

may 10, 19 13:08

server\_thread.h

Page 1/1

```

1  #ifndef SERVER_THREAD_H_
2  #define SERVER_THREAD_H_
3  #include <thread>
4
5  class Thread {
6      private:
7          std::thread thread;
8          bool is_dead;
9
10     public:
11         Thread();
12         void start();
13         void join();
14         virtual void run() = 0;
15         virtual void stop() = 0;
16         virtual bool isDead() = 0;
17         Thread(const Thread&) = delete;
18         Thread& operator=(const Thread&) = delete;
19         virtual ~Thread();
20         Thread(Thread^ other);
21         Thread& operator=(Thread^ other);
22     };
23
24 #endif

```

may 10, 19 13:08

server\_thread.cpp

Page 1/1

```

1  #include <iostream>
2  #include "server_thread.h"
3  Thread::Thread() {}
4
5  void Thread::start() {
6      this->thread = std::thread(&Thread::run, this);
7  }
8
9
10 void Thread::join() {
11     this->thread.join();
12 }
13
14 Thread::~Thread() {
15     // std::cout << "Destruyendo Thread!" << std::endl;
16 }
17
18 Thread::Thread(Thread^ other) {
19     this->thread = std::move(other.thread);
20 }
21

```

may 10, 19 13:08

## server\_revoke\_client\_processor.h

Page 1/1

```

1  #ifndef SERVER_REVOKE_CLIENT_PROCESSOR_H
2  #define SERVER_REVOKE_CLIENT_PROCESSOR_H
3  #include <stdint.h>
4  #include <string>
5  #include "common_socket.h"
6  #include "common_key.h"
7  #include "server_index.h"
8  #include "common_protocol.h"
9  #include "server_client_processor.h"
10
11 class RevokeClientProcessor : public ClientProcessor {
12     private:
13         Protocol protocol;
14         Index& index;
15         Key& server_key;
16         bool is_dead;
17
18     public:
19         RevokeClientProcessor(Protocol& _protocol, Index& _index, Key& _key);
20
21         ~RevokeClientProcessor();
22         /*
23          * Envia a traves del socket recibido por parametro una solucitud con
24          * formato:
25          *
26          * <comando>          1 byte con valor 0.
27          * <subject__size>    4 bytes big endian sin signo
28          * <subject>         String sin âM-^@M-^X\0âM-^@M-^Y
29          * <modul>           2 bytes big endian sin signo
30          * <exponent>        1 byte
31          * <date__size>      4 bytes big endian sin signo
32          * <date_from>       String sin âM-^@M-^X\0âM-^@M-^Y
33          * <date__size>      4 bytes big endian sin signo
34          * <dat_to>          String sin âM-^@M-^X\0âM-^@M-^Y
35          */
36         virtual void run() override;
37         //bool checkCertificate(Socket& protocol);
38         virtual void stop() override;
39         virtual bool isDead() override;
40     };
41
42 #endif

```

may 10, 19 13:08

## server\_revoke\_client\_processor.cpp

Page 1/2

```

1  #include <iostream>
2  #include <string>
3  #include "server_revoke_client_processor.h"
4  #include "server_inexisting_certificate.h"
5  #include "common_certificate.h"
6  #include "common_rsa.h"
7  #include "common_hash.h"
8  #define HASH_ERROR_MSSG 2
9  #define HASH_ERROR 1
10 #define INVALID_CERTIFICATE_MSSG 1
11 #define INVALID_CERTIFICATE 1
12 #define OK 0
13 #define OK_MSSG 0
14
15
16
17 RevokeClientProcessor::RevokeClientProcessor(Protocol& _protocol,\
18                                             Index& _index, Key& _key):
19     index(_index),
20     server_key(_key),
21     is_dead(false) {
22     protocol = std::move(_protocol);
23 }
24
25 RevokeClientProcessor::~RevokeClientProcessor() {}
26
27 void RevokeClientProcessor::run() {
28     Certificate certificate;
29     certificate.receive(protocol);
30     uint32_t encryption = 0;
31     this->protocol.receive(encryption);
32     uint8_t answer;
33     Key client_key = index.find(certificate);
34
35     try {
36         this->index.erase(certificate);
37     }
38     catch(InexistingCertificate) {
39         answer = INVALID_CERTIFICATE_MSSG;
40         try {
41             this->protocol.send(answer);
42         }
43         catch(std::runtime_error) {
44             throw std::runtime_error(\
45 "Error, client could not be notified that there was a certificate error" );
46         }
47         return;
48     }
49     Rsa rsa(client_key, server_key);
50     uint32_t desencryption = rsa.privateDesencryption(encryption);
51     uint32_t client_hash = rsa.publicDesencryption(desencryption);
52     std::string formal_certificate = certificate.toString();
53     Hash hash(formal_certificate);
54     uint32_t my_hash = hash();
55     if (my_hash != client_hash) {
56         answer = HASH_ERROR_MSSG;
57         this->index.putBack(certificate, client_key);
58         try {
59             this->index.putBack(certificate, client_key);
60             this->protocol.send(answer);
61         }
62         catch(std::runtime_error) {
63             throw std::runtime_error(\
64 "Error, the client could not be notified that there was a hash error" );
65         }
66         catch(...) {}

```

may 10, 19 13:08

server\_revoke\_client\_processor.cpp

Page 2/2

```

67         } __throw_exception_again;
68     }
69     return;
70 }
71 //index.erase(certificate);
72 answer = OK_MSSG;
73 try {
74     this->protocol.send(answer);
75 }
76 catch(std::runtime_error) {
77     throw std::runtime_error(\
78         "Error, the client could not be notified that there was no error" );
79 }
80 is_dead = true;
81 return;
82 }
83
84 bool RevokeClientProcessor::isDead() {
85     return is_dead;
86 }
87
88 void RevokeClientProcessor::stop() {
89     this->protocol.stop();
90 }
91 }

```

may 10, 19 13:08

server\_new\_client\_processor.h

Page 1/1

```

1  #ifndef SERVER_NEW_CLIENT_PROCESSOR_H
2  #define SERVER_NEW_CLIENT_PROCESSOR_H
3  #include <stdint.h>
4  #include <string>
5  #include "server_client_processor.h"
6  #include "common_socket.h"
7  #include "common_key.h"
8  #include "server_index.h"
9
10 class New: public ClientProcessor {
11     private:
12         Protocol protocol;
13         Index& index;
14         Key& server_key;
15         std::string subject;
16         Key client_key;
17         std::string date_from;
18         std::string date_to;
19         bool is_dead;
20         void receiveInfo();
21         std::string createCertificate();
22         bool checkCertificate();
23
24     public:
25         /*
26          * Recibe los dos archivos necesarios para solicitar un nuevo aplicante
27          */
28         New(Protocol& protocol, Index& _index, Key& key);
29
30         ~New();
31         /*
32          * Envia a traves del socket recibido por parametro una solucitud con
33          * formato:
34          *
35          * <comando>          1 byte con valor 0.
36          * <subject__size>    4 bytes big endian sin signo
37          * <subject>          String sin âM-^@M-^X\0âM-^@M-^Y
38          * <modul>            2 bytes big endian sin signo
39          * <exponent>        1 byte
40          * <date__size>       4 bytes big endian sin signo
41          * <date_from>       String sin âM-^@M-^X\0âM-^@M-^Y
42          * <date__size>       4 bytes big endian sin signo
43          * <dat_to>          String sin âM-^@M-^X\0âM-^@M-^Y
44          */
45         virtual void run() override;
46         virtual void stop() override;
47         virtual bool isDead() override;
48     };
49
50 #endif

```

may 10, 19 13:08

server\_new\_client\_processor.cpp

Page 1/3

```

1  #include <string>
2  #include <iostream>
3  #include "server_new_client_processor.h"
4  #include "server_inexisting_certificate.h"
5  #include "server_existing_certificate.h"
6  #include "common_certificate.h"
7  #include "common_hash.h"
8  #include "common_rsa.h"
9  #define ENCRYPTION_SIZE 4
10 #define HASH_ERROR 1
11 #define CERTIFICATE_ERROR 0
12 #define CERTIFICATE_OK 1
13 #define CERT_STATUS_SIZE 1
14 #define ERROR_CODE 1
15 #define HASH_STATUS_SIZE 1
16
17 New::New(Protocol& _protocol, \
18         Index& _index, Key& _key) :
19     index(_index),
20     server_key(_key),
21     is_dead(false) {
22     this->protocol = std::move(_protocol);
23 }
24
25 New::~New() {}
26
27 void New::receiveInfo() {
28     protocol.receive(this->subject);
29     this->client_key.receive(protocol);
30
31     protocol.receive(this->date_from);
32
33     protocol.receive(this->date_to);
34     if (subject.size() == 0) {
35         throw std::runtime_error("");
36     }
37 }
38
39 std::string New::createCertificate() {
40     std::string subj = this->subject;
41     Certificate certificate(subj, this->date_from, this->date_to, \
42         this->client_key);
43
44     uint8_t answer = CERTIFICATE_OK;
45     try {
46         this->index.save(certificate);
47     }
48     catch(ExistingCertificate) { //ExistingCertificate
49         answer = CERTIFICATE_ERROR;
50         try {
51             protocol.send(answer);
52         }
53         catch(std::runtime_error) {
54             throw std::runtime_error(
55                 "Error, client could not be notified that there was a certificate error" );
56         }
57         __throw_exception_again;
58     }
59     try {
60         protocol.send(answer);
61     }
62     catch(std::runtime_error) {
63         throw std::runtime_error(\
64             "Error, the client could not be notified that there was no error" );
65     }
66 }

