```
#ifndef SERVER_THREAD_H_
#define SERVER_THREAD_H_
#include <thread>

class Thread {
    private:
        std::thread thread;
        bool is_dead;

    public:
        Thread();
        void start();
        void join();
        virtual void run() = 0;
        virtual void stop() = 0;
        virtual bool isDead() = 0;
        Thread(const Thread&) = delete;
        Thread& operator=(const Thread&) = delete;
        virtual ~Thread();
        Thread(Thread&& other);
        Thread& operator=(Thread&& other);
};

#endif
```

```
#include <iostream>
#include "server_thread.h"
Thread::Thread() {}

void Thread::start() {
   this->thread = std::thread(&Thread::run, this);
}


void Thread::join() {
   this->thread.join();
}

Thread::~Thread() {
   // std::cout << "Destruyendo Thread!" << std::endl;
}

Thread::Thread(Thread&& other) {
this->thread = std::move(other.thread);
}

```

may 10, 19 13:08 **server_revoke_client_processor.h** Page 1/1

```
#ifndef SERVER_REVOKE_CLIENT_PROCESSOR_H
#define SERVER_REVOKE_CLIENT_PROCESSOR_H
#include <stdint.h>
#include <string>
#include "common_socket.h"
#include "common_key.h"
#include "server_index.h"
#include "common_protocol.h"
#include "server_client_processor.h"

class RevokeClientProcessor : public ClientProcessor {
    private:
        Protocol protocol;
        Index& index;
        Key& server_key;
        bool is_dead;

    public:
        RevokeClientProcessor(Protocol& _protocol, Index& _index, Key& _key);

        ~RevokeClientProcessor();
        /*
         * Envia a traves del socket recibido por parametro una solucitud con
         * formato:
         *
         * <comando>        1 byte con valor 0.
         * <subject__size>  4 bytes big endian sin signo
         * <subject>        String sin âM−^@M−^X\0âM−^@M−^Y
         * <modul>          2 bytes big endian sin signo
         * <exponent>       1 byte
         * <date__size>     4 bytes big endian sin signo
         * <date_from>      String sin âM−^@M−^X\0âM−^@M−^Y
         * <date__size>     4 bytes big endian sin signo
         * <dat_to>         String sin âM−^@M−^X\0âM−^@M−^Y
        */
        virtual void run() override;
        //bool checkCertificate(Socket& protocol);
        virtual void stop() override;
        virtual bool isDead() override;
};

#endif
```

may 10, 19 13:08 **server_revoke_client_processor.cpp** Page 1/2

```
#include <iostream>
#include <string>
#include "server_revoke_client_processor.h"
#include "server_inexisting_certificate.h"
#include "common_certificate.h"
#include "common_rsa.h"
#include "common_hash.h"
#define HASH_ERROR_MSSG 2
#define HASH_ERROR 1
#define INVALID_CERTIFICATE_MSSG 1
#define INVALID_CERTIFICATE 1
#define OK 0
#define OK_MSSG 0



RevokeClientProcessor::RevokeClientProcessor(Protocol& _protocol,\
                                             Index& _index, Key& _key):
    index(_index),
    server_key(_key),
    is_dead(false) {
        protocol = std::move(_protocol);
    }

RevokeClientProcessor::~RevokeClientProcessor() {}

void RevokeClientProcessor::run() {
    Certificate certificate;
    certificate.receive(protocol);
    uint32_t encryption = 0;
    this→protocol.receive(encryption);
    uint8_t answer;
    Key client_key = index.find(certificate);

    try {
        this→index.erase(certificate);
    }
    catch(InexistingCertificate) {
        answer = INVALID_CERTIFICATE_MSSG;
        try {
            this→protocol.send(answer);
        }
        catch(std::runtime_error) {
            throw std::runtime_error(\
    "Error, client could not be notified that there was a certificate error");
        }
        return;
    }
    Rsa rsa(client_key, server_key);
    uint32_t desencryption = rsa.privateDesencryption(encryption);
    uint32_t client_hash = rsa.publicDesencryption(desencryption);
    std::string formal_certificate = certificate.toString();
    Hash hash(formal_certificate);
    uint32_t my_hash = hash();
    if (my_hash ≠ client_hash) {
        answer = HASH_ERROR_MSSG;
        this→index.putBack(certificate, client_key);
        try {
            this→index.putBack(certificate, client_key);
            this→protocol.send(answer);
        }
        catch(std::runtime_error) {
            throw std::runtime_error(\
      "Error, the client could not be notified  that there was a hash errort");
        }
        catch(...) {
            __throw_exception_again;
        }
        return;
    }
    //index.erase(certificate);
    answer = OK_MSSG;
    try {
        this→protocol.send(answer);
    }
    catch(std::runtime_error) {
        throw std::runtime_error(\
            "Error, the client could not be notified that there was no error");
    }
    is_dead = true;
    return;
}

bool RevokeClientProcessor::isDead() {
   return is_dead;
}


void RevokeClientProcessor::stop() {
    this→protocol.stop();
}
```

```
#ifndef SERVER_NEW_CLIENT_PROCESSOR_H
#define SERVER_NEW_CLIENT_PROCESSOR_H
#include <stdint.h>
#include <string>
#include "server_client_processor.h"
#include "common_socket.h"
#include "common_key.h"
#include "server_index.h"

class New: public ClientProcessor {
    private:
        Protocol protocol;
        Index& index;
        Key& server_key;
        std::string subject;
        Key client_key;
        std::string date_from;
        std::string date_to;
        bool is_dead;
        void receiveInfo();
        std::string createCertificate();
        bool checkCertificate();


    public:
        /*
        * Recibe los dos archivos necesarios para solicitar un nuevo aplicante
        */
        New(Protocol& protocol, Index& _index, Key& key);

        ~New();
        /*
         * Envia a traves del socket recibido por parametro una solucitud con
         * formato:
         *
         * <comando>        1 byte con valor 0.
         * <subject__size>  4 bytes big endian sin signo
         * <subject>        String sin âM−^@M−^X\0âM−^@M−^Y
         * <modul>          2 bytes big endian sin signo
         * <exponent>       1 byte
         * <date__size>     4 bytes big endian sin signo
         * <date_from>      String sin âM−^@M−^X\0âM−^@M−^Y
         * <date__size>     4 bytes big endian sin signo
         * <dat_to>         String sin âM−^@M−^X\0âM−^@M−^Y
        */
        virtual void run() override;
        virtual void stop() override;
        virtual bool isDead() override;
};

#endif
```

```cpp
#include <string>
#include <iostream>
#include "server_new_client_processor.h"
#include "server_inexisting_certificate.h"
#include "server_existing_certificate.h"
#include "common_certificate.h"
#include "common_hash.h"
#include "common_rsa.h"
#define ENCRYPTION_SIZE 4
#define HASH_ERROR 1
#define CERTIFICATE_ERROR 0
#define CERTIFICATE_OK 1
#define CERT_STATUS_SIZE 1
#define ERROR_CODE 1
#define HASH_STATUS_SIZE 1

New::New(Protocol& _protocol, \
                                        Index& _index, Key& _key) :
    index(_index),
    server_key(_key),
    is_dead(false) {
        this→protocol = std::move(_protocol);
    }

New::~New() {}

void New::receiveInfo() {
    protocol.receive(this→subject);
    this→client_key.receive(protocol);

    protocol.receive(this→date_from);

    protocol.receive(this→date_to);
    if (subject.size() ≡ 0) {
        throw std::runtime_error("");
    }
}

std::string New::createCertificate() {
    std::string subj = this→subject;
    Certificate certificate(subj, this→date_from, this→date_to,\
             this→client_key);

    uint8_t answer = CERTIFICATE_OK;
    try {
        this→index.save(certificate);
    }
    catch(ExistingCertificate) { //ExistingCertificate
        answer = CERTIFICATE_ERROR;
        try {
            protocol.send(answer);
        }
        catch(std::runtime_error) {
            throw std::runtime_error(
     "Error, client could not be notified that there was a certificate error");
        }
        __throw_exception_again;
    }
    try {
        protocol.send(answer);
    }
    catch(std::runtime_error) {
        throw std::runtime_error(\
        "Error, the client could not be notified that there was no error");
    }

