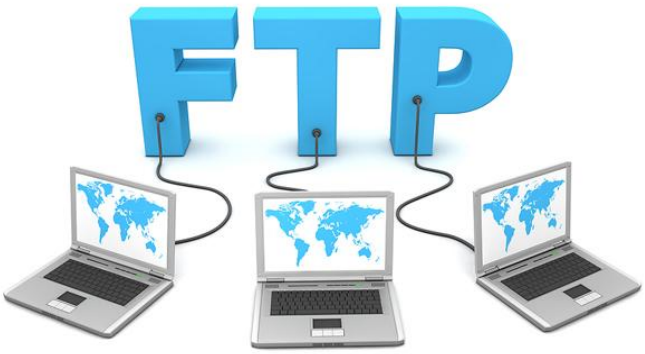
گزارش کار

FTP Server



شبکه‌های کامپیوتری

غزل کلهر (۸۱۰۱۹۶۶۷۵)

محمدامین باقرشاهی (۸۱۰۱۹۷۴۶۴)

فهرست مطالب

مقدمه

در این پروژه به طراحی و پیاده‌سازی یک FTP سرور با قابلیت‌های مدیریت یک فایل سیستم پرداختیم. برای ارائه‌ی خدمات به کاربران از مفاهیم برنامه‌نویسی سوکت که در درس فراگرفته‌ایم بهره برده و یک سیستم کلاینت-سرور طراحی نمودیم.

ساختار پروژه

این پروژه از دو پوشه‌ی src و include تشکیل شده است. در پوشه‌ی include هدر فایل‌های مربوط به هر کلاس قرار گرفته است که در آن‌ها تعریف متدها و فیلدها و نیز کتابخانه‌ها قرار گرفته است. در پوشه‌ی src نیز فایل‌های cpp مربوط به کلاس‌ها آمده است که در آن‌ها بدنه‌ی تمامی متدها قرار گرفته است. در پوشه‌ی configuration نیز فایل config.json قرار گرفته است.

همچنین در پوشه‌ی اصلی پروژه یک MakeFile قرار دارد که از آن برای کامپایل و ساخت فایل‌های اجرایی پروژه بهره می‌گیریم.

نحوه اجرا

برای اجرای پروژه کافی است که با دستور cd ftp-server به پوشه‌ی اصلی پروژه منتقل می‌شویم. سپس با دستور make فایل‌ها را کامپایل می‌کنیم. در این مرحله دو فایل Server.out و Client.out ساخته می‌شوند که با اجرای آن ها در خط فرمان به ترتیب سرور و کلاینت‌ها اجرا می‌شوند. لاگ‌های مربوط به اجرا نیز در یک فایل با نام log.txt در پوشه‌ی اصلی پروژه نوشته می‌شوند.

فایل Utilities

این کلاس برای فراهم آوردن عملیات عمومی موردنیاز روی فایل‌ها که مستقل از اهداف اصلی کلاس‌هاست نوشته شده است.

تابع‌ها

- string read_file_to_string(string file_path)

```
std::string read_file_to_string(std::string file_path) {
    std::ifstream t(file_path);
    std::string str;

    t.seekg(0, std::ios::end);
    str.reserve(t.tellg());
    t.seekg(0, std::ios::beg);

    str.assign((std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>());

    return str;
}
```

این تابع برای خواندن محتوای یک فایل و ذخیره آن در یک رشته نوشته شده است. در ورودی خود آدرس نسبی فایل را دریافت می‌کند و در خروجی نیز رشته مذکور را برمی‌گرداند.

- double read_file_to_double(string file_path)

```
double read_file_to_double(std::string file_path) {
    std::ifstream ifile(file_path, std::ios::in);
    double data;
    ifile >> data;
    return data;
}
```

این تابع برای خواندن محتوای یک فایل که در آن یک عدد از نوع double نوشته شده است و ذخیره آن در یک متغیر از نوع double به کار می‌رود. در ورودی خود آدرس نسبی فایل را دریافت می‌کند و در خروجی نیز عدد مذکور را برمی‌گرداند.

- vector＜string＞ parse_command(char* input)

```
vector<string> parse_command(char* input) {
    vector<string> info;
    char *token = strtok(input, " ");

    while (token != NULL) {
        info.push_back(token);
        token = strtok(NULL, " ");
    }
    return info;
}
```

این تابع برای تجزیه دستور وارد شده از سوی کاربر به اجزای آن شامل اسم دستور و آرگومان‌ها بر حسب white space بین آن‌ها نوشته است. در ورودی خود متن دستور را دریافت می‌کند و در خروجی برداری از رشته‌ها که بیانگر اجزای دستور است را برمی‌گرداند.

- void erase_sub_str(string & main_str, const string & to_erase)

این تابع نیز برای حذف یک زیر رشته از یک رشته نوشته شده است.

کلاس UserIdentityInfo

این کلاس به منظور نگه‌داری مشخصات مربوط به کاربر نوشته شده است.

فیلدها

- username که از نوع string بوده و بیانگر نام کاربری کاربر است.
- password که از نوع string بوده و بیانگر رمز ورود کاربر به سرور است.
- is_admin که از نوع bool بوده و بیانگر سطح دسترسی کاربر است. در صورتی که مقدار آن true باشد کاربر مجاز است که به فایل‌های موجود در .jconfig دسترسی داشته باشد. در غیر این صورت مجاز نخواهد بود.
- available_size که از نوع double بوده و بیانگر حجم مصرفی در دسترس کاربر است.

متدها

- get_username: فیلد username را برمی‌گرداند.
- get_password: فیلد password را برمی‌گرداند.
- get_available_size: فیلد available_size را برمی‌گرداند.
- is_admin_user: فیلد is_admin را برمی‌گرداند.
- is_matched_with: بررسی می‌کند که اطلاعات با نام کاربر و رمز داده شده مطابقت دارد یا خیر.
- decrease_available_size: حجم مصرفی در دسترس کاربر را به میزان داده شده در ورودی کاهش می‌دهد.

