75:42 - Taller de Programación I

Ejercicio Nº ___ Padrón _____

Alumno _____ Firma _____

Nota:		**Corrige**:		**Entrega #1**
				Fecha de entrega
				Fecha de devolución

Nota:		**Corrige**:		**Entrega #2**
				Fecha de entrega
				Fecha de devolución

El presente trabajo, así como la entrega electrónica correspondiente al mismo, constituyen una obra de creación completamente personal, no habiendo sido inspirada ni siendo copia completa o parcial de ninguna fuente pública, privada, de otra persona o naturaleza.

Índice

1. Objetivos

Se desea implementar un Honeypot FTP que permita aceptar múltiples clientes al mismo tiempo. Para ello se hará uso de de los conceptos implementados en trabajos anteriores como Sockets y Threads. El servidor guardará una lista de directorios ficticia a cargar por los clientes y se las enviará al ser solicitada en el caso de que los clientes posean las credenciales correctas configurables en un archivo del servidor.

2. Resolución del problema

La ejecución básica del algoritmo implementado se ilustra en la figura 1.

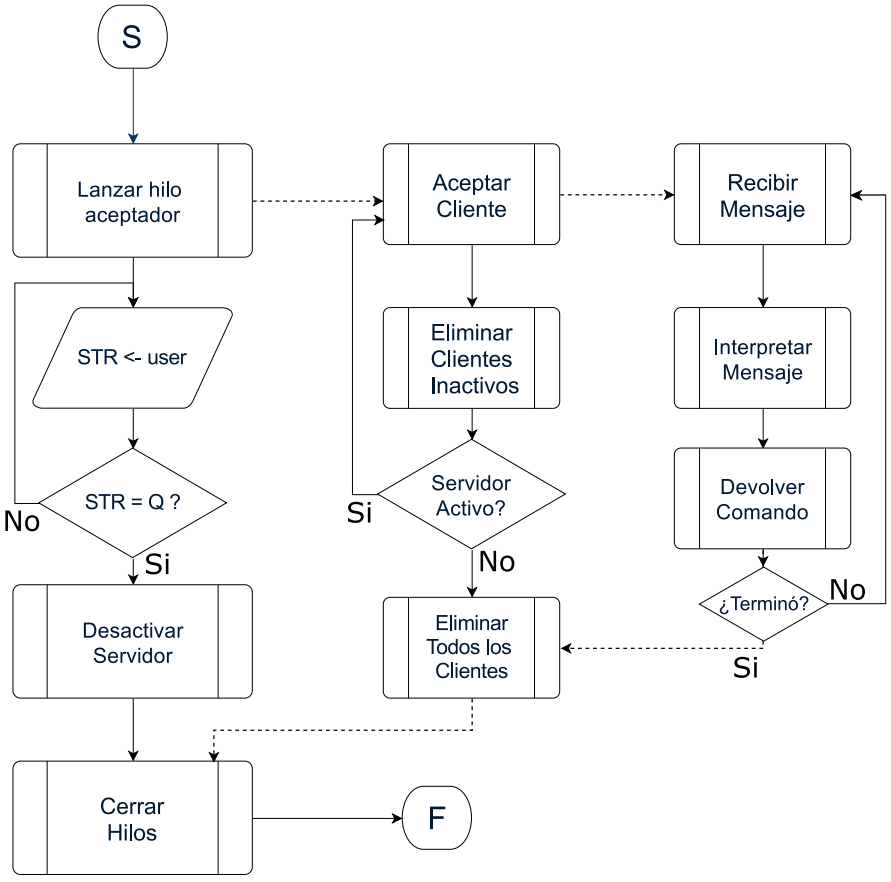

Figura 1: Diagrama del algoritmo.

3. Clases implementadas

- **Server:** Implementa **el lanzamiento de los threads que permiten el envío y recepción** y contiene la lógica necesaria para manejar múltiples clientes al mismo tiempo y almacenar el listado de directorios compartido.
- **Client:** Implementa la lógica del cliente para enviar pedidos y recibir respuestas del servidor. Además permite almacenar la identidad del cliente en el servidor.
- **Error:** Implementa el manejo de excepciones y errores del programa.
- **Protocol:** Contiene la lógica necesaria para ejecutar el envío y recepción de códigos y mensajes.
- **Socket:** Contiene las instrucciones necesarias para aceptar, conectar y configurar Sockets así como también implementa el envío y recepción de determinada cantidad de bytes.
- **Config:** Contiene las instrucciones necesarias para leer el archivo de configuración y cargar la configuración al servidor. Mediante la misma se determina qué mensajes enviar como respuesta.
- **Command:** Contiene la lógica para poder establecer qué mensajes enviar como respuesta al cliente. En esta clase se implementaron conceptos de polimorfismo para que la implementación resulte transparente para el usuario de Command.

4. Changelog

- Nombre de clases en ProperCase.
- Nuevo uso de clases completas Server y Client para mejor división de responsabilidades.
- Nueva clase Error para manejo de excepciones.
- Protocol ahora sólo se encarga de enviar y recibir según el protocolo.
- Protocol ahora manda sólo strings y no números. receive() corta con \n.
- los códigos de respuesta ahora se envían como string.
- Añadido get_code() para obtener codigo de la respuesta si es que existe.
- Los métodos de Command se reordenaron para favorecer legibilidad.
- LIST ahora corta con el código de LIST_END.
- Todas las implementaciones de metodos fueron a los .cpp.
- Excepciones para validar argumentos eliminadas.
- Se reemplazó Socket.destroy() por Socket.shutdown().
- Removidos *class* innecesarios.

