به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

شبکه‌های کامپیوتری

گزارش پروژه اول

سجاد علی‌زاده – ۸۱۰۱۹۷۵۴۷

سارینا همدانی - ۸۱۰۱۹۷۶۰۶

ساختار پروژه

این پروژه سه پوشه اصلی دارد. پوشه bin که فایل‌های قابل اجرا در آن قرار می‌گیرد. پوشه config که فایل کانفیگ مربوط به سرور و کلاینت در آن قرار می‌گیرد. پوشه src که کدهای اصلی برنامه در آن قرار میگیرد. همچنین در کنار این پوشه‌ها Makefile قرار دارد که برای تولید فایل‌های قابل اجرا، اجرا کردن آن کافیست.

برای اجرا کردن سرور به شکل زیر عمل می‌کنیم:

```
./bin/Server.out config/config.json bin/server_files/
```

آرگومان اول آن آدرس فایل کانفیگ و آرگومان دوم آن آدرس پوشه مربوط به سرور ftp است. (همان پوشه‌ای که کلاینت‌ها با وصل شدن به سرور به آن‌ها دسترسی پیدا میکنند) که فایل‌های سرور در آن قرار دارد.

توجه کنید فایل log.txt که لاگ سرور است در همان محلی که از آنجا سرور را اجرا می‌کنیم ساخته می‌شود.

برای اجرا کردن کلاینت به شکل زیر عمل می‌کنیم:

```
./bin/Client.out config/config.json
```

آرگومان کلاینت آدرس فایل کانفیگ است.

Makefile به گونه‌ای نوشته شده است که سرور و کلاینت را به کتابخانه تبدیل کند (که در پوشه bin/lib هستند) و این کتابخانه‌ها را به صورت ایستا به پروژه لینک کند.

سرور و کلاس‌های مربوط

حال به بررسی کدهای برنامه میپردازیم که در پوشه src هستند.

فایل include/logger-inl.h:

این فایل مربوط به کلاس logger است که به شکل زیر است:

```cpp
class Logger
{
public:
    Logger()
    {
        file.open("log.txt", std::fstream::out | std::fstream::app);
    }
    ~Logger()
    {
        file.close();
    }
    void log(const std::string& message)
    {
        if (message.size() != 0)
        {
            auto clock = std::chrono::system_clock::now();
            std::time_t time = std::chrono::system_clock::to_time_t(clock);
            file << message << " At " <<  std::ctime(&time) << std::endl;
        }
    }
    void operator()(std::string msg)
    {
        log(msg);
    }

private:
    std::fstream file;
};
```

Logger()	کانستراکتور این کلاس که یک فایل با نام log.txt باز میکند. توجه کنید تنظیمات به گونه‌ای قرار داده شده که در صورتی که فایل موجود بود از انتهای آن شروع به نوشتن کند.
~Logger()	دیستراکتور این کلاس که فایل باز شده را میبندد.
log()	آرگومان ورودی این تابع همراه با تاریخ و ساعت که با استفاده از کتابخانه chrono به دست می‌آید در فایل نوشته می‌شود.
operator()()	اپراتور پرانتز که برای راحتی کار overload شده است.

فایل include/user.h:

```
class User
{
public:
    std::string username;
    std::string password;
    bool is_admin;
    size_t data_cap;

    User(std::string _username, std::string _password, bool _is_admin, size_t _data_cap) :
        username(_username),
        password(_password),
        is_admin(_is_admin),
        data_cap(_data_cap)
    {
    }
};
```

در این کلاس اطلاعات مربوط به کاربر نگهداری میشود که شامل چهار فیلد نام کاربری، رمز عبور، ادمین بودن یا نبودن و حجم مجاز مصرفی است. همچنین یک کانستراکتور برای مقدار دادن به این فیلدها قرار داده شده است.