کلاس User

این کلاس به منظور نگه‌داری مشخصات کلاینتی است که به سرور وصل شده است نوشته شده است. هنگامی که یک کلاینت به سرور وصل می شود یک instance از این کلاس به لیست کلاینت ها در UserManager اضافه می شود.

فیلدها

- command_socket که از نوع int بوده و مشخص کننده سوکت دستور کاربر است.
- data_socket که از نوع int بوده و مشخص کننده سوکت داده کاربر است.
- state که از نوع State است و مشخص کننده وضعیت login کاربر است.
- current_directory که از نوع string بوده و نگه دارنده پوشه فعلی که کاربر در آن قرار دارد است.
- user_identity_info که از نوع UserIdentityInfo بوده و نگه دارنده اطلاعات هویتی این کلاینت است. این فیلد در صورتی که کاربر login کرده باشد و اطلاعات هویتی اش مشخص شده باشد مقدار دارد و در غیر این صورت nullptr است.

متدها

- get_command_socket: فیلد command_socket را برمیگرداند.
- get_data_socket: فیلد data_socket را برمیگرداند.
- get_state: فیلد state را برمیگرداند.
- get_username: فیلد username از فیلد user_identity_info را برمیگرداند.
- get_current_directory: فیلد current_directory را برمیگرداند.
- get_user_identity_info: فیلد user_identity_info را برمیگرداند.
- set_state: فیلد state را set می کند.
- set_user_identity_info: فیلد user_identity_info را set می کند.
- decrease_available_size: فیلد available_size از فیلد user_identity_info را به اندازه ورودی کم می کند.

```
55  void User::decrease_available_size(double file_size) {
56      user_identity_info->decrease_available_size(file_size);
57  }
```

- is_able_to_download: درصورتی که کاربر سایز کافی برای دانلود داشته باشد true برمی گرداند.

```
59  bool User::is_able_to_access() {
60      return user_identity_info->is_admin_user();
61  }
```

State enum

این enum مشخص کننده وضعیت login است و مقادیر زیر را می تواند بگیرد:

WAITING_FOR_USERNAME, WAITING_FOR_PASSWORD, LOGGED_IN

کلاس UserManager

این کلاس به منظور نگه‌داری و مدیریت کلاینت ها و دسترسی های آن ها است.

فیلدها

- users_identity_info که از نوع vector of UserIdentityInfo بوده و نگه دارنده لیست اطلاعات هویتی کابران است که از configuration خوانده شده و نگه داری می شود.
- users که از نوع vector of User است و نگه دارنده اطلاعات کلاینت هایی است که به سرور وصل شده اند.
- files که از نوع vector of string است و نگه دارنده نام فایل هایی است که از configuration خوانده شده و نگه‌داری می‌شود.

متدها

- UserManager: در constructor با گرفتن configuration به عنوان ورودی اطلاعات آن را گرفته و در خود ذخیره می‌کند.

```
5   UserManager::UserManager(Configuration configuration)
6   : users_identity_info(configuration.get_users_identity_info())
7   , files(configuration.get_files()) {
8   }
```

- add_user: یک instance از User می‌سازد و آن را به لیست users اضافه می‌کند. این تابع در server و هنگام وصل شدن کاربران استفاده می‌شود.

```
19  void UserManager::add_user(int command_socket, int data_socket) {
20      users.push_back(new User(command_socket, data_socket));
21  }
```

- remove_user: این تابع با گرفتن سوکت به عنوان ورودی user متناظر را از لیست حذف می‌کند. این تابع در server و هنگام قطع شدن کاربران استفاده می‌شود.

```
23  void UserManager::remove_user(int socket) {
24      for(size_t i = 0; i < users.size(); ++i) {
25          if (users[i]->get_command_socket() == socket) {
26              users.erase(users.begin() + i);
27              break;
28          }
29      }
30  }
```

- get_user_by_socket: این تابع با گرفتن سوکت به عنوان ورودی user متناظر را پیدا کرده و برمی‌گرداند.

```
32   User* UserManager::get_user_by_socket(int socket) {
33       for(size_t i = 0; i < users.size(); ++i)
34           if (users[i]->get_command_socket() == socket)
35               return users[i];
36       return nullptr;
37   }
```

- get_user_info_by_username: این تابع با گرفتن username به عنوان ورودی user_identity_info متناظر را که حاوی آن username باشد را در صورت وجود برمی‌گرداند.

```
39  UserIdentityInfo* UserManager::get_user_info_by_username(string username) {
40      for(size_t i = 0; i < users_identity_info.size(); ++i)
41          if (users_identity_info[i]->get_username() == username)
42              return users_identity_info[i];
43      return nullptr;
44  }
```

کلاس Configuration

این کلاس به منظور خواندن و parse کردن فایل json استفاده می شود. در constructor مسیر فایل json را میگیرد و پس از خواندن آن اطلاعات را در خود ذخیره می‌کند.

فیلدها

- command_channel_port که از نوع int است و نگه دارنده پورت کانال دستور است.
- data_channel_port که از نوع int است و نگه دارنده پورت کانال داده است.

- users_identity_info که از نوع vector of UserIdentityInfo است و نگه دارنده اطلاعات هویتی کاربران است.
- files که از نوع vector of string است و نگه دارنده نام فایل‌هایی است که از فایل کانفیگ خوانده شده است.

