

may 15, 18 16:13

serverThConnection.h

Page 1/1

```

1  #ifndef THCONNECTION_H_
2  #define THCONNECTION_H_
3
4  #include "commonThread.h"
5  #include "serverConnection.h"
6
7
8  class ThConnection : public Thread {
9      private:
10         Connection connection;
11         bool dead;
12
13     public:
14         bool is_dead();
15         ThConnection(Socket peer, Index & index);
16         virtual void run();
17         void stop();
18     };
19 #endif

```

may 15, 18 16:13

serverThConnection.cpp

Page 1/1

```

1  #include "serverThConnection.h"
2
3  void Thread::start() {
4      thread = std::thread(&Thread::run, this);
5  }
6
7  void Thread::join() {
8      thread.join();
9  }
10
11  ThConnection::ThConnection(Socket peer, Index & index)://
12      connection(std::move(peer), index){}
13
14  bool ThConnection::is_dead() {
15      return this->dead;
16  }
17  void ThConnection::run() {
18      dead = false;
19      this->connection.run();
20      dead = true;
21  }
22
23  void ThConnection::stop() {
24      this->connection.stop();
25  }
26
27
28
29
30
31
32

```

may 15, 18 16:13

serverLock.h

Page 1/1

```
1 #include <mutex>
2
3 class Lock {
4 private:
5     std::mutex &m;
6
7 public:
8     explicit Lock(std::mutex &m);
9
10    ~Lock();
11
12 private:
13     Lock(const Lock&) = delete;
14     Lock& operator=(const Lock&) = delete;
15     Lock(Lock^ ) = delete;
16     Lock& operator=(Lock^ ) = delete;
17 };
```

may 15, 18 16:13

serverLock.cpp

Page 1/1

```
1 #include "serverLock.h"
2
3
4 Lock::Lock(std::mutex &m) : m(m) {
5     m.lock();
6 }
7
8 Lock::~~Lock() {
9     m.unlock();
10 }
11
```

may 15, 18 16:13

serverListener.h

Page 1/1

```

1 #include "serverConnection.h"
2 #include "serverThConnection.h"
3 #include <thread>
4 #include <vector>
5
6 class Server_listener{
7 private:
8     Socket socket;
9     Index index;
10    std::thread principal_thread;
11    std::vector<ThConnection*> client_threads;
12    bool continue_listening;
13    void listen();
14 public:
15    //constructor que crea un socket de servidor
16    Server_listener(char port[], char file_name[]);
17
18    //comienza a escuchar conecciones
19    void start_listening();
20
21    //finaliza las conecciones
22    void stop_listening();
23 };

```

may 15, 18 16:13

serverListener.cpp

Page 1/1

```

1 #include "serverListener.h"
2 #include <vector>
3 Server_listener::Server_listener(char port[], char file_name[])://
4     socket(NULL,port), index(file_name){
5     this->continue_listening = true;
6     this->socket.bind_and_listen();
7 }
8
9 void Server_listener::start_listening(){
10    this->principal_thread = std::thread(&Server_listener::listen, this);
11 }
12
13 void Server_listener::listen(){
14    while(this->continue_listening){
15        try{
16            Socket peer = this->socket.accept_socket();
17            client_threads.push_back(new ThConnection(std::move(peer), this->index));
18            client_threads.back()->start();
19        } catch(Error e){
20            //cerraron el socket
21        }
22    }
23 }
24
25 void Server_listener::stop_listening(){
26    this->continue_listening = false;
27
28    this->socket.stop();
29
30    for(unsigned int i = 0; i < client_threads.size(); i++){
31        client_threads[i]->stop();
32        client_threads[i]->join();
33        delete client_threads[i];
34    }
35
36    this->principal_thread.join();
37    this->index.close();
38 }