```

```cpp
    std::string result = certificate.toString();
    try {
        certificate.send(protocol);
    }
    catch(std::runtime_error) {
        throw std::runtime_error(\
        "Error sending the certificate while processing new client");
    }
    return result;
}


uint32_t encrypt(Key client_key, Key server_key, uint32_t hash) {
    Rsa rsa(client_key, server_key);
    uint32_t encryption = rsa.privateEncryption(hash);
    encryption = rsa.publicEncryption(encryption);
    return encryption;
}

void New::run() {
    try {
        this→receiveInfo();
    }
    catch (...) {
        this→is_dead = true;
        return;
    }


    std::string formal_certificate;
    try {
        formal_certificate = this→createCertificate();
    } catch(ExistingCertificate) {
        return;
    }


    Hash hash(formal_certificate);
    uint32_t hashed_certificate =  hash();
    uint32_t encryption = encrypt(this→client_key, this→server_key,\
                                 hashed_certificate);
    try {
        protocol.send(encryption);
    }
    catch(std::runtime_error) {
        throw std::runtime_error(\
        "Error sending encrypted hash while processing new client");
    }
    uint8_t hash_status = 0;
    protocol.receive(hash_status);
    if (hash_status ≡ HASH_ERROR) {
        try {
            index.erase(this→subject);
        }
        catch(InexistingCertificate){
            throw std::runtime_error("Error erasing certificate");
        }
    }
    this→is_dead = true;
    return;
}

bool New::isDead() {
    return this→is_dead;
}