متد ها

- Configuration: در constructor فایل با آدرسی که در ورودی داده می‌شود خوانده شده و parse می‌شود و سپس اطلاعات استخراج شده در فیلد ها ذخیره می‌شود.

```
9   Configuration::Configuration(const string path) {
10      namespace pt = boost::property_tree;
11
12      pt::ptree root_tree;
13      pt::read_json(path, root_tree);
14
15      command_channel_port = root_tree.get_child("commandChannelPort").get_value<int>();
16      data_channel_port = root_tree.get_child("dataChannelPort").get_value<int>();
17
18      pt::ptree users_tree = root_tree.get_child("users");
19      for (auto& item : users_tree.get_child("")) {
20          string name = item.second.get<string>("user");
21          string password = item.second.get<string>("password");
22          bool is_admin = item.second.get<bool>("admin");
23          double size = item.second.get<double>("size");
24
25          users_identity_info.push_back(new UserIdentityInfo(name, password, is_admin, size));
26      }
27
28      pt::ptree files_tree = root_tree.get_child("files");
29      for (auto& item : files_tree.get_child(""))
30          files.push_back(item.second.get_value<string>());
31  }
```

- get_command_channel_port: فیلد command_channel_port را برمی‌گرداند.
- get_data_channel_port: فیلد data_channel_port را برمی‌گرداند.
- get_users_identity_info: فیلد users_identity_info را برمی‌گرداند.
- get_files: فیلد files را برمی‌گرداند.

کلاس Logger

این کلاس برای ثبت (log) اعمال انجام شده توسط کاربران در سیستم مانند ورود و خروج، دانلود یا حذف یا تغییر نام فایل نوشته شده است.

فیلدها

- path که از نوع string است و بیانگر مسیر فایل log.txt است.

متدها

- Logger(std::string path)

```
Logger::Logger(string path)
: path(path) {
    fstream log_file;
    log_file.open(path, std::fstream::in | std::fstream::out | std::fstream::app);

    if (!log_file) {
        cout << "Cannot open file, log file does not exist. Creating new file..";
        log_file.open(path,  fstream::in | fstream::out | fstream::trunc);
        log_file.close();
    }
}
```

این متد در اصل کانستراکتور این کلاس محسوب می‌شود. در ابتدا مسیر فایل log.txt در فیلد path مربوط به کلاس ذخیره می‌شود. سپس یک متغیر به اسم log_file از نوع fstream تعریف شده است. به کمک این متغیر و تابع open وجود فایل log.txt بررسی می‌شود. در صورتی که وجود نداشت به کمک دستور open ساخته می‌شود و سپس بسته می‌شود.

- void log(std::string message)

```
void Logger::log(string message) {
    fstream log_file;
    log_file.open(path, std::fstream::in | std::fstream::out | std::fstream::app);

    auto curr = std::chrono::system_clock::now();
    std::time_t curr_time = std::chrono::system_clock::to_time_t(curr);

    log_file << std::ctime(&curr_time);
    log_file << message << endl;
    log_file.close();
}
```

این متد برای ثبت وقایع در فایل log.txt نوشته شده است. در ورودی خود یک پیام از نوع string دریافت می‌کند. سپس یک متغیر با نام log_file از نوع fstream تعریف می‌کند تا به کمک آن فایل log.txt را حالت append باز کند که نوشتن وقایع در ادامه آن صورت گیرد.

علاوه بر محتوای پیام باید زمان آن نیز ثبت شود. به این منظور یک متغیر با نام curr_time از نوع time_t تعریف شده است که زمان جاری شمال ساعت و تاریخ در آن ذخیره می‌شود. در نهایت نیز زمان ثبت واقعه و محتوای آن به فایل افزوده می‌شود. و فایل بسته می‌شود.

کلاس CommandHandler

این کلاس برای انجام دستورات وارد شده به سرور از سوی کاربر نوشته شده است.

فیلدها

- user_manager که از نوع پوینتری به UserManager است و از آن برای مدیریت کاربران استفاده می‌شود.
- logger که از نوع پوینتری به Logger است از آن برای ثبت وقایع حین اجرای دستورات استفاده می‌شود.

متدها

- CommandHandler(Configuration configuration, Logger* logger)

```
CommandHandler::CommandHandler(Configuration configuration, Logger* logger) {
    user_manager = new UserManager(configuration);
    this->logger = logger;
}
```

این متد در واقع کانستراکتوری برای این کلاس است. در ورودی خود آبجکتی از نوع Configuration و پوینتری به Logger دریافت می‌کند. به وسیله configuration که پارامتر ورودی کانستراکتور UserManager است فیلد user_manager خود را مقداردهی اولیه می‌کند. به کمک logger نیز پوینتر خود به Logger را تنظیم می‌کند.

- UserManager* get_user_manager()

این متد در اصل getter ای است که فیلد user_manager این کلاس را برمی‌گرداند که از نوع پوینتری به UserManager است.

- bool is_a_file_name(string file_name)

```
bool CommandHandler::is_a_file_name(string file_name) {
    if (file_name.find(BACK_SLASH) != std::string::npos)
        return false;
    return true;
}
```

این متد بررسی می‌کند که رشته‌ی ورودی آن فرمت قابل قبول برای نام یک فایل را دارد یا خیر. به این منظور بررسی می‌کند که در آن علامت "/" که برای آدرس‌ها در سیستم لینوکس به کار می‌رود وجود نداشته باشد.

- bool user_has_access_to_file(string file_name, User* user)

```
bool CommandHandler::user_has_access_to_file(string file_name, User* user) {
    if (!user_manager->contains_as_special_file(file_name))
        return true;
    else if (user->is_able_to_access())
        return true;
    return false;
}
```

این متد بررسی می‌کند که کاربر موردنظر اجازه‌ی دسترسی به فایلی که نام به عنوان ورودی داده شده است را دارد یا خیر. ابتدا با فراخوانی متد contains_as_special_file بر روی user_manager بررسی می‌کند که آیا نام داده شده در لیست فایل‌های موجود در فایل config.json قرار دارد یا خیر. اگر قرار نداشت که نیاز به ادامه‌ی بررسی وجود ندارد و هر نوع کاربری می‌تواند به آن فایل دسترسی یابد ولی در غیر این صورت ادمین بودن کاربر با فراخوانی متد is_able_to_access روی user بررسی می‌شود.