```

may 15, 18 16:13

serverIndex.h

Page 1/2

```

1  #ifndef __INDEX_H__
2  #define __INDEX_H__
3
4  #define FILE_TYPE "f"
5  #define TAG_TYPE "t"
6
7  #include <fstream>
8  #include <string>
9  #include <sstream>
10 #include <algorithm>
11 #include <map>
12 #include <set>
13 #include <mutex>
14 #include "commonError.h"
15 #include "serverLock.h"
16 using std::cout;
17 using std::ios;
18
19
20 struct cmp{
21     bool operator()(const std::string& s1, const std::string& s2) const{
22         for (std::string::const_iterator it1 = s1.begin(), it2 = s2.begin();//
23              it1!=s1.end() ^ it2!=s2.end(); ++it1, ++it2 ){
24             if ((*it1) == (*it2)){
25                 continue;
26             }
27             if (std::isdigit(*it1) ^ !std::isdigit(*it2)){
28                 return true;
29             }
30             if (!std::isdigit(*it1) ^ std::isdigit(*it2)){
31                 return false;
32             }
33             if (std::isdigit(*it1) ^ std::isdigit(*it2)) {
34                 int n1 = (*it1) -48;
35                 int n2 = (*it2) -48;
36                 return n1 < n2;
37             }
38         }
39         return s1 < s2;
40     }
41 };
42
43 class Index{
44     private:
45         std::string index_file_name;
46         std::mutex file_mutex;
47         std::mutex tag_mutex;
48         std::map<std::string, std::set<std::string, cmp>> files;
49         std::map<std::string, std::set<std::string, cmp>> tags;
50
51     bool numeric_string_compare(const std::string& s1, const std::string& s2);
52     //recibe un mapa un caracter de tipo y escribe en el archivo
53     //de indice los datos del mapa con el formato correspondiente
54     void write_index_file(std::map<std::string, //
55                          std::set<std::string, cmp>> & map, //
56                          const std::string & type);
57
58     public:
59         //abre el archivo index_file_name y crea un mapa de files
60         // y otro de tags, en los cuales la key es el nombre y
61         // el value es un set de hashes
62         explicit Index(const std::string & index_file_name);
63
64
65         //devuelve true si el hash existe dentro de algun set de hashes
66         //de alguno de los archivos

```

may 15, 18 16:13

serverIndex.h

Page 2/2

```

67     bool does_hash_exist(const std::string & hash);
68
69     //devuelve el nombre del archivo que tiene ese hash
70     //dentro de su set de hashes
71     std::string get_file_name(const std::string & hash);
72
73     //agrega el nuevo hash para ese archivo
74     //en el mapa de archivos
75     //(si el file name no existia en el mapa lo agtrga)
76     void add_file_hash(const std::string & file_name, const std::string & hash);
77
78     //devuelve un set de archivos asociados a ese tag
79     //si no existe devolvera un ser vacio
80     std::set<std::string, cmp> get_tag_files(const std::string & tag_name);
81
82
83     //agrega el nuevo hash para ese tag
84     //en el mapa de tags
85     //(si el tag no existia en el mapa lo agtrga)
86     void add_tag_hash(const std::string & tag_name, const std::string & hash);
87
88
89     //Sobre escribe el archivo llamado index_file_name
90     //con el mapa de files y el mapa de tags.
91     //la version inicial se pierde.
92     void close();
93 };
94
95 #endif
96

```

may 15, 18 16:13

serverIndex.cpp

Page 1/3

```

1  #include "serverIndex.h"
2  #include <map>
3  #include <string>
4  #include <set>
5
6
7  Index::Index(const std::string & index_file_name){
8      this->index_file_name = index_file_name;
9      std::ifstream index_file(this->index_file_name);
10     if (!index_file.is_open()){
11         throw Error("Unable to open index file %s\n", //
12             this->index_file_name.c_str());
13     }
14     std::string line;
15     std::string type;
16     std::string name;
17     std::string hash;
18     while (std::getline(index_file, line, ',')){
19         line.erase(std::remove(line.begin(), line.end(), '\n'), line.end());
20         std::istringstream linereader(line, std::ios::binary);
21         std::map<std::string, std::set<std::string, cmp>> * map;
22         if (!std::getline(linereader, type, ',')){
23             break;
24         }
25         if (type.compare(FILE_TYPE) == 0){
26             map = &files;
27         } else if (type.compare(TAG_TYPE) == 0){
28             map = &tags;
29         } else{
30             throw Error("reading input file: " //
31                 "%s is not valid type\n" //
32                 "shuld be %s for files and %s for tags\n", //
33                 type.c_str(), FILE_TYPE, TAG_TYPE);
34         }
35         std::getline(linereader, name, ',');
36         std::set<std::string, cmp> hashes;
37
38         while (std::getline(linereader, hash, ',')){
39             hashes.insert(hash);
40         }
41
42         (*map)[name] = hashes;
43     }
44 }
45
46
47 void Index::add_file_hash(const std::string & file_name, //
48     const std::string & hash){
49     Lock l(file_mutex);
50     std::set<std::string, cmp> hashes;
51
52     std::map<std::string, std::set<std::string, cmp>>::iterator //
53     it = files.find(file_name);
54     if (it != files.end()){
55         hashes = it->second;
56     }
57     hashes.insert(hash);
58
59     files[file_name] = hashes;
60 }
61
62 bool Index::does_hash_exist(const std::string & hash){
63     Lock l(file_mutex);
64     for (std::map<std::string, std::set<std::string, cmp>>::iterator //
65         map_iter=files.begin(); map_iter!=files.end(); ++map_iter){
66         if (map_iter->second.find(hash) != map_iter->second.end()){

