cantidad de bytes almacenados en el servidor
4  // recorriendo todos los directorios que se encuentran a partir
5  // del raiz. Soporta directorios anidados
6  int Recolector::cantidadBytesAlmacenados() {
7    // Variables auxiliares
8    std::string dirActual, dirAux, nombreEntrada;
9    int tamTotal = 0;
10   std::stack< std::string > pilaDirectorios;
11   // Variables para recorrer directorios
12   DIR *dir;
13   struct dirent *entrada = 0;
14   struct stat file;
15   unsigned char esDirectorio = 0x4;
16
17   // Se obtiene el directorio raiz
18   Configuracion conf;
19   std::string directorio = conf.obtenerPath();
20
21   // Inserto el directorio raiz en la pila
22   pilaDirectorios.push(directorio);
23
24   // Mientras pila no vacia
25   while (¬pilaDirectorios.empty()) {
26     // Levanto la direccion actual
27     dirActual = pilaDirectorios.top();
28     pilaDirectorios.pop();
29
30     // Abrimos directorio
31     dir = opendir(dirActual.c_str());
32
33     // Iteramos sobre cada objeto del directorio
34     while ((entrada = readdir (dir)) ≠ NULL) {
35       // Guardo el nombre de la entrada
36       nombreEntrada = entrada→d_name;
37
38       // Si es directorio, se guarda en pila
39       if (entrada→d_type ≡ esDirectorio) {
40         if (nombreEntrada ≠ "." ∧ nombreEntrada ≠ "..") {
41           // Se guarda el path entero
42           dirAux = dirActual + nombreEntrada + "/";
43
44           // Se agrega el path a la pila de directorios
45           pilaDirectorios.push(dirAux);
46         }
47       }
48       // Sino, se guarda cantidad de bytes
49       else {
50         // Se obtiene el path entero
51         dirAux = dirActual + nombreEntrada;
52
53         // Se obtiene la info del archivo
54         stat(dirAux.c_str(), &file);
55
56         // Se suma el tamanio al total
57         tamTotal += file.st_size;
58       }
59
60     }
61     closedir(dir);
62   }
63   return tamTotal;
64 }
```

jun 25, 13 13:44 **server_receptor.h** Page 1/1

```
//
//  server_receptor.h
//  CLASE RECEPTOR
//


#ifndef RECEPTOR_H
#define RECEPTOR_H


#include <string>
#include <utility>
#include "common_cola.h"
#include "common_mutex.h"
#include "common_lock.h"
#include "common_logger.h"
#include "common_seguridad.h"




/* ******************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ******************************************************************************/


class Receptor {
private:

  Cola< std::pair < int, std::string > > entrada;     // Cola de entrada
  Mutex me;                          // Mutex de entrada
  Mutex ms;                          // Mutex de salida
  Logger *logger;                        // Logger de eventos
  std::string clave;                       // Clave para
                              // firmar mensajes

public:

  // Constructor
  Receptor(Logger *logger, const std::string &clave);

  // Destructor
  ~Receptor();

  // Ingresa un mensaje de entrada en el receptor
  // PRE: 'id' es el identificador de quien ingresa el mensaje; 'msg' es la
  // cadena que contiene el mensaje de entrada.
  void ingresarMensajeDeEntrada(int id, std::string msg);

  // Permite obtener un mensaje recibido.
  // POST: devuelve un objeto pair con el primer mensaje de la cola de
  // mensajes entrantes. En el objeto pair, el primer miembro contiene el
  // identificador de quien envió el mensaje y el segundo miembro contiene
  // el mensaje. Al destruir al receptor se devuelve una cadena vacía para
  // permitir seguir con el flujo del programa a los usuarios que se
  // encuentren bloqueados por el método.
  std::pair < int, std::string > obtenerMensajeDeEntrada();
};

#endif
```

jun 25, 13 13:44 **server_receptor.cpp** Page 1/1

```
//
//  server_receptor.h
//  CLASE RECEPTOR
//


#include "server_receptor.h"
#include "common_protocolo.h"




/* ******************************************************************************
 * DEFINICIÓN DE LA CLASE
 * ******************************************************************************/


// Constructor
Receptor::Receptor(Logger *logger, const std::string &clave) : logger(logger),
  clave(clave) { }


// Destructor
Receptor::~Receptor() { }


// Ingresa un mensaje de entrada en el receptor
// PRE: 'id' es el identificador de quien ingresa el mensaje; 'msg' es la
// cadena que contiene el mensaje de entrada.
void Receptor::ingresarMensajeDeEntrada(int id, std::string msg) {
  // Bloqueamos mutex de entrada
  Lock l(this→me);

  // Verifico firma antes de insertar
  int delim = msg.find(COMMON_DELIMITER);
  std::string firma = msg.substr(0, delim);
  msg = msg.substr(delim + 1);

  // Insertamos mensaje en la cola
  this→entrada.push(std::make_pair(id, msg));
}


// Permite obtener un mensaje recibido.
// POST: devuelve un objeto pair con el primer mensaje de la cola de
// mensajes entrantes. En el objeto pair, el primer miembro contiene el
// identificador de quien envió el mensaje y el segundo miembro contiene
// el mensaje. Al destruir al receptor se devuelve una cadena vacía para
// permitir seguir con el flujo del programa a los usuarios que se
// encuentren bloqueados por el método.
std::pair < int, std::string > Receptor::obtenerMensajeDeEntrada() {
  // Bloqueamos mutex de salida
  Lock l(this→ms);

  // Desencolamos mensaje
  return this→entrada.pop_bloqueante();
}
```

jun 25, 13 13:44 **server_main.cpp** Page 1/2

```
1  //
2  //  ARCHIVOS UBICUOS
3  //  Programa principal del SERVIDOR
4  //
5  //  ****************************************************************************
6  //
7  //  Facultad de Ingeniería - UBA
8  //  75.42 Taller de Programación I
9  //  Trabajo Práctico N°5
10 //
11 //  ALUMNOS:
12 //  Belén Beltran (91718) - belubeltran@gmail.com
13 //  Fiona Gonzalez Lisella () - dynamo89@gmail.com
14 //  Federico Martín Rossi (92086) - federicomrossi@gmail.com
15 //
16 //  ****************************************************************************
17 //
18 //  Programa servidor el cual se encarga de [...]
19 //
20 //
21 //
22 //  FORMA DE USO
23 //  ============
24 //
25 //  Deberá ejecutarse el programa en la línea de comandos de la siguiente
26 //  manera:
27 //
28 //    # ./server
29 //
30 //  Para detener la ejecución del servidor se debe presionar la tecla 'q'
31 //  seguido de ENTER.
32 //
33
34
35
36 #include <iostream>
37 #include "server_configuracion.h"
38 #include "server_servidor.h"
39
40
41
42 namespace {
43   // Constantes que definen los comandos válidos
44   const std::string CMD_SALIR = "q";
45 }
46
47
48
49
50 /* ****************************************************************************
51  * PROGRAMA PRINCIPAL
52  * ****************************************************************************/
53
54
55 int main(int argc, char** argv) {
56   // Corroboramos cantidad de argumentos
57   if(argc > 1) {
58     // Enviamos a log
59     std::cerr << "ERROR: cantidad incorrecta de argumentos." << std::endl;
60     return 1;
61   }
62
63   // Creamos módulos utilizados
64   Servidor servidor;
65   Configuracion configuracion;
66
```

jun 25, 13 13:44 **server_main.cpp** Page 2/2

```
67   // Mensaje de log
68   std::cout << "Iniciando servidor AU..." << std::endl;
69   std::cout.flush();
70
71   // Iniciamos servidor
72   if(¬servidor.iniciar(configuracion.obtenerPuerto())) {
73     // Enviamos a log
74     std::cerr << "ERROR: No ha sido posible iniciar el servidor."
75       << std::endl;
76
77     return 0;
78   }
79
80   // Mensaje de log
81   std::cout << "Servidor corriendo. Presione 'q' para salir." << std::endl;
82   std::cout.flush();
83
84   // Variable auxiliar
85   std::string comando;
86
87   // Esperamos a que se indique la finalización de la ejecución
88   while(comando ≠ CMD_SALIR)
89     getline(std::cin, comando);
90
91
92   // Mensaje de log
93   std::cout << "Deteniendo el servidor..." << std::endl;
94   std::cout.flush();
95
96   // Damos orden de detener servidor
97   servidor.detener();
98   servidor.join();
99
100  // Mensaje de log
101  std::cout << "Servidor detenido. " << std::endl;
102  std::cout.flush();
103
104  return 0;
105 }
```

```
1  //
2  //  server_emisor.h
3  //  CLASE EMISOR
4  //
5
6
7  #ifndef EMISOR_H
8  #define EMISOR_H
9
10
11 #include <string>
12 #include <utility>
13 #include "common_cola.h"
14 #include "common_lista.h"
15 #include "common_thread.h"
16 #include "common_mutex.h"
17 #include "common_lock.h"
18 #include "common_logger.h"
19 #include "server_conexion_cliente.h"
20 #include "common_seguridad.h"
21
22
23
24
25
26 /* ******************************************************************************
27  * DECLARACIÓN DE LA CLASE
28  * ******************************************************************************/
29
30
31 class Emisor : public Thread {
32 private:
33
34   // Cola de salida
35   Cola< std::pair< std::string, std::pair< int, int > > > salida;
36
37   Lista< ConexionCliente* > *listaConexiones;     // Conexiones
38   Mutex m;                        // Mutex
39   Logger *logger;                      // Logger de eventos
40   std::string clave;                     // Clave utilizada
41                             // para firmar mensajes
42
43 public:
44
45   // Constructor
46   // PRE: 'listaConexiones' es la lista de conexiones de clientes sobre las
47   // que se realizan las emisiones.
48   Emisor(Lista< ConexionCliente* > *listaConexiones, Logger *logger,
49     const std::string &clave);
50
51   // Destructor
52   ~Emisor();
53
54   // Inicia la emisión
55   void iniciar();
56
57   // Detiene la emisión
58   void detener();
59
60   // Ingresa un mensaje de entrada en el receptor
61   // PRE: 'id' es el identificador de a quien se envía el mensaje; 'msg' es
62   // la cadena que contiene el mensaje de entrada; 'idExclusion' es el id
63   // de quien debe excluirse del envío. Este último es útil cuando el id=0
64   // ya que permite obviar el envio de una de todas las conexiones.
65   void ingresarMensajeDeSalida(int id, std::string msg, int idExclusion);
66
```



```
67   // Define tareas a ejecutar en el hilo.
68   // Se encarga de emitir lo que se encuentre en la cola de salida.
69   virtual void run();
70 };
71
72 #endif
```

jun 25, 13 13:44 **server_emisor.cpp** Page 1/2

```
1   //
2   //  server_emisor.h
3   //  CLASE EMISOR
4   //
5
6
7   #include "server_emisor.h"
8
9
10
11  namespace {
12    const std::string COLA_SALIDA_FIN = "COLA-SALIDA-FIN";
13  }
14
15
16
17
18  /* ************************************************************************
19   * DEFINICIÓN DE LA CLASE
20   * ************************************************************************/
21
22
23  // Constructor
24  Emisor::Emisor(Lista< ConexionCliente* > *listaConexiones, Logger *logger,
25    const std::string &clave) :
26    listaConexiones(listaConexiones), logger(logger), clave(clave) { }
27
28
29  // Destructor
30  Emisor::~Emisor() { }
31
32
33  // Inicia la emisión
34  void Emisor::iniciar() {
35    this→start();
36  }
37
38
39  // Detiene la emisión
40  void Emisor::detener() {
41    // Detenemos hilo
42    this→stop();
43
44    // Esperamos a que se termine de emitir los mensajes de la cola
45    while(¬this→salida.vacia());
46
47    // Destrabamos la cola encolando un mensaje de finalización detectable
48    this→salida.push(std::make_pair(COLA_SALIDA_FIN, std::make_pair(0,0)));
49  }
50
51
52  // Ingresa un mensaje de entrada en el receptor
53  // PRE: 'id' es el identificador de a quien se envía el mensaje; 'msg' es
54  // la cadena que contiene el mensaje de entrada; 'idExclusion' es el id
55  // de quien debe excluirse del envío. Este último es útil cuando el id=0
56  // ya que permite obviar el envio de una de todas las conexiones.
57  void Emisor::ingresarMensajeDeSalida(int id, std::string msg,
58    int idExclusion) {
59    // Bloqueamos mutex
60    Lock l(this→m);
61
62    // Insertamos mensaje en la cola
63    this→salida.push(std::make_pair(msg, std::make_pair(id, idExclusion)));
64  }
65
66
```

jun 25, 13 13:44 **server_emisor.cpp** Page 2/2

```
67  // Define tareas a ejecutar en el hilo.
68  // Se encarga de emitir lo que se encuentre en la cola de salida.
69  void Emisor::run() {
70    // Variables auxiliares
71    std::string mensajeFirmado;
72
73    // Emitimos lo que vaya siendo insertado en la cola de salida. Ante una
74    // detención del thread, se seguirá emitiendo hasta vaciar la cola de
75    // salida.
76    while(this→isActive() ∨ ¬this→salida.vacia()) {
77      // Tomamos un mensaje de salida
78      std::pair< std::string, std::pair< int, int > > mensaje;
79      mensaje = this→salida.pop_bloqueante();
80
81      // Corroboramos si no se ha desencolado el mensaje que marca el fin
82      if(mensaje.first ≡ COLA_SALIDA_FIN) return;
83
84      // Se firma el mensaje
85      mensajeFirmado = Seguridad::obtenerFirma(mensaje.first, this→clave) +
86        COMMON_DELIMITER + mensaje.first;
87
88      // Caso en que se debe enviar el mismo mensaje a todos los clientes
89      if(mensaje.second.first ≡ 0) {
90        // Iteramos sobre la lista y enviamos el mensaje uno a uno
91        for(size_t i = 0; i < this→listaConexiones→tamanio(); i++) {
92          ConexionCliente *cc = (*this→listaConexiones)[i];
93
94          // Si es el excluido, salteamos
95          if(cc→id() ≡ mensaje.second.second)
96            continue;
97
98          // Enviamos
99          cc→enviarMensaje(mensajeFirmado);
100       }
101     }
102     // Caso en que se debe enviar el mensaje a un único cliente
103     else {
104       // Iteramos sobre la lista hasta encontrar al destinatario correcto
105       for(size_t i = 0; i < this→listaConexiones→tamanio(); i++) {
106         ConexionCliente *cc = (*this→listaConexiones)[i];
107
108         // Comparamos identificadores para ver si es el cliente deseado
109         if(cc→id() ≡ mensaje.second.first){
110           cc→enviarMensaje(mensajeFirmado);
111           break;
112         }
113       }
114     }
115   }
116 }
```

jun 25, 13 13:44 **server_configuracion.h** Page 1/2

```
//
//  client_configuracion.h
//  CLASE CONFIGURACION
//


#ifndef CONFIGURACION_H
#define CONFIGURACION_H

#include "common_archivoTexto.h"

// CONSTANTES
namespace {

  // Metadatos sobre el archivo de configuración
  const std::string CONFIG_DIR = "config/";
  const std::string CONFIG_FILENAME = "server";
  const std::string CONFIG_FILE_EXT = ".properties";

  // Parámetros configurables
  const std::string CONFIG_P_PORT = "PUERTO";
  const std::string CONFIG_P_PATH = "PATH";
  const std::string CONFIG_P_HOST = "HOST";


  // Separadores
  const std::string CONFIG_SEPARATOR = "=";

  // Indicador de comentarios
  const std::string CONFIG_COMMENT = "#";
}



/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


class Configuracion {
private:
  ArchivoTexto* Archivo;

public:

  // Constructor
  Configuracion();

  // Destructor
  ~Configuracion();

  // Devuelve el valor especifico que se necesita
  std::string getInfo(std :: string &cadena);

  // Devuelve el puerto del servidor.
  int obtenerPuerto();

  // Devuelve el host del servidor.
  std::string obtenerHost();

  // Devuelve el path raiz de las carpetas de los clientes
  std::string obtenerPath();


  // Guarda cambios realizados sobre la configuracion.
  void guardarCambios(string puerto, string host, string path);
```

jun 25, 13 13:44 **server_configuracion.h** Page 2/2

```

};

#endif
```

```
1  //
2  //  server_configuracion.h
3  //  CLASE CONFIGURACION
4  //
5
6
7  #include <iostream>
8  #include <string>
9  #include <sstream>
10 #include "common_convertir.h"
11 #include "server_configuracion.h"
12
13 using namespace std;
14
15
16 /* ************************************************************************
17  * DEFINICIÓN DE LA CLASE
18  * ************************************************************************/
19
20
21 // Constructor
22 Configuracion::Configuracion() {
23
24 }
25
26 // Destructor
27 Configuracion::~Configuracion() { }
28
29
30 // Devuelve el valor especifico que se necesita
31 std::string Configuracion::getInfo(std :: string &cadena) {
32   string val;
33   unsigned pos = cadena.find(CONFIG_SEPARATOR);         // position of "=" in ca
   dena
34   val = cadena.substr (pos+1);
35   return val;
36 }
37
38 // Devuelve el puerto del servidor.
39 int Configuracion::obtenerPuerto() {
40   string* cadena = new string();
41   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
42     CONFIG_FILE_EXT,0);
43   bool estado = false;
44   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_PORT)));
45   string result = getInfo(*cadena);
46   delete(this→Archivo);
47   delete(cadena);
48   return Convertir:: stoi(result);
49 }
50
51 // Devuelve el path de los clientes en el servidor
52 string Configuracion::obtenerHost() {
53
54   string* cadena = new string();
55   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
56     CONFIG_FILE_EXT,0);
57   bool estado = false;
58   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_HOST)));
59   string result = getInfo(*cadena);
60   delete(this→Archivo);
61   delete(cadena);
62   return result;
63 }
64
65
```



```
66 string Configuracion::obtenerPath() {
67
68   string* cadena = new string();
69   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
70     CONFIG_FILE_EXT,0);
71   bool estado = false;
72   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_PATH)));
73   string result = getInfo(*cadena);
74   delete(this→Archivo);
75   delete(cadena);
76   return result;
77
78 }
79
80 void Configuracion::guardarCambios(string puerto, string host, string path) {
81
82   this→Archivo = new ArchivoTexto(CONFIG_DIR + CONFIG_FILENAME +
83     CONFIG_FILE_EXT,1);
84   string* aux = new string();
85   *aux += "#SETTINGS SERVER";
86   *aux += '\n';
87   this→Archivo→escribir(*aux);
88
89   aux→clear();
90   *aux += CONFIG_P_HOST;
91   *aux += CONFIG_SEPARATOR;
92   *aux += host;
93   *aux += '\n';
94   this→Archivo→escribir(*aux);
95
96   aux→clear();
97   *aux += CONFIG_P_PORT;
98   *aux += CONFIG_SEPARATOR;
99   *aux += puerto;
100  *aux += '\n';
101  this→Archivo→escribir(*aux);
102
103  aux→clear();
104  *aux += CONFIG_P_PATH;
105  *aux += CONFIG_SEPARATOR;
106  *aux += path;
107  *aux += '\n';
108  this→Archivo→escribir(*aux);
109
110  delete(this→Archivo);
111  delete(aux);
112
113 }
114
```

jun 25, 13 13:44 **server_config.h** Page 1/1

```
//
//  server_config.h
//
//  Cabecera con constantes de configuración de uso interno
//


#ifndef CONFIG_H
#define CONFIG_H

#include <string>




/* ************************************************************************
 * CONSTANTES DE CONFIGURACIÓN INTERNA
 * ************************************************************************/


// Constantes para los logs
const std::string LOGGER_RUTA_LOG = "logs/";
const std::string LOGGER_NOMBRE_LOG = "eventos_servidor";

// Constantes para las bases de datos
const std::string BD_RUTA = "bd/";

// Constantes para la ubicación y rutas de directorios
const std::string DIR_RAIZ_CARPETAS = "carpetas/";

#endif
```

jun 25, 13 13:44 **server_conexion_cliente.h** Page 1/2

```
//
//  common_conexion_cliente.h
//  CLASE CONEXIONCLIENTE
//


#ifndef CONEXION_CLIENTE_H
#define CONEXION_CLIENTE_H



#include "common_thread.h"
#include "common_socket.h"
#include "common_comunicador.h"
#include "common_logger.h"
#include "server_receptor.h"
#include "server_recolector_de_informacion.h"
#include "server_administrador_de_cuentas.h"
class AdministradorDeClientes;



namespace {
  // Constantes para los nombres de directorio
  const std::string MONITOR_USER = "ADMMONITOR";
}




/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


class ConexionCliente : public Thread {
private:

  Socket *socket;               // Socket de comunicación
  std::string nombreUsuario;          // Nombre de usuario de cliente
  AdministradorDeClientes *admClientes;   // Administrador de clientes
  AdministradorDeCuentas* admCuentas;         // Administrador cuentas de client
e
  Receptor *receptor;             // Receptor a donde se envian
  std::string pathCarpeta;          // Path donde se encuentran
                          // los archivos del cliente
                          // los datos que arrivan
  bool habilitarRecepcion;          // Traba para evitar recepción
                          // hasta que se indique.
  Logger *logger;               // Logger de eventos
  std::string clave;              // Clave utilizada para firmar
                          // mensajes

  // Espera inicio sesion
  int inicioSesion(Comunicador& comunicador);

  // Se ocupa de atender a las solicitudes enviadas por el monitor.
  void atenderAMonitor(std::string& mensaje, Comunicador *com);

public:

  // Constructor
  // PRE: 's' es un socket para la comunicación con el cliente; 'id' es
  // número de cliente que se le ha sido asignado por el servidor; 'serv' es
  // una referencia al servidor al que pertenece la conexión.
```

jun 25, 13 13:44 **server_conexion_cliente.h** Page 2/2

```
66    ConexionCliente(Socket *s, AdministradorDeClientes *adm, AdministradorDeCuenta
    s *ac,
67      Logger *logger);
68
69    // Destructor
70    ~ConexionCliente();
71
72    // Devuelve el id que identifica a la conexión.
73    int id();
74
75    // Define tareas a ejecutar en el hilo.
76    virtual void run();
77
78    // Detiene la conexión con el cliente. No debe utilizarse el método stop()
79    // para detener, sino este mismo en su lugar.
80    void detener();
81
82    // Devuelve el nombre de usuario con el que inicio sesión el cliente.
83    // POST: si aún no ha iniciado sesión, se devuelve una cadena vacía.
84    std::string getNombreUsuario();
85
86    // Asigna un receptor a la conexión, a quien le enviará los datos que se
87    // reciban del cliente.
88    // PRE: 'unReceptor' es el receptor.
89    // POST: la conexión comenzará a derivar los datos llegados hacia el
90    // receptor.
91    void asignarReceptor(Receptor *unReceptor);
92
93    // Envia un mensaje al cliente.
94    // PRE: 'mensaje' es la cadena que desea enviarse.
95    // POST: lanza una excepción si el socket no se encuentra activo.
96    void enviarMensaje(std::string& mensaje);
97  };
98
99  #endif
```

jun 25, 13 13:44 **server_conexion_cliente.cpp** Page 1/5

```
1   //
2   //  common_conexion_cliente.h
3   //  CLASE CONEXIONCLIENTE
4   //
5
6   #include <iostream>
7   #include <sstream>
8   #include "common_protocolo.h"
9   #include "common_parser.h"
10  #include "common_convertir.h"
11  #include "server_administrador_de_clientes.h"
12  #include "server_conexion_cliente.h"
13  #include "common_lista.h"
14
15
16
17
18
19  /* ******************************************************************************
20   * DEFINICIÓN DE LA CLASE
21   * ******************************************************************************/
22
23
24  // Constructor
25  // PRE: 's' es un socket para la comunicación con el cliente; 'id' es
26  // número de cliente que se le ha sido asignado por el servidor; 'serv' es
27  // una referencia al servidor al que pertenece la conexión.
28  ConexionCliente::ConexionCliente(Socket *s,
29    AdministradorDeClientes *adm, AdministradorDeCuentas *ac, Logger *logger) :
30    socket(s), nombreUsuario(""), admClientes(adm), admCuentas(ac),
31    pathCarpeta(""), habilitarRecepcion(false), logger(logger) { }
32
33
34  // Destructor
35  ConexionCliente::~ConexionCliente() {
36    // Liberamos memoria utilizada por el socket
37    delete this→socket;
38  }
39
40
41  // Devuelve el id que identifica a la conexión.
42  int ConexionCliente::id() {
43    return this→socket→obtenerID();
44  }
45
46
47  // Define tareas a ejecutar en el hilo.
48  void ConexionCliente::run() {
49    // Creamos el comunicador para recibir mensajes
50    Comunicador comunicador(this→socket);
51
52    // Variables de procesamiento
53    std::string mensaje;
54
55    // Mensaje de log
56    this→logger→emitirLog("LOGIN: Esperando datos de login del cliente...");
57
58    // Si el inicio de sesión falló, cerramos conexión con cliente
59    if(this→inicioSesion(comunicador) ≠ 1) {
60      this→admClientes→destruirCliente(this);
61
62      // Mensaje de log
63      this→logger→emitirLog("LOGIN: Falló inicio de sesión de cliente.");
64      this→logger→emitirLog("Se ha desconectado al cliente.");
65
66      return;
```

```
67    }
68
69    // Mensaje de log
70    this→logger→emitirLog("LOGIN: Usuario '" + this→nombreUsuario + "'"
71      + " ha iniciado sesión.");
72
73    // Nos autoregistramos en el administrador de clientes
74    this→admClientes→ingresarCliente(this→nombreUsuario, this,
75      this→pathCarpeta, this→clave);
76
77    // Si se conectó una aplicación monitor, derivamos hacia allí
78    if(this→nombreUsuario ≡ MONITOR_USER) {
79      // Se inicia la recepción de mensajes desde el cliente
80      while(this→isActive()) {
81        // Esperamos a recibir mensaje
82        if(comunicador.recibir(mensaje) ≡ −1) break;
83
84        // Derivamos para que sea atendido
85        this→atenderAMonitor(mensaje, &comunicador);
86      }
87    }
88    else {
89      // Esperamos a que se habilite la recepción, es decir, que se
90      // especifique un receptor
91      while(¬habilitarRecepcion)
92        if(¬this→isActive()) return;
93
94      // Se inicia la recepción de mensajes desde el cliente
95      while(this→isActive()) {
96        // Esperamos a recibir mensaje
97        if(comunicador.recibir(mensaje) ≡ −1) break;
98
99        // Enviamos el mensaje al receptor
100       this→receptor→ingresarMensajeDeEntrada(this→id(), mensaje);
101     }
102   }
103
104   // Avisamos al administrador que la conexión debe darse de baja y ser
105   // destruida
106   this→admClientes→darDeBajaCliente(this→nombreUsuario, this);
107   this→admClientes→destruirCliente(this);
108
109   // Mensaje de log
110   this→logger→emitirLog("Usuario '" + this→nombreUsuario + "' "
111     + " se ha desconectado.");
112 }
113
114
115 // Detiene la conexión con el cliente. No debe utilizarse el método stop()
116 // para detener, sino este mismo en su lugar.
117 void ConexionCliente::detener() {
118   // Detenemos hilo
119   this→stop();
120
121   // Forzamos el cierre del socket y destrabamos espera de recepcion de datos
122   try {
123     this→socket→cerrar();
124   }
125   // Ante una eventual detención abrupta, previa a la inicialización del
126   // socket, lanzará un error que daremos por obviado.
127   catch(...) { }
128 }
129
130
131 // Devuelve el nombre de usuario con el que inicio sesión el cliente.
132 // POST: si aún no ha iniciado sesión, se devuelve una cadena vacía.
```



```
133 std::string ConexionCliente::getNombreUsuario() {
134   return this→nombreUsuario;
135 }
136
137
138 // Asigna un receptor a la conexión, a quien le enviará los datos que se
139 // reciban del cliente.
140 // PRE: 'unReceptor' es el receptor.
141 // POST: la conexión comenzará a derivar los datos llegados hacia el
142 // receptor.
143 void ConexionCliente::asignarReceptor(Receptor *unReceptor) {
144   this→receptor = unReceptor;
145
146   // Habilitamos la recepción de datos
147   this→habilitarRecepcion = true;
148 }
149
150
151 // Envia un mensaje al cliente.
152 // PRE: 'mensaje' es la cadena que desea enviarse.
153 // POST: lanza una excepción si el socket no se encuentra activo.
154 void ConexionCliente::enviarMensaje(std::string& mensaje) {
155   // Corroboramos que el socket esté activo
156   if(¬this→socket→estaActivo()) {
157     // Mensaje de log
158     this→logger→emitirLog("ERROR: No se pudo emitir mensaje a usuario '"
159       + this→nombreUsuario + "'");
160
161     // Lanzamos excepción
162     throw "ERROR: No se pudo emitir mensaje a cliente.";
163   }
164
165   // Creamos el comunicador para enviar mensajes
166   Comunicador comunicador(this→socket);
167
168   // Enviamos el mensaje
169   comunicador.emitir(mensaje);
170 }
171
172
173
174
175
176 /*
177  * IMPLEMENTACIÓN DE MÉTODOS PRIVADOS DE LA CLASE
178  */
179
180
181 // Espera inicio sesion
182 int ConexionCliente::inicioSesion(Comunicador& comunicador) {
183   // Variables de procesamiento
184   std::string instruccion;
185   std::string args;
186
187   // Se recibe la instruccion
188   if(comunicador.recibir(instruccion, args) ≡ −1)
189     return −1;
190
191   // Se debe crear nuevo usuario
192   if (instruccion ≡ C_LOGIN_REQUEST) {
193     // Caso en que la verificación es correcta
194     if (admCuentas→verificarCliente(args, this→nombreUsuario,
195       this→pathCarpeta, this→clave) ≡ 1) {
196       comunicador.emitir(S_LOGIN_OK);
197       return 1;
198     }
```

```
199       // Caso en que la verificación es incorrecta
200       else {
201         comunicador.emitir(S_LOGIN_FAIL);
202         return 0;
203       }
204     }
205     return -1;
206   }
207
208
209   // Se ocupa de atender a las solicitudes enviadas por el monitor.
210   void ConexionCliente::atenderAMonitor(std::string& mensaje, Comunicador *com) {
211     // Variables auxiliares
212     std::string instruccion, args;
213     AdministradorDeCuentas admin;
214
215     // Parseamos instruccion
216     Parser::parserInstruccion(mensaje, instruccion, args);
217
218     // Caso en que se solicita información al server
219     if(instruccion ≡ M_SERVER_INFO_REQUEST) {
220       // Armamos respuesta
221       std::string respuesta;
222       respuesta.append(S_SERVER_INFO);
223       respuesta.append(" ");
224       respuesta.append(Convertir::itos(
225         this→admClientes→cantidadDeClientesConectados()));
226       respuesta.append(COMMON_DELIMITER);
227       respuesta.append(Convertir::itos(
228         this→admClientes→cantidadDeCarpetasActivas()));
229       respuesta.append(COMMON_DELIMITER);
230       int bytesAlmacenados = Recolector::cantidadBytesAlmacenados();
231       respuesta.append(Convertir::itos(bytesAlmacenados));
232
233       // Emitimos respuesta
234       com→emitir(respuesta);
235     }
236     else if (instruccion ≡ M_SERVER_USER_LIST_REQUEST) {
237       // Se pide a admin cuentas que devuelva la lista de clientes
238       Lista<std::string> listaUsuarios;
239       admin.obtenerListaUsuarios(listaUsuarios);
240
241       // Se prepara el mensaje
242       size_t i, tam = listaUsuarios.tamanio();
243       std::string respuesta = S_SERVER_USER_LIST + " ";
244
245       for (i = 0; i < tam - 1; i++)
246         respuesta.append(listaUsuarios[i] + COMMON_DELIMITER);
247       // Se agrega el ultimo parametro
248       respuesta.append(listaUsuarios[i]);
249
250       // Emitimos respuesta
251       com→emitir(respuesta);
252     }
253     else if (instruccion ≡ M_SERVER_MODIFY_USER_REQUEST) {
254       // Variables aux
255       Lista<std::string> listaArgs;
256
257       // Se obtiene nombre y clave
258       Parser::dividirCadena(args, &listaArgs, COMMON_DELIMITER[0]);
259
260       // Se modifica la clave
261       admin.modificarCliente(listaArgs[0], listaArgs[1], false);
262     }
263
264     else if (instruccion ≡ M_SERVER_NEW_USER_INFO) {
```



```
265       // Variables aux
266       Lista<std::string> listaArgs;
267
268       // Se obtiene nombre y clave
269       Parser::dividirCadena(args, &listaArgs, COMMON_DELIMITER[0]);
270
271       // Se agrega cliente al archivo
272       admin.agregarCliente(listaArgs[0], listaArgs[1]);
273     }
274     else if (instruccion ≡ M_SERVER_DELETE_USER) {
275       // Se elimina el usuario con el nombre especificado
276       admin.eliminarCliente(args);
277     }
278     else if (instruccion ≡ M_SERVER_LOG_REQUEST) {
279       // Q HACEMO?
280       std::string respuesta;
281       respuesta += S_SERVER_LOG + " ";
282       respuesta.append("mensaje de log  ");
283       com→emitir(respuesta);
284     }
285     else if (instruccion ≡ M_SERVER_MODIFY_USER) {
286       // se te envia usuario viejo/usuario nuevo/clave
287       // buscas por el viejo y modificas la clave siempre..pisas todo
288       std::cout<<"info para modificar usuario"<<std::endl;
289       std::cout<<mensaje<<std::endl;
290     }
291   }
```

jun 25, 13 13:44 **server_carpeta.h** Page 1/1

```
//
//  server_carpeta.h
//  CLASE CARPETA
//


#ifndef CARPETA_H
#define CARPETA_H


#include "common_lista.h"
#include "common_logger.h"
#include "common_manejador_de_archivos.h"
#include "server_receptor.h"
#include "server_emisor.h"
#include "server_sincronizador.h"
#include "server_conexion_cliente.h"




/* ******************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ******************************************************************************/


class Carpeta {
private:

  Lista< ConexionCliente* > listaConexiones;      // Lista de clientes
  Receptor *receptor;                 // Receptor
  Emisor *emisor;                   // Emisor
  Sincronizador *sincronizador;           // Sincronizador
  ManejadorDeArchivos *manejadorDeArchivos;     // Manejador
  Logger *logger;                   // Logger de eventos

  // Crea una carpeta fisica para el usuario si no existe ya una carpeta
  // Devuelve 1 si la operacion es correcta y 0 sino
  int crearCarpeta(const std::string &usuario);

public:

  // Constructor
  Carpeta(const std::string &pathCarpeta, Logger *logger, const
    std::string &clave);

  // Destructor
  ~Carpeta();

  // Vincula a un cliente como miembro activo del directorio
  void vincularCliente(ConexionCliente *unCliente);

  // Desvincula a un cliente del directorio.
  void desvincularCliente(ConexionCliente *unCliente);

  // Devuelve la cantidad de clientes que se encuentran activos en la carpeta
  int cantidadClientes();
};

#endif
```

jun 25, 13 13:44 **server_carpeta.cpp** Page 1/2

```
//
//  server_carpeta.h
//  CLASE CARPETA
//


#include "server_config.h"
#include "server_carpeta.h"
#include "server_conexion_cliente.h"
#include <dirent.h>
#include <sys/stat.h>




/* ******************************************************************************
 * DEFINICIÓN DE LA CLASE
 * ******************************************************************************/


// Constructor
Carpeta::Carpeta(const std::string &pathCarpeta, Logger *logger,
  const std::string &clave) :
  logger(logger) {
  // Creamos el receptor que recibirá los mensajes entrantes
  this→receptor = new Receptor(this→logger, clave);

  // Creamos el emisor que enviará mensajes a los clientes
  this→emisor = new Emisor(&this→listaConexiones, this→logger,
    clave);

  // Si no existe carpeta fisica se crea.
  // Si no lo logra, lanza excepcion
  std::string path = DIR_RAIZ_CARPETAS + pathCarpeta + "/";

  // Si no se pudo crear el directorio, lanzamos error
  if(¬crearCarpeta(path)){
    // Mensaje de log
    this→logger→emitirLog("ERROR: No se pudo crear directorio " + path);
    throw "ERROR: No se pudo crear directorio";
  }

  // Se crea el manejador de archivos
  this→manejadorDeArchivos = new ManejadorDeArchivos(path, this→logger);

  // Creamos el sincronizador
  this→sincronizador = new Sincronizador(this→receptor, this→emisor,
    this→manejadorDeArchivos, this→logger);

  // Iniciamos los hilos
  this→emisor→iniciar();
  this→sincronizador→iniciar();
}


// Destructor
Carpeta::~Carpeta() {
  // Detenemos módulos
  this→sincronizador→detener();
  this→sincronizador→join();

  this→emisor→detener();
  this→emisor→join();

  // Liberamos memoria utilizada
```

jun 25, 13 13:44 **server_carpeta.cpp** Page 2/2

```
67    delete this→sincronizador;
68    delete this→receptor;
69    delete this→emisor;
70    delete this→manejadorDeArchivos;
71  }
72
73
74  // Vincula a un cliente como miembro activo del directorio
75  void Carpeta::vincularCliente(ConexionCliente *unCliente) {
76    // Listamos la conexión
77    this→listaConexiones.insertarUltimo(unCliente);
78
79    // Le asignamos el receptor de la carpeta para que comience a recibir
80    unCliente→asignarReceptor(this→receptor);
81  }
82
83
84  // Desvincula a un cliente del directorio.
85  void Carpeta::desvincularCliente(ConexionCliente *unCliente) {
86    this→listaConexiones.eliminar(unCliente);
87  }
88
89
90  // Devuelve la cantidad de clientes que se encuentran activos en la carpeta
91  int Carpeta::cantidadClientes() {
92    return this→listaConexiones.tamanio();
93  }
94
95
96
97
98
99
100 /*
101  * IMPLEMENTACIÓN DE MÉTODOS PRIVADOS DE LA CLASE
102  */
103
104
105 // Crea una carpeta fisica para el usuario si no existe ya una carpeta
106 // Devuelve 1 si la operacion es correcta y 0 sino
107 int Carpeta::crearCarpeta(const std::string &pathCarpeta) {
108   // Se intenta abrir el directorio
109   DIR* carpeta = opendir(pathCarpeta.c_str());
110   if (carpeta ≡ NULL) {  // No existe, entonces se crea
111     if (¬mkdir(pathCarpeta.c_str(), S_IFDIR | S_IRWXU | S_IFDIR));
112       return 0;
113   }
114   else
115     closedir(carpeta);
116   return 1;
117 }
118
```

jun 25, 13 13:44 **server_administrador_de_cuentas.h** Page 1/2

```
1   // Encargado de administrar altas, bajas y modificaciones de cuentas
2   // de usuario
3
4   #ifndef SERVER_ADMINISTRADOR_DE_CUENTAS_H
5   #define SERVER_ADMINISTRADOR_DE_CUENTAS_H
6
7   #include <iostream>
8   #include <fstream>
9   #include <string>
10  #include "common_lista.h"
11  #include "common_mutex.h"
12  #include "common_lock.h"
13  #include "server_configuracion.h"
14
15  class AdministradorDeCuentas {
16  private:
17    // Mutex
18    Mutex m;
19
20    // Se crea el archivo si no existe con permisos especiales
21    void crearArchivo(const std::string &nombre);
22
23    // Se crea el directorio si no existe con permisos especiales
24    void crearDir(const std::string &nombre);
25
26    // Se elimina un directorio y su contenido
27    void eliminarDir(const std::string &nombre);
28
29    // Verifica si el archivo existe
30    bool existeArchivo(const std::string &nombre);
31
32    // Busqueda secuencial. Devuelve 1 si encuentra clave y usuario
33    // correctamente, 2 si encuentra usuario y no clave correcta,
34    // y −1 si no lo encuentra
35    int buscarCliente(std::string &usuario, std::string &clave,
36      std::string &carpeta);
37
38    // Devuelve la lista de carpetas existentes en el directorio
39    void obtenerListaCarpetas(Lista<std::string> &listaCarpetas);
40
41  public:
42    // Ctor
43    AdministradorDeCuentas();
44
45    // Dtor
46    ~AdministradorDeCuentas();
47
48    // Devuelve la lista de nombres de usuarios que se encuentran
49    // registrados
50    void obtenerListaUsuarios(Lista<std::string> &listaUsuarios);
51
52    // Comprueba nombre de usuario y clave de los clientes
53    // POST: si la verificación es exitosa, se almacena en 'nombreUsuario' el
54    // nombre de usuario del cliente.
55    int verificarCliente(std::string &args, std::string& nombreUsuario,
56      std::string &pathCarpeta, std::string &clave);
57
58    // Agrega un cliente a la lista
59    void agregarCliente(const std::string &nombre, const std::string
60      &clave);
61
62    // Se elimina un cliente existente
63    void eliminarCliente(const std::string &nombre);
64
65    // Se modifica un cliente existente
66    void modificarCliente(const std::string &nombre,
```

jun 25, 13 13:44 **server_administrador_de_cuentas.h** Page 2/2

```
67        const std::string &dato, bool esModifNombre);
68  };
69
70  #endif
```

jun 25, 13 13:44 **server_administrador_de_cuentas.cpp** Page 1/6

```
1   #include "server_administrador_de_cuentas.h"
2   #include <sys/stat.h>
3   #include <sys/types.h>
4   #include "common_protocolo.h"
5   #include "common_parser.h"
6   #include "common_utilidades.h"
7   #include "dirent.h"
8
9   namespace {
10    #define PATH_ARCHIVO "bd/Users_Pass.txt"
11    #define PATH_ARCHIVO_TEMP "bd/Users_Pass_tmp.txt"
12    #define LONG_PATH 10
13    const std::string NOMBRE_MONITOR = "ADMMONITOR";
14  }
15
16  // Ctor
17  AdministradorDeCuentas::AdministradorDeCuentas() {
18  }
19
20  // Dtor
21  AdministradorDeCuentas::~AdministradorDeCuentas() {
22  }
23
24
25  // Devuelve la lista de usuarios que se encuentran registrados
26  void AdministradorDeCuentas::obtenerListaUsuarios(Lista<std::string>
27    &listaUsuarios) {
28
29    Lock l(m);
30
31    // Se crea el archivo si no existe, y se devuelve lista vacia
32    if (¬existeArchivo(PATH_ARCHIVO))
33      crearArchivo(PATH_ARCHIVO);
34
35    else {
36      // Se abre el arcihvo de registros de usuario
37      std::fstream archivo(PATH_ARCHIVO, std::ios_base::in |
38        std::ios_base::out | std::ios_base::app);
39
40      //Variables auxiliares
41      std::string linea, nombre;
42
43      // Se lee registro por registro guardando nombres en la lista
44      while (getline(archivo, linea)) {
45        nombre = linea.substr(0, linea.find(
46          COMMON_DELIMITER));
47        // Si es el monitor, se saltea de la lista
48        if (nombre ≠ NOMBRE_MONITOR)
49          listaUsuarios.insertarUltimo(nombre);
50      }
51
52      archivo.close();
53    }
54  }
55
56
57  // Comprueba nombre de usuario y clave de los clientes
58  // POST: si la verificación es exitosa, se almacena en 'nombreUsuario' el
59  // nombre de usuario del cliente.
60  int AdministradorDeCuentas::verificarCliente(std::string &args,
61    std::string& nombreUsuario, std::string &pathCarpeta,
62    std::string &contrasenia) {
63    // Variables auxiliares
64    std::string usuario, clave, carpeta;
65    int delim;
66
```

```
67    delim = args.find(COMMON_DELIMITER, 0);
68    usuario = args.substr(0, delim);
69    clave = args.substr(delim + 1, std::string::npos);
70    int existe = -1;
71
72    Lock l(m);
73
74    // Si no existe, se crea el archivo
75    if (¬existeArchivo(PATH_ARCHIVO))
76      crearArchivo(PATH_ARCHIVO);
77
78    // Abre el archivo
79    std::fstream archivo(PATH_ARCHIVO, std::ios_base::in);
80
81    // se busca el cliente
82    existe = buscarCliente(usuario, clave, carpeta);
83
84    // Cierra el archivo
85    archivo.close();
86
87    // Si la validación es exitosa, almacenamos el nombre de usuario
88    if(existe) {
89      nombreUsuario = usuario;
90      pathCarpeta = carpeta;
91      contrasenia = clave;
92    }
93
94    return existe;
95  }
96
97
98  // Agrega un cliente a la lista
99  void AdministradorDeCuentas::agregarCliente(const std::string &nombre,
100   const std::string &clave) {
101
102   Lock l(m);
103
104   // Variables aux
105   std::string carpeta;
106
107   // Se crea un archivo si no existe
108   if (¬existeArchivo(PATH_ARCHIVO))
109     crearArchivo(PATH_ARCHIVO);
110
111   // Se abre el arcihvo al final
112   std::fstream archivo(PATH_ARCHIVO, std::ios_base::out |
113       std::ios_base::app);
114
115   // Obtengo lista de nombres de carpeta existentes
116   Lista<std::string> nombresCarpetas;
117   obtenerListaCarpetas(nombresCarpetas);
118
119   // Se crea un nombre de carpeta aleatorio y se comprueba
120   // que no exista
121   Utilidades::randomString(LONG_PATH, carpeta);
122
123   // Si existe el archivo, se buscan colisiones
124   if (¬nombresCarpetas.estaVacia()) {
125     while (nombresCarpetas.buscar(carpeta)) {
126       // Se crea un nombre de carpeta aleatorio y se comprueba
127       // que no exista
128       Utilidades::randomString(LONG_PATH, carpeta);
129     }
130   }
131
132   // Se arma el path y se crea carpeta fisica
```