```

```
void New::stop() {
    this→protocol.stop();
}
```

```
#ifndef SERVER_INEXISTING_CERTIFICATE_H
#define SERVER_INEXISTING_CERTIFICATE_H
#include <exception>
#include <string>

class InexistingCertificate: public std::exception {
protected:
        std::string msg;
    public:
        explicit InexistingCertificate(const char* message);

        explicit InexistingCertificate(const std::string& message);

        virtual ~InexistingCertificate() throw();

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

#endif
```

```cpp
#include "server_inexisting_certificate.h"
#include <string>

InexistingCertificate::InexistingCertificate(const char* message):
      msg(message) {}

InexistingCertificate::InexistingCertificate(const std::string& message):
      msg(message) {}

InexistingCertificate::~InexistingCertificate() throw(){}

const char* InexistingCertificate::what() const throw(){
    return msg.c_str();
}
```

```cpp
#ifndef SERVER_INDEX_H
#define SERVER_INDEX_H
#include <string>
#include <map>
#include <mutex>
#include "common_key.h"
#include "common_certificate.h"

class Index {
    private:
        std::string& filename;
        std::map<std::string, Key> certificates;
        void parseLine(std::string& line);
        std::mutex mutex;
        uint32_t serial_number;



    public:
        Index();
        void write();
        void save(Certificate& certificate);
        //void increaseSerialNumber();

        /*
        * Vuelve a almacenar el certificado pasado por parametro.
        * Con client_key valor.
        */
        void putBack(Certificate& certificate,  Key& client_key);


        Key find(Certificate& cartificate);

        /* Borra el certificado
        *  Si el certificado no existe lanza una exepcion de tipo
        *  InexistentCertificate
        */
        void erase(Certificate& certificate);
        void erase(std::string& str);

        explicit Index(std::string& filename);
        ~Index();


        bool has(std::string& str);
        bool has(Certificate& certificate);
};

#endif
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include <queue>
#include <map>
#include "common_key.h"
#include "server_index.h"
#include "server_inexisting_certificate.h"
#include "server_existing_certificate.h"


Index::~Index() {}

Index::Index(std::string& _filename) : filename(_filename) {
    std::ifstream file;
    file.open(filename);
    if (¬file.good()) {
        throw std::runtime_error("Error with index file");
    }
    std::string line;
    std::getline(file, line, '\n');

    int aux;
    if (line.size() ≠ 0) { //if line emptry aux = random
        std::istringstream sn(line);
        sn >> aux;
        this→serial_number = (uint32_t) aux;
    } else {
        this→serial_number = 1;
    }
    while (std::getline(file, line, '\n')) {
        this→parseLine(line);
    }
}


void split(std::string& str, char c, std::queue<std::string>& container) {
  std::string buff{""};
  int i = 0;
    while (str[i]) {
        if (str[i] ≡ c ∧ buff ≠ "") {
            container.push(buff);
            buff = "";
        } else {
            buff+=str[i];
        }
        i++;
  }
  if (buff ≠ "") {
        container.push(buff);
    }
}

void Index::parseLine(std::string& line) {
    std::queue<std::string> container;
    split(line, ';', container);

    std::string subject = container.front();
    container.pop();
    std::string str_key = container.front();
    container.pop();

    split(str_key, ' ', container);
    std::string str1 = container.front();
    container.pop();
```

```
    std::string str2 = container.front();
    container.pop();
    Key key(str1, str2);
    this→certificates.insert({subject, key});
}


bool Index::has(std::string& str) {
    //std::unique_lock<std::mutex> lock(this->mutex);
    std::map<std::string, Key>::iterator it = this→certificates.find(str);
    bool result = it ≠ this→certificates.end();
    return result;
}

Key Index::find(Certificate& cartificate) {
    std::unique_lock<std::mutex> lock(this→mutex);
    std::map<std::string, Key>::iterator it = \
        this→certificates.find(cartificate.getSubject());
    return it→second;
}

bool Index::has(Certificate& certificate) {
    std::string sbj = certificate.getSubject();
    return this→has(sbj);
}

void Index::save(Certificate& certificate) {
    std::unique_lock<std::mutex> lock(this→mutex);
    if (this→has(certificate)) {
        throw ExistingCertificate("Error saving certificate");
    }
    this→certificates.insert({certificate.getSubject(), certificate.getKey()});
    certificate.addSerial(this→serial_number);
    this→serial_number ++;
}

void Index::putBack(Certificate& certificate, Key& client_key) {
    std::unique_lock<std::mutex> lock(this→mutex);
    this→certificates.insert({certificate.getSubject(), client_key});
}

void Index::erase(std::string& str) {
    std::unique_lock<std::mutex> lock(this→mutex);
    if (¬this→has(str)) {
        throw InexistingCertificate("Error erasing certificate");
    }
    this→certificates.erase(str);
}

void Index::erase(Certificate& certificate) {
    std::string subj = certificate.getSubject();
    try {
        this→erase(subj);
    }
    catch(InexistingCertificate) {
        __throw_exception_again;
    }
}

void Index::write() {
    std::ofstream file;
    file.open(this→filename, std::ofstream::out | std::ofstream::trunc);

    if (¬file.is_open()) {
        //exc
    }
    file << std::to_string(this→serial_number) << '\n';
    std::map<std::string, Key>::iterator it = this→certificates.begin();
    for (; it ≠ this→certificates.end(); ++it) {
        file << it→first << ";" << it→second << '\n';
    }
}
```

```
#ifndef SERVER_EXISTING_CERTIFICATE_H
#define SERVER_EXISTING_CERTIFICATE_H
#include <exception>
#include <string>

class ExistingCertificate: public std::exception {
protected:
        std::string msg;
    public:
        explicit ExistingCertificate(const char* message);

        explicit ExistingCertificate(const std::string& message);

        virtual ~ExistingCertificate() throw();

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

#endif
```

```cpp
#include <string>
#include "server_existing_certificate.h"

ExistingCertificate::ExistingCertificate(const char* message):
      msg(message) {}

ExistingCertificate::ExistingCertificate(const std::string& message):
      msg(message) {}

ExistingCertificate::~ExistingCertificate() throw(){}

const char* ExistingCertificate::what() const throw(){
    return msg.c_str();
}
```

```cpp
#include <string>
#include <iostream>
#include "server_index.h"
#include "server_acceptor.h"


#define ERROR_CODE 1
#define COMMAND_SIZE 1
#define LEN_SIZE 4



int main(int argc, char* argv[]) {
try {
    if (argc ≠ 4) {
        return ERROR_CODE;
    }

    std::string claves = std::string(argv[2]);
    std::string indice = std::string(argv[3]);
     if (argc ≠4) {
      return 1;
    }

    Socket skt;
    skt.connectWithClients(argv[1]);

    std::string index_filename(argv[3]);
    Index index(index_filename);


    std::string key_filename = std::string(argv[2]);
    Key key(key_filename);
    Acceptor acceptor(skt, index, key);
    acceptor.start();

    std::string line;
    while (std::getline(std::cin, line)) {
        if (line ≡ "q") {
            acceptor.stop();
            break;
        }
    }

    acceptor.join();
    index.write();

    return 0;
    }
catch(std::runtime_error &e) {
    std::cerr << e.what() << std::endl;
}
}
```

```
#ifndef SERVER_CLIENT_PROCESSOR_H
#define SERVER_CLIENT_PROCESSOR_H
#include "server_thread.h"


class ClientProcessor : public Thread {
    public:
        ClientProcessor();
        ~ClientProcessor();
};

#endif
```

```
#include "server_client_processor.h"

ClientProcessor::ClientProcessor() {}
ClientProcessor::~ClientProcessor() {}
```

```
#ifndef SERVER_ACCEPTOR_H
#define SERVER_ACCEPTOR_H
#include <string>
#include <mutex>
#include <thread>
#include <iostream>
#include <queue>
#include <vector>
#include "server_thread.h"
#include "common_socket.h"
#include "server_index.h"
#include "server_client_processor.h"
//#include "compare_bf.h"



class Acceptor : public Thread {
    private:
        Socket skt;
        Index& index;
        Key& key;
        Socket client_skt;
        bool keep_talking;
        std::vector<ClientProcessor*> clients;

    public:
        Acceptor(Socket& _skt, Index& index, Key& key);
        ~Acceptor();
        virtual void run() override;
        virtual void stop() override;
        virtual bool isDead() override;
};



#endif
```

```
#include <vector>
#include "server_acceptor.h"
#include "common_protocol.h"
#include "server_new_client_processor.h"
#include "server_revoke_client_processor.h"
#include "common_key.h"
#include "server_client_processor.h"

Acceptor::Acceptor(Socket& _skt, Index& _index, Key& _key):
    index(_index),
    key(_key),
    keep_talking(true) {
        this→skt = std::move(_skt);
    }

void Acceptor::run() {
try {
    while (this→keep_talking) {
        Socket client_skt;
        try {
            client_skt = this→skt.acceptClient();
        }
        catch (std::runtime_error &e) {
            std::runtime_error(e.what());
        }

        Protocol protocol(client_skt);

        uint8_t command;
        protocol.receive(command);

        ClientProcessor* client;
        if (command ≡ 0) {
            client = new New(protocol, index, key);
        }
        if (command ≡ 1) {
            client = new RevokeClientProcessor(protocol, index, key);
        }
        this→clients.push_back(client);
        client→start();
        std::vector<ClientProcessor*>::iterator it = this→clients.begin();
        while (it ≠ this→clients.end()) {
          ClientProcessor* client = *it;
            if (client→isDead()) {
                client→join();
              delete client;
              this→clients.erase(it);
          } else {
                ++it;
            }
        }
    }
}
catch(std::exception &e) {
    std::cerr << e.what() << std::endl;
}
}

void Acceptor::stop() {
    this→keep_talking = false;
    skt.kill();
}

bool Acceptor::isDead() {
    return ¬keep_talking;
}
```

```cpp

Acceptor::~Acceptor() {
    std::vector<ClientProcessor*>::iterator it = this→clients.begin();
    for (; it ≠ this→clients.end(); ++it) {
      ClientProcessor* client = *it;
            client→join();
        delete client;
    }
}
```

```cpp
#ifndef COMMON_SOCKET_H
#define COMMON_SOCKET_H
#include <stdlib.h>
#include <stdbool.h>

class Socket {
    private:
        int skt;

    public:
        Socket();
        ~Socket();

        explicit Socket(int skt);
        Socket(Socket∧ origin);
        Socket& operator=(Socket∧ origin);
        Socket(const Socket& origin);

        void connectWithClients(const char* port);
        Socket acceptClient();

        void connectWithServer(const char* host, const char* port);


        void receiveAll(void* buf, size_t len);
        int receiveSome(void* buf, size_t size);
        int sendAll(void* buf, size_t size);

        void kill();
        void setInvalid();
};


#endif
```

```
#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <errno.h>
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <utility>
#include <stdexcept>
#include <iostream>
#include <string>
#include "common_socket.h"


#define MAX_WAITING_CLIENTS 20

/*******************************************************************************
 * TODOS LOS
 * if (s < 0) {
 *    return
 * }
 * TIENEN QUE SER EXCEPCIONES EN UN FUTURO.
*/



Socket::Socket() {
    int familia_skt = AF_INET;      /* IPv4 (or AF_INET6 for IPv6)     */
    int tipo_skt = SOCK_STREAM;     /* TCP  (or SOCK_DGRAM for UDP)    */
    int protocolo_skt = 0;          /* Any protocol */
    int s = socket(familia_skt, tipo_skt, protocolo_skt);

    if (¬s) {
        throw std::runtime_error("Error creating socket");
    }
    this→skt = s;
}

Socket::Socket(int skt) {
    if (skt ≡ −1) {
        throw std::runtime_error("Error creating socket");
    }
    this→skt = skt;
}


Socket& Socket::operator=(Socket∧ origin) {
    if (this→skt ≠ −1) {
        shutdown(this→skt, SHUT_RDWR);
        close(this→skt);
    }

    this→skt = origin.skt;
    origin.skt = −1;
    return *this;
}

Socket::Socket(Socket∧ origin): skt(origin.skt) {
    origin.skt = −1;
}