```

may 15, 18 16:13

serverIndex.cpp

Page 2/3

```

67         return true;
68     }
69 }
70     return false;
71 }
72
73
74 std::set<std::string, cmp> Index::get_tag_files(const std::string & tag_name){
75     Lock l(tag_mutex);
76     std::map<std::string, std::set<std::string, cmp>>::iterator //
77     it = tags.find(tag_name);
78     if (it == tags.end()){
79         std::set<std::string, cmp> empty_set;
80         return empty_set;
81     }
82     return it->second;
83 }
84
85 std::string Index::get_file_name(const std::string & hash){
86     Lock l(file_mutex);
87     for (std::map<std::string, std::set<std::string, cmp>>::iterator //
88         map_iter=files.begin(); map_iter!=files.end(); ++map_iter){
89         if (map_iter->second.find(hash) != map_iter->second.end()){
90             return map_iter->first;
91         }
92     }
93     throw Error("invalid hash");
94 }
95
96 void Index::add_tag_hash(const std::string & tag_name, //
97     const std::string & hash){
98     Lock l(tag_mutex);
99     std::set<std::string, cmp> hashes;
100
101     std::map<std::string, std::set<std::string, cmp>>::iterator //
102     it = tags.find(tag_name);
103     if (it != tags.end()){
104         hashes = it->second;
105     }
106     hashes.insert(hash);
107
108     tags[tag_name] = hashes;
109 }
110
111
112
113 void Index::write_index_file(std::map<std::string, //
114     std::set<std::string, cmp>> & map, //
115     const std::string & type){
116     std::ofstream index_file(this->index_file_name, ios::out | ios::app);
117
118     if (!index_file.is_open()){
119         throw Error("Unable to open file %s\n", this->index_file_name);
120     }
121     for (std::map<std::string, std::set<std::string, cmp>>::iterator //
122         map_iter=map.begin(); map_iter!=map.end(); ++map_iter){
123         index_file << type << " " << (*map_iter).first;
124
125         for (std::set<std::string>::iterator //
126             set_iter=(*map_iter).second.begin(); //
127             set_iter!=(*map_iter).second.end(); //
128             ++set_iter){
129             index_file << " " << (*set_iter);
130         }
131         index_file << "\n";
132     }

```

may 15, 18 16:13

serverIndex.cpp

Page 3/3

```

133     index_file.close();
134 }
135
136
137 void Index::close(){
138     std::ofstream index(this->index_file_name,ios::out | ios::trunc);
139     index.close();
140
141     write_index_file(files, "f");
142     write_index_file(tags, "t");
143 }

```

may 15, 18 16:13

serverConnection.h

Page 1/1

```

1  #ifndef __SERVER_H__
2  #define __SERVER_H__
3
4  #include "serverIndex.h"
5  #include "commonError.h"
6  #include "commonProtocol.h"
7  #include <string>
8
9
10 using std::cout;
11 using std::endl;
12 using std::string;
13
14 class Connection: Protocol{
15     Index & index;
16     void push();
17     void tag();
18     void pull();
19 public:
20     Connection(Socket peer, Index & index);
21     void run();
22     void stop();
23 };
24
25 #endif

```

may 15, 18 16:13

serverConnection.cpp

Page 1/2

```

1  #include "serverConnection.h"
2  #include <set>
3  #include <string>
4
5
6  Connection::Connection(Socket peer, Index & index)://
7  Protocol(std::move(peer)), index(index){}
8
9
10 void Connection::run(){
11     int code = receive_code();
12     switch(code){
13         case 1:
14             push();
15             break;
16         case 2:
17             tag();
18             break;
19         case 3:
20             pull();
21             break;
22         throw Error("codigo %i no valido", code);
23     }
24 }
25
26 void Connection::push(){
27     string file_name = receive_string();
28     string hash = receive_string();
29     if((this->index).does_hash_exist(hash)){
30         send_code(NOT_OK);
31     }else {
32         send_code(OK);
33         if(receive_file(hash)){
34             this->index.add_file_hash(file_name, hash);
35         }
36     }
37 }
38
39 void Connection::tag(){
40     ssize_t hashes_size = receive_size_first();
41     string tag_name = receive_string();
42     bool already_exist = (this->index).get_tag_files(tag_name).size() > 0;
43     bool invalid_hash = false;
44     for (int i = 0; i < hashes_size; i++){
45         string hash = receive_string();
46         if (!(this->index).does_hash_exist(hash)){
47             invalid_hash = true;
48         } else {
49             (this->index).add_tag_hash(tag_name, hash);
50         }
51     }
52     if (already_exist ∨ hashes_size ≤ 0 ∨ invalid_hash){
53         send_code(NOT_OK);
54         return;
55     }
56     send_code(OK);
57 }
58
59 void Connection::pull(){
60     string tag_name = receive_string();
61     std::set<std::string,cmp> tag_files = (this->index).get_tag_files(tag_name);
62     int size_of_files = tag_files.size();
63     if(size_of_files == 0){
64         send_code(NOT_OK);
65     }else {
66         send_code(OK);