- vector<string> do_command(int user_socket, char* command)

```
vector<string> CommandHandler::do_command(int user_socket, char* command) {
    vector<string> command_parts = parse_command(command);

    User* user = user_manager->get_user_by_socket(user_socket);
    if (user == nullptr)
        return {GENERAL_ERROR, EMPTY};

    if (command_parts[COMMAND] == USER_COMMAND) {
        if (command_parts.size() != 2)
            return {SYNTAX_ERROR, EMPTY};
        return handle_username(command_parts[ARG1], user);
    }
```

این متد برای مدیریت دستورات وارد شده از سوی یک کاربر و فراخوانی هندلر موردنظر نوشته شده است. در ورودی خود سوکت کاربر و دستور وارد شده (همراه آرگومان‌ها) را دریافت می‌کند. در خروجی خود نیز برداری به طول ۲ از نوع string برمی‌گرداند که در اندیس صفرم آن پیام مربوط به کانال دستور و در اندیس اول آن دیتای کانال داده نوشته شده است.

با از تابع parse_command این دستور را به اجزای آن تجزیه می‌کند و در یک بردار از نوع string به نام command_parts ذخیره می‌کند. با فراخوانی متد get_user_by_socket بر روی user_manager نیز پوینتر مربوط به این کاربر را در صورت وجود در user ذخیره می‌کند. در صورتی که این تابع مقدار NULL برگرداند به این معنی است که کاربری متناظر با این سوکت موجود نیست و پیام خطای مورد نظر آن برگردانده می‌شود.

در مرحله بعدی با توجه به بخش command این دستور ابتدا تعداد آرگومان‌ها بررسی می‌شود که در حالتی که متناسب با دستور نبود خطای سینتکس در کانال دستور برگردانده شود. در غیر این صورت نیز هندلر متناظر با آن با آرگمان‌های دستور فراخوانی می‌شود. به عنوان مثال این قسمت از کد برای دستور USER آورده شده است. بقیه دستورات نیز به همین ترتیب بررسی خواهند شده. البته در دستور DELE آرگومان اول آن که option مربوط به فایل/دایرکتوری است نیز بررسی می‌شود تا هندلر متناظر با آن فراخوانی شود.

در انتها نیز در صورتی که دستور وارد شده با هیچیک از دستورات تعریف شده در سیستم انطباق نداشت خطای سینتکس در کانال دستور برگردانده می‌شود.

- vector<string> handle_username(string username, User* user)

```
vector<std::string> CommandHandler::handle_username(string username, User* user) {
    if(user->get_state() != User::State::WAITING_FOR_USERNAME)
        return {BAD_SEQUENCE, EMPTY};

    UserIdentityInfo* user_identity_info = user_manager->get_user_info_by_username(username);

    if (user_identity_info == nullptr)
        return {INVALID_USER_PASS, EMPTY};

    user->set_state(User::State::WAITING_FOR_PASSWORD);
    user->set_user_identity_info(user_identity_info);
    user->set_current_directory(ROOT);

    return {USERNAME_ACCEPTED, EMPTY};
}
```

این متد در واقع هندلری برای دستور USER است. در ورودی خود username که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا بررسی می‌شود که کاربر موردنظر در وضعیت "انتظار برای نام کاربری" قرار دارد یا خیر. اگر در این وضعیت نبود خطای "ترتیب نادرست دستور" در کانال دستور برگردانده می‌شود.

در ادامه صحت نام کاربری وارد شده بررسی می‌شود. به این منظور متد get_user_info_by_username روی user_manager فراخوانی می‌شود. در صورتی که کاربر با این نام کاربری وجود نداشته باشد مقدار NULL توسط آن برگردانده می‌شود. در این حالت خطای "نامعتبر بودن نام کاربری یا رمز" در کانال دستور برگردانده می‌شود.

در نهایت نیز وضعیت کاربر به "انتظار برای رمز" تنظیم می‌شود و اطلاعات آن نیز به پوینتر موردنظر نسبت داده می‌شود و دایرکتوری فعلی آن به ریشه تنظیم می‌شود. در آخر نیز پیام "پذیرش نام کاربری" در کانال دستور برگردانده می‌شود.

- vector<string> handle_password(string password, User* user)

```
vector<std::string> CommandHandler::handle_password(string password, User* user) {
    if(user->get_state() != User::State::WAITING_FOR_PASSWORD)
        return {BAD_SEQUENCE, EMPTY};

    if (user->get_user_identity_info()->get_password() != password)
        return {INVALID_USER_PASS, EMPTY};

    user->set_state(User::State::LOGGED_IN);

    logger->log(user->get_username() + COLON + "logged in.");

    return {SUCCESSFUL_LOGIN, EMPTY};
}
```

این متد هندلری برای دستور PASS است. در ورودی خود password که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا بررسی می‌شود که کاربر موردنظر در وضعیت "انتظار برای رمز" قرار دارد یا خیر. اگر در این وضعیت نبود خطای "ترتیب نادرست دستور" در کانال دستور برگردانده می‌شود.

در ادامه صحت نام رمز وارد شده بررسی می‌شود. به این منظور متد get_password روی user_identity_info متناظر با کاربر فراخوانی می‌شود و مقدار بازگشتی آن با رمز وارد شده مقایسه می‌شود. در صورتی که مقدار آن انطباق نداشت خطای "نامعتبر بودن نام کاربری یا رمز" در کانال دستور برگردانده می‌شود.

در نهایت نیز وضعیت کاربر به "وارد شده" تنظیم می‌شود و ورود کاربر نیز توسط با فراخوانی متد log بر روی logger ثبت می‌شود. در آخر نیز پیام "ورود موفقیت آمیز" در کانال دستور برگردانده می‌شود.