```

may 10, 19 13:08

server\_new\_client\_processor.cpp

Page 2/3

```

67     std::string result = certificate.toString();
68     try {
69         certificate.send(protocol);
70     }
71     catch(std::runtime_error) {
72         throw std::runtime_error(\
73             "Error sending the certificate while processing new client" );
74     }
75     return result;
76 }
77
78
79 uint32_t encrypt(Key client_key, Key server_key, uint32_t hash) {
80     Rsa rsa(client_key, server_key);
81     uint32_t encryption = rsa.privateEncryption(hash);
82     encryption = rsa.publicEncryption(encryption);
83     return encryption;
84 }
85
86 void New::run() {
87     try {
88         this->receiveInfo();
89     }
90     catch (...) {
91         this->is_dead = true;
92         return;
93     }
94
95     std::string formal_certificate;
96     try {
97         formal_certificate = this->createCertificate();
98     } catch(ExistingCertificate) {
99         return;
100     }
101
102
103     Hash hash(formal_certificate);
104     uint32_t hashed_certificate = hash();
105     uint32_t encryption = encrypt(this->client_key, this->server_key, \
106         hashed_certificate);
107     try {
108         protocol.send(encryption);
109     }
110     catch(std::runtime_error) {
111         throw std::runtime_error(\
112             "Error sending encrypted hash while processing new client" );
113     }
114
115     uint8_t hash_status = 0;
116     protocol.receive(hash_status);
117     if (hash_status == HASH_ERROR) {
118         try {
119             index.erase(this->subject);
120         }
121         catch(InexistingCertificate) {
122             throw std::runtime_error("Error erasing certificate" );
123         }
124     }
125     this->is_dead = true;
126     return;
127 }
128
129 bool New::isDead() {
130     return this->is_dead;
131 }
132

```

may 10, 19 13:08

server\_new\_client\_processor.cpp

Page 3/3

```
133 void New::stop() {  
134     this->protocol.stop();  
135 }
```

may 10, 19 13:08

server\_inexisting\_certificate.h

Page 1/1

```
1  #ifndef SERVER_INEXISTING_CERTIFICATE_H  
2  #define SERVER_INEXISTING_CERTIFICATE_H  
3  #include <exception>  
4  #include <string>  
5  
6  class InexistingCertificate: public std::exception {  
7  protected:  
8      std::string msg;  
9  public:  
10     explicit InexistingCertificate(const char* message);  
11  
12     explicit InexistingCertificate(const std::string& message);  
13  
14     virtual ~InexistingCertificate() throw();  
15  
16     virtual const char* what() const throw();  
17 };  
18  
19 #endif
```

may 10, 19 13:08

server\_inexisting\_certificate.cpp

Page 1/1

```

1 #include "server_inexisting_certificate.h"
2 #include <string>
3
4 InexistingCertificate::InexistingCertificate(const char* message):
5     msg(message) {}
6
7 InexistingCertificate::InexistingCertificate(const std::string& message):
8     msg(message) {}
9
10 InexistingCertificate::~InexistingCertificate() throw(){}
11
12 const char* InexistingCertificate::what() const throw(){
13     return msg.c_str();
14 }

```

may 10, 19 13:08

server\_index.h

Page 1/1

```

1 #ifndef SERVER_INDEX_H
2 #define SERVER_INDEX_H
3 #include <string>
4 #include <map>
5 #include <mutex>
6 #include "common_key.h"
7 #include "common_certificate.h"
8
9 class Index {
10     private:
11         std::string& filename;
12         std::map<std::string, Key> certificates;
13         void parseLine(std::string& line);
14         std::mutex mutex;
15         uint32_t serial_number;
16
17
18     public:
19         Index();
20         void write();
21         void save(Certificate& certificate);
22         //void increaseSerialNumber();
23
24         /*
25          * Vuelve a almacenar el certificado pasado por parametro.
26          * Con client_key valor.
27          */
28         void putBack(Certificate& certificate, Key& client_key);
29
30
31         Key find(Certificate& cartificate);
32
33         /* Borra el certificado
34          * Si el certificado no existe lanza una exepcion de tipo
35          * InexistentCertificate
36          */
37         void erase(Certificate& certificate);
38         void erase(std::string& str);
39
40         explicit Index(std::string& filename);
41         ~Index();
42
43
44         bool has(std::string& str);
45         bool has(Certificate& certificate);
46     };
47
48 #endif

```

may 10, 19 13:08

server\_index.cpp

Page 1/3

```

1  #include <fstream>
2  #include <sstream>
3  #include <iostream>
4  #include <string>
5  #include <queue>
6  #include <map>
7  #include "common_key.h"
8  #include "server_index.h"
9  #include "server_inexisting_certificate.h"
10 #include "server_existing_certificate.h"
11
12 Index::~Index() {}
13
14 Index::Index(std::string& _filename) : filename(_filename) {
15     std::ifstream file;
16     file.open(filename);
17     if (!file.good()) {
18         throw std::runtime_error("Error with index file");
19     }
20     std::string line;
21     std::getline(file, line, '\n');
22
23     int aux;
24     if (line.size() != 0) { //if line empty aux = random
25         std::istringstream sn(line);
26         sn >> aux;
27         this->serial_number = (uint32_t) aux;
28     } else {
29         this->serial_number = 1;
30     }
31     while (std::getline(file, line, '\n')) {
32         this->parseLine(line);
33     }
34 }
35
36 void split(std::string& str, char c, std::queue<std::string>& container) {
37     std::string buff("");
38     int i = 0;
39     while (str[i]) {
40         if (str[i] == c ^ buff != "") {
41             container.push(buff);
42             buff = "";
43         } else {
44             buff += str[i];
45         }
46         i++;
47     }
48     if (buff != "") {
49         container.push(buff);
50     }
51 }
52
53 void Index::parseLine(std::string& line) {
54     std::queue<std::string> container;
55     split(line, ';', container);
56
57     std::string subject = container.front();
58     container.pop();
59     std::string str_key = container.front();
60     container.pop();
61
62     split(str_key, ' ', container);
63     std::string str1 = container.front();
64     container.pop();

```

may 10, 19 13:08

server\_index.cpp

Page 2/3

```

67     std::string str2 = container.front();
68     container.pop();
69     Key key(str1, str2);
70     this->certificates.insert({subject, key});
71 }
72
73 bool Index::has(std::string& str) {
74     //std::unique_lock<std::mutex> lock(this->mutex);
75     std::map<std::string, Key>::iterator it = this->certificates.find(str);
76     bool result = it != this->certificates.end();
77     return result;
78 }
79
80 Key Index::find(Certificate& certificate) {
81     std::unique_lock<std::mutex> lock(this->mutex);
82     std::map<std::string, Key>::iterator it = \
83         this->certificates.find(certificate.getSubject());
84     return it->second;
85 }
86
87 bool Index::has(Certificate& certificate) {
88     std::string sbj = certificate.getSubject();
89     return this->has(sbj);
90 }
91
92 void Index::save(Certificate& certificate) {
93     std::unique_lock<std::mutex> lock(this->mutex);
94     if (this->has(certificate)) {
95         throw ExistingCertificate("Error saving certificate");
96     }
97     this->certificates.insert({certificate.getSubject(), certificate.getKey()});
98     certificate.addSerial(this->serial_number);
99     this->serial_number++;
100 }
101
102 void Index::putBack(Certificate& certificate, Key& client_key) {
103     std::unique_lock<std::mutex> lock(this->mutex);
104     this->certificates.insert({certificate.getSubject(), client_key});
105 }
106
107 void Index::erase(std::string& str) {
108     std::unique_lock<std::mutex> lock(this->mutex);
109     if (!this->has(str)) {
110         throw InexistingCertificate("Error erasing certificate");
111     }
112     this->certificates.erase(str);
113 }
114
115 void Index::erase(Certificate& certificate) {
116     std::string subj = certificate.getSubject();
117     try {
118         this->erase(subj);
119     }
120     catch(InexistingCertificate) {
121         __throw_exception_again;
122     }
123 }
124
125 void Index::write() {
126     std::ofstream file;
127     file.open(this->filename, std::ofstream::out | std::ofstream::trunc);
128
129     if (!file.is_open()) {
130         //exc
131     }
132 }

```

may 10, 19 13:08

server\_index.cpp

Page 3/3

```

133     file << std::to_string(this->serial_number) << '\n';
134     std::map<std::string, Key>::iterator it = this->certificates.begin();
135     for (; it != this->certificates.end(); ++it) {
136         file << it->first << ";" << it->second << '\n';
137     }
138 }

```

may 10, 19 13:08

server\_existing\_certificate.h

Page 1/1

```

1  #ifndef SERVER_EXISTING_CERTIFICATE_H
2  #define SERVER_EXISTING_CERTIFICATE_H
3  #include <exception>
4  #include <string>
5
6  class ExistingCertificate: public std::exception {
7  protected:
8      std::string msg;
9  public:
10     explicit ExistingCertificate(const char* message);
11
12     explicit ExistingCertificate(const std::string& message);
13
14     virtual ~ExistingCertificate() throw();
15
16     virtual const char* what() const throw();
17 };
18
19 #endif

```



may 10, 19 13:08

server\_existing\_certificate.cpp

Page 1/1

```

1 #include <string>
2 #include "server_existing_certificate.h"
3
4 ExistingCertificate::ExistingCertificate(const char* message):
5     msg(message) {}
6
7 ExistingCertificate::ExistingCertificate(const std::string& message):
8     msg(message) {}
9
10 ExistingCertificate::~ExistingCertificate() throw(){}
11
12 const char* ExistingCertificate::what() const throw(){
13     return msg.c_str();
14 }