```

may 15, 18 16:13

serverConnection.cpp

Page 2/2

```

67     send_size_first(size_of_files);
68
69     for (std::set<std::string>::iterator//
70         set_iter=tag_files.begin();//
71         set_iter#tag_files.end();//
72         ++set_iter){
73         string hash = (*set_iter);
74         string name = this->index.get_file_name(hash);
75         send_string(name+"."+tag_name);
76         send_file(hash);
77     }
78 }
79
80
81 void Connection::stop(){
82     (this->socket).stop();
83 }
84
85
86

```

may 15, 18 16:13

serverApp.cpp

Page 1/1

```

1
2 #define PORT 1
3 #define INDEX_FILE 2
4
5 #include "serverListener.h"
6 #include <thread>
7
8
9 int main(int argc, char * argv[]){
10     if(argc < 2){
11         throw Error("Parametros incorrectos");
12     }
13
14     Server_listener s(argv[PORT],argv[INDEX_FILE]);
15     s.start_listening();
16
17     char c = getchar();
18     while(c != 'q'){
19         c = getchar();
20     }
21
22     s.stop_listening();
23
24     return 0;
25 }

```

may 15, 18 16:13

commonThread.h

Page 1/1

```

1 #ifndef THREAD_H_
2 #define THREAD_H_
3
4 #include <thread>
5
6 class Thread {
7     private:
8         std::thread thread;
9
10    public:
11        Thread() {}
12
13        void start();
14
15        void join();
16
17        virtual void run() = 0;
18        virtual ~Thread() {}
19
20        Thread(const Thread&) = delete;
21        Thread& operator=(const Thread&) = delete;
22
23        Thread(Thread^ other) {
24            this->thread = std::move(other.thread);
25        }
26
27        Thread& operator=(Thread^ other) {
28            this->thread = std::move(other.thread);
29            return *this;
30        }
31    };
32 #endif

```


may 15, 18 16:13

commonSocket.h

Page 1/2

```

1  #ifndef __SOCKET_H__
2  #define __SOCKET_H__
3
4  #include <iostream>
5  #include <stdbool.h>
6  #include <stddef.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <netdb.h>
10 #include <unistd.h>
11 #include <unistd.h>
12 #include <fcntl.h>
13 #include "commonError.h"
14
15
16 #define SUCCESS 0
17 #define ERROR 1
18
19 class Socket {
20     private:
21         int socket_num;
22         const char * host_name;
23         const char * port;
24         bool is_connected;
25
26         //data una estructura del tipo addrinfo, la recorre hasta
27         //poder crear un socket y setea entonces el numero del socket
28         struct addrinfo * define_socket_num(struct addrinfo * node);
29
30         //crea un socket inicial
31         void init();
32
33         //devuelve el primer nodo de addrinfo
34         struct addrinfo * addrinfo();
35
36         //valida que el puerto sea valido
37         bool is_valid_port(const char * port);
38
39
40
41     public:
42         //Constructor por copia anulado
43         Socket(const Socket& other) = delete;
44         //Asignacion por copia anulada
45         Socket& operator=(const Socket &other) = delete;
46         //Constructor por movimiento anulado
47         Socket(Socket& other);
48         //Asignacion por movimiento anulada
49         Socket& operator=(Socket& other);
50
51         //constructor del socket
52         Socket(const char * host_name, const char * port);
53
54         //constructor del peer socket ya conectado
55         explicit Socket(int socket_num);
56
57         //trata de conectar al socket
58         void connection();
59
60         //sirve para servidores,
61         //el socket se quedara esperando que se conecten con el
62         void bind_and_listen();
63
64         //sirve para servidores,
65         //acepta a otro socket que quiera conectarse con el
66         Socket accept_socket();

```

may 15, 18 16:13

commonSocket.h

Page 2/2

```

67
68         //Debe estar conectado con el socket cuyo numero sea skt_num
69         //porque a partir de ahí recibirá datos
70         //Tratará de recibir tantos datos como se especifique
71         //en el parametro size.
72         //guarda los datos recibidos en buffer que debe ser un array
73         //de un tamaño mayor o igual a size
74         int receive_message(char* buffer, size_t size);
75
76
77
78         //Tratará de enviar tantos datos como se especifique
79         //en el parámetro size.
80         //enciará los datos en buffer que debe ser un array
81         //de un tamaño mayor o igual a size
82         int send_message(const char* buffer, size_t size);
83
84         void stop();
85
86         //destructor del socket
87         ~Socket();
88     };
89
90 #endif