```
133   Configuracion conf;
134   std::string path = conf.obtenerPath() + carpeta + "/";
135   crearDir(path);
136
137   // Se agrega el usuario al final
138   archivo << nombre + COMMON_DELIMITER + clave +
139       COMMON_DELIMITER + carpeta + "\n";
140   archivo.flush();
141
142   archivo.close();
143 }
144
145 // Se elimina un cliente existente y su carpeta relacionada
146 void AdministradorDeCuentas::eliminarCliente(const std::string &nombre) {
147
148   Lock l(m);
149
150   // Se abre el archivo
151   if (existeArchivo(PATH_ARCHIVO)) {
152     // Variables aux
153     std::string linea, nombreActual, carpeta;
154
155     // Se crea un archivo temporal
156     std::fstream temp(PATH_ARCHIVO_TEMP, std::ios_base::out |
157         std::ios_base::app);
158
159     std::fstream archivo(PATH_ARCHIVO, std::ios_base::in);
160
161     // Se va leyendo el archivo original y se escriben los clientes
162     // Si se encuentra el buscado, se saltea
163     while (getline(archivo, linea)) {
164       nombreActual = linea.substr(0, linea.find(
165         COMMON_DELIMITER));
166       // si no es el que hay que eliminar, se escribe
167       if (nombreActual ≠ nombre) {
168         temp << linea << std::endl;
169         temp.flush();
170       }
171       // Si lo encontre, se elimina su carpeta
172       else {
173         // Se obtiene la carpeta
174         carpeta = linea.substr(linea.find_last_of(
175           COMMON_DELIMITER));
176         // Se elimina del directorio
177         eliminarDir(carpeta);
178       }
179     }
180     // Se elimina el archivo viejo y se renombra el temporal
181     archivo.close();
182     temp.close();
183     remove(PATH_ARCHIVO);
184     rename(PATH_ARCHIVO_TEMP, PATH_ARCHIVO);
185   }
186 }
187
188 // Se modifica un cliente existente. Si esModifNombre = true, entonces
189 // el 'dato' es un nombre de cliente, sino si esModifNombre = false
190 // es una clave.
191 void AdministradorDeCuentas::modificarCliente(const std::string &nombre,
192   const std::string &dato, bool esModifNombre) {
193
194   Lock l(m);
195
196   // Se abre el archivo
197   if (existeArchivo(PATH_ARCHIVO)) {
198     // Variables aux
```

```
199     std::string linea, nombreActual, aux;
200
201     // Se crea un archivo temporal
202     std::fstream temp(PATH_ARCHIVO_TEMP, std::ios_base::out |
203       std::ios_base::app);
204
205     std::fstream archivo(PATH_ARCHIVO, std::ios_base::in);
206
207     // Se va leyendo el archivo original y se escriben los clientes
208     // Si se encuentra el buscado, se modifica y escribe
209     while (getline(archivo, linea)) {
210       // Se obtiene el nombre del registro recien leido
211       nombreActual = linea.substr(0, linea.find(
212         COMMON_DELIMITER));
213       // si no es el que hay que modificar, se escribe
214       if (nombreActual ≠ nombre)
215         temp << linea << std::endl;
216       // Si encontre el que hay que modificar, se modifica
217       else {
218         if (esModifNombre) {
219           aux = dato + linea.substr(linea.find(
220               COMMON_DELIMITER));
221         }
222         else {
223           aux = nombre + COMMON_DELIMITER + dato
224           + linea.substr(linea.find_last_of(
225           COMMON_DELIMITER));
226         }
227         // Se guardan cambios en archivo
228         temp << aux << std::endl;
229       }
230     }
231     // Se elimina el archivo viejo y se renombra el temporal
232     archivo.close();
233     temp.close();
234     remove(PATH_ARCHIVO);
235     rename(PATH_ARCHIVO_TEMP, PATH_ARCHIVO);
236   }
237 }
238
239
240
241 /** Implementacion de metodos privados **/
242
243
244
245 // Se crea el archivo si no existe con permisos especiales
246 void AdministradorDeCuentas::crearDir(const std::string &nombre) {
247   if(mkdir(nombre.c_str(), S_IRWXU | S_ISVTX) ≡ -1)
248     throw "ERROR: No se pudo crear archivo.";
249 }
250
251 // Se elimina un directorio y su contenido
252 void AdministradorDeCuentas::eliminarDir(const std::string &nombre) {
253
254   // Variables auxiliares
255   DIR *dir;
256   struct dirent *entrada = 0;
257   unsigned char esDirectorio = 0x4;
258   Configuracion conf;
259   std::string directorio = conf.obtenerPath() + nombre + "/";
260   std::string nombreEntrada;
261
262   // Abrimos directorio y procesamos si fue exitosa la apertura
263   if((dir = opendir (directorio.c_str())) ≠ NULL) {
264     // Iteramos sobre cada objeto del directorio
```



```
265     while ((entrada = readdir (dir)) ≠ NULL) {
266       // Se lee una entrada
267       nombreEntrada = entrada→d_name;
268       if (nombreEntrada ≠ "." ∧ nombreEntrada ≠ "..") {
269         // Si no es directorio, se elimina
270         if (entrada→d_type ≠ esDirectorio) {
271           nombreEntrada = directorio +
272             entrada→d_name;
273           remove(nombreEntrada.c_str());
274         }
275       }
276     }
277     closedir(dir);
278     remove(directorio.c_str());
279   }
280 }
281
282 void AdministradorDeCuentas::crearArchivo(const std::string &nombre) {
283   std::fstream arch(nombre.c_str(), std::ios_base::out);
284   arch.close();
285   chmod(nombre.c_str(), S_IRWXU | S_ISVTX);
286 }
287
288 // Verifica si el archivo existe
289 bool AdministradorDeCuentas::existeArchivo(const std::string &nombre) {
290   std::fstream arch(nombre.c_str(), std::ios_base::in);
291   if (¬arch.is_open())
292     return false;
293   arch.close();
294   return true;
295 }
296
297 // Busqueda secuencial. Devuelve 1 si encuentra clave y usuario
298 // correctamente, 2 si encuentra usuario y no clave correcta,
299 // y -1 si no lo encuentra
300 int AdministradorDeCuentas::buscarCliente(std::string &usuario, std::string &cla
    ve,
301   std::string &carpeta) {
302   // Variables auxiliares
303   std::string usuarioActual, claveActual, lineaActual;
304   int codigo = -1;
305   Lista<std::string> param;
306
307   // Va al principio del archivo
308   std::fstream archivo(PATH_ARCHIVO, std::ios_base::in);
309   archivo.seekg(0, std::ios_base::beg);
310
311   // Busca uno por uno si encuentra al cliente
312   while (getline(archivo, lineaActual)) {
313     // Se parsea la linea
314     Parser::dividirCadena(lineaActual, &param, COMMON_DELIMITER[0]);
315     // Se obtienen los datos
316     usuarioActual = param[0];
317     claveActual = param[1];
318     // Compara
319     if (usuario.compare(usuarioActual) ≡ 0) {
320       if (clave.compare(claveActual) ≡ 0) {
321         if (param.tamanio() > 2)
322           carpeta = param[2];
323         codigo = 1;  // Encontre usuario y clave
324         break;
325       }
326       codigo = 2;  // Encontre usuario y no clave
327       break;
328     }
329     param.vaciar();
```

jun 25, 13 13:44 **server_administrador_de_cuentas.cpp** Page 6/6

```
330   }
331   return codigo;  // No encontre ni usuario ni clave
332 }
333
334
335 // Devuelve la lista de carpetas existentes en el directorio
336 void AdministradorDeCuentas::obtenerListaCarpetas(Lista<std::string>
337   &listaCarpetas) {
338
339   // Se crea el archivo si no existe, y se devuelve lista vacia
340   if (¬existeArchivo(PATH_ARCHIVO))
341     crearArchivo(PATH_ARCHIVO);
342
343   else {
344     // Se abre el arcihvo de registros de usuario
345     std::fstream archivo(PATH_ARCHIVO, std::ios_base::in |
346       std::ios_base::out | std::ios_base::app);
347
348     //Variables auxiliares
349     std::string linea, carpeta;
350
351     // Se lee registro por registro guardando nombres en la lista
352     while (getline(archivo, linea)) {
353       carpeta = linea.substr(linea.find_last_of(
354         COMMON_DELIMITER), std::string::npos);
355       listaCarpetas.insertarUltimo(carpeta);
356     }
357   }
358 }
```

jun 25, 13 13:44 **server_administrador_de_clientes.h** Page 1/2

```
1  //
2  //  server_administrador_de_clientes.h
3  //  CLASE ADMINISTRADORDECLIENTES
4  //
5
6
7  #ifndef ADMINISTRADOR_DE_CLIENTES_H
8  #define ADMINISTRADOR_DE_CLIENTES_H
9
10
11 #include <string>
12 #include <map>
13 #include "common_thread.h"
14 #include "common_cola.h"
15 #include "common_lista.h"
16 #include "common_logger.h"
17 #include "server_conexion_cliente.h"
18 #include "server_carpeta.h"
19
20
21
22
23
24 /* ******************************************************************************
25  * DECLARACIÓN DE LA CLASE
26  * ******************************************************************************/
27
28
29 class AdministradorDeClientes : public Thread {
30 private:
31
32   std::map< std::string, Carpeta* > carpetas;   // Diccionario de carpetas
33   Lista< ConexionCliente* > listaMonitores;   // Lista de monitores
34   Cola< ConexionCliente* > conexionesMuertas;   // Cola de conexiones que
35                                                 // deben ser destruidas
36   Logger *logger;                 // Logger de eventos
37
38 public:
39
40   // Constructor
41   AdministradorDeClientes(Logger *logger);
42
43   // Destructor
44   ~AdministradorDeClientes();
45
46   // Ingresa un cliente como miembro activo del directorio al que se
47   // encuentra vinculado.
48   void ingresarCliente(std::string usuario, ConexionCliente *unCliente,
49     const std::string &pathCarpeta, const std::string &clave);
50
51   // Da de baja a un cliente, el cual debe haber sido ingresado previamente
52   // como miembro activo del directorio al que se encuentra vinculado.
53   void darDeBajaCliente(std::string usuario, ConexionCliente *unCliente);
54
55   // Da aviso al administrador de que debe destruirse un cliente
56   void destruirCliente(ConexionCliente *unCLiente);
57
58   // Define tareas a ejecutar en el hilo.
59   virtual void run();
60
61   // Inicia el administrador de clientes
62   void iniciar();
63
64   // Detiene el administrador de clientes
65   void detener();
66
```

```
67    // Devuelve la cantidad de clientes conectados actualmente.
68    unsigned int cantidadDeClientesConectados();
69
70    // Devuelve la cantidad de carpetas activas actualmente.
71    unsigned int cantidadDeCarpetasActivas();
72  };
73
74  #endif
```



```
1   //
2   //  server_administrador_de_clientes.cpp
3   //  CLASE ADMINISTRADORDECLIENTES
4   //
5
6
7   #include "server_administrador_de_clientes.h"
8
9
10
11
12
13  /* ************************************************************************
14   * DEFINICIÓN DE LA CLASE
15   * ************************************************************************/
16
17
18  // Constructor
19  AdministradorDeClientes::AdministradorDeClientes(Logger *logger) :
20    logger(logger) { }
21
22
23  // Destructor
24  AdministradorDeClientes::~AdministradorDeClientes() { }
25
26
27  // Ingresa un cliente como miembro activo del directorio al que se
28  // encuentra vinculado.
29  void AdministradorDeClientes::ingresarCliente(std::string usuario,
30    ConexionCliente *unCliente, const std::string &pathCarpeta,
31    const std::string &clave) {
32    // Si es un cliente monitor, insertamos en lista de monitores solamente
33    if(usuario ≡ MONITOR_USER) {
34      this→listaMonitores.insertarUltimo(unCliente);
35      return;
36    }
37
38    // Corroboramos si ya hay una carpeta activa para dicho usuario
39    // Si no existe una carpeta activa, creamos una carpeta
40    if(this→carpetas.count(usuario) ≡ 0)
41      this→carpetas[usuario] = new Carpeta(pathCarpeta,
42        this→logger, clave);
43
44    // Vinculamos al cliente con la carpeta
45    this→carpetas[usuario]→vincularCliente(unCliente);
46  }
47
48
49  // Da de baja a un cliente, el cual debe haber sido ingresado previamente
50  // como miembro activo del directorio al que se encuentra vinculado.
51  void AdministradorDeClientes::darDeBajaCliente(std::string usuario,
52      ConexionCliente *unCliente) {
53    // Si es un cliente monitor, eliminamos de la lista de monitores
54    if(usuario ≡ MONITOR_USER) {
55      this→listaMonitores.eliminar(unCliente);
56      return;
57    }
58    // Si no hay una carpeta vinculada al usuario, no hacemos nada
59    else if(this→carpetas.count(usuario) ≡ 0) return;
60
61    // Desvinculamos la conexión de la carpeta
62    this→carpetas[usuario]→desvincularCliente(unCliente);
63
64    // Si la carpeta no contiene mas clientes activos, la destruimos
65    if(this→carpetas[usuario]→cantidadClientes() ≡ 0) {
66      // Liberamos espacio usado por la carpeta
```

```
67      delete this→carpetas[usuario];
68      // Quitamos el registro del contenedor de carpetas
69      this→carpetas.erase(usuario);
70    }
71  }
72
73
74  // Da aviso al administrador de que debe destruirse un cliente
75  void AdministradorDeClientes::destruirCliente(ConexionCliente *unCliente) {
76    this→conexionesMuertas.push(unCliente);
77  }
78
79
80  // Define tareas a ejecutar en el hilo.
81  void AdministradorDeClientes::run() {
82    // Desencolamos de la cola de conexiones muertas y las destruimos
83    while(this→isActive() ∨ ¬this→conexionesMuertas.vacia()) {
84      ConexionCliente *cc = this→conexionesMuertas.pop_bloqueante();
85
86      // Si se detecta una conexión fantasma, salteamos.
87      if(cc ≡ 0) continue;
88
89      // Detenemos conexión con el cliente
90      cc→detener();
91      // Esperamos a que finalice
92      cc→join();
93      // Liberamos memoria
94      delete cc;
95    }
96  }
97
98
99  // Inicia el administrador de clientes
100 void AdministradorDeClientes::iniciar() {
101   // Iniciamos el hilo
102   this→start();
103 }
104
105
106 // Detiene el administrador de clientes
107 void AdministradorDeClientes::detener() {
108   // Detenemos hilo
109   this→stop();
110
111   // Destrabamos la cola encolando una conexión fantasma.
112   this→conexionesMuertas.push(0);
113 }
114
115
116 // Devuelve la cantidad de clientes conectados actualmente.
117 unsigned int AdministradorDeClientes::cantidadDeClientesConectados() {
118   // Si no hay carpetas, devolvemos cero
119   if(this→carpetas.size() ≡ 0) return 0;
120
121   // Variables auxiliares
122   unsigned int cantClientes = 0;
123   std::map< std::string, Carpeta* >::iterator it;
124
125   // Iteramos sobre las carpetas
126   for (it = this→carpetas.begin(); it ≠ this→carpetas.end(); ++it)
127       cantClientes += it→second→cantidadClientes();
128
129     return cantClientes;
130 }
131
132
```



```
133 // Devuelve la cantidad de carpetas activas actualmente.
134 unsigned int AdministradorDeClientes::cantidadDeCarpetasActivas() {
135   return this→carpetas.size();
136 }
```

jun 25, 13 13:44 **monitor_vista_linea.h** Page 1/1

```
1  #ifndef VISTALINEA_H_
2  #define VISTALINEA_H_
3
4  #include <gtkmm.h>
5  #include <gtkmm/drawingarea.h>
6
7  #include <iostream>
8  #include <cairomm/surface.h>
9
10 #include <cairomm/context.h>
11 #include <cairomm/refptr.h>
12
13 #include "monitor_vista.h"
14
15 class VistaLinea : public Vista {
16 public:
17   int xini;
18   int yini;
19   int xfin;
20   int yfin;
21   int escala;
22   VistaLinea(int xini, int yini, int xfin, int yfin);
23   virtual ~VistaLinea();
24   void correrIzquierda(int x);
25   void draw(const Cairo::RefPtr<Cairo::Context>& cr);
26 };
27
28 #endif
```

jun 25, 13 13:44 **monitor_vista_linea.cpp** Page 1/1

```
1  #include "monitor_vista_linea.h"
2
3  VistaLinea::VistaLinea(int xini, int yini, int xfin, int yfin) {
4    this→xini = xini;
5    this→yini = yini;
6    this→xfin = xfin;
7    this→yfin = yfin;
8    this→escala = 1;
9  }
10
11 VistaLinea::~VistaLinea() {}
12 void VistaLinea::correrIzquierda(int x){
13   xini = xini − x;
14   xfin = xfin − x;
15 }
16
17 void VistaLinea::draw(const Cairo::RefPtr<Cairo::Context>& cr) {
18
19   cr→save();
20
21   // dividimos por cuatro −> 1024 / 4 = 256 que es la altura del form (logramos
   q los resultados sean representables)
22
23   // ahora haces si la escala q viene *100  es menor a yini y la escala q viene
   *100 es menor yfin entonces
24
25   //GRAFICAR A ESCALA
26   //CASO A) cualquier representacion menos ( 0 a 100mb)
27   // se toma el valor se lo divide por la escala y por 4 para que entre en el gr
   afico
28   // y luego con el indicador, se muestra en que escala  se trabaja
29
30   //CASO B) Representar 1 a 100mb
31   //entro en el if
32
33   if (escala ≡ 1048576 ∧ yini < (escala*100) ∧ yfin < (escala*100)) {
34         // para que las lineas se desparramen mejor en el grafico de 256 bytes,
35     //ya que no vamos a distribuir de 1 a 1024 valores sino que de 1 a 100, mult
   iplico x 10
36   cr→move_to (xini, 256 − ((yini/escala)*10)/4);
37     cr→line_to (xfin, 256 − ((yfin/escala)*10)/4);
38
39   }else {
40
41   cr→move_to (xini, 256 − (yini/escala)/4);
42     cr→line_to (xfin, 256 − (yfin/escala)/4);
43   }
44   cr→set_source_rgb(255.0, 204.0, 53.0);
45
46     cr→stroke();
47   cr→restore();  // back to opaque black
48
49
50 // El evento on_draw de graficador llama a los eventos draw de todas las vistas
   que se quieren mostrar,
51 // pasandole el objeto cr, en el cual se dibuja
52
53 }
```

jun 25, 13 13:44 **monitor_vistaIndicador.h** Page 1/1

```
#ifndef VISTAINDICADOR_H_
#define VISTAINDICADOR_H_

#include <gtkmm.h>
#include <gtkmm/drawingarea.h>

#include <iostream>
#include <cairomm/surface.h>

#include <cairomm/context.h>
#include <cairomm/refptr.h>

#include "monitor_vista.h"
#include "monitor_monitor.h"

class VistaIndicador : public Vista {
private:
  Monitor* monitor;
  int escala(int bytes);
  std::string escalaStr(int bytes);
public:
  VistaIndicador(Monitor* monitor);
  virtual ~VistaIndicador();
  void draw(const Cairo::RefPtr<Cairo::Context>& cr);
};

#endif /*VISTAINDICADOR_H_*/
```

jun 25, 13 13:44 **monitor_vistaIndicador.cpp** Page 1/1

```
#include "monitor_vistaIndicador.h"
#include "pangomm.h"
#include <iostream>
#define B 1
#define KB 1024
#define MB 1048576

VistaIndicador::VistaIndicador(Monitor* monitor) {
  this→monitor = monitor;

}

VistaIndicador::~VistaIndicador() {
}

void VistaIndicador::draw(const Cairo::RefPtr<Cairo::Context>& cr) {
  // se selecciona formato (font, tamaño y color)
  // El objeto Font es la fuente
  cr→save();
  Pango::FontDescription font;
  font.set_family("Monospace");
  font.set_weight(Pango::WEIGHT_BOLD);
  font.set_absolute_size(20*Pango::SCALE);
  // Instanciamos el Layout, dandole un texto y un font
  // calculamos el numero de bytes / escala + rotulo de escala
  Glib::ustring aux = Convertir::itos(monitor→getBytesOcupados()/escala(monitor
→getBytesOcupados())) + escalaStr(monitor→getBytesOcupados());

  Gtk::DrawingArea win;
  Glib::RefPtr<Pango::Layout> layout = win.create_pango_layout(aux);
  //ponemos el layout (label) en la drawing.
  layout→set_font_description(font);

  cr→move_to(5,5);
  cr→set_source_rgb(255.0, 204.0, 53.0);
  layout→show_in_cairo_context(cr);

}
int VistaIndicador::escala(int bytes) {

  if (bytes < KB) return B;
  if (bytes < MB) return KB;
  return MB;
}

std::string VistaIndicador::escalaStr(int bytes) {

  if (bytes < KB) return "B";
  if (bytes < MB) return "KB";
  return "MB";
}
```

jun 25, 13 13:44 **monitor_vista.h** Page 1/1

```
1  #ifndef VISTA_H_
2  #define VISTA_H_
3
4  #include <iostream>
5
6
7  #include <gtkmm.h>
8  #include <gtkmm/drawingarea.h>
9
10 #include <cairomm/surface.h>
11 #include <cairomm/context.h>
12 #include <cairomm/refptr.h>
13
14
15 class Graficador;
16
17 class Vista {
18 protected:
19   Graficador* graficador;
20
21 public:
22   Vista();
23   virtual ~Vista();
24   virtual void draw(const Cairo::RefPtr<Cairo::Context>& cr)=0;
25   void setGraficador(Graficador* graficador);
26 };
27
28 #endif /*VISTA_H_*/
29
30
31
```

jun 25, 13 13:44 **monitor_vistaFondo.h** Page 1/1

```
1  #ifndef VISTAFONDO_H_
2  #define VISTAFONDO_H_
3
4  #include <gtkmm.h>
5  #include <gtkmm/drawingarea.h>
6
7  #include <iostream>
8  #include <cairomm/surface.h>
9
10 #include <cairomm/context.h>
11 #include <cairomm/refptr.h>
12
13 #include "monitor_vista.h"
14
15 class VistaFondo : public Vista {
16 private:
17   Glib::RefPtr<Gdk::Pixbuf> imagen;
18
19 public:
20   VistaFondo();
21   virtual ~VistaFondo();
22   void draw(const Cairo::RefPtr<Cairo::Context>& cr);
23
24 };
25
26 #endif /*VISTAFONDO_H_*/
```

jun 25, 13 13:44 **monitor_vistaFondo.cpp** Page 1/1

```cpp
#include "monitor_vistaFondo.h"
#include <iostream>

VistaFondo::VistaFondo(){

  std::string imgPath = "interfaz/imagenes/fondo.png";
  this→imagen = Gdk::Pixbuf::create_from_file(imgPath,500,256,true);

}

VistaFondo::~VistaFondo() {
}

void VistaFondo::draw(const Cairo::RefPtr<Cairo::Context>& cr) {

  cr→save();
 // destino a donde se vuelca la imagen de fondo
  Gdk::Cairo::set_source_pixbuf(cr, this→imagen, 0, 0);

  cr→fill();
  cr→paint();
  cr→restore();
  cr→stroke();
}
```

jun 25, 13 13:44 **monitor_vista.cpp** Page 1/1

```cpp
#include "monitor_vista.h"
#include "monitor_graficador.h"
#include <iostream>
Vista::Vista()
{
}

Vista::~Vista()
{
}

void Vista::setGraficador(Graficador* graficador) {
  this→graficador = graficador;
}
```

jun 25, 13 13:44 **monitor_receptorDatos.h** Page 1/2

```
1  //
2  //  monitor_receptorDatos.h
3  //  CLASE RECEPTOR
4  //
5
6
7  #ifndef RECEPTOR_H
8  #define RECEPTOR_H
9
10 #include <string>
11 #include "common_comunicador.h"
12 #include "common_thread.h"
13 #include "common_lista.h"
14 #include "common_socket.h"
15 #include "common_protocolo.h"
16
17 class Comunicador;
18
19
20
21
22
23
24 /* ******************************************************************************
25  * DECLARACIÓN DE LA CLASE
26  * ******************************************************************************/
27
28
29 class Receptor : public Thread {
30 private:
31
32   // Atributos generales
33   Socket *socket;         // Socket con el que se comunica
34   int puerto;           // Puerto de conexión.
35   int timer;            // Timer de actualizacion de informacion
36   std::string nombreHost;     // Nombre del host de conexión
37   bool estadoConexion;      // Censa si se encuentra conectado
38   Lista<std::string> valores;     // valores recibidos
39   // Inicia sesion con usuario existente
40   int iniciarSesion(std::string usuario, std::string clave);
41
42 public:
43
44   // Constructor
45   Receptor();
46
47   // Destructor
48   ~Receptor();
49
50
51   // Define tareas a ejecutar en el hilo.
52   virtual void run();
53
54   // Detiene la conexión con el monitor. No debe utilizarse el método stop()
55   // para detener, sino este mismo en su lugar.
56   void detener();
57
58   // Establece el nombre de host al que se conectará el monitor.
59   void especificarNombreHost(std::string nombreHost);
60
61   // Establece el puerto del host al que se conectará el monitor
62   void especificarPuerto(int puerto);
63
64   // Establece el tiempo actualizacion de informacion
65   void especificarTiempo(int tiempo);
66
```

jun 25, 13 13:44 **monitor_receptorDatos.h** Page 2/2

```
67
68
69
70   // Realiza la conexión inicial con el servidor.
71   // PRE: 'usuario' y 'clave' son el nombre de usuario y contraseña con el
72   // que se desea conectar al servidor. Debe haberse especificado el nombre
73   // de host y puerto.
74   // POST: devuelve '-1' si falló la conexión, '0' si falló el login y '1' si
75   // se conectó y loggeó con éxito.
76   int conectar(std::string usuario, std::string clave);
77
78   // Se desconecta del servidor
79   void desconectar();
80
81   // Envia un mensaje al cliente.
82   // PRE: 'mensaje' es la cadena que desea enviarse.
83   // POST: lanza una excepción si el socket no se encuentra activo.
84   void enviarMensaje(std::string& mensaje);
85
86   int recibirMensaje(std::string& mensaje);
87
88   // Retorna la ultima version de valores obtenidos del servidor
89   Lista <std::string> getValores();
90
91   // Retorna estado de conexion
92   bool getEstadoConexion();
93
94 };
95
96 #endif
```

```
1  //
2  //  monitor_receptorDatos.cpp
3  //  CLASE RECEPTOR
4  //
5
6
7  #include <iostream>
8  #include <sstream>
9
10 #include "common_parser.h"
11 #include "common_convertir.h"
12 #include "monitor_receptorDatos.h"
13 #include "common_hash.h"
14
15
16
17
18
19 /* ************************************************************************
20  * DEFINICIÃM-^SN DE LA CLASE
21  * ************************************************************************/
22
23
24 // Constructor
25 Receptor::Receptor() : estadoConexion(false) { }
26
27
28 // Destructor
29 Receptor::~Receptor() {
30   // Liberamos la memoria utilizada por el socket
31   delete this→socket;
32 }
33
34
35 // Establece el nombre de host al que se conectarÃ¡ el cliente.
36 void Receptor::especificarNombreHost(std::string nombreHost) {
37   this→nombreHost = nombreHost;
38 }
39
40
41 // Establece el puerto del host al que se conectarÃ¡ el cliente
42 void Receptor::especificarPuerto(int puerto) {
43   this→puerto = puerto;
44 }
45
46 void Receptor::especificarTiempo(int timer){
47   this→timer = timer;
48 }
49
50 //no hayy deja de buscar recibir xq no hay pera q lo agarro de un lugar q creo q
    esta hecho...
51 // Realiza la conexiÃ³n inicial con el servidor.
52 // PRE: 'usuario' y 'clave' son el nombre de usuario y contraseÃ±a con el
53 // que se desea conectar al servidor. Debe haberse especificado el nombre
54 // de host, puerto y directorio.
55 // POST: devuelve '-1' si fallÃ³ la conexiÃ³n, '0' si fallÃ³ el login y '1' si
56 // se conectÃ³ y loggeÃ³ con Ã©xito.
57 int Receptor::conectar(std::string usuario, std::string clave) {
58   // Creamos socket
59   this→socket = new Socket();
60   this→socket→crear();
61
62   // Mensaje de log
63   std::cout << "Conectando con " << this→nombreHost << " en el puerto "
64     << this→puerto << "... ";
65     std::cout.flush();
```



```
66
67   try {
68     // Conectamos el socket
69     this→socket→conectar(nombreHost, puerto);
70   }
71   catch(char const * e) {
72     // Mensaje de logjando
73     std::cout << "DESCONECTADO" << std::endl;
74     std::cerr << e << std::endl;
75
76     // Liberamos memoria
77     delete this→socket;
78
79     // FallÃ³ la conexiÃ³n
80     return -1;
81   }
82
83   // Mensaje de log
84   std::cout << "CONECTADO" << std::endl;
85   std::cout.flush();
86
87
88   // Si se iniciÃ³ sesiÃ³n con Ã©xito, salimos y mantenemos socket activo
89   if(iniciarSesion(usuario, clave) ≡ 1) {
90     // Cambiamos el estado de la conexiÃ³n
91     this→estadoConexion = true;
92     this→start();
93
94     return 1;
95   }
96
97   // Destruimos el socket en caso de fallar el inicio de sesiÃ³n
98   desconectar();
99   delete this→socket;
100
101  // FallÃ³ el Ãºoggin
102  return 0;
103 }
104
105
106
107 void Receptor::detener() {
108   // Detenemos hilo
109   this→stop();
110   this→estadoConexion = false;
111
112   // Forzamos el cierre del socket y destrabamos espera de recepcion de datos
113   try {
114     this→socket→cerrar();
115   }
116   // Ante una eventual detenciÃ³n abrupta, previa a la inicializaciÃ³n del
117   // socket, lanzarÃ¡ un error que daremos por obviado.
118   catch(...) { }
119 }
120
121 bool Receptor::getEstadoConexion() {
122   return this→estadoConexion;
123 }
124 void Receptor::run() {
125   // Creamos el comunicador para recibir mensajes
126   Comunicador comunicador(this→socket);
127
128   // Variables de procesamiento
129   std::string mensaje;
130   std::string instruccion;
131   std::string args;
```

```
132   this→estadoConexion = true;
133   int r = 0;
134   while ((r ≡ 0 ) ∧ (this→isActive())) {
135
136     mensaje = M_SERVER_INFO_REQUEST;
137
138     enviarMensaje(mensaje);
139     mensaje.clear();
140     // Enviamos el mensaje al servidor
141     Lista <std::string> aux;
142     instruccion.clear();
143     args.clear();
144     r = comunicador.recibir(mensaje);
145     if (r ≠ −1 ) {
146
147       Parser::parserInstruccion(mensaje, instruccion, args);
148       std::cout << "imprimo argumentos del mensaje "<< args <<std::endl;
149       Parser::dividirCadena(args, &aux, COMMON_DELIMITER[0]);
150       this→valores = aux;
151       std::cout<< "Receptor actualizando valores  "<< mensaje <<std::endl;
152       mensaje.clear();
153     } else {
154       this→estadoConexion = false;
155     }
156
157     this→sleep(this→timer);
158   }
159   this→estadoConexion = false;
160 }
161
162 // Envia un mensaje al servidor.
163 // PRE: 'mensaje' es la cadena que desea enviarse.
164 // POST: lanza una excepciÃ³n si el socket no se encuentra activo.
165 void Receptor::enviarMensaje(std::string& mensaje) {
166   // Corroboramos que el socket estÃ© activo
167   if(¬this→socket→estaActivo()) {
168     throw "ERROR: No se pudo emitir mensaje al servidor.";
169     this→estadoConexion = false;
170   }
171   // Creamos el comunicador para enviar mensajes
172   Comunicador comunicador(this→socket);
173
174   comunicador.emitir(mensaje);
175
176 }
177 int Receptor::recibirMensaje(std::string& mensaje){
178   // Creamos el comunicador para recibir mensajes
179   this→stop();
180   Comunicador comunicador(this→socket);
181
182   int ret = comunicador.recibir(mensaje);
183
184   return ret;
185 }
186
187
188 // Se desconecta del servidor
189 void Receptor::desconectar() {
190   // Mensaje de log
191   std::cout << "Cerrando conexiÃ³n... ";
192     std::cout.flush();
193
194   // Desconectamos el socket
195   this→socket→cerrar();
196
197   // Cambiamos el estado de la conexiÃ³n
```



```
198   this→estadoConexion = false;
199
200   // Mensaje de log
201   std::cout << "DESCONECTADO" << std::endl;
202   std::cout.flush();
203 }
204
205
206
207 // Inicia sesion con Admin existente
208 int Receptor::iniciarSesion(std::string usuario, std::string clave) {
209   // Creamos comunicador
210   Comunicador com(this→socket);
211
212   // Mensaje de log
213   std::cout << "Emitiendo solicitud de LOGIN... " << std::endl;
214     std::cout.flush();
215
216   // Se preparan los argumentos
217   std::string claveHash;
218   claveHash = Hash::funcionDeHash(clave);
219   std::string mensaje = usuario + COMMON_DELIMITER + claveHash;
220   std::cout<<claveHash<<std::endl;
221
222   // Enviamos peticiÃ³n de inicio de sesion
223   if(com.emitir(C_LOGIN_REQUEST, mensaje) ≡ −1) {
224     return −1;
225   }
226
227   // Se obtiene respuesta del servidor
228   std::string args;
229   if(com.recibir(mensaje, args) ≡ −1) {
230     return −1;
231   }
232
233   if (mensaje ≡ S_LOGIN_OK) {
234     std::cout << "Inicio de sesion exitoso" << std::endl;
235     std::cout.flush();
236     return 1;
237   }
238   if (mensaje ≡ S_LOGIN_FAIL) {
239     std::cout << "Inicio de sesion fallo, compruebe nombre de usuario y contrasenia" << std::endl;
240     std::cout.flush();
241     return 0;
242   }
243   return −1;
244 }
245 // actualiza los valores del monitor
246 Lista <std::string> Receptor::getValores() {
247   Lista<std::string> destino = this→valores;
248
249   return destino;
250
251 }
252
```

```
1  //
2  //  monitor_monitor.h
3  //  CLASE MONITOR
4  //
5
6
7  #ifndef MONITOR_H
8  #define MONITOR_H
9
10 #include "common_archivoTexto.h"
11 #include "monitor_receptorDatos.h"
12 #include "common_convertir.h"
13
14
15 /* ******************************************************************************
16  * DECLARACIÓN DE LA CLASE
17  * ******************************************************************************/
18
19
20 class Monitor {
21 private:
22
23   // Atributos generales
24   ArchivoTexto* archivoLog;
25   int carpetasActivas;
26   int clientesConectados;
27   int bytesOcupados;
28   bool estado;
29   Receptor* receptor;
30
31
32 protected:
33   int posLecturaLog;
34
35 public:
36   Lista<std::string> usuarios;
37   // Constructor
38   Monitor(Receptor* receptor);
39
40   // Destructor
41   ~Monitor();
42
43   // Actualizar valores
44
45   void actualizarValores();
46   // Retorna un string con la ultima linea del buffer leida
47   string* getBufferLog();
48
49   // Retorna un string con la cantidad de clientes conectados
50   string getClientesConectados();
51   // Retorna un string con la cantidad de carpetas activas
52   string getCarpetasActivas();
53
54   int getBytesOcupados();
55
56   // Pide al servidor la lista de usuarios existentes
57   void getUsuarios();
58
59   // Retorna si el servidor sigue levantado o no
60   bool getEstadoConexion();
61
62   Receptor* getReceptor();
63
64
65 };
66
```



```
67 #endif
```

```
1
2   #include <iostream>
3   #include <sstream>
4   #include "common_parser.h"
5   #include "monitor_monitor.h"
6   #include "common_protocolo.h"
7
8
9   /* ************************************************************************
10   * DEFINICIÓN DE LA CLASE
11   * ************************************************************************/
12
13
14  // Constructor
15  Monitor::Monitor(Receptor* receptor) {
16    this→posLecturaLog = 0;
17    this→carpetasActivas = 0;
18    this→clientesConectados = 0;
19    this→receptor = receptor;
20
21  }
22
23  Receptor* Monitor::getReceptor(){
24    return receptor;
25  }
26
27
28  void Monitor::actualizarValores() {
29    Lista<std::string> nuevos = this→receptor→getValores();
30    if ((this→receptor→getEstadoConexion()) ≡ true) {
31      this→estado = true;
32      this→clientesConectados = Convertir::stoi(nuevos[0]);
33      this→carpetasActivas = Convertir::stoi(nuevos[1]);
34      this→bytesOcupados = Convertir::stoi(nuevos[2]);
35    } else {
36      this→estado = false;
37    }
38
39  }
40
41  bool Monitor::getEstadoConexion(){
42    return this→estado;
43  }
44
45  string Monitor::getClientesConectados() {
46    return Convertir::itos(this→clientesConectados);
47  }
48  string Monitor::getCarpetasActivas() {
49    return Convertir::itos(this→carpetasActivas);
50  }
51
52  int Monitor::getBytesOcupados() {
53    return this→bytesOcupados;
54  }
55
56  void Monitor::getUsuarios(){
57    string usuarios;
58
59    string query = M_SERVER_USER_LIST_REQUEST; // hay q meter la constante que cor
    responda dsd protocolo.h
60    receptor→enviarMensaje(query);
61
62    receptor→recibirMensaje(usuarios); // tenemos el mensaje
63
64    Lista <std::string> aux;
65    string instruccion;
```



```
66    string args;
67    Parser::parserInstruccion(usuarios, instruccion, args);
68    Parser::dividirCadena(args, &aux, COMMON_DELIMITER[0]);
69    this→usuarios = aux;
70  }
71
72
73
74  Monitor::~Monitor() {
75  }
76
```

jun 25, 13 13:44 **monitor_main.cpp** Page 1/1

```cpp
//  ************************************************************************
//
//  Facultad de Ingeniería − UBA
//  75.42 Taller de Programación I
//  Trabajo Práctico N°5
//
//  ALUMNOS:
//  Belén Beltran (91718) − belubeltran@gmail.com
//  Fiona Gonzalez Lisella (91454) − fgonzalezlisella@gmail.com
//  Federico Martín Rossi (92086) − federicomrossi@gmail.com
//
//  ************************************************************************
//
//  Programa monitor que permite modificar y obtener estadisticas del servidor
//


#include <iostream>
#include "common_thread.h"
#include "monitor_interfaz_principal.h"
#include "monitor_receptorDatos.h"
#include "monitor_monitor.h"
#include "monitor_configuracion.h"
#include "monitor_interfaz_conexion.h"


int main (int argc, char** argv) {
  // Iniciamos interfaz de la ventana principal
  Gtk::Main kit(argc, argv);

  // Creamos la configuracion del servidor
  Configuracion* configs = new Configuracion();

  // Creamos el receptor de datos que se comunica con el servidor
  Receptor* receptor = new Receptor();

  //Iniciamos ventana de conexion
  Conexion ventanaConexion(receptor,configs);
  if (ventanaConexion.correr() ≡ 1) return 0;

  // Si llego aca, el tipo se logeo
  // Creamos el monitor
  Monitor* monitor = new Monitor(receptor);

  std::cout<<"vuelvo a main "<<endl;
  // ventana principal del programa
  MenuPrincipal ventanaMonitor(monitor,configs);
  ventanaMonitor.correr();

  // Liberamos toda la memoria

  delete monitor;
  delete configs;
  delete receptor;

  return 0;
}
```

jun 25, 13 13:44 **monitor_interfaz_usuarios.h** Page 1/2

```cpp
//
//  monitor_interfaz_usuarios.h
//  CLASE MENUUSUARIOS
//


#ifndef MENUUSUARIOS_H_
#define MENUUSUARIOS_H_


#include "gtkmm.h"
#include "common_lista.h"
#include "monitor_monitor.h"


class MenuUsuarios {
private:

  // Atributos de la interfaz
  Gtk::Window *main;         // Ventana De ABM de usuarios


  Gtk::Button *botonNuevo;
  Gtk::Button *botonEliminar;
  Gtk::Button *botonModificar;
  Gtk::Button *botonVolver;


protected:

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_name);}

    Gtk::TreeModelColumn<std::string> m_col_name;


  };

  Gtk::Grid *grid;
  ModelColumns m_Columns;
  Gtk::TreeView tree;
  Glib::RefPtr<Gtk::ListStore> listaUsuarios;
  Glib::RefPtr<Gtk::TreeSelection>  seleccionado;
  Lista<std::string> usuarios; //

  // Atributos del modelo

  Monitor *monitor;


public:
// Constructor
  MenuUsuarios(Monitor *monitor);

  // Destructor
  virtual ~MenuUsuarios();

  // Inicia la ejecución de la ventana
  void correr();

  void on_buttonNuevo_clicked();
```

jun 25, 13 13:44 **monitor_interfaz_usuarios.h** Page 2/2

```
67
68    void on_buttonEliminar_clicked();
69
70    void on_buttonModificar_clicked();
71
72    void on_buttonVolver_clicked();
73    void on_selection_changed();
74
75  };
76
77  #endif /* MENUUSUARIOS_H_ */
```

jun 25, 13 13:44 **monitor_interfaz_usuarios.cpp** Page 1/3

```
1   #include <iostream>
2   #include <string>
3   #include "monitor_interfaz_usuarios.h"
4   #include "monitor_interfaz_formUsuario.h"
5   #include "monitor_interfaz_eliminarUsuario.h"
6   #include "monitor_interfaz_modificarUsuario.h"
7   #include "common_convertir.h"
8
9   #include <iostream>
10  #include <string>
11
12
13
14
15  MenuUsuarios::MenuUsuarios(Monitor *monitor) : monitor(monitor) {
16    // Cargamos la ventana
17    Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
18
19
20    // Cargamos elementos
21    refBuilder→add_from_file("./interfaz/monitor_adminUsuarios.glade");
22    refBuilder→get_widget("main", this→main); // linkeo el form
23    refBuilder→get_widget("grid1", this→grid);
24
25
26    // Botones
27
28    refBuilder→get_widget("btn_nuevoUsuario", this→botonNuevo);
29    refBuilder→get_widget("btn_eliminarUsuario", this→botonEliminar);
30    refBuilder→get_widget("btn_modificarUsuario", this→botonModificar);
31    refBuilder→get_widget("btn_volver", this→botonVolver);
32
33    this→grid→attach(this→tree, 0, 0, 1, 1);
34
35
36    //Create the Tree model:
37
38     this→listaUsuarios = Gtk::ListStore::create(m_Columns);
39     this→tree.set_model(this→listaUsuarios);
40
41    // Actualizo la lista de usuarios existentes
42    this→monitor→getReceptor()→stop();
43    this→monitor→getUsuarios();
44
45
46
47
48    // Cargo la lisa de usuarios a la lista que se muestra por pantalla.
49      //Fill the TreeView's model
50
51       for (size_t i = 0; i < (this→monitor→usuarios.tamanio()); i++) {
52        Gtk::TreeModel::Row row = *(this→listaUsuarios→append());
53        row[m_Columns.m_col_name] =  this→monitor→usuarios[i];
54       }
55
56      //Establacemos el titulo de a columna a mostrar
57      this→tree.append_column("Usuario         |", m_Columns.m_col_name);
58
59
60    // Acciones
61    // Acciones -> Bontones
62
63
64      this→botonEliminar→set_sensitive( false );
65      this→seleccionado = this→tree.get_selection();
66
```

```
67    this→botonNuevo→signal_clicked().connect(sigc::mem_fun(*this, &MenuUsuarios:
      :on_buttonNuevo_clicked));
68    this→botonEliminar→signal_clicked().connect(sigc::mem_fun(*this, &MenuUsuari
      os::on_buttonEliminar_clicked));
69    this→botonModificar→signal_clicked().connect(sigc::mem_fun(*this, &MenuUsuar
      ios::on_buttonModificar_clicked));
70    this→botonVolver→signal_clicked().connect(sigc::mem_fun(*this, &MenuUsuarios
      ::on_buttonVolver_clicked));
71
72    this→seleccionado→signal_changed().connect( sigc::mem_fun(*this, &MenuUsuari
      os::on_selection_changed) );
73
74
75    main→show_all_children();
76
77  }
78
79
80  void MenuUsuarios::on_buttonNuevo_clicked() {
81
82    this→main→set_sensitive(false);
83    FormUsuario ventanaDeEdicion(this→monitor);
84    ventanaDeEdicion.correr();
85    this→main→set_sensitive(true);
86    size_t pos = this→monitor→usuarios.tamanio();
87    Gtk::TreeModel::Row row  = *(this→listaUsuarios→append());
88    row[m_Columns.m_col_name] = this→monitor→usuarios[pos-1]; // ingreso el ulti
      mo agregado a la vista
89
90  }
91
92
93  void MenuUsuarios::on_buttonEliminar_clicked() {
94
95    this→main→set_sensitive(false);
96    Gtk::TreeModel::iterator store_iter = this→seleccionado→get_selected();
97    string aBorrar;
98    if(store_iter) {
99
100     Gtk::TreeModel::Row row = *store_iter;
101
102       aBorrar = row[m_Columns.m_col_name];
103
104   }
105
106   FormConfirmacion ventanaConfirmacion(this→monitor, aBorrar);
107   int resultado = ventanaConfirmacion.correr();
108
109   if(resultado ≡ 1) { //Borro solo si sale por aceptar
110     this→listaUsuarios→erase(store_iter);
111     this→monitor→usuarios.eliminar(aBorrar);
112   }
113   this→main→set_sensitive(true);
114
115   }
116
117
118
119 void MenuUsuarios::on_buttonModificar_clicked() {
120
121
122   Gtk::TreeModel::iterator store_iter = this→seleccionado→get_selected();
123   string aModificar;
124   if(store_iter) {
125
126     Gtk::TreeModel::Row row = *store_iter;
```



