may 13, 13 21:40 **server_servidor.h** Page 1/1

```
//
//  server_servidor.h
//  CLASE SERVIDOR
//


#ifndef SERVIDOR_H
#define SERVIDOR_H


#include "common_thread.h"
#include "common_socket.h"
#include "common_lista.h"
#include "server_conexion_cliente.h"
#include "server_controlador_de_tareas.h"




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


class Servidor : public Thread {
private:

  Lista<ConexionCliente*> *clientes;      // Lista de clientes conectados
  int puerto;                 // Puerto en el que se escucha.
  Socket socket;                // Socket en el que escucha el
                        // servidor.
  ControladorDeTareas *controlador;     // Controlador de tareas.
  Lista<std::string> *claves;         // Lista de posibles claves.

  // Cierra todas las conexiones existentes con clientes y elimina todo
  // registro de estos,quedando vacía la lista de clientes.
  void cerrarConexionesConClientes();

public:

  // Constructor
  Servidor(int puerto, std::string& msg, int numDigitosClave,
    int numClientes);

  // Destructor
  ~Servidor();

  // Define tareas a ejecutar en el hilo.
  // Mantiene a la escucha al servidor y acepta nuevos clientes.
  virtual void run();

  // Inicia la ejecución del servidor. No debe utilizarse el método start()
  // para iniciar. En caso de error lanza una excepción.
  void iniciar();

  // Detiene la ejecución del servidor. No debe utilizarse el método stop()
  // para detener.
  void detener();

  // Envía a la salida estándar la situación en la que se encuentra al
  // momento de ser invocada.
  void imprimirSituacion();
};

#endif
```

may 13, 13 21:40 **server_servidor.cpp** Page 1/3

```
//
//  server_servidor.cpp
//  CLASE SERVIDOR
//


#include <iostream>
#include "server_servidor.h"
#include "common_lock.h"



// Constantes
namespace {
  const int MAX_CONEXIONES = 10;
}



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


// Constructor
Servidor::Servidor(int puerto, std::string& msg,
  int numDigitosClave, int numClientes) : puerto(puerto) {
  // Creamos la lista de clientes conectados
  this→clientes = new Lista<ConexionCliente*>;

  // Creamos la lista de posibles claves
  this→claves = new Lista<std::string>;

  // Creamos el controlador de tareas para el servidor
  this→controlador = new ControladorDeTareas(numDigitosClave,
    numClientes, msg, this→claves);
}


// Destructor
Servidor::~Servidor() {
  // Concluimos la conexión con clientes existentes
  this→cerrarConexionesConClientes();

  // Liberamos espacio utilizado por atributos
  delete this→clientes;
  delete this→claves;
  delete this→controlador;
}


// Define tareas a ejecutar en el hilo.
// Mantiene a la escucha al servidor y acepta nuevos clientes.
void Servidor::run() {
  try {
    // Iniciamos la escucha del servidor
    this→socket.crear();
    this→socket.escuchar(MAX_CONEXIONES, this→puerto);
  }
  catch(char const * e) {
    // Mensaje de error
    std::cerr << e << std::endl;
    // Detenemos servidor de inmediato
    this→detener();
  }