```

may 15, 18 16:13

commonSocket.cpp

Page 1/4

```

1  #include "commonSocket.h"
2
3
4  struct addrinfo * Socket::define_socket_num(struct addrinfo * node){
5      while (node != NULL) {
6          socket_num = //
7              socket(node->ai_family, //
8                    node->ai_socktype, //
9                    node->ai_protocol);
10         if (socket_num != -1) {
11             this->socket_num = socket_num;
12             return node;
13         }
14         node = node->ai_next;
15     }
16     return NULL;
17 }
18
19
20
21 void Socket::init(){
22     struct addrinfo *addrinfoNode = addrinfo();
23     if(define_socket_num(addrinfoNode) == NULL){
24         throw Error("Error en init\n");
25     }
26     free(addrinfoNode);
27 }
28
29 struct addrinfo * Socket::addrinfo(){
30     struct addrinfo hints;
31     memset(&hints, 0, sizeof(struct addrinfo));
32     hints.ai_family = AF_INET; //IPv4
33     hints.ai_socktype = SOCK_STREAM; //TCP
34     hints.ai_flags = 0;
35
36     struct addrinfo *addrinfoNode;
37
38
39     int addrinfo_returnvalue = //
40         getaddrinfo(this->host_name, this->port, &hints, &addrinfoNode);
41
42     if (addrinfo_returnvalue != SUCCESS){
43         gai_strerror(addrinfo_returnvalue);
44     }
45     return addrinfoNode;
46 }
47
48 bool Socket::is_valid_port(const char * port){
49     for (unsigned int i = 0; i < strlen(port); i++){
50         if (!isdigit(port[i])){
51             return false;
52         }
53     }
54     return true;
55 }
56
57
58
59
60 Socket::Socket(const char * host_name, const char * port){
61     if(!is_valid_port(port)){
62         throw Error("%s no es un puerto valido\n"//
63                   "deben ser todos caracteres numÃ©ricos", port);
64     }
65
66     this->host_name = host_name;

```

may 15, 18 16:13

commonSocket.cpp

Page 2/4

```

67     this->port = port;
68     this->is_connected = false;
69
70     init();
71     int val = 1;
72     if (setsockopt(this->socket_num, SOL_SOCKET, SO_REUSEADDR, &val, //
73                 sizeof(val)) == -1) {
74         close(this->socket_num);
75         throw Error("Error in reuse socket: %s\n", strerror(errno));
76     }
77 }
78
79 Socket::Socket(int socket_num){
80     this->is_connected = true;
81     this->socket_num = socket_num;
82 }
83
84
85
86 Socket::Socket(Socket& other){
87     this->socket_num = other.socket_num;
88     this->host_name = other.host_name;
89     this->port = other.port;
90     this->is_connected = other.is_connected;
91
92     other.socket_num = -1;
93     other.host_name = "null";
94     other.port = "null";
95     other.is_connected = false;
96 }
97
98 Socket& Socket::operator=(Socket& other){
99     if (this == &other) {
100         return *this; // other is myself!
101     }
102     this->socket_num = other.socket_num;
103     this->host_name = other.host_name;
104     this->port = other.port;
105     this->is_connected = other.is_connected;
106
107     other.socket_num = -1;
108     other.host_name = "null";
109     other.port = "null";
110     other.is_connected = false;
111     return *this;
112 }
113
114
115 void Socket::connection(){
116     struct addrinfo *addrinfoNode = addrinfo();
117
118     struct addrinfo *node = define_socket_num(addrinfoNode);
119
120     while (node != NULL ^ this->is_connected == false) {
121         int connection_returnvalue = connect(this->socket_num, //
122                                             node->ai_addr, node->ai_addrlen);
123         this->is_connected = (connection_returnvalue != -1);
124         if (this->is_connected){
125             break;
126         }
127         node = define_socket_num(addrinfoNode);
128     }
129     freeaddrinfo(addrinfoNode);
130     if (this->is_connected == false) {
131         close(this->socket_num);
132         throw Error("Error in socket connection: %s\n", strerror(errno));

```

may 15, 18 16:13

commonSocket.cpp

Page 3/4

```

133 }
134 }
135
136 void Socket::bind_and_listen(){
137     struct addrinfo *addrinfoNode = addrinfo();
138
139     int bindReturnValue = -1;
140     while (addrinfoNode != NULL) {
141         bindReturnValue = bind(this->socket_num, //
142             addrinfoNode->ai_addr, //
143             addrinfoNode->ai_addrlen);
144         if (bindReturnValue != -1) {
145             break;
146         }
147         addrinfoNode = addrinfoNode->ai_next;
148     }
149     free(addrinfoNode);
150
151     if (bindReturnValue == -1) {
152         throw Error("Error in bind: %s\n", strerror(errno));
153     }
154     int listenReturnValue = listen(this->socket_num, 20);
155     if (listenReturnValue == -1) {
156         throw Error("Error in listen: %s\n", strerror(errno));
157     }
158     this->is_connected = true;
159 }
160
161 Socket Socket::accept_socket(){
162     int new_sockfd = accept(this->socket_num, NULL, NULL);
163     if (new_sockfd == -1){
164         throw Error("Error in accept");
165     }
166     return std::move(Socket(new_sockfd));
167 }
168
169 int Socket::receive_message(char* buffer, const size_t size){
170     int total_received = 0;
171     int bytes_recived = 0;
172
173     while ((bytes_recived = recv(this->socket_num, //
174         &buffer[total_received], //
175         size - total_received, MSG_NOSIGNAL)) > 0) {
176         total_received += bytes_recived;
177         if (size - total_received == 0){
178             break;
179         }
180     }
181     if (bytes_recived < 0) {
182         throw Error("Error recibing info: %s\n", strerror(errno));
183     }
184     return total_received;
185 }
186
187
188 int Socket::send_message(const char* buffer, const size_t size){
189     int total_sent = 0;
190     int bytes_sent = 0;
191
192     while ((bytes_sent = send(this->socket_num, //
193         &buffer[total_sent], size - total_sent, MSG_NOSIGNAL)) > 0){
194         total_sent += bytes_sent;
195     }
196     if (bytes_sent < 0) {
197         throw Error("Error sending info: %s\n", strerror(errno));
198     }