```
127
128       aModificar = row[m_Columns.m_col_name];
129
130   }
131   this→main→set_sensitive(false);
132
133   ModificarUsuario ventanaModificacion(this→monitor, aModificar);
134   ventanaModificacion.correr();
135   this→main→set_sensitive(true);
136
137
138
139
140 }
141
142 void MenuUsuarios::on_buttonVolver_clicked() {
143   this→monitor→getReceptor()→start();
144   this→main→hide();
145 }
146
147 void MenuUsuarios::on_selection_changed() {
148
149       this→botonEliminar→set_sensitive(
150     this→seleccionado→count_selected_rows() > 0 );
151 }
152
153
154 void MenuUsuarios::correr() {
155   this→main→set_sensitive(true);
156   Gtk::Main::run(*main);
157
158 }
159
160 MenuUsuarios::~MenuUsuarios() { }
161
```

```
//
//  monitor_interfaz_principal.h
//  CLASE MENUPRINCIPAL
//


#ifndef MENUPRINCIPAL_H_
#define MENUPRINCIPAL_H_


#include "gtkmm.h"
#include "common_thread.h"
#include "monitor_monitor.h"
#include "monitor_configuracion.h"



class MenuPrincipal : public Thread {
private:

  // Atributos de la interfaz
  Gtk::Window *main;         // Ventana Conexion

  Gtk::Label  *estado;       // Estado del servidor
  Gtk::Label  *clientesConectados;
  Gtk::Label  *carpetasActivas;


  Gtk::Button *botonConfiguracion; // Botón Salir
  Gtk::Button *botonSalir;

  //Atributos del menu
  //Monitor
  Gtk::ImageMenuItem *menuConfiguracion;
  Gtk::ImageMenuItem *menuSalir;
  //Opciones
  Gtk::ImageMenuItem *menuAdminUsers;
  Gtk::ImageMenuItem *menuEstadisticas;

  //Ayuda
  Gtk::ImageMenuItem *menuManualUsuario;



  // Atributos del modelo

  Monitor *monitor;
  Configuracion *serverConfig;

public:

  // Constructor
  MenuPrincipal(Monitor *monitor, Configuracion *config);

  // Destructor
  virtual ~MenuPrincipal();

  // Inicia la ejecución de la ventana
  void correr();


  // Define tareas a ejecutar en el hilo.
  virtual void run();

protected:
  // Acciones de botones
```



```

  void on_buttonConfiguracion_clicked();
  void on_buttonSalir_clicked();

  // Acciones del menu
  void on_menuConfiguracion_activate();
  void on_menuSalir_activate();
  void on_menuAdminUsers_activate();
  void on_menuEstadisticas_activate();

  void on_menuManualUsuario_activate();

};

#endif /* MENUPRINCIPAL_H_ */
```

```
1   #include <iostream>
2   #include <string>
3   #include "monitor_interfaz_principal.h"
4   #include "monitor_interfaz_configuracion.h"
5   #include "monitor_interfaz_usuarios.h"
6   #include "monitor_interfaz_estadisticas.h"
7   #include "common_convertir.h"
8
9
10
11  MenuPrincipal::MenuPrincipal(Monitor *monitor, Configuracion *config) : monitor(
    monitor), serverConfig(config) {
12    // Cargamos la ventana
13    Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
14
15
16    // Cargamos elementos
17    refBuilder→add_from_file("./interfaz/monitor_principal.glade");
18    refBuilder→get_widget("main", this→main); // linkeo el form
19
20    // Etiquetas
21    refBuilder→get_widget("lblEstado", this→estado);
22    refBuilder→get_widget("lblClientesConectados", this→clientesConectados);
23    refBuilder→get_widget("lblCarpetasActivas", this→carpetasActivas);
24
25    // Botones
26
27    refBuilder→get_widget("btnConfiguracion", this→botonConfiguracion);
28    refBuilder→get_widget("btnSalir", this→botonSalir);
29
30    // Menu
31    refBuilder→get_widget("mi_configuracion", this→menuConfiguracion);
32    refBuilder→get_widget("mi_salir", this→menuSalir);
33    refBuilder→get_widget("mi_admUsers", this→menuAdminUsers);
34
35
36    refBuilder→get_widget("mi_estadisticas", this→menuEstadisticas);
37
38
39
40    refBuilder→get_widget("mi_manual", this→menuManualUsuario);
41
42
43    // Acciones
44    // Acciones -> Bontones
45
46    this→botonConfiguracion→signal_clicked().connect(sigc::mem_fun(*this, &MenuP
    rincipal::on_buttonConfiguracion_clicked));
47    this→botonSalir→signal_clicked().connect(sigc::mem_fun(*this, &MenuPrincipal
    ::on_buttonSalir_clicked));
48
49    // Acciones -> Menu
50
51    this→menuConfiguracion→signal_activate().connect(sigc::mem_fun(*this, &MenuP
    rincipal::on_menuConfiguracion_activate));
52    this→menuSalir→signal_activate().connect(sigc::mem_fun(*this, &MenuPrincipal
    ::on_menuSalir_activate));
53    this→menuAdminUsers→signal_activate().connect(sigc::mem_fun(*this, &MenuPrin
    cipal::on_menuAdminUsers_activate));
54    this→menuEstadisticas→signal_activate().connect(sigc::mem_fun(*this, &MenuPr
    incipal::on_menuEstadisticas_activate));
55    this→menuManualUsuario→signal_activate().connect(sigc::mem_fun(*this, &MenuP
    rincipal::on_menuManualUsuario_activate));
56
57    main→show_all_children();
58
```



```
59  }
60
61
62  void MenuPrincipal::on_buttonSalir_clicked() {
63
64    Gtk::Main::quit();
65  }
66
67  void MenuPrincipal::on_buttonConfiguracion_clicked() {
68    this→main→set_sensitive(false);
69    IConfiguracion ventanaConfiguracion(this→serverConfig,1);
70    ventanaConfiguracion.correr();
71    this→main→set_sensitive(true);
72  }
73
74  // Acciones del menu
75  void MenuPrincipal::on_menuConfiguracion_activate() {
76    this→main→set_sensitive(false);
77    IConfiguracion ventanaConfiguracion(this→serverConfig,this→monitor→getEstado
    Conexion());
78    ventanaConfiguracion.correr();
79    this→main→set_sensitive(true);
80  }
81
82
83  void MenuPrincipal::on_menuAdminUsers_activate() {
84
85    this→main→set_sensitive(false);
86    MenuUsuarios ventanaUsuarios(this→monitor);
87    ventanaUsuarios.correr();
88    this→main→set_sensitive(true);
89  }
90
91
92  void MenuPrincipal::on_menuEstadisticas_activate(){
93    this→main→set_sensitive(false);
94    MenuEstadisticas estadisticas(this→monitor);
95    estadisticas.correr();
96    this→main→set_sensitive(true);
97  }
98
99
100 void MenuPrincipal::on_menuManualUsuario_activate(){}
101
102 void MenuPrincipal::on_menuSalir_activate() {
103
104   std::cout << "Monitoreo detenido"<< std::endl;
105   Gtk::Main::quit();
106 }
107
108 void MenuPrincipal::correr(){
109   this→main→set_sensitive(true);
110   this→start();
111   Gtk::Main::run(*main);
112
113
114 }
115
116 void MenuPrincipal::run() {
117   this→monitor→actualizarValores();
118   while(this→isActive()) {
119     this→monitor→actualizarValores();
120     if (this→monitor→getEstadoConexion() ≡ true) {
121       this→estado→set_text("Conectado");
122       this→menuEstadisticas→set_sensitive(true);
123       this→menuAdminUsers→set_sensitive(true);
```

jun 25, 13 13:44 **monitor_interfaz_principal.cpp** Page 3/3

```
124         this→clientesConectados→set_text(monitor→getClientesConectados());
125         this→carpetasActivas→set_text(monitor→getCarpetasActivas());
126       }
127       if ((this→monitor→getEstadoConexion()) ≡ false) {
128         this→menuEstadisticas→set_sensitive(false);
129         this→menuAdminUsers→set_sensitive(false);
130         this→estado→set_text("Desconectado");
131         this→clientesConectados→set_text("-");
132         this→carpetasActivas→set_text("-");
133       }
134       this→sleep(2);
135     }
136   }
137
138
139   MenuPrincipal::~MenuPrincipal() { }
```

jun 25, 13 13:44 **monitor_interfaz_modificarUsuario.h** Page 1/1

```
1   //
2   //  monitor_interfaz_modificarUsuario.h
3   //  CLASE MODIFICARUSUARIO
4   //
5
6
7   #ifndef MODIFICARUSUARIO_H_
8   #define MODIFICARUSUARIO_H_
9
10
11  #include "gtkmm.h"
12
13  #include "monitor_monitor.h"
14
15
16
17
18  class ModificarUsuario {
19  private:
20
21    // Atributos de la interfaz
22    Gtk::Window *main;
23
24    Gtk::Button *botonGuardar;
25    Gtk::Button *botonCancelar;
26    Gtk::Label  *labelError;
27
28    Gtk::Entry  *usuarioTxt;    // Textbox de nombre de usuario
29    Gtk::Entry  *passTxt;     // Textbox de la contraseña de usuario
30
31    Monitor *monitor;
32    string modificar;
33
34
35
36  public:
37
38    // Constructor
39    ModificarUsuario(Monitor *monitor, string aModificar);
40
41    // Destructor
42    virtual ~ModificarUsuario();
43
44    // Inicia la ejecución de la ventana
45    void correr();
46
47  protected:
48    // Acciones de botones
49
50    void on_buttonGuardar_clicked();
51    void on_buttonCancelar_clicked();
52
53  };
54
55  #endif /* FORMUSUARIO_H_ */
```

```
1   #include <iostream>
2   #include <string>
3   #include "monitor_interfaz_principal.h"
4   #include "common_lista.h"
5   #include "common_parser.h"
6   #include "monitor_interfaz_usuarios.h"
7   #include "common_protocolo.h"
8   #include "common_convertir.h"
9   #include "common_hash.h"
10  #include "monitor_interfaz_modificarUsuario.h"
11
12
13
14
15  ModificarUsuario::ModificarUsuario(Monitor *monitor, string aModificar) {
16    // Cargamos la ventana
17    Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
18
19    this→monitor = monitor;
20    this→modificar = aModificar;
21
22    // Cargamos elementos
23    refBuilder→add_from_file("./interfaz/monitor_formUsuario.glade");
24    refBuilder→get_widget("main", this→main); // linkeo el form
25
26
27    refBuilder→get_widget("lbl_error", this→labelError);
28    refBuilder→get_widget("btn_guardar", this→botonGuardar);
29    refBuilder→get_widget("btn_cancelar", this→botonCancelar);
30
31    refBuilder→get_widget("txt_contrasenia", this→passTxt);
32    refBuilder→get_widget("txt_usuario",  this→usuarioTxt);
33
34    // Acciones
35    // Acciones -> Bontones
36
37    this→botonGuardar→signal_clicked().connect(sigc::mem_fun(*this, &ModificarUs
    uario::on_buttonGuardar_clicked));
38    this→botonCancelar→signal_clicked().connect(sigc::mem_fun(*this, &ModificarU
    suario::on_buttonCancelar_clicked));
39
40    this→usuarioTxt→set_text(modificar);
41    this→usuarioTxt→set_sensitive(false);
42
43
44    main→show_all_children();
45
46  }
47
48
49  void ModificarUsuario::on_buttonGuardar_clicked() {
50
51    string msg;
52    string res;
53    string user;
54    string pass;
55
56    pass = this→passTxt→get_text();
57    user = this→usuarioTxt→get_text();
58
59
60    this→labelError→set_visible(false);
61
62
63    if ((pass.compare("") ≡ 0)) {
64
```



```
65      this→labelError→set_visible(false);
66      msg.clear();
67      msg = "contraseña inválida";
68      this→labelError→set_text(msg);
69      this→labelError→set_visible(true);
70      return;
71
72    }
73
74    this→labelError→set_visible(false);
75
76    res.append(M_SERVER_MODIFY_USER_REQUEST);
77    res.append(" ");
78
79    res.append(user);
80    res.append(COMMON_DELIMITER);
81    res.append(Hash::funcionDeHash(pass));
82
83    this→monitor→getReceptor()→enviarMensaje(res);
84    this→main→hide();
85
86  }
87
88
89
90
91
92  void ModificarUsuario::on_buttonCancelar_clicked() {
93
94    this→main→hide();
95  }
96
97  void ModificarUsuario::correr(){
98    this→main→set_sensitive(true);
99    Gtk::Main::run(*main);
100
101 }
102
103 ModificarUsuario::~ModificarUsuario() { }
```

jun 25, 13 13:44 **monitor_interfaz_formUsuario.h** Page 1/1

```
1  //
2  //  monitor_interfaz_formUsuario.h
3  //  CLASE FORMUSUARIO
4  //
5
6
7  #ifndef FORMUSUARIO_H_
8  #define FORMUSUARIO_H_
9
10
11 #include "gtkmm.h"
12
13 #include "monitor_monitor.h"
14
15
16
17
18 class FormUsuario {
19 private:
20
21   // Atributos de la interfaz
22   Gtk::Window *main;
23
24   Gtk::Button *botonGuardar;
25   Gtk::Button *botonCancelar;
26   Gtk::Label  *labelError;
27
28
29   Gtk::Entry  *usuarioTextBox;    // Textbox de nombre de usuario
30   Gtk::Entry  *passTextBox;     // Textbox de la contraseña de usuario
31
32   Monitor *monitor;
33
34
35 public:
36
37   // Constructor
38   FormUsuario(Monitor *monitorm);
39
40   // Destructor
41   virtual ~FormUsuario();
42
43   // Inicia la ejecución de la ventana
44   void correr();
45
46 protected:
47   // Acciones de botones
48
49   void on_buttonGuardar_clicked();
50   void on_buttonCancelar_clicked();
51
52 };
53
54 #endif /* FORMUSUARIO_H_ */
```

jun 25, 13 13:44 **monitor_interfaz_formUsuario.cpp** Page 1/2

```
1  #include <iostream>
2  #include <string>
3  #include "monitor_interfaz_principal.h"
4
5  #include "monitor_interfaz_usuarios.h"
6  #include "common_protocolo.h"
7  #include "common_convertir.h"
8  #include "monitor_interfaz_formUsuario.h"
9  #include "common_hash.h"
10
11
12
13
14 FormUsuario::FormUsuario(Monitor *monitor) {
15   // Cargamos la ventana
16   Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
17
18   this→monitor = monitor;
19   // Cargamos elementos
20   refBuilder→add_from_file("./interfaz/monitor_formUsuario.glade");
21   refBuilder→get_widget("main", this→main); // linkeo el form
22
23
24   refBuilder→get_widget("lbl_error", this→labelError);
25   refBuilder→get_widget("btn_guardar", this→botonGuardar);
26   refBuilder→get_widget("btn_cancelar", this→botonCancelar);
27   refBuilder→get_widget("txt_contrasenia", this→passTextBox);
28   refBuilder→get_widget("txt_usuario",  this→usuarioTextBox);
29
30
31
32   // Acciones
33   // Acciones -> Bontones
34
35   this→botonGuardar→signal_clicked().connect(sigc::mem_fun(*this, &FormUsuario
     ::on_buttonGuardar_clicked));
36   this→botonCancelar→signal_clicked().connect(sigc::mem_fun(*this, &FormUsuari
     o::on_buttonCancelar_clicked));
37
38   // Acciones -> Menu
39
40   main→show_all_children();
41
42 }
43
44
45 void FormUsuario::on_buttonGuardar_clicked() {
46
47   string msg;
48   string res;
49   string user = this→usuarioTextBox→get_text();
50   string pass = this→passTextBox→get_text();
51
52   // recorro la lista de usuarios y comparo si hay alguno igual a lo
53   // guardado en el textbox si hay uno igual..cambio el estado de una label
54
55   this→labelError→set_visible(false);
56   int flag = 0;
57
58   if ((user.compare("") ≡ 0) ∨  (user.compare(" ") ≡ 0) ∨ (pass.compare("") ≡ 0)
     ) {
59
60     this→labelError→set_visible(false);
61     msg.clear();
62     msg = "Contenido Inválido";
63     this→labelError→set_text(msg);
```

jun 25, 13 13:44 **monitor_interfaz_formUsuario.cpp** Page 2/2

```
64      this→labelError→set_visible(true);
65      return;
66
67    }
68
69    this→labelError→set_visible(false);
70    for (size_t i = 0; i < (this→monitor→usuarios.tamanio()); i++) {
71
72      if (user.compare(this→monitor→usuarios[i]) ≡ 0) {
73        msg = "El usuario ingresado ya existe";
74        this→labelError→set_text(msg);
75        this→labelError→set_visible(true);
76        flag = 1;
77        break;
78      }
79
80    }
81    if (flag ≡ 1) return;
82
83    res.append(M_SERVER_NEW_USER_INFO);
84    res.append(" ");
85    res.append(user);
86    res.append(COMMON_DELIMITER);
87    res.append(Hash::funcionDeHash(pass));
88    this→monitor→usuarios.insertarUltimo(user);
89    this→monitor→getReceptor()→enviarMensaje(res);
90    this→main→hide();
91  }
92
93
94
95
96
97  void FormUsuario::on_buttonCancelar_clicked() {
98
99    this→main→hide();
100 }
101
102 void FormUsuario::correr(){
103   this→main→set_sensitive(true);
104   Gtk::Main::run(*main);
105
106 }
107
108 FormUsuario::~FormUsuario() { }
```

jun 25, 13 13:44 **monitor_interfaz_estadisticas.h** Page 1/1

```
1   //
2   //  monitor_interfaz_menuEstadisticas.h
3   //  CLASE MENUESTADISTICAS
4   //
5
6
7   #ifndef MENUESTADISTICAS_H_
8   #define MENUESTADISTICAS_H_
9
10
11  #include "gtkmm.h"
12  #include "monitor_graficador.h"
13  #include "monitor_monitor.h"
14  #include "monitor_configuracion.h"
15
16
17  class MenuEstadisticas : public Gtk::Window {
18  private:
19
20    // Atributos de la interfaz
21    Gtk::Window *main;          // Ventana Conexion
22
23
24    // Atributos del modelo
25    Graficador c;
26    Monitor *monitor;
27
28  public:
29
30    // Constructor
31    MenuEstadisticas(Monitor *monitor);
32
33    // Destructor
34    virtual ~MenuEstadisticas();
35
36    // Inicia la ejecución de la ventana
37    void correr();
38
39  };
40
41  #endif /* MENUESTADISTICAS_H_ */
```

jun 25, 13 13:44 **monitor_interfaz_estadisticas.cpp** Page 1/1

```cpp
#include <iostream>
#include <string>
#include "monitor_interfaz_estadisticas.h"



MenuEstadisticas::MenuEstadisticas(Monitor *monitor) {

  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();


  // Cargamos elementos
  refBuilder→add_from_file("./interfaz/monitor_estadisticas.glade");
  refBuilder→get_widget("main", this→main); // linkeo el form en blanco donde se muestra el grafico

  main→show_all_children();
  this→monitor = monitor;
  this→c.monitor = monitor;
}

void MenuEstadisticas::correr() {

  //cargamos drawing(graficador) a la ventana(main) se corre el hilo del graficador
  c.show_all();
  main→add(c);

  c.correr();
  Gtk::Main::run(*main);
  c.detener();
  main→hide();
}

MenuEstadisticas::~MenuEstadisticas() { }
```

jun 25, 13 13:44 **monitor_interfaz_eliminarUsuario.h** Page 1/1

```cpp
//
//  monitor_interfaz_eliminarUsuario.h
//  CLASE FORMCONFIRMACION
//


#ifndef FORMCONFIRMACION_H_
#define FORMCONFIRMACION_H_


#include "gtkmm.h"
#include <string>
#include "monitor_monitor.h"



class FormConfirmacion {
private:

  // Atributos de la interfaz
  Gtk::Window *main;

  Gtk::Button *botonAceptar;
  Gtk::Button *botonCancelar;
  Gtk::Label  *mensaje;
  // Atributos del modelo
  Monitor *monitor;
  string borrar;
  int seleccion;

public:

  // Constructor
  FormConfirmacion(Monitor *monitor, string borrar);

  // Destructor
  virtual ~FormConfirmacion();

  // Inicia la ejecución de la ventana
  int correr();

protected:
  // Acciones de botones

  void on_buttonAceptar_clicked();
  void on_buttonCancelar_clicked();

};

#endif /* FORMUSUARIO_H_ */
```

jun 25, 13 13:44 **monitor_interfaz_eliminarUsuario.cpp** Page 1/2

```
1  #include <iostream>
2
3  #include "monitor_interfaz_usuarios.h"
4  #include "common_protocolo.h"
5  #include "common_convertir.h"
6  #include "monitor_interfaz_eliminarUsuario.h"
7
8
9
10
11 FormConfirmacion::FormConfirmacion(Monitor *monitor, string borrar ) {
12   // Cargamos la ventana
13   Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
14
15   this→monitor = monitor;
16   this→borrar = borrar;
17   this→seleccion = 0;
18
19   // Cargamos elementos
20   refBuilder→add_from_file("./interfaz/monitor_confirmarEliminacion.glade");
21   refBuilder→get_widget("main", this→main); // linkeo el form
22
23
24   refBuilder→get_widget("lbl_msg", this→mensaje);
25   refBuilder→get_widget("btn_aceptar", this→botonAceptar);
26   refBuilder→get_widget("btn_cancelar", this→botonCancelar);
27
28
29   // Acciones
30   // Acciones -> Bontones
31
32   this→botonAceptar→signal_clicked().connect(sigc::mem_fun(*this, &FormConfirm
     acion::on_buttonAceptar_clicked));
33   this→botonCancelar→signal_clicked().connect(sigc::mem_fun(*this, &FormConfir
     macion::on_buttonCancelar_clicked));
34
35   // preparamos la ventana
36   string lbl;
37   this→mensaje→set_visible(false);
38   lbl.append("Se eliminará al usuario: ");
39   lbl.append(this→borrar);
40   this→mensaje→set_text(lbl);
41   this→mensaje→set_visible(true);
42
43   main→show_all_children();
44
45 }
46
47 void FormConfirmacion::on_buttonAceptar_clicked() {
48   string msg;
49   msg.append(M_SERVER_DELETE_USER);
50   msg.append(" ");
51   msg.append(this→borrar);
52
53   this→monitor→getReceptor()→enviarMensaje(msg);
54   this→seleccion = 1;
55   this→main→hide();
56 }
57 void FormConfirmacion::on_buttonCancelar_clicked() {
58   this→seleccion = 0;
59   this→main→hide();
60 }
61
62 int FormConfirmacion::correr() {
63   this→main→set_sensitive(true);
64   Gtk::Main::run(*main);
```

jun 25, 13 13:44 **monitor_interfaz_eliminarUsuario.cpp** Page 2/2

```
65   return this→seleccion;
66 }
67
68 FormConfirmacion::~FormConfirmacion() { }
```

jun 25, 13 13:44 **monitor_interfaz_configuracion.h** Page 1/1

```
1  //
2  //  monitor_interfaz_configuracion.h
3  //  CLASE INTERFAZ DE CONFIGURACION
4  //
5
6
7  #ifndef ICONFIGURACION_H_
8  #define ICONFIGURACION_H_
9
10
11 #include "monitor_configuracion.h"
12 #include "gtkmm.h"
13
14
15
16
17 class IConfiguracion : public Gtk::Window {
18 private:
19
20   // Atributos de la interfaz
21   Gtk::Window* main;
22
23   Gtk::Button *botonGuardar;
24   Gtk::Button *botonCancelar;
25
26   Gtk::Entry  *puerto;
27   Gtk::Entry  *host;
28   Gtk::Entry  *tiempo;
29   Configuracion* config;
30   int flag; // Indica si el servidor esta conectado
31
32
33 public:
34
35   // Constructor
36   IConfiguracion(Configuracion *config, bool flag);
37
38   // Destructor
39   virtual ~IConfiguracion();
40
41   // Inicia la ejecución de la ventana
42   void correr();
43
44 protected:
45
46   void on_buttonGuardar_clicked();
47   void on_buttonCancelar_clicked();
48 };
49
50 #endif /* ICONFIGURACION_H_ */
```

jun 25, 13 13:44 **monitor_interfaz_configuracion.cpp** Page 1/2

```
1  #include <iostream>
2  #include <string>
3  #include "monitor_interfaz_configuracion.h"
4  #include "common_convertir.h"
5
6
7
8
9  IConfiguracion::IConfiguracion(Configuracion *config, bool flag) {
10   // Cargamos la ventana
11   Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
12
13
14
15   // Cargamos elementos
16   refBuilder->add_from_file("./interfaz/monitor_configuracion.glade");
17
18
19   refBuilder->get_widget("main", this->main); // linkeo el form
20
21
22   refBuilder->get_widget("puerto", this->puerto);
23   refBuilder->get_widget("host", this->host);
24   refBuilder->get_widget("tiempo", this->tiempo);
25
26   refBuilder->get_widget("guardar", this->botonGuardar);
27   refBuilder->get_widget("cancelar", this->botonCancelar);
28
29
30   this->botonGuardar->signal_clicked().connect(sigc::mem_fun(*this, &IConfigurac
     ion::on_buttonGuardar_clicked));
31   this->botonCancelar->signal_clicked().connect(sigc::mem_fun(*this, &IConfigura
     cion::on_buttonCancelar_clicked));
32
33   this->flag = flag;
34
35   main->show_all_children();
36 }
37
38
39 void IConfiguracion::on_buttonGuardar_clicked() {
40
41   //obtengo cada valor almacenado en los textBox
42
43   string unPuerto = this->puerto->get_text();
44   string unHost = this->host->get_text();
45   string unTiempo = this->tiempo->get_text();
46
47   this->config->guardarCambios(unPuerto, unHost, unTiempo);
48
49   this->main->hide();
50
51 }
52
53 void IConfiguracion::on_buttonCancelar_clicked() {
54   // No hago nada, retorno sin cambios en el archivo de settings.
55   this->main->hide();
56
57 }
58
59
60 void IConfiguracion::correr() {
61
62   //cargo los textBox con info
63   string auxPuerto = Convertir::itos(this->config->obtenerPuerto());
64   string auxHost = this->config->obtenerHost();
```

jun 25, 13 13:44 **monitor_interfaz_configuracion.cpp** Page 2/2

```
65    string auxTiempo = Convertir::itos(this→config→obtenerTiempo());
66
67    this→puerto→set_text(auxPuerto);
68    this→host→set_text(auxHost);
69    this→tiempo→set_text(auxTiempo);
70
71    if (this→flag ≡ true) {
72      this→tiempo→set_sensitive(false);
73      this→puerto→set_sensitive(false);
74      this→host→set_sensitive(false);
75
76    }
77    //Muestro configuracion actual
78    Gtk::Main::run(*main);
79
80
81  }
82
83  IConfiguracion::~IConfiguracion() { }
```

jun 25, 13 13:44 **monitor_interfaz_conexion.h** Page 1/1

```
1   //
2   //  monitor_interfaz_conexion.h
3   //  CLASE CONEXION
4   //
5
6
7   #ifndef CONEXION_H_
8   #define CONEXION_H_
9
10
11  #include "gtkmm.h"
12  #include "monitor_receptorDatos.h"
13  #include "monitor_configuracion.h"
14  #include "monitor_interfaz_configuracion.h"
15
16
17
18  class Conexion : public Gtk::Window {
19  private:
20
21    // Atributos de la interfaz
22    Gtk::Window* main;        // Ventana Conexion
23
24    Gtk::Label  *lblError;      // Etiqueta de error
25    Gtk::Button *botonConectar;   // Botón Conectar
26    Gtk::Button *botonSalir;    // Botón Salir
27    Gtk::Entry  *usuarioTextBox;    // Textbox de nombre de usuario
28    Gtk::Entry  *passTextBox;   // Textbox de la contraseña de usuario
29
30    //Atributos del menu
31    Gtk::ImageMenuItem *menuPref;
32    Gtk::ImageMenuItem *menuSalir;
33
34
35    // Atributos del modelo
36    Receptor *receptor;         // Cliente a través del cual se conecta
37    Configuracion *receptorConfig;
38    int estadoConexion;
39    int flagSalida;
40
41  public:
42
43    // Constructor
44    Conexion(Receptor *receptor, Configuracion* receptorConfig);
45
46    // Destructor
47    virtual ~Conexion();
48
49    // Inicia la ejecución de la ventana
50    int correr();
51
52  protected:
53
54    void on_buttonConectar_clicked();
55    void on_buttonSalir_clicked();
56    void on_menuPref_activate();
57    void on_menuSalir_activate();
58
59  };
60
61  #endif /* CONEXION_H_ */
```

```
#include <iostream>
#include <string>
#include "monitor_configuracion.h"
#include "monitor_interfaz_conexion.h"



Conexion::Conexion(Receptor *receptor, Configuracion* receptorConfig) : receptor
(receptor), receptorConfig(receptorConfig) {
  // Cargamos la ventana
  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();


  // Cargamos elementos
  refBuilder→add_from_file("./interfaz/monitor_conexion.glade");


  refBuilder→get_widget("conexion", this→main); // linkeo el form

  refBuilder→get_widget("usuarioTxt", this→usuarioTextBox);
  refBuilder→get_widget("passTxt", this→passTextBox);

  refBuilder→get_widget("conectar", this→botonConectar);
  refBuilder→get_widget("lblError", this→lblError);
  refBuilder→get_widget("preferencias", this→menuPref);
  refBuilder→get_widget("msalir", this→menuSalir);
  refBuilder→get_widget("Salir", this→botonSalir);

  this→botonConectar→signal_clicked().connect(sigc::mem_fun(*this, &Conexion::
on_buttonConectar_clicked));

  this→botonSalir→signal_clicked().connect(sigc::mem_fun(*this, &Conexion::on_
buttonSalir_clicked));
  this→menuPref→signal_activate().connect(sigc::mem_fun(*this, &Conexion::on_m
enuPref_activate));
  this→menuSalir→signal_activate().connect(sigc::mem_fun(*this, &Conexion::on_
menuSalir_activate));

  main→show_all_children();
  this→flagSalida = 0;
  this→estadoConexion = 0;

}


void Conexion::on_buttonConectar_clicked() {
  // Deshabilitamos objetos de la ventana
  this→botonConectar→set_sensitive(false);
  this→usuarioTextBox→set_sensitive(false);
  this→passTextBox→set_sensitive(false);

  // Obtenemos la configuracion actual del servidor

  receptor→especificarNombreHost(this→receptorConfig→obtenerHost());
  receptor→especificarPuerto(this→receptorConfig→obtenerPuerto());
  receptor→especificarTiempo(this→receptorConfig→obtenerTiempo());

  std::string user = this→usuarioTextBox→get_text();
  std::string pass = this→passTextBox→get_text();

  // Iniciamos conexión
  this→estadoConexion = receptor→conectar(user, pass);

  if(this→estadoConexion ≡ 1) {

    this→main→set_sensitive(false);
```



```
    this→main→hide();

    // se puede incluir una ventana de actualizacion
  }
  else if(this→estadoConexion ≡ 0) {
    // Mostramos mensaje de error en ventana
    this→lblError→set_text("Usuario y/o contraseña inválidos");
    this→lblError→set_visible(true);

    // Borramos el contenido del password para ser nuevamente escrito
    this→passTextBox→set_text("");

    // Habilitamos objetos de la ventana
    this→botonConectar→set_sensitive(true);
    this→usuarioTextBox→set_sensitive(true);
    this→passTextBox→set_sensitive(true);
  }
  else if(this→estadoConexion ≡ -1) {
    // Mostramos mensaje de error en ventana
    this→lblError→set_text("Falló la conexión con el servidor.");
    this→lblError→set_visible(true);

    // Habilitamos objetos de la ventana
    this→botonConectar→set_sensitive(true);
    this→usuarioTextBox→set_sensitive(true);
    this→passTextBox→set_sensitive(true);
  }
}

void Conexion::on_buttonSalir_clicked() {
  Gtk::Main::quit();
  this→flagSalida = 1;
}

void Conexion::on_menuPref_activate() {
  IConfiguracion ventanaSettings(this→receptorConfig, this→estadoConexion);
  ventanaSettings.correr();
  this→main→set_sensitive(true);

}


void Conexion::on_menuSalir_activate() {
  Gtk::Main::quit();
  this→flagSalida = 1;

}


int Conexion::correr() {
  Gtk::Main::run(*main);
  return this→flagSalida;
}

Conexion::~Conexion() { }
```

jun 25, 13 13:44 **monitor_graficador.h** Page 1/1

```
1  #ifndef GRAFICADOR_H_
2  #define GRAFICADOR_H_
3
4  #include <gtkmm.h>
5  #include <gtkmm/drawingarea.h>
6  #include <list>
7  #include <iostream>
8  #include <map>
9  #include <sigc++/signal.h>
10
11 #include "monitor_vista.h"
12 #include "common_thread.h"
13 #include <cairomm/surface.h>
14 #include <cairomm/context.h>
15 #include <cairomm/refptr.h>
16 #include "monitor_monitor.h"
17 #include "monitor_vistaIndicador.h"
18 using namespace std;
19
20 class Graficador : public Gtk::DrawingArea , Thread {
21 private:
22   list<Vista*> vistasLinea;
23   Vista* fondo;
24   VistaIndicador* indicador;
25 protected:
26   bool estado;
27   virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
28
29 public:
30   Graficador();
31   virtual ~Graficador();
32   void agregarVista(Vista* v);
33   void quitarVistas();
34   int escalaRequerida(int bytes);
35   virtual void run();
36   void correr();
37   void detener();
38   Monitor *monitor;
39 };
40
41 #endif /*GRAFICADOR_H_*/
```

jun 25, 13 13:44 **monitor_graficador.cpp** Page 1/2

```
1  #include "monitor_graficador.h"
2  #include <iostream>
3  #include "monitor_vista.h"
4  #include "monitor_vista_linea.h"
5  #include "monitor_vistaFondo.h"
6
7  #define B 1
8  #define KB 1024
9  #define MB 1048576
10
11 Graficador::Graficador() {
12   fondo = new VistaFondo();
13 }
14
15 Graficador::~Graficador() {
16   vistasLinea.clear();
17   delete(fondo);
18   delete(indicador);
19 }
20
21 void Graficador::agregarVista(Vista* v) {
22   vistasLinea.push_back(v);
23   v→setGraficador(this);
24   add_events(Gdk::BUTTON_PRESS_MASK);
25 }
26
27 void Graficador::run() {
28
29   // Se selecciona escala de acuerdo a los bytes que ocupados en el servidor
30   int escalaActual = escalaRequerida(monitor→getBytesOcupados());
31
32   VistaLinea* primera = new VistaLinea( 500−10, monitor→getBytesOcupados() , 50
     0, monitor→getBytesOcupados());
33   primera→escala = escalaActual;
34   this→agregarVista(primera);
35
36   indicador = new VistaIndicador(monitor);
37
38   while (estado){
39
40       //si se llena el grafico, borro el primero
41       if (vistasLinea.size() > 50) {
42         vistasLinea.pop_front();
43       }
44       //actualizo la escala actual segun la actual medicion de bytes (B,Kb,MB)
45       escalaActual = escalaRequerida(monitor→getBytesOcupados());
46
47       //corro toda lo dibujado a la izquierda
48       //refresco las lineas actuales a la escala actual
49       list<Vista*>::iterator it;
50       for ( it=vistasLinea.begin() ; it ≠ vistasLinea.end(); it++ ){
51           ((VistaLinea*)(*it))→correrIzquierda(10);
52           ((VistaLinea*)(*it))→escala = escalaActual; // actualizo la escala de
       }
53       //creo la linea para la actual medicion
54       VistaLinea* horizontal = new VistaLinea( 500 − 10, ((VistaLinea*)vistasLin
     ea.back())→yfin , 500, monitor→getBytesOcupados());
55       //cargo la linea en el lienzo
56       this→agregarVista(horizontal);
57       //rompo la pantalla para que se ejecute el evento "on_draw" de nuevo
58       Glib::RefPtr<Gdk::Window> win = get_window();
59       if (win) {
60         Gdk::Rectangle r(0, 0, get_allocation().get_width(), get_allocation().ge
     t_height());
61         win→invalidate_rect(r, false);
62
```

```
63        }
64      sleep(1);
65
66      // En este ciclo cada 1 segundo se crea una nueva linea, se corre y se actua
    liza la escala de las lineas
67      // anteriores y se vuelve a ejectuar el evento ondraw-> se dibuja todo de nu
    evo
68
69    }
70
71
72
73  }
74
75  int Graficador::escalaRequerida(int bytes) {
76
77    if (bytes < KB) return B;
78    if (bytes < MB) return KB;
79    return MB;
80  }
81
82
83  void Graficador::correr(){
84    this→estado = true;
85    this→start();
86
87  }
88  void Graficador::detener(){
89
90    this→estado = false;
91    this→join();
92  }
93  //aca este es el evento ondraw q se ejecuta y este recorre todos los objetos vis
    ta diciendoles "draw"
94  bool Graficador::on_draw(const Cairo::RefPtr<Cairo::Context>& cr) {
95    //el cr (cairorefprt) es donde se puede dibujar
96    //le decimos al fondo y al incador q se dibujen
97    fondo→draw(cr);
98    indicador→draw(cr);
99    //le decimos a todas las lineas q se dibujen
100   list<Vista*>::iterator it;
101   for ( it=vistasLinea.begin() ; it ≠ vistasLinea.end(); it++ ) ((Vista*)(*it))
    →draw(cr);
102
103   return true;
104 }
105
106 void Graficador::quitarVistas() {
107   vistasLinea.clear();
108 }
```



```
1   //
2   //  monitor_configuracion.h
3   //  CLASE CONFIGURACION
4   //
5
6
7   #ifndef CONFIGURACION_H
8   #define CONFIGURACION_H
9
10  #include "common_archivoTexto.h"
11
12  // CONSTANTES
13  namespace {
14
15    // Metadatos sobre el archivo de configuración
16    const std::string CONFIG_DIR = "config/";
17    const std::string CONFIG_FILENAME = "monitor";
18    const std::string CONFIG_FILE_EXT = ".properties";
19
20    // Parámetros configurables
21
22    const std::string CONFIG_P_HOST = "HOSTNAME";
23    const std::string CONFIG_P_PORT = "PUERTO";
24    const std::string CONFIG_P_TIME = "TIME";
25    const std::string CONFIG_LOG = "LOG";
26
27    // Separadores
28    const std::string CONFIG_SEPARATOR = "=";
29
30    // Indicador de comentarios
31    const std::string CONFIG_COMMENT = "#";
32  }
33
34
35
36
37
38  /* ******************************************************************************
39   * DECLARACIÓN DE LA CLASE
40   * ******************************************************************************/
41
42
43  class Configuracion {
44  private:
45    ArchivoTexto* Archivo;
46
47  public:
48
49    // Constructor
50    Configuracion();
51
52    // Destructor
53    ~Configuracion();
54
55    // Devuelve el valor especifico que se necesita
56    std::string getInfo(std :: string &cadena);
57
58    // Devuelve el puerto del servidor al que se quiere conectar.
59    int obtenerPuerto();
60
61    // Devuelve el intervalo de actualizacion del monitor.
62    int obtenerTiempo();
63
64    // Devuelve el host del servidor al que se quiere conectar.
65    string obtenerHost();
66
```

jun 25, 13 13:44 **monitor_configuracion.h** Page 2/2

```
67    //Devuelve el puntero al archivo de Log
68    string obtenerLog();
69
70    // Guarda cambios realizados sobre la configuracion.
71    void guardarCambios(string unPuerto, string unHost, string time);
72
73  };
74
75  #endif
```

jun 25, 13 13:44 **monitor_configuracion.cpp** Page 1/2

```
1   //
2   //  monitor_configuracion.cpp
3   //  CLASE CONFIGURACION
4   //
5
6
7   #include <iostream>
8   #include <string>
9   #include <sstream>
10  #include "common_convertir.h"
11  #include "monitor_configuracion.h"
12
13  using namespace std;
14
15
16
17
18  /* ******************************************************************************
19   * DEFINICIÓN DE LA CLASE
20   * ******************************************************************************/
21
22
23  // Constructor
24  Configuracion::Configuracion() {
25
26  }
27
28  // Destructor
29  Configuracion::~Configuracion() { }
30
31
32  // Devuelve el valor especifico que se necesita
33  std::string Configuracion::getInfo(std :: string &cadena) {
34    string val;
35    unsigned pos = cadena.find(CONFIG_SEPARATOR);
36    val = cadena.substr (pos+1);
37    return val;
38  }
39
40  // Devuelve el puerto del servidor.
41  int Configuracion::obtenerPuerto() {
42    string* cadena = new string();
43    this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME + CONFIG_FILE_E
    XT,0);
44    bool estado = false;
45    while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_PORT)));
46    string result = getInfo(*cadena);
47    delete(this→Archivo);
48    delete(cadena);
49    return Convertir:: stoi(result);
50  }
51
52  //Devuelve el intervalo de actualizacion de informacion del monitor
53  int Configuracion::obtenerTiempo() {
54    string* cadena = new string();
55    this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME + CONFIG_FILE_E
    XT,0);
56    bool estado = false;
57    while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_TIME)));
58    string result = getInfo(*cadena);
59    delete(this→Archivo);
60    delete(cadena);
61    return Convertir:: stoi(result);
62  }
63
64
```

jun 25, 13 13:44 **monitor_configuracion.cpp** Page 2/2

```
65  std::string Configuracion::obtenerHost() {
66    string* cadena = new string();
67    this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
68      CONFIG_FILE_EXT,0);
69    bool estado = false;
70    while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_HOST)));
71    string result = getInfo(*cadena);
72    delete(this→Archivo);
73    delete(cadena);
74    return result;
75  }
76
77  string Configuracion::obtenerLog() {
78
79    string* cadena = new string();
80    this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME + CONFIG_FILE_E
    XT,0);
81    bool estado = false;
82    while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_LOG)));
83    string result = getInfo(*cadena);
84    delete(this→Archivo);
85    delete (cadena);
86
87    return result;
88  }
89
90  void Configuracion::guardarCambios(string puerto, string host, string time) {
91
92
93  this→Archivo = new ArchivoTexto(CONFIG_DIR + CONFIG_FILENAME +
94      CONFIG_FILE_EXT,1);
95
96    string* aux = new string();
97    *aux += "#SETTINGS MONITOR";
98    *aux += '\n';
99    this→Archivo→escribir(*aux);
100   aux→clear();
101   *aux += CONFIG_P_HOST;
102   *aux += CONFIG_SEPARATOR;
103   *aux += host;
104   *aux += '\n';
105   this→Archivo→escribir(*aux);
106
107   aux→clear();
108   *aux += CONFIG_P_PORT;
109   *aux += CONFIG_SEPARATOR;
110   *aux += puerto;
111   *aux += '\n';
112   this→Archivo→escribir(*aux);
113
114   aux→clear();
115   *aux += CONFIG_P_TIME;
116   *aux += CONFIG_SEPARATOR;
117   *aux += time;
118   *aux += '\n';
119   this→Archivo→escribir(*aux);
120
121   delete(this→Archivo);
122   delete(aux);
123
124 }
125
```

jun 25, 13 13:44 **common_utilidades.h** Page 1/1

```
1   //
2   //  common_utilidades.h
3   //  CLASE UTILIDADES
4   //
5
6
7   #ifndef COMMON_UTILIDADES_H
8   #define COMMON_UTILIDADES_H
9
10  #include <string>
11  #include <time.h>
12  #include <cstdlib>
13
14
15
16  class Utilidades {
17  public:
18
19    // Devuelve un string de caracteres alfanumericos aleatorios de
20    // tamanio 'longitud'
21    static void randomString(int longitud, std::string &s);
22  };
23
24
25  #endif /* COMMON_UTILIDADES_H */
```

jun 25, 13 13:44 **common_utilidades.cpp** Page 1/1

```cpp
//
//  common_utilidades.cpp
//  CLASE UTILIDADES
//


#include "common_utilidades.h"



// Devuelve un string de caracteres alfanumericos aleatorios de
// tamanio 'longitud'
void Utilidades::randomString(int longitud, std::string &s) {
  // Se limpia la cadena
  s.clear();

  // Tabla de caracteres posibles
  char alphanumerico[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  // Se randomiza el set
  srand(time(NULL));
  int tamanio = sizeof(alphanumerico) - 1;

  for (int i = 0; i < longitud; ++i)
    s += alphanumerico[rand() % tamanio];
}
```

jun 25, 13 13:44 **common_thread.h** Page 1/2

```cpp
//
//  common_thread.h
//  CLASE THREAD
//
//  Clase que implementa la interfaz para la creación de un hilo de ejecución.
//