```

may 10, 19 13:08

server.cpp

Page 1/1

```

1 #include <string>
2 #include <iostream>
3 #include "server_index.h"
4 #include "server_acceptor.h"
5
6
7 #define ERROR_CODE 1
8 #define COMMAND_SIZE 1
9 #define LEN_SIZE 4
10
11
12
13 int main(int argc, char* argv[]) {
14     try {
15         if (argc != 4) {
16             return ERROR_CODE;
17         }
18
19         std::string claves = std::string(argv[2]);
20         std::string indice = std::string(argv[3]);
21         if (argc != 4) {
22             return 1;
23         }
24
25         Socket skt;
26         skt.connectWithClients(argv[1]);
27
28         std::string index_filename(argv[3]);
29         Index index(index_filename);
30
31
32         std::string key_filename = std::string(argv[2]);
33         Key key(key_filename);
34         Acceptor acceptor(skt, index, key);
35         acceptor.start();
36
37         std::string line;
38         while (std::getline(std::cin, line)) {
39             if (line == "q") {
40                 acceptor.stop();
41                 break;
42             }
43         }
44
45         acceptor.join();
46         index.write();
47
48         return 0;
49     }
50     catch(std::runtime_error &e) {
51         std::cerr << e.what() << std::endl;
52     }
53 }

```

may 10, 19 13:08

server\_client\_processor.h

Page 1/1

```
1  #ifndef SERVER_CLIENT_PROCESSOR_H
2  #define SERVER_CLIENT_PROCESSOR_H
3  #include "server_thread.h"
4
5
6  class ClientProcessor : public Thread {
7      public:
8          ClientProcessor();
9          ~ClientProcessor();
10 };
11
12 #endif
```

may 10, 19 13:08

server\_client\_processor.cpp

Page 1/1

```
1  #include "server_client_processor.h"
2
3  ClientProcessor::ClientProcessor() {}
4  ClientProcessor::~~ClientProcessor() {}
```

may 10, 19 13:08

server\_acceptor.h

Page 1/1

```

1  #ifndef SERVER_ACCEPTOR_H
2  #define SERVER_ACCEPTOR_H
3  #include <string>
4  #include <mutex>
5  #include <thread>
6  #include <iostream>
7  #include <queue>
8  #include <vector>
9  #include "server_thread.h"
10 #include "common_socket.h"
11 #include "server_index.h"
12 #include "server_client_processor.h"
13 // #include "compare_bf.h"
14
15
16 class Acceptor : public Thread {
17     private:
18         Socket skt;
19         Index& index;
20         Key& key;
21         Socket client_skt;
22         bool keep_talking;
23         std::vector<ClientProcessor*> clients;
24
25     public:
26         Acceptor(Socket& _skt, Index& index, Key& key);
27         ~Acceptor();
28         virtual void run() override;
29         virtual void stop() override;
30         virtual bool isDead() override;
31 };
32
33
34
35
36 #endif

```

may 10, 19 13:08

server\_acceptor.cpp

Page 1/2

```

1  #include <vector>
2  #include "server_acceptor.h"
3  #include "common_protocol.h"
4  #include "server_new_client_processor.h"
5  #include "server_revoke_client_processor.h"
6  #include "common_key.h"
7  #include "server_client_processor.h"
8
9  Acceptor::Acceptor(Socket& _skt, Index& _index, Key& _key):
10     index(_index),
11     key(_key),
12     keep_talking(true) {
13     this->skt = std::move(_skt);
14 }
15
16 void Acceptor::run() {
17     try {
18         while (this->keep_talking) {
19             Socket client_skt;
20             try {
21                 client_skt = this->skt.acceptClient();
22             }
23             catch (std::runtime_error &e) {
24                 std::runtime_error(e.what());
25             }
26
27             Protocol protocol(client_skt);
28
29             uint8_t command;
30             protocol.receive(command);
31
32             ClientProcessor* client;
33             if (command == 0) {
34                 client = new New(protocol, index, key);
35             }
36             if (command == 1) {
37                 client = new RevokeClientProcessor(protocol, index, key);
38             }
39             this->clients.push_back(client);
40             client->start();
41             std::vector<ClientProcessor*>::iterator it = this->clients.begin();
42             while (it != this->clients.end()) {
43                 ClientProcessor* client = *it;
44                 if (client->isDead()) {
45                     client->join();
46                     delete client;
47                     this->clients.erase(it);
48                 } else {
49                     ++it;
50                 }
51             }
52         }
53     }
54     catch (std::exception &e) {
55         std::cerr << e.what() << std::endl;
56     }
57 }
58
59 void Acceptor::stop() {
60     this->keep_talking = false;
61     skt.kill();
62 }
63
64 bool Acceptor::isDead() {
65     return !keep_talking;
66 }

```

may 10, 19 13:08

server\_acceptor.cpp

Page 2/2

```

67
68 Acceptor::~Acceptor() {
69     std::vector<ClientProcessor*>::iterator it = this->clients.begin();
70     for (; it != this->clients.end(); ++it) {
71         ClientProcessor* client = *it;
72         client->join();
73         delete client;
74     }
75 }

```

may 10, 19 13:08

common\_socket.h

Page 1/1

```

1  #ifndef COMMON_SOCKET_H
2  #define COMMON_SOCKET_H
3  #include <stdlib.h>
4  #include <stdbool.h>
5
6  class Socket {
7      private:
8          int skt;
9
10     public:
11         Socket();
12         ~Socket();
13
14         explicit Socket(int skt);
15         Socket(Socket^ origin);
16         Socket& operator=(Socket^ origin);
17         Socket(const Socket& origin);
18
19         void connectWithClients(const char* port);
20         Socket acceptClient();
21
22         void connectWithServer(const char* host, const char* port);
23
24
25         void receiveAll(void* buf, size_t len);
26         int receiveSome(void* buf, size_t size);
27         int sendAll(void* buf, size_t size);
28
29         void kill();
30         void setInvalid();
31     };
32
33
34 #endif

```

may 10, 19 13:08

common\_socket.cpp

Page 1/4

```

1  #define _POSIX_C_SOURCE 200112L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include <errno.h>
7  #include <stdbool.h>
8
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include <unistd.h>
13 #include <utility>
14 #include <stdexcept>
15 #include <iostream>
16 #include <string>
17 #include "common_socket.h"
18
19
20 #define MAX_WAITING_CLIENTS 20
21
22 /*****
23  * TODOS LOS
24  * if (s < 0) {
25  *     return
26  * }
27  * TIENEN QUE SER EXCEPCIONES EN UN FUTURO.
28  */
29
30
31
32 Socket::Socket() {
33     int familia_skt = AF_INET;      /* IPv4 (or AF_INET6 for IPv6) */
34     int tipo_skt = SOCK_STREAM;     /* TCP (or SOCK_DGRAM for UDP) */
35     int protocolo_skt = 0;          /* Any protocol */
36     int s = socket(familia_skt, tipo_skt, protocolo_skt);
37
38     if (!s) {
39         throw std::runtime_error("Error creating socket");
40     }
41     this->skt = s;
42 }
43
44 Socket::Socket(int skt) {
45     if (skt == -1) {
46         throw std::runtime_error("Error creating socket");
47     }
48     this->skt = skt;
49 }
50
51
52 Socket& Socket::operator=(Socket& origin) {
53     if (this->skt != -1) {
54         shutdown(this->skt, SHUT_RDWR);
55         close(this->skt);
56     }
57
58     this->skt = origin.skt;
59     origin.skt = -1;
60     return *this;
61 }
62
63 Socket::Socket(Socket& origin): skt(origin.skt) {
64     origin.skt = -1;
65 }
66

```

may 10, 19 13:08

common\_socket.cpp

Page 2/4

```

67
68 void Socket::kill(){
69     if (this->skt != -1) {
70         shutdown(this->skt, SHUT_RDWR);
71         close(this->skt);
72         this->skt = -1;
73     }
74 }
75
76 Socket::~~Socket() {
77     if (this->skt != -1) {
78         shutdown(this->skt, SHUT_RDWR);
79         close(this->skt);
80     }
81 }
82
83
84 void Socket::connectWithClients(const char* port) {
85     struct addrinfo hints;
86     struct addrinfo *results, *ptr;
87     memset(&hints, 0, sizeof(struct addrinfo));
88     hints.ai_family = AF_INET;      /* IPv4 (or AF_INET6 for IPv6) */
89     hints.ai_socktype = SOCK_STREAM; /* TCP (or SOCK_DGRAM for UDP) */
90     hints.ai_flags = AI_PASSIVE;    /* AI_PASSIVE for server */
91
92     int s = 0;
93     s = getaddrinfo(NULL, port, &hints, &results);
94
95     if (s != 0) {
96         throw std::runtime_error(strerror(s));
97     }
98
99     int val = 1;
100    s = setsockopt(this->skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
101    if (s == -1) {
102        throw std::runtime_error("Error at setsockopt");
103    }
104
105    ptr = results;
106    s = bind(this->skt, ptr->ai_addr, ptr->ai_addrlen);
107    if (s == -1) {
108        throw std::runtime_error("Error while binding");
109    }
110    freeaddrinfo(results);
111    s = listen(this->skt, MAX_WAITING_CLIENTS);
112    if (s == -1) {
113        throw std::runtime_error("Error while listening");
114    }
115    return;
116 }
117
118
119 void Socket::connectWithServer(const char* host, const char* port) {
120     struct addrinfo hints;
121     struct addrinfo *result, *ptr;
122     memset(&hints, 0, sizeof(struct addrinfo));
123     hints.ai_family = AF_INET;      /* IPv4 (or AF_INET6 for IPv6) */
124     hints.ai_socktype = SOCK_STREAM; /* TCP (or SOCK_DGRAM for UDP) */
125     hints.ai_flags = 0;              /* None */
126
127     int s = 0;
128     s = getaddrinfo(host, port, &hints, &result);
129
130     bool are_we_connected = false;
131
132     for (ptr = result; ptr != NULL & are_we_connected == false; \