فایل include/responses.h:

```
#define ERROR RED("500: Error")
#define PWD_SUCCESS(arg) GREEN("257: " + arg)
#define QUIT_SUCCESS GREEN("221: Successful Quit.")
#define CWD_SUCCESS GREEN("250: Successful change.")
#define LS_SUCCESS GREEN("<202b><202a>226:<202c><202c> <202b><202a>List<202c><202c> <202b><202a>transfer<202c><202c> <202b><202a>done.<202c><202c>")
#define BAD_SQ RED("503: Bad sequence of commands.")
#define NEED_LOGIN RED("332: Need account for login.")
#define FILE_UNAVAILABLE RED("550: File unavailable.")
#define RETR_SUCCESS GREEN("<202b><202a>226:<202c><202c> <202b><202a>Successful<202c><202c> <202b><202a>Download.<202c><202c>")
#define RENAME_SUCCESS GREEN("250: Successful change.")
#define DELE_SUCCESS(arg) GREEN("<202b><202a>250:<202c><202c> " + arg + " deleted.")
#define MKDIR_SUCCESS(arg) GREEN("257: " + arg + " created.")
#define DATA_CAP_ERROR RED("425: Can't open data connection.")
#define USERNAME_OK GREEN("331: User name okay, need password")
#define INVALID_USERNAME RED("430: Invalid username or password")
#define INVALID_PASSWORD RED("430: Invalid username or password")
#define SYNTAX_ERROR RED("<202b><202a>501:<202c><202c> <202b><202a>Syntax<202c><202c> <202b><202a>error<202c><202c> <202b><202a>in<202c><202c> <202b><
2c> <202b><202a>or<202c><202c> <202b><202a>arguments.<202c><202c>")
#define PASSWORD_OK GREEN("230: User logged in, proceed. Logged out if appropriate.")

#define RED(arg) (string("\033[1;31m") + arg + "\033[0m")
#define GREEN(arg) (string("\033[1;32m") + arg + "\033[0m")
```

در این تابع به ازای هر پیامی که در صورت پروژه آمده است یک ماکرو Define شده است. هر کجا نیاز به این پیام‌ها بود از ماکرو مخصوص به آن استفاده میکنیم. همچنین برای اینکه پیام‌ها در ترمینال رنگی شود از دو ماکرو دیگر با نام‌های RED و GREEN استفاده شده است که پیام را رنگی میکنند.

فایل include/server.h:

این فایل header کلاس سرور است. متدهای عمومی آن به شرح زیر هستند:

```
public:
    Server(const std::string& config_path, const std::string& server_path);
    ~Server();
    void run();
```

Server()	کانستراکتور کلاس سرور که آرگومان‌های آن به ترتیب آدرس فایل کانفیگ و آدرس فایل‌های سرور هستند.
~Server()	دیستراکتور کلاس سرور
run()	وقتی میخواهیم سرور شروع به اجرا کند و connection ها را دریافت کند این متد را صدا میزنیم

متدهای خصوصی نیز به شرح زیر هستند:

```
void create_socket(int& sock);
void make_listen(int sock) const;
std::pair<int, int> accept_connection();
void read_config_file(const std::string& config_path);
void bind_socket(int sock, int port, struct sockaddr_in* address) const;

static void* manage_request(void* new_sockets);
static User* find_user(const std::string& username) noexcept;
static size_t send_buffer(int sock, const std::string& buffer);
static std::vector<std::string> parse_command(char command[]) noexcept;
static size_t send_buffer(int sock, const std::string& buffer, size_t n);
static bool verify_username(const std::string& username, std::string& response) noexcept;
static User* verify_password(const std::string& logged_in_user, const std::string& password, std::string& response) noexcept;
```

create_socket()	این تابع رفرنس یک int را دریافت می‌کند و با استفاده از آن یک سوکت tcp ایجاد میکند. در صورت عدم موفقیت یک استثنا پرتاب میشود.
make_listen()	این تابع یک سوکت دریافت میکند و بر روی آن گوش میکند. در صورت عدم موفقیت یک استثنا پرتاب میشود.
accept_connection()	این تابع connectionهایی که دریافت میشود را accept میکند. خروجی آن یک دوتایی از سوکت است که یکی از آن‌ها سوکت دستور و دیگری سوکت داده است. در صورت عدم موفقیت یک استثنا پرتاب میشود.

read_config_file()	این تابع وظیفه خواندن فایل کانفیگ را دارد و اطلاعات آن را ذخیره میکند. در صورت عدم موفقیت یک استثنا پرتاب میشود.
bind_socket()	این تابع سوکت را به یک آدرس و پورت bind میکند. در صورت عدم موفقیت یک استثنا پرتاب میشود.
manage_request()	هنگامی که یک کانکشن جدید می‌رسد، یک thread جدید ایجاد میشود. این تابع همان تابع thread است و ورودی آن سوکت‌های دستور و داده است.
find_user()	این تابع با دریافت یک نام کاربری، تمام اطلاعات مربوط را برمیگرداند. درصورتی که نام کاربری یافت نشد nullptr برمیگردد.
send_buffer()	این تابع با دریافت یک سوکت و یک رشته، رشته را به سوکت داده شده ارسال میکند. در صورت عدم موفقیت استثنا پرتاب میشود.
parse_command()	این تابع یک رشته دستور را پارس کرده و آرگومان‌های آن را در یک بردار میریزد و آن بردار را برمیگرداند.
send_buffer()	این تابع overload تابع قبلی است با این تفاوت که علاوه بر سوکت و رشته برای ارسال، اندازه‌ای که به آن مقدار ارسال صورت پذیرد هم داده میشود.
verify_username()	این تابع با دریافت یک نام کاربری چک میکند که آیا این نام کاربری در سیستم وجود دارد یا خیر. در صورتی که وجود دارد true و در غیر این صورت false را برمیگرداند. همچنین یکی از آرگومان‌های آن رفرنس رشته است که پاسخ متناظر با درخواست نام کاربری در آن ریخته میشود.
verify_password()	با نام کاربری و رمز عبور داده شده کاربر مربوطه یافت میشود. در صورتی که اطلاعات غلط بود nullptr به عنوان خروجی بازگردانده میشود و response هم با توجه به سناریوی پیش آمده مقدار می‌گیرد.