- vector<string> handle_get_current_directory(User* user)

```
vector<string> CommandHandler::handle_get_current_directory(User* user) {
    string current_path = user->get_current_directory();
    if (current_path == ROOT)
        current_path = ".";

    string bash_command = "realpath " + current_path + " > file.txt";
    int status = system(bash_command.c_str());
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    string result = read_file_to_string("file.txt");
    result.pop_back();
    status = system("rm file.txt");
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    return {"257: " + result, EMPTY};
}
```

این متد در اصل هندلری برای دستور PWD است. در ورودی خود user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

با فراخوانی متد get_current_directory بر روی user آدرس نسبی آن در current_path ذخیره می‌شود. سپس با استفاده از دستوری از bash به نام realpath که در ورودی خود یک آدرس نسبی را دریافت می‌کند آدرس مطلق آن را دریافت می‌کنیم و در یک فایل موقتی به نام file.txt ذخیره می‌کنیم زیرا این دستور در خروجی خود فقط وضعیت را برمی‌گرداند. سپس محتوای این فایل را خوانده و در یک رشته به نام result ذخیره می‌کنیم. در نهایت نیز با دستور دیگری از bash به نام rm فایل موقتی را حذف می‌کنیم. در صورتی که تمامی این مراحل به درستی انجام شد آدرس مطلق را در کانال دستور برمی‌گرداند.

- vector<string> handle_create_new_directory(string dir_path, User* user)

```
vector<string> CommandHandler::handle_create_new_directory(string dir_path, User* user) {
    string bash_command = "mkdir " + user->get_current_directory() + dir_path;
    int status = system(bash_command.c_str());
    if (status == SUCCESS) {
        string message = COLON + dir_path + " created.";
        logger->log(user->get_username() + message);
        return {CREATE_CODE + message, EMPTY};
    }
    return {GENERAL_ERROR, EMPTY};
}
```

این متد هندلری برای دستور MKD است. در ورودی خود dir_path که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

با استفاده از دستوری از bash به نام mkdir که آدرس نسبی دایرکتوری موردنظر را به عنوان آرگومان خود دریافت می‌کند دایرکتوری موردنظر را می‌سازیم. آرگومان آن نیز از کنار هم قرار دادن آدرس فعلی کاربر و آدرس نسبی داده شده در ورودی متد محاسبه می‌شود.در صورتی که تمامی این مراحل به درستی انجام شد پیام ایجاد دایرکتوری موردنظر در کانال دستور برگردانده می‌شود و توسط logger نیز ثبت می‌شود.

- vector<string> handle_delete_directory(string dir_path, User* user)

```
vector<string> CommandHandler::handle_delete_directory(string dir_path, User* user) {
    string bash_command = "rm -r " + user->get_current_directory() + dir_path;
    int status = system(bash_command.c_str());
    if (status == SUCCESS) {
        string message = COLON + dir_path + " deleted.";
        logger->log(user->get_username() + message);
        return {DELETE_CODE + message, EMPTY};
    }
    return {GENERAL_ERROR, EMPTY};
}
```

این متد هندلری برای دستور DELE با آرگومان اول -d است. در ورودی خود dir_path که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

با استفاده از دستوری از bash به نام rm -r که آدرس نسبی دایرکتوری موردنظر را به عنوان آرگومان خود دریافت می‌کند دایرکتوری موردنظر را حذف می‌کنیم. آرگومان آن نیز از کنار هم قرار دادن آدرس فعلی کاربر و آدرس نسبی داده شده در ورودی متد محاسبه می‌شود.در صورتی که تمامی این مراحل به درستی انجام شد پیام حذف دایرکتوری موردنظر در کانال دستور برگردانده می‌شود و توسط logger نیز ثبت می‌شود.

- vector<string> handle_delete_file(string file_name, User* user)

```
vector<string> CommandHandler::handle_delete_file(string file_name, User* user) {
    if (!is_a_file_name(file_name))
        return {SYNTAX_ERROR, EMPTY};

    if (!user_has_access_to_file(file_name, user))
        return {FILE_UNAVAILABLE, EMPTY};

    string bash_command = "rm " + user->get_current_directory() + file_name;
    int status = system(bash_command.c_str());
    if (status == SUCCESS) {
        string message = COLON + file_name + " deleted.";
        logger->log(user->get_username() + message);
        return {DELETE_CODE + message, EMPTY};
    }
    return {GENERAL_ERROR, EMPTY};
}
```

این متد هندلری برای دستور DELE با آرگومان اول -f است. در ورودی خود file_name که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا به کمک متد is_a_file_name بررسی می‌شود که نام فایل داده شده فرمت درستی داشته باشد. در غیر این صورت خطای سینتکس در کانال دستور برگردانده می‌شود.

سپس به کمک فراخوانی متد user_has_access_to_file مجاز بودن دسترسی کاربر به فایل موردنظر بررسی می‌شود. در صورتی که مجاز نبود خطای "در دسترس نبودن فایل" در کانال دستور برگردانده می‌شود.

با استفاده از دستوری از bash به نام rm که آدرس نسبی فایل موردنظر را به عنوان آرگومان خود دریافت می‌کند فایل موردنظر را حذف می‌کنیم. آرگومان آن نیز از کنار هم قرار دادن آدرس فعلی کاربر و نام فایل داده شده در ورودی متد محاسبه می‌شود.در صورتی که تمامی این مراحل به درستی انجام شد پیام حذف فایل موردنظر در کانال دستور برگردانده می‌شود و توسط logger نیز ثبت می‌شود.