```

may 10, 19 13:08

common\_socket.cpp

Page 3/4

```

133     ptr = ptr->ai_next) {
134     s = connect(this->skt, ptr->ai_addr, ptr->ai_addrlen);
135     are_we_connected = (s != -1);
136 }
137 freeaddrinfo(result);
138
139 if (!are_we_connected) {
140     throw std::runtime_error("Error while connecting with client");
141 }
142 }
143
144
145 void Socket::receiveAll(void* buf, size_t len) {
146     memset(buf, 0, len);
147     size_t received = 0;
148     int status = 0;
149     bool keep_going = true;
150     char* aux = (char*) buf;
151     while (received < len ^ keep_going) {
152         status = this->receiveSome(&aux[received], len - received);
153         //if (status == 0) {
154             //    keep_going = true;
155         //} else
156         if (status < 0) {
157             keep_going = false;
158         } else {
159             received += status;
160         }
161     }
162 }
163
164 int Socket::receiveSome(void* buf, size_t size) {
165     return ::recv(this->skt, buf, size, MSG_NOSIGNAL);
166 }
167
168 int Socket::sendAll(void* buf, size_t size) {
169     int bytes_sent = 0;
170     int s = 0;
171     bool is_the_socket_valid = true;
172     char* aux = (char*) buf;
173
174     while (bytes_sent < (int)size ^ is_the_socket_valid) {
175         s = ::send(this->skt, &aux[bytes_sent], \
176             size-bytes_sent, MSG_NOSIGNAL);
177         if (s <= 0) {
178             std::string error("Error: ");
179             throw std::runtime_error(error + strerror(errno));
180         } else {
181             bytes_sent += s;
182         }
183     }
184     return bytes_sent;
185 }
186
187
188 Socket Socket::acceptClient(){
189     int peerskt = accept(this->skt, NULL, NULL);
190     if (peerskt == -1) {
191         throw std::runtime_error("Error accepting client");
192     }
193     Socket skt(peerskt);
194     return std::move(skt);
195 }
196
197 void Socket::setInvalid() {
198     this->skt = -1;

```

may 10, 19 13:08

common\_socket.cpp

Page 4/4

```

199 }

```

may 10, 19 13:08

common\_rsa.h

Page 1/1

```

1  #ifndef RSA_H
2  #define RSA_H
3  #include <stdint.h>
4  #include "common_key.h"
5
6  /*
7   * Algoritmo de encriptaci3n asim3trico.
8   */
9  class Rsa {
10     private:
11         uint8_t public_exponent;
12         uint16_t public_module;
13         uint8_t private_exponent;
14         uint16_t private_module;
15     public:
16         Rsa(Key& public_key, Key& private_key);
17         Rsa();
18         ~Rsa();
19
20         void set(Key& public_key, Key& private_key);
21
22         /*
23          * Recibe un mensaje (que es un n3mero de 4 bytes sin signo) y lo
24          * encrita en otro entero de 4 bytes sin signo.
25          */
26         uint32_t publicEncryption(const uint32_t hash);
27         uint32_t privateEncryption(const uint32_t hash);
28         uint32_t publicDesencryption(const uint32_t hash);
29         uint32_t privateDesencryption(const uint32_t hash);
30     };
31
32 #endif

```

may 10, 19 13:08

common\_rsa.cpp

Page 1/1

```

1  #include "common_rsa.h"
2  //size in bytes of the message to encrypt
3  #define HASH_SIZE 4
4
5  Rsa::Rsa(Key& public_key, Key& private_key) {
6      this->public_exponent = public_key.getPublicExponent();
7      this->public_module = public_key.getModule();
8      this->private_exponent = private_key.getPrivateExponent();
9      this->private_module = private_key.getModule();
10 }
11
12 Rsa::Rsa(){}
13
14 void Rsa::set(Key& public_key, Key& private_key) {
15     this->public_exponent = public_key.getPublicExponent();
16     this->public_module = public_key.getModule();
17     this->private_exponent = private_key.getPrivateExponent();
18     this->private_module = private_key.getModule();
19 }
20
21 Rsa::~Rsa() {}
22
23 uint32_t encrypt(const uint32_t exponent, const uint32_t module,\
24                 const uint32_t hash) {
25     uint32_t result;
26     uint32_t base;
27
28     uint32_t ret = 0;
29
30     for (uint32_t i = 0; i < HASH_SIZE; ++i) {
31         result = (hash >> (i * 8)) & 0xff;
32         base = result;
33         for (uint32_t j = 1; j < exponent; ++j)
34             result = (result * base) % module;
35         ret = ret + (result << (i * 8));
36     }
37     return ret;
38 }
39
40
41 uint32_t Rsa::publicEncryption(const uint32_t hash) {
42     return encrypt(this->public_exponent, this->public_module, hash);
43 }
44
45 uint32_t Rsa::privateEncryption(const uint32_t hash) {
46     return encrypt(this->private_exponent, this->private_module, hash);
47 }
48
49 uint32_t Rsa::publicDesencryption(const uint32_t hash){
50     return encrypt(this->public_exponent, this->public_module, hash);
51 }
52
53 uint32_t Rsa::privateDesencryption(const uint32_t hash) {
54     return encrypt(this->private_exponent, this->private_module, hash);
55 }
56

```

may 10, 19 13:08

common\_protocol.h

Page 1/1

```

1 #ifndef COMMON_PROTOCOL_H
2 #define COMMON_PROTOCOL_H
3 #include "common_socket.h"
4 #include <string>
5
6 /*
7  * Encapsulaci3n del socket que conoce el protocolo de comunicaci3n pedido
8  */
9 class Protocol {
10     private:
11         Socket skt;
12
13     public:
14         Protocol();
15         explicit Protocol(Socket& _skt);
16         ~Protocol();
17         Protocol(Protocol& origin);
18         Protocol& operator=(Protocol& origin);
19         void receive(uint8_t& n);
20         void receive(uint16_t& n);
21         void receive(uint32_t& n);
22
23         void send(uint8_t& n);
24         void send(uint16_t& n);
25         void send(uint32_t& n);
26
27         int send(std::string& str);
28         int receive(std::string& str);
29         void stop();
30 };
31
32 #endif

```

may 10, 19 13:08

common\_protocol.cpp

Page 1/2

```

1 #include "common_protocol.h"
2 #include <string>
3 #include <iostream>
4 #define UINT8_SIZE 1
5 #define UINT16_SIZE 2
6 #define UINT32_SIZE 4
7
8 Protocol::Protocol(Socket& _skt) {
9     skt = std::move(_skt);
10 }
11
12 Protocol::Protocol() {}
13
14 Protocol::~Protocol() {}
15
16 void Protocol::send(uint8_t& n) {
17     try {
18         this->skt.sendAll(&n, UINT8_SIZE);
19     }
20     catch(std::runtime_error &e) {
21         throw std::runtime_error("Error in protocol while sending");
22     }
23 }
24
25 void Protocol::send(uint16_t& n) {
26     uint16_t aux = htobe16(n);
27     try {
28         this->skt.sendAll(&aux, UINT16_SIZE);
29     }
30     catch(std::runtime_error &e) {
31         throw std::runtime_error("Error in protocol while sending");
32     }
33 }
34
35 void Protocol::send(uint32_t& n) {
36     uint32_t aux = htobe32(n);
37     try {
38         this->skt.sendAll(&aux, UINT32_SIZE);
39     }
40     catch(std::runtime_error &e) {
41         throw std::runtime_error("Error in protocol while sending");
42     }
43 }
44
45 int Protocol::send(std::string& str) {
46     uint32_t len = (uint32_t) str.length();
47     int sent = 0;
48     try {
49         this->send(len);
50         for (size_t i = 0; i < len ^ sent >= 0; ++i){
51             sent = this->skt.sendAll(&str[i], 1);
52         }
53     }
54     catch(std::runtime_error &e) {
55         throw std::runtime_error("Error in protocol while sending");
56     }
57     return sent;
58 }
59
60
61 int Protocol::receive(std::string& str) {
62     uint32_t len = (uint32_t) str.length();
63     this->receive(len);
64     char c;
65     int received = 0;
66     for (size_t i = 0; i < len; ++i){

```



may 10, 19 13:08

## common\_protocol.cpp

Page 2/2

```

67         received += this->skt.receiveSome(&c, 1);
68         str.append(1, c);
69     }
70     return received;
71 }
72
73 void Protocol::receive(uint8_t& n) {
74     this->skt.receiveAll(&n, UINT8_SIZE);
75 }
76
77 void Protocol::receive(uint16_t& n) {
78     uint16_t aux;
79     this->skt.receiveAll(&aux, UINT16_SIZE);
80     n = htobe16(aux);
81 }
82
83
84 void Protocol::receive(uint32_t& n) {
85     uint32_t aux;
86     this->skt.receiveAll(&aux, UINT32_SIZE);
87     n = htobe32(aux);
88 }
89
90 void Protocol::stop() {}
91
92 Protocol& Protocol::operator=(Protocol& origin) {
93     this->skt = std::move(origin.skt);
94     return *this;
95 }
96
97 Protocol::Protocol(Protocol& origin) {
98     this->skt = std::move(origin.skt);
99     origin.skt.setInvalid();
100 }
101