```

```
67    // Nos ponemos a la espera de clientes que se conecten
68    while(this→isActive()) {
69      Socket *socketCLI = 0;
70
71      // Aceptamos nuevo cliente
72      socketCLI = this→socket.aceptar();
73
74      // Salimos si el socket no esta activo o si se interrumpió
75      // la escucha de solicitudes de conexión
76      if(¬this→socket.estaActivo() ∨ ¬socketCLI) break;
77
78      // Generamos una nueva conexión para escuchate
79      ConexionCliente *conexionCLI = new ConexionCliente(socketCLI,
80        this→controlador);
81
82      // Censamos al cliente en el servidor
83      this→clientes→insertarUltimo(conexionCLI);
84
85      // Damos la orden de que comience a ejecutarse el hilo del cliente.
86      conexionCLI→start();
87    }
88  }
89
90
91  // Inicia la ejecución del servidor. No debe utilizarse el método start()
92  // para iniciar. En caso de error lanza una excepción.
93  void Servidor::iniciar() {
94    // Iniciamos hilo de ejecución
95    this→start();
96  }
97
98
99  // Detiene la ejecución del servidor. No debe utilizarse el método stop()
100 // para detener.
101 void Servidor::detener() {
102   // Detenemos hilo
103   this→stop();
104
105   // Forzamos el cierre del socket para evitar nuevas conexiones entrantes
106   try {
107     this→socket.cerrar();
108   }
109   // Ante una eventual detención abrupta, previa a la inicialización del
110   // socket, lanzará un error que daremos por obviado.
111   catch(...) { }
112
113   // Concluimos la conexión con clientes existentes
114   this→cerrarConexionesConClientes();
115 }
116
117
118 // Envía a la salida estándar la situación en la que se encuentra al
119 // momento de ser invocada.
120 void Servidor::imprimirSituacion() {
121   // Caso en que no se completaron las tareas
122   if(¬this→controlador→seCompletaronTareas())
123     std::cout << "Not finished" << std::endl;
124   // Caso en que se completaron las tareas
125   else {
126     // No se encontraron claves
127     if(this→claves→tamanio() ≡ 0)
128       std::cout << "No keys found" << std::endl;
129     // Se encontró una única clave
130     else if (this→claves→tamanio() ≡ 1)
131       std::cout << this→claves→verPrimero() << std::endl;
132     // Se encontraron múltiples claves
```



```
133     else
134       std::cout << "Multiple keys found" << std::endl;
135   }
136 }
137
138
139 // Cierra todas las conexiones existentes con clientes y elimina todo
140 // registro de estos,quedando vacía la lista de clientes.
141 void Servidor::cerrarConexionesConClientes() {
142   // Detenemos y liberamos espacio utilizado por cada conexión cliente,
143   // dejando vacía la lista de clientes activos.
144   while(¬this→clientes→estaVacia()) {
145     // Obtenemos cliente y lo eliminamos de la lista
146     ConexionCliente *cc = this→clientes→verPrimero();
147     this→clientes→eliminarPrimero();
148
149     // Detenemos conexión con el cliente
150     cc→detener();
151     // Esperamos a que finalice
152     cc→join();
153     // Liberamos memoria
154     delete cc;
155   }
156 }
```

```
//
//  EL CÓDIGO DRAKA
//  Programa principal del SERVIDOR
//
//  ***************************************************************************
//
//  Facultad de Ingeniería − UBA
//  75.42 Taller de Programación I
//  Trabajo Práctico N°4
//
//  ALUMNO: Federico Martín Rossi
//  PADRÓN: 92086
//  EMAIL:  federicomrossi@gmail.com
//
//  ***************************************************************************
//
//  Programa servidor el cual se encarga de estar a la escucha de conexiones
//  entrantes por parte de clientes, y de fraccionar el trabajo a enviarles a
//  estos a medida que se van conectando. El trabajo se reparte entre un número
//  fijo de clientes especificado al iniciar el programa, provocando que, una
//  vez repartidas todas las partes, los demás clientes sean notificados de la
//  inexistencia de parte de trabajo a asignarles.
//
//
//
//  FORMA DE USO
//  ============
//
//  Deberá ejecutarse el programa en la línea de comandos de la siguiente
//  manera:
//
//    # ./server [PUERTO] [ARCHIVO] [NUM−DIGITOS−CLAVE] [NUM−CLIENTES]
//
//  donde,
//
//    PUERTO: es el servidor donde deberá escuchar el servidor;
//    ARCHIVO: es la ruta al archivo binario conteniendo los datos
//         encriptados;
//    NUM−DIGITOS−CLAVE: es el npumero de dígitos de la clave
//    NUM−CLIENTES: es el número de clientes entre los que se dividirá el
//    trabajo
//
//



#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "common_convertir.h"
#include "common_mutex.h"
#include "common_lock.h"
#include "server_servidor.h"



namespace {
  // Constantes que definen los comandos válidos
  const std::string CMD_SALIR = "q";
}



/* ***************************************************************************
 * FUNCIONES AUXILIARES
 * ***************************************************************************/
```



```


// Función que se encarga de abrir un archivo, devolviendo su contenido en
// hexadecimla como un string.
// PRE: 'archivo' es la ruta del archivo, incluyendo directorio y extensión.
// POST: se devuelve un string con contenido en formato hexadecimal.
std::string abrirArchivoEncriptado(const std::string& archivo) {
  // Abrimos el archivo con el mensaje encriptado
  std::ifstream archivoMsg(archivo.c_str(),
    std::ios::in | std::ios::binary | std::ios::ate);

  if(¬archivoMsg.is_open())
    throw "ERROR: Archivo de entrada inválido.";

  std::ifstream::pos_type size;
  uint8_t * msgTemp;

  // Almacenamos momentaneamente el mensaje original
  size = archivoMsg.tellg();
  msgTemp = new uint8_t[size];
  archivoMsg.seekg(0, std::ios::beg);
  archivoMsg.read((char*)msgTemp, size);
  archivoMsg.close();

  // Convertimos el mensaje encriptado a hexadecimal
  std::string msg_hex(Convertir::uitoh(msgTemp, size));
  delete[] msgTemp;

  return msg_hex;
}



/* ***************************************************************************
 * PROGRAMA PRINCIPAL
 * ***************************************************************************/


int main(int argc, char* argv[]) {
  // Corroboramos cantidad de argumentos
  if(argc ≠ 5) {
    std::cerr << "ERROR: cantidad incorrecta de argumentos." << std::endl;
    return 1;
  }

  std::string msg;

  try {
    // Leemos archivo
    msg = abrirArchivoEncriptado(argv[2]);
  }
  catch(char const * e) {
    std::cerr << e << std::endl;
    return 1;
  }

  // Creamos el servidor
  Servidor *servidor = new Servidor(atoi(argv[1]), msg, atoi(argv[3]),
    atoi(argv[4]));

  try {
    // Iniciamos servidor
    servidor→iniciar();
  }
  catch(char const * e) {
    std::cerr << e << std::endl;
```

```
133       delete servidor;
134       return 1;
135     }
136
137
138     std::string comando;
139
140     // Esperamos a que se indique la finalización de la ejecución
141     while(comando ≠ CMD_SALIR)
142       getline(std::cin, comando);
143
144
145     servidor→detener();
146     servidor→join();
147
148     // Imprimimos situación del servidor luego de la ejecución del mismo
149     servidor→imprimirSituacion();
150
151     // delete terminal;
152     delete servidor;
153
154     return 0;
155   }
```



```
1   //
2   //  server_controlador_de_tareas.h
3   //  CLASE CONTROLADORDETAREAS
4   //
5
6