پیاده سازی این توابع در فایل server.cpp آمده است که در بخش بعد به توضیح آن میپردازیم.

سایر فیلدهای این کلاس به شرح زیر هستند:

```
static constexpr size_t DEFAULT_LISTEN_QUEUE_SIZE = 5;
static constexpr size_t DEFAULT_BUFFER_SIZE = 4096;

static char server_files_path[DEFAULT_BUFFER_SIZE];
static std::vector<std::string> private_files;
static std::vector<User*> users;
static Logger logger;

struct sockaddr_in command_address;
struct sockaddr_in data_address;

std::string config_path;
int command_socket_descriptor;
int data_socket_descriptor;
int command_channel_port;
int data_channel_port;

friend class ServiceThread;
```

DEFAULT_LISTEN_QUEUE_SIZE	اندازه صف که در تابع listen استفاده میشود
DEFAULT_BUFFER_SIZE	اندازه دیفالت بافرها که در طول کد ساخته میشود
server_files_path	آدرس فایل‌های سرور که یکی از آرگومان‌های برنامه بود
private_files	لیستی از فایل‌های پرایویت که در فایل کانفیگ داده شده بود
users	لیست کاربرانی که اطلاعات آنها در فایل کانفیگ داده شده بود
logger	یک نمونه از کلاس Logger که از آن برای ثبت لاگ سیستم استفاده میشود
command_address	آدرس کانال دستور
data_address	آدرس کانال داده
config_path	آدرس فایل کانفیگ که یکی از آرگومان‌های برنامه بود
command_socket_descriptor	سوکت مربوط به کانال دستور
data_socket_descriptor	سوکت مربوط به کانال داده
command_channel_port	پورت مربوط به کانال دستور
data_channel_port	پورت مربوط به کانال داده

فایل server.cpp:

تابع read_config_file:

```
void Server::read_config_file(const string& config_path)
{
    Json::CharReaderBuilder reader;
    Json::Value config_root;
    ifstream config_file(config_path);
    config_file >> config_root;

    command_channel_port = config_root["commandChannelPort"].asInt();
    data_channel_port = config_root["dataChannelPort"].asInt();

    for (Json::ValueIterator user = config_root["users"].begin(); user != config_root["users"].end(); user++)
        users.push_back(new User((*user)["user"].asString(), (*user)["password"].asString(),
                (*user)["admin"].asString() == "true", stoi((*user)["size"].asCString())));

    for (Json::ValueIterator user = config_root["files"].begin(); user != config_root["files"].end(); user++)
        private_files.push_back((*user).asString());
}
```

در این تابع قصد داریم تا اطلاعات موجود در فایل کانفیگ را استخراج کنیم. برای این کار از کتابخانه jsoncpp استفاده شده که کدهای آن در مسیر src/lib/json قرار دارد و از طریق makefile کتابخانه مربوط به آن ساخته می‌شود. این تابع مسیر فایل کانفیگ را به عنوان آرگومان دریافت می‌کند و سپس با توابع خود این فایل را میخواند و در یک مپ ذخیره میکند. با استفاده از این مپ مقادیر command_channel_port و data_channel_port و وکتورهای users و private_files پر شده است و این مقادیر از فایل کانفیگ استخراج می‌شود.

تابع create_socket:

```
void Server::create_socket(int& sock)
{
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        throw system_error(errno, system_category(), "Creating socket failed");

    int on = 1;
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &on, sizeof(on)))
        throw system_error(errno, system_category(), "Set socket option failed");
}
```

در این تابع ابتدا یک سوکت tcp ساخته میشود. در صورت عدم موفقیت در ساخت این سوکت یک استثنا پرتاب میشود. بعد از ساخت سوکت آپشنی بر روی سوکت تنظیم میشود برای اینکه بتوان از آدرس و پورت مربوطه دوباره استفاده کرد. این مورد برای وقتی است که بر روی یک سیستم چندین سرور داشته باشیم. اگر این عملیات ناموفقیت بود یک استثنا پرتاب میشود. توجه کنید سوکتی که ساخته شده به آرگومان تابع نسبت داده شده است.