```

may 10, 19 13:08

## common\_key.h

Page 1/1

```

1  #ifndef COMMON_KEY_H
2  #define COMMON_KEY_H
3  #include <stdint.h>
4  #include <string>
5  #include <queue>
6  #include <functional>
7  #include "common_protocol.h"
8
9  class Key {
10     private:
11         uint8_t public_exponent;    //1 byte
12         uint8_t private_exponent;  //1 byte
13         uint16_t module;           //2 bytes
14
15     public:
16         Key();
17         Key(const Key &self);
18         Key(std::string& _public_exponent, std::string& _module);
19         explicit Key(std::string& filename);
20         ~Key();
21         /*
22          * Recibe el nombre de un archivo de tipo
23          * <exp_publico> <exp_privado> <modulo>
24          *
25          * Los distintos campos pueden estar separados por 1 o m  s espacios,
26          *   nicamente se garantiza que estos se encuentran en una misma l  nea.
27          */
28         void set(std::string& filename);
29         void set(std::string& _public_exponent, std::string& _module);
30         /*
31          * Env  a, a tr  ves del socket pasado por parametro:
32          * M  dulo: en formato 2 bytes en big endian sin signo.
33          * Exponente: en 1 byte.
34          */
35         void send(Protocol& protocol);
36         void receive(Protocol& protocol);
37         friend std::ostream& operator<<(std::ostream&, const Key&);
38         /*
39          * Recibe un string y una funci  n que al pasarle el exponente publico y
40          * su tama  o en bytes devuelve en string una representaci  n del mismo
41          *
42          * Se imprime en el string tanto el exponente p  blico como la tranfor-
43          * maci  n del mismo.
44          */
45         void printPublicExponent(std::string& o, \
46                                 std::function<std::string(int, int)> transform);
47         /*
48          * Recibe un string y una funci  n que al pasarle el modulo y su tama  o
49          * en bytes devuelve en string una representaci  n del mismo. Se imprime
50          * en el string tanto el modulo como la transformaci  n del mismo.
51          */
52         void printModule(std::string& o, \
53                          std::function<std::string(int, int)> transform);
54         uint8_t getPublicExponent();
55         uint8_t getPrivateExponent();
56         uint16_t getModule();
57     };
58 #endif

```

may 10, 19 13:08

common\_key.cpp

Page 1/2

```

1  #include <fstream>
2  #include <sstream>
3  #include <iostream>
4  #include <string>
5  #include <algorithm>
6  #include "common_key.h"
7  #define EXPONENT_SIZE 1
8  #define MODULE_SIZE 2
9  #define PUBLIC_EXP_POS 0
10 #define PRIVATE_EXP_POS 1
11 #define MODULE_POS 2
12
13
14 Key::Key(std::string& _public_exponent, std::string& _module) {
15     this->set(_public_exponent, _module);
16 }
17
18 Key::Key() {}
19
20 void Key::set(std::string& _public_exponent, std::string& _module){
21     int aux;
22     std::istringstream pe(_public_exponent);
23     pe >> aux;
24     this->public_exponent = (uint8_t) aux;
25     std::istringstream m(_module);
26     m >> aux;
27     this->module = (uint16_t) aux;
28 }
29
30 Key::Key(std::string& filename) {
31     this->set(filename);
32 }
33
34 void Key::set(std::string& filename) {
35     std::ifstream file;
36     file.open(filename);
37     if (!file.good()) {
38         throw std::runtime_error("Error with file while seting key");
39     }
40     std::string line;
41     int i = 0;
42     int aux;
43     while (std::getline(file, line, ' ')) {
44         if (line.length() == 0) continue;
45         if (i == PUBLIC_EXP_POS) {
46             std::istringstream pbe(line);
47             pbe >> aux;
48             this->public_exponent = (uint8_t) aux;
49         } else if (i == PRIVATE_EXP_POS) {
50             std::istringstream pre(line);
51             pre >> aux;
52             this->private_exponent = (uint8_t) aux;
53         } else if (i == MODULE_POS) {
54             std::istringstream m(line);
55             m >> aux;
56             this->module = (uint16_t) aux;
57         }
58         i++;
59     }
60     if (i == MODULE_POS) {
61         this->module = this->private_exponent;
62     }
63     //file.close();
64 }
65
66 Key::Key(const Key &key) {

```

may 10, 19 13:08

common\_key.cpp

Page 2/2

```

67     this->private_exponent = key.private_exponent;
68     this->public_exponent = key.public_exponent;
69     this->module = key.module;
70 }
71
72 Key::~Key() {}
73
74 void Key::receive(Protocol& protocol) {
75     protocol.receive(this->public_exponent);
76     protocol.receive(this->module);
77 }
78
79 void Key::send(Protocol& protocol) {
80     try {
81         protocol.send(this->public_exponent);
82         protocol.send(this->module);
83     }
84     catch(std::runtime_error) {
85         throw std::runtime_error("Error while sending key");
86     }
87 }
88
89 std::ostream& operator<<(std::ostream& o, const Key& self) {
90     std::string s = std::to_string(self.public_exponent);
91     o << s << ' ';
92     s = std::to_string(self.module);
93     o << s;
94     return o;
95 }
96
97 void Key::printPublicExponent(std::string& o, \
98     std::function<std::string(int, int)> transform) {
99     int exp = (int) this->public_exponent;
100    std::string transformed_exp = transform(exp, EXPONENT_SIZE);
101    o += std::to_string(exp) + ' ' + transformed_exp;
102 }
103
104 void Key::printModule(std::string& o, \
105     std::function<std::string(int, int)> transform) {
106     int module = (int) this->module;
107     std::string transformed_module = transform(module, MODULE_SIZE);
108     o += std::to_string(module) + ' ' + transformed_module;
109 }
110
111
112 uint8_t Key::getPublicExponent() {
113     return this->public_exponent;
114 }
115 uint8_t Key::getPrivateExponent() {
116     return this->private_exponent;
117 }
118 uint16_t Key::getModule() {
119     return this->module;
120 }
121
122 /***** PRINT *****/
123 std::cerr << "\tKEY:\n";
124 fprintf(stderr, "< %d > < %d >\n", this->public_exponent, this->module);
125
126 */

```

may 10, 19 13:08

common\_hash.h

Page 1/1

```

1  #ifndef HASH_H
2  #define HASH_H
3  #include <stdint.h>
4  #include <string>
5
6  /*
7   * Recibe un texto que es cargado de a segmentos y lo hashea.
8   */
9  class Hash {
10     private:
11         uint32_t count;
12
13     public:
14         Hash();
15         explicit Hash(std::string& str);
16         ~Hash();
17         /*
18          * Recibe una cadena y actualiza el resultado de la funciÃ³n de hash.
19          */
20         void load(std::string& str);
21
22         /*
23          * Devuelve el resultado (entero de 2 bytes) de hashear el texto
24          * completo.
25          */
26         uint32_t operator()();
27     };
28
29 #endif

```

may 10, 19 13:08

common\_hash.cpp

Page 1/1

```

1  #include <string>
2  #include <iostream>
3  #include "common_hash.h"
4  #define INITIAL_COUNT 0
5
6  Hash::Hash() {
7      this->count = INITIAL_COUNT;
8  }
9  Hash::~Hash() {}
10
11 Hash::Hash(std::string& str) {
12     this->count = INITIAL_COUNT;
13     this->load(str);
14 }
15
16 void Hash::load(std::string& str) {
17     std::string::iterator it = str.begin();
18     for (; it != str.end(); ++it) {
19         this->count += (uint8_t)*it;
20     }
21 }
22
23 uint32_t Hash::operator()() {
24     return this->count;
25 }

```

may 10, 19 13:08

common\_certificate.h

Page 1/1

```

1  #ifndef COMMON_CERTIFICATE_H
2  #define COMMON_CERTIFICATE_H
3  #include <stdint.h>
4  #include <string>
5  #include <queue>
6  #include <istream>
7  #include "common_key.h"
8  #include "common_protocol.h"
9
10 class Certificate {
11     private:
12         uint32_t serial_number;
13         std::string subject;
14         std::string issuer;
15         std::string not_before;
16         std::string not_after;
17         Key key;
18
19     public:
20         Certificate(std::string& _subject, std::string& _not_before, \
21                     std::string& _not_after, Key _key);
22         Certificate();
23         ~Certificate();
24         /*
25          * Se envian los datos del certificado por el socket.
26          */
27         void send(Protocol& protocol);
28         /*
29          * Se lee el archivo contenedor del certificado,
30          * se envian los datos a traves del socket
31          * y se devuelve el valor del certificado hashado
32          */
33         uint32_t send(std::string& filename, Protocol& protocol);
34         /*
35          * Se reciben los datos del certificado por el socket.
36          */
37         void receive(Protocol& protocol);
38         /*
39          * Setter del serial number
40          */
41         void addSerial(uint32_t serial_number);
42         /*
43          * Se imprime el certificado con formato:
44          *
45          * certificate:
46          *   serial number: n (hexa n 4 bytes)
47          *   subject: <subject>
48          *   issuer: Taller de programacion 1
49          *   validity:
50          *     not before: <MMM DD HH:mm:ss YYYY>
51          *     not after: <MMM DD HH:mm:ss YYYY>
52          *   subject public key info:
53          *     modulus: n (hexa n 2 bytes)
54          *     exponent: 19 (hexa n 1 byte)
55          */
56         std::string toString();
57         std::string getSubject();
58         Key getKey();
59         friend std::ostream& operator<<(std::ostream& o, Certificate& self);
60         //friend std::istream& operator>>(std::istream& in, Certificate& self);
61     };
62
63 #endif