- Se agregó respond_query() para que el servidor procese 1 pedido de 1 cliente más ordenadamente.
- run_client() con mejor legibilidad y más coherente.
- atributo running_threads eliminado.
- Mejor uso de const.
- Excepciones detalladas en socket con errno.
- load_config(...) ahora es el constructor Config(...).
- Directorios thread safe en funciones individuales con lock_guard en lugar de la creación del comando.
- La lista de directorios se obtiene del servidor como un único string en lugar de copiar toda la lista.

5. Sistemas y software utilizados

- WSL 18.04 (Windows Subsystem for Linux).
- Ubuntu 18.04 nativo.
- SERCOM.
- `gdb`
- `Valgrind`
- VSCode
- CLion

6. Conclusiones

Si bien la división de responsabilidades no es óptima, se llevó a cabo una primera aproximación a los conceptos de Polimorfismo, RAII, uso de STL, así como también se integraron conexiones cliente-servidor y threads. El software presentado posee amplio margen de mejora pero permitió desarrollar los conceptos adquiridos por el alumno.

oct 15, 19 5:35 **Socket.h** Page 1/1

```
#ifndef TP3_SOCKET_H
#define TP3_SOCKET_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/types.h>
#include <netdb.h>
#include <iostream>
#include <vector>
#include "common_error.h"

#define WAITING_CLIENTS 4  // max clients number

#define ERROR_ADDRINFO_SERVER "Error getting server's address info"
#define ERROR_SOCKET_ENDPOINT "Error creating socket's endpoint"
#define ERROR_CONNECTING "Error connecting socket to server"
#define ERROR_WAKING_SOCKET "Error setting socket to alive"
#define ERROR_CONFIGURING_CLIENT_SOCKET "Error using setsockopt()"
#define ERROR_SETTING_LISTEN "Error setting socket as listening."
#define ERROR_BINDING "Error making bind."
#define ERROR_SOCKET_ACCEPTOR "Error accepting socket."
#define ERROR_SHUTDOWN "Error on socket shutdown"
#define ERROR_RECEPTION "Error in reception of socket"
#define ERROR_SENDING "Error in sending of socket"

typedef struct addrinfo addrinfo_t;

/* Handles the connection between server and
 * client. Implements the send/receive logic */
class Socket {
 protected:
    int socket_id;
    bool is_alive = false;

 public:
    explicit Socket(char *, char *);  // client
    explicit Socket(char * port);  // server
    explicit Socket(int id);
    Socket();

    void send(int, char *, int);
    void bind(addrinfo_t *res);
    void shutdown();
    void listen();
    int receive(int, char *, int);
    int accept();
    int get_id();
    bool is_accepted();
    bool is_dead();
    std::ostream& operator<<(std::ostream&) const;
};

#endif  // TP3_SOCKET_H
```

oct 15, 19 5:35 **server_main.cpp** Page 1/1

```
#include <thread>
#include "Protocol.h"
#include "Config.h"
#include "common_server.h"

int main(int argc, char * argv[]) {
    Protocol ftp(argc, argv, 'S');
    try {
        Server server(argv[1], argv[2]);
        Config configuration(argv[2]);
        std::thread server_acceptor = std::thread(&Server::run_server,
                                                  &server,
                                                  std::ref(configuration),
                                                  std::ref(server),
                                                  std::ref(ftp));
        server.quit();
        server.close_all_sockets();
        server_acceptor.join();
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
    return 0;
}
```

oct 15, 19 5:35 **Protocol.h** Page 1/1

```
#ifndef TP3_PROTOCOL_H
#define TP3_PROTOCOL_H

#include <thread>
#include <set>
#include <mutex>
#include <vector>
#include <string>
#include "Socket.h"
#include "Config.h"

#define PROTOCOL_ERR_CMD_CLIENT "Use: ./client <ip/hostname> <port/service>"
#define PROTOCOL_ERR_CMD_SERVER "Use: ./server <port> <config file>"
#define SERVER_QUIT_CMD1 "q"
#define SERVER_QUIT_CMD2 "q\n"
#define RECEIVED_STRING_DELIMITER '\n'

/* Defines transmission methods. Stores address, port
 * and configuration file.*/
class Protocol {
 private:
    class Socket socket;
    char * config_file;
    char * port;
    char * hostname;

 public:
    Protocol(int, char*[], char);

    bool validate_args(int argc, char **argv, char type) const;
    bool validate_server_args(int argc, char **argv) const;
    bool validate_client_args(int argc, char **argv) const;
    void send_string(const std::string &cmd, int socket_id);
    std::string receive_string(int id);
};
#endif  // TP3_PROTOCOL_H
```

oct 15, 19 5:35 **Config.h** Page 1/1

```
#ifndef SERVER_CONFIG_H
#define SERVER_CONFIG_H

#include <string>
#include <iostream>
#include <fstream>
#include <unordered_map>

typedef std::unordered_map<std::string, std::string>::iterator mapit;

/* Stores configuration file parameters in memory */
class Config {
 private:
    std::ifstream configstream;
    std::unordered_map<std::string, std::string> configmap;

 public:
    explicit Config(char * config_file);
    std::string get_message(const std::string &key);
};

#endif  // SERVER_CONFIG_H
```