```

may 15, 18 16:13

commonSocket.cpp

Page 4/4

```

199     }
200     return total_sent;
201 }
202
203 void Socket::stop(){
204     shutdown(this->socket_num, SHUT_RDWR);
205     close(this->socket_num);
206 }
207
208 Socket::~Socket(){
209     if (this->socket_num > 0){
210         shutdown(this->socket_num, SHUT_RDWR);
211         close(this->socket_num);
212     }
213 }
214

```

may 15, 18 16:13

commonProtocol.h

Page 1/2

```

1  #ifndef __PROTOCOL_H__
2  #define __PROTOCOL_H__
3
4
5  #include "commonSocket.h"
6
7
8  #include <string>
9  #include <stdio.h>
10 #include <iostream>
11 #include <fstream>
12
13
14 using std::cout;
15 using std::endl;
16 using std::string;
17
18
19 #define CHUNK_LEN 64
20
21
22 #define PROTOCOL_MSG_SIZE 4
23 #define PROTOCOL_CODE 2
24 #define NO_PEER_SKT -1
25
26 #define PUSH_CODE 1
27 #define TAG_CODE 2
28 #define PULL_CODE 3
29
30 #define OK 1
31 #define NOT_OK 0
32
33
34
35 class Protocol{
36 private:
37 int get_digits(unsigned int num){
38     int digits = 1;
39     while ( num > 0 ) {
40         num /= 10;
41         digits++;
42     }
43     return digits;
44 }
45
46
47
48 protected:
49     Socket socket;
50     explicit Protocol(Socket socket);
51
52
53     //envia en un solo byte un numero
54     void send_code(int code_to_send);
55
56     //envia un string
57     //siempre antes enviando el tamaño
58     void send_string(const string & string_to_send);
59
60     //envia un archivo de a chunks
61     //siempre antes enviando el tamaño total
62     void send_file(const std::string & file_name);
63
64     //envia 4 bytes con el tamaño
65     void send_size_first(unsigned int size);
66

```

may 15, 18 16:13

commonProtocol.h

Page 2/2

```

67
68     //recibe un solo byte con un numero
69     int receive_code();
70
71     //recibe primero la longitud y luego un string
72     std::string receive_string();
73
74     //recibe primero la longitud total y
75     //luego el archivo de a chunks
76     bool receive_file(const std::string & hash);
77
78
79     //recibe el tamaño en 4 bytes
80     ssize_t receive_size_first();
81 };
82
83 #endif
84

```

may 15, 18 16:13

commonProtocol.cpp

Page 1/2

```

1  #include "commonProtocol.h"
2  #include <string>
3
4  Protocol::Protocol(Socket socket):socket(std::move(socket)) {}
5
6  void Protocol::send_code(int code_to_send){
7      char code[PROTOCOL_CODE];
8      snprintf(code,PROTOCOL_CODE, "%d", code_to_send);
9      (this->socket).send_message(code,PROTOCOL_CODE);
10 }
11
12
13 void Protocol::send_string(const string & string_to_send){
14     int size = string_to_send.size();
15     send_size_first(size);
16     (this->socket).send_message(string_to_send.c_str(),size);
17 }
18
19
20 void Protocol::send_file(const std::string & file_name){
21     std::ifstream file(file_name,std::ifstream::binary);
22     if (file.fail()){
23         send_code(NOT_OK);
24         throw Error("No se encontro el archivo %s\n", file_name.c_str());
25     }
26     send_code(OK);
27     file.seekg(0, file.end);
28     int file_len = file.tellg();
29     file.seekg(0, file.beg);
30     send_size_first(file_len);
31
32     int bytes_sent = 0;
33     int total_sent = 0;
34     char request[CHUNK_LEN+1];
35
36     while (total_sent < file_len ^ ~file.eof()){
37         memset(request, 0, CHUNK_LEN+1);
38         file.read(request, CHUNK_LEN);
39
40         int request_len = CHUNK_LEN;
41         if (file_len - total_sent < CHUNK_LEN){
42             request_len = file_len - total_sent;
43         }
44         bytes_sent = (this->socket).send_message(request, request_len);
45
46         if (bytes_sent < 0){
47             break;
48         }
49         total_sent += bytes_sent;
50     }
51 }
52
53 void Protocol::send_size_first(unsigned int size){
54     int digitos = get_digits(size);
55     if(digitos > PROTOCOL_MSG_SIZE){
56         throw Error("Mensaje demasiado largo!");
57     }
58     char msg_size[PROTOCOL_MSG_SIZE];
59     memset(msg_size, 0, PROTOCOL_MSG_SIZE);
60     snprintf(msg_size,PROTOCOL_MSG_SIZE, "%d", size);
61     (this->socket).send_message(msg_size,PROTOCOL_MSG_SIZE);
62 }
63
64
65 int Protocol::receive_code(){
66     char code[PROTOCOL_CODE];