7   #ifndef CONTROLADOR_DE_TAREAS_H
8   #define CONTROLADOR_DE_TAREAS_H
9
10
11  #include "common_mutex.h"
12  #include "common_lista.h"
13
14
15
16
17  /* ******************************************************************************
18   * DECLARACIÓN DE LA CLASE
19   * ******************************************************************************/
20
21
22  class ControladorDeTareas {
23  private:
24
25    int numDigitosClave;          // Número de dígitos de la
26                                  // clave.
27    int numClientes;              // Número de clientes entre los
28                                  // que se dividirá el trabajo.
29    std::string msgEncriptado;        // Referencia al mensaje
30    Lista<std::string> *claves;       // Lista de posibles claves.
31    Mutex m;                  // Mutex
32    int numPartes;                // Cantidad de partes asignadas
33    bool asignacionCompleta;          // Flag que contiene info sobre
34                                  // si se asigno todo el trabajo
35    int clientesCorriendo;            // Contador de los clientes que
36                                  // se encuentran procesando
37
38    // Devuelve la clave inicial del espacio de claves a asignar a un cliente.
39    // PRE: 'numCliente' es el número de cliente que le ha sido asignado.
40    // POST: se devuelve una cadena con la clave inicial.
41    std::string claveInicialDeRangoDeClaves();
42
43    // Devuelve la clave final del espacio de claves a asignar a un cliente.
44    // PRE: 'numCliente' es el número de cliente que le ha sido asignado.
45    // POST: se devuelve una cadena con la clave final.
46    std::string claveFinalDeRangoDeClaves();
47
48  public:
49
50    // Constructor
51    ControladorDeTareas(int numDigitosClave, int numClientes,
52      std::string msgEncriptado, Lista<std::string> *claves);
53
54    // Destructor
55    ~ControladorDeTareas();
56
57    // Genera el mensaje a ser devuelto al cliente con las indicaciones de
58    // la tarea que debe realizar, si es que las hay.
59    // PRE: 'msg_tarea' es una referencia a la variable en donde se depositará
60    // el mensaje de indicación de tarea.
61    // POST: devuelve false si no hay tareas para asignar o true si se asignó
62    // tarea
63    bool obtenerIndicacion(std::string& msg_tarea);
64
65    // Recibe una posible clave para almacenar en el servidor
66    void ingresarClave(std::string clave);
```

```
67
68    // Permite a un cliente notificar que ha finalizado su tarea
69    void clienteTerminoTarea();
70
71    // Corrobora si se han terminado las tareas.
72    // POST: devuelve true si se completaron o false en su defecto.
73    bool seCompletaronTareas();
74  };
75
76  #endif
```



```
1   //
2   //  server_controlador_de_tareas.h
3   //  CLASE CONTROLADORDETAREAS
4   //
5
6
7   #include <iomanip>
8   #include <math.h>
9   #include <sstream>
10  #include "server_controlador_de_tareas.h"
11  #include "common_convertir.h"
12  #include "common_lock.h"
13  #include "common_protocolo.h"
14
15
16
17
18  /* ************************************************************************
19   * DEFINICIÓN DE LA CLASE
20   * ************************************************************************/
21
22
23  // Constructor
24  ControladorDeTareas::ControladorDeTareas(int numDigitosClave, int numClientes,
25      std::string msgEncriptado, Lista<std::string> *claves) :
26      numDigitosClave(numDigitosClave), numClientes(numClientes),
27      msgEncriptado(msgEncriptado), claves(claves), numPartes(0),
28      asignacionCompleta(false), clientesCorriendo(0) { }
29
30
31  // Destructor
32  ControladorDeTareas::~ControladorDeTareas() { }
33
34
35  // Genera el mensaje a ser devuelto al cliente con las indicaciones de
36  // la tarea que debe realizar, si es que las hay.
37  // PRE: 'msg_tarea' es una referencia a la variable en donde se depositará
38  // el mensaje de indicación de tarea.
39  // POST: devuelve false si no hay tareas para asignar o true si se asignó
40  // tarea
41  bool ControladorDeTareas::obtenerIndicacion(std::string& msg_tarea) {
42    // Bloqueamos el mutex
43    Lock l(this→m);
44
45    // Caso en el que se han asignado ya todas las partes
46    if(asignacionCompleta) {
47      msg_tarea = S_NO_JOB_PART;
48      return false;
49    };
50
51    // Calculamos rango de claves
52    std::string claveIni = claveInicialDeRangoDeClaves();
53    std::string claveFin = claveFinalDeRangoDeClaves();
54    std::string ind("");
55
56    // Generamos el mensaje
57    ind += S_JOB_PART + "" + this→msgEncriptado + "" +
58      Convertir::itos(this→numPartes) + "" +
59      Convertir::itos(this→numDigitosClave) + "" + claveIni + "" +
60      claveFin;
61
62    this→numPartes++;
63    msg_tarea = ind;
64
65    // Si esta fue la última parte que se debia asignar, seteamos en
66    // true el flag de asignaciones completadas
```

```
67    if(this→numPartes ≡ this→numClientes)
68      this→asignacionCompleta = true;
69
70    this→clientesCorriendo++;
71
72    return true;
73  }
74
75
76  // Recibe una posible clave para almacenar en el servidor
77  void ControladorDeTareas::ingresarClave(std::string clave) {
78    // Bloqueamos el mutex
79    Lock l(this→m);
80
81    this→claves→insertarUltimo(clave);
82  }
83
84
85  // Permite a un cliente notificar que ha finalizado su tarea
86  void ControladorDeTareas::clienteTerminoTarea() {
87    // Bloqueamos el mutex
88    Lock l(this→m);
89
90    // Decrementamos una unidad la cantidad de clientes corriendo
91    this→clientesCorriendo−−;
92  }
93
94
95  // Corrobora si se han terminado las tareas.
96  // POST: devuelve true si se completaron o false en su defecto.
97  bool ControladorDeTareas::seCompletaronTareas() {
98    // Bloqueamos el mutex
99    Lock l(this→m);
100
101   return (this→asignacionCompleta ∧ (clientesCorriendo ≡ 0));
102 }
103
104
105 // Devuelve la clave inicial del espacio de claves a asignar a un cliente.
106 // PRE: 'numCliente' es el número de cliente que le ha sido asignado.
107 // POST: se devuelve una cadena con la clave inicial.
108 std::string ControladorDeTareas::claveInicialDeRangoDeClaves() {
109   std::stringstream cotaInf;
110   int num = this→numPartes*(pow(10, this→numDigitosClave) /
111     this→numClientes);
112
113   cotaInf << std::setw(this→numDigitosClave) << std::setfill('0')
114     << num;
115
116   return cotaInf.str();
117 }
118
119
120 // Devuelve la clave final del espacio de claves a asignar a un cliente.
121 // PRE: 'numCliente' es el número de cliente que le ha sido asignado.
122 // POST: se devuelve una cadena con la clave final.
123 std::string ControladorDeTareas::claveFinalDeRangoDeClaves() {
124   std::stringstream cotaSup;
125
126   if(this→numPartes < this→numClientes − 1) {
127     int num = (this→numPartes + 1)*(pow(10, this→numDigitosClave) /
128       this→numClientes) − 1;
129
130     cotaSup << std::setw(this→numDigitosClave) << std::setfill('0')
131       << num;
132
```



```
133     return cotaSup.str();
134   }
135   else if (this→numPartes ≡ this→numClientes − 1) {
136     int num = pow(10, this→numDigitosClave) − 1;
137
138     cotaSup << std::setw(this→numDigitosClave) << std::setfill('0')
139       << num;
140
141     return cotaSup.str();
142   }
143
144   return "";
145 }
```