oct 15, 19 5:35 **common_socket.cpp** Page 1/4

```
#define _POSIX_C_SOURCE 200112L

#include <iostream>
#include "Socket.h"

/* Constructor: not connected socket */
Socket::Socket() {
    socket_id = -1;
}

/* Constructor: based on connected socket */
Socket::Socket(int id) {
    socket_id = id;
    is_alive = true;
}

/* Sends len bytes of buf using the socket
 * identified by socket_id */
void Socket::send(int len, char *buf, int socket_id) {
    bool socket_running = true;
    int bytes_sent = 0;
    int st = 0;

    while (socket_running ∧ bytes_sent < len) {
        st = ::send(socket_id, &buf[bytes_sent], len - bytes_sent, 0);
        if (st ≡ 0) {
            socket_running = false;
            is_alive = false;
        } else if (st ≡ -1) {
            socket_running = false;

        } else {
            bytes_sent = bytes_sent + st;
        }
    }
    if (st ≡ -1) {
        throw Error("%s (%s%d). Error number: %d.",
                    ERROR_SENDING,
                    "socket_id = ",
                    socket_id,
                    errno);
    }
}

/* Receives len bytes of buf using the socket
 * identified by socket */
int Socket::receive(int len, char *buf, int socket) {
    bool socket_running = true;
    int bytes_received = 0;
    int st = 0;
    while (socket_running ∧ bytes_received < len) {
        st = recv(socket, &buf[bytes_received],
                  len - bytes_received,
                  MSG_NOSIGNAL);
        if (st ≡ 0) {
            socket_running = false;
        } else if (st ≡ -1) {
            socket_running = false;
        } else {
            bytes_received = bytes_received + st;
        }
    }
    if ( bytes_received > 0 ) {
        return bytes_received;
    } else if (bytes_received ≡ 0) {
        is_alive = false;
```

oct 15, 19 5:35 **common_socket.cpp** Page 2/4

```
        return -1;
    } else {
        throw Error("%s (%s%d). Error number: %d.",
                    ERROR_RECEPTION,
                    "socket_id = ",
                    socket_id,
                    errno);
    }
}

/* Constructor: creates Socket to work as client */
Socket::Socket(char *hostname, char *port) {
    int st;
    struct addrinfo hints {};
    struct addrinfo *res;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = 0;

    st = getaddrinfo(hostname, port, &hints, &res);
    if (st ≠ 0)
        throw Error("%s (%s%d).",
                    ERROR_ADDRINFO_SERVER,
                    "st = ",
                    st);

    socket_id = ::socket(res→ai_family, res→ai_socktype, res→ai_protocol);
    if (socket_id ≡ -1) {
        freeaddrinfo(res);
        throw Error("%s (%s%d).",
                    ERROR_SOCKET_ENDPOINT,
                    "socket_id = ",
                    socket_id);
    }

    st = ::connect(socket_id, res→ai_addr, res→ai_addrlen);
    if (st ≡ -1) {
        freeaddrinfo(res);
        close(socket_id);
        throw Error("%s (%s%d).",
                    ERROR_CONNECTING,
                    "st = ",
                    st);
    }

    freeaddrinfo(res);
    if (st ≠0 ∨ socket_id ≡ -1)
        throw Error("%s (%s%d, %s%d).",
                    ERROR_WAKING_SOCKET,
                    "st = ",
                    st,
                    "socket_id = ",
                    socket_id);
    is_alive = true;
}

/* Constructor: creates Socket to work as server */
Socket::Socket(char *port) {
    int c = 1;
    int st;
    struct addrinfo hints {};
    struct addrinfo *res;

    memset(&hints, 0, sizeof(struct addrinfo));
```

```
133     hints.ai_family = AF_INET;
134     hints.ai_socktype = SOCK_STREAM;
135     hints.ai_flags = AI_PASSIVE;
136
137     st = getaddrinfo(nullptr, port, &hints, &res);
138     if (st ≠ 0)
139         throw Error("%s (%s%d). Error number: %d.",
140                     ERROR_ADDRINFO_SERVER,
141                     "st = ",
142                     st,
143                     errno);
144
145     socket_id = ::socket(res→ai_family, res→ai_socktype, res→ai_protocol);
146     if (socket_id ≡ −1) {
147         freeaddrinfo(res);
148         throw Error("%s (%s%d). Error number: %d.",
149                     ERROR_SOCKET_ENDPOINT,
150                     "socket_id = ",
151                     socket_id,
152                     errno);
153     }
154
155     st = setsockopt(socket_id, SOL_SOCKET, SO_REUSEADDR, &c, sizeof(&c));
156     if (st ≡ −1) {
157         freeaddrinfo(res);
158         throw Error("%s (%s%d). Error number: %d.",
159                     ERROR_CONFIGURING_CLIENT_SOCKET,
160                     "st = ",
161                     st,
162                     errno);
163     }
164     bind(res);
165     listen();
166     freeaddrinfo(res);
167     is_alive = true;
168 }
169
170 /* Sets server socket to listen incoming connections. */
171 void Socket::listen() {
172     if (::listen(socket_id, WAITING_CLIENTS) ≡ −1) {
173         close(socket_id);
174         throw Error("%s Error number: %d.",
175                     ERROR_SETTING_LISTEN,
176                     errno);
177     }
178 }
179
180 /* Assigns address specified by addr to the socket */
181 void Socket::bind(addrinfo_t *res) {
182     if (::bind(socket_id, res→ai_addr, res→ai_addrlen) ≡ −1) {
183         close(socket_id);
184         throw Error("%s Error number: %d.",
185                     ERROR_BINDING,
186                     errno);
187     }
188 }
189
190 /* Accepts incoming connection from client to server */
191 int Socket::accept() {
192     int peer_id;
193     peer_id = ::accept(socket_id, nullptr, nullptr);
194     if (peer_id ≡ −1) {
195         close(socket_id);
196         throw Error("%s (%s%d). Error number: %d.",
197                     ERROR_SOCKET_ACCEPTOR,
198                     "socket_id = ",
```



```
199                     socket_id,
200                     errno);
201     }
202     return peer_id;
203 }
204
205 /* Sends shutdown signal and closes the socket */
206 void Socket::shutdown() {
207     if (::shutdown(socket_id, SHUT_RDWR) ≡ −1) {
208         throw Error("%s (%s%d). Error number: %d.",
209                     ERROR_SHUTDOWN,
210                     "socket_id = ",
211                     socket_id,
212                     errno);
213     }
214     close(socket_id);
215 }
216
217 /* Checks if socket has been accepted */
218 bool Socket::is_accepted() { return socket_id ≠ −1; }
219
220 /* Gets socket identification number */
221 int Socket::get_id() {return socket_id;}
222
223 /* Checks if socket is ON or OFF */
224 bool Socket::is_dead() {return ¬is_alive;}
225
226 /* Prints socket's identification number */
227 std::ostream& Socket::operator<<(std::ostream& os) const {
228     os << socket_id;
229     return os;
230 }
```