تابع bind_socket:

```
void Server::bind_socket(int sock, int port, struct sockaddr_in* address) const
{
    address->sin_family = AF_INET;
    address->sin_addr.s_addr = INADDR_ANY;
    address->sin_port = htons(port);

    if (bind(sock, reinterpret_cast<struct sockaddr*>(address), sizeof(*address)) < 0)
        throw system_error(errno, system_category(), "Binding socket failed");
}
```

در این تابع ابتدا ساختار مربوط به آدرس داده شده در آرگومان پر میشود و بعد از آن سوکت داده شده در آرگومان به آن آدرس bind میشود. توجه کنید پورت مربوط به آدرس در آرگومان داده شده است و عملیات مقداردهی در این تابع انجام میشود. در صورتی که bind موفقیت آمیز نبود یک استثنا پرتاب می‌شود.

تابع make_listen:

```
void Server::make_listen(int sock) const
{
    if (listen(sock, DEFAULT_LISTEN_QUEUE_SIZE) < 0)
        throw system_error(errno, system_category(), "Listening socket failed");
}
```

در این تابع، تابع listen بر روی سوکتی که در آرگومان داده شده صدا زده می‌شود و در صورت عدم موفقیت یک استثنا پرتاب می‌شود.

تابع parse_command:

```
vector<string> Server::parse_command(char command[]) noexcept
{
    int index = 0;
    string word;
    istringstream ss(command);
    vector<string> result;
    while (ss >> word)
        result.push_back(word);

    return result;
}
```

در این تابع ابتدا یک stringstream از ورودی ساخته میشود و با خواندن از آن و اضافه کردن رشته خوانده شده به یک وکتور نتیجه محاسبه میشود. توجه کنید کار این تابع این است که دستور را از حالت رشته به یک وکتور از آرگومان‌ها تبدیل کند و این وکتور را برگرداند.

تابع find_user():

```
User* Server::find_user(const string& username) noexcept
{
    for (User* user : users)
        if (user->username == username)
            return user;
    return nullptr;
}
```

این تابع یک نام کاربری در ورودی می‌گیرد و در وکتور مربوط به یوزرها به دنبال این نام کاربری میگردد. در صورتی که کلاس مربوط به آن پیدا شد آن کلاس را برمیگرداند و در غیر این صورت مقدار نال بازگردانده میشود.

تابع verify_username():

```
bool Server::verify_username(const string& username, string& response) noexcept
{
    if (find_user(username) != nullptr)
    {
        response = USERNAME_OK;
        return true;
    }

    response = INVALID_USERNAME;
    return false;
}
```

این تابع یک نام کاربری و یک رفرنس استرینگ دریافت میکند. این تابع در ابتدا به دنبال نام کاربری میگردد و در صورتی که نام کاربری یافت شد مقدار response را برابر USERNAME_OK (یکی از همان ماکروهایی که قبلا اشاره شد) قرار میدهد و true بازمیگرداند. در صورتی که نام کاربری یافت نشد مقدار response برابر INVALID_USERNAME قرار داده میشود و مقدار false بازگردانده میشود.

تابع verify_password():

```
User* Server::verify_password(const string& username, const string& password, string& response) noexcept
{
    if (username == "")
    {
        response = BAD_SQ;
        return nullptr;
    }

    User* user;
    if ((user = find_user(username)) != nullptr && user->password == password)
    {
        response = PASSWORD_OK;
        return user;
    }
    response = INVALID_PASSWORD;
    return nullptr;
}
```

این تابع یک نام کاربری و رمز عبور به همراه رفرنس یک رشته که باید نتیجه در آن ریخته شود دریافت میکند. در صورتی که نام کاربری مقدار خالی باشد بدان معناست که کاربر هنوز نام کاربری را وارد نکرده پس مقدار response برابر BAD_SEQUENCE قرار داده میشود و مقدار نال را برمیگرداند. بعد از آن به جستجوی نام کاربری پرداخته میشود و در صورتی که کاربر مربوط به آن رمز عبور درستی وارد کرده بود response برابر PASSWORD_OK قرار داده میشود و کاربر مربوط به عنوان مقدار بازگشتی بازگردانده میشود. در صورتی که نام کاربری و رمز عبور مطابقت نداشتند INVALID_PASSWORD در response ریخته میشود و مقدار نال بازگردانده می‌شود.

توابع send_buffer():

```
size_t Server::send_buffer(int sock, const string& buffer, size_t n)
{
    int size;
    if ((size = send(sock, buffer.c_str(), n, 0)) < 0)
        throw system_error(errno, system_category(), "Send response failed");
    return size;
}

size_t Server::send_buffer(int sock, const string& buffer)
{
    int size;
    if ((size = send(sock, buffer.c_str(), DEFAULT_BUFFER_SIZE, 0)) < 0)
        throw system_error(errno, system_category(), "Send response failed");
    return size;
}
```

در این توابع یک سوکت و یک رشته گرفته میشود و رشته مربوط به سوکت داده شده ارسال میشود. در صورت عدم موفقیت یک استثنا پرتاب میشود و در صورت موفقیت تعداد بایت‌هایی که به درستی ارسال شده است بازگردانده میشود. تفاوت این دو تابع در آرگومان سوم است که در یکی وجود دارد و

بیان میکند چه تعداد بایت در تابع send ارسال شود. در تابع دیگر به مقدار پیش فرض، که در declaration کلاس تعریف شده، ارسال انجام می‌پذیرد.