#ifndef THREAD_H
#define THREAD_H


#include <pthread.h>




/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


class Thread {
private:

  pthread_t thread;   // Identificador del hilo
  bool status;      // Estado del thread
  bool asleep;      // Sensa si esta dormido el thread

  // Ejecuta el método run().
  // PRE: 'threadID' es un puntero al thread.
  static void* callback(void *threadID);

  // Constructor privado
  Thread(const Thread &c);

public:

  // Constructor
  Thread();

  // Destructor
  virtual ~Thread();

  // Inicia el thread
  virtual void start();

  // Detiene el thread
  virtual void stop();

  // Envía una solicitud de cancelación al hilo, deteniendo abruptamente
  // su ejecución
  virtual void cancel();

  // Bloquea hasta que el hilo finalice su ejecución en caso de estar
  // ejecutandose.
  virtual void join();

  // Define tareas a ejecutar en el hilo.
  virtual void run() = 0;

  // Suspende la ejecución del hilo durante cierto intervalo de tiempo.
  // Puede ser interrumpido llamando al metodo kill().
  virtual void sleep(unsigned int seconds);

  // Interrumpe el sleep
```

jun 25, 13 13:44 **common_thread.h** Page 2/2

```
67    void interruptSleep();
68
69    // Envía una señal al hilo.
70    //virtual void kill();
71
72    // Verifica si el hilo se encuentra activo.
73    // POST: devuelve true si está activo o false en caso contrario.
74    bool isActive();
75  };
76
77  #endif
```

jun 25, 13 13:44 **common_thread.cpp** Page 1/2

```
1   //
2   //  common_thread.cpp
3   //  CLASE THREAD
4   //
5   //  Clase que implementa la interfaz para la creación de un hilo de ejecución.
6   //
7
8
9   #include "common_thread.h"
10  #include <signal.h>
11  #include <time.h>
12
13
14
15
16  /* ******************************************************************************
17   * DEFINICIÓN DE LA CLASE
18   * ******************************************************************************/
19
20
21  // Constructor
22  Thread::Thread() : status(false), asleep(false) { }
23
24
25  // Constructor privado
26  Thread::Thread(const Thread &c) { }
27
28
29  // Destructor
30  Thread::~Thread() { }
31
32
33  // Inicia el hilo
34  void Thread::start() {
35    pthread_create(&this→thread, 0, callback, this);
36  }
37
38
39  // Detiene el hilo
40  void Thread::stop() {
41    this→status = false;
42  }
43
44
45  // Envía una solicitud de cancelación al hilo, deteniendo abruptamente
46  // su ejecución
47  void Thread::cancel() {
48    pthread_cancel(this→thread);
49  }
50
51
52  // Bloquea hasta que el hilo finalice su ejecución en caso de estar
53  // ejecutandose.
54  void Thread::join() {
55    pthread_join(this→thread, 0);
56  }
57
58
59  // Suspende la ejecución del hilo durante cierto intervalo de tiempo.
60  // Puede ser interrumpido llamando al metodo kill().
61  void Thread::sleep(unsigned int seconds) {
62    asleep = true;
63    unsigned int i = 0;
64
65    struct timespec t, t_aux;
66    t.tv_sec = 0;
```

jun 25, 13 13:44 **common_thread.cpp** Page 2/2

```
67    t.tv_nsec = 100000000;
68
69    while (asleep ≡ true ∧ i < (seconds * 10)) {
70      nanosleep(&t, &t_aux);
71      i++;
72    }
73  }
74
75
76  // Interrumpe el sleep
77  void Thread::interruptSleep() {
78    asleep = false;
79  }
80
81
82  // Verifica si el hilo se encuentra activo.
83  // POST: devuelve true si está activo o false en caso contrario.
84  bool Thread::isActive() {
85    return this→status;
86  }
87
88
89  // Ejecuta el método run().
90  // PRE: 'threadID' es un puntero al thread.
91  void* Thread::callback(void *threadID) {
92    ((Thread*)threadID)→status = true;
93    ((Thread*)threadID)→run();
94    ((Thread*)threadID)→status = false;
95    return 0;
96  }
```

jun 25, 13 13:44 **common_socket.h** Page 1/2

```
1   //
2   //  common_socket.h
3   //  CLASE SOCKET
4   //
5   //  Clase que implementa la interfaz de los sockets de flujo (utilizando el
6   //  protocolo TCP), proporcionando un conjunto medianamente extenso de métodos
7   //  y propiedades para las comunicaciones en red.
8   //
9
10
11  #ifndef SOCKET_H
12  #define SOCKET_H
13
14
15  #include <netinet/in.h>
16  #include <string>
17
18
19
20
21
22  /* ************************************************************************
23   * DECLARACIÓN DE LA CLASE
24   * ************************************************************************/
25
26
27  class Socket {
28  private:
29
30    int sockfd;             // Filedescriptor del socket.
31    struct sockaddr_in miDir;     // Dirección del socket.
32    struct sockaddr_in destinoDir;    // Dirección del socket destino.
33    bool activo;            // Sensa si esta activo el socket
34
35    // Constructor privado.
36    // Crea un nuevo socket.
37    // PRE: 'sockfd' es un filedescriptor que identifica a un socket.
38    explicit Socket(const int sockfd);
39
40    // Enlaza (asocia) al socket con un puerto y una dirección IP.
41    // PRE: 'ip' es una cadena que contiene el nombre del host o la dirección
42    // IP a la que se desea asociar; 'puerto' es el puerto al que se desea
43    // enlazar.
44    // POST: devuelve 1 si se logró enlazar satisfactoriamente o −1 en caso de
45    // error
46    void enlazar(int puerto, std::string ip = "");
47
48  public:
49
50    // Constructor.
51    Socket();
52
53    // Destructor.
54    // Cierra el socket.
55    ~Socket();
56
57    // Crea el socket
58    // POST: lanza una excepción si no se logra llevar a cabo la creación.
59    void crear();
60
61    // Devuelve el ID del socket.
62    // PRE: para considerarse válido, debe haberse creado previamente el
63    // socket.
64    int obtenerID();
65
66    // Conecta el socket a una dirección y puerto destino.
```

```
67    // PRE: 'hostDestino' es una cadena que contiene el nombre del host o la
68    // dirección IP a la que se desea conectar; 'puertoDestino' es el puerto
69    // al que se desea conectar.
70    // POST: determina dirección y puertos locales si no se utilizó el método
71    // bind() previamente. Además, lanza una excepción si no se pudo llevar a
72    // cabo la conexión.
73    void conectar(std::string hostDestino, int puertoDestino);
74
75    // Configura el socket para recibir conexiones en la dirección y puerto
76    // previamente asociados mediante el método enlazar();
77    // PRE: 'maxConexiones' es el número de conexiones entrantes permitidas en
78    // la cola de entrada.
79    // POST: lanza una excepción si no se pudo inicializar la escucha.
80    void escuchar(int maxConexiones, int puerto, std::string ip = "");
81
82    // Espera una conexión en el socket previamente configurado con el método
83    // escuchar().
84    // POST: lanza una excepción si no pudo aceptar la conexión.
85    Socket* aceptar();
86
87    // Envía datos a través del socket de forma completa.
88    // PRE: 'dato' es el dato que se desea enviar; 'longDato' es la longitud
89    // de los datos en bytes.
90    // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
91    // de error.
92    int enviar(const void* dato, int longDato);
93
94    // Recibe datos a través del socket.
95    // PRE: 'buffer' es el buffer en donde se va a depositar la información
96    // leida; 'longBuffer' es la longitud máxima del buffer.
97    // POST: devuelve el número de bytes que han sido leidos o 0 (cero) si el
98    // host remoto a cerrado la conexión.
99    int recibir(void* buffer, int longBuffer);
100
101   // Cierra el socket. Brinda distintos tipos de formas de cerrar permitiendo
102   // realizar un cierre del envío y recepción de datos en forma ordenada.
103   // PRE: si 'modo' es 0, no se permite recibir más datos; si es 1, no se
104   // permite enviar más datos; si es 2, no se permite enviar ni recibir más
105   // datos, quedando inutilizable el socket. Si no se especifica ningún modo
106   // al llamar al método, se utiliza por defecto el modo 2.
107   // POST: el socket quedará parcial o completamente inutilizable
108   // dependiendo del modo elegido.
109   int cerrar(int modo = 2);
110
111   // Corrobora si el socket se encuentra activo. Que no este activo significa
112   // da cuenta de que el socket se encuentra inutilizable para la transmisión
113   // y recepción de datos.
114   // POST: devuelve true si el socket se encuentra activo o false en su
115   // defecto.
116   bool estaActivo();
117 };
118
119 #endif
```



```
1  //
2  //  common_socket.cpp
3  //  CLASE SOCKET
4  //
5  //  Clase que implementa la interfaz de los sockets de flujo (utilizando el
6  //  protocolo TCP), proporcionando un conjunto medianamente extenso de métodos
7  //  y propiedades para las comunicaciones en red.
8  //
9
10
11 #include <iostream>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <errno.h>
16 #include <string.h>
17 #include <sys/types.h>
18 #include <sys/socket.h>
19 #include <arpa/inet.h>
20 #include <sys/wait.h>
21 #include <signal.h>
22 #include <netdb.h>
23 #include "common_socket.h"
24
25
26
27
28
29 /* ******************************************************************************
30  * DEFINICIÓN DE LA CLASE
31  * ******************************************************************************/
32
33
34 // Constructor.
35 Socket::Socket() : activo(false) { }
36
37
38 // Constructor privado.
39 // Crea un nuevo socket.
40 // PRE: 'sockfd' es un filedescriptor que identifica a un socket.
41 Socket::Socket(const int sockfd) : sockfd(sockfd), activo(true) { }
42
43
44 // Destructor.
45 // Cierra el socket.
46 Socket::~Socket() {
47   if(close(this→sockfd) ≡ -1)
48     std::cerr << "ERROR: No se ha podido cerrar el socket." << std::endl;
49 }
50
51
52 // Crea el socket
53 // POST: lanza una excepción si no se logra llevar a cabo la creación.
54 void Socket::crear() {
55   if((this→sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
56     throw "ERROR: No se ha podido crear el socket.";
57
58   // Cambiamos el estado del socket
59   this→activo = true;
60 }
61
62
63 // Devuelve el ID del socket.
64 // PRE: para considerarse válido, debe haberse creado previamente el
65 // socket.
66 int Socket::obtenerID() {
```

```
67    return this→sockfd;
68  }
69
70
71  // Conecta el socket a una dirección y puerto destino.
72  // PRE: 'hostDestino' es una cadena que contiene el nombre del host o la
73  // dirección IP a la que se desea conectar; 'puertoDestino' es el puerto
74  // al que se desea conectar.
75  // POST: determina dirección y puertos locales si no se utilizó el método
76  // bind() previamente. Además, lanza una excepción si no se pudo llevar a
77  // cabo la conexión.
78  void Socket::conectar(std::string hostDestino, int puertoDestino) {
79    // Obtenemos host
80    struct hostent *he = gethostbyname(hostDestino.c_str());
81
82    // Cargamos datos de la conexión a realizar
83    destinoDir.sin_family = AF_INET;
84    destinoDir.sin_port = htons(puertoDestino);
85    // destinoDir.sin_addr.s_addr = inet_addr(ipDestino.c_str());
86    destinoDir.sin_addr = *((struct in_addr *)he→h_addr);
87    memset(&(destinoDir.sin_zero), '\0', sizeof(destinoDir.sin_zero));
88
89    // Conectamos
90    if(connect(this→sockfd, (struct sockaddr *)&destinoDir,
91      sizeof(struct sockaddr)) ≡ −1)
92      throw "ERROR: No se pudo llevar a cabo la conexión.";
93  }
94
95
96  // Configura el socket para recibir conexiones en la dirección y puerto
97  // previamente asociados mediante el método enlazar();
98  // PRE: 'maxConexiones' es el número de conexiones entrantes permitidas en
99  // la cola de entrada.
100 // POST: lanza una excepción si no se pudo inicializar la escucha.
101 void Socket::escuchar(int maxConexiones, int puerto, std::string ip) {
102   // Enlazamos
103   enlazar(puerto, ip);
104
105   // Comenzamos la escucha
106   if(listen(this→sockfd, maxConexiones) ≡ −1)
107     throw "ERROR: No se pudo comenzar a escuchar.";
108 }
109
110
111 // Espera una conexión en el socket previamente configurado con el método
112 // escuchar().
113 // POST: lanza una excepción si no pudo aceptar la conexión.
114 Socket* Socket::aceptar() {
115   unsigned sin_size = sizeof(struct sockaddr_in);
116   int sCliente = accept(sockfd, (struct sockaddr *)&destinoDir, &sin_size);
117
118   // Corroboramos si no se cerró el socket
119   if(¬this→estaActivo()) return 0;
120   // Corroboramos si se produjo un error
121   else if (sCliente < 0)
122     throw "ERROR: No se pudo aceptar la conexión";
123
124   return (new Socket(sCliente));
125 }
126
127
128 // Envía datos a través del socket de forma completa.
129 // PRE: 'dato' es el dato que se desea enviar; 'longDato' es la longitud
130 // de los datos en bytes.
131 // POST: devuelve 0 si se ha realizado el envio correctamente o −1 en caso
132 // de error.
```



```
133 int Socket::enviar(const void* dato, int longDato) {
134   // Cantidad de bytes que han sido enviados
135   int bytesTotal = 0;
136   // Cantidad de bytes que faltan enviar
137   int bytesRestantes = longDato;
138   // Variable auxiliar
139   int n = 0;
140
141   while(bytesRestantes > 0) {
142     // Realizamos envío de bytes
143     n = send(this→sockfd, (char *) dato + bytesTotal, bytesRestantes, 0);
144
145     // En caso de error, salimos
146     if(n ≡ −1)  break;
147
148     // Incrementamos la cantidad de bytes ya enviados
149     bytesTotal += n;
150
151     // Decrementamos cantidad de bytes restantes
152     bytesRestantes −= n;
153   }
154
155   return (n ≡ −1) ? −1:0;
156 }
157
158
159 // Recibe datos a través del socket.
160 // PRE: 'buffer' es el buffer en donde se va a depositar la información
161 // leida; 'longBuffer' es la longitud máxima del buffer.
162 // POST: devuelve el número de bytes que han sido leidos o 0 (cero) si el
163 // host remoto a cerrado la conexión.
164 int Socket::recibir(void* buffer, int longBuffer) {
165   // Limpiamos buffer
166   memset(buffer, '\0', longBuffer);
167   // Recibimos datos en buffer
168   return recv(this→sockfd, buffer, longBuffer, 0);
169 }
170
171
172 // Cierra el socket. Brinda distintos tipos de formas de cerrar permitiendo
173 // realizar un cierre del envío y recepción de datos en forma controlada.
174 // PRE: si 'modo' es 0, no se permite recibir más datos; si es 1, no se
175 // permite enviar más datos; si es 2, no se permite enviar ni recibir más
176 // datos, quedando inutilizable el socket. Si no se especifica ningún modo
177 // al llamar al método, se utiliza por defecto el modo 2.
178 // POST: el socket quedará parcial o completamente inutilizable
179 // dependiendo del modo elegido.
180 int Socket::cerrar(int modo) {
181   if(modo ≡ 2)
182     this→activo = false;
183
184   return shutdown(this→sockfd, modo);
185 }
186
187
188 // Corrobora si el socket se encuentra activo. Que no este activo significa
189 // da cuenta de que el socket se encuentra inutilizable para la transmisión
190 // y recepción de datos.
191 // POST: devuelve true si el socket se encuentra activo o false en su
192 // defecto.
193 bool Socket::estaActivo() {
194   return this→activo;
195 }
196
197
198 // Enlaza (asocia) al socket con un puerto y una dirección IP.
```

jun 25, 13 13:44 **common_socket.cpp** Page 4/4

```
199 // PRE: 'ip' es una cadena que contiene el nombre del host o la dirección
200 // IP a la que se desea asociar; 'puerto' es el puerto al que se desea
201 // enlazar.
202 // POST: lanza una excepción si no se logra llevar a cabo el enlace.
203 void Socket::enlazar(int puerto, std::string ip) {
204   int yes = 1;
205
206   // Reutilizamos socket
207   if(setsockopt(this→sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
208     ≡ −1)
209     throw "ERROR: Antes de enlazar, no se pudo reutilizar socket.";
210
211   // Cargamos datos del enlace a realizar
212   this→miDir.sin_family = AF_INET;
213   this→miDir.sin_port = htons(puerto);
214
215   // Obtenemos host
216   if(ip ≡ "")
217     this→miDir.sin_addr.s_addr = htonl(INADDR_ANY);
218   else {
219     struct hostent *he = gethostbyname(ip.c_str());
220     this→miDir.sin_addr = *((struct in_addr *)he→h_addr);
221   }
222
223   memset(miDir.sin_zero, '\0', sizeof(miDir.sin_zero));
224
225   // Enlazamos
226   if(bind(this→sockfd, (struct sockaddr *)&miDir, sizeof(miDir)) < 0)
227     throw "ERROR: No se pudo llevar a cabo el enlace.";
228 }
```

jun 25, 13 13:44 **common_sha256.h** Page 1/1

```
1  /*
2   *  FIPS−180−2 compliant SHA−256 implementation
3   *
4   *  Copyright (C) 2001−2003  Christophe Devine
5   *
6   *  This program is free software; you can redistribute it and/or modify
7   *  it under the terms of the GNU General Public License as published by
8   *  the Free Software Foundation; either version 2 of the License, or
9   *  (at your option) any later version.
10  *
11  *  This program is distributed in the hope that it will be useful,
12  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
13  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  *  GNU General Public License for more details.
15  *
16  *  You should have received a copy of the GNU General Public License
17  *  along with this program; if not, write to the Free Software
18  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111−1307  USA
19  */
20
21 #ifndef COMMON_SHA256_H
22 #define COMMON_SHA256_H
23
24 #ifndef uint8
25 #define uint8  unsigned char
26 #endif
27
28 #ifndef uint32
29 #define uint32 unsigned long int
30 #endif
31
32 typedef struct
33 {
34     uint32 total[2];
35     uint32 state[8];
36     uint8 buffer[64];
37 }
38 sha256_context;
39
40 void sha256_starts( sha256_context *ctx );
41 void sha256_update( sha256_context *ctx, uint8 *input, uint32 length );
42 void sha256_finish( sha256_context *ctx, uint8 digest[32] );
43
44 #endif
```

```
/*
 *  FIPS-180-2 compliant SHA-256 implementation
 *
 *  Copyright (C) 2001-2003  Christophe Devine
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <string.h>

#include "common_sha256.h"

#define GET_UINT32(n,b,i)                       \
{                                               \
    (n) = ( (uint32) (b)[(i)    ] << 24 )       \
        | ( (uint32) (b)[(i) + 1] << 16 )       \
        | ( (uint32) (b)[(i) + 2] <<  8 )       \
        | ( (uint32) (b)[(i) + 3]       );      \
}

#define PUT_UINT32(n,b,i)                       \
{                                               \
    (b)[(i)    ] = (uint8) ( (n) >> 24 );       \
    (b)[(i) + 1] = (uint8) ( (n) >> 16 );       \
    (b)[(i) + 2] = (uint8) ( (n) >>  8 );       \
    (b)[(i) + 3] = (uint8) ( (n)       );       \
}

void sha256_starts( sha256_context *ctx )
{
    ctx->total[0] = 0;
    ctx->total[1] = 0;

    ctx->state[0] = 0x6A09E667;
    ctx->state[1] = 0xBB67AE85;
    ctx->state[2] = 0x3C6EF372;
    ctx->state[3] = 0xA54FF53A;
    ctx->state[4] = 0x510E527F;
    ctx->state[5] = 0x9B05688C;
    ctx->state[6] = 0x1F83D9AB;
    ctx->state[7] = 0x5BE0CD19;
}

void sha256_process( sha256_context *ctx, uint8 data[64] )
{
    uint32 temp1, temp2, W[64];
    uint32 A, B, C, D, E, F, G, H;

    GET_UINT32( W[0],  data,  0 );
    GET_UINT32( W[1],  data,  4 );
    GET_UINT32( W[2],  data,  8 );
    GET_UINT32( W[3],  data, 12 );
    GET_UINT32( W[4],  data, 16 );
    GET_UINT32( W[5],  data, 20 );
```



```
    GET_UINT32( W[6],  data, 24 );
    GET_UINT32( W[7],  data, 28 );
    GET_UINT32( W[8],  data, 32 );
    GET_UINT32( W[9],  data, 36 );
    GET_UINT32( W[10], data, 40 );
    GET_UINT32( W[11], data, 44 );
    GET_UINT32( W[12], data, 48 );
    GET_UINT32( W[13], data, 52 );
    GET_UINT32( W[14], data, 56 );
    GET_UINT32( W[15], data, 60 );

#define  SHR(x,n) ((x & 0xFFFFFFFF) >> n)
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))

#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))

#define S2(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define S3(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))

#define F0(x,y,z) ((x & y) | (z & (x | y)))
#define F1(x,y,z) (z ^ (x & (y ^ z)))

#define R(t)                                    \
(                                               \
    W[t] = S1(W[t -  2]) + W[t -  7] +          \
           S0(W[t - 15]) + W[t - 16]            \
)

#define P(a,b,c,d,e,f,g,h,x,K)                  \
{                                               \
    temp1 = h + S3(e) + F1(e,f,g) + K + x;      \
    temp2 = S2(a) + F0(a,b,c);                  \
    d += temp1; h = temp1 + temp2;              \
}

    A = ctx->state[0];
    B = ctx->state[1];
    C = ctx->state[2];
    D = ctx->state[3];
    E = ctx->state[4];
    F = ctx->state[5];
    G = ctx->state[6];
    H = ctx->state[7];

    P( A, B, C, D, E, F, G, H, W[ 0], 0x428A2F98 );
    P( H, A, B, C, D, E, F, G, W[ 1], 0x71374491 );
    P( G, H, A, B, C, D, E, F, W[ 2], 0xB5C0FBCF );
    P( F, G, H, A, B, C, D, E, W[ 3], 0xE9B5DBA5 );
    P( E, F, G, H, A, B, C, D, W[ 4], 0x3956C25B );
    P( D, E, F, G, H, A, B, C, W[ 5], 0x59F111F1 );
    P( C, D, E, F, G, H, A, B, W[ 6], 0x923F82A4 );
    P( B, C, D, E, F, G, H, A, W[ 7], 0xAB1C5ED5 );
    P( A, B, C, D, E, F, G, H, W[ 8], 0xD807AA98 );
    P( H, A, B, C, D, E, F, G, W[ 9], 0x12835B01 );
    P( G, H, A, B, C, D, E, F, W[10], 0x243185BE );
    P( F, G, H, A, B, C, D, E, W[11], 0x550C7DC3 );
    P( E, F, G, H, A, B, C, D, W[12], 0x72BE5D74 );
    P( D, E, F, G, H, A, B, C, W[13], 0x80DEB1FE );
    P( C, D, E, F, G, H, A, B, W[14], 0x9BDC06A7 );
    P( B, C, D, E, F, G, H, A, W[15], 0xC19BF174 );
    P( A, B, C, D, E, F, G, H, R(16), 0xE49B69C1 );
    P( H, A, B, C, D, E, F, G, R(17), 0xEFBE4786 );
    P( G, H, A, B, C, D, E, F, R(18), 0x0FC19DC6 );
    P( F, G, H, A, B, C, D, E, R(19), 0x240CA1CC );
    P( E, F, G, H, A, B, C, D, R(20), 0x2DE92C6F );
```

```
    P( D, E, F, G, H, A, B, C, R(21), 0x4A7484AA );
    P( C, D, E, F, G, H, A, B, R(22), 0x5CB0A9DC );
    P( B, C, D, E, F, G, H, A, R(23), 0x76F988DA );
    P( A, B, C, D, E, F, G, H, R(24), 0x983E5152 );
    P( H, A, B, C, D, E, F, G, R(25), 0xA831C66D );
    P( G, H, A, B, C, D, E, F, R(26), 0xB00327C8 );
    P( F, G, H, A, B, C, D, E, R(27), 0xBF597FC7 );
    P( E, F, G, H, A, B, C, D, R(28), 0xC6E00BF3 );
    P( D, E, F, G, H, A, B, C, R(29), 0xD5A79147 );
    P( C, D, E, F, G, H, A, B, R(30), 0x06CA6351 );
    P( B, C, D, E, F, G, H, A, R(31), 0x14292967 );
    P( A, B, C, D, E, F, G, H, R(32), 0x27B70A85 );
    P( H, A, B, C, D, E, F, G, R(33), 0x2E1B2138 );
    P( G, H, A, B, C, D, E, F, R(34), 0x4D2C6DFC );
    P( F, G, H, A, B, C, D, E, R(35), 0x53380D13 );
    P( E, F, G, H, A, B, C, D, R(36), 0x650A7354 );
    P( D, E, F, G, H, A, B, C, R(37), 0x766A0ABB );
    P( C, D, E, F, G, H, A, B, R(38), 0x81C2C92E );
    P( B, C, D, E, F, G, H, A, R(39), 0x92722C85 );
    P( A, B, C, D, E, F, G, H, R(40), 0xA2BFE8A1 );
    P( H, A, B, C, D, E, F, G, R(41), 0xA81A664B );
    P( G, H, A, B, C, D, E, F, R(42), 0xC24B8B70 );
    P( F, G, H, A, B, C, D, E, R(43), 0xC76C51A3 );
    P( E, F, G, H, A, B, C, D, R(44), 0xD192E819 );
    P( D, E, F, G, H, A, B, C, R(45), 0xD6990624 );
    P( C, D, E, F, G, H, A, B, R(46), 0xF40E3585 );
    P( B, C, D, E, F, G, H, A, R(47), 0x106AA070 );
    P( A, B, C, D, E, F, G, H, R(48), 0x19A4C116 );
    P( H, A, B, C, D, E, F, G, R(49), 0x1E376C08 );
    P( G, H, A, B, C, D, E, F, R(50), 0x2748774C );
    P( F, G, H, A, B, C, D, E, R(51), 0x34B0BCB5 );
    P( E, F, G, H, A, B, C, D, R(52), 0x391C0CB3 );
    P( D, E, F, G, H, A, B, C, R(53), 0x4ED8AA4A );
    P( C, D, E, F, G, H, A, B, R(54), 0x5B9CCA4F );
    P( B, C, D, E, F, G, H, A, R(55), 0x682E6FF3 );
    P( A, B, C, D, E, F, G, H, R(56), 0x748F82EE );
    P( H, A, B, C, D, E, F, G, R(57), 0x78A5636F );
    P( G, H, A, B, C, D, E, F, R(58), 0x84C87814 );
    P( F, G, H, A, B, C, D, E, R(59), 0x8CC70208 );
    P( E, F, G, H, A, B, C, D, R(60), 0x90BEFFFA );
    P( D, E, F, G, H, A, B, C, R(61), 0xA4506CEB );
    P( C, D, E, F, G, H, A, B, R(62), 0xBEF9A3F7 );
    P( B, C, D, E, F, G, H, A, R(63), 0xC67178F2 );

    ctx->state[0] += A;
    ctx->state[1] += B;
    ctx->state[2] += C;
    ctx->state[3] += D;
    ctx->state[4] += E;
    ctx->state[5] += F;
    ctx->state[6] += G;
    ctx->state[7] += H;
}

void sha256_update( sha256_context *ctx, uint8 *input, uint32 length )
{
    uint32 left, fill;

    if( ! length ) return;

    left = ctx->total[0] & 0x3F;
    fill = 64 - left;

    ctx->total[0] += length;
    ctx->total[0] &= 0xFFFFFFFF;

```



```
    if( ctx->total[0] < length )
        ctx->total[1]++;

    if( left && length >= fill )
    {
        memcpy( (void *) (ctx->buffer + left),
                (void *) input, fill );
        sha256_process( ctx, ctx->buffer );
        length -= fill;
        input  += fill;
        left = 0;
    }

    while( length >= 64 )
    {
        sha256_process( ctx, input );
        length -= 64;
        input  += 64;
    }

    if( length )
    {
        memcpy( (void *) (ctx->buffer + left),
                (void *) input, length );
    }
}

static uint8 sha256_padding[64] =
{
 0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

void sha256_finish( sha256_context *ctx, uint8 digest[32] )
{
    uint32 last, padn;
    uint32 high, low;
    uint8 msglen[8];

    high = ( ctx->total[0] >> 29 )
         | ( ctx->total[1] <<  3 );
    low  = ( ctx->total[0] <<  3 );

    PUT_UINT32( high, msglen, 0 );
    PUT_UINT32( low,  msglen, 4 );

    last = ctx->total[0] & 0x3F;
    padn = ( last < 56 ) ? ( 56 - last ) : ( 120 - last );

    sha256_update( ctx, sha256_padding, padn );
    sha256_update( ctx, msglen, 8 );

    PUT_UINT32( ctx->state[0], digest,  0 );
    PUT_UINT32( ctx->state[1], digest,  4 );
    PUT_UINT32( ctx->state[2], digest,  8 );
    PUT_UINT32( ctx->state[3], digest, 12 );
    PUT_UINT32( ctx->state[4], digest, 16 );
    PUT_UINT32( ctx->state[5], digest, 20 );
    PUT_UINT32( ctx->state[6], digest, 24 );
    PUT_UINT32( ctx->state[7], digest, 28 );
}
```

jun 25, 13 13:44 **common_sha1.h** Page 1/1

```
/*
 Copyright (c) 2011, Micael Hildenborg
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of Micael Hildenborg nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY Micael Hildenborg ''AS IS'' AND ANY
 EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 DISCLAIMED. IN NO EVENT SHALL Micael Hildenborg BE LIABLE FOR ANY
 DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef SHA1_DEFINED
#define SHA1_DEFINED

namespace sha1
{

    /**
     @param src points to any kind of data to be hashed.
     @param bytelength the number of bytes to hash from the src pointer.
     @param hash should point to a buffer of at least 20 bytes of size for stori
ng the sha1 result in.
     */
    void calc(const void* src, const int bytelength, unsigned char* hash);

    /**
     @param hash is 20 bytes of sha1 hash. This is the same data that is the res
ult from the calc function.
     @param hexstring should point to a buffer of at least 41 bytes of size for
 storing the hexadecimal representation of the hash. A zero will be written at po
sition 40, so the buffer will be a valid zero ended string.
     */
    void toHexString(const unsigned char* hash, char* hexstring);

} // namespace sha1

#endif // SHA1_DEFINED
```

jun 25, 13 13:44 **common_sha1.cpp** Page 1/3

```
/*
 Copyright (c) 2011, Micael Hildenborg
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of Micael Hildenborg nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY Micael Hildenborg ''AS IS'' AND ANY
 EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 DISCLAIMED. IN NO EVENT SHALL Micael Hildenborg BE LIABLE FOR ANY
 DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/*
 Contributors:
 Gustav
 Several members in the gamedev.se forum.
 Gregory Petrosyan
 */

#include "common_sha1.h"

namespace sha1
{
    namespace // local
    {
        // Rotate an integer value to left.
        inline const unsigned int rol(const unsigned int value,
                const unsigned int steps)
        {
            return ((value << steps) | (value >> (32 - steps)));
        }

        // Sets the first 16 integers in the buffert to zero.
        // Used for clearing the W buffert.
        inline void clearWBuffert(unsigned int* buffert)
        {
            for (int pos = 16; --pos >= 0;)
            {
                buffert[pos] = 0;
            }
        }

        void innerHash(unsigned int* result, unsigned int* w)
        {
            unsigned int a = result[0];
            unsigned int b = result[1];
            unsigned int c = result[2];
            unsigned int d = result[3];
            unsigned int e = result[4];

            int round = 0;
```

jun 25, 13 13:44 **common_sha1.cpp** Page 2/3

```
67
68              #define sha1macro(func,val) \
69          { \
70                  const unsigned int t = rol(a, 5) + (func) + e + val + w[round];
    \
71            e = d; \
72            d = c; \
73            c = rol(b, 30); \
74            b = a; \
75            a = t; \
76          }
77
78              while (round < 16)
79              {
80                  sha1macro((b & c) | (~b & d), 0x5a827999)
81                  ++round;
82              }
83              while (round < 20)
84              {
85                  w[round] = rol((w[round - 3] ^ w[round - 8] ^ w[round - 14] ^ w[
    round - 16]), 1);
86                  sha1macro((b & c) | (~b & d), 0x5a827999)
87                  ++round;
88              }
89              while (round < 40)
90              {
91                  w[round] = rol((w[round - 3] ^ w[round - 8] ^ w[round - 14] ^ w[
    round - 16]), 1);
92                  sha1macro(b ^ c ^ d, 0x6ed9eba1)
93                  ++round;
94              }
95              while (round < 60)
96              {
97                  w[round] = rol((w[round - 3] ^ w[round - 8] ^ w[round - 14] ^ w[
    round - 16]), 1);
98                  sha1macro((b & c) | (b & d) | (c & d), 0x8f1bbcdc)
99                  ++round;
100             }
101             while (round < 80)
102             {
103                 w[round] = rol((w[round - 3] ^ w[round - 8] ^ w[round - 14] ^ w[
    round - 16]), 1);
104                 sha1macro(b ^ c ^ d, 0xca62c1d6)
105                 ++round;
106             }
107
108             #undef sha1macro
109
110             result[0] += a;
111             result[1] += b;
112             result[2] += c;
113             result[3] += d;
114             result[4] += e;
115         }
116     } // namespace
117
118     void calc(const void* src, const int bytelength, unsigned char* hash)
119     {
120         // Init the result array.
121         unsigned int result[5] = { 0x67452301, 0xefcdab89, 0x98badcfe, 0x1032547
    6, 0xc3d2e1f0 };
122
123         // Cast the void src pointer to be the byte array we can work with.
124         const unsigned char* sarray = (const unsigned char*) src;
125
126         // The reusable round buffer
```

jun 25, 13 13:44 **common_sha1.cpp** Page 3/3

```
127         unsigned int w[80];
128
129         // Loop through all complete 64byte blocks.
130         const int endOfFullBlocks = bytelength - 64;
131         int endCurrentBlock;
132         int currentBlock = 0;
133
134         while (currentBlock ≤ endOfFullBlocks)
135         {
136             endCurrentBlock = currentBlock + 64;
137
138             // Init the round buffer with the 64 byte block data.
139             for (int roundPos = 0; currentBlock < endCurrentBlock; currentBlock
    += 4)
140             {
141                 // This line will swap endian on big endian and keep endian on l
    ittle endian.
142                 w[roundPos++] = (unsigned int) sarray[currentBlock + 3]
143                         | (((unsigned int) sarray[currentBlock + 2]) << 8)
144                         | (((unsigned int) sarray[currentBlock + 1]) << 16)
145                         | (((unsigned int) sarray[currentBlock]) << 24);
146             }
147             innerHash(result, w);
148         }
149
150         // Handle the last and not full 64 byte block if existing.
151         endCurrentBlock = bytelength - currentBlock;
152         clearWBuffert(w);
153         int lastBlockBytes = 0;
154         for (;lastBlockBytes < endCurrentBlock; ++lastBlockBytes)
155         {
156             w[lastBlockBytes >> 2] |= (unsigned int) sarray[lastBlockBytes + cur
    rentBlock] << ((3 - (lastBlockBytes & 3)) << 3);
157         }
158         w[lastBlockBytes >> 2] |= 0x80 << ((3 - (lastBlockBytes & 3)) << 3);
159         if (endCurrentBlock ≥ 56)
160         {
161             innerHash(result, w);
162             clearWBuffert(w);
163         }
164         w[15] = bytelength << 3;
165         innerHash(result, w);
166
167         // Store hash in result pointer, and make sure we get in in the correct
    order on both endian models.
168         for (int hashByte = 20; --hashByte ≥ 0;)
169         {
170             hash[hashByte] = (result[hashByte >> 2] >> (((3 - hashByte) & 0x3) <
    < 3)) & 0xff;
171         }
172     }
173
174     void toHexString(const unsigned char* hash, char* hexstring)
175     {
176         const char hexDigits[] = { "0123456789abcdef" };
177
178         for (int hashByte = 20; --hashByte ≥ 0;)
179         {
180             hexstring[hashByte << 1] = hexDigits[(hash[hashByte] >> 4) & 0xf];
181             hexstring[(hashByte << 1) + 1] = hexDigits[hash[hashByte] & 0xf];
182         }
183         hexstring[40] = 0;
184     }
185 } // namespace sha1
```

jun 25, 13 13:44 **common_seguridad.h** Page 1/1

```
1  // Seguridad que utiliza hmac_sha1
2
3  #ifndef COMMON_SEGURIDAD_H
4  #define COMMON_SEGURIDAD_H
5
6  #include "common_hash.h"
7  #include <string>
8  #include <cstring>
9
10 class Seguridad {
11 private:
12   // Se realiza clave XOR cadena
13   static std::string XOR(const std::string &clave, const char* cadena);
14
15 public:
16   // Se obtiene la firma del mensaje a ser enviado con la clave pasada por param
   etros
17   static std::string obtenerFirma(const std::string &mensaje, const std::string
   &clave);
18
19   // Se compara la firma pasada por parametros con la firma que se calcula sobre
    el mensaje original
20   // Devuelve 'true' si es valida, o 'false' sino
21   static bool firmaValida(const std::string &mensaje, const std::string &clave,
   const std::string &firmaRecibida);
22 };
23
24 #endif /* COMMON_SEGURIDAD_H */
```

jun 25, 13 13:44 **common_seguridad.cpp** Page 1/1

```
1  #include "common_seguridad.h"
2
3
4  // Se obtiene la firma del mensaje a ser enviado con la clave pasada por paramet
   ros
5  std::string Seguridad::obtenerFirma(const std::string &mensaje, const std::strin
   g &clave) {
6    char ipad[64];  // inner
7    char opad[64];  // outer
8    std::string k = clave;
9
10   // Se inicializan las variables anteriores en 0x36 y 0x5C
11   memset(ipad, 0x36, 64);
12   memset(opad, 0x5C, 64);
13
14   if (k.length() < 64)
15     // Se hace padding con 0 en la clave hasta completar 64 bytes
16     k.append(64 - clave.length(), 0x00);
17   else {
18     // Se calcula el hash de la clave para acortarla
19     k = Hash::funcionDeHashBin(k);
20   }
21
22   // K XOR ipad -> primerHash
23   std::string primerHash = XOR(k, ipad);
24
25   // K XOR opad ->segundoHash
26   std::string segundoHash = XOR(k, opad);
27
28   // Se agrega el mensaje
29   primerHash += mensaje;
30
31   // Hash (primerHash) -> primerHash
32   primerHash = Hash::funcionDeHash(primerHash);
33
34   // Se unen segundoHash + primerHash
35   segundoHash.append(primerHash);
36
37   // Hash (segundoHash + primerHash) -> segundoHash
38   segundoHash = Hash::funcionDeHash(segundoHash);
39
40   return segundoHash;
41 }
42
43 // Se compara la firma pasada por parametros con la firma que se calcula sobre e
   l mensaje original
44 bool Seguridad::firmaValida(const std::string &mensaje, const std::string &clave
   , const std::string &firmaRecibida) {
45   if (obtenerFirma(mensaje, clave) == firmaRecibida)
46     return true;
47   return false;
48 }
49
50
51 /* Implementacion de metodos privados */
52
53 // Devuelve clave XOR cadena
54 std::string Seguridad::XOR(const std::string &clave, const char* cadena) {
55   int i;
56   std::string aux;
57   for (i = 0; i < 64; i++)
58     aux += (clave[i] ^ cadena[i]);
59   return aux;
60 }
61
```

```
1  //
2  //  common_protocolo.h
3  //
4  //  Cabecera con constantes que especifican el protocolo de mensajes a
5  //  ser utilizados por el cliente y el servidor.
6  //
7
8
9  #ifndef PROTOCOLO_H
10 #define PROTOCOLO_H
11
12 #include <string>
13
14
15
16
17 /* ************************************************************************
18  * PROTOCOLO DE INSTRUCCIONES
19  * ************************************************************************/
20
21
22 // Constantes para los identificadores de instrucciones enviadas por el
23 // cliente
24 // FORMATO DE LAS CONSTANTES: C_[instruccion]
25 const std::string C_LOGIN_REQUEST = "LOGIN-REQUEST";
26 const std::string C_GET_FILES_LIST = "GET-FILES-LIST";
27 const std::string C_FILE_REQUEST = "FILE-REQUEST";
28 const std::string C_FILE_PARTS_REQUEST = "FILE-PARTS-REQUEST";
29 const std::string C_MODIFY_FILE = "MODIFY-FILE";
30
31 // Constantes para los identificadores de instrucciones enviadas por el
32 // servidor
33 // FORMATO DE LAS CONSTANTES: S_[instruccion]
34 const std::string S_LOGIN_OK = "LOGIN-OK";
35 const std::string S_LOGIN_FAIL = "LOGIN-FAIL";
36 const std::string S_FILES_LIST = "FILES-LIST";
37 const std::string S_FILE_CHANGED = "FILE-CHANGED";
38 const std::string S_NEW_FILE = "NEW-FILE";
39 const std::string S_NO_SUCH_FILE = "NO-SUCH-FILE";
40 const std::string S_CORRUPT_MESSAGE = "CORRUPT-MESSAGE";
41 const std::string S_SERVER_INFO = "SERVER-INFO";
42 const std::string S_SERVER_USER_LIST = "SERVER-USER-LIST";
43 const std::string S_SERVER_USER_PASS = "SERVER-USER-PASS";
44 const std::string S_SERVER_LOG = "SERVER-LOG";
45
46
47 // Constates para los identificadores de instrucciones enviadas por el monitor
48 const std::string M_SERVER_INFO_REQUEST = "SERVER-INFO-REQUEST";
49 const std::string M_SERVER_USER_LIST_REQUEST = "SERVER-USER-LIST-REQUEST";
50 const std::string M_SERVER_NEW_USER_INFO = "SERVER-NEW-USER-INFO";
51 const std::string M_SERVER_DELETE_USER = "SERVER-DELETE-USER";
52 const std::string M_SERVER_MODIFY_USER_REQUEST = "SERVER-MODIFY-USER-REQUEST";
53 const std::string M_SERVER_MODIFY_USER = "SERVER-MODIFIED-USER";
54 const std::string M_SERVER_LOG_REQUEST = "SERVER-LOG-REQUEST";
55
56 // Constantes para los identificadores de instrucciones comunes al servidor
57 // y al cliente
58 // FORMATO DE LAS CONSTANTES: COMMON_[instruccion]
59 const std::string COMMON_SEND_FILE = "SEND-FILE";
60 const std::string COMMON_DELETE_FILE = "DELETE-FILE";
61 const std::string COMMON_FILE_PARTS = "FILE-PARTS";
62 const std::string COMMON_DELIMITER = "/";
63
64 // Constante para caracter de fin de instrucción
65 const char FIN_MENSAJE = '\n';
66
```



```
67
68 #endif
```

jun 25, 13 13:44 **common_parser.h** Page 1/1

```
1  // Parser de mensajes
2
3  #ifndef COMMON_PARSER_H
4  #define COMMON_PARSER_H
5
6  #include <string>
7  #include <sstream>
8  #include "common_lista.h"
9
10 //DEBUG
11 #include <iostream>
12
13 class Parser {
14 public:
15   // Parsea el mensaje separando la instruccion de sus argumentos.
16   // PRE: 'msg' es el mensaje que desea parsearse; 'instruccion' y 'args' son
17   // referencias a variables en donde se desea almacenar la instruccion y sus
18   // argumentos respectivamente.
19   static void parserInstruccion(const std::string& msg,
20     std::string& instruccion, std::string& args);
21
22   // Parsea el mensaje separando los argumentos y los devuelve en una lista
23   // en el orden en que se leyeron. Si el mensaje esta vacio, no se modifica la
   lista
24   static void dividirCadena(const std::string &msg,
25     Lista<std::string>* args, char delim);
26
27 };
28
29 #endif
```

jun 25, 13 13:44 **common_parser.cpp** Page 1/1

```
1  #include "common_parser.h"
2
3  // Parsea el mensaje separando la instruccion de sus argumentos.
4  // PRE: 'msg' es el mensaje que desea parsearse; 'instruccion' y 'args' son
5  // referencias a variables en donde se desea almacenar la instruccion y sus
6  // argumentos respectivamente.
7  void Parser::parserInstruccion(const std::string& msg,
8    std::string& instruccion, std::string& args) {
9    std::stringstream msgTemp(msg);
10
11   // Tomamos la instrucción
12   msgTemp >> instruccion;
13   getline(msgTemp, args);
14
15   // Eliminamos el espacio inicial sobrante de los argumentos
16   if(args ≠ "") args.erase(0, 1);
17 }
18
19
20 // Parsea el mensaje separando los argumentos y los devuelve en una lista
21 // en el orden en que se leyeron. Si el mensaje esta vacio, no se modifica la li
   sta
22 void Parser::dividirCadena(const std::string &msg, Lista<std::string>* args,
23   char delim) {
24   // Se procesa solo si el mensaje tiene contenido
25   if (¬msg.empty()) {
26     // Variables auxiliares
27     std::string m = msg;
28
29     // posicion del delim
30     size_t d = 0;
31
32     while(d ≠ std::string::npos){
33       d = m.find(delim);
34       args→insertarUltimo(m.substr(0, d));
35       m.erase(0, d+1);
36     }
37   }
38 }
```

jun 25, 13 13:44 **common_mutex.h** Page 1/1