```

may 10, 19 13:08

common\_certificate.cpp

Page 1/3

```

1  #include <string>
2  #include <sstream>
3  #include <iostream>
4  #include <fstream>
5  #include <ios>
6  #include "common_certificate.h"
7  #include "common_hash.h"
8  #define CERTIFICATE "certificate:\n"
9  #define SERIAL_NUMBER "\tserial number: "
10 #define SN_SIZE 4
11 #define SUBJECT "\tsubject: "
12 #define ISSUER "\tissuer: Taller de programacion 1\n"
13 #define VALIDITY "\tvalidity:\n"
14 #define NOT_BEFOR "\t\tnot before: "
15 #define NOT_AFTER "\t\tnot after: "
16 #define KEY "\tsubject public key info:\n"
17 #define MODULE "\t\tmodulus: "
18 #define EXPONENT "\t\texponent: "
19 Certificate::Certificate() {}
20
21 Certificate::Certificate(std::string& _subject, std::string& _not_before, \
22                         std::string& _not_after, Key _key):
23     key(_key) {
24         subject = std::move(_subject);
25         issuer = std::string(ISSUER);
26         not_before = std::move(_not_before);
27         not_after = std::move(_not_after);
28     }
29
30 void Certificate::send(Protocol& protocol) {
31     try {
32         protocol.send(this->serial_number);
33         protocol.send(this->subject);
34         protocol.send(this->not_before);
35         protocol.send(this->not_after);
36         this->key.send(protocol);
37     }
38     catch (std::runtime_error) {
39         throw std::runtime_error("Error while sending certificate");
40     }
41 }
42
43
44 void Certificate::receive(Protocol& protocol) {
45     protocol.receive(this->serial_number);
46     protocol.receive(this->subject);
47     protocol.receive(this->not_before);
48     protocol.receive(this->not_after);
49     this->key.receive(protocol);
50 }
51
52 std::string toHexString(int n, int len) {
53     std::string hexa;
54     hexa += std::string("(0x");
55
56     std::stringstream stream;
57     stream << std::hex << n;
58     std::string number = stream.str();
59     //2 digits per byte
60     int to = len*2 - number.length();
61     for (int i = 0; i < to; i++) {
62         hexa += '0';
63     }
64     return hexa + number + ')';
65 }
66

```

may 10, 19 13:08

common\_certificate.cpp

Page 2/3

```

67 std::string Certificate::getSubject() {
68     return this->subject;
69 }
70
71 Key Certificate::getKey() {
72     return this->key;
73 }
74
75 std::string Certificate::toString() {
76     std::string o;
77     o += std::string(CERTIFICATE);
78     int n = (int) this->serial_number;
79     std::string sn = toHexString(n, SN_SIZE);
80     o += SERIAL_NUMBER + std::to_string(n) + ' ' + sn + '\n';
81
82     o += SUBJECT + this->subject + '\n';
83     o += ISSUER;
84     o += VALIDITY;
85     o += NOT_BEFORE + this->not_before + '\n';
86     o += NOT_AFTER + this->not_after + '\n';
87
88     o += KEY;
89
90     Key k = this->key;
91     o += MODULE;
92     k.printModule(o, toHexString);
93     o += "\n";
94     o += EXPONENT;
95     k.printPublicExponent(o, toHexString);
96
97     return o;
98 }
99
100 uint32_t Certificate::send(std::string& filename, Protocol& protocol) {
101     std::ifstream file;
102     file.open(filename);
103     if (!file.good()) {
104         throw std::runtime_error("Error with certificate file");
105     }
106     std::string line;
107     int count = 0;
108     int pos;
109     int len;
110     std::string module;
111     std::string exp;
112     Hash hash;
113     while (std::getline(file, line, '\n')) {
114         if (!file.eof()) {
115             std::string aux = line + '\n';
116             hash.load(aux);
117         } else {
118             hash.load(line);
119         }
120         pos = line.find(':');
121         line = line.substr(pos);
122         len = line.length();
123         if (pos + 2 > len) { //certificate:\0
124             ++count;
125             continue;
126         }
127         line = line.substr(pos + 2, len);
128         if (count == 1) {
129             len = line.find(' ');
130             uint32_t n = (uint32_t) std::stoi(line.substr(0, len));
131             try {
132                 protocol.send(n);

```

may 10, 19 13:08

common\_certificate.cpp

Page 3/3

```

133     }
134     catch (std::runtime_error) {
135         throw std::runtime_error("Error while sending certificate");
136     }
137     } else if ((count == 2) | (count == 5) | (count == 6)) {
138         try {
139             protocol.send(line);
140         }
141         catch (std::runtime_error) {
142             throw std::runtime_error("Error while sending certificate");
143         }
144     } else if (count == 8) {
145         len = line.find(' ');
146         module = line.substr(0, len);
147     } else if (count == 9) {
148         len = line.find(' ');
149         exp = line.substr(0, len);
150     }
151     ++count;
152 }
153 Key key(exp, module);
154 try {
155     key.send(protocol);
156 }
157 catch (std::runtime_error) {
158     throw std::runtime_error("Error while sending certificate");
159 }
160 return hash();
161 }
162
163 std::ostream& operator<<(std::ostream &o, Certificate& self) {
164     std::string formal_certificate;
165     formal_certificate = self.toString();
166     o << formal_certificate;
167     return o;
168 }
169
170 Certificate::~Certificate() {}
171
172 void Certificate::addSerial(uint32_t _serial_number) {
173     this->serial_number = _serial_number;
174 }

```

may 10, 19 13:08

client\_time.h

Page 1/1

```

1  #ifndef COMMON_TIME_H
2  #define COMMON_TIME_H
3  #include <time.h>
4  #include <chrono>
5  #include <string>
6
7  class Time {
8  private:
9      std::chrono::system_clock::time_point date;
10 public:
11     /*
12      * Contenedor de tiempo que se inicializa con la fecha de su creaci³n.
13      */
14     Time();
15     //Time(std::string filename, std::string& to, std::string& from);
16
17     /*
18      * Le suma 60 dias a la fecha actual.
19      */
20     void validTo();
21     /*
22      * Almacena en str la fecha en formato MMM DD HH:mm:ss YYYY
23      */
24     void toString(std::string& str);
25 };
26
27 #endif

```

may 10, 19 13:08

client\_time.cpp

Page 1/1

```

1  #include <ctime>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <sstream>
6  #include "client_time.h"
7  #define BEGIN 4
8  #define DATE_LEN 27 //buffer which should have room for at least 26 bytes
9
10 Time::Time() {
11     this->date = std::chrono::system_clock::now();
12 }
13
14 void Time::validTo() {
15     // common_time.cpp:18: Consider using ctime_r(...) instead of ctime(...)
16     // for improved thread safety. [runtime/threadsafe_fn] [2]
17     std::chrono::duration<int, std::ratio<60*60*60*24> > month(1);
18     this->date = this->date + month;
19 }
20
21 void Time::toString(std::string& str) {
22     std::time_t t = std::chrono::system_clock::to_time_t(this->date);
23
24     struct tm* time = 0; //âM-^@M-^XtimeâM-^@M-^Y is used uninitialized
25     localtime_r(&t, time);
26     std::stringstream ssTp;
27     ssTp << std::put_time(time, "%b %d %T %Y");
28     str = ssTp.str();
29 }
30
31
32 /***** PUT_TIME *****/
33 * %b Abbreviated month name
34 * %d Day of the month, zero-padded (01-31)
35 * %T ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S
36 * %Y Year complete number
37 * str << std::put_time(this->date, "%b %d %T %Y");
38 *
39 * => STR = MMM DD HH:mm:ss YYYY
40 */

```

may 10, 19 13:08

## client\_revoke\_processor.h

Page 1/1

```

1  #ifndef CLIENT_REVOKE_PROCESSOR_H
2  #define CLIENT_REVOKE_PROCESSOR_H
3  #include "common_protocol.h"
4  #include "client_processor.h"
5  #include <string>
6
7  class RevokeProcessor : Processor{
8      private:
9          //Protocol& protocol;
10
11      public:
12          explicit RevokeProcessor(Protocol& _protocol);
13          ~RevokeProcessor();
14          virtual void run(std::string& certificate_filename, \
15                          std::string& client_key_filename, \
16                          std::string& server_key_filename) override;
17      };
18
19 #endif

```

may 10, 19 13:08

## client\_revoke\_processor.cpp

Page 1/2

```

1  #include <string>
2  #include <iostream>
3  #include "client_revoke_processor.h"
4  #include "client_invalid_request.h"
5  #include "common_certificate.h"
6  #include "common_rsa.h"
7  #define HASH_ERROR_SM 2
8  #define HASH_ERROR_MSSG "Error: los hashes no coinciden.\n"
9  #define INVALID_CERTIFICATE 1
10 #define INVALID_CERTIFICATE_MSSG "Error: usuario no registrado.\n"
11 #define HASH "Hash calculado: "
12 #define PRIVATE_ENCRYPTION "Hash encriptado con la clave privada: "
13 #define PUBLIC_ENCRYPTION "Huella enviada: "
14
15 RevokeProcessor::RevokeProcessor(Protocol& _protocol) : Processor(_protocol) {}
16 RevokeProcessor::~RevokeProcessor() {}
17
18 void RevokeProcessor::run(std::string& certificate_filename, \
19                          std::string& client_key_filename, std::string& server_key_filename) {
20     uint8_t command = 1;
21     try {
22         this->protocol.send(command);
23     }
24     catch (std::runtime_error) {
25         throw std::runtime_error("Error while sending command new");
26     }
27
28     Certificate certificate;
29     Rsa rsa;
30     try {
31         Key server_key(server_key_filename);
32         Key client_key(client_key_filename);
33         rsa.set(server_key, client_key);
34     }
35     catch(...) {
36         throw std::runtime_error("Error creating key");
37     }
38     uint32_t hash;
39
40     try {
41         hash = certificate.send(certificate_filename, protocol);
42     }
43     catch(...) {
44         __throw_exception_again;
45     }
46
47     uint32_t priv_encryption = rsa.privateEncryption(hash);
48     uint32_t publ_encryption = rsa.publicEncryption(priv_encryption);
49     try {
50         this->protocol.send(publ_encryption);
51     }
52     catch (std::runtime_error) {
53         throw std::runtime_error("Error while sending encrypted hash");
54     }
55     uint8_t status = 0;
56     this->protocol.receive(status);
57
58     std::cout << HASH << (int) hash << '\n';
59     std::cout << PRIVATE_ENCRYPTION << (int) priv_encryption << '\n';
60     std::cout << PUBLIC_ENCRYPTION << (int) publ_encryption << '\n';
61     if (status == HASH_ERROR_SM) {
62         throw InvalidRequest(HASH_ERROR_MSSG);
63     }
64     if (status == INVALID_CERTIFICATE) {
65         throw InvalidRequest(INVALID_CERTIFICATE_MSSG);
66     }