تابع accept_connection():

```
pair<int, int> Server::accept_connection()
{
    int new_command_socket, new_data_socket;
    socklen_t command_address_length = sizeof(command_address);
    socklen_t data_address_length = sizeof(data_address);
    if ((new_command_socket = accept(command_socket_descriptor, reinterpret_cast<struct sockaddr*>(&command_address),
            &command_address_length)) < 0)
        throw system_error(errno, system_category(), "Accept failed");
    if ((new_data_socket = accept(data_socket_descriptor, reinterpret_cast<struct sockaddr*>(&data_address),
            &data_address_length)) < 0)
        throw system_error(errno, system_category(), "Accept failed");
    logger("Accepting new client successful");
    return pair<int, int>(new_command_socket, new_data_socket);
}
```

در این تابع دوبار تابع accept صدا زده میشود. در یکی از آنها سوکت دستور accept میشود و در دیگری سوکت داده accept میشود. در صورت عدم موفقیت در هر کدام از آنها یک استثنا پرتاب میشود و هنگامی که هر دوی آنها با موفقیت accept شدند logger یک پیغام مبنی بر اینکه یک کاربر جدید با موفقیت وارد شد ثبت میکند و در نهایت سوکت های accept شده بازگردانده میشود.

کانستراکتور Server():

```
Server::Server(const string& config_path, const string& server_path)
{
    read_config_file(config_path);
    logger("Reading from config file successful");
    create_socket(command_socket_descriptor);
    create_socket(data_socket_descriptor);
    logger("Creating sockets successful");
    bind_socket(command_socket_descriptor, command_channel_port, &command_address);
    bind_socket(data_socket_descriptor, data_channel_port, &data_address);
    logger("Binding sockets successful");
    make_listen(command_socket_descriptor);
    make_listen(data_socket_descriptor);
    logger("Listening sockets successful");
    chdir(server_path.c_str());     // Change program folder.
    getcwd(server_files_path, DEFAULT_BUFFER_SIZE);
    logger("Server is up");
}
```

در کانستراکتور این تابع ابتدا اطلاعات از فایل کانفیگ خوانده می‌شود (آدرس فایل کانفیگ قبل‌تر توسط آرگومان‌های خط فرمان به برنامه داده شده است) و پس از آن سوکت‌های داده و دستور ایجاد میشود. بعد از آن به آدرس و پورت مربوط به خودشان bind میشود. سپس هر دوی سوکت‌ها شروع به listen میکنند. در نهایت محل اجرای پردازه سرور به آدرس داده شده در آرگومان (که قبل‌تر توسط آرگومان‌های خط فرمان به تابع داده شده است) تغییر میکند و این محل در فیلد این کلاس به نام

server_files_path (که قبل تر توضیح داده شد) ذخیره میشود. یعنی با استفاده از تابع chdir محل اجرای پردازه عوض میشود و با تابع getcwd این محل دریافت شده و در فیلد داده شده ذخیره میشود. در میان تمام این مراحل به تناسب کاری که انجام شده لاگ‌های مناسب ذخیره می‌شود.

دیستراکتور ~Server():

```
Server::~Server()
{
    close(command_socket_descriptor);
    close(data_socket_descriptor);
    for (vector<User*>::iterator user = users.begin(); user != users.end(); user++)
        delete *user;
    logger("Server shuts down");
}
```

در ابتدا سوکت‌های داده و دستور بسته میشوند. سپس حافظه‌هایی که جهت ذخیره سازی اطلاعات کاربران (در تابع read_config_file) تخصیص پیدا کرده است آزاد میشود و در انتها لاگ مربوط ثبت می‌شود.

تابع run():

```
void Server::run()
{
    pthread_t thread;
    while (true)
    {
        pair<int, int> sockets = this->accept_connection();
        pthread_create(&thread, NULL, Server::manage_request, static_cast<void*>(&sockets.first));
    }
}
```

در این تابع یک حلقه بی‌نهایت وجود دارد که در آن ابتدا یک کانکشن اکسپت می‌شود و بعد از آن یک thread برای رسیدگی به این کانکشن اختصاص می‌یابد. تابعی که مخصوص رسیدگی به thread است manage_request نام دارد که در ادامه به توضیح آن می‌پردازیم. به thread ساخته شده سوکت اکسپت شده را نیز پاس می‌دهیم.