```
1  //
2  //  common_mutex.h
3  //  CLASE MUTEX
4  //
5  //  Clase que implementa el tipo de objetos mutex, es decir, objetos con dos
6  //  estados posibles: tomado y liberado. Este puede ser manipulado desde
7  //  varios hilos simultáneamente.
8  //
9
10
11 #ifndef MUTEX_H
12 #define MUTEX_H
13
14
15 #include <pthread.h>
16
17
18
19
20 /* *****************************************************************************
21  * DECLARACIÓN DE LA CLASE
22  * *****************************************************************************/
23
24
25 class Mutex {
26 private:
27
28   pthread_mutex_t mutex;      // Mutex
29   pthread_cond_t cond_var;    // Condition variable
30
31   // Constructor privado
32   Mutex(const Mutex &c);
33
34   // Bloquea la ejecución en un una condition variable hasta que se produzca
35   // una señalización.
36   void wait();
37
38   // Desbloquea al menos uno de los hilos que están bloqueados en la
39   // condition variable.
40   void signal();
41
42   // Desbloquea todos los hilos bloqueados actualmente en la condition
43   // variable.
44   void broadcast();
45
46   // Bloquea el mutex
47   void lock();
48
49   // Desbloquea el mutex
50   void unlock();
51
52 public:
53
54   // Constructor
55   Mutex();
56
57   // Destructor
58   ~Mutex();
59
60   friend class Lock;
61 };
62
63 #endif
```

jun 25, 13 13:44 **common_mutex.cpp** Page 1/2

```
1  //
2  //  common_mutex.cpp
3  //  CLASE MUTEX
4  //
5  //  Clase que implementa el tipo de objetos mutex, es decir, objetos con dos
6  //  estados posibles: tomado y liberado. Este puede ser manipulado desde
7  //  varios hilos simultáneamente.
8  //
9
10
11 #include "common_mutex.h"
12
13
14
15
16 /* *****************************************************************************
17  * DEFINICIÓN DE LA CLASE
18  * *****************************************************************************/
19
20
21 // Constructor
22 Mutex::Mutex() {
23   pthread_mutex_init(&this→mutex, 0);
24   pthread_cond_init(&this→cond_var, 0);
25 }
26
27
28 // Constructor privado
29 Mutex::Mutex(const Mutex &c) { }
30
31
32 // Destructor
33 Mutex::~Mutex() {
34   pthread_mutex_destroy(&this→mutex);
35   pthread_cond_destroy(&this→cond_var);
36 }
37
38
39 // Bloquea la ejecución en un una condition variable hasta que se produzca
40 // una señalización.
41 void Mutex::wait() {
42   pthread_cond_wait(&this→cond_var, &this→mutex);
43 }
44
45
46 // Desbloquea al menos uno de los hilos que están bloqueados en la
47 // condition variable.
48 void Mutex::signal() {
49   pthread_cond_signal(&this→cond_var);
50 }
51
52
53 // Desbloquea todos los hilos bloqueados actualmente en la condition
54 // variable.
55 void Mutex::broadcast() {
56   pthread_cond_broadcast(&this→cond_var);
57 }
58
59
60 // Bloquea el mutex
61 void Mutex::lock() {
62   pthread_mutex_lock(&this→mutex);
63 }
64
65
66 // Desbloquea el mutex
```

jun 25, 13 13:44 **common_mutex.cpp** Page 2/2

```
67 void Mutex::unlock() {
68   pthread_mutex_unlock(&this→mutex);
69 }
```

jun 25, 13 13:44 **common_manejador_de_archivos.h** Page 1/4

```
1  //
2  //  common_manejador_de_archivos.h
3  //  CLASE MANEJADORDEARCHIVOS
4  //
5
6
7  #ifndef MANEJADOR_DE_ARCHIVOS_H
8  #define MANEJADOR_DE_ARCHIVOS_H
9
10 #include "common_mutex.h"
11 #include "common_lock.h"
12 #include "common_hash.h"
13 #include "common_lista.h"
14 #include "common_cola.h"
15 #include "common_logger.h"
16 #include <string>
17 #include <stdio.h>
18 #include <iostream>
19 #include <fstream>
20 #include <utility>
21 #include <math.h>
22
23
24
25
26
27 /* ******************************************************************************
28  * DECLARACIÓN DE LA CLASE
29  * ******************************************************************************/
30
31
32 class ManejadorDeArchivos {
33 private:
34
35   std::string directorio;       // Directorio sobre el cual se trabaja
36   Mutex mArc;             // Mutex para accesos fisicos
37   Mutex mReg;             // Mutex para accesos sobre registros
38   Logger *logger;           // Logger de eventos
39
40
41
42   // Procesa dos hashes pertenecientes al contenido de un archivo y
43   // obtiene los bloques que han cambiado.
44   // PRE: 'hashViejo' y 'hashNuevo' son los hashes de los archivos a
45   // procesar; 'cantNuevaBloques' es la cantidad de bloques del archivo
46   // que es representado por 'hashNuevo'
47   // POST: se listan en 'listaBLoquesDiferentes' los numero de bloques
48   // que han cambiado; Se devuelve true si se encontraron diferencias o
49   // false en caso contrario.
50   bool obtenerDiferencias(std::string& hashViejo, std::string& hashNuevo,
51     int& cantNuevaBloques, Lista<int> *listaBloquesDiferentes);
52
53   // Separa de una linea el nombre y el hash
54   void separarNombreYHash(const std::string &linea, std::string& nombre, std::st
   ring &hash);
55
56 public:
57
58   // Constructor
59   ManejadorDeArchivos(const std::string& directorio, Logger *logger);
60
61   // Destructor
62   ~ManejadorDeArchivos();
63
64   // Devuelve una lista con los archivos (ordenados por nombre) que se
65   // encuentran ubicados en el directorio administrado por el manejador.
```

jun 25, 13 13:44 **common_manejador_de_archivos.h** Page 2/4

```
66    void obtenerArchivosDeDirectorio(Lista<std::string>* listaArchivos);
67
68    // Devuelve una lista con los archivos (ordenados por nombre) que se
69    // encuentran ubicados en el registro administrado por el manejador.
70    void obtenerArchivosDeRegistro(Lista<std::string>* listaArchivos,
71      Lista< std::pair< std::string, std::string> >* listaPares);
72
73    // Agrega un nuevo archivo al directorio.
74    // PRE: 'nombreArchivo' es el nombre del archivo nuevo; 'contenido' es el
75    // contenido del archivo nuevo expresado en formato hexadecimal como una
76    // cadena.
77    // Tira una excepcion si no logra crear un archivo nuevo
78    void agregarArchivo(const std::string& nombreArchivo,
79      const std::string& contenido);
80
81    // Elimina un archivo del directorio.
82    // PRE: 'nombreArchivo' es el nombre de archivo.
83    // POST: devuelve true si se eliminó con éxito o false en su defecto.
84    bool eliminarArchivo(const std::string& nombreArchivo);
85
86    // Realiza modificaciones sobre los bloques de un archivo.
87    // PRE: 'nombreArchivo' es el nombre del archivo a modificar;
88    // 'cantloquesDelArchivo' es la cantidad nueva de bloques que debe
89    // contener el archivo; 'listaBloquesAReemplazar' es una lista que
90    // contiene los números de bloque y su respectivo contenido, los
91    // cuales reemplazarán a los bloques actuales.
92    void modificarArchivo(std::string& nombreArchivo,
93      unsigned int cantBytesDelArchivo,
94      Lista< std::pair< int, std::string > >& listaBloquesAReemplazar);
95
96    // Comprueba la existencia de un archivo en el directorio.
97    // PRE: 'nombreArchivo' es el nombre de archivo a buscar.
98    // POST: devuelve true si existe o false en caso contrario.
99    bool existeArchivo(std::string& nombreArchivo);
100
101   // Calcula el hash del archivo, el cual se encuentra conformado
102   // por los hashes de cada bloque concatenados.
103   // PRE: 'nombreArchivo' es el nombre de archivo, 'hashArchivo' es
104   // el string en donde se depositará el hash.
105   // POST: se devuelve la cantidad de bloques que posee actualmente el
106   // archivo.
107   int obtenerHash(const std::string& nombreArchivo,
108     std::string& hashArchivo);
109
110   // Devuelve el hash del archivo, el cual se encuentra conformado
111   // por los hashes de cada bloque concatenados.
112   // PRE: 'nombreArchivo' es el nombre de archivo de registro,
113   // 'listaNombreHashReg' es una lista de pares de <nombre,hash> del
114   // archivo registro
115   // POST: Se devuelve el hash
116   std::string obtenerHashRegistro(Lista< std::pair< std::string,
117     std::string> > * listaNombreHashReg, std::string& nombreArchivo);
118
119   // Devuelve el hash del bloque de un archivo.
120   // PRE: 'nombreArchivo' es el nombre de archivo del bloque; 'numBloque'
121   // es el número de bloque del que se desea obtener el hash.
122   // POST: se devuelve una cadena con el hash del bloque.
123   std::string obtenerHashDeBloque(const std::string& nombreArchivo,
124     int numBloque);
125
126   // Devuelve el contenido de un archivo en formato hexadecimal expresado
127   // en una cadena de caracteres.
128   // PRE: 'nombreArchivo' es el nombre de archivo; 'numBloque' es el
129   // número de bloque que se desea obtener el archivo. Si no se especifica
130   // o si es cero, se devuelve el contenido completo del archivo.
131   // POST: devuelve una cadena que representa el contenido.
```

jun 25, 13 13:44 **common_manejador_de_archivos.h** Page 3/4

```
132   std::string obtenerContenido(const std::string& nombreArchivo,
133     int numBloque = 0);
134
135   // Recibe una lista de archivos, compara con la que se encuentra localmente
136   // * ListaExterna: lista de archivos con la cual se compara
137   // * Faltantes: lista de archivos que estan modificados en el dir local
138   // * Sobrantes: lista de archivos que no estan en la lista que se deben
139   // eliminar del dir local
140   // * Nuevos: lista de archivos que no estan en el dir local
141   void obtenerListaDeActualizacion(Lista< std::pair< std::string,
142     std::pair< std::string, int > > >* listaExterna,
143     Lista< std::pair< std::string, Lista<int> > >* faltantes,
144     Lista<std::string>* sobrantes);
145
146   // Devuelve la cantidad de bloques de un archivo
147   int obtenerCantBloques(const std::string &nombreArchivo);
148
149   // Devuelve la cantidad de Bytes de un archivo
150   // PRE: 'nombreArchivo' es el nombre de archivo. El archivo no debe
151   // sobrepasar los 4Gb de tamaño.
152   // POST: en caso de no poder abrir el archivo (a causa de no existencia),
153   // se devuelve 0. Se recomienda al usuario verificar la existencia
154   // previamente para no confundir el cero de error con el valor nulo de
155   // que puede poseer cierto archivo.
156   unsigned int obtenerCantBytes(const std::string &nombreArchivo);
157
158   // Crea un archivo de registro.
159   // PRE: 'nombreArchivo' es la ruta hacia el archivo junto a su nombre.
160   // POST: devuelve true si se realizó la creación con éxito o false en su
161   // defecto.
162   bool crearRegistroDeArchivos();
163
164   // Actualiza el registro local de archivos.
165   // PRE: 'nuevos', 'modificados' y 'eliminados' son punteros a cola donde
166   // se insertarán los nombres de archivo correspondientes a la situación
167   // en la que se encuentren.
168   // POST: se devuelve 'false' si se produjeron cambios en el registro o
169   // 'true' en su defecto; esto evita tener que revisar las colas para
170   // comprobar cambios.
171   bool actualizarRegistroDeArchivos(Cola< std::pair< std::string,
172     std::string > > *nuevos, Cola< std::pair< std::string, Lista<int> > >
173     *modificados, Cola< std::string > *eliminados);
174
175   // Actualiza el registro local de archivos.
176   // PRE: las listas corresponden a que archivos nuevos o modificados deben
177   // tenerse en cuenta, siendo que los demás detectados en el momento de la
178   // actualización, son salteados.
179   // POST: se devuelve 'false' si se produjeron cambios en el registro o
180   // 'true' en su defecto; esto evita tener que revisar las colas para
181   // comprobar cambios.
182   bool actualizarRegistroDeArchivos(Lista< std::string >&
183     nuevosActualizables, Lista< std::string >& modificadosActualizables);
184
185   // Elimina el registro que identifica a un archivo en el registro de
186   // archivos.
187   // PRE: 'nombreArchivo' es el nombre del archivo a eliminar del registro.
188   void borrarDeRegistroDeArchivos(const std::string& nombreArchivo);
189
190   // Comprueba si existe cierto registro de archivos.
191   // PRE: 'nombreArchivo' es la ruta hacia el archivo junto a su nombre.
192   // POST: devuelve true si existe o false en su defecto.
193   bool existeRegistroDeArchivos();
194
195   // Corrobora si se encuentra registrado un archivo en el registro de
196   // archivos.
197   // PRE: 'nombreArchivo' es el nombre de archivo.
```

```
198   // POST: devuelve true si el archivo se encuentra registrado o false
199   // en caso contrario.
200   bool existeArchivoEnRegitro(const std::string nombreArchivo);
201
202   // Compara el hash actual de cierto bloque de archivo con un hash pasado
203   // por parámetro.
204   // PRE: 'nombreArchivo' es el nombre de archivo; 'numBloque' es el
205   // número del bloque que se desea comparar; 'hash' es el hash que
206   // se comparará con el del bloque del archivo.
207   // POST: devuelve true si son iguales o false si presentan diferencias.
208   bool compararBloque(const std::string& nombreArchivo, const int numBloque,
209     const std::string& hash);
210 };
211
212 #endif
```



```
1  //
2  //  common_manejador_de_archivos.h
3  //  CLASE MANEJADORDEARCHIVOS
4  //
5
6
7  #include "common_manejador_de_archivos.h"
8  #include "common_parser.h"
9  #include "common_lista.h"
10 #include "common_hash.h"
11 #include "common_utilidades.h"
12 #include <stdlib.h>
13 #include <cstring>
14 #include <sys/types.h>
15 #include <sys/stat.h>
16 #include "dirent.h"
17
18
19
20
21 namespace {
22   // Constantes para los nombres de directorio
23   const std::string DIR_AU = ".au";
24
25   // Constantes para los nombres de archivo
26   const std::string ARCHIVO_REG_ARCHIVOS = ".reg_archivos";
27
28   // Delimitador de campos del registro
29   const std::string DELIMITADOR = ",";
30
31   // Constante que define el tamaño de los bloques de archivos en cantidad
32   // de caracteres hexadecimales (ej: si se quiere un tamaño de bloque de
33   // 10 Bytes, se debe insertar el valor 20).
34   // const int TAMANIO_BLOQUE = 20;     // 10 bytes por bloque
35   // const int TAMANIO_BLOQUE = 524288;   // 256K por bloque
36   const int TAMANIO_BLOQUE = 2097152;     // 1Mb por bloque
37
38   // Constante que define el tamaño de los bloques de hash de archivos.
39   const int TAMANIO_BLOQUE_HASH = 40;
40
41   // Longitud de nombre del archivo temporal
42   const int LONGITUD_TEMP = 40;
43
44   // Caracter no permitido en los nombres de archivo
45   const char CHAR_PROHIBIDO = '~';
46   const std::string PREF_ARCHIVO_PROHIBIDO = ".fuse_hidden" ;
47 }
48
49
50
51
52
53 /* ******************************************************************************
54  * DEFINICIÓN DE LA CLASE
55  * ******************************************************************************/
56
57
58 // Constructor
59 ManejadorDeArchivos::ManejadorDeArchivos(const std::string& directorio,
60   Logger *logger) : directorio(directorio), logger(logger) { }
61
62
63 // Destructor
64 ManejadorDeArchivos::~ManejadorDeArchivos() { }
65
66
```

```
67  // Devuelve una lista con los archivos (ordenados por nombre) que se
68  // encuentran ubicados en el directorio administrado por el manejador.
69  void ManejadorDeArchivos::obtenerArchivosDeDirectorio(
70    Lista<std::string>* listaArchivos) {
71    // Variables auxiliares
72    DIR *dir;
73    struct dirent *entrada = 0;
74    unsigned char esDirectorio = 0x4;
75
76    // Abrimos directorio y procesamos si fue exitosa la apertura
77    if((dir = opendir (this→directorio.c_str())) ≠ NULL) {
78      // Iteramos sobre cada objeto del directorio
79      while ((entrada = readdir (dir)) ≠ NULL) {
80        // Salteamos directorios
81        if (entrada→d_type ≡ esDirectorio)
82          continue;
83
84        // Si tiene el char ~ se saltea
85        if(strchr(entrada→d_name, CHAR_PROHIBIDO))
86          continue;
87
88        std::string s(entrada→d_name);
89        if(std::string::npos ≠ s.find(PREF_ARCHIVO_PROHIBIDO)) continue;
90
91        // Insertamos el nombre de archivo en la lista
92        listaArchivos→insertarUltimo(entrada→d_name);
93      }
94
95      closedir(dir);
96    }
97    else {
98      // Mensaje de log
99      this→logger→emitirLog("ERROR: No se ha podido abrir el directorio " +this→directorio);
100     throw "ERROR: No se ha podido abrir el directorio.";
101   }
102
103   // Ordenamos la lista de archivos
104   listaArchivos→ordenar();
105 }
106
107
108 // Devuelve una lista con los archivos (ordenados por nombre) que se
109 // encuentran ubicados en el registro administrado por el manejador.
110 void ManejadorDeArchivos::obtenerArchivosDeRegistro(Lista<std::string>* listaArc
    hivos,
111   Lista< std::pair< std::string, std::string> >* listaPares) {
112   // Bloqueamos el mutex
113   Lock l(mReg);
114
115   // variables auxiliares
116   std::string nombre, hash, linea;
117
118
119   // Armamos ruta del registro
120   std::string registro = this→directorio + DIR_AU + "/"
121     + ARCHIVO_REG_ARCHIVOS;
122
123   // se abre el archivo
124   std::ifstream archivo(registro.c_str(), std::ios_base::in);
125
126   if (archivo.is_open()) {
127     // Se leen y guardan los nombres de archivos + hash en la lista
128     while(std::getline(archivo, linea)) {
129       separarNombreYHash(linea, nombre, hash);
130       std::pair< std::string, std::string> datos =
131         make_pair(nombre, hash);
```



```
132       listaPares→insertarUltimo(datos);
133       listaArchivos→insertarUltimo(nombre);
134     }
135
136     // Se cierra el archivo
137     archivo.close();
138   }
139
140   listaArchivos→ordenar();
141 }
142
143
144 // Agrega un nuevo archivo al directorio.
145 // PRE: 'nombreArchivo' es el nombre del archivo nuevo; 'contenido' es el
146 // contenido del archivo nuevo expresado en formato hexadecimal como una
147 // cadena.
148 // POST: devuelve true si se agregó el archivo con éxito o false en caso
149 // contrario
150 void ManejadorDeArchivos::agregarArchivo(const std::string& nombreArchivo,
151   const std::string& contenido) {
152   // Bloqueamos el mutex
153   Lock l(mArc);
154
155   // Variables auxiliares
156   std::fstream archivo;
157
158   // Armamos ruta del archivo
159   std::string ruta = this→directorio + "/" + nombreArchivo;
160
161   // Intenta abrir el archivo
162   archivo.open(ruta.c_str(), std::ios_base::in);
163
164   // Si abre, se elimina
165   if (archivo.is_open()) {
166     // Se cierra
167     archivo.close();
168
169     // Se elimina
170     remove(ruta.c_str());
171   }
172
173   // Se crea
174   archivo.open(ruta.c_str(), std::ios_base::out | std::ios_base::app);
175
176   // No se pudo crear el archivo
177   if (¬archivo.is_open()) {
178     // Mensaje de log
179     this→logger→emitirLog("ERROR: Archivo nuevo " + nombreArchivo +
180       " no pudo ser creado.");
181     throw "ERROR: Archivo nuevo no pudo ser creado.";
182   }
183
184   // Se convierte el archivo de hexa a char nuevamente
185   uint8_t *archivoBin = Convertir::htoui(contenido);
186   size_t len = contenido.size() / 2;
187
188   // Se escribe el contenido en el archivo
189   archivo.write((char*) archivoBin, len);
190
191   // Se cierra el archivo
192   archivo.close();
193
194 }
195
196
197 // Elimina un archivo del directorio.
```

```
198 // PRE: 'nombreArchivo' es el nombre de archivo.
199 // POST: devuelve true si se eliminó con éxito o false en su defecto.
200 bool ManejadorDeArchivos::eliminarArchivo(const std::string& nombreArchivo) {
201   // Bloqueamos el mutex
202   Lock l(mArc);
203
204   // Variables auxiliares
205   std::fstream archivo;
206
207   // Armamos ruta del archivo
208   std::string ruta = this→directorio + "/" + nombreArchivo;
209
210   // Busca el archivo y si lo encuentra, lo borra
211   archivo.open(ruta.c_str(), std::ios_base::in);
212
213   // Comprobamos si se abrió el archivo, señal de que existe
214   if(¬archivo.is_open()) return false;
215
216   // Cerramos el archivo y lo eliminamos del directorio
217   archivo.close();
218   remove(ruta.c_str());
219
220   return true;
221 }
222
223
224 // Realiza modificaciones sobre los bloques de un archivo.
225 // PRE: 'nombreArchivo' es el nombre del archivo a modificar;
226 // 'cantloquesDelArchivo' es la cantidad nueva de bloques que debe
227 // contener el archivo; 'listaBloquesAReemplazar' es una lista que
228 // contiene los números de bloque y su respectivo contenido, los
229 // cuales reemplazarán a los bloques actuales.
230 void ManejadorDeArchivos::modificarArchivo(std::string& nombreArchivo,
231   unsigned int cantBytesDelArchivo, Lista< std::pair< int, std::string > >&
232   listaBloquesAReemplazar) {
233   // Variables auxiliares
234   std::fstream archivo, archivoTemp;
235   std::string sRandom;
236   Utilidades::randomString(LONGITUD_TEMP, sRandom);
237   std::string nombreArchivoTemp = "." + sRandom + "~";
238   unsigned int cantBytesParcial = 0;
239   unsigned int cantBytesPorBloque = TAMANIO_BLOQUE / 2;
240   int i = 1;
241
242   // Armamos ruta de archivos
243   std::string ruta = this→directorio + "/" + nombreArchivo;
244   std::string rutaTemp = this→directorio + "/" + nombreArchivoTemp;
245
246   // Intentamos abrir el archivos
247   archivo.open(ruta.c_str(), std::ios::in);
248   archivoTemp.open(rutaTemp.c_str(), std::ios_base::out |
249     std::ios_base::app);
250
251   // Verificamos si la apertura fue exitosa
252   if(¬archivo.is_open() ∨ ¬archivoTemp.is_open()) {
253     // Mensaje de log
254     this→logger→emitirLog("ERROR: El archivo " + nombreArchivo +
255         " no pudo ser abierto.");
256     throw "ERROR: El archivo no pudo ser abierto.";
257   }
258
259
260   // Vamos insertando bytes, reemplazando los actualizados hasta
261   // llegar al tamaño en bytes que debe tener el archivo
262   while(cantBytesParcial < cantBytesDelArchivo) {
263     // Variable auxiliar para el contenido
```



```
264     std::string contenidoBloque;
265
266     // Verificamos si debe reemplazarse bloque
267     if(¬listaBloquesAReemplazar.estaVacia() ∧
268       (i ≡ listaBloquesAReemplazar.verPrimero().first)) {
269       // Insertamos el contenido nuevo en el bloque
270       contenidoBloque = listaBloquesAReemplazar.verPrimero().second;
271       listaBloquesAReemplazar.eliminarPrimero();
272     }
273     else
274       contenidoBloque = this→obtenerContenido(nombreArchivo, i);
275
276     // Corroboramos si debemos truncar el bloque
277     unsigned int v = cantBytesDelArchivo − cantBytesParcial;
278
279     if(v < cantBytesPorBloque)
280       // Obtenemos la cantidad de bytes que necesitamos del bloque
281       contenidoBloque = contenidoBloque.substr(0, v * 2);
282
283     // Se convierte el contenido de hexa a char nuevamente
284     uint8_t *contenidoBloqueBin = Convertir::htoui(contenidoBloque);
285     size_t len = contenidoBloque.size() / 2;
286
287     // Se escribe el contenido en el archivo
288     archivoTemp.write((char*) contenidoBloqueBin, len);
289
290     // Incrementamos la cantidad parcial de bytes a guardar
291     cantBytesParcial += len;
292
293     i++;
294   }
295
296   archivo.close();
297   archivoTemp.close();
298
299   // Eliminamos el archivo original y convertimos el temporal en el oficial
300   remove(ruta.c_str());
301   rename(rutaTemp.c_str(), ruta.c_str());
302 }
303
304
305 // Comprueba la existencia de un archivo en el directorio.
306 // PRE: 'nombreArchivo' es el nombre de archivo a buscar.
307 // POST: devuelve true si existe o false en caso contrario.
308 bool ManejadorDeArchivos::existeArchivo(std::string& nombreArchivo) {
309   // Variables auxiliares
310   Lista<std::string> l;
311
312   // Relevamos los archivos del directorio
313   this→obtenerArchivosDeDirectorio(&l);
314
315   return l.buscar(nombreArchivo);
316 }
317
318
319 // Calcula el hash del archivo, el cual se encuentra conformado
320 // por los hashes de cada bloque concatenados.
321 // PRE: 'nombreArchivo' es el nombre de archivo, 'hashArchivo' es
322 // el string en donde se depositará el hash.
323 // POST: se devuelve la cantidad de bloques que posee actualmente el
324 // archivo.
325 int ManejadorDeArchivos::obtenerHash(const std::string& nombreArchivo,
326   std::string& hashArchivo) {
327   // Limpiamos el argumento en donde se depositará el hash del contenido
328   hashArchivo.clear();
329
```

```
330   // Obtenemos la cantidad de bloques del archivo
331   int cantBloques = this→obtenerCantBloques(nombreArchivo);
332
333   for(int i = 1; i ≤ cantBloques; i++) {
334     // Obtenemos el bloque i del contenido
335     std::string bloque = this→obtenerContenido(nombreArchivo, i);
336
337     if(i ≡ cantBloques) {
338       unsigned int bytes = this→obtenerCantBytes(nombreArchivo);
339
340       // Quiltamos el relleno
341       bloque = bloque.substr(0, (bytes * 2) − (TAMANIO_BLOQUE *
342         (i − 1)));
343     }
344
345     // Concatenamos el hash del bloque
346     hashArchivo.append(Hash::funcionDeHash(bloque));
347   }
348
349   return cantBloques;
350 }
351
352
353 // Devuelve el hash del archivo, el cual se encuentra conformado
354 // por los hashes de cada bloque concatenados.
355 // PRE: 'nombreArchivo' es el nombre de archivo de registro,
356 // 'listaNombreHashReg' es una lista de pares de <nombre,hash> del
357 // archivo registro
358 // POST: Se devuelve el hash
359 std::string ManejadorDeArchivos::obtenerHashRegistro(Lista< std::pair<
360   std::string, std::string> > * listaNombreHashReg, std::string& nombreArchivo)
    {
361   // Variables auxiliares
362   int i, tam = listaNombreHashReg→tamanio();
363   std::pair< std::string, std::string> par;
364   std::string hash;
365
366   // Se busca en la lista
367   for (i = 0; i < tam; i++) {
368     // Se levanta un elemento
369     par = (*listaNombreHashReg)[i];
370     // Si se encuentra, se guarda el hash
371     if (par.first ≡ nombreArchivo) {
372       hash = par.second;
373       break;
374     }
375   }
376   return hash;
377 }
378
379
380 // Devuelve el hash del bloque de un archivo.
381 // PRE: 'nombreArchivo' es el nombre de archivo del bloque; 'numBloque'
382 // es el número de bloque del que se desea obtener el hash.
383 // POST: se devuelve una cadena con el hash del bloque.
384 std::string ManejadorDeArchivos::obtenerHashDeBloque(
385   const std::string& nombreArchivo, int numBloque) {
386   // Tomamos el contenido del bloque
387   std::string contenido = this→obtenerContenido(nombreArchivo, numBloque);
388   int cantTotalBloques = this→obtenerCantBloques(nombreArchivo);
389
390   // Si el bloque es el mayor de todos, le quitamos el relleno de ceros
391   if(cantTotalBloques ≡ numBloque) {
392     unsigned int bytes = this→obtenerCantBytes(nombreArchivo);
393
394     // Quiltamos el relleno
```



```
395     contenido = contenido.substr(0, (bytes * 2) − (TAMANIO_BLOQUE *
396       (numBloque − 1)));
397   }
398
399   // Si no existe el bloque, no devolvemos nada
400   if(contenido ≡ "") return "";
401
402   return Hash::funcionDeHash(contenido);
403 }
404
405
406 // Devuelve el contenido de un archivo en formato hexadecimal expresado
407 // en una cadena de caracteres.
408 // PRE: 'nombreArchivo' es el nombre de archivo; 'numBloque' es el
409 // número de bloque que se desea obtener el archivo. Si no se especifica
410 // o si es cero, se devuelve el contenido completo del archivo.
411 std::string ManejadorDeArchivos::obtenerContenido(
412   const std::string& nombreArchivo, int numBloque) {
413   // Bloqueamos el mutex
414   Lock l(mArc);
415
416   // Armamos la ruta hacia el archivo
417   std::string ruta = this→directorio + "/" + nombreArchivo;
418
419   // Abrimos el archivo
420   std::ifstream archivo(ruta.c_str(),
421     std::ios::in | std::ios::binary | std::ios::ate);
422
423   if(¬archivo.is_open())
424     throw "ERROR: Archivo de entrada inválido.";
425
426   // Variables auxiliares
427   int size;
428   uint8_t * contenidoTemp;
429   int inicio, fin;
430
431   // Almacenamos momentaneamente el contenido del archivo original
432   size = archivo.tellg();
433
434   if (numBloque ≡ 0) {
435     inicio = 0;
436     fin = size;
437   }
438   else {
439     inicio = (TAMANIO_BLOQUE / 2) * (numBloque − 1);
440     fin = inicio + (TAMANIO_BLOQUE / 2);
441   }
442
443   if(inicio < fin) {
444     // Se posiciona en el bloque correspondiente
445     archivo.seekg(inicio);
446
447     // Se crea e inicializa una variable contenedora
448     contenidoTemp = new uint8_t[fin − inicio];
449     memset(contenidoTemp, '\0', fin − inicio);
450
451     // se lee del archivo
452     archivo.read((char*)contenidoTemp, fin − inicio);
453
454     // Convertimos el contenido a hexadecimal
455     std::string contenidoHex(Convertir::uitoh((uint8_t*)contenidoTemp, (size_t)(
    fin − inicio)));
456
457     // Se devuelve el contenido
458     delete[] contenidoTemp;
459     return contenidoHex;
```

```
460   }
461
462   archivo.close();
463
464   return "";
465 }
466
467
468 // Devuelve la cantidad de bloques de un archivo
469 // Si no existe el archivo, devuelve -1
470 int ManejadorDeArchivos::obtenerCantBloques(const std::string &nombreArchivo) {
471   // Armamos la ruta hacia el archivo
472   std::string ruta = this→directorio + "/" + nombreArchivo;
473
474   // Variables auxiliares
475   int longitud, cantBloques = -1;
476
477   // Se busca el archivo
478   std::fstream archivo;
479
480   // Se abre el archivo
481   archivo.open(ruta.c_str(), std::ios_base::in | std::ios_base::binary
482     | std::ios_base::ate);
483
484   // Si no existe, se devuelve -1
485   if (archivo.is_open()) {
486     // Se obtiene longitud archivo y se la multiplica por 2 para
487     // considerarlo en hexa
488     longitud = archivo.tellg();
489
490     // Se calcula cantBloques
491     cantBloques = floor((double)(longitud/(TAMANIO_BLOQUE/2)));
492
493     if (longitud % (TAMANIO_BLOQUE/2) > 0)
494       cantBloques++;
495   }
496
497   // Se devuelve la cantidad de bloques hexadecimales que hay
498   return cantBloques;
499 }
500
501
502 // Devuelve la cantidad de Bytes de un archivo
503 // PRE: 'nombreArchivo' es el nombre de archivo. El archivo no debe
504 // sobrepasar los 4Gb de tamaño.
505 // POST: en caso de no poder abrir el archivo (a causa de no existencia),
506 // se devuelve 0. Se recomienda al usuario verificar la existencia
507 // previamente para no confundir el cero de error con el valor nulo de
508 // que puede poseer cierto archivo.
509 unsigned int ManejadorDeArchivos::obtenerCantBytes(
510   const std::string &nombreArchivo) {
511   // Bloqueamos el mutex
512   Lock l(mArc);
513
514   // Variables auxiliares
515   unsigned int cantBytes = 0;
516
517   // Armamos la ruta hacia el archivo
518   std::string ruta = this→directorio + "/" + nombreArchivo;
519
520   // Abrimos archivo
521   std::ifstream archivo(ruta.c_str(),
522     std::ios::in | std::ios::binary | std::ios::ate);
523
524   // Si no pudo ser abierto, lo pasamos por alto
525   if(¬archivo.is_open()) return 0;
```



```
526
527   // Contabilizamos bytes
528   cantBytes = archivo.tellg();
529
530   // Cerramos archivo
531   archivo.close();
532
533   return cantBytes;
534 }
535
536
537 // Recibe una lista de archivos, compara con la que se encuentra localmente
538 // * ListaExterna: lista de archivos con la cual se compara
539 // * Faltantes: lista de archivos que estan modificados en el dir local
540 // * Sobrantes: lista de archivos que no estan en la lista que se deben
541 // eliminar del dir local
542 // * Nuevos: lista de archivos que no estan en el dir local
543 void ManejadorDeArchivos::obtenerListaDeActualizacion(
544   Lista< std::pair< std::string, std::pair< std::string,
545   int > > >* listaExterna, Lista< std::pair< std::string,
546   Lista<int> > >* faltantes, Lista<std::string>* sobrantes) {
547   // La primer lista contiene nombre, hash y cantidad de bloques (en ese
548   // orden). La segunda tiene hash y una cola de numeros de bloque.
549
550   // Variables auxiliares
551   std::pair< std::string, std::pair< std::string, int > > externo;
552   std::string hash, registro;
553
554   // Si esta vacia, no realizar accion
555   if (listaExterna→estaVacia())
556     return;
557
558   // Se crea una lista y se guarda una lista de archivos en registro
559   Lista<std::string> listaRegistro;
560   Lista< std::pair< std::string, std::string> > listaNombreHashReg;
561   obtenerArchivosDeRegistro(&listaRegistro, &listaNombreHashReg);
562
563   // Iterador para la lista a comparar y su tamanio
564   int it = 0;
565   int tam = listaExterna→tamanio();
566
567   for (it = 0; it < tam; it++) {
568     // Se lee un item
569     externo = (*listaExterna)[it];
570
571     // Si no existe en el local o en el registro,
572     // lo guarda en faltantes
573     if (¬existeArchivo(externo.first) ∨
574       ¬listaRegistro.buscar(externo.first)) {
575
576       // Se pide todo el archivo
577       Lista<int> bloques;
578       bloques.insertarUltimo(0);
579       std::pair< std::string, Lista<int> > aPedir =
580         std::make_pair(externo.first, bloques);
581       faltantes→insertarUltimo(aPedir);
582     }
583     // Existe el archivo en registro, entonces se comparan
584     // los respectivos hashes
585     else {
586       // Se obtiene el hash
587       hash = obtenerHashRegistro(&listaNombreHashReg,
588         externo.first);
589
590       // Si son distintos los hashes
591       if (hash ≠ externo.second.first) {
```

```
592           Lista<int> bloques;
593
594           // Se buscan las diferencias
595           obtenerDiferencias(hash, externo.second.first,
596             externo.second.second, &bloques);
597
598           // Se deben pedir las diferencias
599           if (¬bloques.estaVacia()) {
600             std::pair< std::string, Lista<int> > aPedir =
601               std::make_pair(externo.first, bloques);
602             faltantes→insertarUltimo(aPedir);
603           }
604         }
605         // Si los hashes son iguales, entonces se busca el
606         // archivo y se compara realmente
607         else {
608           obtenerHash(externo.first, hash);
609           // Si son distintos los hashes
610           if (hash ≠ externo.second.first) {
611             Lista<int> bloques;
612
613             // Se buscan las diferencias
614             obtenerDiferencias(hash, externo.second.first,
615               externo.second.second, &bloques);
616
617             // Se deben pedir las diferencias
618             if (¬bloques.estaVacia()) {
619               std::pair< std::string, Lista<int> > aPedir =
620                 std::make_pair(externo.first, bloques);
621               faltantes→insertarUltimo(aPedir);
622             }
623           }
624         }
625         // Se elimina para luego saber los que sobran
626         listaRegistro.eliminar(externo.first);
627       }
628     }
629     // Se guardan en lista de sobrantes los que queden en lista registro
630     tam = listaRegistro.tamanio();
631
632     for (it = 0; it < tam; it++) {
633       registro = listaRegistro[it];
634       sobrantes→insertarUltimo(registro);
635     }
636   }
637
638
639   // Crea un archivo de registro.
640   // POST: devuelve true si se realizó la creación con éxito o false en su
641   // defecto. Si ya se encuentra existente también devuelve false.
642   bool ManejadorDeArchivos::crearRegistroDeArchivos() {
643     // Comprobamos si existia previamente el archivo
644     if(this→existeRegistroDeArchivos()) return false;
645
646     // Armamos ruta del registro
647     std::string registro = this→directorio + DIR_AU + "/"
648       + ARCHIVO_REG_ARCHIVOS;
649
650     // Creamos la carpeta que contiene los registros
651     mkdir((this→directorio + DIR_AU).c_str(), S_IRWXU | S_IRWXG | S_IROTH |
652       S_IXOTH);
653
654     // Creamos el registro
655     std::ofstream archivo;
656     archivo.open(registro.c_str(), std::ios::out);
657
```



```
658     return true;
659   }
660
661
662   // Actualiza el registro local de archivos.
663   // PRE: 'nuevos', 'modificados' y 'eliminados' son punteros a cola donde
664   // se insertarán los nombres de archivo correspondientes a la situación
665   // en la que se encuentren.
666   // POST: se devuelve 'false' si se produjeron cambios en el registro o
667   // 'true' en su defecto; esto evita tener que revisar las colas para
668   // comprobar cambios.
669   bool ManejadorDeArchivos::actualizarRegistroDeArchivos(
670     Cola< std::pair< std::string, std::string > > *nuevos,
671     Cola< std::pair< std::string, Lista<int> > > *modificados,
672     Cola< std::string > *eliminados){
673     // Bloqueamos el mutex
674     Lock l(mReg);
675
676     // Variables auxiliares
677     std::ifstream registro;
678     std::ofstream registroTmp;
679     bool huboCambio = false;
680
681     // Armamos rutas de archivos
682     std::string regNombre = this→directorio + "/" + DIR_AU + "/"
683       + ARCHIVO_REG_ARCHIVOS;
684     std::string regTmpNombre = this→directorio + "/" + DIR_AU + "/"
685       + ARCHIVO_REG_ARCHIVOS + "~";
686
687     // Eliminamos posible registro temporal basura
688     remove(regTmpNombre.c_str());
689
690     // Abrimos el registro original y el registro temporal
691     registro.open(regNombre.c_str(), std::ios::in);
692     registroTmp.open(regTmpNombre.c_str(), std::ios::app);
693
694     // Verificamos si la apertura fue exitosa
695     if(¬registro.is_open() ∨ ¬registroTmp.is_open()) {
696       // Creamos registros de archivos que pueden faltar por eliminación
697       // manual
698       this→crearRegistroDeArchivos();
699
700       // Intentamos abrir nuevamente el registro original y el
701       // registro temporal
702       registro.open(regNombre.c_str(), std::ios::in);
703       registroTmp.open(regTmpNombre.c_str(), std::ios::app);
704
705       if(¬registro.is_open() ∨ ¬registroTmp.is_open())
706         throw "ERROR: El registro no pudo ser abierto.";
707     }
708
709     // Relevamos los nombres de archivos ubicados actualmente en el directorio
710     Lista< std::string > ld;
711     this→obtenerArchivosDeDirectorio(&ld);
712
713     // Variables auxiliares de procesamiento
714     std::string reg_archivoNombre, reg_archivoHash;
715     std::string buffer;
716     bool eof = false;
717
718     // Tomamos el primer registro
719     if(¬std::getline(registro, buffer)) eof = true;
720     this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
721
722     // Iteramos sobre los nombres de archivos existentes en el directorio
723     for(size_t i = 0; i < ld.tamanio(); i++) {
```

```
724     // Caso en el que no hay mas registros y se han agregado archivos
725     if(eof) {
726
727       std::string hashNuevo;
728
729       // Si se lanza un error por la no existencia de archivo seguimos
730       try {
731         // Calculamos el hash del archivo nuevo
732         this→obtenerHash(ld[i], hashNuevo);
733       }
734       catch(...){
735         continue;
736       }
737
738       // Registramos archivo nuevo
739       registroTmp << ld[i] << DELIMITADOR << hashNuevo << std::endl;
740
741       // Insertamos archivo en cola de nuevos
742       nuevos→push(std::make_pair(ld[i], hashNuevo));
743
744       huboCambio = true;
745       continue;
746     }
747
748     // Caso en que se han eliminado archivos
749     while(ld[i] > reg_archivoNombre ∧ ¬eof) {
750       // Insertamos en cola de eliminados
751       eliminados→push(reg_archivoNombre);
752
753       // Tomamos el registro siguiente
754       buffer.clear();
755       if(¬std::getline(registro, buffer)) eof = true;
756       this→separarNombreYHash(buffer, reg_archivoNombre,
757         reg_archivoHash);
758
759       huboCambio = true;
760     }
761
762     // Caso en el que el archivo se mantiene existente
763     if(ld[i] ≡ reg_archivoNombre) {
764       // Corroboramos si ha sido modificado
765       std::string hash_aux;
766
767       // Si se lanza un error por la no existencia de archivo seguimos
768       try {
769         this→obtenerHash(reg_archivoNombre, hash_aux);
770       }
771       catch(...) {
772         continue;
773       }
774
775       int cantBloques_aux = this→obtenerCantBloques(reg_archivoNombre);
776       Lista<int> listaDiferencias;
777
778       // Caso en que el archivo ha sido modificado
779       if(this→obtenerDiferencias(reg_archivoHash,
780         hash_aux, cantBloques_aux, &listaDiferencias)) {
781
782         // Actualizamos el hash del archivo
783         registroTmp << reg_archivoNombre << DELIMITADOR <<
784           hash_aux << std::endl;
785
786         // Insertamos archivo en cola de modificados
787         modificados→push(make_pair(reg_archivoNombre,
788           listaDiferencias));
789
```



```
790         huboCambio = true;
791       }
792       // Caso en que no ha sido modificado
793       else {
794         registroTmp << reg_archivoNombre << DELIMITADOR
795           << hash_aux << std::endl;
796       }
797
798       // Tomamos el registro siguiente
799       buffer.clear();
800       if(¬std::getline(registro, buffer)) eof = true;
801       this→separarNombreYHash(buffer, reg_archivoNombre,
802         reg_archivoHash);
803     }
804     // Caso en el que se han agregado nuevos archivos
805     else if(ld[i] < reg_archivoNombre ∨ eof) {
806       // Calculamos el hash del archivo nuevo
807       std::string hashNuevo;
808
809       // Si se lanza un error por la no existencia de archivo seguimos
810       try {
811         this→obtenerHash(ld[i], hashNuevo);
812       }
813       catch(...) {
814         continue;
815       }
816
817       // Registramos archivo nuevo
818       registroTmp << ld[i] << DELIMITADOR << hashNuevo << std::endl;
819
820       // Insertamos archivo en cola de nuevos
821       nuevos→push(std::make_pair(ld[i], hashNuevo));
822
823       huboCambio = true;
824     }
825   }
826
827   // Encolamos los últimos registros pertenecientes a archivos eliminados
828   while(¬eof) {
829     eliminados→push(reg_archivoNombre);
830
831     // Tomamos el registro siguiente
832     buffer.clear();
833     if(¬std::getline(registro, buffer)) eof = true;
834     this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
835
836     huboCambio = true;
837   }
838
839
840   // Cerramos archivos
841   registro.close();
842   registroTmp.close();
843
844   // Eliminamos el registro original y convertimos el temporal en el oficial
845   remove(regNombre.c_str());
846   rename(regTmpNombre.c_str(), regNombre.c_str());
847
848   return huboCambio;
849 }
850
851
852 // Actualiza el registro local de archivos.
853 // PRE: las listas corresponden a que archivos nuevos o modificados deben
854 // tenerse en cuenta, siendo que los demás detectados en el momento de la
855 // actualización, son salteados.
```