```

```

void Socket::kill(){
    if (this→skt ≠ −1) {
        shutdown(this→skt, SHUT_RDWR);
        close(this→skt);
        this→skt = −1;
    }
}

Socket::~Socket() {
    if (this→skt ≠ −1) {
        shutdown(this→skt, SHUT_RDWR);
        close(this→skt);
    }
}


void Socket::connectWithClients(const char* port) {
    struct addrinfo hints;
    struct addrinfo *results, *ptr;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;          /* IPv4 (or AF_INET6 for IPv6)     */
    hints.ai_socktype = SOCK_STREAM;    /* TCP  (or SOCK_DGRAM for UDP)    */
    hints.ai_flags = AI_PASSIVE;        /* AI_PASSIVE for server           */

    int s = 0;
    s = getaddrinfo(NULL, port, &hints, &results);

    if (s ≠ 0) {
        throw std::runtime_error(strerror(s));
    }

    int val = 1;
    s = setsockopt(this→skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
    if (s ≡ −1) {
        throw std::runtime_error("Error at setsockopt");
    }

    ptr = results;
    s = bind(this→skt, ptr→ai_addr, ptr→ai_addrlen);
    if (s ≡ −1) {
        throw std::runtime_error("Error while binding");
    }
    freeaddrinfo(results);
    s = listen(this→skt, MAX_WAITING_CLIENTS);
    if (s ≡ −1) {
        throw std::runtime_error("Error while listening");
    }
    return;
}


void Socket::connectWithServer(const char* host, const char* port) {
    struct addrinfo hints;
    struct addrinfo *result, *ptr;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;          /* IPv4 (or AF_INET6 for IPv6)     */
    hints.ai_socktype = SOCK_STREAM;    /* TCP  (or SOCK_DGRAM for UDP)    */
    hints.ai_flags = 0;                 /* None                            */

    int s = 0;
    s = getaddrinfo(host, port, &hints, &result);

    bool are_we_connected = false;

    for (ptr = result; ptr ≠ NULL ∧ are_we_connected ≡ false;\
        ptr = ptr→ai_next) {
        s = connect(this→skt, ptr→ai_addr, ptr→ai_addrlen);
        are_we_connected = (s ≠ −1);
    }
    freeaddrinfo(result);

    if (¬are_we_connected) {
        throw std::runtime_error("Error while connecting with client");
    }
}


void Socket::receiveAll(void* buf, size_t len) {
    memset(buf, 0, len);
    size_t received = 0;
    int status = 0;
    bool keep_going = true;
    char* aux = (char*) buf;
    while (received < len ∧ keep_going) {
        status = this→receiveSome(&aux[received], len − received);
        //if (status == 0) {
        //    keep_going = true;
        //} else
        if (status < 0) {
            keep_going = false;
        } else {
            received += status;
        }
    }
}

int Socket::receiveSome(void* buf, size_t size) {
    return ::recv(this→skt, buf , size, MSG_NOSIGNAL);
}

int Socket::sendAll(void* buf, size_t size) {
    int bytes_sent = 0;
    int s = 0;
    bool is_the_socket_valid = true;
    char* aux = (char*) buf;

    while (bytes_sent < (int)size ∧ is_the_socket_valid) {
        s = ::send(this→skt, &aux[bytes_sent], \
                size−bytes_sent, MSG_NOSIGNAL);
        if (s ≤ 0) {
            std::string error("Error: ");
            throw std::runtime_error(error + strerror(errno));
        } else {
            bytes_sent += s;
        }
    }
    return bytes_sent;
}


Socket Socket::acceptClient(){
    int peerskt = accept(this→skt, NULL, NULL);
    if (peerskt ≡ −1) {
        throw std::runtime_error("Error accepting  client");
    }
    Socket skt(peerskt);
    return std::move(skt);
}

void Socket::setInvalid() {
    this→skt = −1;
```

```
}
```

```
#ifndef RSA_H
#define RSA_H
#include <stdint.h>
#include "common_key.h"

/*
* Algoritmo de encriptaciÃ³n asimÃ©trico.
*/
class Rsa {
    private:
        uint8_t public_exponent;
        uint16_t public_module;
        uint8_t private_exponent;
        uint16_t private_module;
    public:
        Rsa(Key& public_key, Key& private_key);
        Rsa();
        ~Rsa();

        void set(Key& public_key, Key& private_key);

        /*
        * Recibe un mensaje (que es un nÃºmero de 4 bytes sin signo) y lo
        * encrita en otro entero de 4 bytes sin signo.
        */
        uint32_t publicEncryption(const uint32_t hash);
        uint32_t privateEncryption(const uint32_t hash);
        uint32_t publicDesencryption(const uint32_t hash);
        uint32_t privateDesencryption(const uint32_t hash);
};

#endif
```

```
#include "common_rsa.h"
//size in bytes of the message to encrypt
#define HASH_SIZE 4

Rsa::Rsa(Key& public_key, Key& private_key) {
    this→public_exponent = public_key.getPublicExponent();
    this→public_module = public_key.getModule();
    this→private_exponent = private_key.getPrivateExponent();
    this→private_module = private_key.getModule();
}

Rsa::Rsa(){}

void Rsa::set(Key& public_key, Key& private_key) {
    this→public_exponent = public_key.getPublicExponent();
    this→public_module = public_key.getModule();
    this→private_exponent = private_key.getPrivateExponent();
    this→private_module = private_key.getModule();
}

Rsa::~Rsa() {}

uint32_t encrypt(const uint32_t exponent, const uint32_t module,\
                 const uint32_t hash) {
    uint32_t result;
    uint32_t base;

    uint32_t ret = 0;

    for (uint32_t i = 0; i < HASH_SIZE; ++i) {
        result = (hash >> (i * 8)) & 0xff;
        base = result;
        for (uint32_t j = 1; j < exponent; ++j)
            result = (result * base) % module;
        ret = ret + (result << (i * 8));
    }
    return ret;
}


uint32_t Rsa::publicEncryption(const uint32_t hash) {
    return encrypt(this→public_exponent, this→public_module, hash);
}

uint32_t Rsa::privateEncryption(const uint32_t hash) {
    return encrypt(this→private_exponent, this→private_module, hash);
}

uint32_t Rsa::publicDesencryption(const uint32_t hash){
    return encrypt(this→public_exponent, this→public_module, hash);
}

uint32_t Rsa::privateDesencryption(const uint32_t hash) {
    return encrypt(this→private_exponent, this→private_module, hash);
}
```

```
#ifndef COMMON_PROTOCOL_H
#define COMMON_PROTOCOL_H
#include "common_socket.h"
#include <string>

/*
 * EncapsulaciÃ³n del socket que conoce el protocolo de comunicaciÃ³n pedido
*/
class Protocol {
    private:
        Socket skt;

    public:
        Protocol();
        explicit Protocol(Socket& _skt);
        ~Protocol();
        Protocol(Protocol∧ origin);
        Protocol& operator=(Protocol∧ origin);