may 13, 13 21:40 **server_conexion_cliente.h** Page 1/1

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


#ifndef CONEXION_CLIENTE_H
#define CONEXION_CLIENTE_H



#include "common_thread.h"
#include "common_socket.h"
#include "server_controlador_de_tareas.h"




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


class ConexionCliente : public Thread {
private:

  Socket *socket;               // Socket de comunicación
  ControladorDeTareas *controlador;     // Controlador de tareas.

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.
  ConexionCliente(Socket *s, ControladorDeTareas *controlador);

  // Destructor
  ~ConexionCliente();

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

  // Detiene la conexión con el cliente. No debe utilizarse el método stop()
  // para detener, sino este mismo en su lugar.
  void detener();
};

#endif
```

may 13, 13 21:40 **server_conexion_cliente.cpp** Page 1/2

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


#include <sstream>
#include "server_conexion_cliente.h"
#include "common_comunicador.h"




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


// 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.
ConexionCliente::ConexionCliente(Socket *s, ControladorDeTareas *controlador)
  : socket(s), controlador(controlador) { }


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


// Define tareas a ejecutar en el hilo.
void ConexionCliente::run() {
  // Creamos el comunicador para enviar y recibir mensajes
  Comunicador comunicador(this→socket);

  // Variables de procesamiento
  std::string instruccion;
  std::string args;
  std::stringstream msg_in;
  std::string msg_tarea;


  // Esperamos hasta recibir el mensaje correcto
  while(instruccion ≠ C_GET_JOB_PART)
    if(comunicador.recibir(instruccion, args) ≡ −1) return;

  if(¬this→controlador→obtenerIndicacion(msg_tarea)) {
    // No hay tarea asignada
    comunicador.emitir(msg_tarea);
    this→socket→cerrar();
    return;
  }

  // Enviamos la parte del trabajo correspondiente
  if(comunicador.emitir(msg_tarea) ≡ −1) return;


  // Nos ponemos a la espera de posibles claves o de indicación de
  // finalización de tarea por parte del cliente
  while(this→isActive()) {
    // Recibimos mensaje
    if(comunicador.recibir(instruccion, args) ≡ −1) break;

```

may 13, 13 21:40 **server_conexion_cliente.cpp** Page 2/2

```
67      // Caso en que se recibe una posible clave
68      if(instruccion ≡ C_POSSIBLE_KEY)
69        this→controlador→ingresarClave(args);
70      else if (instruccion ≡ C_JOB_PART_FINISHED) {
71        this→controlador→clienteTerminoTarea();
72        break;
73      }
74    }
75
76    // Cerramos conexión
77    this→socket→cerrar();
78  }
79
80
81  // Detiene la conexión con el cliente. No debe utilizarse el método stop()
82  // para detener, sino este mismo en su lugar.
83  void ConexionCliente::detener() {
84    // Detenemos hilo
85    this→stop();
86
87    // Forzamos el cierre del socket y destrabamos espera de recepcion de datos
88    try {
89      this→socket→cerrar();
90    }
91    // Ante una eventual detención abrupta, previa a la inicialización del
92    // socket, lanzará un error que daremos por obviado.
93    catch(...) { }
94  }
```

may 13, 13 21:40 **common_thread.h** Page 1/1

```
1   //
2   //  common_thread.h
3   //  CLASE THREAD
4   //
5   //  Clase que implementa la interfaz para la creación de un hilo de ejecución.
6   //
7
8
9   #ifndef THREAD_H
10  #define THREAD_H
11
12
13  #include <pthread.h>
14
15
16
17
18  /* *****************************************************************************
19   * DECLARACIÓN DE LA CLASE
20   * ****************************************************************************/
21
22
23  class Thread {
24  private:
25
26    pthread_t thread;   // Identificador del hilo
27    bool status;      // Estado del thread
28
29    // Ejecuta el método run().
30    // PRE: 'threadID' es un puntero al thread.
31    static void* callback(void *threadID);
32
33    // Constructor privado
34    Thread(const Thread &c);
35
36  public:
37
38    // Constructor
39    Thread();
40
41    // Destructor
42    virtual ~Thread();
43
44    // Inicia el thread
45    virtual void start();
46
47    // Detiene el thread
48    virtual void stop();
49
50    // Envía una solicitud de cancelación al hilo, deteniendo abruptamente
51    // su ejecución
52    virtual void cancel();
53
54    // Bloquea hasta que el hilo finalice su ejecución en caso de estar
55    // ejecutandose.
56    virtual void join();
57
58    // Define tareas a ejecutar en el hilo.
59    virtual void run() = 0;
60
61    // Verifica si el hilo se encuentra activo.
62    // POST: devuelve true si está activo o false en caso contrario.
63    bool isActive();
64  };
65
66  #endif
```

```
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
11
12
13
14 /* ******************************************************************************
15  * DEFINICIÓN DE LA CLASE
16  * ******************************************************************************/
17
18
19 // Constructor
20 Thread::Thread() : status(false) { }
21
22
23 // Constructor privado
24 Thread::Thread(const Thread &c) { }
25
26
27 // Destructor
28 Thread::~Thread() { }
29
30
31 // Inicia el hilo
32 void Thread::start() {
33   pthread_create(&this→thread, 0, callback, this);
34 }
35
36
37 // Detiene el hilo
38 void Thread::stop() {
39   this→status = false;
40 }
41
42
43 // Envía una solicitud de cancelación al hilo, deteniendo abruptamente
44 // su ejecución
45 void Thread::cancel() {
46   pthread_cancel(this→thread);
47 }
48
49
50 // Bloquea hasta que el hilo finalice su ejecución en caso de estar
51 // ejecutandose.
52 void Thread::join() {
53   pthread_join(this→thread, 0);
54 }
55
56
57 // Verifica si el hilo se encuentra activo.
58 // POST: devuelve true si está activo o false en caso contrario.
59 bool Thread::isActive() {
60   return this→status;
61 }
62
63
64 // Ejecuta el método run().
65 // PRE: 'threadID' es un puntero al thread.
66 void* Thread::callback(void *threadID) {
```



```
67   ((Thread*)threadID)→status = true;
68   ((Thread*)threadID)→run();
69   ((Thread*)threadID)→status = false;
70   return 0;
71 }
```

```
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   * DECLARACIÓN DE LA CLASE
23   * ******************************************************************************/
24
25
26  class Socket {
27  private:
28
29    int sockfd;             // Filedescriptor del socket.
30    struct sockaddr_in miDir;     // Dirección del socket.
31    struct sockaddr_in destinoDir;    // Dirección del socket destino.
32    bool activo;            // Sensa si esta activo el socket
33
34    // Constructor privado.
35    // Crea un nuevo socket.
36    // PRE: 'sockfd' es un filedescriptor que identifica a un socket.
37    explicit Socket(const int sockfd);
38
39    // Enlaza (asocia) al socket con un puerto y una dirección IP.
40    // PRE: 'ip' es una cadena que contiene el nombre del host o la dirección
41    // IP a la que se desea asociar; 'puerto' es el puerto al que se desea
42    // enlazar.
43    // POST: devuelve 1 si se logró enlazar satisfactoriamente o −1 en caso de
44    // error
45    void enlazar(int puerto, std::string ip = "");
46
47  public:
48
49    // Constructor.
50    Socket();
51
52    // Destructor.
53    // Cierra el socket.
54    ~Socket();
55
56    // Crea el socket
57    // POST: lanza una excepción si no se logra llevar a cabo la creación.
58    void crear();
59
60    // Conecta el socket a una dirección y puerto destino.
61    // PRE: 'hostDestino' es una cadena que contiene el nombre del host o la
62    // dirección IP a la que se desea conectar; 'puertoDestino' es el puerto
63    // al que se desea conectar.
64    // POST: determina dirección y puertos locales si no se utilizó el método
65    // bind() previamente. Además, lanza una excepción si no se pudo llevar a
66    // cabo la conexión.
```