```
856 // POST: se devuelve 'false' si se produjeron cambios en el registro o
857 // 'true' en su defecto; esto evita tener que revisar las colas para
858 // comprobar cambios.
859 bool ManejadorDeArchivos::actualizarRegistroDeArchivos(
860   Lista< std::string >& nuevosActualizables,
861   Lista< std::string >& modificadosActualizables) {
862   // Bloqueamos el mutex
863   Lock l(mReg);
864
865   // Variables auxiliares
866   std::ifstream registro;
867   std::ofstream registroTmp;
868   bool huboCambio = false;
869
870   // Armamos rutas de archivos
871   std::string regNombre = this→directorio + "/" + DIR_AU + "/"
872     + ARCHIVO_REG_ARCHIVOS;
873   std::string regTmpNombre = this→directorio + "/" + DIR_AU + "/"
874     + ARCHIVO_REG_ARCHIVOS + "~";
875
876   // Eliminamos posible registro temporal basura
877   remove(regTmpNombre.c_str());
878
879   // Abrimos el registro original y el registro temporal
880   registro.open(regNombre.c_str(), std::ios::in);
881   registroTmp.open(regTmpNombre.c_str(), std::ios::app);
882
883   // Verificamos si la apertura fue exitosa
884   if(¬registro.is_open() ∨ ¬registroTmp.is_open())
885     throw "ERROR: El registro no pudo ser abierto.";
886
887   // Relevamos los nombres de archivos ubicados actualmente en el directorio
888   Lista< std::string > ld;
889   this→obtenerArchivosDeDirectorio(&ld);
890
891   // Variables auxiliares de procesamiento
892   std::string reg_archivoNombre, reg_archivoHash;
893   std::string buffer;
894   bool eof = false;
895
896   // Tomamos el primer registro
897   if(¬std::getline(registro, buffer)) eof = true;
898   this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
899
900   // Iteramos sobre los nombres de archivos existentes en el directorio
901   for(size_t i = 0; i < ld.tamanio(); i++) {
902     // Caso en el que no hay mas registros y se han agregado archivos
903     if(eof) {
904       // Si no está en la lista de nuevos actualizables salteamos
905       if(¬nuevosActualizables.buscar(ld[i]))
906         continue;
907
908       // Calculamos el hash del archivo nuevo
909       std::string hashNuevo;
910
911       // Si se lanza un error por la no existencia de archivo seguimos
912       try {
913         this→obtenerHash(ld[i], hashNuevo);
914       }
915       catch(...) {
916         continue;
917       }
918
919       // Registramos archivo nuevo
920       registroTmp << ld[i] << DELIMITADOR << hashNuevo << std::endl;
921
```



```
922       huboCambio = true;
923       continue;
924     }
925
926     // Caso en que se han eliminado archivos
927     while(ld[i] > reg_archivoNombre ∧ ¬eof) {
928       // Tomamos el registro siguiente
929       buffer.clear();
930       if(¬std::getline(registro, buffer)) eof = true;
931       this→separarNombreYHash(buffer, reg_archivoNombre,
932         reg_archivoHash);
933
934       huboCambio = true;
935     }
936
937     // Caso en el que el archivo se mantiene existente
938     if(ld[i] ≡ reg_archivoNombre) {
939       // Corroboramos si ha sido modificado
940       std::string hash_aux;
941
942       // Si se lanza un error por la no existencia de archivo seguimos
943       try {
944         this→obtenerHash(reg_archivoNombre, hash_aux);
945       }
946       catch(...) {
947         continue;
948       }
949
950       int cantBloques_aux = this→obtenerCantBloques(reg_archivoNombre);
951       Lista<int> listaDiferencias;
952
953       // Caso en que el archivo ha sido modificado
954       if(modificadosActualizables.buscar(ld[i]) ∧
955         this→obtenerDiferencias(reg_archivoHash, hash_aux,
956           cantBloques_aux, &listaDiferencias)) {
957         // Actualizamos el hash del archivo
958         registroTmp << reg_archivoNombre << DELIMITADOR <<
959           hash_aux << std::endl;
960
961         huboCambio = true;
962       }
963       // Caso en que no ha sido modificado
964       else {
965         registroTmp << reg_archivoNombre << DELIMITADOR
966           << hash_aux << std::endl;
967       }
968
969       // Tomamos el registro siguiente
970       buffer.clear();
971       if(¬std::getline(registro, buffer)) eof = true;
972       this→separarNombreYHash(buffer, reg_archivoNombre,
973         reg_archivoHash);
974     }
975     // Caso en el que se han agregado nuevos archivos
976     else if(ld[i] < reg_archivoNombre ∨ eof) {
977       // Si no está en la lista de nuevos actualizables salteamos
978       if(¬nuevosActualizables.buscar(ld[i]))
979         continue;
980
981       // Calculamos el hash del archivo nuevo
982       std::string hashNuevo;
983
984       // Si se lanza un error por la no existencia de archivo seguimos
985       try {
986         this→obtenerHash(ld[i], hashNuevo);
987       }
```

```
988         catch(...) {
989           continue;
990         }
991
992         // Registramos archivo nuevo
993         registroTmp << ld[i] << DELIMITADOR << hashNuevo << std::endl;
994
995         huboCambio = true;
996       }
997     }
998
999     // Cerramos archivos
1000    registro.close();
1001    registroTmp.close();
1002
1003    // Eliminamos el registro original y convertimos el temporal en el oficial
1004    remove(regNombre.c_str());
1005    rename(regTmpNombre.c_str(), regNombre.c_str());
1006
1007    return huboCambio;
1008  }
1009
1010
1011  // Elimina el registro que identifica a un archivo en el registro de
1012  // archivos.
1013  // PRE: 'nombreArchivo' es el nombre del archivo a eliminar del registro.
1014  void ManejadorDeArchivos::borrarDeRegistroDeArchivos(
1015    const std::string& nombreArchivo) {
1016    // Bloqueamos el mutex
1017    Lock l(mReg);
1018
1019    // Variables auxiliares
1020    std::ifstream registro;
1021    std::ofstream registroTmp;
1022
1023    // Armamos rutas de archivos
1024    std::string regNombre = this→directorio + "/" + DIR_AU + "/"
1025      + ARCHIVO_REG_ARCHIVOS;
1026    std::string regTmpNombre = this→directorio + "/" + DIR_AU + "/"
1027      + ARCHIVO_REG_ARCHIVOS + "~";
1028
1029    // Eliminamos posible registro temporal basura
1030    remove(regTmpNombre.c_str());
1031
1032    // Abrimos el registro original y el registro temporal
1033    registro.open(regNombre.c_str(), std::ios::in);
1034    registroTmp.open(regTmpNombre.c_str(), std::ios::app);
1035
1036    // Verificamos si la apertura fue exitosa
1037    if(¬registro.is_open() ∨ ¬registroTmp.is_open())
1038      throw "ERROR: El registro no pudo ser abierto.";
1039
1040
1041    // Variables auxiliares de procesamiento
1042    std::string reg_archivoNombre, reg_archivoHash;
1043    std::string buffer;
1044
1045    // Iteramos sobre los registros obviando el que coincide con el que
1046    // se desea eliminar.
1047    while(std::getline(registro, buffer)) {
1048      // Parseamos el nombre de archivo
1049      this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
1050
1051      // Obviamos la copia si coinciden los nombres de archivo
1052      if(reg_archivoNombre ≡ nombreArchivo) continue;
1053
```



```
1054      // Mantenemos el registro que no se desea borrar
1055      registroTmp << buffer << std::endl;
1056    }
1057
1058    // Cerramos archivos
1059    registro.close();
1060    registroTmp.close();
1061
1062    // Eliminamos el registro original y convertimos el temporal en el oficial
1063    remove(regNombre.c_str());
1064    rename(regTmpNombre.c_str(), regNombre.c_str());
1065  }
1066
1067
1068  // Comprueba si existe el registro de archivos.
1069  // POST: devuelve true si existe o false en su defecto.
1070  bool ManejadorDeArchivos::existeRegistroDeArchivos() {
1071    // Armamos ruta del registro
1072    std::string registro = this→directorio + DIR_AU + "/"
1073      + ARCHIVO_REG_ARCHIVOS;
1074
1075    // Tratamos de abrir el archivo
1076    std::ifstream archivo;
1077    archivo.open(registro.c_str());
1078
1079    // Si no se abrió retornamos false
1080    if(¬archivo.good()) return false;
1081
1082    // Si se abrió, lo cerramos y retornamos true
1083    archivo.close();
1084    return true;
1085  }
1086
1087
1088  // Corrobora si se encuentra registrado un archivo en el registro de
1089  // archivos.
1090  // PRE: 'nombreArchivo' es el nombre de archivo.
1091  // POST: devuelve true si el archivo se encuentra registrado o false
1092  // en caso contrario.
1093  bool ManejadorDeArchivos::existeArchivoEnRegitro(
1094    const std::string nombreArchivo) {
1095    // Bloqueamos el mutex
1096    Lock l(mReg);
1097
1098    // Variables auxiliares
1099    std::ifstream registro;
1100    std::string reg_archivoNombre, reg_archivoHash;
1101    std::string buffer;
1102
1103    // Armamos rutas de archivos
1104    std::string regNombre = this→directorio + "/" + DIR_AU + "/"
1105      + ARCHIVO_REG_ARCHIVOS;
1106
1107
1108    // Abrimos el archivo de registros
1109    registro.open(regNombre.c_str(), std::ios::in);
1110
1111    // Verificamos si la apertura fue exitosa
1112    if(¬registro.is_open())
1113      throw "ERROR: El registro no pudo ser abierto.";
1114
1115    // Iteramos sobre las lineas del archivo para buscar el registro
1116    while(std::getline(registro, buffer)) {
1117      // Parseamos
1118      this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
1119
```

jun 25, 13 13:44 **common_manejador_de_archivos.cpp** Page 18/20

```
1120      // Comparamos nombre de archivo del registro con el buscado
1121      if(reg_archivoNombre ≡ nombreArchivo) {
1122        // Cerramos archivos
1123        registro.close();
1124
1125        // Se encontró registro
1126        return true;
1127      }
1128
1129      buffer.clear();
1130    }
1131
1132    // Cerramos archivos
1133    registro.close();
1134
1135    // No se encontró registro
1136    return false;
1137  }
1138
1139
1140  // Compara el hash actual de cierto bloque de archivo con un hash pasado
1141  // por parámetro.
1142  // PRE: 'nombreArchivo' es el nombre de archivo; 'numBloque' es el
1143  // número del bloque que se desea comparar; 'hash' es el hash que
1144  // se comparará con el del bloque del archivo.
1145  // POST: devuelve true si son iguales o false si presentan diferencias.
1146  bool ManejadorDeArchivos::compararBloque(const std::string& nombreArchivo,
1147    const int numBloque, const std::string& hash) {
1148    // Bloqueamos el mutex
1149    Lock l(mReg);
1150
1151    // Variables auxiliares
1152    std::ifstream registro;
1153    std::string reg_archivoNombre, reg_archivoHash;
1154    std::string buffer;
1155
1156    // Armamos rutas de archivos
1157    std::string regNombre = this→directorio + "/" + DIR_AU + "/"
1158      + ARCHIVO_REG_ARCHIVOS;
1159
1160
1161    // Abrimos el archivo de registros
1162    registro.open(regNombre.c_str(), std::ios::in);
1163
1164    // Verificamos si la apertura fue exitosa
1165    if(¬registro.is_open())
1166      throw "ERROR: El registro no pudo ser abierto.";
1167
1168    // Iteramos sobre las lineas del archivo para buscar el registro
1169    while(std::getline(registro, buffer)) {
1170      // Parseamos
1171      this→separarNombreYHash(buffer, reg_archivoNombre, reg_archivoHash);
1172
1173      // Comparamos nombre de archivo del registro con el buscado
1174      if(reg_archivoNombre ≡ nombreArchivo) {
1175        bool coincide = false;
1176
1177        // Corroboramos si la cantidad de bloques es menor al pasado
1178        if(¬(this→obtenerCantBloques(nombreArchivo) < numBloque)) {
1179          // Extraemos el hash del bloque
1180          std::string hashBloque = reg_archivoHash.substr((numBloque − 1)
1181            * TAMANIO_BLOQUE_HASH, TAMANIO_BLOQUE_HASH);
1182
1183          // Comparamos la igualdad de hashes
1184          if(hashBloque ≡ hash) coincide = true;
1185        }
```

jun 25, 13 13:44 **common_manejador_de_archivos.cpp** Page 19/20

```
1186
1187        // Cerramos archivos
1188        registro.close();
1189
1190        // Se encontró registro
1191        return coincide;
1192      }
1193
1194      buffer.clear();
1195    }
1196
1197    // Cerramos archivos
1198    registro.close();
1199
1200    // No se encontró registro
1201    return false;
1202  }
1203
1204
1205
1206
1207
1208
1209  /*
1210   * IMPLEMENTACIÓN DE MÉTODOS PRIVADOS DE LA CLASE
1211   */
1212
1213
1214
1215  // Procesa dos hashes pertenecientes al contenido de un archivo y
1216  // obtiene los bloques que han cambiado.
1217  // PRE: 'hashViejo' y 'hashNuevo' son los hashes de los archivos a
1218  // procesar; 'cantNuevaBloques' es la cantidad de bloques del archivo
1219  // que es representado por 'hashNuevo'
1220  // POST: se listan en 'listaBLoquesDiferentes' los numero de bloques
1221  // que han cambiado; Se devuelve true si se encontraron diferencias o
1222  // false en caso contrario.
1223  bool ManejadorDeArchivos::obtenerDiferencias(std::string& hashViejo,
1224    std::string& hashNuevo, int& cantNuevaBloques,
1225    Lista<int> *listaBloquesDiferentes) {
1226    // Si los hashes refieren a archivos vacios, devolvemos false
1227    if(hashViejo ≡ "" ∧ hashNuevo ≡ "") return false;
1228
1229    int cantViejaBloques = (hashViejo.size()) / TAMANIO_BLOQUE_HASH;
1230
1231    // Caso en que el tamaño del hashViejo es nulo
1232    if(hashViejo.size() ≡ 0) {
1233      // Insertamos todos los números de bloques
1234      for(int i = 1; i ≤ cantNuevaBloques; i++)
1235        listaBloquesDiferentes→insertarUltimo(i);
1236
1237      return true;
1238    }
1239
1240    // Flag para notificar cambios
1241    bool hubieronCambios = false;
1242
1243
1244    // Contador de bloques de hash viejo
1245    int i = 0;
1246
1247    // Iteramos sobre los bloques
1248    while((i < cantViejaBloques) ∧ (i < cantNuevaBloques)) {
1249      // Obtenemos el bloque i del hash viejo
1250      std::string bloqueViejo = hashViejo.substr(i * TAMANIO_BLOQUE_HASH,
1251        TAMANIO_BLOQUE_HASH);
```

jun 25, 13 13:44 **common_manejador_de_archivos.cpp** Page 20/20

```
1252
1253      // Obtenemos el bloque i del hash nuevo
1254      std::string bloqueNuevo = hashNuevo.substr(i * TAMANIO_BLOQUE_HASH,
1255        TAMANIO_BLOQUE_HASH);
1256
1257      // Incrementamos contador de bloques
1258      i++;
1259
1260      // Si son iguales los bloques, sigo de largo
1261      if(bloqueViejo ≡ bloqueNuevo) continue;
1262
1263      // Insertamos el número de bloque en la lista
1264      listaBloquesDiferentes→insertarUltimo(i);
1265      hubieronCambios = true;
1266    }
1267
1268    // Agregamos bloques remanentes
1269    while(i < cantNuevaBloques) {
1270      // Insertamos el número de bloque en la lista
1271      listaBloquesDiferentes→insertarUltimo(i + 1);
1272      hubieronCambios = true;
1273
1274      // Incrementamos el número de bloque
1275      i++;
1276    }
1277
1278    // Caso en que no hay cambios pero se ha achicado el tamaño del hash
1279    if(¬hubieronCambios ∧ (cantViejaBloques > cantNuevaBloques))
1280      hubieronCambios = true;
1281
1282    return hubieronCambios;
1283  }
1284
1285
1286  // Separa de una linea el nombre y el hash
1287  void ManejadorDeArchivos::separarNombreYHash(const std::string &linea,
1288    std::string& nombre, std::string &hash) {
1289    // Se limpian las variables
1290    nombre.clear();
1291    hash.clear();
1292
1293    // Se separa si hay contenido
1294    if (¬linea.empty()) {
1295      // Se busca el ultimo espacio
1296      int delim = linea.find_last_of(DELIMITADOR[0]);
1297
1298      // Se guarda nombre y hash por separado
1299      nombre = linea.substr(0, delim);
1300
1301      hash = linea.substr(delim + 1);
1302    }
1303  }
```

jun 25, 13 13:44 **common_logger.h** Page 1/1

```
1  //
2  //  common_logger.h
3  //  CLASE LOGGER
4  //
5  //  Clase que escribe un en un archivo de log "<Fecha> <mensaje de log>"
6  //
7
8
9  #ifndef LOGGER_H
10 #define LOGGER_H
11
12 #include <time.h>
13 #include <iostream>
14 #include <fstream>
15 #include <stdio.h>
16 #include "common_mutex.h"
17 #include "common_lock.h"
18
19 namespace {
20   #define LONG_FECHA 24
21 }
22
23
24
25 /* ******************************************************************************
26  * DECLARACIÓN DE LA CLASE
27  * ******************************************************************************/
28
29
30 class Logger {
31 private:
32   Mutex* mutex;
33   std::fstream* archivo;
34   std::string pathArchivo;
35
36   void crearArchivo();
37
38 public:
39
40   // Constructores
41   Logger(const std::string& nombre_archivo);
42
43   Logger(const char* nombre_archivo);
44
45   // Destructor
46   ~Logger();
47
48   // Crea una nueva entrada de log.
49   // PRE: 'log' es el mensaje que se insertará en el log.
50   void emitirLog(const std::string& log);
51
52   // Limpia el archivo de log existente eliminando todas las entradas
53   // contenidas en él. Devuelve 0 si se eliminaron las entradas
54   // correctamente o un valor distinto de 0 sino.
55   int limpiarLog();
56 };
57
58 #endif
```

```
1  #include <string>
2  #include "common_logger.h"
3
4
5  // Constantes
6  namespace {
7    const std::string LOG_EXTENSION = ".log";
8  }
9
10
11
12
13 Logger::Logger(const std::string& nombre_archivo) {
14   this→pathArchivo = nombre_archivo + LOG_EXTENSION;
15   crearArchivo();
16   this→mutex = new Mutex;
17 }
18
19 Logger::Logger(const char* nombre_archivo) {
20   this→pathArchivo.assign(nombre_archivo);
21   this→pathArchivo += LOG_EXTENSION;
22   crearArchivo();
23   this→mutex = new Mutex;
24 }
25
26 Logger::~Logger() {
27   if (this→mutex) {
28     delete(this→mutex);
29     this→mutex = NULL;
30   }
31   if (this→archivo) {
32     if (this→archivo→is_open())
33       this→archivo→close();
34     delete(this→archivo);
35     this→archivo = NULL;
36   }
37 }
38
39 void Logger::emitirLog(const std::string& log) {
40   Lock lock(*this→mutex);
41   // Se obtiene la hora y fecha actual
42   time_t tiempo;
43   time(&tiempo);
44   struct tm* infoTiempo;
45   infoTiempo = localtime(&tiempo);
46   // Se abre el archivo si es que no se encuentra abierto
47   if (¬this→archivo→is_open())
48     this→archivo→open(pathArchivo.c_str(), std::ios_base::out |
49       std::ios_base::app);
50   // Se va al final del archivo
51   this→archivo→seekp(0, std::ios_base::end);
52   // Se escribe en el log la fecha y luego el mensaje
53   this→archivo→write(asctime(infoTiempo), LONG_FECHA);
54   this→archivo→put(' ');
55   this→archivo→write(log.c_str(), log.length());
56   this→archivo→put('\n');
57   this→archivo→close();
58 }
59
60 int Logger::limpiarLog() {
61   Lock lock(*this→mutex);
62   int error;
63   // Se cierra y se intenta eliminar el archivo
64   if (this→archivo→is_open())
65     this→archivo→close();
66   delete(this→archivo);
```



```
67   this→archivo = NULL;
68   error = remove(pathArchivo.c_str());
69   if (error ≡ 0)  // Si logro eliminarlo,
70     // Se vuelve a crear el archivo con el mismo nombre
71     crearArchivo();
72   return error;
73 }
74
75 // Metodos privados
76
77 void Logger::crearArchivo() {
78   this→archivo = new std::fstream(pathArchivo.c_str(), std::ios_base::in
79     | std::ios_base::out | std::ios_base::app);
80     if (¬archivo→is_open()) {
81     archivo→clear();
82     //Se crea el archivo
83     archivo→open(pathArchivo.c_str(), std::fstream::out);
84     archivo→close();
85   }
86 }
```

jun 25, 13 13:44 **common_lock.h** Page 1/1

```
1  //
2  //  common_lock.h
3  //  CLASE LOCK
4  //
5  //  Clase que implementa el bloqueador del mutex.
6  //
7
8
9  #ifndef LOCK_H
10 #define LOCK_H
11
12
13 #include "common_mutex.h"
14
15
16
17
18
19 /* ******************************************************************************
20  * DECLARACIÓN DE LA CLASE
21  * ******************************************************************************/
22
23
24 class Lock {
25 private:
26
27   Mutex &mutex;       // Mutex
28
29   // Constructor privado
30   Lock(const Lock &c);
31
32 public:
33
34   // Constructor
35   explicit Lock(Mutex &m);
36
37   // Destructor
38   ~Lock();
39
40   // Bloquea el mutex;
41   void lock();
42
43   // Desbloquea el mutex;
44   void unlock();
45
46   // Bloquea la ejecución en un una condition variable hasta que se produzca
47   // una señalización.
48   void wait();
49
50   // Desbloquea al menos uno de los hilos que están bloqueados en la
51   // condition variable.
52   void signal();
53
54   // Desbloquea todos los hilos bloqueados actualmente en la condition
55   // variable.
56   void broadcast();
57 };
58
59 #endif
```

jun 25, 13 13:44 **common_lock.cpp** Page 1/1

```
1  //
2  //  common_lock.cpp
3  //  CLASE LOCK
4  //
5  //  Clase que implementa el bloqueador del mutex.
6  //
7
8
9  #include "common_lock.h"
10
11
12
13
14 /* ******************************************************************************
15  * DEFINICIÓN DE LA CLASE
16  * ******************************************************************************/
17
18
19 // Constructor
20 Lock::Lock(Mutex &m) : mutex(m) {
21   this→mutex.lock();
22 }
23
24
25 // Constructor privado
26 Lock::Lock(const Lock &c) : mutex(c.mutex) { }
27
28
29 // Destructor
30 Lock::~Lock() {
31   this→mutex.unlock();
32 }
33
34
35 // Bloquea el mutex;
36 void Lock::lock() {
37   this→mutex.lock();
38 }
39
40
41 // Desbloquea el mutex;
42 void Lock::unlock() {
43   this→mutex.unlock();
44 }
45
46
47 // Bloquea la ejecución en un una condition variable hasta que se produzca
48 // una señalización.
49 void Lock::wait() {
50   this→mutex.wait();
51 }
52
53
54 // Desbloquea al menos uno de los hilos que están bloqueados en la
55 // condition variable.
56 void Lock::signal() {
57   this→mutex.signal();
58 }
59
60
61 // Desbloquea todos los hilos bloqueados actualmente en la condition
62 // variable.
63 void Lock::broadcast() {
64   this→mutex.broadcast();
65 }
```

```
1   //
2   //  common_lista.h
3   //  CLASE LISTA
4   //
5   //  Clase que implementa una lista con la caracteristica de ser thread-safe.
6   //
7
8
9   #ifndef LISTA_H
10  #define LISTA_H
11
12
13  #include <list>
14  #include <algorithm>
15  #include "common_mutex.h"
16  #include "common_lock.h"
17
18
19
20
21  /* ******************************************************************************
22   * DECLARACIÓN DE LA CLASE
23   * ******************************************************************************/
24
25
26  template < typename Tipo >
27  class Lista {
28  private:
29
30    std::list< Tipo > lista;      // Lista
31    Mutex m;              // Mutex
32
33  public:
34
35    // Constructor
36    Lista();
37
38    // Constructor copia
39    Lista(const Lista< Tipo>& l);
40
41    // Destructor
42    ~Lista();
43
44    // Inserta un nuevo elemento al final de la lista.
45    // PRE: 'dato' es el dato a insertar.
46    void insertarUltimo(Tipo dato);
47
48    // Devuelve un puntero al primer elemento
49    Tipo verPrimero();
50
51    // Elimina el primer elemento de la lista.
52    // POST: se destruyó el elemento removido.
53    void eliminarPrimero();
54
55    // Elimina de la lista todos los elementos iguales al valor especificado.
56    // POST: se llama al destructor de estos elementos.
57    void eliminar(Tipo valor);
58
59    // Devuelve la cantidad de elementos contenidos en la lista.
60    size_t tamanio();
61
62    // Ordena los elementos de la lista (deben poder ser comparables)
63    void ordenar();
64
65    // Busca un elemento en la lista.
66    // PRE: 'valor' es el valor del elemento a buscar.
```



```
67    // POST: devuelve true si se encuentra o false en caso contrario
68    bool buscar(Tipo valor);
69
70    // Verifica si una lista se encuentra vacía.
71    // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
72    // caso contrario.
73    bool estaVacia();
74
75    // Elimina los elementos que estan en la lista.
76    // Si esta vacia, no realiza accion
77    void vaciar();
78
79    // Operador []
80    // Permite acceder a los índices de la lista mediante la notación
81    // lista[i], donde i es un número entero comprendido entre [0, n-1],
82    // siendo n el tamanio de la lista.
83    Tipo operator[] (const size_t indice);
84
85    // Asigna el contenido de una lista a otra
86    Lista& operator= (const Lista& lista);
87  };
88
89
90
91
92  /* ******************************************************************************
93   * DEFINICIÓN DE LA CLASE
94   * ******************************************************************************/
95
96
97  // Constructor
98  template <typename Tipo >
99  Lista< Tipo >::Lista() { }
100
101
102 // Constructor copia
103 template <typename Tipo >
104 Lista< Tipo >::Lista(const Lista< Tipo>& l) {
105   this→lista = l.lista;
106 }
107
108 // Destructor
109 template <typename Tipo >
110 Lista< Tipo >::~Lista() { }
111
112
113 // Inserta un nuevo elemento al final de la lista.
114 // PRE: 'dato' es el dato a insertar.
115 template <typename Tipo >
116 void Lista< Tipo >::insertarUltimo(Tipo dato) {
117   Lock l(m);
118   this→lista.push_back(dato);
119 }
120
121
122 // Devuelve un puntero al primer elemento
123 template <typename Tipo >
124 Tipo Lista< Tipo >::verPrimero() {
125   Lock l(m);
126   return this→lista.front();
127 }
128
129
130 // Elimina el primer elemento de la lista.
131 // POST: se destruyó el elemento removido.
132 template <typename Tipo >
```

```
133 void Lista< Tipo >::eliminarPrimero() {
134   Lock l(m);
135   this→lista.pop_front();
136 }
137
138
139 // Elimina de la lista todos los elementos iguales al valor especificado.
140 // POST: se llama al destructor de estos elementos.
141 template <typename Tipo >
142 void Lista< Tipo >::eliminar(Tipo valor) {
143   Lock l(m);
144   this→lista.remove(valor);
145 }
146
147
148 // Devuelve la cantidad de elementos contenidos en la lista.
149 template <typename Tipo >
150 size_t Lista< Tipo >::tamanio() {
151   Lock l(m);
152   return this→lista.size();
153 }
154
155 template <typename Tipo >
156 void Lista< Tipo >::ordenar() {
157   Lock l(m);
158   this→lista.sort();
159 }
160
161
162 // Busca un elemento en la lista.
163 // PRE: 'valor' es el valor del elemento a buscar.
164 // POST: devuelve true si se encuentra o false en caso contrario
165 template <typename Tipo >
166 bool Lista< Tipo >::buscar(Tipo valor) {
167   Lock l(m);
168
169   // Buscamos elemento en lista interna
170   typename std::list< Tipo >::iterator i;
171   i = std::find(this→lista.begin(), this→lista.end(), valor);
172
173   // Devolvemos el resultado de la busqueda
174   if(i ≠ this→lista.end()) return true;
175   return false;
176 }
177
178
179 // Verifica si una lista se encuentra vacía.
180 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
181 // caso contrario.
182 template <typename Tipo >
183 bool Lista< Tipo >::estaVacia() {
184   Lock l(m);
185   return this→lista.empty();
186 }
187
188 // Elimina los elementos que estan en la lista.
189 // Si esta vacia, no realiza accion
190 template <typename Tipo >
191 void Lista< Tipo >::vaciar() {
192   int i, tam = tamanio();
193   for (i = 0; i < tam; i++)
194     eliminarPrimero();
195 }
196
197
198 // Operador []
```



```
199 // Permite acceder a los índices de la lista mediante la notación lista[i],
200 // donde i es un número entero comprendido entre [0, n−1], siendo n el tamaño
201 // de la lista.
202 template <typename Tipo >
203 Tipo Lista< Tipo >::operator[] (const size_t indice) {
204   Lock l(m);
205
206   // Corroboramos que no este vacia
207   if(this→lista.empty())
208     throw "ERROR: Se intenta acceder a una lista que se encuentra vacia.";
209   // Corroboramos que el índice sea válido
210   else if(indice ≥ this→lista.size())
211     throw "ERROR: Indice de lista inválido.";
212
213   // Creamos iterador y nos posicionamos en el índice deseado
214   typename std::list< Tipo >::const_iterator it = this→lista.begin();
215   for(size_t i = 0; i < indice; i++) ++it;
216
217   return *it;
218 }
219
220
221 // Asigna el contenido de una lista a otra
222 template <typename Tipo >
223 Lista< Tipo >& Lista< Tipo >::operator= (const Lista< Tipo >& l) {
224   this→lista = l.lista;
225   return(*this);
226 }
227
228 #endif
```

jun 25, 13 13:44 **common_hash.h** Page 1/1

```
1  //
2  //  common_hash.h
3  //  LIBRERIA HASH
4  //
5  //  Librería de funciones de hash.
6  //
7
8
9  #ifndef HASH_H
10 #define HASH_H
11
12
13 #include <iostream>
14 #include <string.h>
15 #include "common_sha1.h"
16 #include "common_convertir.h"
17
18
19
20
21
22 /* ******************************************************************************
23  * DECLARACIÓN DE LA CLASE
24  * ******************************************************************************/
25
26
27 class Hash {
28 public:
29   // Devuelve la longitud de la salida del hash
30   static int longHash();
31
32   // Aplica la función de hash al string entrante y devuelve el resultado 'impri
   mible'
33   // en caracteres representando solamente numeros hexadecimales en mayuscula
34   static std::string funcionDeHash(std::string cadena);
35
36   // Aplica la función de hash al string entrante y devuelve el resultado binari
   o
37   static std::string funcionDeHashBin(std::string cadena);
38
39   // Aplica la función de hash al char* entrante
40   static std::string funcionDeHash(const char* cadena, int longitud);
41 };
42
43
44 #endif
```

jun 25, 13 13:44 **common_hash.cpp** Page 1/2

```
1  //
2  //  common_hash.cpp
3  //  LIBRERIA HASH
4  //
5  //  Librería de funciones de hash.
6  //
7
8
9  #include "common_hash.h"
10
11 namespace {
12   #define TAM_HASH 20
13   #define TAM_HEXA 42
14 }
15
16
17
18
19 /* ******************************************************************************
20  * DEFINICIÓN DE LA CLASE
21  * ******************************************************************************/
22
23
24 // Devuelve la longitud de la salida del hash
25 int Hash::longHash() {
26   return TAM_HASH;
27 }
28
29 // Aplica la función de hash al string entrante
30 std::string Hash::funcionDeHash(std::string cadena) {
31   // Variables auxiliares
32   unsigned char hash[TAM_HASH];
33   char hex_hash[TAM_HEXA];
34   std::string aux;
35
36   // Se inicializan
37   memset(hash, 0, TAM_HASH);
38   memset(hex_hash, 0, TAM_HEXA);
39
40   // Se calcula
41   sha1::calc(cadena.c_str(), cadena.length(), hash);
42
43   // Se transforma a hexa
44   sha1::toHexString(hash, hex_hash);
45
46   aux = hex_hash;
47
48   return aux;
49 }
50
51 // Aplica la función de hash al string entrante
52 std::string Hash::funcionDeHashBin(std::string cadena) {
53   // Variables auxiliares
54   unsigned char hash[TAM_HASH];
55   std::string aux;
56
57   // Se inicializan
58   memset(hash, 0, TAM_HASH);
59
60   // Se calcula
61   sha1::calc(cadena.c_str(), cadena.length(), hash);
62
63   aux = (char*)hash;
64
65   return aux;
66 }
```

jun 25, 13 13:44 **common_hash.cpp** Page 2/2

```
67
68  // Aplica la función de hash al char* entrante
69  std::string Hash::funcionDeHash(const char* cadena, int longitud) {
70    // Variables auxiliares
71    unsigned char hash[TAM_HASH];
72    char hex_hash[TAM_HEXA];
73    std::string aux;
74
75    // Se inicializan
76    memset(hash, 0, TAM_HASH);
77    memset(hex_hash, 0, TAM_HEXA);
78
79    // Se calcula
80    sha1::calc(cadena, longitud, hash);
81
82    // Se transforma a hexa
83    sha1::toHexString(hash, hex_hash);
84
85    aux = hex_hash;
86
87    return aux;
88  }
```

jun 25, 13 13:44 **common_convertir.h** Page 1/1

```
1   //
2   //  common_convertir.h
3   //  LIBRERIA CONVERTIR
4   //
5   //  Librería de funciones conversoras.
6   //
7
8
9   #ifndef CONVERTIR_H
10  #define CONVERTIR_H
11
12
13  #include <string.h>
14  #include <stdint.h>
15  #include <ctype.h>
16
17
18  /* ******************************************************************************
19   * DECLARACIÓN DE LA CLASE
20   * ******************************************************************************/
21
22
23  class Convertir {
24  public:
25
26    // Devuelve el equivalente entero de un caracter hexadecimal
27    static int htoi(char a);
28
29    // Convierte un unsigned int a un string de contenido hexadecimal
30    static std::string uitoh(uint8_t *a, size_t size);
31
32    // Convierte un string de contenido hexadecimal a un unsigned int
33    static uint8_t* htoui(const std::string& s);
34
35    // Convierte un string en un integer
36    static int stoi(const std::string& s);
37
38    // Convierte un string en un unsigned integer
39    static unsigned int stoui(const std::string& s);
40
41    // Convierte un integer en un string
42    static std::string itos(const int i);
43
44    // Convierte un unsigned integer en un string
45    static std::string uitos(const unsigned int i);
46
47    // Convierte un string en un string en minusculas
48    static std::string toLowercase(const std::string &s);
49  };
50
51
52  #endif
```

```
1   //
2   //  common_convertir.h
3   //  LIBRERIA CONVERTIR
4   //
5   //  Librería de funciones conversoras.
6   //
7
8
9   #include <iomanip>
10  #include <sstream>
11  #include "common_convertir.h"
12
13
14
15
16  /* ************************************************************************
17   * DEFINICIÓN DE LA CLASE
18   * ************************************************************************/
19
20
21  // Devuelve el equivalente entero de un caracter hexadecimal
22  int Convertir::htoi(char a) {
23    if(a > 'F') return -1;
24    else if (a < 'A') return (a - '0');
25    return (a - 'A' + 10);
26  }
27
28
29  // Convierte un unsigned int a un string de contenido hexadecimal
30  std::string Convertir::uitoh(uint8_t *a, size_t size) {
31    std::stringstream stream;
32
33    for(unsigned int i = 0; i < size; i++) {
34      stream << std::uppercase << std::setfill('0') << std::setw(2) <<
35        std::hex << int(a[i]);
36    }
37
38    return stream.str();
39  }
40
41
42  // Convierte un string de contenido hexadecimal a un unsigned int
43  uint8_t* Convertir::htoui(const std::string& s) {
44    uint8_t *a = new uint8_t[s.size() / 2];
45    int j = 0;
46
47    for(unsigned int i = 0; i < s.size(); i += 2) {
48      uint8_t pri = Convertir::htoi(s[i]);
49      uint8_t seg = Convertir::htoi(s[i+1]);
50
51      a[j] = pri * 16 + seg;
52      j++;
53    }
54
55    return a;
56  }
57
58
59  // Convierte un string en un integer
60  int Convertir::stoi(const std::string& s) {
61    int i;
62    std::stringstream ss(s);
63    ss >> i;
64    return i;
65  }
66
```



```
67
68  // Convierte un string en un unsigned integer
69  unsigned int Convertir::stoui(const std::string& s) {
70    unsigned int i;
71    std::stringstream ss(s);
72    ss >> i;
73    return i;
74  }
75
76
77  // Convierte un integer en un string
78  std::string Convertir::itos(const int i) {
79      std::ostringstream s;
80      s << i;
81      return s.str();
82  }
83
84
85  // Convierte un unsigned integer en un string
86  std::string Convertir::uitos(const unsigned int i) {
87    std::ostringstream s;
88      s << i;
89      return s.str();
90  }
91
92
93  // Convierte un string en un string en minusculas
94  std::string Convertir::toLowercase(const std::string &s) {
95    std::string d;
96    int i;
97    for (i = 0; i < (int)s.length(); i++) {
98      if (isalpha(s[i]))
99        d += (char)tolower(s[i]);
100     else
101       d += s[i];
102   }
103   return d;
104 }
105
```

jun 25, 13 13:44 **common_comunicador.h** Page 1/1

```
1  //
2  //  common_comunicador.h
3  //  CLASE COMUNICADOR
4  //
5  //  Clase que implementa la interfaz de comunicación entre servidor y clientes.
6  //
7
8
9  #ifndef COMUNICADOR_H
10 #define COMUNICADOR_H
11
12
13
14 #include <string>
15 #include "common_socket.h"
16 #include "common_protocolo.h"
17
18
19
20
21 /* ******************************************************************************
22  * DECLARACIÓN DE LA CLASE
23  * ******************************************************************************/
24
25
26 class Comunicador {
27 private:
28
29   Socket *socket;       // Socket de comunicación
30
31 public:
32
33   // Constructor
34   // PRE: 'socket' es un socket por el que se desea hacer el envío y
35   // transmisión de mensajes
36   explicit Comunicador(Socket *socket);
37
38   // Emite una instrucción.
39   // PRE: 'instruccion' es una cadena que identifica la instrucción a emitir;
40   // 'args' son los argumentos de dicha instrucción separadas entre si por
41   // un espacio.
42   // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
43   // de error.
44   int emitir(const std::string& instruccion, const std::string& args);
45
46   // Emite un mensaje.
47   // PRE: 'msg' es el mensaje que se desea enviar.
48   // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
49   // de error.
50   int emitir(const std::string& msg);
51
52   // Recibe una instrucción.
53   // POST: se almacenó la instrucción recibida en 'instruccion' y los
54   // argumentos en args, los cuales se encuentran separados entre si por un
55   // espacio. De producirse un error, 'instruccion' y 'args' queda vacíos y
56   // se retorna -1. En caso de exito se devuelve 0.
57   int recibir(std::string& instruccion, std::string& args);
58
59   // Recibe un mensaje
60   // POST: se almacenó el mensaje recibido en 'msg'. De producirse un error,
61   // 'msg' quedará vacia y se retorna -1. En caso de éxito se devuelve 0.
62   int recibir(std::string& msg);
63 };
64
65 #endif
```

jun 25, 13 13:44 **common_comunicador.cpp** Page 1/2

```
1  //
2  //  common_comunicador.h
3  //  CLASE COMUNICADOR
4  //
5  //  Clase que implementa la interfaz de comunicación entre servidor y clientes.
6  //
7
8
9  #include "common_comunicador.h"
10 #include <sstream>
11
12
13
14
15 /* ******************************************************************************
16  * DEFINICIÓN DE LA CLASE
17  * ******************************************************************************/
18
19
20 // Constructor
21 // PRE: 'socket' es un socket por el que se desea hacer el envío y
22 // transmisión de mensajes
23 Comunicador::Comunicador(Socket *socket) : socket(socket) { }
24
25
26 // Emite una instrucción y sus argumentos.
27 // PRE: 'instruccion' es una cadena que identifica la instrucción a emitir;
28 // 'args' son los argumentos de dicha instrucción separadas entre si por
29 // un espacio.
30 // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
31 // de error.
32 int Comunicador::emitir(const std::string& instruccion,
33   const std::string& args) {
34   // Armamos mensaje a enviar
35   std::string msg = instruccion + " " + args + FIN_MENSAJE;
36
37   // Enviamos el mensaje
38   return this->socket->enviar(msg.c_str(), msg.size());
39 }
40
41
42 // Emite un mensaje.
43 // PRE: 'msg' es el mensaje que se desea enviar.
44 // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
45 // de error.
46 int Comunicador::emitir(const std::string& msg) {
47   // Armamos mensaje a enviar
48   std::string msg_n = msg + FIN_MENSAJE;
49
50   // Enviamos el mensaje
51   return this->socket->enviar(msg_n.c_str(), msg_n.size());
52 }
53
54
55 // Recibe una instrucción.
56 // POST: se almacenó la instrucción recibida en 'instruccion' y los
57 // argumentos en args, los cuales se encuentran separados entre si por un
58 // espacio. De producirse un error, 'instruccion' y 'args' queda vacíos y
59 // se retorna -1. En caso de exito se devuelve 0.
60 int Comunicador::recibir(std::string& instruccion, std::string& args) {
61   // Variable auxiliar para armar mensaje
62   std::stringstream msg_in;
63   // Limpiamos argumentos que recibiran datos
64   instruccion = "";
65   args = "";
66
```

jun 25, 13 13:44 **common_comunicador.cpp** Page 2/2

```
67    // Recibimos de a 1 Byte hasta recibir el caractér de fin de mensaje
68    while(true) {
69      // Definimos buffer de 1 Byte
70      char bufout;
71
72      // Si se produce un error, devolvemos una instrucción vacía
73      if(this→socket→recibir(&bufout, 1) ≤ 0) return −1;
74
75      // Si se recibió el caractér de fin de mensaje, salimos
76      if(bufout ≡ FIN_MENSAJE) break;
77
78      // Agregamos el caractér a los datos ya recibidos
79      msg_in << bufout;
80    }
81
82    // Paresamos instrucción y argumentos
83    msg_in >> instruccion;
84    getline(msg_in, args);
85
86    // Eliminamos el espacio inicial sobrante de los argumentos
87    if(args ≠ "") args.erase(0, 1);
88
89    return 0;
90  }
91
92
93  // Recibe un mensaje
94  // POST: se almacenó el mensaje recibido en 'msg'. De producirse un error,
95  // 'msg' quedará vacia y se retorna −1. En caso de éxito se devuelve 0.
96  int Comunicador::recibir(std::string& msg) {
97    // Variable auxiliar para armar mensaje
98    std::stringstream msg_in;
99    // Limpiamos argumentos que recibiran datos
100   msg = "";
101
102   // Recibimos de a 1 Byte hasta recibir el caractér de fin de mensaje
103   while(true) {
104     // Definimos buffer de 1 Byte
105     char bufout;
106
107     // Si se produce un error, devolvemos una instrucción vacía
108     if(this→socket→recibir(&bufout, 1) ≤ 0) return −1;
109
110     // Si se recibió el caractér de fin de mensaje, salimos
111     if(bufout ≡ FIN_MENSAJE) break;
112
113     // Agregamos el caractér a los datos ya recibidos
114     msg_in << bufout;
115   }
116
117   // Copiamos mensaje en variable de salida
118   msg = msg_in.str();
119
120   return 0;
121 }
```

jun 25, 13 13:44 **common_cola.h** Page 1/1

```
1   #ifndef COMMON_COLA_H_
2   #define COMMON_COLA_H_
3
4   #include "common_mutex.h"
5   #include "common_lock.h"
6   #include <queue>
7
8   template <class T>
9   class Cola {
10  private:
11    std::queue<T>* cola;
12    Mutex* mutex;
13  public:
14    Cola() {
15      this→cola = new std::queue<T>;
16      this→mutex = new Mutex;
17    };
18
19    virtual ~Cola() {
20      if (this→cola) {
21        delete(this→cola);
22        this→cola = NULL;
23      }
24      if (this→mutex) {
25        delete(this→mutex);
26        this→mutex = NULL;
27      }
28    };
29
30    void push(const T &dato) {
31      Lock lock(*this→mutex);
32      cola→push(dato);
33      lock.signal();
34    };
35
36    void pop() {
37      Lock lock(*this→mutex);
38      cola→pop();
39    };
40
41    T& front() {
42      Lock lock(*this→mutex);
43      return (cola→front());
44    };
45
46    int tamanio() {
47      Lock lock(*this→mutex);
48      return (cola→size());
49    };
50
51    T pop_bloqueante() {
52      Lock lock(*this→mutex);
53      if (cola→size() ≡ 0)
54        lock.wait();
55      T t = cola→front();
56      cola→pop();
57      return t;
58    };
59
60    bool vacia() {
61      Lock lock(*this→mutex);
62      return(cola→empty());
63    }
64  };
65
66  #endif /* COMMON_COLA_H_ */
```

jun 25, 13 13:44 **common_archivoTexto.h** Page 1/1

```
1  //
2  //  common_archivoTexto.h
3  //  CLASE ARCHIVOTEXTO
4  //
5
6
7  #ifndef __ARCHIVOTEXTO_H__
8  #define __ARCHIVOTEXTO_H__
9
10 #ifndef MAX_LENGTH
11 #define MAX_LENGTH 25
12 #endif
13
14 #include <iostream>
15 #include <fstream>
16 #include <stdio.h>
17
18 using namespace std;
19
20 class ArchivoTexto {
21 private:
22   fstream  archivo;  // referencia al archivo
23
24 public:
25   ArchivoTexto(const std::string& path,int flag);
26   ~ArchivoTexto();
27   void escribir(const std::string& cadena);
28   bool leerLinea(std::string &cadena, char separador,string buscado);
29   void leerLinea(string& cadena, int pos);
30   bool validarLinea(std:: string &cadena, string buscado);
31 };
32
33 #endif
```

jun 25, 13 13:44 **common_archivoTexto.cpp** Page 1/1