- vector<string> handle_get_list_of_files(User* user)

```
vector<string> CommandHandler::handle_get_list_of_files(User* user) {
    string bash_command = "ls " + user->get_current_directory() + " > file.txt";
    int status = system(bash_command.c_str());
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    string result = read_file_to_string("file.txt");
    result.pop_back();
    status = system("rm file.txt");
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    erase_sub_str(result, "file.txt\n");

    return {LIST_TRANSFER_DONE, result};
}
```

این متد هندلری برای دستور LS است. در ورودی خود user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

با استفاده از دستوری از bash به نام ls که در ورودی خود یک آدرس نسبی را دریافت می‌کند لیست فایل‌ها و دایرکتوری‌ها را دریافت می‌کنیم و در یک فایل موقتی به نام file.txt ذخیره می‌کنیم زیرا این دستور در خروجی خود فقط وضعیت را برمی‌گرداند. سپس محتوای این فایل را خوانده و در یک رشته به نام result ذخیره می‌کنیم. سپس نام این فایل موقتی را از نتیجه حذف می‌کنیم. در نهایت نیز با دستور دیگری از bash به نام rm فایل موقتی را حذف می‌کنیم. در صورتی که تمامی این مراحل به درستی انجام شد لیست فایل‌ها را در قالب یک رشته در کانال داده برمی‌گرداند. پیام "انجام انتقال لیست" در کانال دستور برگردانده می‌شود.

- vector<string> handle_change_working_directory(string dir_path, User* user)

```
std::vector<std::string> CommandHandler::handle_change_working_directory(string dir_path,
User* user) {
    string check_validity_command = "realpath " + dir_path + " > file.txt";
    int status1 = system(check_validity_command.c_str());
    int status2 = system("rm file.txt");
    if (status1 != SUCCESS || status2 != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    if(dir_path == ROOT)
        user->set_current_directory(ROOT);
    else
        user->set_current_directory(user->get_current_directory() + dir_path + "/");

    return {SUCCESSFUL_CHANGE, EMPTY};
}
```

این متد هندلری برای دستور CWD است. در ورودی خود dir_path که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

در صورتی که مسیر داده شده رشته خالی بود (بیانگر ریشه) دایرکتوری فعلی کاربر به ریشه یعنی پوشه‌ای که در آن پروژه اجرا می‌شود تنظیم خواهد شد. در غیر این صورت آدرس نسبی داده شده با آدرس فعلی آن جمع می‌شود و آدرس فعلی آن با این مقدار جایگزین می‌شود.

توجه داریم که صحت آدرس داده شده نیز باید بررسی شود که به این منظور از یک دستور bash به نام realpath استفاده شده است و از وضعیت برگردانده شده توسط آن به معتبر بودن آدرس پی می‌بریم.

در نهایت نیز در صورت برقرار بودن تمام شرایط پیام "تغییر موفقیت آمیز" در کانال دستور برگردانده می‌شود.

- vector<string> handle_rename_file(string old_name, string new_name, User* user)

```
std::vector<std::string> CommandHandler::handle_rename_file(string old_name, string new_name,
    User* user) {
    if (!is_a_file_name(old_name) || !is_a_file_name(new_name))
        return {SYNTAX_ERROR, EMPTY};

    if (!user_has_access_to_file(old_name, user))
        return {FILE_UNAVAILABLE, EMPTY};

    string bash_command = "mv " + user->get_current_directory() + old_name + " " +
            user->get_current_directory() + new_name;
    int status = system(bash_command.c_str());
    if (status == SUCCESS)
        return {SUCCESSFUL_CHANGE, EMPTY};
    return {GENERAL_ERROR, EMPTY};
}
```

این متد هندلری برای دستور RENAME است. در ورودی خود old_name و new_name که به ترتیب آرگومان‌های اول و دوم این دستور هستند و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا به کمک متد is_a_file_name بررسی می‌شود که اسامی فایل‌های داده شده فرمت درستی داشته باشد. در غیر این صورت خطای سینتکس در کانال دستور برگردانده می‌شود.

سپس به کمک فراخوانی متد user_has_access_to_file مجاز بودن دسترسی کاربر به فایل موردنظر بررسی می‌شود. در صورتی که مجاز نبود خطای "در دسترس نبودن فایل" در کانال دستور برگردانده می‌شود.

با استفاده از دستوری از bash به نام mv که آدرس نسبی فایل موردنظر را همراه نام قبلی و جدید آن به عنوان آرگومان‌های اول و دوم خود دریافت می‌کند نام فایل موردنظر را تغییر می‌دهیم. آرگومان‌های آن نیز از کنار هم قرار دادن آدرس فعلی کاربر و نام‌های قبلی و جدید فایل داده شده در ورودی متد محاسبه می‌شود. در صورتی که تمامی

این مراحل به درستی انجام شد پیام "تغییر موفقیت آمیز" در کانال دستور برگردانده می‌شود و توسط logger نیز ثبت می‌شود.

- vector<string> handle_download_file(string file_name, User* user)

```
std::vector<std::string> CommandHandler::handle_download_file(string file_name, User* user) {
    if (!is_a_file_name(file_name))
        return {SYNTAX_ERROR, EMPTY};

    if (!user_has_access_to_file(file_name, user))
        return {FILE_UNAVAILABLE, EMPTY};

    string file_path = user->get_current_directory() + file_name;
    string size_command = "stat -c%s " + file_path + " > " + "size.txt";
    int status = system(size_command.c_str());
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    double file_size = read_file_to_double("size.txt");
    status = system("rm size.txt");
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    if (user -> is_able_to_download(file_size) == false)
        return {DOWNLOAD_LIMIT_SIZE, EMPTY};
```

این متد هندلری برای دستور RETR است. در ورودی خود file_name که آرگومان این دستور است و نیز user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا به کمک متد is_a_file_name بررسی می‌شود که نام فایل داده شده فرمت درستی داشته باشد. در غیر این صورت خطای سینتکس در کانال دستور برگردانده می‌شود.

سپس به کمک فراخوانی متد user_has_access_to_file مجاز بودن دسترسی کاربر به فایل موردنظر بررسی می‌شود. در صورتی که مجاز نبود خطای "در دسترس نبودن فایل" در کانال دستور برگردانده می‌شود.