تابع manage_request():

در ابتدا متغیرهای مورد نیاز تعریف می‌شود.

```
pair<int,int> sockets = *(static_cast<pair<int, int>*>(new_sockets));
bool finish = false;
User* logged_in_user;
string username = "", log;
ServiceThread service_thread(sockets);
```

در ابتدا سوکت‌هایی که به عنوان ورودی داده شده است به نوع صحیح cast میشود. بعد از آن متغیرهای مورد نیاز تعریف شده و در نهایت یک نمونه از کلاس ServiceThread ایجاد می‌شود که در آینده به توضیح آن خواهیم پرداخت.

بعد از تعریف متغیرها یک حلقه وجود دارد که تا وقتی پرچم finish برابر false است کارهای زیر انجام می‌شود:

```
int size;
char buffer[DEFAULT_BUFFER_SIZE];
memset(buffer, 0, DEFAULT_BUFFER_SIZE);
if ((size = recv(sockets.first, buffer, DEFAULT_BUFFER_SIZE, 0)) < 0)
    throw system_error(errno, system_category(), "Receive command failed");
else if (size == 0)
    break;
```

ابتدا از سوکت اول که سوکت دستور است دستوری که کاربر وارد میکند دریافت می‌شود. در صورتی که تعداد بایت‌های دریافت شده برابر صفر باشد از حلقه خارج میشویم.

```
vector<string> command = parse_command(buffer);
string response = "";
```

بعد از دریافت دستور نوبت به پارس کردن آن میرسد و یک رشته تعریف می‌شود که نتیجه رسیدگی به این دستور است.

حال با توجه به نوع دستوری که وارد شده تصمیمات مختلفی اتخاد می‌شود که به شرح زیر است.

```
if (command[0] == "user")
{
    if (verify_username(command[1], response))
        username = command[1];
}
```

اگر نوع دستور user باشد تابع verify_username صدا زده می‌شود و در صورت موفقیت آمیز بودن نام کاربری وارد شده در متغیر username ریخته می‌شود.

```
else if (command[0] == "pass")
{
    logged_in_user = verify_password(username, command[1], response);
    if (logged_in_user != nullptr)
    {
        service_thread.set_logged_in_user(logged_in_user);
        pthread_mutex_lock(&lock);
        logger(username + " has logged in successfully");
        pthread_mutex_unlock(&lock);
    }
}
```

در صورتی که نوع دستور pass باشد نوبت به چک کردن صحت رمز عبور با تابع verify_password میرسد. در صورتی که ورود موفقیت آمیز بود (یعنی مقدار خروجی مخالف نال بود) در نمونه service_thread کاربر وارد شده ست میشود و بعد از آن در فایل لاگ پیامی مبنی بر اینکه ورود کاربر موفقیت آمیز بود ثبت میشود. توجه کنید چون ممکن است چندین کاربر (و بالتبع چندین thread) در حال نوشتن در فایل باشد یک قفل ایجاد شده که اینجا استفاده می‌شود.

```
else
{
    log = "";
    response = service_thread.manage_command(command, finish, log);
    pthread_mutex_lock(&lock);
    logger(log);
    pthread_mutex_unlock(&lock);
}
```

در صورتی که دستور وارد شده هیچ کدام از این دو دستور نبود از نمونه service_thread استفاده می‌کنیم. ابتدا متغیر لاگ ریست میشود. بعد از آن تابع manage_command از این نمونه صدا زده میشود. پس از آن لاگی که توسط این تابع تولید شده است توسط logger ثبت میشود. توجه کنید به دلایلی که ذکر شد از قفل استفاده شده است. تابع manage_command به عنوان ورودی دستور وارد شده، پرچم پایان و رشته لاگ را دریافت میکند و متغیرهای finish و log را مقداردهی میکند. همچنین به عنوان خروجی response متناظر با این دستور بازگردانده می‌شود.

```
send_buffer(sockets.first, response, response.size());
```

در نهایت response تولید شده از طریق سوکت دستور به کلاینت بازگردانده میشود.

```
close(sockets.first);
close(sockets.second);
return nullptr;
```

هنگامی که از حلقه خارج می‌شویم سوکت‌ها را میبندیم و از thread خارج می‌شویم. توجه کنید این سوکت‌ها، سوکت‌هایی هستند که accept شده‌اند و نه سوکت‌های اصلی.