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



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

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



```
199   int yes = 1;
200
201   // Reutilizamos socket
202   if(setsockopt(this→sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
203     ≡ −1)
204     throw "ERROR: Antes de enlazar, no se pudo reutilizar socket.";
205
206   // Cargamos datos del enlace a realizar
207   this→miDir.sin_family = AF_INET;
208   this→miDir.sin_port = htons(puerto);
209
210   // Obtenemos host
211   if(ip ≡ "")
212     this→miDir.sin_addr.s_addr = htonl(INADDR_ANY);
213   else {
214     struct hostent *he = gethostbyname(ip.c_str());
215     this→miDir.sin_addr = *((struct in_addr *)he→h_addr);
216   }
217
218   memset(miDir.sin_zero, '\0', sizeof(miDir.sin_zero));
219
220   // Enlazamos
221   if(bind(this→sockfd, (struct sockaddr *)&miDir, sizeof(miDir)) < 0)
222     throw "ERROR: No se pudo llevar a cabo el enlace.";
223 }
```

may 13, 13 21:40 **common_protocolo.h** Page 1/1

```
//
//  common_protocolo.h
//
//  Cabecera con constantes que especifican el protocolo de mensajes a
//  ser utilizados por el cliente y el servidor.
//


#ifndef PROTOCOLO_H
#define PROTOCOLO_H




/* ************************************************************************
 * PROTOCOLO DE INSTRUCCIONES
 * ************************************************************************/


// Constantes para los identificadores de instrucciones enviadas por el
// cliente
const std::string C_GET_JOB_PART = "GET-JOB-PART";
const std::string C_POSSIBLE_KEY = "POSSIBLE-KEY";
const std::string C_JOB_PART_FINISHED = "JOB-PART-FINISHED";

// Constantes para los identificadores de instrucciones enviadas por el
// servidor
const std::string S_JOB_PART = "JOB-PART";
const std::string S_NO_JOB_PART = "NO-JOB-PART";

// Constante para caracter de fin de instrucción
const char FIN_MENSAJE = '\n';


#endif
```

may 13, 13 21:40 **common_mutex.h** Page 1/1

```
//
//  common_mutex.h
//  CLASE MUTEX
//
//  Clase que implementa el tipo de objetos mutex, es decir, objetos con dos
//  estados posibles: tomado y liberado. Este puede ser manipulado desde
//  varios hilos simultáneamente.
//


#ifndef MUTEX_H
#define MUTEX_H


#include <pthread.h>




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


class Mutex {
private:

  pthread_mutex_t mutex;      // Mutex
  pthread_cond_t cond_var;    // Condition variable

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

  // Bloquea la ejecución en un una condition variable hasta que se produzca
  // una señalización.
  void wait();

  // Desbloquea al menos uno de los hilos que están bloqueados en la
  // condition variable.
  void signal();

  // Desbloquea todos los hilos bloqueados actualmente en la condition
  // variable.
  void broadcast();

  // Bloquea el mutex
  void lock();

  // Desbloquea el mutex
  void unlock();

public:

  // Constructor
  Mutex();

  // Destructor
  ~Mutex();

  friend class Lock;
};

#endif
```

may 13, 13 21:40 **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
```

may 13, 13 21:40 **common_mutex.cpp** Page 2/2

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

may 13, 13 21:40 **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  * DECLARACIÓN DE LA CLASE
20  * ******************************************************************************/
21
22
23 class Lock {
24 private:
25
26   Mutex &mutex;       // Mutex
27
28   // Constructor privado
29   Lock(const Lock &c);
30
31 public:
32
33   // Constructor
34   explicit Lock(Mutex &m);
35
36   // Destructor
37   ~Lock();
38
39   // Bloquea el mutex;
40   void lock();
41
42   // Desbloquea el mutex;
43   void unlock();
44
45   // Bloquea la ejecución en un una condition variable hasta que se produzca
46   // una señalización.
47   void wait();
48
49   // Desbloquea al menos uno de los hilos que están bloqueados en la
50   // condition variable.
51   void signal();
52
53   // Desbloquea todos los hilos bloqueados actualmente en la condition
54   // variable.
55   void broadcast();
56 };
57
58 #endif
```

may 13, 13 21:40 **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 "common_mutex.h"
15  #include "common_lock.h"
16
17
18
19
20  /* ******************************************************************************
21   * DECLARACIÓN DE LA CLASE
22   * ******************************************************************************/
23
24
25  template < typename Tipo >
26  class Lista {
27  private:
28
29    std::list< Tipo > lista;      // Lista
30    Mutex m;              // Mutex
31
32  public:
33
34    // Constructor
35    Lista();
36
37    // Destructor
38    ~Lista();
39
40    // Inserta un nuevo elemento al final de la lista.
41    // PRE: 'dato' es el dato a insertar.
42    void insertarUltimo(Tipo dato);
43
44    // Devuelve un puntero al primer elemento
45    Tipo verPrimero();
46
47    // Elimina el primer elemento de la lista.
48    // POST: se destruyó el elemento removido.
49    void eliminarPrimero();
50
51    // Elimina de la lista todos los elementos iguales al valor especificado.
52    // POST: se llama al destructor de estos elementos.
53    void eliminar(Tipo valor);
54
55    // Devuelve la cantidad de elementos contenidos en la lista.
56    size_t tamanio();
57
58    // Verifica si una lista se encuentra vacía.
59    // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
60    // caso contrario.
61    bool estaVacia();
62  };
63
64
65
66
```