        void receive(uint8_t& n);
        void receive(uint16_t& n);
        void receive(uint32_t& n);

        void send(uint8_t& n);
        void send(uint16_t& n);
        void send(uint32_t& n);

        int send(std::string& str);
        int receive(std::string& str);
        void stop();
};

#endif
```

```
#include "common_protocol.h"
#include <string>
#include <iostream>
#define UINT8_SIZE 1
#define UINT16_SIZE 2
#define UINT32_SIZE 4

Protocol::Protocol(Socket& _skt) {
    skt = std::move(_skt);
}

Protocol::Protocol() {}

Protocol::~Protocol() {}

void Protocol::send(uint8_t& n) {
    try {
        this→skt.sendAll(&n, UINT8_SIZE);
    }
    catch(std::runtime_error &e) {
        throw std::runtime_error("Error in protocol while sending");
    }
}

void Protocol::send(uint16_t& n) {
    uint16_t aux = htobe16(n);
    try {
        this→skt.sendAll(&aux, UINT16_SIZE);
    }
    catch(std::runtime_error &e) {
        throw std::runtime_error("Error in protocol while sending");
    }
}

void Protocol::send(uint32_t& n) {
    uint32_t aux = htobe32(n);
    try {
        this→skt.sendAll(&aux, UINT32_SIZE);
    }
    catch(std::runtime_error &e) {
        throw std::runtime_error("Error in protocol while sending");
    }
}

int Protocol::send(std::string& str) {
    uint32_t len = (uint32_t) str.length();
    int sent = 0;
    try {
        this→send(len);
        for (size_t i = 0; i < len ∧ sent ≥ 0; ++i){
            sent = this→skt.sendAll(&str[i], 1);
        }
    }
    catch(std::runtime_error &e) {
        throw std::runtime_error("Error in protocol while sending");
    }
    return sent;
}


int Protocol::receive(std::string& str) {
    uint32_t len = (uint32_t) str.length();
    this→receive(len);
    char c;
    int received = 0;
    for (size_t i = 0; i < len; ++i){
        received += this→skt.receiveSome(&c, 1);
        str.append(1, c);
    }
    return received;
}

void Protocol::receive(uint8_t& n) {
    this→skt.receiveAll(&n, UINT8_SIZE);
}


void Protocol::receive(uint16_t& n) {
    uint16_t aux;
    this→skt.receiveAll(&aux, UINT16_SIZE);
    n = htobe16(aux);
}


void Protocol::receive(uint32_t& n) {
    uint32_t aux;
    this→skt.receiveAll(&aux, UINT32_SIZE);
    n = htobe32(aux);
}

void Protocol::stop() {}

Protocol& Protocol::operator=(Protocol∧ origin) {
    this→skt = std::move(origin.skt);
    return *this;
}

Protocol::Protocol(Protocol∧ origin) {
    this→skt = std::move(origin.skt);
    origin.skt.setInvalid();
}
```

```
#ifndef COMMON_KEY_H
#define COMMON_KEY_H
#include <stdint.h>
#include <string>
#include <queue>
#include <functional>
#include "common_protocol.h"

class Key {
    private:
        uint8_t public_exponent;   //1 byte
        uint8_t private_exponent; //1 byte
        uint16_t module;    //2 bytes

    public:
        Key();
        Key(const Key &self);
        Key(std::string& _public_exponent, std::string& _module);
        explicit Key(std::string& filename);
        ~Key();
        /*
         * Recibe el nombre de un archivo de tipo
         * <exp_publico> <exp_privado> <modulo>
         *
         * Los distintos campos pueden  estar separados  por 1 o mÃ¡s espacios,
         * Ãºnicamente se garantiza que estos se encuentran en una misma lÃnea.
        */
        void set(std::string& filename);
        void set(std::string& _public_exponent, std::string& _module);
        /*
         * Envia, a traves del socket pasado por parametro:
         * MÃ³dulo: en formato 2 bytes en big endian sin signo.
         * Exponente: en 1 byte.
        */
        void send(Protocol& protocol);
        void receive(Protocol& protocol);
        friend std::ostream& operator<<(std::ostream&, const Key&);
        /*
         * Recibe un string y una funciÃ³n que al pasarle el exponente publico y
         * su tamaÃ±o en bytes devuelve en  string una representaciÃ³n del mismo
.
         * Se imprime en el string tanto  el exponente pÃºblico como la tranfor−
         * maciÃ³n del mismo.
        */
        void printPublicExponent(std::string& o, \
                                 std::function<std::string(int, int)> transform);
        /*
         * Recibe un string y una funciÃ³n  que al pasarle el modulo y su tamaÃ±
o
         * en bytes devuelve en string una representaciÃ³n del mismo. Se imprime
         * en el string tanto el modulo como la tranformaciÃ³n del mismo.
        */
        void printModule(std::string& o, \
                                 std::function<std::string(int, int)> transform);
        uint8_t getPublicExponent();
        uint8_t getPrivateExponent();
        uint16_t getModule();
};

#endif
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include <algorithm>
#include "common_key.h"
#define EXPONENT_SIZE 1
#define MODULE_SIZE 2
#define PUBLIC_EXP_POS 0
#define PRIVATE_EXP_POS 1
#define MODULE_POS 2


Key::Key(std::string& _public_exponent, std::string& _module) {
    this→set(_public_exponent, _module);
}

Key::Key() {}

void Key::set(std::string& _public_exponent, std::string& _module){
    int aux;
    std::istringstream pe(_public_exponent);
    pe >> aux;
    this→public_exponent = (uint8_t) aux;
    std::istringstream m(_module);
    m >> aux;
    this→module = (uint16_t) aux;
}

Key::Key(std::string& filename) {
    this→set(filename);
}

void Key::set(std::string& filename) {
    std::ifstream file;
    file.open(filename);
    if (¬file.good()) {
        throw std::runtime_error("Error with file while seting key");
    }
    std::string line;
    int i = 0;
    int aux;
    while (std::getline(file, line, ’ ’)) {
        if (line.length() ≡ 0) continue;
        if (i ≡ PUBLIC_EXP_POS) {
            std::istringstream pbe(line);
            pbe >> aux;
            this→public_exponent = (uint8_t) aux;
        } else if (i ≡ PRIVATE_EXP_POS) {
            std::istringstream pre(line);
            pre >> aux;
            this→private_exponent = (uint8_t) aux;
        } else if (i ≡ MODULE_POS) {
                std::istringstream m(line);
                m >> aux;
                this→module = (uint16_t) aux;
        }
        i++;
    }
    if (i ≡ MODULE_POS) {
        this→module = this→private_exponent;
    }
    //file.close();
}

Key::Key(const Key &key) {
```

```
    this→private_exponent = key.private_exponent;
    this→public_exponent = key.public_exponent;