فایل include/service_thread.h:

این کلاس برای رسیدگی به یک thread به کار می‌رود. متدهای عمومی این تابع به شرح زیر است:

```
public:
    ServiceThread(std::pair<int, int> sockets);
    void set_logged_in_user(User* user) noexcept;
    std::string manage_command(const std::vector<std::string>& command, bool& finish, std::string& log);
```

ServiceThread()	کانستراکتور این کلاس که یک جفت سوکت به عنوان ورودی می‌گیرد. یکی از این سوکت‌ها مربوط به دستور و دیگری مربوط به داده است.
set_logged_in_user()	یک setter برای ست کردن logged_in_user که از فیلدهای این کلاس است. در ادامه این فیلد را خواهیم دید.
manage_command()	برای مدیریت یک دستور که از سمت کلاینت ارسال شده است. ورودی این تابع دستور ارسال شده، یک پرچم برای پر کردن (این پرچم را قبلا دیدیم. اگر کار کلاینت تمام شود true می‌شود) و یک رشته برای پر کردن که لاگ سرور است. خروجی این تابع یک رشته است که پاسخی است که در جواب دستور کلاینت. همانطور که دیدیم این پاسخ برای کلاینت ارسال می‌شود.

متدهای خصوصی نیز به شرح زیر هستند:

```
private:
    int get_file_size(const std::string& filename) const noexcept;
    bool check_user_permission(const std::string& file_name) const noexcept;
    std::string ls() const noexcept;
    std::string pwd() const noexcept;
    std::string cwd(const std::string& path) noexcept;
    std::string get_path(const std::string& name) const noexcept;
    std::string retr(const std::vector<std::string>& command) const noexcept;
    std::string dele(const std::vector<std::string>& command) const noexcept;

    static int del(const char* path, const struct stat* sb, int typeflag, struct FTW* ftwbuf);
```

get_file_size()	با دریافت نام فایل اندازه آن را در خروجی می‌دهد.
check_user_permission()	نام یک فایل را دریافت میکند. اگر کاربر وارد شده اجازه دسترسی به فایل را داشت false و در غیر این صورت true برمیگرداند.
ls()	وقتی دستور ls ارسال شده باشد این تابع صدا می‌شود.
pwd()	وقتی دستور pwd ارسال شده باشد این تابع صدا می‌شود.

cwd()	وقتی دستور cwd ارسال شده باشد این تابع صدا می‌شود.
get_path()	با دریافت نام یک فایل، آدرسی که آن فایل در آن قرار دارد بازگردانده می‌شود. توجه کنید این آدرس نسبت به فایل‌های سرور داده می‌شود.
retr()	وقتی دستور retr ارسال شده باشد این تابع صدا می‌شود.
dele()	وقتی دستور dele ارسال شده باشد این تابع صدا می‌شود.
del()	این تابع استاتیک هنگام پاک کردن دایرکتوری به کار میرود. در ادامه کاربرد این تابع را خواهیم دید.

توجه کنید خروجی توابعی که متناظر با دستورات است response نسبت به آن دستور است.

فیلدهای این کلاس نیز به شرح زیر است:

```
User* logged_in_user;
std::vector<std::string> working_directory;
int command_socket;
int data_socket;
```

logged_in_user	کاربری که وارد شده است در این فیلد نگهداری می‌شود
working_directory	محلی که در حال حاضر thread در آن مشغول کار است در این فیلد ذخیره می‌شود. کارکرد دقیق‌تر آن در ادامه توضیح داده خواهد شد
command_socket	سوکت مربوط به کانال دستور
data_socket	سوکت مربوط به کانال داده

پیاده سازی این توابع در فایل service_thread.cpp آمده است که در ادامه به توضیح آن می‌پردازیم.

کاستراکتور ServiceThread():

```
ServiceThread::ServiceThread(pair<int, int> sockets)
{
    logged_in_user = nullptr;
    command_socket = sockets.first;
    data_socket = sockets.second;
}
```

در کانستراکتور این کلاس فیلدهای logged_in_user و command_socket و data_socket مقداردهی می‌شود.

تابع set_logged_in_user():

```
void ServiceThread::set_logged_in_user(User* user) noexcept
{
    logged_in_user = user;
}
```

فیلد logged_in_user ست می‌شود.

تابع check_user_permission():

```
bool ServiceThread::check_user_permission(const string& file_name) const noexcept
{
    if (find(Server::private_files.begin(), Server::private_files.end(), file_name) != Server::private_files.end() && !logged_in_user->is_admin)
        return false;
    return true;
}
```

در این تابع ابتدا چک میشود فایل داده شده از فایل‌های خصوصی هست یا خیر. اگر فایل خصوصی بود ادمین بودن کاربر وارد شده چک میشود. اگر این کاربر اجازه دسترسی به فایل را داشت false و در غیر این صورت true بازگردانده می‌شود.