```
1
2
3  #include "common_archivoTexto.h"
4
5  using namespace std;
6
7  ArchivoTexto::ArchivoTexto(const std::string& path, int flag) {
8    if (flag ≡ 0) {
9      archivo.open(path.c_str(), fstream::in);
10     if (¬archivo.is_open())
11       throw ios_base::failure("No fue posible acceder al archivo de configuracion");
12   }
13   if (flag ≡ 1) {
14     archivo.open(path.c_str(), fstream::out | fstream::trunc);
15     if (¬archivo.is_open())
16     throw ios_base::failure("No fue posible acceder al archivo de configuracion");
17   }
18 }
19
20
21
22 ArchivoTexto::~ArchivoTexto() {
23   archivo.close();
24 }
25
26
27 void ArchivoTexto::escribir(const string& cadena) {
28   archivo << cadena;
29 }
30
31 void ArchivoTexto::leerLinea(string& cadena, int pos) {
32   char linea[MAX_LENGTH];
33   archivo.seekg(pos, archivo.beg);
34   archivo.getline((char*)&linea, MAX_LENGTH,'\n');
35   cadena = linea;
36
37 }
38
39 bool ArchivoTexto::validarLinea(string &cadena, string buscado) {
40   int i = 0;
41   int j = (int) buscado.size();
42   for (i = 0; i < j ; i++) {
43     if ((cadena[i]) ≠ (buscado[i])) return false;
44   }
45   return true;
46 }
47
48 bool ArchivoTexto::leerLinea(string &cadena, char separador,string buscado) {
49   char linea[MAX_LENGTH];
50    //  lee del archivo a la linea, hasta haber leido:
51   //  MAX_LENGTH caracteres, o un fin de linea
52     archivo.getline((char*)&linea , MAX_LENGTH, separador);
53     cadena = linea; //cadena tiene lo leido.
54
55     if (validarLinea(cadena,buscado)) return true;
56
57     return false;
58 }
59
60
```

```
1  //
2  //  client_sincronizador.h
3  //  CLASE SINCRONIZADOR
4  //
5
6
7  #ifndef SINCRONIZADOR_H
8  #define SINCRONIZADOR_H
9
10
11 #include <string>
12 #include "common_mutex.h"
13 #include "common_lock.h"
14 #include "common_lista.h"
15 #include "common_logger.h"
16 #include "client_emisor.h"
17
18
19
20
21
22 /* ******************************************************************************
23  * DECLARACIÓN DE LA CLASE
24  * ******************************************************************************/
25
26
27 class Sincronizador {
28 private:
29
30   Mutex m;                  // Mutex
31   Emisor *emisor;               // Emisor de mensajes
32   Logger *logger;               // Logger de eventos
33
34 public:
35
36   // Constructor
37   Sincronizador(Emisor *emisor, Logger *logger);
38
39   // Destructor
40   ~Sincronizador();
41
42   // Crea el evento de envío de un archivo nuevo
43   void enviarArchivo(std::string& nombreArchivo, std::string& contenido,
44     std::string hash);
45
46   // Crea el evento de modificar un archivo existente.
47   // PRE: 'nombreArchivo' es el nombre de archivo que debe modificarse;
48   // 'bloques' son pares de (bloque, contenido) los cuales son enviados para
49   // ser actualizados en el servidor.
50   void modificarArchivo(std::string& nombreArchivo, unsigned int bytesTotal,
51     Lista< std::pair< int, std::string > > bloques);
52
53   // Crea el evento de eliminación de un archivo.
54   // PRE: 'nombreArchivo' es el nombre de archivo que debe eliminarse.
55   void eliminarArchivo(std::string& nombreArchivo);
56
57   // Crea el evento de solicitud de un archivo nuevo.
58   // PRE: 'nombreArchivo' esl el nombre del archivo.
59   void solicitarArchivoNuevo(std::string& nombreArchivo);
60
61   // Crea el evento de solicitud de modificación de un archivo.
62   // PRE: 'nombreArchivo' esl el nombre del archivo; 'bloquesASolicitar' es
63   // una lista de números de bloque a solicitar.
64   void solicitarBloquesModificados(std::string& nombreArchivo,
65     Lista< int > bloquesASolicitar);
66 };
```



```
67
68 #endif
```

```
1  //
2  //  client_sincronizador.cpp
3  //  CLASE SINCRONIZADOR
4  //
5
6
7  #include <string>
8  #include "common_protocolo.h"
9  #include "common_convertir.h"
10 #include "client_sincronizador.h"
11
12
13
14
15 /* ************************************************************************
16  * DEFINICIÓN DE LA CLASE
17  * ************************************************************************/
18
19
20 // Constructor
21 Sincronizador::Sincronizador(Emisor *emisor, Logger *logger) : emisor(emisor),
22   logger(logger) { }
23
24
25 // Destructor
26 Sincronizador::~Sincronizador() { }
27
28
29 // Crea el evento de envío de un archivo nuevo
30 void Sincronizador::enviarArchivo(std::string& nombreArchivo, std::string& conte
   nido, std::string hash) {
31   // Bloqueamos el mutex
32   Lock l(m);
33
34   // Armamos mensaje
35   std::string mensaje;
36   mensaje.append(COMMON_SEND_FILE);
37   mensaje.append(" ");
38   mensaje.append(nombreArchivo);
39   mensaje.append(COMMON_DELIMITER);
40   mensaje.append(contenido);
41   mensaje.append(COMMON_DELIMITER);
42   mensaje.append(hash);
43
44
45   // Mensaje de log
46   this→logger→emitirLog("Se envió archivo '" + nombreArchivo + "' " +
47     "al servidor.");
48
49   // Enviamos mensaje al emisor
50   this→emisor→ingresarMensajeDeSalida(mensaje);
51 }
52
53
54 // Crea el evento de modificar un archivo existente.
55 // PRE: 'nombreArchivo' es el nombre de archivo que debe modificarse;
56 // 'bloques' son pares de (bloque, contenido) los cuales son enviados para
57 // ser actualizados en el servidor.
58 void Sincronizador::modificarArchivo(std::string& nombreArchivo,
59   unsigned int bytesTotal, Lista< std::pair< int, std::string > > bloques) {
60   // Bloqueamos el mutex
61   Lock l(m);
62
63   // Armamos mensaje
64   std::string mensaje;
65   mensaje.append(C_MODIFY_FILE);
```



```
66   mensaje.append(" ");
67   mensaje.append(nombreArchivo);
68   mensaje.append(COMMON_DELIMITER);
69   mensaje.append(Convertir::uitos(bytesTotal));
70
71   // Iteramos sobre los bloques modificados. Si solo se cambio el tamaño
72   // del archivo por el truncamiento de parte de su contenido final, solo
73   // se envía la cantidad tota de bytes.
74   while(¬bloques.estaVacia()) {
75     std::pair< int, std::string > bloque = bloques.verPrimero();
76     bloques.eliminarPrimero();
77
78     mensaje.append(COMMON_DELIMITER);
79     mensaje.append(Convertir::uitos(bloque.first));
80     mensaje.append(COMMON_DELIMITER);
81     mensaje.append(bloque.second);
82   }
83
84   // Mensaje de log
85   std::string log = "Se enviaron modificaciones hechas sobre el archivo '";
86   log += nombreArchivo + "' al servidor.";
87   this→logger→emitirLog(log);
88
89   // Enviamos mensaje al emisor
90   this→emisor→ingresarMensajeDeSalida(mensaje);
91 }
92
93
94 // Crea el evento de eliminación de un archivo.
95 // PRE: 'nombreArchivo' es el nombre de archivo que debe eliminarse.
96 void Sincronizador::eliminarArchivo(std::string& nombreArchivo) {
97   // Bloqueamos el mutex
98   Lock l(m);
99
100  // Armamos mensaje
101  std::string mensaje;
102  mensaje.append(COMMON_DELETE_FILE);
103  mensaje.append(" ");
104  mensaje.append(nombreArchivo);
105
106  // Mensaje de log
107  this→logger→emitirLog("Se envió notificación de eliminar archivo '" +
108    nombreArchivo + "' al servidor");
109
110  // Enviamos mensaje al emisor
111  this→emisor→ingresarMensajeDeSalida(mensaje);
112 }
113
114
115 // Crea el evento de solicitud de un archivo nuevo.
116 // PRE: 'nombreArchivo' esl el nombre del archivo.
117 void Sincronizador::solicitarArchivoNuevo(std::string& nombreArchivo) {
118   // Bloqueamos el mutex
119   Lock l(m);
120
121   // Armamos mensaje
122   std::string mensaje;
123   mensaje.append(C_FILE_REQUEST);
124   mensaje.append(" ");
125   mensaje.append(nombreArchivo);
126
127   // Mensaje de log
128   this→logger→emitirLog("Se solicitó archivo '" + nombreArchivo + "' " +
129     "al servidor.");
130
131   // Enviamos mensaje al emisor
```

jun 25, 13 13:44 **client_sincronizador.cpp** Page 3/3

```
132   this→emisor→ingresarMensajeDeSalida(mensaje);
133 }
134
135
136 // Crea el evento de solicitud de modificación de un archivo.
137 // PRE: 'nombreArchivo' esl el nombre del archivo; 'bloquesASolicitar' es
138 // una lista de números de bloque a solicitar.
139 void Sincronizador::solicitarBloquesModificados(std::string& nombreArchivo,
140   Lista< int > bloquesASolicitar) {
141   // Bloqueamos el mutex
142   Lock l(m);
143
144   // Armamos mensaje
145   std::string mensaje;
146   mensaje.append(C_FILE_PARTS_REQUEST);
147   mensaje.append(" ");
148   mensaje.append(nombreArchivo);
149
150   // Insertamos numeros de bloque en mensaje
151   for(size_t i = 0; i < bloquesASolicitar.tamanio(); i++) {
152     mensaje.append(COMMON_DELIMITER);
153     mensaje.append(Convertir::itos(bloquesASolicitar[i]));
154   }
155
156   // Mensaje de log
157   this→logger→emitirLog("Se solicitaron partes del archivo '" +
158     nombreArchivo + "' al servidor");
159
160   // Enviamos mensaje al emisor
161   this→emisor→ingresarMensajeDeSalida(mensaje);
162 }
```

jun 25, 13 13:44 **client_receptor.h** Page 1/1

```
1  //
2  //  client_receptor.h
3  //  CLASE RECEPTOR
4  //
5
6
7  #ifndef RECEPTOR_H
8  #define RECEPTOR_H
9
10 #include <string>
11 #include "common_cola.h"
12 #include "common_thread.h"
13 #include "common_socket.h"
14 #include "common_comunicador.h"
15 #include "common_logger.h"
16 #include "common_seguridad.h"
17
18
19
20 /* ******************************************************************************
21  * DECLARACIÓN DE LA CLASE
22  * ******************************************************************************/
23
24
25 class Receptor : public Thread {
26 private:
27
28   Socket *socket;             // Socket por el que recibe datos
29   Cola< std::string > entrada;      // Cola de entrada
30   Comunicador com;            // Comunicador del receptor
31   Logger *logger;             // Logger de eventos
32   std::string clave;            // Clave utilizada para firmar7
33                         // mensajes
34   bool activa;              // Flag de recepción activa
35
36 public:
37
38   // Constructor
39   Receptor(Socket *socket, Logger *logger, const std::string &clave);
40
41   // Destructor
42   ~Receptor();
43
44   // Inicia la recepción
45   void iniciar();
46
47   // Detiene la recepción
48   void detener();
49
50   // Permite obtener un mensaje recibido.
51   // POST: Devuelve el primer mensaje de la cola de mensajes entrantes.
52   std::string obtenerMensajeDeEntrada();
53
54   // Define tareas a ejecutar en el hilo.
55   // Se encarga de recibir y guardar en la cola de entrada.
56   virtual void run();
57
58   // Comprueba si la recepción se encuentra activa. Se encontrará activa
59   // mientras el socket permanezca activo, lo cual se considera desde que se
60   // inicia el objeto con el metodo iniciar(). En caso de cerrarse el socket
61   // se devolverá false, mientras que al estar activa la recepción se
62   // retornará true.
63   bool recepcionActiva();
64 };
65
66 #endif
```

jun 25, 13 13:44 **client_receptor_de_archivos.h** Page 1/1

```
//
//  client_receptor_de_archivos.h
//  CLASE RECEPTORDEARCHIVOS
//


#ifndef RECEPTOR_DE_ARCHIVOS_H
#define RECEPTOR_DE_ARCHIVOS_H


#include <string>
#include "common_manejador_de_archivos.h"
#include "common_protocolo.h"
#include "common_logger.h"
class Mutex;
class Lock;




/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


class ReceptorDeArchivos {
private:

  Mutex m;                    // Mutex
  ManejadorDeArchivos *manejadorDeArchivos;   // Manejador de archivos
  Logger *logger;                // Logger de eventos

public:

  // Constructor
  ReceptorDeArchivos(ManejadorDeArchivos *unManejador, Logger *logger);

  // Destructor
  ~ReceptorDeArchivos();

  // Se encarga de procesar la recepción de un archivo nuevo.
  // PRE: 'nombreArchivo' es el nombre del archivo a recibir; 'contenido' es
  // el contenido del archivo a recibir.
  void recibirArchivo(const std::string& nombreArchivo,
    const std::string& contenido);

  // Se encarga de procesar la eliminación de un archivo
  void eliminarArchivo(std::string& nombreArchivo);

  // Se encarga de procesar la recepción de modificaciones en archivo.
  // PRE: 'nombreArchivo' es el nombre del archivo a modificar;
  // 'cantloquesDelArchivo' es la cantidad nueva de bloques que debe
  // contener el archivo; 'listaBloquesAReemplazar' es una lista que
  // contiene los números de bloque y su respectivo contenido, los
  // cuales reemplazarán a los bloques actuales.
  void recibirModificaciones(std::string& nombreArchivo,
    unsigned int cantBytesDelArchivo,
    Lista< std::pair< int, std::string > >& listaBloquesAReemplazar);
};

#endif
```

jun 25, 13 13:44 **client_receptor_de_archivos.cpp** Page 1/2

```
//
//  client_receptor_de_archivos.cpp
//  CLASE RECEPTORDEARCHIVOS
//



#include "common_mutex.h"
#include "common_lock.h"
#include "client_receptor_de_archivos.h"




/* ************************************************************************
 * DEFINICIÓN DE LA CLASE
 * ************************************************************************/


// Constructor
ReceptorDeArchivos::ReceptorDeArchivos(ManejadorDeArchivos *unManejador,
  Logger *logger) : manejadorDeArchivos(unManejador), logger(logger) { }


// Destructor
ReceptorDeArchivos::~ReceptorDeArchivos() { }


// Se encarga de procesar la recepción de un archivo nuevo.
// PRE: 'nombreArchivo' es el nombre del archivo a recibir; 'contenido' es
// el contenido del archivo a recibir.
void ReceptorDeArchivos::recibirArchivo(const std::string& nombreArchivo,
  const std::string& contenido) {
  // Damos la orden de agregar el archivo
  this→manejadorDeArchivos→agregarArchivo(nombreArchivo, contenido);

  // Mensaje de log
  this→logger→emitirLog("Se agregó archivo nuevo '" + nombreArchivo + "'");

  // Actualizamos el registro local de archivos
  Lista< std::string > nuevos, mod;
  nuevos.insertarUltimo(nombreArchivo);

  this→manejadorDeArchivos→actualizarRegistroDeArchivos(nuevos, mod);

  // Mensaje de log
  this→logger→emitirLog("Se actualizó el registro de archivos.");
}


// Se encarga de procesar la eliminación de un archivo
void ReceptorDeArchivos::eliminarArchivo(std::string& nombreArchivo) {
  // Damos la orden de eliminar el archivo
  this→manejadorDeArchivos→eliminarArchivo(nombreArchivo);

  // Mensaje de log
  this→logger→emitirLog("Se eliminó el archivo '" + nombreArchivo + "'.");

  // Eliminamos el archivo del registro local
  this→manejadorDeArchivos→borrarDeRegistroDeArchivos(nombreArchivo);

  // Mensaje de log
  this→logger→emitirLog("Se actualizó el registro de archivos.");
}
```

```
67  // Se encarga de procesar la recepción de modificaciones en archivo.
68  // PRE: 'nombreArchivo' es el nombre del archivo a modificar;
69  // 'cantloquesDelArchivo' es la cantidad nueva de bloques que debe
70  // contener el archivo; 'listaBloquesAReemplazar' es una lista que
71  // contiene los números de bloque y su respectivo contenido, los
72  // cuales reemplazarán a los bloques actuales.
73  void ReceptorDeArchivos::recibirModificaciones(std::string& nombreArchivo,
74    unsigned int cantBytesDelArchivo,
75    Lista< std::pair< int, std::string > >& listaBloquesAReemplazar) {
76    // Damos la orden de modificar el archivo
77    this→manejadorDeArchivos→modificarArchivo(nombreArchivo,
78      cantBytesDelArchivo, listaBloquesAReemplazar);
79
80    // Mensaje de log
81    this→logger→emitirLog("Se modificó el archivo '" + nombreArchivo + "'.");
82
83    // Actualizamos el registro local de archivos
84    Lista< std::string > nuevos, mod;
85
86    this→manejadorDeArchivos→actualizarRegistroDeArchivos(nuevos, mod);
87
88    // Mensaje de log
89    this→logger→emitirLog("Se actualizó el registro de archivos.");
90  }
```



```
1   //
2   //  client_receptor.h
3   //  CLASE RECEPTOR
4   //
5
6
7   #include "client_receptor.h"
8
9
10  namespace {
11    const std::string COLA_SALIDA_FIN = "COLA-SALIDA-FIN";
12  }
13
14
15
16
17  /* ************************************************************************
18   * DEFINICIÓN DE LA CLASE
19   * ************************************************************************/
20
21
22  // Constructor
23  Receptor::Receptor(Socket *socket, Logger *logger, const std::string &clave) :
24    socket(socket), com(socket), logger(logger), clave(clave),
25    activa(false) { }
26
27
28  // Destructor
29  Receptor::~Receptor() { }
30
31
32  // Inicia la recepción
33  void Receptor::iniciar() {
34    // Iniciamos el hilo
35    this→start();
36
37    // Habilitamos flag de recepción
38    this→activa = true;
39  }
40
41
42  // Detiene la recepción
43  void Receptor::detener() {
44    // Detenemos hilo
45    this→stop();
46
47    // Destrabamos la cola encolando un mensaje de finalización detectable
48    this→entrada.push(COLA_SALIDA_FIN);
49
50    // Cerramos el socket
51    this→socket→cerrar();
52  }
53
54
55  // Permite obtener un mensaje recibido.
56  // POST: Devuelve el primer mensaje de la cola de mensajes entrantes.
57  std::string Receptor::obtenerMensajeDeEntrada() {
58    return this→entrada.pop_bloqueante();
59  }
60
61
62  // Define tareas a ejecutar en el hilo.
63  // Se encarga de emitir lo que se encuentre en la cola de salida.
64  void Receptor::run() {
65    // Variables auxiliares
66    std::string firma;
```

jun 25, 13 13:44 **client_receptor.cpp** Page 2/2

```
67    int delim;
68
69    // Nos ponemos a la espera de mensajes de entrada
70    while(this→isActive()) {
71      // Esperamos recepción de mensaje
72      std::string mensaje;
73      if(this→com.recibir(mensaje) ≡ −1) {
74        // Mensaje de log
75        this→logger→emitirLog("RECEPTOR: Se desconectó el servidor.");
76
77        // Deshabilitamos flag de recepción
78        this→activa = false;
79
80        break;
81      }
82      // Se separa la firma del mensaje
83      delim = mensaje.find(COMMON_DELIMITER);
84      firma = mensaje.substr(0, delim);
85      mensaje = mensaje.substr(delim + 1);
86
87      // Se comprueba la validez de la firma
88      if (Seguridad::firmaValida(mensaje, this→clave, firma)) {
89        // Encolamos el mensaje en cola de entrada
90        this→entrada.push(mensaje);
91      }
92      else {
93        // Enviar mensaje de desconexion
94
95        // Detener ejecucion
96        this→detener();
97      }
98    }
99  }
100
101
102 // Comprueba si la recepción se encuentra activa. Se encontrará activa
103 // mientras el socket permanezca activo, lo cual se considera desde que se
104 // inicia el objeto con el metodo iniciar(). En caso de cerrarse el socket
105 // se devolverá false, mientras que al estar activa la recepción se
106 // retornará true.
107 bool Receptor::recepcionActiva() {
108   return this→activa;
109 }
```

jun 25, 13 13:44 **client_manejador_de_notificaciones.h** Page 1/1

```
1   //
2   //  client_manejador_de_notificaciones.h
3   //  CLASE MANEJADORDENOTIFICACIONES
4   //
5
6
7   #ifndef MANEJADOR_DE_NOTIFICACIONES_H
8   #define MANEJADOR_DE_NOTIFICACIONES_H
9
10
11  #include <string>
12  #include "common_thread.h"
13  #include "common_logger.h"
14  #include "client_receptor.h"
15  #include "client_inspector.h"
16  #include "client_receptor_de_archivos.h"
17
18
19
20
21
22
23  /* ************************************************************************
24   * DECLARACIÓN DE LA CLASE
25   * ************************************************************************/
26
27
28  class ManejadorDeNotificaciones : public Thread {
29  private:
30
31    Receptor *receptor;              // Receptor de mensajes
32    Inspector *inspector;            // Inspector
33    ReceptorDeArchivos *receptorDeArchivos;     // Receptor de archivos
34    Logger *logger;                 // Logger de eventos
35
36  public:
37
38    // Constructor
39    ManejadorDeNotificaciones(Receptor *receptor, Inspector *inspector,
40      ReceptorDeArchivos *receptorDeArchivos, Logger *logger);
41
42    // Destructor
43    ~ManejadorDeNotificaciones();
44
45    // Define tareas a ejecutar en el hilo.
46    // Routea las notifiaciones y mensajes de entrada
47    virtual void run();
48  };
49
50  #endif
```

jun 25, 13 13:44 **client_manejador_de_notificaciones.cpp** Page 1/3

```
1  //
2  //  client_manejador_de_notificaciones.cpp
3  //  CLASE MANEJADORDENOTIFICACIONES
4  //
5
6
7  #include <sstream>
8  #include "common_parser.h"
9  #include "common_lista.h"
10 #include "common_protocolo.h"
11 #include "client_manejador_de_notificaciones.h"
12
13
14
15
16
17
18 /* ************************************************************************
19  * DEFINICIÓN DE LA CLASE
20  * ************************************************************************/
21
22
23 // Constructor
24 ManejadorDeNotificaciones::ManejadorDeNotificaciones(Receptor *receptor,
25   Inspector *inspector, ReceptorDeArchivos *receptorDeArchivos,
26   Logger *logger) : receptor(receptor), inspector(inspector),
27   receptorDeArchivos(receptorDeArchivos), logger(logger) { }
28
29
30 // Destructor
31 ManejadorDeNotificaciones::~ManejadorDeNotificaciones() { }
32
33
34 // Define tareas a ejecutar en el hilo.
35 // Routea las notifiaciones y mensajes de entrada
36 void ManejadorDeNotificaciones::run() {
37   // Procesamos mensajes entrantes
38   while (this→isActive()) {
39     // Pide un mensaje de la cola al receptor
40     std::string mensaje;
41     mensaje = this→receptor→obtenerMensajeDeEntrada();
42
43     if(¬this→isActive()) break;
44
45     // Tomamos instrucción y sus argumentos
46     std::string instruccion, args;
47     Parser::parserInstruccion(mensaje, instruccion, args);
48
49
50     // Caso en que se notifica la existencia de un nuevo archivo
51     if(instruccion ≡ S_NEW_FILE) {
52       // Parseamos argumentos
53       Lista< std::string > listaArgumentos;
54       Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
55
56       // Tomamos nombre de archivo
57       std::string nombreArchivo = listaArgumentos.verPrimero();
58       listaArgumentos.eliminarPrimero();
59
60       // Tomamos nombre de archivo
61       std::string hash = listaArgumentos.verPrimero();
62       listaArgumentos.eliminarPrimero();
63
64       // Mensaje de log
65       this→logger→emitirLog("NOTIFIACIÓN: Nuevo archivo '" +
66         nombreArchivo + "'.");
```

jun 25, 13 13:44 **client_manejador_de_notificaciones.cpp** Page 2/3

```
67
68       // Derivamos al inspector
69       this→inspector→inspeccionarExisteArchivo(nombreArchivo, hash);
70     }
71     else if(instruccion ≡ S_FILE_CHANGED) {
72       // Parseamos argumentos
73       Lista< std::string > listaArgumentos;
74       Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
75
76       // Tomamos nombre de archivo
77       std::string nombreArchivo = listaArgumentos.verPrimero();
78       listaArgumentos.eliminarPrimero();
79
80       // Tomamos cantidad de bytes que debe tener el archivo ahora
81       std::string sCantBytesTotal = listaArgumentos.verPrimero();
82       unsigned int cantBytesTotal = Convertir::stoui(sCantBytesTotal);
83       listaArgumentos.eliminarPrimero();
84
85       // Mensaje de log
86       this→logger→emitirLog("NOTIFICACIÓN: Archivo '" + nombreArchivo
87         + "' ha sido modificado.");
88
89       // Lista de bloques a inspeccionar
90       Lista< std::pair< int, std::string > > bloques;
91
92       // Tomamos los bloques y sus hashes
93       while(¬listaArgumentos.estaVacia()) {
94         int bloque = Convertir::stoui(listaArgumentos.verPrimero());
95         listaArgumentos.eliminarPrimero();
96         std::string hash = listaArgumentos.verPrimero();
97         listaArgumentos.eliminarPrimero();
98
99         bloques.insertarUltimo(std::make_pair(bloque, hash));
100      }
101
102      // Derivamos al inspector
103      this→inspector→inspeccionarArchivo(nombreArchivo,
104        cantBytesTotal, bloques);
105    }
106    else if(instruccion ≡ COMMON_SEND_FILE) {
107      // Parseamos argumentos para obtener nombre y contenido del archivo
108      Lista< std::string > listaArgumentos;
109      Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
110
111      // Mensaje de log
112      this→logger→emitirLog("NOTIFICACIÓN: Arribo de archivo '" +
113        listaArgumentos[0] + "'.");
114
115      // Derivamos al receptor de archivos
116      this→receptorDeArchivos→recibirArchivo(listaArgumentos[0],
117        listaArgumentos[1]);
118    }
119    else if(instruccion ≡ COMMON_DELETE_FILE) {
120      // Mensaje de log
121      this→logger→emitirLog("NOTIFICACIÓN: Archivo '" + args
122        + "' ha sido eliminado.");
123
124      // Derivamos al receptor de archivos
125      this→receptorDeArchivos→eliminarArchivo(args);
126    }
127    else if(instruccion ≡ COMMON_FILE_PARTS) {
128      // Parseamos los argumentos de la respuesta
129      Lista< std::string > listaArgumentos;
130      Parser::dividirCadena(args, &listaArgumentos, COMMON_DELIMITER[0]);
131
132      // Tomamos el nombre de archivo
```

jun 25, 13 13:44 **client_manejador_de_notificaciones.cpp** Page 3/3

```
133         std::string archivoEntrante = listaArgumentos.verPrimero();
134         listaArgumentos.eliminarPrimero();
135
136         // Mensaje de log
137         std::string log = "NOTIFICACIÓN: Arribo de partes del archivo '";
138         log += archivoEntrante + "'.";
139         this→logger→emitirLog(log);
140
141         // Tomamos la cantidad total de bytes del archivo
142         unsigned int cantTotalBytes;
143         cantTotalBytes = Convertir::stoui(listaArgumentos.verPrimero());
144         listaArgumentos.eliminarPrimero();
145
146         // Creamos lista de bloques a reemplazar
147         Lista< std::pair< int, std::string > > listaBloquesAReemplazar;
148
149         // LLenamos la lista de bloques a reemplazar
150         while(¬listaArgumentos.estaVacia()) {
151           // Tomamos un número de bloque
152           int numBloque = Convertir::stoi(listaArgumentos.verPrimero());
153           listaArgumentos.eliminarPrimero();
154
155           // Tomamos el contenido del bloque
156           std::string contenidoBloque = listaArgumentos.verPrimero();
157           listaArgumentos.eliminarPrimero();
158
159           // Insertamos bloque en lista de bloques
160           listaBloquesAReemplazar.insertarUltimo(std::make_pair(
161             numBloque, contenidoBloque));
162         }
163
164         // Derivamos al receptor de archivos
165         this→receptorDeArchivos→recibirModificaciones(archivoEntrante,
166           cantTotalBytes, listaBloquesAReemplazar);
167       }
168       else if(instruccion ≡ S_NO_SUCH_FILE) {
169         // Mensaje de log
170         this→logger→emitirLog("NOTIFICACIÓN: Archivo '" + args
171           + "' ya no se encuentra en el servidor.");
172       }
173     }
174 }
```

jun 25, 13 13:44 **client_main.cpp** Page 1/1

```
1  //
2  //  ARCHIVOS UBICUOS
3  //  Programa principal del CLIENTE
4  //
5  //  ********************************************************************************
6  //
7  //  Facultad de Ingeniería − UBA
8  //  75.42 Taller de Programación I
9  //  Trabajo Práctico N°5
10 //
11 //  ALUMNOS:
12 //  Belén Beltran (91718) − belubeltran@gmail.com
13 //  Fiona Gonzalez Lisella (91454) − fgonzalezlisella@gmail.com
14 //  Federico Martín Rossi (92086) − federicomrossi@gmail.com
15 //
16 //  ********************************************************************************
17
18
19
20 #include <iostream>
21 #include "client_interfaz_conexion.h"
22 #include "client_cliente.h"
23
24
25
26
27 int main (int argc, char** argv) {
28   try{
29     // Iniciamos interfaz de la ventana principal
30     Gtk::Main kit(argc, argv);
31
32     // Creamos el cliente
33     Cliente *cli = new Cliente;
34
35     // Creamos la configuracion del cliente
36     Configuracion* configs = new Configuracion();
37
38     // ventana principal del programa
39     Conexion ventanaConexion(cli, configs);
40     ventanaConexion.correr();
41
42     // Liberamos toda la memoria
43     delete configs;
44     delete cli;
45   }
46   catch(char const * e) {
47     std::cout << e << std::endl;
48   }
49
50   return 0;
51 }
```

jun 25, 13 13:44 **client_interfaz_configuracion.h** Page 1/1

```
1  //
2  //  client_interfaz_iconfiguracion.h
3  //  CLASE INTERFAZ DE CONFIGURACION
4  //
5
6
7  #ifndef ICONFIGURACION_H_
8  #define ICONFIGURACION_H_
9
10
11 #include "client_configuracion.h"
12 #include "gtkmm.h"
13
14
15
16
17 class IConfiguracion : public Gtk::Window {
18 private:
19
20   // Atributos de la interfaz
21   Gtk::Window* main;
22
23   Gtk::Button *botonGuardar;
24   Gtk::Button *botonCancelar;
25   Gtk::Entry  *host;
26   Gtk::Entry  *puerto;
27   Gtk::Entry  *directorio;
28   Gtk::Entry  *iPolling;
29
30
31   Configuracion* config;
32   int flag; // Indica si el cliente esta conectado
33
34
35 public:
36
37   // Constructor
38   IConfiguracion(Configuracion *config, int flag);
39
40   // Destructor
41   virtual ~IConfiguracion();
42
43   // Inicia la ejecución de la ventana
44   void correr();
45
46 protected:
47
48   void on_buttonGuardar_clicked();
49   void on_buttonCancelar_clicked();
50 };
51
52 #endif /* ICONFIGURACION_H_ */
```

jun 25, 13 13:44 **client_interfaz_configuracion.cpp** Page 1/2

```
1  #include <iostream>
2  #include <string>
3
4  #include "client_interfaz_configuracion.h"
5  #include "common_convertir.h"
6
7
8
9
10 IConfiguracion::IConfiguracion(Configuracion *config, int flag) {
11   // Cargamos la ventana
12   Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
13
14
15
16   // Cargamos elementos
17   refBuilder→add_from_file("./interfaz/client_configuracion.glade");
18
19
20   refBuilder→get_widget("main", this→main); // linkeo el form
21
22   refBuilder→get_widget("host", this→host);
23   refBuilder→get_widget("port", this→puerto);
24   refBuilder→get_widget("dir", this→directorio);
25   refBuilder→get_widget("polling", this→iPolling);
26
27   refBuilder→get_widget("guardar", this→botonGuardar);
28   refBuilder→get_widget("cancelar", this→botonCancelar);
29
30
31   this→botonGuardar→signal_clicked().connect(sigc::mem_fun(*this, &IConfiguracion::on_buttonGuardar_clicked));
32   this→botonCancelar→signal_clicked().connect(sigc::mem_fun(*this, &IConfiguracion::on_buttonCancelar_clicked));
33
34   this→flag = flag;
35   main→show_all_children();
36 }
37
38
39 void IConfiguracion::on_buttonGuardar_clicked() {
40
41   //obtengo cada valor almacenado en los textBox
42
43   string unHost = this→host→get_text();
44   string unPuerto = this→puerto→get_text();
45   string unDir = this→directorio→get_text();
46   string unPolling = this→iPolling→get_text();
47
48   this→config→guardarCambios(unHost, unPuerto, unDir, unPolling);
49
50   this→main→hide();
51
52 }
53
54 void IConfiguracion::on_buttonCancelar_clicked() {
55   // No hago nada, retorno sin cambios en el archivo de settings.
56   this→main→hide();
57
58 }
59
60
61 void IConfiguracion::correr() {
62
63   //cargo los textBox con info
64   string auxPuerto = Convertir::itos(this→config→obtenerPuerto());
```

jun 25, 13 13:44 **client_interfaz_configuracion.cpp** Page 2/2

```
65    string auxPolling = Convertir::itos(this→config→obtenerIntervaloDePolling())
      ;
66
67    this→host→set_text(this→config→obtenerHost());
68    this→puerto→set_text(auxPuerto);
69    this→directorio→set_text(this→config→obtenerDirectorio());
70    this→iPolling→set_text(auxPolling);
71
72    if (this→flag ≡ 1) {
73      this→host→set_sensitive(false);
74      this→puerto→set_sensitive(false);
75      this→iPolling→set_sensitive(false);
76      this→directorio→set_sensitive(false);
77    }
78    //Muestro configuracion actual
79    Gtk::Main::run(*main);
80
81
82  }
83
84  IConfiguracion::~IConfiguracion() { }
85
```

jun 25, 13 13:44 **client_interfaz_conexion.h** Page 1/1

```
1   //
2   //  client_interfaz_conexion.h
3   //  CLASE CONEXION
4   //
5
6
7   #ifndef CONEXION_H_
8   #define CONEXION_H_
9
10
11  #include "gtkmm.h"
12  #include "client_cliente.h"
13  #include "common_thread.h"
14  #include "client_interfaz_configuracion.h"
15
16
17
18  class Conexion : public Gtk::Window , Thread {
19  private:
20
21    // Atributos de la interfaz
22    Gtk::Window* main;        // Ventana Conexion
23
24    Gtk::Label  *lblError;      // Etiqueta de error
25    Gtk::Button *botonConectar;   // Botón Conectar
26    Gtk::Button *botonSalir;    // Botón Salir
27    Gtk::Entry  *usuarioTextBox;    // Textbox de nombre de usuario
28    Gtk::Entry  *passTextBox;   // Textbox de la contraseña de usuario
29
30    //Atributos del menu
31    Gtk::ImageMenuItem *menuPref;
32    Gtk::ImageMenuItem *menuSalir;
33
34
35
36
37    // Atributos del modelo
38    Cliente *cliente;         // Cliente a través del cual se conecta
39    Configuracion *clienteConfig;
40    int estadoConexion;
41
42  public:
43
44    // Constructor
45    Conexion(Cliente *cliente, Configuracion* clienteConfig);
46
47    // Destructor
48    virtual ~Conexion();
49
50    // Inicia la ejecución de la ventana
51    void correr();
52
53  protected:
54
55    void on_buttonConectar_clicked();
56    void on_buttonSalir_clicked();
57    void on_menuPref_activate();
58    void on_menuSalir_activate();
59    void run();
60
61  };
62
63  #endif /* CONEXION_H_ */
```

```
#include <iostream>
#include <string>
#include "client_configuracion.h"

#include "client_interfaz_conexion.h"



Conexion::Conexion(Cliente *cliente, Configuracion* clienteConfig) : cliente(cliente), clienteConfig(clienteConfig) {
  // Cargamos la ventana
  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();


  // Cargamos elementos
  refBuilder->add_from_file("./interfaz/client_conexion.glade");


  refBuilder->get_widget("conexion", this->main); // linkeo el form

  refBuilder->get_widget("usuarioTxt", this->usuarioTextBox);
  refBuilder->get_widget("passTxt", this->passTextBox);

  refBuilder->get_widget("conectar", this->botonConectar);
  refBuilder->get_widget("lblError", this->lblError);
  refBuilder->get_widget("preferencias", this->menuPref);
  refBuilder->get_widget("msalir", this->menuSalir);
  refBuilder->get_widget("Salir", this->botonSalir);

  this->botonConectar->signal_clicked().connect(sigc::mem_fun(*this, &Conexion::on_buttonConectar_clicked));

  this->botonSalir->signal_clicked().connect(sigc::mem_fun(*this, &Conexion::on_buttonSalir_clicked));
  this->menuPref->signal_activate().connect(sigc::mem_fun(*this, &Conexion::on_menuPref_activate));
  this->menuSalir->signal_activate().connect(sigc::mem_fun(*this, &Conexion::on_menuSalir_activate));

  main->show_all_children();

}


void Conexion::on_buttonConectar_clicked() {
  // Deshabilitamos objetos de la ventana
  this->botonConectar->set_sensitive(false);
  this->usuarioTextBox->set_sensitive(false);
  this->passTextBox->set_sensitive(false);
  this->lblError->set_text("");

  // Obtenemos la configuracion actual del cliente

  cliente->especificarNombreHost(this->clienteConfig->obtenerHost());
  cliente->especificarPuerto(this->clienteConfig->obtenerPuerto());
  cliente->especificarDirectorio(this->clienteConfig->obtenerDirectorio());

  std::string user = this->usuarioTextBox->get_text();
  std::string pass = this->passTextBox->get_text();

  // Iniciamos conexión
  this->estadoConexion = cliente->conectar(user, pass);

  if(this->estadoConexion == 1) {
    // Abrimos ventana de actualización
    this->main->set_sensitive(false);
```



```
    this->lblError->set_text("");


     this->lblError->set_text("Sincronizando datos");

    this->cliente->iniciarSincronizacion(
      this->clienteConfig->obtenerIntervaloDePolling());
    this->start();


      // Habilitamos ventana luego de la actualización
    this->main->set_sensitive(true);
  }
  else if(this->estadoConexion == 0) {
    // Mostramos mensaje de error en ventana
    this->lblError->set_text("Usuario y/o contraseña inválidos");
    this->lblError->set_visible(true);

    // Borramos el contenido del password para ser nuevamente escrito
    this->passTextBox->set_text("");

    // Habilitamos objetos de la ventana
    this->botonConectar->set_sensitive(true);
    this->usuarioTextBox->set_sensitive(true);
    this->passTextBox->set_sensitive(true);

  }
  else if(this->estadoConexion == -1) {
    // Mostramos mensaje de error en ventana
    this->lblError->set_text("Falló la conexión con el servidor.");
    this->lblError->set_visible(true);


    // Habilitamos objetos de la ventana
    this->botonConectar->set_sensitive(true);
    this->usuarioTextBox->set_sensitive(true);
    this->passTextBox->set_sensitive(true);

  }
}

void Conexion::on_buttonSalir_clicked() {
  this->estadoConexion = 0;
  this->join();
  this->cliente->desconectar();

  Gtk::Main::quit();
}

void Conexion::on_menuPref_activate() {


  IConfiguracion ventanaSettings(this->clienteConfig, this->estadoConexion);
  ventanaSettings.correr();

  this->main->set_sensitive(true);
}

void Conexion::on_menuSalir_activate() {
  this->estadoConexion = 0;
  this->join();
  this->cliente->desconectar();

  Gtk::Main::quit();

}
```

jun 25, 13 13:44 **client_interfaz_conexion.cpp** Page 3/3

```
128
129 void Conexion::run() {
130
131
132 while(this→cliente→estaSincronizando() ≡ true ∧ this→estadoConexion ≡ 1 ){
133     this→lblError→set_text("Conectado al servidor");
134     sleep(1);
135 }
136
137   this→usuarioTextBox→set_sensitive(true);
138   this→passTextBox→set_sensitive(true);
139   this→lblError→set_text("Se cayó la conexion, ingrese para sincronizar");
140   this→botonConectar→set_sensitive(true);
141   this→estadoConexion = 0;
142   this→cliente→detenerSincronizacion();
143
144
145
146 }
147
148 void Conexion::correr(){
149
150   Gtk::Main::run(*main);
151 }
152
153 Conexion::~Conexion() { }
154
```

jun 25, 13 13:44 **client_interfaz_actualizacion.h** Page 1/1

```
1  //
2  //  client_interfaz_actualizacion.h
3  //  CLASE INTERFAZ DE ACTUALIZACION
4  //
5
6
7  #ifndef IACTUALIZACION_H_
8  #define IACTUALIZACION_H_
9
10
11 #include "gtkmm.h"
12 #include "common_thread.h"
13 #include "client_cliente.h"
14
15
16
17 class IActualizacion : public Gtk::Window, public Thread {
18 private:
19
20   // Atributos de la interfaz
21   Gtk::Window* main;      // Ventana
22   Cliente *cliente;       // Cliente que se actualiza
23
24 public:
25
26   // Constructor
27   IActualizacion (Cliente *cliente);
28
29   // Destructor
30   virtual ~IActualizacion();
31
32   // Define tareas a ejecutar en el hilo.
33   virtual void run();
34   void detener();
35
36   void correr();
37 };
38
39 #endif
```

jun 25, 13 13:44 **client_interfaz_actualizacion.cpp** Page 1/1

```cpp
//
//  client_interfaz_actualizacion.h
//  CLASE INTERFAZ DE ACTUALIZACION
//


#include <iostream>
#include <string>
#include "client_interfaz_actualizacion.h"


// Constructor
IActualizacion::IActualizacion(Cliente *cliente) : cliente(cliente) {
  // Cargamos la ventana
  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();

  // Cargamos elementos
  refBuilder→add_from_file("./interfaz/client_actualizacion.glade");

  refBuilder→get_widget("ventanaActualizacion", this→main);
  this→main→show_all_children();
}



IActualizacion::~IActualizacion() { }


// Define tareas a ejecutar en el hilo.
void IActualizacion::run() {
  this→main→show();

  // Esperamos a que se termine de actualizar el directorio
  while(this→isActive()) {
    sleep(2);
  }


  this→main→hide();
  this→stop();
}

void IActualizacion::correr() {
  Gtk::Main::run(*main);


}

void IActualizacion::detener() {
  this→stop();
}
```

jun 25, 13 13:44 **client_inspector.h** Page 1/2

```cpp
//
//  client_inspector.h
//  CLASE INSPECTOR
//


#ifndef INSPECTOR_H
#define INSPECTOR_H

#include <string>
#include "common_thread.h"
#include "common_lista.h"
#include "common_manejador_de_archivos.h"
#include "common_logger.h"
#include "client_sincronizador.h"




/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


class Inspector : public Thread {
private:

  ManejadorDeArchivos *manejadorDeArchivos;   // Manejador de archivos
  Sincronizador *sincronizador;          // Sincronizador
  unsigned int intervalo;              // Intervalo de inspección
                                 // en segundos
  Logger *logger;                  // Logger de eventos
  Mutex m;                    // Mutex

  // Bloquea actividades hasta que haya transcurrido el intervalo de polling
  void alarmaDeInspeccion();

public:

  // Constructor
  // PRE: 'intervalo' es el intervalo de inspección en segundos.
  Inspector(ManejadorDeArchivos *unManejador, Sincronizador *sincronizador,
    unsigned int intervalo, Logger *logger);

  // Destructor
  ~Inspector();

  // Inicia el ciclo de inspecciones
  void iniciar();

  // Detiene el ciclo de inspecciones
  void detener();

  // Setea los segundos de intervalo entre sucesivas inspecciones.
  // PRE: 'segundos' es la cantidad de segundos entre inspecciones.
  void establecerIntervaloDeInspeccion(unsigned int segundos);

  // Define tareas a ejecutar en el hilo.
  // Realiza una inspección cada un intervalo predeterminado.
  virtual void run();

  // Inspecciona si los hashes de los bloques pasados por parámetro
  // coinciden con los del archivo local. Si no coinciden, se encarga de
  // indicar que bloques deben ser solicitados al servidor.
  // PRE: 'nombreArchivo' es el nombre del archivo a verificar;
  // 'cantBYtesTotal' es la cantidad total de bytes que debe tener el
```

jun 25, 13 13:44 **client_inspector.h** Page 2/2