```
67  /* ******************************************************************************
68   * DEFINICIÓN DE LA CLASE
69   * ******************************************************************************/
70
71
72  // Constructor
73  template <typename Tipo >
74  Lista< Tipo >::Lista() { }
75
76
77  // Destructor
78  template <typename Tipo >
79  Lista< Tipo >::~Lista() { }
80
81
82  // Inserta un nuevo elemento al final de la lista.
83  // PRE: 'dato' es el dato a insertar.
84  template <typename Tipo >
85  void Lista< Tipo >::insertarUltimo(Tipo dato) {
86    Lock l(m);
87    this→lista.push_back(dato);
88  }
89
90
91  // Devuelve un puntero al primer elemento
92  template <typename Tipo >
93  Tipo Lista< Tipo >::verPrimero() {
94    Lock l(m);
95    return this→lista.front();
96  }
97
98
99  // Elimina el primer elemento de la lista.
100 // POST: se destruyó el elemento removido.
101 template <typename Tipo >
102 void Lista< Tipo >::eliminarPrimero() {
103   Lock l(m);
104   this→lista.pop_front();
105 }
106
107
108 // Elimina de la lista todos los elementos iguales al valor especificado.
109 // POST: se llama al destructor de estos elementos.
110 template <typename Tipo >
111 void Lista< Tipo >::eliminar(Tipo valor) {
112   Lock l(m);
113   this→lista.remove(valor);
114 }
115
116
117 // Devuelve la cantidad de elementos contenidos en la lista.
118 template <typename Tipo >
119 size_t Lista< Tipo >::tamanio() {
120   Lock l(m);
121   return this→lista.size();
122 }
123
124
125 // Verifica si una lista se encuentra vacía.
126 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
127 // caso contrario.
128 template <typename Tipo >
129 bool Lista< Tipo >::estaVacia() {
130   Lock l(m);
131   return this→lista.empty();
132 }
```

may 13, 13 21:40 **common_lista.h** Page 3/3

```
133
134
135 #endif
```

may 13, 13 21:40 **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
16
17
18
19 /* ************************************************************************
20  * DECLARACIÓN DE LA CLASE
21  * ************************************************************************/
22
23
24 class Convertir {
25 public:
26
27   // Devuelve el equivalente entero de un caracter hexadecimal
28   static int htoi(char a);
29
30   // Convierte un unsigned int a un string de contenido hexadecimal
31   static std::string uitoh(uint8_t *a, size_t size);
32
33   // Convierte un string de contenido hexadecimal a un unsigned int
34   static uint8_t* htoui(std::string& s);
35
36   // Convierte un string en un integer
37   static int stoi(const std::string& s);
38
39   // Convierte un integer en un string
40   static std::string itos(const int i);
41 };
42
43
44 #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::string hexa;
32
33   for(unsigned int i = 0; i < size; i++) {
34     std::stringstream stream;
35     stream << std::uppercase << std::setfill('0') << std::setw(2) <<
36       std::hex << int(a[i]);
37     std::string result(stream.str());
38
39     hexa.append(result);
40   }
41
42   return hexa;
43 }
44
45
46 // Convierte un string de contenido hexadecimal a un unsigned int
47 uint8_t* Convertir::htoui(std::string& s) {
48   uint8_t *a = new uint8_t[s.size() / 2];
49   int j = 0;
50
51   for(unsigned int i = 0; i < s.size(); i += 2) {
52     uint8_t pri = Convertir::htoi(s[i]);
53     uint8_t seg = Convertir::htoi(s[i+1]);
54
55     a[j] = pri * 16 + seg;
56     j++;
57   }
58
59   return a;
60 }
61
62
63 // Convierte un string en un integer
64 int Convertir::stoi(const std::string& s) {
65   int i;
66   std::stringstream ss(s);
```



```
67   ss >> i;
68   return i;
69 }
70
71
72 // Convierte un integer en un string
73 std::string Convertir::itos(const int i) {
74     std::ostringstream s;
75     s << i;
76     return s.str();
77 }
```

may 13, 13 21:40 **common_comunicador.h** Page 1/1

```
//
//  common_comunicador.h
//  CLASE COMUNICADOR
//
//  Clase que implementa la interfaz de comunicación entre servidor y clientes.
//


#ifndef COMUNICADOR_H
#define COMUNICADOR_H



#include <string>
#include "common_socket.h"
#include "common_protocolo.h"




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


class Comunicador {
private:

  Socket *socket;       // Socket de comunicación

public:

  // Constructor
  // PRE: 'socket' es un socket por el que se desea hacer el envío y
  // transmisión de mensajes
  explicit Comunicador(Socket *socket);

  // Emite una instrucción.
  // PRE: 'instruccion' es una cadena que identifica la instrucción a emitir;
  // 'args' son los argumentos de dicha instrucción separadas entre si por
  // un espacio.
  // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
  // de error.
  int emitir(const std::string& instruccion, const std::string& args);

  // Emite un mensaje.
  // PRE: 'msg' es el mensaje que se desea enviar.
  // POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
  // de error.
  int emitir(const std::string& msg);

  // Recibe una instrucción.
  // POST: se almacenó la instrucción recibida en 'instruccion' y los
  // argumentos en args, los cuales se encuentran separados entre si por un
  // espacio. De producirse un error, 'instruccion' y 'args' queda vacíos y
  // se retorna -1. En caso de exito se devuelve 0.
  int recibir(std::string& instruccion, std::string& args);
};

#endif
```