oct 15, 19 5:35 **common_server.h** Page 1/1

```h
#ifndef SERVER_COMMON_SERVER_H
#define SERVER_COMMON_SERVER_H

#include <vector>
#include <string>
#include <set>
#include "Client.h"

typedef std::vector<Client>::iterator client_it;
typedef std::set<std::string>::iterator dir_it;

/* Implements the logic necessary to process
 * clients connections and saves the shared list
 * of directories. Stores a list of clients. */
class Server {
 private:
    std::vector<Client> clients;
    std::set<std::string> directories;
    Socket socket;
    bool is_running = true;
    std::mutex m;
    std::mutex dir_m;
    char * port;
    char * config;

 public:
    Server(char *string, char *string1);

    static void send_hello(const Client& cli, Config &cfg, Protocol &protocol);
    void respond_query(std::string &client_query,
                       Config &cfg,
                       Client &cli,
                       Protocol *protocol);
    void respond_client(Config &cfg, int socket_id, Protocol &protocol);
    void run_server(Config &cfg, Server &server, Protocol &protocol);
    void close_dead_sockets();
    void close_all_sockets();
    void stop();
    void quit();
    bool remove_directory(const std::string &directory);
    bool add_directory(const std::string&);
    bool running() const;
    int directories_size();
    std::string get_list();
};


#endif  // SERVER_COMMON_SERVER_H
```

oct 15, 19 5:35 **common_server.cpp** Page 1/3

```cpp
#include <string>
#include <vector>
#include <set>
#include "common_server.h"
#include "Command.h"

/* Implements the acceptor thread. Constantly waits for incoming clients.
 * If a client is accepted, launches a thread for that client and checks
 * if other clients ceased operation closing their threads and sockets */
void Server::run_server(Config &cfg, Server &server, Protocol &protocol) {
    int socket_id = -1;
    std::vector<std::thread> threads(200);

    try {
        socket = Socket(port);
        while (is_running) {
            socket_id = socket.accept();
            Client client(socket_id);
            if (client.is_accepted()) {
                threads.emplace_back(std::thread(&Server::respond_client,
                                                 this,
                                                 std::ref(cfg),
                                                 socket_id,
                                                 std::ref(protocol)));
                client.save_thread(&threads.back());
                clients.push_back(client);
                close_dead_sockets();
            }
        }
        close_all_sockets();
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
        close_all_sockets();
    }
}

/* Implements individual client's thread. Receives strings from the client
 * and processes them to determine if they constitute a valid command. If so,
 * answers properly sending the required information, updates directories
 * or verifies the login credentials */
void Server::respond_client(Config &cfg, int socket_id, Protocol &protocol) {
    std::string client_query;
    Client cli(socket_id);
    try {
        send_hello(cli, cfg, protocol);
        while (¬cli.quit) {
            client_query = protocol.receive_string(cli.get_id());
            if (client_query.length() > 1) {
                respond_query(client_query, cfg, cli, &protocol);
            } else {
                cli.quit = true;
            }
        }
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}

/* Creates a command to give the client a response. Updates directories
 * or credentials if necessary */
void Server::respond_query(std::string &client_query,
                           Config &cfg,
                           Client &cli,
                           Protocol *protocol) {
    Command * cmd;
    CommandCreator creator;
```

```
67      cmd = creator.create_command(client_query);
68      cmd→send_response(cfg, protocol, this, cli);
69      delete cmd;
70  }
71
72  /* Sends initial message to the client to start communication */
73  void Server::send_hello(const Client& cli, Config &cfg, Protocol &protocol) {
74      protocol.send_string(std::string(ID_NEW_CLIENT) +
75                  "" +
76                  cfg.get_message(MSG_NEW_CLIENT),
77                  cli.socket_id);
78  }
79
80  /* Closes all sockets if quit signal has been received */
81  void Server::close_all_sockets() {
82      std::lock_guard<std::mutex> lock(m);
83      while (¬clients.empty()) {
84          clients.back().shutdown_socket();
85          clients.back().close_thread();
86          clients.pop_back();
87      }
88  }
89
90  /* Closes dead sockets if the socket ceased the connection */
91  void Server::close_dead_sockets() {
92      std::lock_guard<std::mutex> lock(m);
93      for (client_it cli = clients.begin();
94           cli ≠ clients.end();
95           cli++) {
96          if (cli→is_dead()) {
97              cli→close_thread();
98              cli→shutdown_socket();
99              clients.erase(cli);
100         }
101     }
102 }
103
104 /* Thread-safe. If absent, adds the directory requested by client */
105 bool Server::add_directory(const std::string& directory) {
106     std::lock_guard<std::mutex> lock(dir_m);
107     if (directories.find(directory) ≡ directories.end()) {
108         directories.insert(directory);
109         return true;
110     }
111     return false;
112 }
113
114 /* Thread-safe. If present, adds the directory requested by client */
115 bool Server::remove_directory(const std::string & directory) {
116     std::lock_guard<std::mutex> lock(dir_m);
117     std::set<std::string>::iterator it;
118     it = directories.find(directory);
119     if (it ≠ directories.end()) {
120         directories.erase(it);
121         return true;
122     }
123     return false;
124 }
125
126 /* Thread-safe. Gets a formatted list of all directories for the client */
127 std::string Server::get_list() {
128     std::lock_guard<std::mutex> lock(dir_m);
129     std::string list;
130     dir_it it;
131     for (it=directories.begin(); it ≠ directories.end(); it++)
132         list += DIR_FORMAT + *it;
```



```
133     return list;
134 }
135
136 /* Gets the directory list's size */
137 int Server::directories_size() {
138     std::lock_guard<std::mutex> lock(dir_m);
139     return directories.size();
140 }
141
142 /* Waits upon the user inputs a quit command for server shutdown*/
143 void Server::quit() {
144     std::string str;
145     while (running()) {
146         getline(std::cin, str);
147         if ( str ≡ SERVER_QUIT_CMD1 ∨ str ≡ SERVER_QUIT_CMD2 )
148             stop();
149     }
150 }
151
152 /* Checks if server is running */
153 bool Server::running() const {return is_running;}
154
155 /* If server's quit command has been received,
156  * closes all the sockets, producing the acceptor
157  * thread to close all threads. Also closes server's
158  * socket.*/
159 void Server::stop() {
160     close_all_sockets();
161     is_running = false;
162     socket.shutdown();
163 }
164
165 /* Constructor: creates server in specified
166  * port and configuration file */
167 Server::Server(char *h, char *c) {
168     port = h;
169     config = c;
170 }
```