```

may 10, 19 13:08

client\_revoke\_processor.cpp

Page 2/2

```
67     return;  
68 }
```

may 10, 19 13:08

client\_processor.h

Page 1/1

```
1  #ifndef CLIENT_PROCESSOR_H_  
2  #define CLIENT_PROCESSOR_H_  
3  #include <string>  
4  #include "common_protocol.h"  
5  
6  class Processor {  
7      protected:  
8          Protocol& protocol;  
9      public:  
10         explicit Processor(Protocol& _protocol);  
11         ~Processor();  
12         virtual void run(std::string& filename1, std::string& filename2, \  
13                         std::string& filename3) = 0;  
14     };  
15  
16 #endif
```



may 10, 19 13:08

client\_processor.cpp

Page 1/1

```
1 #include "client_processor.h"
2
3 Processor::Processor(Protocol& _protocol) : protocol(_protocol) {}
4 Processor::~Processor() {}
```

may 10, 19 13:08

client\_new\_processor.h

Page 1/1

```
1 #ifndef CLIENT_NEW_PROCESSOR_H
2 #define CLIENT_NEW_PROCESSOR_H
3 #include <string>
4 #include "common_protocol.h"
5 #include "client_processor.h"
6
7 class NewProcessor : Processor{
8     private:
9         // Protocol& protocol;
10     public:
11         explicit NewProcessor(Protocol& protocol);
12         ~NewProcessor();
13         virtual void run(std::string& certificate_information_filename, \
14                         std::string& client_key_filename, \
15                         std::string& server_key_filename) override;
16     };
17
18 #endif
```

may 10, 19 13:08

client\_new\_processor.cpp

Page 1/2

```

1  #include <string>
2  #include <iostream>
3  #include <fstream>
4  #include "client_new_processor.h"
5  #include "client_applicant_request.h"
6  #include "common_socket.h"
7  #include "common_certificate.h"
8  #include "common_hash.h"
9  #include "common_rsa.h"
10 #include "client_invalid_request.h"
11 #define USER_ERROR_MSSG "Error: usuario no registrado.\n"
12 #define USER_ERROR 0
13 #define CERTIFICATE_ERROR_MSSG "Error: ya existe un certificado.\n"
14
15 #define CERIFICATE_ERROR_RECIVED_MSSG 0
16 #define HASH_ERROR_MSSG "Error: los hashes no coinciden.\n"
17 #define HASH_ERROR_SERVER_MSSG 1
18 #define HASH_OK_SERVER_MSSG 0
19 #define CERT_FP "Huella del servidor: "
20 #define SH "Hash del servidor: "
21 #define MH "Hash calculado: "
22
23
24 NewProcessor::NewProcessor(Protocol& _protocol) : Processor(_protocol) {}
25
26 NewProcessor::~NewProcessor() {}
27
28 void NewProcessor::run(std::string& certificate_information_filename, \
29                      std::string& client_key_filename, \
30                      std::string& server_key_filename) {
31     ApplicantRequest request(certificate_information_filename, \
32                             client_key_filename);
33     try {
34         request.send(protocol);
35     }
36     catch (std::runtime_error) {
37         __throw_exception_again;
38     }
39     uint8_t answer = 1;
40     protocol.receive(answer);
41     if (answer == CERIFICATE_ERROR_RECIVED_MSSG) {
42         throw InvalidRequest(CERTIFICATE_ERROR_MSSG);
43     }
44
45     Certificate certificate;
46     certificate.receive(protocol);
47     std::string formal_certificate;
48     formal_certificate = certificate.toString();
49
50
51     Hash hash(formal_certificate);
52     uint32_t my_hash = hash();
53
54
55     uint32_t certificate_footprint = 0;
56     protocol.receive(certificate_footprint);
57
58     Key client_key = request.getClientKey();
59     Key server_key;
60     try {
61         server_key.set(server_key_filename);
62     }
63     catch (std::runtime_error) {
64         throw std::runtime_error("Error seting server key");
65     }
66     Rsa rsa(server_key, client_key);

```

may 10, 19 13:08

client\_new\_processor.cpp

Page 2/2

```

67     uint32_t pd = rsa.privateDesencryption(certificate_footprint);
68     uint32_t server_hash = rsa.publicDesencryption(pd);
69
70     std::cout << CERT_FP << certificate_footprint << '\n';
71     std::cout << SH << server_hash << '\n';
72     std::cout << MH << my_hash << '\n';
73
74     uint8_t notification = HASH_OK_SERVER_MSSG;
75     if (my_hash != server_hash) {
76         notification = HASH_ERROR_SERVER_MSSG;
77         try {
78             protocol.send(notification);
79         }
80         catch (std::runtime_error) {
81             throw std::runtime_error(\
82                 "Error , the server could not be notified there was a hash error" );
83         }
84
85         throw InvalidRequest(HASH_ERROR_MSSG);
86     }
87
88     try {
89         protocol.send(notification);
90     }
91     catch (std::runtime_error) {
92         throw std::runtime_error(\
93             "Error , the server could not be notified there was no hash error" );
94     }
95
96     std::string filename = request.getSubject() + ".cert";
97     std::ofstream file;
98     file.open(filename, std::ofstream::out | std::ofstream::trunc);
99
100     if (!file.is_open()) {
101         //exc
102     }
103     file << formal_certificate;
104     return;
105 }

```

may 10, 19 13:08

client\_invalid\_request.h

Page 1/1

```

1  #ifndef CLIENT_INVALID_REQUEST_H
2  #define CLIENT_INVALID_REQUEST_H
3  #include <exception>
4  #include <string>
5
6  class InvalidRequest: public std::exception {
7  protected:
8      std::string msg;
9  public:
10     explicit InvalidRequest(const char* message);
11
12     explicit InvalidRequest(const std::string& message);
13
14     virtual ~InvalidRequest() throw();
15
16     virtual const char* what() const throw();
17 };
18
19 #endif

```

may 10, 19 13:08

client\_invalid\_request.cpp

Page 1/1

```

1  #include "client_invalid_request.h"
2  #include <string>
3
4  InvalidRequest::InvalidRequest(const char* message):
5      msg(message) {}
6
7  InvalidRequest::InvalidRequest(const std::string& message):
8      msg(message) {}
9
10 InvalidRequest::~InvalidRequest() throw(){}
11
12 const char* InvalidRequest::what() const throw(){
13     return msg.c_str();
14 }

```

may 10, 19 13:08

client.cpp

Page 1/2

```

1  #include <iostream>
2  #include <ostream>
3  #include <fstream>
4  #include <string>
5  #include "common_socket.h"
6  #include "client_new_processor.h"
7  #include "client_revoke_processor.h"
8  #include "client_invalid_request.h"
9  #define ARGUMENT_ERROR_MSSG "Error: argumentos invalidos.\n"
10 #define CF_SIZE 4 //certificate footprint size in bytes
11 #define HASH_SIZE 4
12
13
14 int main(int argc, char* argv[]) {
15     try {
16         if (argc != 7) {
17             std::cout << ARGUMENT_ERROR_MSSG;
18             return 0;
19         }
20
21         std::string model("new");
22         std::string mode2("revoke");
23         std::string mode = std::string(argv[3]);
24         if (mode != model ^ mode != mode2) {
25             std::cout << ARGUMENT_ERROR_MSSG; //no es un throw
26                                             //porque no hay donde catchearlo
27             return 0;
28         }
29
30         char* host = argv[1];
31         char* port = argv[2];
32         Socket skt;
33         skt.connectWithServer(host, port);
34
35         Protocol protocol(skt);
36
37
38         std::string client_key_filename = std::string(argv[5]);
39         std::string server_key_filename = std::string(argv[6]);
40
41
42         if (mode == model) {
43             std::string certificate_information_filename(argv[4]);
44             NewProcessor np(protocol);
45             np.run(certificate_information_filename, \
46                  client_key_filename, \
47                  server_key_filename);
48             return 0;
49         }
50
51         if (mode == mode2) { //std::string(argv[4])
52             std::string certificate_filename = std::string(argv[4]);
53             RevokeProcessor rp(protocol);
54             rp.run(certificate_filename, \
55                  client_key_filename, \
56                  server_key_filename);
57             return 0;
58         }
59         return 0;
60     }
61     catch(InvalidRequest &e) {
62         std::cout << e.what();
63     }
64     catch(std::runtime_error &e) {
65         std::cerr << e.what() << std::endl;
66     }