سپس با دستوری از bash به نام stat سایز فایل موردنظ بر حسب بایت در یک فایل موقتی به نام size.txt ذخیره می‌شود. به کمک متد read_file_to_double محتوای این فایل خوانده شده و در یک متغیر از نوع double به نام file_size ذخیره می‌شود. حال با فراخوانی متد is_able_to_download بر روی user بررسی می‌کنیم که او حجم کافی برای دانلود فایل داشته باشد. در غیر این صورت خطای ناکافی بودن حجم را در کانال دستور برمی‌گردانیم.

```
    string bash_command = "cp " + file_path + " file.txt";
    status = system(bash_command.c_str());
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    string result = read_file_to_string("file.txt");
    status = system("rm file.txt");
    if (status != SUCCESS)
        return {GENERAL_ERROR, EMPTY};

    user->decrease_available_size(file_size);

    string message = COLON + file_name + " downloaded.";
    logger->log(user->get_username() + message);

    return {SUCCESSFUL_DOWNLOAD, result};
}
```

در ادامه به کمک دستوری از bash به نام cp محتوای فایل موردنظر را در یک فایل موقتی به نام file.txt ذخیره می‌کنیم. این دستور در آرگومان خود آدرس نسبی فایل موردنظر و نام فایل موقتی را می‌گیرد. سپس به کمک تابع read_file_to_string محتوای این فایل را در رشته به نام result ذخیره می‌کنیم. در گام بعدی به کمک متد decrease_available_size حجم در دسترس کاربر را به میزان حجم فایل کاهش می‌دهیم. در صورتی که تمامی این مراحل به درستی انجام شد پیام "دانلود موفقیت آمیز" در کانال دستور برگردانده می‌شود و توسط logger نیز ثبت می‌شود.

- vector<string> handle_help()

```
std::vector<std::string> CommandHandler::handle_help() {
    string info = "214\n";
    info += USER_DESCRIPTION;
    info += PASS_DESCRIPTION;
    info += PWD_DESCRIPTION;
    info += MKD_DESCRIPTION;
    info += DELE_DESCRIPTION;
    info += LS_DESCRIPTION;
    info += CWD_DESCRIPTION;
    info += RENAME_DESCRIPTION;
    info += RETR_DESCRIPTION;
    info += HELP_DESCRIPTION;
    info += QUIT_DESCRIPTION;
    return {info, EMPTY};
}
```

این متد هندلری برای دستور HELP است. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

در این متد ابتدا کد دستور به همراه توضیح مربوط به هر دستور که در هدر فایل تعریف شده‌اند کنار یکدیگر در یک رشته قرار می‌گیرند. سپس این رشته در کانال داده برگردانده می‌شود.

- vector<string> handle_logout(User* user)

```
vector<string> CommandHandler::handle_logout(User* user) {
    if (user->get_state() != User::State::LOGGED_IN)
        return {GENERAL_ERROR, EMPTY};

    user->set_state(User::State::WAITING_FOR_USERNAME);

    logger->log(user->get_username() + COLON + "logged out.");

    return {SUCCESSFUL_QUIT, EMPTY};
}
```

این متد در اصل هندلری برای دستور QUIT است. در ورودی خود user که پوینتری به کاربری است که دستور را وارد کرده دریافت می‌کند. در خروجی خود نیز مشابه متد do_command پیام متناظر با کانال‌های دستور و داده را برمی‌گرداند.

ابتدا بررسی می‌شود که کاربر موردنظر در وضعیت "وارد شده" قرار دارد یا خیر. اگر در این وضعیت نبود در کانال دستور خطا برگردانده می‌شود.

در نهایت نیز وضعیت کاربر به "انتظار برای نام کاربری" تنظیم می‌شود و خروج کاربر نیز توسط با فراخوانی متد log بر روی logger ثبت می‌شود. در آخر نیز پیام "خروج موفقیت آمیز" در کانال دستور برگردانده می‌شود.

کلاس Server

این کلاس وظیفه برقراری ارتباط با کلاینت ها، دریافت دستور های آن ها، انتقال دستور ها به command_handler و ارسال خروجی command_handler برای آن دستور ها به کلاینت ها را بر عهده دارد.

فیلدها

- command_handler که ثابتی از نوع CommandHandler است و وظیفه هندل کردن دستوراتی که از سمت کلاینت ها می آید را به عهده دارد.
- logger که از نوع Logger است و وظیفه ثبت log ها در هنگام وقوع اتفاق های مختلف را به عهده دارد.
- command_channel_port که از نوع int است و پورت کانال دستور را مشخص می‌کند.
- data_channel_port که از نوع int است و پورت کانال داده را مشخص می‌کند.

متد ها

- Server: در constructor با گرفتن configuration به عنوان ورودی فیلدهای command_channel_port و data_channel_port مقداردهی می‌شوند و همچنین command_handler و logger نیز ساخته می‌شوند.

```
6  Server::Server(Configuration configuration)
7  : command_channel_port(configuration.get_command_channel_port())
8  , data_channel_port(configuration.get_data_channel_port()) {
9      logger = new Logger(LOG_FILE);
10     command_handler = new CommandHandler(configuration, logger);
11 }
```

- start: در این متد که متد اصلی کلاس server است در ابتدا سوکت هایی برای کانال دستور و داده ساخته می شود.

```
18      int command_server_fd = socket(AF_INET, SOCK_STREAM, 0);
19      int data_server_fd = socket(AF_INET, SOCK_STREAM, 0);
20      if (command_server_fd < 0 || data_server_fd < 0)
21          return;
```

و در ادامه bind به پورت هایی که از کانفیگ خوانده شده است انجام می شود و سپس listen انجام می شود.

```
38      if (bind(command_server_fd, (struct sockaddr *)&command_address, sizeof(command_address)) < 0)
39          return;
40      if (bind(data_server_fd, (struct sockaddr *)&data_address, sizeof(data_address)) < 0)
41          return;
42
43      constexpr int backlog = 10;
44      if (listen(command_server_fd, backlog) < 0)
45          return;
46      if (listen(data_server_fd, backlog) < 0)
47          return;
```