may 13, 13 21:40 **common_comunicador.cpp** Page 1/2

```
//
//  common_comunicador.h
//  CLASE COMUNICADOR
//
//  Clase que implementa la interfaz de comunicación entre servidor y clientes.
//


#include "common_comunicador.h"
#include <sstream>




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


// Constructor
// PRE: 'socket' es un socket por el que se desea hacer el envío y
// transmisión de mensajes
Comunicador::Comunicador(Socket *socket) : socket(socket) { }


// Emite una instrucción y sus argumentos.
// PRE: 'instruccion' es una cadena que identifica la instrucción a emitir;
// 'args' son los argumentos de dicha instrucción separadas entre si por
// un espacio.
// POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
// de error.
int Comunicador::emitir(const std::string& instruccion,
  const std::string& args) {
  // Armamos mensaje a enviar
  std::string msg = instruccion + " " + args + FIN_MENSAJE;

  // Enviamos el mensaje
  return this->socket->enviar(msg.c_str(), msg.size());
}


// Emite un mensaje.
// PRE: 'msg' es el mensaje que se desea enviar.
// POST: devuelve 0 si se ha realizado el envio correctamente o -1 en caso
// de error.
int Comunicador::emitir(const std::string& msg) {
  // Armamos mensaje a enviar
  std::string msg_n = msg + FIN_MENSAJE;

  // Enviamos el mensaje
  return this->socket->enviar(msg_n.c_str(), msg_n.size());
}


// Recibe una instrucción.
// POST: se almacenó la instrucción recibida en 'instruccion' y los
// argumentos en args, los cuales se encuentran separados entre si por un
// espacio. De producirse un error, 'instruccion' y 'args' queda vacíos y
// se retorna -1. En caso de exito se devuelve 0.
int Comunicador::recibir(std::string& instruccion, std::string& args) {
  // Variable auxiliar para armar mensaje
  std::stringstream msg_in;
  // Limpiamos argumentos que recibiran datos
  instruccion = "";
  args = "";

```

may 13, 13 21:40 **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[1];
71
72      // Si se produce un error, devolvemos una instrucción vacía
73      if(this→socket→recibir(bufout, 1) ≡ −1) return −1;
74
75      // Si se recibió el caractér de fin de mensaje, salimos
76      if(bufout[0] ≡ FIN_MENSAJE) break;
77
78      // Agregamos el caractér a los datos ya recibidos
79      msg_in << bufout[0];
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  }
```

may 13, 13 21:40 **common_codigo_draka.h** Page 1/1

```
1   //
2   //  common_codigo_draka.h
3   //  LIBRERIA CODIGODRAKA
4   //
5   //  Librería de funciones relacionadas al código Draka.
6   //
7
8
9   #ifndef CODIGO_DRAKA_H
10  #define CODIGO_DRAKA_H
11
12
13  #include <string>
14  #include <stdint.h>
15
16
17
18  class CodigoDraka {
19  public:
20
21    // Código utilizado por los Draka para encriptar y desencriptar mensajes.
22    static void ed(uint8_t *data, size_t ndata, const uint8_t *key,
23      size_t nkey);
24
25    // Función que prueba una clave en el código Draka y corrobora si, al
26    // desencriptar el mensaje encriptado, da como resultado texto ASCII válido
27    // PRE: 'dato' es un puntero al mensaje encriptado; 'lenDato' es el
28    // tamanio en bytes de 'dato'; 'clave' es la clave a probar.
29    // POST: devuelve true si la prueba da como resultado texto ASCII válido o
30    // false en su defecto.
31    static bool probarClave(uint8_t *dato, size_t lenDato,
32      const std::string& clave);
33  };
34
35
36  #endif
```

may 13, 13 21:40 **common_codigo_draka.cpp** Page 1/1

```
//
//  common_codigo_draka.h
//  LIBRERIA CODIGODRAKA
//
//  Librería de funciones relacionadas al código Draka.
//


#include "common_codigo_draka.h"



// Código utilizado por los Draka para encriptar y desencriptar mensajes.
void CodigoDraka::ed(uint8_t *data, size_t ndata, const uint8_t *key,
  size_t nkey) {
  uint8_t i = 0, j = 0, s[256];
  do {
    s[i] = i;
  } while (++i);

  do {
    j += s[i] + key[i % nkey];
    i - j ^ (s[i] ^= s[j], s[j] ^= s[i], s[i] ^= s[j]);
  } while (++i);

  j = 0;

  while(ndata--) {
    i++; j += s[i];
    i - j ^ (s[i] ^= s[j], s[j] ^= s[i], s[i] ^= s[j]);
    *data++ ^= s[(s[i] + s[j]) & 0xff];
  }
}


// Función que prueba una clave en el código Draka y corrobora si, al
// desencriptar el mensaje encriptado, da como resultado texto ASCII válido
// PRE: 'dato' es un puntero al mensaje encriptado; 'lenDato' es el
// tamanio en bytes de 'dato'; 'clave' es la clave a probar.
// POST: devuelve true si la prueba da como resultado texto ASCII válido o
// false en su defecto.
bool CodigoDraka::probarClave(uint8_t *dato, size_t lenDato,
  const std::string& clave) {
  size_t lenClave = clave.size();

  CodigoDraka::ed(dato, lenDato, (const uint8_t *)(clave.c_str()), lenClave);

  for(size_t i = 0; i < lenDato; i++)
    if(dato[i] > 127)
      return false;

  return true;
}
```

may 13, 13 21:40 **client_main.cpp** Page 1/2

```
//
//  EL CÓDIGO DRAKA
//  Programa principal del CLIENTE
//
//  ***************************************************************************
//
//  Facultad de Ingeniería - UBA
//  75.42 Taller de Programación I
//  Trabajo Práctico N°4
//
//  ALUMNO: Federico Martín Rossi
//  PADRÓN: 92086
//  EMAIL:  federicomrossi@gmail.com
//
//  ***************************************************************************
//
//  Programa cliente que permite conectarse a un servidor para recibir un
//  mensaje encriptado, en hexadecimal, junto con un rango de claves que
//  deberán ser probadas por fuerza bruta de manera de poder recuperar el
//  texto. El cliente notificará al servidor de aquellas claves que resultan
//  correctas al ser probadas.
//
//
//  FORMA DE USO
//  ============
//
//  Deberá ejecutarse el programa en la línea de comandos de la siguiente
//  manera:
//
//    # ./client [NOMBRE-HOST] [PUERTO]
//
//  donde,
//
//    NOMBRE-HOST: es el nombre del host donde está el servidor;
//    PUERTO: es el puerto donde el servidor está escuchando
//
//