```

may 10, 19 13:08

client.cpp

Page 2/2

```

67 }

```

may 10, 19 13:08

client\_certificate\_info\_parser.h

Page 1/1

```

1  #ifndef CLIENT_CERTIFICATE_INFO_PARSER_H
2  #define CLIENT_CERTIFICATE_INFO_PARSER_H
3  #include <string>
4
5  /*****
6   * Lee una rchivo de tipo
7   * <subject>\n<date from>\n<date to>
8   * Donde las fechas son de tipo MMM DD HH:mm:ss YYYY
9   * Y almacena los atributos leidos en los repectivos string pasados por parame-
10  * tro.
11  * Si las fechas no existen from sera la fecha actual y to 30 dias despues.
12  */
13
14 class CertificateInfoParser {
15     public:
16         CertificateInfoParser();
17         void run(std::string& filename, std::string& subject,\
18                 std::string& from, std::string& to);
19         ~CertificateInfoParser();
20 };
21
22 #endif

```

may 10, 19 13:08

client\_certificate\_info\_parser.cpp

Page 1/1

```

1  #include <fstream>
2  #include <sstream>
3  #include <iostream>
4  #include <string>
5  #include "client_certificate_info_parser.h"
6  #include "client_time.h"
7
8  void CertificateInfoParser::run(std::string& filename, \
9                                std::string& subject, std::string& from,\
10                                std::string& to) {
11
12     std::ifstream file;
13     file.open(filename);
14     if (!file.good()) {
15         throw std::runtime_error("Error with file while parsing certificate");
16     }
17     std::string line;
18     int i = 0;
19     while (std::getline(file, line, '\n')) {
20         if (i == 0) {
21             subject = line.c_str();
22         } else if (i == 1) {
23             from = line.c_str();
24         } else if (i == 2) {
25             to = line.c_str();
26         }
27         ++i;
28     }
29     if (i == 1) {
30         Time tm;
31         tm.toString(from);
32         tm.validTo();
33         tm.toString(to);
34     }
35
36     CertificateInfoParser::CertificateInfoParser() {}
37     CertificateInfoParser::~CertificateInfoParser() {}

```

may 10, 19 13:08

client\_applicant\_request.h

Page 1/1

```

1  #ifndef CLIENT_APPLICANT_REQUEST_H
2  #define CLIENT_APPLICANT_REQUEST_H
3  #include <stdint.h>
4  #include <string>
5  #include "common_key.h"
6  #include "common_protocol.h"
7
8
9  class ApplicantRequest {
10     private:
11         std::string subject;
12         Key key;
13         std::string date_from;
14         std::string date_to;
15
16     public:
17         /*
18          * Recibe los dos archivos necesarios para solicitar un nuevo apicante
19          */
20         ApplicantRequest(std::string& cert_filename, std::string& key_filename);
21
22         ~ApplicantRequest();
23         /*
24          * Envia a traves del socket recibido por parametro una solucitud con
25          * formato:
26          *
27          * <comando>          1 byte con valor 0.
28          * <subject__size>    4 bytes big endian sin signo
29          * <subject>          String sin âM-^@M-^X\0âM-^@M-^Y
30          * <modul>            2 bytes big endian sin signo
31          * <exponent>        1 byte
32          * <date__size>       4 bytes big endian sin signo
33          * <date_from>       String sin âM-^@M-^X\0âM-^@M-^Y
34          * <date__size>       4 bytes big endian sin signo
35          * <date_to>         String sin âM-^@M-^X\0âM-^@M-^Y
36          */
37         void send(Protocol& protocol);
38         Key getClientKey();
39         std::string getSubject();
40     };
41
42 #endif

```

may 10, 19 13:08

client\_applicant\_request.cpp

Page 1/1

```

1  #include <fstream>
2  #include <sstream>
3  #include <iostream>
4  #include <string>
5  #include "client_applicant_request.h"
6  #include "client_certificate_info_parser.h"
7  #define COMMAND_SIZE 1
8
9
10 ApplicantRequest::ApplicantRequest(std::string& cert_filename,\
11                                     std::string& key_filename) {
12     CertificateInfoParser cir;
13     try {
14         this->key.set(key_filename);
15         cir.run(cert_filename, this->subject, \
16                this->date_from, this->date_to);
17     }
18     catch(...) {
19         throw std::runtime_error("Error creating aplicant request");
20     }
21 }
22
23 ApplicantRequest::~ApplicantRequest() {}
24
25 void ApplicantRequest::send(Protocol& protocol) {
26     uint8_t command = 0;
27     try {
28         protocol.send(command);
29         protocol.send(this->subject);
30         key.send(protocol);
31         protocol.send(this->date_from);
32         protocol.send(this->date_to);
33     }
34     catch (std::runtime_error) {
35         throw std::runtime_error("Error while sending aplicant request");
36     }
37 }
38
39 std::string ApplicantRequest::getSubject(){
40     return this->subject;
41 }
42
43 Key ApplicantRequest::getClientKey() {
44     return this->key;
45 }

```

may 10, 19 13:08

## Table of Content

Page 1/1

1	<b>Table of Contents</b>	
2	1 server_thread.h..... sheets	1 to 1 ( 1) pages 1- 1 25 lines
3	2 server_thread.cpp... sheets	1 to 1 ( 1) pages 2- 2 22 lines
4	3 server_revoke_client_processor.h sheets	2 to 2 ( 1) pages 3- 3 43 lines
5	4 server_revoke_client_processor.cpp sheets	2 to 3 ( 2) pages 4- 5 92 lines
6	5 server_new_client_processor.h sheets	3 to 3 ( 1) pages 6- 6 52 lines
7	6 server_new_client_processor.cpp sheets	4 to 5 ( 2) pages 7- 9 136 lines
8	7 server_inexisting_certificate.h sheets	5 to 5 ( 1) pages 10- 10 20 lines
9	8 server_inexisting_certificate.cpp sheets	6 to 6 ( 1) pages 11- 11 15 lines
10	9 server_index.h..... sheets	6 to 6 ( 1) pages 12- 12 50 lines
11	10 server_index.cpp.... sheets	7 to 8 ( 2) pages 13- 15 139 lines
12	11 server_existing_certificate.h sheets	8 to 8 ( 1) pages 16- 16 20 lines
13	12 server_existing_certificate.cpp sheets	9 to 9 ( 1) pages 17- 17 15 lines
14	13 server.cpp..... sheets	9 to 9 ( 1) pages 18- 18 54 lines
15	14 server_client_processor.h sheets	10 to 10 ( 1) pages 19- 19 13 lines
16	15 server_client_processor.cpp sheets	10 to 10 ( 1) pages 20- 20 5 lines
17	16 server_acceptor.h... sheets	11 to 11 ( 1) pages 21- 21 37 lines
18	17 server_acceptor.cpp. sheets	11 to 12 ( 2) pages 22- 23 76 lines
19	18 common_socket.h..... sheets	12 to 12 ( 1) pages 24- 24 35 lines
20	19 common_socket.cpp... sheets	13 to 14 ( 2) pages 25- 28 200 lines
21	20 common_rsa.h..... sheets	15 to 15 ( 1) pages 29- 29 33 lines
22	21 common_rsa.cpp..... sheets	15 to 15 ( 1) pages 30- 30 57 lines
23	22 common_protocol.h... sheets	16 to 16 ( 1) pages 31- 31 33 lines
24	23 common_protocol.cpp. sheets	16 to 17 ( 2) pages 32- 33 102 lines
25	24 common_key.h..... sheets	17 to 17 ( 1) pages 34- 34 59 lines
26	25 common_key.cpp..... sheets	18 to 18 ( 1) pages 35- 36 127 lines
27	26 common_hash.h..... sheets	19 to 19 ( 1) pages 37- 37 30 lines
28	27 common_hash.cpp..... sheets	19 to 19 ( 1) pages 38- 38 26 lines
29	28 common_certificate.h sheets	20 to 20 ( 1) pages 39- 39 64 lines
30	29 common_certificate.cpp sheets	20 to 21 ( 2) pages 40- 42 175 lines
31	30 client_time.h..... sheets	22 to 22 ( 1) pages 43- 43 28 lines
32	31 client_time.cpp..... sheets	22 to 22 ( 1) pages 44- 44 41 lines
33	32 client_revoke_processor.h sheets	23 to 23 ( 1) pages 45- 45 20 lines
34	33 client_revoke_processor.cpp sheets	23 to 24 ( 2) pages 46- 47 69 lines
35	34 client_processor.h.. sheets	24 to 24 ( 1) pages 48- 48 17 lines
36	35 client_processor.cpp sheets	25 to 25 ( 1) pages 49- 49 5 lines
37	36 client_new_processor.h sheets	25 to 25 ( 1) pages 50- 50 19 lines
38	37 client_new_processor.cpp sheets	26 to 26 ( 1) pages 51- 52 106 lines
39	38 client_invalid_request.h sheets	27 to 27 ( 1) pages 53- 53 20 lines
40	39 client_invalid_request.cpp sheets	27 to 27 ( 1) pages 54- 54 15 lines
41	40 client.cpp..... sheets	28 to 28 ( 1) pages 55- 56 68 lines
42	41 client_certificate_info_parser.h sheets	29 to 29 ( 1) pages 57- 57 23 lines
43	42 client_certificate_info_parser.cpp sheets	29 to 29 ( 1) pages 58- 58 38 lines
44	43 client_applicant_request.h sheets	30 to 30 ( 1) pages 59- 59 43 lines
45	44 client_applicant_request.cpp sheets	30 to 30 ( 1) pages 60- 60 46 lines