    this→module = key.module;
}

Key::~Key() {}

void Key::receive(Protocol& protocol) {
    protocol.receive(this→public_exponent);
    protocol.receive(this→module);
}

void Key::send(Protocol& protocol) {
    try {
        protocol.send(this→public_exponent);
        protocol.send(this→module);
    }
    catch(std::runtime_error) {
        throw std::runtime_error("Error while sending key");
    }
}

std::ostream& operator<<(std::ostream& o, const Key& self) {
    std::string s = std::to_string(self.public_exponent);
    o << s << ’ ’;
    s = std::to_string(self.module);
    o << s;
    return o;
}

void Key::printPublicExponent(std::string& o, \
                    std::function<std::string(int, int)> transform) {
    int exp = (int) this→public_exponent;
    std::string transformed_exp = transform(exp, EXPONENT_SIZE);
    o += std::to_string(exp) + ’ ’ + transformed_exp;
}

void Key::printModule(std::string& o, \
                    std::function<std::string(int, int)> transform) {
    int module = (int) this→module;
    std::string transformed_module = transform(module, MODULE_SIZE);
    o += std::to_string(module) + ’ ’ + transformed_module;
}


uint8_t Key::getPublicExponent() {
    return this→public_exponent;
}
uint8_t Key::getPrivateExponent() {
    return this→private_exponent;
}
uint16_t Key::getModule() {
    return this→module;
}

/*********  PRINT   *************************
    std::cerr << "\tKEY:\n";
    fprintf(stderr,"< %d > < %d >\n", this->public_exponent, this->module);

*/
```

```
#ifndef HASH_H
#define HASH_H
#include <stdint.h>
#include <string>

/*
* Recibe un texto que es cargado de a segmentos y lo hashea.
*/
class Hash {
    private:
        uint32_t count;

    public:
        Hash();
        explicit Hash(std::string& str);
        ~Hash();
        /*
        * Recibe una cadena y actualiza el resultado de la funciÃ³n de hash.
        */
        void load(std::string& str);

        /*
        * Devuelve el resultado (entero de 2 bytes) de hashear el texto
        * completo.
        */
        uint32_t operator()();
};

#endif
```

```
#include <string>
#include <iostream>
#include "common_hash.h"
#define INITIAL_COUNT 0

Hash::Hash() {
    this→count = INITIAL_COUNT;
}
Hash::~Hash() {}

Hash::Hash(std::string& str) {
    this→count = INITIAL_COUNT;
    this→load(str);
}

void Hash::load(std::string& str) {
    std::string::iterator it = str.begin();
    for (; it≠str.end(); ++it) {
        this→count += (uint8_t)*it;
    }
}

uint32_t Hash::operator()() {
    return this→count;
}
```

```cpp
#ifndef COMMON_CERTIFICATE_H
#define COMMON_CERTIFICATE_H
#include <stdint.h>
#include <string>
#include <queue>
#include <istream>
#include "common_key.h"
#include "common_protocol.h"

class Certificate {
    private:
        uint32_t serial_number;
        std::string subject;
        std::string issuer;
        std::string not_before;
        std::string not_after;
        Key key;

    public:
        Certificate(std::string& _subject, std::string& _not_before,\
                     std::string& _not_after, Key _key);
        Certificate();
        ~Certificate();
        /*
        * Se envian los datos del certificado por el socket.
        */
        void send(Protocol& protocol);
        /*
        * Se lee el archivo contenedor del certificado,
        * se envian los datos a traves del socket
        * y se devuelve el valor del certificado hasheado
        */
        uint32_t send(std::string& filename, Protocol& protocol);
        /*
        * Se reciben los datos del certificado por el socket.
        */
        void receive(Protocol& protocol);
        /*
         * Seter del serial number
        */
        void addSerial(uint32_t serial_number);
        /*
         * Se imprime el certificado con formato:
         *
         * certificate:
         *    serial number: n (hexa n 4 bytes)
         *    subject: <subject>
         *    issuer: Taller de programacion 1
         *    validity:
         *        not before: <MMM DD HH:mm:SS YYYY>
         *        not after: <MMM DD HH:mm:SS YYYY>
         *    subject public key info:
         *        modulus: n (hexa n 2 bytes)
         *        exponent: 19 (hexa n 1 byte)
        */
        std::string toString();
        std::string getSubject();
        Key getKey();
        friend std::ostream& operator<<(std::ostream &o, Certificate& self);
        //friend std::istream& operator>> (std::istream &in, Certificate& self);
};

#endif
```

```cpp
#include <string>
#include <sstream>
#include <iostream>
#include <fstream>
#include <ios>
#include "common_certificate.h"
#include "common_hash.h"
#define CERTIFICATE "certificate:\n"
#define SERIAL_NUMBER "\tserial number: "
#define SN_SIZE 4
#define SUBJECT "\tsubject: "
#define ISSUER "\tissuer: Taller de programacion 1\n"
#define VALIDITY "\tvalidity:\n"
#define NOT_BEFOR "\t\tnot before: "
#define NOT_AFTER "\t\tnot after: "
#define KEY "\tsubject public key info:\n"
#define MODULE "\t\tmodulus: "
#define EXPONENT "\t\texponent: "
Certificate::Certificate() {}

Certificate::Certificate(std::string& _subject, std::string& _not_before, \
                        std::string& _not_after, Key _key):
    key(_key) {
            subject = std::move(_subject);
            issuer = std::string(ISSUER);
            not_before = std::move(_not_before);
            not_after = std::move(_not_after);
}

void Certificate::send(Protocol& protocol) {
    try {
        protocol.send(this→serial_number);
        protocol.send(this→subject);
        protocol.send(this→not_before);
        protocol.send(this→not_after);
        this→key.send(protocol);
    }
    catch (std::runtime_error) {
        throw std::runtime_error("Error while sending certificate");
    }
}


void Certificate::receive(Protocol& protocol) {
    protocol.receive(this→serial_number);
    protocol.receive(this→subject);
    protocol.receive(this→not_before);
    protocol.receive(this→not_after);
    this→key.receive(protocol);
}

std::string toHexaString(int n, int len) {
    std::string hexa;
    hexa += std::string("(0x");

    std::stringstream stream;
    stream << std::hex << n;
    std::string number = stream.str();
    //2 digits per byte
    int to = len*2 − number.length();
    for (int i = 0; i < to; i++) {
        hexa += ’0’;
    }
    return hexa + number + ’)’;
}