#include <iostream>
#include <string>
#include <sstream>
#include "client_cliente.h"
#include "common_convertir.h"



/* ***************************************************************************
 * PROGRAMA PRINCIPAL
 * ***************************************************************************/


int main(int argc, char* argv[]) {
  // Corroboramos cantidad de argumentos
  if(argc ≠ 2) {
    std::cerr << "ERROR: cantidad incorrecta de argumentos." << std::endl;
    return 1;
  }

  // Datos para la conexión
  int puerto;
  std::string nombreHost;

  // Variables auxiliares
```

may 13, 13 21:40 **client_main.cpp** Page 2/2

```
67    std::string sPuerto;
68    std::stringstream argumento(argv[1]);
69
70    // Obtenemos dirección IP o nombre del host
71    getline(argumento, nombreHost, ':');
72
73    // Obtenemos el puerto
74    getline(argumento, sPuerto, ':');
75    puerto = Convertir::stoi(sPuerto);
76
77    // Creamos el cliente
78    Cliente cliente(nombreHost, puerto);
79
80    // Iniciamos su ejecución
81    cliente.ejecutar();
82
83    return 0;
84  }
```

may 13, 13 21:40 **client_cliente.h** Page 1/1

```
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
13
14
15
16  /* ******************************************************************************
17   * DECLARACIÓN DE LA CLASE
18   * ******************************************************************************/
19
20
21  class Cliente {
22  private:
23
24    Socket socket;            // Socket con el que se comunica
25    int puerto;             // Puerto de conexión.
26    std::string nombreHost;       // Nombre del host de conexión
27
28    // Recibe un mensaje entrante
29    // POST: devuelve un string con el mensaje recibido
30    std::string recibirMensaje();
31
32    // Prueba una a una las claves en el código Draka y envía al servidor
33    // aquellas claves que pasen la prueba.
34    void procesarClaves(std::string msgEncriptado, int numDig,
35      int claveIni, int claveFin);
36
37  public:
38
39    // Constructor
40    Cliente(std::string nombreHost, int puerto);
41
42    // Destructor
43    ~Cliente();
44
45    // Mantiene la comunicación con el servidor.
46    void ejecutar();
47  };
48
49  #endif
```

```
1   //
2   //  client_cliente.h
3   //  CLASE CLIENTE
4   //
5
6
7   #include <iostream>
8   #include <sstream>
9   #include "common_convertir.h"
10  #include "common_codigo_draka.h"
11  #include "common_comunicador.h"
12  #include "client_cliente.h"
13
14
15
16
17  /* ************************************************************************
18   * DEFINICIÓN DE LA CLASE
19   * ************************************************************************/
20
21
22  // Constructor
23  Cliente::Cliente(std::string nombreHost, int puerto) :
24    puerto(puerto), nombreHost(nombreHost) { }
25
26
27  // Destructor
28  Cliente::~Cliente() {
29    this→socket.cerrar();
30  }
31
32
33  // Mantiene la comunicación con el servidor.
34  void Cliente::ejecutar() {
35    // Creamos socket
36    this→socket.crear();
37
38    try {
39      // Conectamos el socket
40      this→socket.conectar(nombreHost, puerto);
41    }
42    catch(char const * e) {
43      std::cerr << e << std::endl;
44      return;
45    }
46
47    // Creamos el comunicador para enviar y recibir mensajes
48    Comunicador comunicador(&this→socket);
49
50    // Enviamos petición de parte de trabajo
51    if(comunicador.emitir(C_GET_JOB_PART) ≡ −1) return;
52
53    // Variables de procesamiento
54    std::string instruccion;
55    std::string args;
56
57    // Recibimos respuesta del servidor
58    if(comunicador.recibir(instruccion, args) ≡ −1) return;
59
60    // Caso en que no hay trabajo para realizar
61    if(instruccion ≡ S_NO_JOB_PART) {
62      // Desconectamos el socket y salimos
63      this→socket.cerrar();
64      return;
65    }
66    else if (instruccion ≡ S_JOB_PART) {
```



```
67      // Variables auxiliares para datos
68      std::string msgEncriptado, numParte;
69      int numDig, claveIni, claveFin;
70      std::stringstream args_stream(args);
71
72      // Parseamos y obtenemos datos del argumento
73      args_stream >> msgEncriptado >> numParte >> numDig >> claveIni
74        >> claveFin;
75
76      // Probamos el rango de claves indicado por el servidor
77      procesarClaves(msgEncriptado, numDig, claveIni, claveFin);
78
79      // Avisamos al servidor la finalización del trabajo
80      if(comunicador.emitir(C_JOB_PART_FINISHED, numParte) ≡ −1) return;
81    }
82    else
83      std::cerr << "Mensaje inválido del servidor" << std::endl;
84
85    // Desconectamos el socket
86    this→socket.cerrar();
87  }
88
89
90  // Prueba una a una las claves en el código Draka y envía al servidor
91  // aquellas claves que pasen la prueba.
92  void Cliente::procesarClaves(std::string msgEncriptado, int numDig,
93    int claveIni, int claveFin) {
94    // Creamos el comunicador para enviar y recibir mensajes
95    Comunicador comunicador(&this→socket);
96
97    // Iteramos hasta procesar todo el rango de claves
98    for(int i = claveIni; i ≤ claveFin; i++) {
99      // Convertimos clave en string
100     std::string clave(Convertir::itos(i));
101
102     // Convertimos el mensaje encriptado en el tipo necesario
103     uint8_t *uintMsgEncriptado = Convertir::htoui(msgEncriptado);
104     size_t len = msgEncriptado.size() / 2;
105
106     // Probamos la clave
107     if (CodigoDraka::probarClave(uintMsgEncriptado, len, clave))
108       // Enviamos mensaje de posible clave si da positiva la prueba
109       if(comunicador.emitir(C_POSSIBLE_KEY, clave) ≡ −1) return;
110
111     delete[] uintMsgEncriptado;
112   }
113 }
```

may 13, 13 21:40 **Table of Content** Page 1/1

Table of Contents