oct 15, 19 5:35 **common_protocol.cpp** Page 1/1

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

/* Sets up host and port for client, port and config for server. */
Protocol::Protocol(int argc, char * argv[], char type) {
    if ( validate_args(argc, argv, type) ) {
        switch (type) {
            case 'S':
                port = argv[1];
                config_file = argv[2];
                break;
            case 'C':
                hostname = argv[1];
                port = argv[2];
                break;
        }
    }
}

/* Sends entire string at once. */
void Protocol::send_string(const std::string &cmd, int socket_id) {
    socket.send(cmd.length(), const_cast<char *>(cmd.c_str()), socket_id);
}

/* Receives string one byte at a time.
 * String delimited by RECEIVED_STRING_DELIMITER. */
std::string Protocol::receive_string(int id) {
    std::string str;
    char aux = 0;

    while (aux ≠ RECEIVED_STRING_DELIMITER) {
        socket.receive(1, &aux, id);
        str.push_back(aux);
    }
    return str;
}

/* Validates client execution arguments. */
bool Protocol::validate_client_args(int argc, char **argv) const {
    return argv ≠ nullptr ∧ argc ≡ 3;
}

/* Validates server execution arguments. */
bool Protocol::validate_server_args(int argc, char **argv) const {
    return argv ≠ nullptr ∧ argc ≡ 3;
}

/* Validates client or server arguments. */
bool Protocol::validate_args(int argc, char **argv, char type) const {
    switch (type) {
        case 'C':
            return validate_client_args(argc, argv);
        case 'S':
            return validate_server_args(argc, argv);
        default:
            return false;
    }
}
```

oct 15, 19 5:35 **common_error.h** Page 1/1

```cpp
#ifndef ERROR_H
#define ERROR_H

#include <exception>
#include <cstdarg>

#define ERROR_MSG_LENGTH 100

/* Handles exceptions and error messages */
class Error : public std::exception {
 private:
    char error_msg[ERROR_MSG_LENGTH]{};

 public:
    explicit Error(const char * fmt, ...) noexcept;
    const char * what() const noexcept override;
};

#endif  // ERROR_H
```

oct 15, 19 5:35 **common_error.cpp** Page 1/1

```cpp
#include <cstdio>
#include "common_error.h"

/* Creates an exception receiving multiple arguments of
 * different formats to form a detailed string message */
Error::Error(const char *fmt, ...) noexcept {
    va_list args;

    va_start(args, fmt);
    vsnprintf(error_msg, ERROR_MSG_LENGTH, fmt, args);
    va_end(args);
}

/* Gets formatted error message */
const char * Error::what() const noexcept {
    return error_msg;
}
```

oct 15, 19 5:35 **common_config.cpp** Page 1/1

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

/* Constructor: saves file and configuration specified in file*/
Config::Config(char * config_file) {
    std::string aux_str1;
    std::string aux_str2;
    configstream.open(config_file, std::ios::in);
    while (¬configstream.eof()) {
        std::getline(configstream, aux_str1, '=');
        std::getline(configstream, aux_str2, '\n');
        aux_str2.push_back('\n');
        configmap.emplace(aux_str1, aux_str2);
    }
    configstream.close();
}

/* Gets message loaded in memory from specified config file.*/
std::string Config::get_message(const std::string & key) {
    mapit it = configmap.find(key);
    return it→second;
}
```

```
#include <string>

#include "Command.h"
#include "Config.h"
#include "Protocol.h"

/* Sends response to USER command */
void UserCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    std::string CMD = format_cmd(ID_PASS_REQUIRED, MSG_PASS_REQUIRED, cfg);
    protocol→send_string(CMD, cli.socket_id);
    cli.possible_user = possible_user;
}

/* Sends response to PASS command attempting to
 * log in if credentials match the config file */
void PassCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    if (valid_credentials(cfg, cli)) {
        std::string CMD = format_cmd(ID_LOGIN_SUCCESS,
                                     MSG_LOGIN_SUCCESS,
                                     cfg);
        protocol→send_string(CMD, cli.socket_id);
        cli.logged = true;
    } else {
        std::string CMD = format_cmd(ID_LOGIN_FAILED,
                                     MSG_LOGIN_FAILED,
                                     cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Checks if credentials match with the config file */
bool PassCommand::valid_credentials(Config & cfg, Client & cli) {
    return (cli.possible_user ≡ cfg.get_message(KEY_USER)) ∧
           (possible_pass ≡ cfg.get_message(KEY_PASS));
}

/* Sends server's system info if logged*/
void SystCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    if (cli.logged) {
        std::string CMD = format_cmd(ID_SYST, MSG_SYST, cfg);
        protocol→send_string(CMD, cli.socket_id);
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Sends list of directories if logged */
void ListCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    if (cli.logged) {
        std::string CMD = format_cmd(ID_LIST_BEGIN, MSG_LIST_BEGIN, cfg);
        protocol→send_string(CMD, cli.socket_id);
        if (server→directories_size() > 0)
            protocol→send_string(server→get_list(), cli.socket_id);
```



```
        CMD = format_cmd(ID_LIST_END, MSG_LIST_END, cfg);
        protocol→send_string(CMD, cli.socket_id);
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Sends valid commands if logged */
void HelpCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    if (cli.logged) {
        std::string CMD = format_cmd(ID_HELP, MSG_HELP, cfg);
        protocol→send_string(CMD, cli.socket_id);
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Sends current directory if logged */
void PwdCommand::send_response(Config &cfg,
                               Protocol *protocol,
                               Server *server,
                               Client &cli) {
    if (cli.logged) {
        std::string CMD = format_cmd(ID_PWD, MSG_PWD, cfg);
        protocol→send_string(CMD, cli.socket_id);
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Creates a directory if it's not present and
 * sends a response to the client if the operation
 * has been successful or not */
void MkdCommand::send_response(Config &cfg,
                               Protocol *protocol,
                               Server *server,
                               Client &cli) {
    if (cli.logged) {
        if (server→add_directory(dirname)) {
            std::string CMD = format_cmd(ID_MKD_SUCCESS,
                                         dirname,
                                         MSG_MKD_SUCCESS,
                                         cfg);
            protocol→send_string(CMD, cli.socket_id);
        } else {
            std::string CMD = format_cmd(ID_MKD_FAILURE,
                                         MSG_MKD_FAILURE,
                                         cfg);
            protocol→send_string(CMD, cli.socket_id);
        }
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Removes a directory if it's present and
 * sends a response to the client if the operation
 * has been successful or not */
void RmdCommand::send_response(Config &cfg,
```