```

may 15, 18 16:13

commonProtocol.cpp

Page 2/2

```

67     (this->socket).receive_message(code, PROTOCOL_CODE);
68     return atoi(code);
69 }
70
71
72
73 std::string Protocol::receive_string(){
74     ssize_t msg_size = receive_size_first();
75     char chunk[CHUNK_LEN];
76     (this->socket).receive_message(chunk, msg_size);
77     chunk[msg_size] = 0;
78     std::string string_received(chunk);
79     return string_received;
80 }
81
82
83 bool Protocol::receive_file(const std::string & name){
84     if (receive_code() == NOT_OK){
85         return false;
86     }
87     std::ofstream file(name,std::ofstream::out| std::ofstream::binary);
88     ssize_t file_len = receive_size_first();
89     char chunk[CHUNK_LEN+1];
90     int total_received = 0;
91     int bytes_received = 0;
92     while (total_received < file_len){
93         memset(chunk, 0, CHUNK_LEN+1);
94
95         int request_len = CHUNK_LEN;
96         if (file_len - total_received < CHUNK_LEN){
97             request_len = file_len - total_received;
98         }
99         bytes_received = (this->socket).receive_message(chunk, request_len);
100
101         total_received +=bytes_received;
102         if (bytes_received ≤ 0){
103             break;
104         }
105         file.write(chunk, bytes_received);
106     }
107     return true;
108 }
109
110 ssize_t Protocol::receive_size_first(){
111     char msg_size[PROTOCOL_MSG_SIZE];
112     (this->socket).receive_message(msg_size, PROTOCOL_MSG_SIZE);
113     return atoi(msg_size);
114 }
115
116
117

```

may 15, 18 16:13

commonError.h

Page 1/1

```

1  #ifndef __ERROR_H__
2  #define __ERROR_H__
3
4
5  #include <iostream>
6  #include <cstdio>
7  #include <cstdarg>
8
9  #define BUF_LEN 256
10
11 class Error : public std::exception {
12 private:
13     char msg[BUF_LEN];
14 public:
15     explicit Error(const char * format,...) noexcept;
16     virtual const char * what() const noexcept;
17     virtual ~Error() noexcept;
18 };
19
20 #endif

```

may 15, 18 16:13

commonError.cpp

Page 1/1

```

1  #include "commonError.h"
2
3  Error::Error(const char* format, ...) noexcept {
4      va_list args;
5      va_start(args, format);
6      int s = vsnprintf(msg, BUF_LEN, format, args);
7      msg[s+1] = 0;
8      va_end(args);
9  }
10
11 //devuelve el error
12 const char * Error::what() const noexcept{
13     return msg;
14 }
15 Error::~Error() noexcept{}
16

```

may 15, 18 16:13

client.h

Page 1/1

```

1  #ifndef __CLIENT_H__
2  #define __CLIENT_H__
3  #include "commonProtocol.h"
4  #include "error.h"
5  #include <stack>
6  #include <string>
7  #include <stdio.h>
8  #include <iostream>
9
10
11 using std::cout;
12 using std::endl;
13 using std::string;
14
15 #define SERVER_NAME 1
16 #define PORT 2
17 #define COMAND 3
18 #define FILE_NAME 4
19 #define HASH 5
20 #define TAG_NAME 4
21 class Client: Protocol{
22     void push(const string & file_name, const string & hash);
23
24     void tag(const string & tag_name, std::stack<string> & hashes);
25
26     void pull(const string & tag_name);
27
28 public:
29     Client(char * host_name, char * port);
30
31     //ejecuta push tag o pull segun corresponda
32     void execute_command(int argc, char * argv[]);
33 };
34
35 #endif