```

```
std::string Certificate::getSubject() {
    return this→subject;
}

Key Certificate::getKey() {
    return this→key;
}

std::string Certificate::toString() {
    std::string o;
    o += std::string(CERTIFICATE);
    int n = (int) this→serial_number;
    std::string sn = toHexaString(n, SN_SIZE);
    o += SERIAL_NUMBER + std::to_string(n) + ' ' + sn + '\n';

    o += SUBJECT + this→subject + '\n';
    o += ISSUER;
    o += VALIDITY;
    o += NOT_BEFOR + this→not_before + '\n';
    o += NOT_AFTER + this→not_after + '\n';

    o += KEY;

    Key k = this→key;
    o += MODULE;
    k.printModule(o, toHexaString);
    o += "\n";
    o += EXPONENT;
    k.printPublicExponent(o, toHexaString);

    return o;
}

uint32_t Certificate::send(std::string& filename, Protocol& protocol) {
    std::ifstream file;
    file.open(filename);
    if (¬file.good()) {
        throw std::runtime_error("Error with certificate file");
    }
    std::string line;
    int count = 0;
    int pos;
    int len;
    std::string module;
    std::string exp;
    Hash hash;
    while (std::getline(file, line, '\n')) {
        if (¬file.eof()) {
            std::string aux = line + '\n';
            hash.load(aux);
        } else {
            hash.load(line);
        }
        pos = line.find(':');
        line = line.c_str();
        len = line.length();
        if (pos + 2 > len) { //certificate:\0
            ++count;
            continue;
        }
        line = line.substr(pos + 2,len);
        if (count ≡ 1) {
            len = line.find(' ');
            uint32_t n = (uint32_t) std::stoi(line.substr(0, len));
            try {
                protocol.send(n);
```

```
            }
            catch (std::runtime_error) {
                throw std::runtime_error("Error while sending certificate");
            }
        } else if ((count ≡ 2) | (count ≡ 5) | (count ≡ 6)) {
            try {
                protocol.send(line);
            }
            catch (std::runtime_error) {
                throw std::runtime_error("Error while sending certificate");
            }
        } else if (count ≡ 8) {
            len = line.find(' ');
            module = line.substr(0, len);
        } else if (count ≡ 9) {
            len = line.find(' ');
            exp = line.substr(0, len);
        }
        ++count;
    }
    Key key(exp, module);
    try {
        key.send(protocol);
    }
    catch (std::runtime_error) {
        throw std::runtime_error("Error while sending certificate");
    }
    return hash();
}

std::ostream& operator<<(std::ostream &o, Certificate& self) {
    std::string formal_certificate;
    formal_certificate = self.toString();
    o << formal_certificate;
    return o;
}

Certificate::~Certificate() {}

void Certificate::addSerial(uint32_t _serial_number) {
    this→serial_number = _serial_number;
}
```

```
#ifndef COMMON_TIME_H
#define COMMON_TIME_H
#include <time.h>
#include <chrono>
#include <string>

class Time {
    private:
        std::chrono::system_clock::time_point date;
    public:
        /*
         * Contenedor de tiempo que se inicializa con la fecha de su creaciÃ³n.
        */
        Time();
        //Time(std::string filename, std::string& to, std::string& from);

        /*
         * Le suma 60 dias a la fecha actual.
        */
        void validTo();
        /*
         * Almacena en str la fecha en formato MMM DD HH:mm:SS YYYY
        */
        void toString(std::string& str);
};

#endif
```

```
#include <ctime>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include "client_time.h"
#define BEGIN 4
#define DATE_LEN 27 //buffer which should have room for at least 26 bytes

Time::Time() {
  this→date = std::chrono::system_clock::now();
}

void Time::validTo() {
    // common_time.cpp:18:  Consider using ctime_r(...) instead of ctime(...)
    // for improved thread safety.  [runtime/threadsafe_fn] [2]
    std::chrono::duration<int,std::ratio<60*60*60*24> > month(1);
    this→date = this→date + month;
}

void Time::toString(std::string& str) {
    std::time_t t = std::chrono::system_clock::to_time_t(this→date);

    struct tm* time = 0; //âM−^@M−^XtimeâM−^@M−^Y is used uninitialized
    localtime_r(&t, time);
    std::stringstream ssTp;
  ssTp << std::put_time(time,"%b %d %T %Y");
    str = ssTp.str();
}


/*********************************** PUT_TIME *******************************
 * %b Abbreviated month name
 * %d Day of the month, zero−padded (01−31)
 * %T ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S
 * %Y Year complete number
 * str << std::put_time(this−>date,"%b %d %T %Y");
 *
 * => STR = MMM DD HH:mm:SS YYYY
*/
```

```cpp
#ifndef CLIENT_REVOKE_PROCESSOR_H
#define CLIENT_REVOKE_PROCESSOR_H
#include "common_protocol.h"
#include "client_processor.h"
#include <string>

class RevokeProcessor : Processor{
    private:
        //Protocol& protocol;

    public:
        explicit RevokeProcessor(Protocol& _protocol);
        ~RevokeProcessor();
        virtual void run(std::string& certificate_filename, \
             std::string& client_key_filename, \
             std::string& server_key_filename) override;
};

#endif
```

```cpp
#include <string>
#include <iostream>
#include "client_revoke_processor.h"
#include "client_invalid_request.h"
#include "common_certificate.h"
#include "common_rsa.h"
#define HASH_ERROR_SM 2
#define HASH_ERROR_MSSG "Error: los hashes no coinciden.\n"
#define INVALID_CERTIFICATE 1
#define INVALID_CERTIFICATE_MSSG "Error: usuario no registrado.\n"
#define HASH "Hash calculado: "
#define PRIVATE_ENCRYPTION "Hash encriptado con la clave privada: "
#define PUBLIC_ENCRYPTION "Huella enviada: "

RevokeProcessor::RevokeProcessor(Protocol& _protocol) : Processor(_protocol) {}
RevokeProcessor::~RevokeProcessor() {}

void RevokeProcessor::run(std::string& certificate_filename,\
         std::string& client_key_filename, std::string& server_key_filename) {
    uint8_t command = 1;
    try {
        this→protocol.send(command);
    }
    catch (std::runtime_error) {
        throw std::runtime_error("Error while sending command new");
    }

    Certificate certificate;
    Rsa rsa;
    try {
        Key server_key(server_key_filename);
        Key client_key(client_key_filename);
        rsa.set(server_key, client_key);
    }
    catch(...) {
        throw std::runtime_error("Error creating key");
    }
    uint32_t hash;

    try {
        hash = certificate.send(certificate_filename, protocol);
    }
    catch(...) {
        __throw_exception_again;
    }

    uint32_t priv_encryption = rsa.privateEncryption(hash);
    uint32_t publ_encryption = rsa.publicEncryption(priv_encryption);
    try {
        this→protocol.send(publ_encryption);
    }
    catch (std::runtime_error) {
        throw std::runtime_error("Error while sending encrypted hash");
    }
    uint8_t status = 0;
    this→protocol.receive(status);

    std::cout << HASH << (int) hash << '\n';
    std::cout << PRIVATE_ENCRYPTION << (int) priv_encryption << '\n';
    std::cout << PUBLIC_ENCRYPTION << (int) publ_encryption << '\n';
    if (status ≡ HASH_ERROR_SM) {
        throw InvalidRequest(HASH_ERROR_MSSG);
    }
    if (status ≡ INVALID_CERTIFICATE) {
        throw InvalidRequest(INVALID_CERTIFICATE_MSSG);
    }
    return;
}
```

```
#ifndef CLIENT_PROCESSOR_H_
#define CLIENT_PROCESSOR_H_
#include <string>
#include "common_protocol.h"

class Processor {
    protected:
        Protocol& protocol;
    public:
        explicit Processor(Protocol& _protocol);
        ~Processor();
        virtual void run(std::string& filename1, std::string& filename2, \
                                std::string& filename3) = 0;
};

#endif
```

```
#include "client_processor.h"

Processor::Processor(Protocol& _protocol) : protocol(_protocol) {}
Processor::~Processor() {}
```

```
#ifndef CLIENT_NEW_PROCESSOR_H
#define CLIENT_NEW_PROCESSOR_H
#include <string>
#include "common_protocol.h"
#include "client_processor.h"

class NewProcessor : Processor{
    private:
//          Protocol& protocol;
    public:
        explicit NewProcessor(Protocol& protocol);
        ~NewProcessor();
        virtual void run(std::string& certificate_information_filename, \
                        std::string& client_key_filename, \
                        std::string& server_key_filename) override;
};

#endif
```

```
#include <string>
#include <iostream>
#include <fstream>
#include "client_new_processor.h"
#include "client_applicant_request.h"
#include "common_socket.h"
#include "common_certificate.h"
#include "common_hash.h"
#include "common_rsa.h"
#include "client_invalid_request.h"
#define USER_ERROR_MSSG "Error: usuario no registrado.\n"
#define USER_ERROR 0
#define CERTIFICATE_ERROR_MSSG "Error: ya existe un certificado.\n"

#define CERIFICATE_ERROR_RECIVED_MSSG 0
#define HASH_ERROR_MSSG "Error: los hashes no coinciden.\n"
#define HASH_ERROR_SERVER_MSSG 1
#define HASH_OK_SERVER_MSSG 0
#define CERT_FP "Huella del servidor: "
#define SH "Hash del servidor: "
#define MH "Hash calculado: "


NewProcessor::NewProcessor(Protocol& _protocol) : Processor(_protocol) {}

NewProcessor::~NewProcessor() {}

void NewProcessor::run(std::string& certificate_information_filename, \
                        std::string& client_key_filename, \
                        std::string& server_key_filename) {
    ApplicantRequest request(certificate_information_filename,\
                                client_key_filename);
    try {
        request.send(protocol);
    }
    catch (std::runtime_error) {
        __throw_exception_again;
    }
    uint8_t answer = 1;
    protocol.receive(answer);
    if (answer ≡ CERIFICATE_ERROR_RECIVED_MSSG) {
        throw InvalidRequest(CERTIFICATE_ERROR_MSSG);
    }

    Certificate certificate;
    certificate.receive(protocol);
    std::string formal_certificate;
    formal_certificate = certificate.toString();


    Hash hash(formal_certificate);
    uint32_t my_hash =  hash();


    uint32_t certificate_footprint = 0;
    protocol.receive(certificate_footprint);

    Key client_key = request.getClientKey();
    Key server_key;
    try {
        server_key.set(server_key_filename);
    }
    catch(std::runtime_error) {
        throw std::runtime_error("Error seting server key");
    }
    Rsa rsa(server_key, client_key);
```

```
    uint32_t pd =  rsa.privateDesencryption(certificate_footprint);
    uint32_t server_hash = rsa.publicDesencryption(pd);

    std::cout << CERT_FP << certificate_footprint << '\n';
    std::cout << SH << server_hash << '\n';
    std::cout << MH << my_hash << '\n';

    uint8_t notification = HASH_OK_SERVER_MSSG;
    if (my_hash ≠ server_hash) {
        notification = HASH_ERROR_SERVER_MSSG;
        try {
            protocol.send(notification);
        }
        catch(std::runtime_error) {
            throw std::runtime_error(\
            "Error , the server could not be notified there was a hash error");
        }

        throw InvalidRequest(HASH_ERROR_MSSG);
    }

    try {
        protocol.send(notification);
    }
    catch(std::runtime_error) {
        throw std::runtime_error(\
        "Error , the server could not be notified there was no hash error");
    }

    std::string filename = request.getSubject() + ".cert";
    std::ofstream file;
    file.open(filename, std::ofstream::out | std::ofstream::trunc);

    if (¬file.is_open()) {
        //exc
    }
    file << formal_certificate;
    return;
}
```