oct 15, 19 5:35 **common_command.cpp** Page 3/4

```
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    if (cli.logged) {
        if (server→remove_directory(dirname)) {
            std::string CMD = format_cmd(ID_RMD_SUCCESS,
                                         dirname,
                                         MSG_RMD_SUCCESS,
                                         cfg);
            protocol→send_string(CMD, cli.socket_id);
        } else {
            std::string CMD = format_cmd(ID_RMD_FAILURE,
                                         MSG_RMD_FAILED,
                                         cfg);
            protocol→send_string(CMD, cli.socket_id);
        }
    } else {
        std::string CMD = format_cmd(ID_NOT_LOGGED, MSG_NOT_LOGGED, cfg);
        protocol→send_string(CMD, cli.socket_id);
    }
}

/* Sets quit signal in the client and tells the client
 * to disconnect */
void QuitCommand::send_response(Config &cfg,
                                Protocol *protocol,
                                Server *server,
                                Client &cli) {
    cli.quit = true;
    std::string CMD = format_cmd(ID_QUIT, MSG_QUIT, cfg);
    protocol→send_string(CMD, cli.socket_id);
}

/* If an unknown command has been received, warns the user */
void InvalidCommand::send_response(Config &cfg,
                                   Protocol *protocol,
                                   Server *server,
                                   Client &cli) {
    std::string CMD = format_cmd(ID_UNKNOWN, MSG_UNKNOWN, cfg);
    protocol→send_string(CMD, cli.socket_id);
}

/* Creates a command with some validation */
Command * CommandCreator::create_command(std::string & input) {
    std::string cmd_argument;
    size_t argument_start_pos;

    if (input.length() > 5) {
        argument_start_pos = input.find(' ') + 1;
        cmd_argument = input.substr(argument_start_pos, std::string::npos);
        input = input.substr(0, argument_start_pos - 1);
    } else {
        input.pop_back();  // removes new line char
    }

    Command * cmd = identify_command(input, cmd_argument);
    return cmd;
}

/* Checks if the command matches with the commands specified
 * in the config file */
Command * CommandCreator::identify_command(std::string & cmd,
                                           std::string & arg) {
    std::string cmds_id[] = {CMD_USER, CMD_PASS, CMD_SYST,
                             CMD_LIST, CMD_HELP, CMD_PWD,
                             CMD_MKD, CMD_RMD, CMD_QUIT};
```

oct 15, 19 5:35 **common_command.cpp** Page 4/4

```

    if (¬cmd.compare(cmds_id[0]))
        return new UserCommand(arg);
    if (¬cmd.compare(cmds_id[1]))
        return new PassCommand(arg);
    if (¬cmd.compare(cmds_id[2]))
        return new SystCommand;
    if (¬cmd.compare(cmds_id[3]))
        return new ListCommand;
    if (¬cmd.compare(cmds_id[4]))
        return new HelpCommand;
    if (¬cmd.compare(cmds_id[5]))
        return new PwdCommand;
    if (¬cmd.compare(cmds_id[6]))
        return new MkdCommand(arg);
    if (¬cmd.compare(cmds_id[7]))
        return new RmdCommand(arg);
    if (¬cmd.compare(cmds_id[8]))
        return new QuitCommand;
    return new InvalidCommand;
}

/* Formats response to send */
std::string Command::format_cmd(const char * ID,
                                const char * message,
                                Config & cfg) {
    return std::string(ID) + " " + cfg.get_message(message);
}

/* Formats response to send with directory name */
std::string Command::format_cmd(const char * ID,
                                std::string & dir,
                                const char * message,
                                Config & cfg) {
    dir.pop_back();  // removes new line char
    return std::string(ID) +
           " " +
           "\"" + dir + "\" " +
           cfg.get_message(message);
}

/* Constructor: creates user command with a possible user */
UserCommand::UserCommand(const std::string & arg) { possible_user = arg;}

/* Constructor: creates pass command with a possible password */
PassCommand::PassCommand(const std::string & arg) { possible_pass = arg;}

/* Constructor: creates mkd command with a directory name */
MkdCommand::MkdCommand(const std::string &arg) { dirname = arg;}

/* Constructor: creates rmd command with a directory name */
RmdCommand::RmdCommand(const std::string &arg) { dirname = arg;}