```

may 15, 18 16:13

client.cpp

Page 1/2

```

1  #include "client.h"
2  #include <string>
3  #include <stack>
4
5  void Client::push(const string & file_name, const string & hash){
6      send_code(PUSH_CODE);
7      send_string(file_name);
8      send_string(hash);
9      if(receive_code() == OK){
10         try{
11             send_file(file_name);
12         } catch(Error e){
13             cout << "Error: archivo inexistente.\n";
14         }
15     }
16 }
17
18 void Client::tag(const string & tag_name, std::stack<string> & hashes){
19     send_code(TAG_CODE);
20     send_size_first(hashes.size());
21     send_string(tag_name);
22     while (!hashes.empty()){
23         string hash = hashes.top();
24         hashes.pop();
25         send_string(hash);
26     }
27     if(receive_code() != OK){
28         cout << "Error: tag/hash incorrecto.\n";
29     }
30 }
31
32 void Client::pull(const string & tag_name){
33     send_code(PULL_CODE);
34     send_string(tag_name);
35     if(receive_code() == NOT_OK){
36         cout << "Error: tag/hash incorrecto.\n";
37         return;
38     }
39     int size_of_files = receive_size_first();
40     for(int i = 0; i < size_of_files; i++){
41         string file_name = receive_string();
42         receive_file(file_name);
43     }
44 }
45
46 Client::Client(char * host_name, char * port)://
47 Protocol(std::move(Socket(host_name, port))){
48     this->socket.connection();
49 }
50
51 void Client::execute_command(int argc, char * argv[]){
52     const string command = argv[COMAND];
53
54     if (command.compare("pull") == 0){
55         const string tag_name = argv[TAG_NAME];
56         pull(tag_name);
57         return;
58     }
59     if(argc < 6){
60         cout << "Error: argumentos invalidos.\n";
61         return;
62     }
63
64     if (command.compare("push") == 0){
65         const string file = argv[FILE_NAME];
66         const string hash = argv[HASH];

```

may 15, 18 16:13

client.cpp

Page 2/2

```
67     push(file, hash);
68     return;
69 }
70 if (command.compare("tag") == 0){
71     const string tag_name = argv[TAG_NAME];
72     std::stack<string> hashes;
73     for(int i = TAG_NAME+1; i< argc; i++){
74         hashes.push(argv[i]);
75     }
76     tag(tag_name,hashes);
77     return;
78 }
79 cout << "Error: argumentos invalidos.\n" ;
80 }
81
82
```

may 15, 18 16:13

clientApp.cpp

Page 1/1

```
1  #include "error.h"
2  #include "client.h"
3
4  #define SERVER_NAME 1
5  #define PORT 2
6
7
8  int main(int argc, char * argv[]){
9      if (argc < 4){
10         throw Error("Parametros incorrectos");
11     }
12
13     Client cliente(argv[SERVER_NAME],argv[PORT]);
14     cliente.execute_command(argc, argv);
15     return 0;
16 }
17
```


may 15, 18 16:13

Table of Content

Page 1/1

1	Table of Contents				
2	1	serverThConnection.h	sheets	1 to 1 (1) pages	1- 1 20 lines
3	2	serverThConnection.cpp	sheets	1 to 1 (1) pages	2- 2 33 lines
4	3	serverLock.h.....	sheets	2 to 2 (1) pages	3- 3 18 lines
5	4	serverLock.cpp.....	sheets	2 to 2 (1) pages	4- 4 12 lines
6	5	serverListener.h....	sheets	3 to 3 (1) pages	5- 5 24 lines
7	6	serverListener.cpp..	sheets	3 to 3 (1) pages	6- 6 39 lines
8	7	serverIndex.h.....	sheets	4 to 4 (1) pages	7- 8 97 lines
9	8	serverIndex.cpp.....	sheets	5 to 6 (2) pages	9- 11 144 lines
10	9	serverConnection.h..	sheets	6 to 6 (1) pages	12- 12 26 lines
11	10	serverConnection.cpp	sheets	7 to 7 (1) pages	13- 14 87 lines
12	11	serverApp.cpp.....	sheets	8 to 8 (1) pages	15- 15 26 lines
13	12	commonThread.h.....	sheets	8 to 8 (1) pages	16- 16 33 lines
14	13	commonSocket.h.....	sheets	9 to 9 (1) pages	17- 18 91 lines
15	14	commonSocket.cpp....	sheets	10 to 11 (2) pages	19- 22 215 lines
16	15	commonProtocol.h....	sheets	12 to 12 (1) pages	23- 24 85 lines
17	16	commonProtocol.cpp..	sheets	13 to 13 (1) pages	25- 26 118 lines
18	17	commonError.h.....	sheets	14 to 14 (1) pages	27- 27 21 lines
19	18	commonError.cpp.....	sheets	14 to 14 (1) pages	28- 28 17 lines
20	19	client.h.....	sheets	15 to 15 (1) pages	29- 29 36 lines
21	20	client.cpp.....	sheets	15 to 16 (2) pages	30- 31 83 lines
22	21	clientApp.cpp.....	sheets	16 to 16 (1) pages	32- 32 18 lines