```
67    // archivo; bloques es una lista de pares (bloque, hash), con los números
68    // de bloque a verificar.
69    void inspeccionarArchivo(std::string nombreArchivo, unsigned int&
70      cantBytesTotal, Lista< std::pair< int, std::string > > bloques);
71
72    // Inspecciona si existe un archivo en el directorio local. Si no existe
73    // se encarga de indicar que debe ser solicitado al servidor.
74    void inspeccionarExisteArchivo(std::string& nombreArchivo,
75      std::string hashArchivo);
76  };
77
78  #endif
```

jun 25, 13 13:44 **client_inspector.cpp** Page 1/5

```
1   //
2   //  client_inspector.h
3   //  CLASE INSPECTOR
4   //
5
6
7
8   #include "client_inspector.h"
9   #include "common_cola.h"
10  #include "common_lista.h"
11  #include <string>
12  #include <utility>
13  #include <unistd.h>
14
15
16
17
18
19  /* ******************************************************************************
20   * DEFINICIÓN DE LA CLASE
21   * ******************************************************************************/
22
23
24  // Constructor
25  // PRE: 'intervalo' es el intervalo de inspección en segundos.
26  Inspector::Inspector(ManejadorDeArchivos *unManejador,
27    Sincronizador *sincronizador, unsigned int intervalo, Logger *logger) :
28    manejadorDeArchivos(unManejador), sincronizador(sincronizador),
29    intervalo(intervalo), logger(logger) { }
30
31
32  // Destructor
33  Inspector::~Inspector() { }
34
35
36  // Inicia el ciclo de inspecciones
37  void Inspector::iniciar() {
38    this→start();
39  }
40
41
42  // Detiene el ciclo de inspecciones
43  void Inspector::detener() {
44    this→stop();
45
46    // Interrumpimos el intervalo de inspección
47    this→interruptSleep();
48  }
49
50
51  // Setea los segundos de intervalo entre sucesivas inspecciones.
52  // PRE: 'segundos' es la cantidad de segundos entre inspecciones.
53  void Inspector::establecerIntervaloDeInspeccion(unsigned int segundos) {
54    this→intervalo = segundos;
55  }
56
57
58  // Define tareas a ejecutar en el hilo.
59  // Realiza una inspección cada un intervalo predeterminado.
60  void Inspector::run() {
61    // Inspeccionamos cada cierto intervalo hasta detener hilo
62    while(this→isActive()) {
63      // Nos detenemos hasta que suene la alarma de inspección
64      this→alarmaDeInspeccion();
65
66      // Si se detuvo al inspector, salimos
```

```
67      if(¬this→isActive()) return;
68
69      // Bloqueamos el mutex
70      Lock l(m);
71
72      // Realizamos la inspección
73      Cola< std::pair< std::string, std::string > > nuevos;
74      Cola< std::pair< std::string, Lista< int > > > modificados;
75      Cola< std::string > eliminados;
76
77      if(this→manejadorDeArchivos→actualizarRegistroDeArchivos(&nuevos,
78        &modificados, &eliminados)) {
79        // Mensaje de log
80        this→logger→emitirLog("INSPECTOR: Se detectaron cambios.");
81
82        while(¬nuevos.vacia()) {
83          // Tomamos nuevo
84          std::pair< std::string, std::string > nuevo;
85          nuevo = nuevos.pop_bloqueante();
86          std::string contenido;
87
88          try {
89            contenido = this→manejadorDeArchivos→obtenerContenido(
90              nuevo.first);
91          }
92          catch(char const * e) {
93            // Si no es posible abrir el archivo, dejamos que la
94            // próxima inspección se encarge de detectar que debe
95            // hacer con este.
96            continue;
97          }
98
99          // Mensaje de log
100         this→logger→emitirLog("INSPECTOR: Archivo nuevo '" +
101           nuevo.first + "'");
102
103         // Enviamos al sincronizador
104         this→sincronizador→enviarArchivo(nuevo.first, contenido,
105           nuevo.second);
106       }
107
108       while(¬modificados.vacia()) {
109         // Tomamos modificado
110         std::pair< std::string, Lista<int> > mod;
111         mod = modificados.pop_bloqueante();
112
113         // Lista de bloques auxiliar
114         Lista< std::pair< int, std::string > > bloques;
115
116         // Iteramos sobre los bloques que fueron modificados
117         while(¬mod.second.estaVacia()) {
118           // Tomamos el número de bloque y su contenido
119           int bloque = mod.second.verPrimero();
120           mod.second.eliminarPrimero();
121           std::string contenido;
122
123           try {
124             contenido = this→manejadorDeArchivos→obtenerContenido
125               (mod.first, bloque);
126           }
127           catch(char const * e) {
128             // Si no es posible abrir el archivo, dejamos que la
129             // próxima inspección se encarge de detectar que debe
130             // hacer con este.
131             continue;
132           }
```



```
133
134           // Insertamos par de bloques con sus contenidos en lista
135           bloques.insertarUltimo(std::make_pair(bloque, contenido));
136         }
137
138         // Mensaje de log
139         this→logger→emitirLog("INSPECTOR: Archivo '" + mod.first
140           + "' fue modificado.");
141
142         // Enviamos modificaciones del archivo al sincronizador
143         this→sincronizador→modificarArchivo(mod.first,
144           this→manejadorDeArchivos→obtenerCantBytes(mod.first),
145             bloques);
146       }
147
148       while(¬eliminados.vacia()) {
149         // Tomamos eliminado
150         std::string elim = eliminados.pop_bloqueante();
151
152         // Mensaje de log
153         this→logger→emitirLog("INSPECTOR: Archivo '" + elim
154           + "' fue eliminado.");
155
156         // Enviamos a sincronizador
157         this→sincronizador→eliminarArchivo(elim);
158       }
159     }
160   }
161 }
162
163
164 // Inspecciona si los hashes de los bloques pasados por parámetro
165 // coinciden con los del archivo local. Si no coinciden, se encarga de
166 // indicar que bloques deben ser solicitados al servidor.
167 // PRE: 'nombreArchivo' es el nombre del archivo a verificar;
168 // 'cantBYtesTotal' es la cantidad total de bytes que debe tener el
169 // archivo; bloques es una lista de pares (bloque, hash), con los números
170 // de bloque a verificar.
171 void Inspector::inspeccionarArchivo(std::string nombreArchivo, unsigned int&
172     cantBytesTotal, Lista< std::pair< int, std::string > > bloques) {
173   // Bloqueamos el mutex
174   Lock l(m);
175
176   // Mensaje de log
177   this→logger→emitirLog("INSPECTOR: Inspeccionando archivo '" +
178     nombreArchivo + "' en directorio local.");
179
180   // Si no existe, lo solicitamos al servidor
181   if(¬this→manejadorDeArchivos→existeArchivo(nombreArchivo)) {
182     this→sincronizador→solicitarArchivoNuevo(nombreArchivo);
183     return;
184   }
185
186   // Tomamos bytes actuales de archivo local
187   unsigned int b;
188   b = this→manejadorDeArchivos→obtenerCantBytes(nombreArchivo);
189
190   // Caso en que solo se modificó el largo del archivo
191   if(bloques.estaVacia() ∧ (cantBytesTotal ≠ b)) {
192     Lista< std::pair< int, std::string > > auxiliar;
193
194     // Enviamos a modificar el archivo para que se achice su tamaño
195     this→manejadorDeArchivos→modificarArchivo(nombreArchivo,
196       cantBytesTotal, auxiliar);
197
198     return;
```

```
199   }
200
201   Lista< int > bloquesASolicitar;
202
203   // Corroboramos que bloques se necesitan medir
204   while(¬bloques.estaVacia()) {
205     // Comprobamos si es necesario solicitar el bloque
206     bool solicitar;
207     solicitar = ¬this→manejadorDeArchivos→compararBloque(nombreArchivo,
208       bloques.verPrimero().first, bloques.verPrimero().second);
209
210     // Si se requiere el1 bloque, lo listamos para solicitarlo
211     if(solicitar)
212       bloquesASolicitar.insertarUltimo(bloques.verPrimero().first);
213
214     bloques.eliminarPrimero();
215   }
216
217   // Si los bloques son compatibles con la versión enviada, retornamos
218   if(bloquesASolicitar.estaVacia()) return;
219
220   // Solicitamos al servidor los bloques
221   this→sincronizador→solicitarBloquesModificados(nombreArchivo,
222     bloquesASolicitar);
223 }
224
225
226 // Inspecciona si existe un archivo en el directorio local. Si no existe
227 // se encarga de indicar que debe ser solicitado al servidor.
228 void Inspector::inspeccionarExisteArchivo(std::string& nombreArchivo,
229   std::string hashArchivo) {
230   // Bloqueamos el mutex
231   Lock l(m);
232
233   // Mensaje de log
234   this→logger→emitirLog("INSPECTOR: Inspeccionando existencia de archivo '"
235     + nombreArchivo + "' en directorio local.");
236
237   // Corroboramos si existe el archivo
238   if(this→manejadorDeArchivos→existeArchivoEnRegitro(nombreArchivo)) {
239     // Tomamos el hash del archivo local
240     std::string hashArchivoLoc;
241     this→manejadorDeArchivos→obtenerHash(nombreArchivo, hashArchivoLoc);
242
243     // Si tenemos archivos iguales, retornamos sin solicitar nada
244     if(hashArchivo ≡ hashArchivoLoc)
245       return;
246     // Sino, se debe pedir
247     else {
248       this→sincronizador→solicitarArchivoNuevo(nombreArchivo);
249       return;
250     }
251   }
252
253   // Si no existe o es viejo, lo solicitamos al servidor
254   if(¬this→manejadorDeArchivos→existeArchivo(nombreArchivo))
255     this→sincronizador→solicitarArchivoNuevo(nombreArchivo);
256 }
257
258
259
260
261
262 /*
263  * IMPLEMENTACIÓN DE MÉTODOS PRIVADOS DE LA CLASE
264  */
```



```
265
266
267 // Bloquea actividades hasta que haya transcurrido el intervalo de polling
268 void Inspector::alarmaDeInspeccion() {
269   this→sleep(this→intervalo);
270 }
```

```
1  //
2  //  client_emisor.h
3  //  CLASE EMISOR
4  //
5
6
7  #ifndef EMISOR_H
8  #define EMISOR_H
9
10 #include <string>
11 #include "common_cola.h"
12 #include "common_thread.h"
13 #include "common_socket.h"
14 #include "common_comunicador.h"
15 #include "common_logger.h"
16 #include "common_seguridad.h"
17
18
19
20
21 /* ******************************************************************************
22  * DECLARACIÓN DE LA CLASE
23  * ******************************************************************************/
24
25
26 class Emisor : public Thread {
27 private:
28
29   Socket *socket;             // Socket por el que envía datos
30   Cola< std::string > salida;       // Cola de salida
31   Comunicador com;            // Comunicador del emisor
32   Logger *logger;             // Logger de eventos
33   std::string clave;            // Clave utilizada para firmar
34                       // mensajes
35   bool activa;              // Flag de emisión activa
36
37
38 public:
39
40   // Constructor
41   Emisor(Socket *socket, Logger *logger, const std::string &clave);
42
43   // Destructor
44   ~Emisor();
45
46   // Inicia la emisión
47   void iniciar();
48
49   // Detiene la emisión
50   void detener();
51
52   // Agrega un mensaje a enviar en la cola de salida del emisor.
53   // PRE: 'mensaje' es la cadena de texto que se desea enviar
54   void ingresarMensajeDeSalida(std::string mensaje);
55
56   // Define tareas a ejecutar en el hilo.
57   // Se encarga de emitir lo que se encuentre en la cola de salida.
58   virtual void run();
59
60   // Comprueba si la emisión se encuentra activa. Se encontrará activa
61   // mientras el socket permanezca activo, lo cual se considera desde que se
62   // inicia el objeto con el metodo iniciar(). En caso de cerrarse el socket
63   // se devolverá false, mientras que al estar activa la recepción se
64   // retornará true.
65   bool emisionActiva();
66 };
```



```
67
68 #endif
```

```
1  //
2  //  client_emisor.h
3  //  CLASE EMISOR
4  //
5
6
7  #include "client_emisor.h"
8
9  // DEBUG
10 #include <iostream>
11 // END DEBUG
12
13
14
15 namespace {
16   const std::string COLA_SALIDA_FIN = "COLA-SALIDA-FIN";
17 }
18
19
20
21
22
23 /* ************************************************************************
24  * DEFINICIÓN DE LA CLASE
25  * ************************************************************************/
26
27
28 // Constructor
29 Emisor::Emisor(Socket *socket, Logger *logger, const std::string &clave) :
30   socket(socket), com(socket), logger(logger), clave(clave),
31   activa(false) { }
32
33
34 // Destructor
35 Emisor::~Emisor() { }
36
37
38 // Inicia la emisión
39 void Emisor::iniciar() {
40   // Iniciamos el hilo
41   this→start();
42
43   // Habilitamos flag de recepción
44   this→activa = true;
45 }
46
47
48 // Detiene la emisión
49 void Emisor::detener() {
50   // Detenemos hilo
51   this→stop();
52
53   // Esperamos a que se termine de emitir los mensajes de la cola
54   while(¬this→salida.vacia());
55
56   // Destrabamos la cola encolando un mensaje de finalización detectable
57   this→salida.push(COLA_SALIDA_FIN);
58 }
59
60
61 // Agrega un mensaje a enviar en la cola de salida del emisor.
62 // PRE: 'mensaje' es la cadena de texto que se desea enviar
63 void Emisor::ingresarMensajeDeSalida(std::string mensaje) {
64   this→salida.push(mensaje);
65 }
66
```



```
67
68 // Define tareas a ejecutar en el hilo.
69 // Se encarga de emitir lo que se encuentre en la cola de salida.
70 void Emisor::run() {
71   // Emitimos lo que vaya siendo insertado en la cola de salida. Ante una
72   // detención del thread, se seguirá emitiendo hasta vaciar la cola de
73   // salida.
74   while(this→isActive() ∨ ¬this→salida.vacia()) {
75     // Tomamos un mensaje de salida
76     std::string mensaje = this→salida.pop_bloqueante();
77
78     // Corroboramos si no se ha desencolado el mensaje que marca el fin
79     if(mensaje ≡ COLA_SALIDA_FIN) return;
80
81     // Se firma el mensaje
82     mensaje = Seguridad::obtenerFirma(mensaje, this→clave) +
83       COMMON_DELIMITER + mensaje;
84
85     // Enviamos mensaje
86     if(this→com.emitir(mensaje) ≡ -1) {
87       // Mensaje de log
88       this→logger→emitirLog("ERROR: Emisor no pudo emitir mensaje.");
89
90       // Deshabilitamos flag de emisión
91       this→activa = false;
92     }
93   }
94 }
95
96
97 // Comprueba si la emisión se encuentra activa. Se encontrará activa
98 // mientras el socket permanezca activo, lo cual se considera desde que se
99 // inicia el objeto con el metodo iniciar(). En caso de cerrarse el socket
100 // se devolverá false, mientras que al estar activa la recepción se
101 // retornará true.
102 bool Emisor::emisionActiva() {
103   return this→activa;
104 }
```

```
1  //
2  //  client_configuracion.h
3  //  CLASE CONFIGURACION
4  //
5
6
7  #ifndef CONFIGURACION_H
8  #define CONFIGURACION_H
9
10 #include "common_archivoTexto.h"
11
12 // CONSTANTES
13 namespace {
14
15   // Metadatos sobre el archivo de configuración
16   const std::string CONFIG_DIR = "config/";
17   const std::string CONFIG_FILENAME = "general";
18   const std::string CONFIG_FILE_EXT = ".properties";
19
20   // Parámetros configurables
21   const std::string CONFIG_P_DIR = "DIRECTORIO";
22   const std::string CONFIG_P_HOST = "HOSTNAME";
23   const std::string CONFIG_P_PORT = "PUERTO";
24   const std::string CONFIG_P_POLL = "POLLING";
25
26   // Separadores
27   const std::string CONFIG_SEPARATOR = "=";
28
29   // Indicador de comentarios
30   const std::string CONFIG_COMMENT = "#";
31 }
32
33
34
35
36 /* ************************************************************************
37  * DECLARACIÓN DE LA CLASE
38  * ************************************************************************/
39
40
41 class Configuracion {
42 private:
43   ArchivoTexto* Archivo;
44
45 public:
46
47   // Constructor
48   Configuracion();
49
50   // Destructor
51   ~Configuracion();
52
53   // Devuelve el valor especifico que se necesita
54   std::string getInfo(std :: string &cadena);
55
56   // Devuelve el directorio en el que se desea sincronizar.
57   std::string obtenerDirectorio();
58
59   // Devuelve el host del servidor.
60   std::string obtenerHost();
61
62   // Devuelve el puerto del servidor.
63   int obtenerPuerto();
64
65   // Devuelve el intervalo de polling en segundos.
66   int obtenerIntervaloDePolling();
```



```
67
68   // Guarda cambios realizados sobre la configuracion.
69   void guardarCambios(string host,string puerto,string dir, string polling);
70
71 };
72
73 #endif
```

```
1  //
2  //  client_configuracion.h
3  //  CLASE CONFIGURACION
4  //
5
6
7  #include <iostream>
8  #include <string>
9  #include <sstream>
10 #include "common_convertir.h"
11 #include "client_configuracion.h"
12
13 using namespace std;
14
15
16
17
18
19
20
21 /* ************************************************************************
22  * DEFINICIÓN DE LA CLASE
23  * ************************************************************************/
24
25
26 // Constructor
27 Configuracion::Configuracion() {
28
29 }
30
31
32 // Destructor
33 Configuracion::~Configuracion() { }
34
35
36 // Devuelve el valor especifico que se necesita
37 std::string Configuracion::getInfo(std :: string &cadena) {
38   string val;
39   unsigned pos = cadena.find(CONFIG_SEPARATOR);         // position of "=" in ca
   dena
40   val = cadena.substr (pos+1);
41   return val;
42 }
43
44 // Devuelve el directorio en el que se desea sincronizar.
45 std::string Configuracion::obtenerDirectorio() {
46   string* cadena = new string();
47   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
48     CONFIG_FILE_EXT,0);
49   bool estado = false;
50   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_DIR)));
51   string result = getInfo(*cadena);
52   delete(this→Archivo);
53   delete(cadena);
54   return result;
55 }
56
57 // Devuelve el host del servidor.
58 std::string Configuracion::obtenerHost() {
59   string* cadena = new string();
60   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
61     CONFIG_FILE_EXT,0);
62   bool estado = false;
63   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_HOST)));
64   string result = getInfo(*cadena);
65   delete(this→Archivo);
```



```
66   delete(cadena);
67   return result;
68 }
69
70
71 // Devuelve el puerto del servidor.
72 int Configuracion::obtenerPuerto() {
73   string* cadena = new string();
74   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
75     CONFIG_FILE_EXT,0);
76   bool estado = false;
77   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_PORT)));
78   string result = getInfo(*cadena);
79   delete(this→Archivo);
80   delete(cadena);
81   return Convertir:: stoi(result);
82 }
83
84 // Devuelve el intervalo de polling en segundos.
85 int Configuracion::obtenerIntervaloDePolling() {
86   string* cadena = new string();
87   this→Archivo = new ArchivoTexto (CONFIG_DIR + CONFIG_FILENAME +
88     CONFIG_FILE_EXT,0);
89   bool estado = false;
90   while(estado ≡ (this→Archivo→leerLinea(*cadena, '\n', CONFIG_P_POLL)));
91   string result = getInfo(*cadena);
92   delete(this→Archivo);
93   delete(cadena);
94   return Convertir:: stoi(result);
95 }
96
97 void Configuracion::guardarCambios(string host,string puerto,string dir, string
   polling) {
98
99   this→Archivo = new ArchivoTexto(CONFIG_DIR + CONFIG_FILENAME +
100    CONFIG_FILE_EXT,1);
101  string* aux = new string();
102  *aux += "#SETTINGS USER";
103  *aux += '\n';
104  this→Archivo→escribir(*aux);
105  aux→clear();
106  *aux += CONFIG_P_HOST;
107  *aux += CONFIG_SEPARATOR;
108  *aux += host;
109  *aux += '\n';
110  this→Archivo→escribir(*aux);
111
112  aux→clear();
113  *aux += CONFIG_P_PORT;
114  *aux += CONFIG_SEPARATOR;
115  *aux += puerto;
116  *aux += '\n';
117  this→Archivo→escribir(*aux);
118
119  aux→clear();
120  *aux += CONFIG_P_DIR;
121  *aux += CONFIG_SEPARATOR;
122  *aux += dir;
123  *aux += '\n';
124  this→Archivo→escribir(*aux);
125
126  aux→clear();
127  *aux += CONFIG_P_POLL;
128  *aux += CONFIG_SEPARATOR;
129  *aux += polling;
130  *aux += '\n';
```

jun 25, 13 13:44 **client_configuracion.cpp** Page 3/3

```
131     this→Archivo→escribir(*aux);
132
133     delete(this→Archivo);
134     delete(aux);
135
136 }
137
```

jun 25, 13 13:44 **client_config.h** Page 1/1

```
1  //
2  //  client_config.h
3  //
4  //  Cabecera con constantes de configuración de uso interno
5  //
6
7
8  #ifndef CONFIG_H
9  #define CONFIG_H
10
11 #include <string>
12
13
14
15
16 /* ************************************************************************
17  * CONSTANTES DE CONFIGURACIÓN INTERNA
18  * ************************************************************************/
19
20
21 // Constantes para los logs
22 const std::string LOGGER_RUTA_LOG = "logs/";
23 const std::string LOGGER_NOMBRE_LOG = "eventos_cliente";
24
25
26 #endif
```

jun 25, 13 13:44 **client_cliente.h** Page 1/2

```
1  //
2  //  client_cliente.h
3  //  CLASE CLIENTE
4  //
5
6
7  #ifndef CLIENTE_H
8  #define CLIENTE_H
9
10
11 #include "common_socket.h"
12 #include "common_manejador_de_archivos.h"
13 #include "common_logger.h"
14 #include "client_emisor.h"
15 #include "client_receptor.h"
16 #include "client_sincronizador.h"
17 #include "client_receptor_de_archivos.h"
18 #include "client_inspector.h"
19 #include "client_manejador_de_notificaciones.h"
20 #include "common_protocolo.h"
21 class Comunicador;
22
23
24
25
26 /* ************************************************************************
27  * DECLARACIÓN DE LA CLASE
28  * ************************************************************************/
29
30
31 class Cliente {
32 private:
33
34   // Atributos generales
35   Socket *socket;         // Socket con el que se comunica
36   int puerto;           // Puerto de conexión.
37   std::string nombreHost;     // Nombre del host de conexión
38   std::string directorio;     // Directorio que será sincronizado
39   bool estadoConexion;      // Censa si se encuentra conectado
40   bool actualizando;        // Censa si se encuentra actualizando
41                 // el directorio.
42   std::string clave;    // Clave con la que firman los mensajes
43
44   //Atributos de módulos
45   Emisor *emisor;
46   Receptor *receptor;
47   ManejadorDeArchivos *manejadorDeArchivos;
48   Sincronizador *sincronizador;
49   ReceptorDeArchivos *receptorDeArchivos;
50   Inspector *inspector;
51   ManejadorDeNotificaciones *manejadorDeNotificaciones;
52   Logger *logger;
53
54
55   // Inicia sesion con usuario existente
56   int iniciarSesion(std::string usuario, std::string clave);
57
58
59 public:
60
61   // Constructor
62   Cliente();
63
64   // Destructor
65   ~Cliente();
66
```

jun 25, 13 13:44 **client_cliente.h** Page 2/2

```
67   // Establece el nombre de host al que se conectará el cliente.
68   void especificarNombreHost(std::string nombreHost);
69
70   // Establece el puerto del host al que se conectará el cliente
71   void especificarPuerto(int puerto);
72
73   // Establece el directorio que sincronizará el cliente
74   void especificarDirectorio(std::string directorio);
75
76   // Realiza la conexión inicial con el servidor.
77   // PRE: 'usuario' y 'clave' son el nombre de usuario y contraseña con el
78   // que se desea conectar al servidor. Debe haberse especificado el nombre
79   // de host, puerto y directorio.
80   // POST: devuelve '-1' si falló la conexión, '0' si falló el login y '1' si
81   // se conectó y loggeó con éxito.
82   int conectar(std::string usuario, std::string clave);
83
84   // Se desconecta del servidor
85   void desconectar();
86
87   // Inicializa la sincronización del cliente con el servidor.
88   // PRE: debe ejecutarse previamente el método conectar() y debe haberse
89   // también especificado el nombre de host, puerto y directorio. De lo
90   // contrario, no se inicializará la sincronización. 'intervaloPolling' es
91   // el intervalo de polling que se desea al inicializar la sincronización.
92   void iniciarSincronizacion(int intervaloPolling);
93
94   // Permite cambiar el intervalo de polling estando en curso la
95   // sincronización.
96   // PRE: debe haber sido iniciada la sincronización. 'intervalo' es el
97   // intervalo de polling expresado en segundos.
98   void cambiarIntervaloPolling(unsigned int intervalo);
99
100  // Detiene la sincronización y se desconecta del servidor.
101  // PRE: previamente debió haberse iniciado la sincronización.
102  // POST: la conexión con el servidor finalizó. Si se desea volver a iniciar
103  // la sincronización, debe realizarse la conexión nuevamente.
104  void detenerSincronizacion();
105
106  // Comprueba si se encuentra realizando la actualización inicial
107  // que se inicia al invocar al metodo iniciarSincronizacion().
108  // POST: devuelve true si se encuentra actualizando o false en
109  // caso contrario.
110  bool estaActualizando();
111
112  // Comprueba si se encuentra activa la conexión con el servidor y si se
113  // encuentra sincronizando.
114  // POST: devuelve true si se encuentra activo o false en su defecto.
115  bool estaSincronizando();
116 };
117
118 #endif
```

```
//
//  client_cliente.h
//  CLASE CLIENTE
//


#include <iostream>
#include <sstream>
#include "common_comunicador.h"
#include "common_convertir.h"
#include "client_config.h"
#include "client_actualizador.h"
#include "client_cliente.h"




/* ******************************************************************************
 * DEFINICIÃM−^SN DE LA CLASE
 * ******************************************************************************/


// Constructor
Cliente::Cliente() : estadoConexion(false), actualizando(true) {
  // Creamos el logger
  this→logger = new Logger(LOGGER_RUTA_LOG + LOGGER_NOMBRE_LOG);
}


// Destructor
Cliente::~Cliente() {
  // Liberamos la memoria utilizada por el socket
  delete this→socket;
  delete this→logger;
}


// Establece el nombre de host al que se conectarÃ¡ el cliente.
void Cliente::especificarNombreHost(std::string nombreHost) {
  this→nombreHost = nombreHost;
}


// Establece el puerto del host al que se conectarÃ¡ el cliente
void Cliente::especificarPuerto(int puerto) {
  this→puerto = puerto;
}


// Establece el directorio que sincronizarÃ¡ el cliente
void Cliente::especificarDirectorio(std::string directorio) {
  this→directorio = directorio;
}


// Realiza la conexiÃ³n inicial con el servidor.
// PRE: 'usuario' y 'clave' son el nombre de usuario y contraseÃ±a con el
// que se desea conectar al servidor. Debe haberse especificado el nombre
// de host, puerto y directorio.
// POST: devuelve '−1' si fallÃ³ la conexiÃ³n, '0' si fallÃ³ el login y '1' si
// se conectÃ³ y loggeÃ³ con Ã©xito.
int Cliente::conectar(std::string usuario, std::string clave) {
  // Creamos socket
  this→socket = new Socket();
  this→socket→crear();

```



```
  // Mensaje de log
  this→logger→emitirLog("Conectando con " + this→nombreHost +
    " en el puerto " + Convertir::itos(this→puerto));

  try {
    // Conectamos el socket
    this→socket→conectar(nombreHost, puerto);
  }
  catch(char const * e) {
    // Mensaje de log
    this→logger→emitirLog("No se ha podido conectar con servidor.");
    this→logger→emitirLog(e);

    // Liberamos memoria
    delete this→socket;

    // FallÃ³ la conexiÃ³n
    return −1;
  }

  // Mensaje de log
  this→logger→emitirLog("ConexiÃ³n establecida con servidor.");

  // Si se iniciÃ³ sesiÃ³n con Ã©xito, salimos y mantenemos socket activo
  if(iniciarSesion(usuario, clave) ≡ 1) {
    // Cambiamos el estado de la conexiÃ³n
    this→estadoConexion = true;

    return 1;
  }

  // Destruimos el socket en caso de fallar el inicio de sesiÃ³n
  desconectar();
  delete this→socket;

  // FallÃ³ el Äºoggin
  return 0;
}


// Se desconecta del servidor
void Cliente::desconectar() {
  // Mensaje de log
  this→logger→emitirLog("Cerrando conexiÃ³n...");

  // Desconectamos el socket
  this→socket→cerrar();

  // Cambiamos el estado de la conexiÃ³n
  this→estadoConexion = false;

  // Mensaje de log
  this→logger→emitirLog("Se ha cerrado la conexiÃ³n con el servidor.");
}


// Inicializa la sincronizaciÃ³n del cliente con el servidor.
// PRE: debe ejecutarse previamente el mÃ©todo conectar() y debe haberse
// tambiÃ©n especificado el nombre de host, puerto y directorio. De lo
// contrario, no se inicializarÃ¡ la sincronizaciÃ³n. 'intervaloPolling' es
// el intervalo de polling que se desea al inicializar la sincronizaciÃ³n.
void Cliente::iniciarSincronizacion(int intervaloPolling) {
  // Si la conexiÃ³n no se encuentra activa, no hacemos nada
  if(¬estadoConexion) return;

  // Activamos flag de actualizaciÃ³n
```

```
133   this→actualizando = true;
134
135   // Creamos los mÃ³dulos primarios
136   this→emisor = new Emisor(this→socket, this→logger, this→clave);
137   this→receptor = new Receptor(this→socket, this→logger, this→clave);
138   this→manejadorDeArchivos = new ManejadorDeArchivos(this→directorio,
139     this→logger);
140
141   // Ponemos en marcha los mÃ³dulos
142   this→receptor→iniciar();
143   this→emisor→iniciar();
144
145   // Iniciamos la actualizaciÃ³n del directorio local
146   Actualizador actualizador(this→emisor, this→receptor,
147     this→manejadorDeArchivos, this→logger);
148   actualizador.ejecutarActualizacion();
149
150   // Creamos los mÃ³dulos para la sincronizaciÃ³n en tiempo real
151   this→sincronizador = new Sincronizador(emisor, this→logger);
152   this→receptorDeArchivos = new ReceptorDeArchivos(manejadorDeArchivos,
153     this→logger);
154   this→inspector = new Inspector(manejadorDeArchivos, sincronizador,
155     intervaloPolling, this→logger);
156   this→manejadorDeNotificaciones = new ManejadorDeNotificaciones(receptor,
157     inspector, receptorDeArchivos, this→logger);
158
159   // Activamos flag de actualizaciÃ³n
160   this→actualizando = false;
161
162   // Ponemos en marcha los mÃ³dulos
163   this→inspector→iniciar();
164   this→manejadorDeNotificaciones→start();
165 }
166
167
168 // Permite cambiar el intervalo de polling estando en curso la
169 // sincronizaciÃ³n.
170 // PRE: debe haber sido iniciada la sincronizaciÃ³n. 'intervalo' es el
171 // intervalo de polling expresado en segundos.
172 void Cliente::cambiarIntervaloPolling(unsigned int intervalo) {
173   try {
174     this→inspector→establecerIntervaloDeInspeccion(intervalo);
175   }
176   catch (...) {
177     std::cerr << "ERROR: Debe inicializarse la sincronizaciÃ³n"
178       << std::endl;
179   }
180 }
181
182
183 // Detiene la sincronizaciÃ³n y se desconecta del servidor.
184 // PRE: previamente debiÃ³ haberse iniciado la sincronizaciÃ³n.
185 // POST: la conexiÃ³n con el servidor finalizÃ³. Si se desea volver a iniciar
186 // la sincronizaciÃ³n, debe realizarse la conexiÃ³n nuevamente.
187 void Cliente::detenerSincronizacion() {
188   // Detenemos los mÃ³dulos
189   this→inspector→detener();
190   this→inspector→join();
191   this→emisor→detener();
192   this→receptor→detener();
193   this→manejadorDeNotificaciones→stop();
194   this→manejadorDeNotificaciones→join();
195   this→emisor→join();
196
197   // Se desconecta del servidor
198   this→desconectar();
```



```
199
200   this→receptor→join();
201
202   // Liberamos la memoria utilizada por los mÃ³dulos
203   delete this→emisor;
204   delete this→receptor;
205   delete this→manejadorDeArchivos;
206   delete this→sincronizador;
207   delete this→receptorDeArchivos;
208   delete this→inspector;
209   delete this→manejadorDeNotificaciones;
210 }
211
212
213 // Comprueba si se encuentra realizando la actualizaciÃ³n inicial
214 // que se inicia al invocar al metodo iniciarSincronizacion().
215 // POST: devuelve true si se encuentra actualizando o false en
216 // caso contrario.
217 bool Cliente::estaActualizando() {
218   return this→actualizando;
219 }
220
221
222 // Comprueba si se encuentra activa la conexiÃ³n con el servidor y si se
223 // encuentra sincronizando.
224 // POST: devuelve true si se encuentra activo o false en su defecto.
225 bool Cliente::estaSincronizando() {
226   try{
227     bool e = this→emisor→emisionActiva();
228     bool r = this→receptor→recepcionActiva();
229
230     if(e ∨ r) return true;
231     else return false;
232   }
233   catch(...) {
234     return false;
235   }
236 }
237
238
239
240
241 /*
242  * IMPLEMENTACIÃM−^SN DE MÃM−^ITODOS PRIVADOS DE LA CLASE
243  */
244
245
246 // Inicia sesion con usuario existente
247 int Cliente::iniciarSesion(std::string usuario, std::string clave) {
248   // Creamos comunicador
249   Comunicador com(this→socket);
250
251   // Mensaje de log
252   this→logger→emitirLog("Emitiendo solicitud de LOGIN...");
253
254   // Se preparan los argumentos
255   std::string hashClave = Hash::funcionDeHash(clave);
256   std::string mensaje = usuario + COMMON_DELIMITER + hashClave;
257
258   // Enviamos peticiÃ³n de inicio de sesion.
259   if(com.emitir(C_LOGIN_REQUEST, mensaje) ≡ −1) return −1;
260
261   // Se obtiene respuesta del servidor
262   std::string args;
263   if(com.recibir(mensaje, args) ≡ −1) return −1;
264
```

jun 25, 13 13:44 **client_cliente.cpp** Page 5/5

```
265   if (mensaje ≡ S_LOGIN_OK) {
266     // Mensaje de log
267     this→logger→emitirLog("Inicio de sesiÃ³n exitoso con usuario '" +
268       usuario + "'");
269
270     // Se guarda la clave para enviar mensajes con firma
271     this→clave = hashClave;
272
273     return 1;
274   }
275   else if (mensaje ≡ S_LOGIN_FAIL) {
276     // Mensaje de log
277     this→logger→emitirLog("FallÃ³ inicio de sesiÃ³n con usuario '" +
278       usuario + "'");
279
280     return 0;
281   }
282
283   return −1;
284 }
```

jun 25, 13 13:44 **client_actualizador.h** Page 1/1

```
1  //
2  //  client_actualizador.h
3  //  CLASE ACTUALIZADOR
4  //
5
6
7  #ifndef ACTUALIZADOR_H
8  #define ACTUALIZADOR_H
9
10
11 #include <string>
12 #include "common_lista.h"
13 #include "common_logger.h"
14 #include "common_manejador_de_archivos.h"
15 #include "client_emisor.h"
16 #include "client_receptor.h"
17
18
19
20
21
22 /* ******************************************************************************
23  * DECLARACIÓN DE LA CLASE
24  * ******************************************************************************/
25
26
27 class Actualizador {
28 private:
29
30   Emisor *emisor;                // Emisor de mensajes
31   Receptor *receptor;              // Receptor de mensajes
32   ManejadorDeArchivos *manejadorDeArchivos;   // Manejador de archivos
33   int porcentajeDeActualizacion;          // Contador que sensa
34                                 // cuanto se ha actualizado
35   Logger *logger;                // Logger de eventos
36
37 public:
38
39   // Constructor
40   Actualizador(Emisor *emisor, Receptor *receptor,
41     ManejadorDeArchivos *manejadorDeArchivos, Logger *logger);
42
43   // Destructor
44   ~Actualizador();
45
46   // Inicia la actualización del directorio
47   void ejecutarActualizacion();
48 };
49
50 #endif
```

```
1   //
2   //  client_actualizador.h
3   //  CLASE ACTUALIZADOR
4   //
5
6
7   #include <sstream>
8   #include <utility>
9   #include "common_protocolo.h"
10  #include "common_parser.h"
11  #include "common_convertir.h"
12  #include "common_lista.h"
13  #include "client_actualizador.h"
14
15
16
17
18
19  /* ************************************************************************
20   * DEFINICIÓN DE LA CLASE
21   * ************************************************************************/
22
23
24  // Constructor
25  Actualizador::Actualizador(Emisor *emisor, Receptor *receptor,
26      ManejadorDeArchivos *manejadorDeArchivos, Logger *logger) :
27      emisor(emisor), receptor(receptor),
28      manejadorDeArchivos(manejadorDeArchivos), porcentajeDeActualizacion(0),
29      logger(logger) { }
30
31
32  // Destructor
33  Actualizador::~Actualizador() { }
34
35
36  // Inicia la recepción
37  void Actualizador::ejecutarActualizacion() {
38    // Mensaje de log
39    this→logger→emitirLog("Actualizando directorio...");
40    this→logger→emitirLog("Solicitando lista de archivos del servidor...");
41
42    // Creamos el registro de archivos en caso de que no exista
43    if(this→manejadorDeArchivos→crearRegistroDeArchivos());
44
45    // Solicitamos la lista de archivos del servidor
46    this→emisor→ingresarMensajeDeSalida(C_GET_FILES_LIST);
47
48    std::string instruccion, args;
49
50    // Esperamos a recibir la lista de archivos desde el servidor
51    while(instruccion ≠ S_FILES_LIST) {
52      std::string msg = this→receptor→obtenerMensajeDeEntrada();
53      Parser::parserInstruccion(msg, instruccion, args);
54    }
55
56    // Mensaje de log
57    this→logger→emitirLog("Se recibió lista de archivos del servidor...");
58    this→logger→emitirLog("Procesando lista de archivos...");
59
60    // Parseamos la lista de archivos enviada por el servidor
61    Lista< std::string > listaArgumentos_1;
62    Parser::dividirCadena(args, &listaArgumentos_1, COMMON_DELIMITER[0]);
63
64    // Obtenemos la cantidad de archivos envió el servidor, de acuerdo al
65    // protocolo
66    int cantidadArchivos = Convertir::stoi(listaArgumentos_1.verPrimero());
```



```
67    listaArgumentos_1.eliminarPrimero();
68
69    // Armamos lista de pares para poder procesar en manejador de archivos
70    Lista< std::pair< std::string, std::pair< std::string, int > > >
71      listaServidor;
72
73    for(int i = 0; i < cantidadArchivos; i++) {
74      // Tomamos nombre de archivo
75      std::string nombreArchivo = listaArgumentos_1.verPrimero();
76      listaArgumentos_1.eliminarPrimero();
77      // Tomamos hash de archivo
78      std::string hashArchivo = listaArgumentos_1.verPrimero();
79      listaArgumentos_1.eliminarPrimero();
80      // Tomamos cantidad de bloques de archivo
81      int cantBloquesArchivo;
82      cantBloquesArchivo = Convertir::stoi(listaArgumentos_1.verPrimero());
83      listaArgumentos_1.eliminarPrimero();
84
85      // Formamos el par con la información necesaria del archivo
86      std::pair< std::string, int > infoArchivo;
87      infoArchivo = std::make_pair(hashArchivo, cantBloquesArchivo);
88
89      std::pair< std::string, std::pair< std::string, int > > archivo;
90      archivo = std::make_pair(nombreArchivo, infoArchivo);
91
92      listaServidor.insertarUltimo(archivo);
93    }
94
95    // Procesamos lista de archivos del servidor comparándola con el directorio
96    // local y obteniendo las actualizaciones pertinentes
97    Lista< std::pair< std::string, Lista< int > > > listaFaltantes;
98    Lista< std::string > listaSobrantes;
99
100   this→manejadorDeArchivos→obtenerListaDeActualizacion(&listaServidor,
101     &listaFaltantes, &listaSobrantes);
102
103
104   // Eliminamos archivos sobrantes
105   for(size_t i = 0; i < listaSobrantes.tamanio(); i++) {
106     std::string archivo = listaSobrantes[i];
107     this→manejadorDeArchivos→eliminarArchivo(archivo);
108   }
109
110   // Se crea una lista de nuevos archivos y otra de modificados
111   // en el server
112   Lista<std::string> nuevosActualizables;
113   Lista<std::string> modificadosActualizables;
114
115   // Realizamos la petición de envío y espera de recepción de archivos
116   // faltantes
117   for(size_t i = 0; i < listaFaltantes.tamanio(); i++) {
118     // Tomamos uno de los archivos faltantes de la lista
119     std::string nombreArchivoFaltante = listaFaltantes[i].first;
120     Lista< int > listaBloques = listaFaltantes[i].second;
121
122     // Emisión de la petición de archivo
123     std::string mensaje;
124     mensaje.append(C_FILE_PARTS_REQUEST);
125     mensaje.append(" ");
126     mensaje.append(nombreArchivoFaltante);
127
128     // Insertamos numeros de bloque en mensaje
129     for(size_t i = 0; i < listaBloques.tamanio(); i++) {
130       mensaje.append(COMMON_DELIMITER);
131       mensaje.append(Convertir::itos(listaBloques[i]));
132     }
```

```
133
134       // Mensaje de log
135       this→logger→emitirLog("Se solicitaron archivos y partes faltantes.");
136
137       // Emitimos mensaje
138       this→emisor→ingresarMensajeDeSalida(mensaje);
139
140       std::string instr, args;
141
142       // Esperamos a recibir el archivo
143       while(instr ≠ COMMON_FILE_PARTS ∧ instr ≠ S_NO_SUCH_FILE) {
144         std::string msg = this→receptor→obtenerMensajeDeEntrada();
145         Parser::parserInstruccion(msg, instr, args);
146       }
147
148       // Mensaje de log
149       this→logger→emitirLog("Se recibieron archivos y partes faltantes.");
150
151       // Si el servidor notifica que ya no existe el archivo, salteamos
152       if(instr ≡ S_NO_SUCH_FILE) continue;
153
154
155       // Parseamos los argumentos de la respuesta
156       Lista< std::string > listaArgumentos_2;
157       Parser::dividirCadena(args, &listaArgumentos_2, COMMON_DELIMITER[0]);
158
159       // Descartamos el primer argumento, referido al nombre de archivo, el
160       // cual es ya conocido
161       std::string archivoFaltanteEntrante = listaArgumentos_2.verPrimero();
162       listaArgumentos_2.eliminarPrimero();
163
164       // Tomamos la cantidad total de bytes del archivo
165       unsigned int cantTotalBytes;
166       cantTotalBytes = Convertir::stoui(listaArgumentos_2.verPrimero());
167       listaArgumentos_2.eliminarPrimero();
168
169       // Creamos lista de bloques a reemplazar
170       Lista< std::pair< int, std::string > > listaBloquesAReemplazar;
171
172       // LLenamos la lista de bloques a reemplazar
173       while(¬listaArgumentos_2.estaVacia()) {
174         // Tomamos un número de bloque
175         int numBloque = Convertir::stoi(listaArgumentos_2.verPrimero());
176         listaArgumentos_2.eliminarPrimero();
177
178         // Tomamos el contenido del bloque
179         std::string contenidoBloque = listaArgumentos_2.verPrimero();
180         listaArgumentos_2.eliminarPrimero();
181
182         // Caso en que estamos recibiendo un archivo entero
183         if(numBloque ≡ 0) {
184           this→manejadorDeArchivos→agregarArchivo(
185             nombreArchivoFaltante, contenidoBloque);
186
187           // Se agrega a lista de nuevos
188           nuevosActualizables.insertarUltimo(
189             archivoFaltanteEntrante);
190
191           continue;
192         }
193
194         // Insertamos bloque en lista de bloques
195         listaBloquesAReemplazar.insertarUltimo(std::make_pair(numBloque,
196           contenidoBloque));
197
198         // Se agrega a lista de modificados
```



```
199         modificadosActualizables.insertarUltimo(
200           archivoFaltanteEntrante);
201       }
202
203       // Si la lista de bloques esta vacía, salteamos
204       if(listaBloquesAReemplazar.estaVacia()) continue;
205
206       // Modificamos el archivo
207       this→manejadorDeArchivos→modificarArchivo(nombreArchivoFaltante,
208         cantTotalBytes, listaBloquesAReemplazar);
209     }
210
211     // Mensaje de log
212     this→logger→emitirLog("Actualizando registro de archivos locales...");
213
214     // Actualizamos el registro de archivos
215     this→manejadorDeArchivos→actualizarRegistroDeArchivos(
216       nuevosActualizables, modificadosActualizables);
217
218     // Mensaje de log
219     this→logger→emitirLog("Finalizada la actualización de archivos.");
220   }
```

jun 25, 13 13:44 **Table of Content** Page 1/3

Table of Contents

jun 25, 13 13:44 **Table of Content** Page 2/3

jun 25, 13 13:44 **Table of Content** Page 3/3