```

oct 15, 19 5:35 **common_client.cpp** Page 1/2

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

/* Constructor: disconnected client */
Client::Client() {
    socket_id = -1;
}

/* Constructor: based on existing connection */
Client::Client(int id) {
    socket_id = id;
    if (id > 0)
        is_alive = true;
}

/* Saves thread's address used to attend
 * client's requests on server side. Used
 * for later closure */
void Client::save_thread(std::thread * th) {
    thread = th;
}

/* Closes thread used to attend the client
 * in the server */
void Client::close_thread() {
    if (thread->joinable())
        thread->join();
}

/* Sends shutdown signal and closes the
 * socket related to that client */
void Client::shutdown_socket() {
    ::shutdown(socket_id, SHUT_RDWR);
    close(socket_id);
}

/* Checks if client is connected */
bool Client::is_dead() {
    return ¬is_alive;
}

/* Implements the logic to send requests to the
 * server and receives proper responses. The server
 * checks if the requests are valid commands */
void Client::run_client(int argc, char * argv[]) {
    try {
        Socket socket = Socket(argv[1], argv[2]);
        socket_id = socket.get_id();
        std::string user_input, response;
        Protocol ftp(argc, argv, 'C');
        bool running = true;

        receive_hello(ftp);
        while (running) {
            get_user_input(user_input);
            if (user_input ≠ "\n") {
                ftp.send_string(user_input, socket.get_id());
                response = ftp.receive_string(socket.get_id());
                print_response(response);
                if (is_list_start(response))
                    receive_list(ftp);
            } else if (user_input ≡ "\n" ∨ quit_received(response)) {
                ftp.send_string(user_input, socket.get_id());
                running = false;
            }
        }
```

oct 15, 19 5:35 **common_client.cpp** Page 2/2

```
        socket.shutdown();
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}

/* Checks if server accepted the quit signal */
bool Client::quit_received(const std::string & response) {
    return get_code(response) ≡ std::string(ID_QUIT);
}

/* Waits until reception of initial message from server */
void Client::receive_hello(Protocol & protocol) {
    std::string response;
    response = protocol.receive_string(socket_id);
    print_response(response);
}

/* Reads user's query from input */
void Client::get_user_input(std::string & user_input) {
    user_input.clear();
    getline(std::cin, user_input);
    user_input.push_back('\n');
}

/* Checks if the server started to send a list of directories */
bool Client::is_list_start(const std::string & str) {
    return get_code(str) ≡ std::string(ID_LIST_BEGIN);
}

/* Prints code and message sent by server */
void Client::print_response(const std::string & received_string) const {
    std::cout << received_string;
}

/* Receives list of directories and prints it */
void Client::receive_list(Protocol & protocol) {
    std::string list_item;
    while (get_code(list_item) ≠ ID_LIST_END) {
        list_item = protocol.receive_string(socket_id);
        print_response(list_item);
    }
}

/* Gets response's code from the string*/
std::string Client::get_code(std::string const & received_string) {
    std::string code;
    for (int i=0; i < CODE_LENGTH; i++)
        code.push_back(received_string[i]);
    return code;
}

/* Checks if client has been accepted by server */
bool Client::is_accepted() { return socket_id ≠ -1; }

/* Gets socket's identification number of a client*/
int Client::get_id() { return socket_id; }
```

```
#ifndef SERVER_COMMAND_H
#define SERVER_COMMAND_H

#include <string>
#include "Client.h"
#include "common_server.h"

#define CMD_USER "USER"
#define CMD_PASS "PASS"
#define CMD_SYST "SYST"
#define CMD_LIST "LIST"
#define CMD_HELP "HELP"
#define CMD_PWD "PWD"
#define CMD_MKD "MKD"
#define CMD_RMD "RMD"
#define CMD_QUIT "QUIT"

#define CODE_LENGTH 3

#define ID_NEW_CLIENT "220"
#define ID_NOT_LOGGED "530"
#define ID_PASS_REQUIRED "331"
#define ID_LOGIN_SUCCESS "230"
#define ID_LOGIN_FAILED "530"
#define ID_UNKNOWN "530"
#define ID_SYST "215"
#define ID_HELP "214"
#define ID_LIST_BEGIN "150"
#define ID_LIST_END "226"
#define ID_PWD "257"
#define ID_MKD_SUCCESS "257"
#define ID_MKD_FAILURE "550"
#define ID_RMD_SUCCESS "250"
#define ID_RMD_FAILURE "550"
#define ID_QUIT "221"

#define MSG_NEW_CLIENT "newClient"
#define KEY_USER "user"
#define KEY_PASS "password"
#define MSG_PASS_REQUIRED "passRequired"
#define MSG_LOGIN_FAILED "loginFailed"
#define MSG_LOGIN_SUCCESS "loginSuccess"
#define MSG_NOT_LOGGED "clientNotLogged"
#define MSG_PWD "currentDirectoryMsg"
#define MSG_LIST_BEGIN "listBegin"
#define MSG_LIST_END "listEnd"
#define MSG_MKD_SUCCESS "mkdSuccess"
#define MSG_MKD_FAILURE "mkdFailed"
#define MSG_RMD_SUCCESS "rmdSuccess"
#define MSG_RMD_FAILED "rmdFailed"
#define MSG_QUIT "quitSuccess"
#define MSG_SYST "systemInfo"
#define MSG_UNKNOWN "unknownCommand"
#define MSG_HELP "commands"