may 10, 19 13:08 **client_invalid_request.h** Page 1/1

```
#ifndef CLIENT_INVALID_REQUEST_H
#define CLIENT_INVALID_REQUEST_H
#include <exception>
#include <string>

class InvalidRequest: public std::exception {
protected:
        std::string msg;
    public:
        explicit InvalidRequest(const char* message);

        explicit InvalidRequest(const std::string& message);

        virtual ~InvalidRequest() throw();

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

#endif
```

may 10, 19 13:08 **client_invalid_request.cpp** Page 1/1

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

InvalidRequest::InvalidRequest(const char* message):
      msg(message) {}

InvalidRequest::InvalidRequest(const std::string& message):
      msg(message) {}

InvalidRequest::~InvalidRequest() throw(){}

const char* InvalidRequest::what() const throw(){
    return msg.c_str();
}
```

```
#include <iostream>
#include <ostream>
#include <fstream>
#include <string>
#include "common_socket.h"
#include "client_new_processor.h"
#include "client_revoke_processor.h"
#include "client_invalid_request.h"
#define ARGUMENT_ERROR_MSSG "Error: argumentos invalidos.\n"
#define CF_SIZE 4 //certificate footprint size in bytes
#define HASH_SIZE 4


int main(int argc, char* argv[]) {
try {
    if (argc ≠ 7) {
        std::cout << ARGUMENT_ERROR_MSSG;
        return 0;
    }

    std::string mode1("new");
    std::string mode2("revoke");
    std::string mode = std::string(argv[3]);
    if (mode ≠ mode1 ∧ mode ≠ mode2) {
        std::cout << ARGUMENT_ERROR_MSSG; //no es un throw
                                          //porque no hay donde catchearlo
        return 0;
    }

    char* host = argv[1];
    char* port = argv[2];
    Socket skt;
    skt.connectWithServer(host, port);

    Protocol protocol(skt);


    std::string client_key_filename = std::string(argv[5]);
    std::string server_key_filename = std::string(argv[6]);


    if (mode ≡ mode1) {
        std::string certificate_information_filename(argv[4]);
        NewProcessor np(protocol);
        np.run(certificate_information_filename, \
                    client_key_filename, \
                    server_key_filename);
        return 0;
    }

    if (mode ≡ mode2) { //std::string(argv[4])
        std::string certificate_filename = std::string(argv[4]);
        RevokeProcessor rp(protocol);
        rp.run(certificate_filename, \
                    client_key_filename, \
                     server_key_filename);
        return 0;
    }
    return 0;
}
catch(InvalidRequest &e) {
    std::cout << e.what();
}
catch(std::runtime_error &e) {
    std::cerr << e.what() << std::endl;
}
```

```
}
```

```
#ifndef CLIENT_CERTIFICATE_INFO_PARSER_H
#define CLIENT_CERTIFICATE_INFO_PARSER_H
#include <string>

/******************************************************************************
 * Lee una rchivo de tipo
 *   <subject>\n<date from>\n<date to>
 * Donde las fechas son de tipo MMM DD HH:mm:SS YYYY
 * Y almacena los atributos leidos en los repectivos string pasados por parame−
 * tro.
 * Si las fechas no existen from sera la fecha actual y to 30 dias despues.
*/

class CertificateInfoParser {
    public:
        CertificateInfoParser();
        void run(std::string& filename, std::string& subject,\
                                 std::string& from, std::string& to);
        ~CertificateInfoParser();
};

#endif
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include "client_certificate_info_parser.h"
#include "client_time.h"

void CertificateInfoParser::run(std::string& filename, \
                                std::string& subject, std::string& from,\
                                std::string& to) {
    std::ifstream file;
    file.open(filename);
    if (¬file.good()) {
        throw std::runtime_error("Error with file while parsing certificate");
    }
    std::string line;
    int i = 0;
    while (std::getline(file, line, '\n')) {
        if (i ≡ 0) {
            subject = line.c_str();
        } else if (i ≡ 1) {
            from = line.c_str();
        } else if (i ≡ 2) {
            to = line.c_str();
        }
        ++i;
    }
    if (i ≡ 1) {
        Time tm;
        tm.toString(from);
        tm.validTo();
        tm.toString(to);
    }
}

CertificateInfoParser::CertificateInfoParser() {}
CertificateInfoParser::~CertificateInfoParser() {}
```

```
#ifndef CLIENT_APPLICANT_REQUEST_H
#define CLIENT_APPLICANT_REQUEST_H
#include <stdint.h>
#include <string>
#include "common_key.h"
#include "common_protocol.h"


class ApplicantRequest {
    private:
        std::string subject;
        Key key;
        std::string date_from;
        std::string date_to;

    public:
        /*
        * Recibe los dos archivos necesarios para solicitar un nuevo aplicante
        */
        ApplicantRequest(std::string& cert_filename, std::string& key_filename);

        ~ApplicantRequest();
        /*
         * Envia a traves del socket recibido por parametro una solucitud con
         * formato:
         *
         * <comando>        1 byte con valor 0.
         * <subject__size>  4 bytes big endian sin signo
         * <subject>        String sin âM−^@M−^X\0âM−^@M−^Y
         * <modul>          2 bytes big endian sin signo
         * <exponent>       1 byte
         * <date__size>     4 bytes big endian sin signo
         * <date_from>      String sin âM−^@M−^X\0âM−^@M−^Y
         * <date__size>     4 bytes big endian sin signo
         * <dat_to>         String sin âM−^@M−^X\0âM−^@M−^Y
        */
        void send(Protocol& protocol);
        Key getClientKey();
        std::string getSubject();
};

#endif
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include "client_applicant_request.h"
#include "client_certificate_info_parser.h"
#define COMMAND_SIZE 1


ApplicantRequest::ApplicantRequest(std::string& cert_filename,\
                                 std::string& key_filename) {
        CertificateInfoParser cir;
        try {
            this→key.set(key_filename);
            cir.run(cert_filename, this→subject, \
                       this→date_from, this→date_to);
        }
        catch(...) {
            throw std::runtime_error("Error creating aplicant request");
        }
}

ApplicantRequest::~ApplicantRequest() {}

void ApplicantRequest::send(Protocol& protocol) {
    uint8_t command = 0;
    try {
        protocol.send(command);
        protocol.send(this→subject);
        key.send(protocol);
        protocol.send(this→date_from);
        protocol.send(this→date_to);
    }
    catch (std::runtime_error) {
        throw std::runtime_error("Error while sending aplicant request");
    }
}

std::string ApplicantRequest::getSubject(){
    return this→subject;
}

Key ApplicantRequest::getClientKey() {
    return this→key;
}
```

Table of Contents