در ادامه در یک حلقه بی انتها داریم که تابع select در آن صدا زده شده است. در صورتی که یک کلاینت جدید connect شود با استفاده از تابع UserManager::add_user یک user به لیست users که در user_manager است اضافه می شود.

```
// New connection.
if (fd == command_server_fd) {
    int new_command_socket = accept(command_server_fd, NULL, NULL);
    if (new_command_socket < 0)
        return;

    int new_data_socket = accept(data_server_fd, NULL, NULL);
    if (new_data_socket < 0)
        return;

    command_handler->get_user_manager()->add_user(new_command_socket, new_data_socket);
    FD_SET(new_command_socket, &copy_fds);
    if (new_command_socket > max_fd)
        max_fd = new_command_socket;
}
```

همچنین در صورت آمدن یک دستور از سمت کلاینت‌ها با فراخوانی تابع recv دستور دریافت می‌شود و به تابع CommandHandler::do_command داده می شود تا هندل شود. در ادامه خروجی های برگردانده

شده از command_handler (خروجی دستور و داده) از طریق کانال های دستور و داده به کلاینت مورد نظر ارسال می‌شود.

```
// New readable socket.
else {
    bool close_connection = false;
    memset(received_buffer, 0, sizeof received_buffer);
    int result = recv(fd, received_buffer, sizeof(received_buffer), 0);

    if (result < 0)
        if (errno != EWOULDBLOCK)
            close_connection = true;

    // Check to see if the connection has been closed by client.
    if (result == 0)
        close_connection = 1;

    // Data is received.
    if (result > 0) {
        vector<string> output = command_handler->do_command(fd, received_buffer);

        send(fd , output[COMMAND].c_str() , output[COMMAND].size() , 0);
        send(command_handler->get_user_manager()->get_user_by_socket(fd)->get_data_socket(),
                output[CHANNEL].c_str() , output[CHANNEL].size() , 0);
    }
```

همچنین در صورتی که یک یوزر قطع شود سوکت هایش close می شود و یوزر متناظر از لیست users در user_manager حذف می شود.

```
if (close_connection) {
    close(fd);
    close(command_handler->get_user_manager()->get_user_by_socket(fd)->get_data_socket());
    command_handler->get_user_manager()->remove_user(fd);
    FD_CLR(fd, &copy_fds);
    if (fd == max_fd)
        while (FD_ISSET(max_fd, &copy_fds) == 0)
            max_fd -= 1;
}
```

نکات

- برای دریافت دستور هایی که از سمت کلاینت می آید از یک buffer با سایز 2048 استفاده شده است.

کلاس Client

این کلاس وظیفه برقراری ارتباط با سرور، دریافت دستورات از command line، ارسال آن به سرور و دریافت خروجی ها را به عهده دارد.

فیلدها

- MAX_COMMAND_LENGTHZ که ثابتی از نوع int است که بیشترین طول دستور را مشخص می کند.

متد ها

- start: در این متد که متد اصلی کلاس client است در ابتدا سوکت هایی برای کانال های دستور و داده ساخته می شود.

```
16  void Client::start(int command_channel_port, int data_channel_port) {
17      int client_command_fd = socket(AF_INET, SOCK_STREAM, 0);
18      int client_data_fd = socket(AF_INET, SOCK_STREAM, 0);
19      if (client_command_fd < 0 || client_data_fd < 0)
20          return;
```

و در ادامه این دو سوکت با سرور connect می شوند (پورت های کانال دستور و داده ای که به آن کانکت می شود در ورودی داده شده است).

```
34      if (connect(client_command_fd, (struct sockaddr*)&server_command_address, sizeof(server_command_address)) < 0)
35          return;
36
37      if (connect(client_data_fd, (struct sockaddr*)&server_data_address, sizeof(server_data_address)) < 0)
38          return;
```

و ادامه در یک حلقه بی انتها هر بار یک دستور گرفته می شود و به سمت سرور ارسال می شود و سپس جواب دستور و داده از طریق کانال های دستور و داده گرفته می شود و نمایش داده می شود.

```
40      char received_command_output[2048] = {0};
41      char received_data_output[4048] = {0};
42      while (true) {
43          // Receive command from command line.
44          cout << "> ";
45          char command[MAX_COMMAND_LENGTH];
46          memset(command, 0, MAX_COMMAND_LENGTH);
47          cin.getline (command, MAX_COMMAND_LENGTH);
48
49          // Send command to server.
50          send(client_command_fd, command, MAX_COMMAND_LENGTH, 0);
51
52          // Receive command output.
53          memset(received_command_output, 0, sizeof received_command_output);
54          recv(client_command_fd, received_command_output, sizeof(received_command_output), 0);
55          cout << "Command output: " << received_command_output << endl;
56
57          // Receive data output.
58          memset(received_data_output, 0, sizeof received_data_output);
59          recv(client_data_fd, received_data_output, sizeof(received_data_output), 0);
60          cout << "Data output: " << received_data_output << endl;
61      }
```

نکات

- برای ارسال دریافت خروجی دستور و داده به ترتیب از buffer هایی با سایز های 2048 و 4096 استفاده شده است.

نتیجه‌گیری

در این پروژه امکانات متداول یک فایل سیستم را پیاده‌سازی نمودیم و با یکی از کاربردهای برنامه‌نویسی سوکت آشنا شدیم. همچنین به طور گسترده با نحوه‌ی کار با کتابخانه‌های زبان C++ برای ایجاد یک سیستم کلاینت - سرور آشنا شدیم. در انتها نیز به اهمیت مفاهیم شبکه‌های کامپیوتری در طراحی خدمات مربوط به سیستم‌های کامپیوتری پی بردیم.