#define DIR_FORMAT "drwxrwxrwx 0 1000 1000 4096 Sep 24 12:34 "

class Config;
class Protocol;
class Client;

/* Creates commands based on the messages received
 * by the client. Uses polymorphic classes */
class Command {
 protected:
    std::string dirname;
```



```
    std::string possible_user;
    std::string possible_pass;

 public:
    virtual void send_response(Config &,
                               Protocol *protocol,
                               Server *,
                               Client &) {}
    virtual ~Command() {}
    static std::string format_cmd(const char * ID,
                                  const char * message,
                                  Config & cfg);
    static std::string format_cmd(const char * ID,
                                  std::string &,
                                  const char * message,
                                  Config & cfg);
};

class CommandCreator : public Command{
 public:
    Command * create_command(std::string &);
    static Command * identify_command(std::string &, std::string &);
};

class UserCommand : public Command{
 public:
    void send_response(Config &cfg,
                       Protocol *protocol,
                       Server *server,
                       Client &cli) override;
    explicit UserCommand(const std::string &arg);
    ~UserCommand() {}
};

class PassCommand : public Command{
 public:
    void send_response(Config &cfg,
                       Protocol *protocol,
                       Server *server,
                       Client &cli) override;
    explicit PassCommand(const std::string &arg);
    bool valid_credentials(Config &, Client &);
    ~PassCommand() {}
};

class SystCommand : public Command{
 public:
    void send_response(Config &cfg,
                       Protocol *protocol,
                       Server *server,
                       Client &cli) override;
    ~SystCommand() {}
};

class ListCommand : public Command{
 public:
    void send_response(Config &cfg,
                       Protocol *protocol,
                       Server *server,
                       Client &cli) override;
    ~ListCommand() override {}
};

class HelpCommand : public Command{
 public:
    void send_response(Config &cfg,
```

```
133                         Protocol *protocol,
134                         Server *server,
135                         Client &cli) override;
136     ~HelpCommand() {}
137 };
138
139 class PwdCommand : public Command{
140  public:
141     void send_response(Config &cfg,
142                         Protocol *protocol,
143                         Server *server,
144                         Client &cli) override;
145     ~PwdCommand() {}
146 };
147
148 class MkdCommand : public Command{
149  public:
150     void send_response(Config &cfg,
151                         Protocol *protocol,
152                         Server *server,
153                         Client &cli) override;
154     explicit MkdCommand(const std::string &arg);
155     ~MkdCommand() {}
156 };
157
158 class RmdCommand : public Command{
159  public:
160     void send_response(Config &cfg,
161                         Protocol *protocol,
162                         Server *server,
163                         Client &cli) override;
164     explicit RmdCommand(const std::string &arg);
165     ~RmdCommand() {}
166 };
167
168 class QuitCommand : public Command{
169  public:
170     void send_response(Config &cfg,
171                         Protocol *protocol,
172                         Server *server,
173                         Client &cli) override;
174     ~QuitCommand() {}
175 };
176
177 class InvalidCommand : public Command{
178  public:
179     void send_response(Config &cfg,
180                         Protocol *protocol,
181                         Server *server,
182                         Client &cli) override;
183     ~InvalidCommand() {}
184 };
185
186 #endif  // SERVER_COMMAND_H
```



```
1  #include "Client.h"
2
3  int main(int argc, char * argv[]) {
4      Client client;
5      client.run_client(argc, argv);
6      return 0;
7  }
```

oct 15, 19 5:35 **Client.h** Page 1/1

```
#ifndef SERVER_CLIENT_H
#define SERVER_CLIENT_H

#include <string>
#include <thread>
#include "Protocol.h"
#define ID_LIST_BEGIN "150"
#define ID_LIST_END "226"
#define ID_QUIT "221"
#define CODE_LENGTH 3

/* Individual client. Used by server to make a
 * client list */
class Client {
 private:
    std::thread * thread;

 public:
    int socket_id;
    bool is_alive = true;
    bool logged = false;
    bool quit = false;
    std::string possible_user;

    Client();
    explicit Client(int id);

    int get_id();
    void print_response(const std::string & received_string) const;
    void run_client(int argc, char *argv[]);
    void receive_hello(Protocol & protocol);
    void receive_list(Protocol & protocol);
    void save_thread(std::thread *);
    void shutdown_socket();
    void close_thread();
    static void get_user_input(std::string & user_input);
    bool quit_received(const std::string & response);
    bool is_list_start(const std::string & str);
    bool is_accepted();
    bool is_dead();
    std::string get_code(std::string const &);
};

#endif  // SERVER_CLIENT_H
```

oct 15, 19 5:35 **Table of Content** Page 1/1

Table of Contents

Referencias

[1] **http://www.cplusplus.com/reference/thread/thread/**

[2] **https://es.cppreference.com/w/cpp/container/set**

[3] **https://es.cppreference.com/w/cpp/container/map**

[4] **http://valgrind.org**

[5] **https://www.gnu.org/software/gdb/**

[6] **http://man7.org/linux/man-pages/man3/getaddrinfo.3.html**

[7] **http://man7.org/linux/man-pages/man2/socket.2.html**

[8] **http://man7.org/linux/man-pages/man2/accept.2.html**

[9] **http://man7.org/linux/man-pages/man2/bind.2.html**

[10] **http://man7.org/linux/man-pages/man2/close.2.html**

[11] **http://man7.org/linux/man-pages/man2/connect.2.html**

[12] **http://man7.org/linux/man-pages/man2/listen.2.html**

[13] **http://man7.org/linux/man-pages/man2/shutdown.2.html**

[14] **http://man7.org/linux/man-pages/man2/send.2.html**

[15] **http://man7.org/linux/man-pages/man2/recv.2.html**