تابع cwd():

```
string ServiceThread::cwd(const string& path) noexcept
{
    if (path == "")
    {
        working_directory.erase(working_directory.begin(), working_directory.end());
        return CWD_SUCCESS;
    }
    if (path[0] == '/')
    {
        cwd("");
        return cwd(path.substr(1, path.size()));
    }
    if (path == "..")
    {
        if (working_directory.size() != 0)
            working_directory.pop_back();
        return CWD_SUCCESS;
    }
    string dir;
    istringstream ss(path);
    struct stat sb;
    if (stat(get_path(path).c_str(), &sb) == 0 && S_ISDIR(sb.st_mode))
    {
        while (getline(ss, dir, '/'))
            working_directory.push_back(dir);
        return CWD_SUCCESS;
    }
    return ERROR;
}
```

فیلد working_directory مشخص میکند ما در چه مسیری هستیم. اگر وارد یک پوشه شویم نام آن پوشه به این وکتور اضافه می‌شود و اگر یک پوشه به عقب برگردیم عضو آخر این وکتور pop میشود. با این منطق به توضیح این تابع می‌پردازیم. اگر مسیر وارد شده خالی باشد یعنی باید به اولین پوشه بازگردیم پس working_directory را خالی میکنیم و موفقیت را بازمیگردانیم. اگر در ابتدای مسیری که وارد شده / وجود داشت یعنی آدرس دهی از پوشه اولیه انجام شده است. پس ابتدا با صدا کردن cwd(“”) به پوشه اولیه بازمیگردیم و بعد از آن ادامه مسیر گفته شده را طی می‌کنیم. اگر مسیر وارد شده .. باشد یعنی باید به پوشه قبلی بازگردیم. یعنی از working_directory عضو آخر را pop میکنیم. (اگر خالی نباشد). اگر این وکتور خالی باشد یعنی در پوشه اولیه هستیم. در این صورت بازگشت به پوشه عقب ارور نمیدهد و صرفا اتفاقی نمی‌افتد. این همان رفتاری است که سیستم عامل ubuntu 20.04 دارد. اگر هیچ کدام از حالات ذکر شده نبود، ابتدا چک میشود که مسیر وارد شده یک دایرکتوری است. پس از آن در یک حلقه فولدرهای آن به ترتیب جدا میشود و به working_directory اضافه می‌شود. در نهایت اگر فولدر وجود داشت پیام موفقیت و در غیر این صورت پیام شکست به عنوان response بازگردانده می‌شود.

تابع get_path():

```
string ServiceThread::get_path(const string& name) const noexcept
{
    string result = "";
    for (auto wd : working_directory)
        result += wd + "/";
    return result + name;
}
```

این تابع نام فایل را دریافت میکند و مسیر کامل این فایل نسبت به پوشه اولیه بازمیگرداند.

تابع get_file_size():

```
int ServiceThread::get_file_size(const string& filename) const noexcept
{
    struct stat stat_buf;
    int rc = stat(get_path(filename).c_str(), &stat_buf);
    return rc == 0 ? stat_buf.st_size : -1;
}
```

این تابع با دریافت نام فایل اندازه آن را بازمیگرداند. در صورتی که همچین فایلی وجود نداشت منفی یک بازگردانده می‌شود. توجه کنید سایز به بایت است.

تابع dele():

```
string ServiceThread::dele(const vector<string>& command) const noexcept
{
    if (!check_user_permission(command[2]))
        return FILE_UNAVAILABLE;
    int result = -1;
    if (command[1] == "-d")
        result = nftw(get_path(command[2]).c_str(), ServiceThread::del, 64, FTW_DEPTH | FTW_PHYS);
    else if (command[1] == "-f")
        result = unlink(get_path(command[2]).c_str());
    else
        return SYNTAX_ERROR;
    if (result < 0)
        return ERROR;
    return DELE_SUCCESS(command[2]);
}
```

در این تابع ابتدا چک میشود کاربر اجازه دسترسی به فایل خواسته شده را دارد یا خیر. در صورت نداشتن دسترسی FILE_UNAVALABLE به عنوان response بازگردانده میشود. بعد از آن به آرگومان دوم توجه میشود. اگر این آرگومان d- بود یعنی میخواهیم یک دایرکتوری را حذف کنیم. از تابع nftw برای حرکت روی فایل‌ها استفاده میکنیم و تابع آن را del میدهیم که در ادامه این تابع توضیح داده خواهد شد. عمق حرکت را نیز ۶۴ قرار میدهیم. اگر آرگومان دوم f- بود از تابع unlink استفاده میکنیم که با دریافت آدرس فایل آن را پاک میکند. اگر هیچ کدام از این دو مورد نبود SYNTAX_ERROR را به عنوان response بازمیگردانیم و اگر هر کدام از توابع ذکر شده دچار ارور شدند ارور میدهیم. در غیر این صورت پیام موفقیت را به عنوان response باز میگردانیم.

تابع del():

```
int ServiceThread::del(const char* path, const struct stat* sb, int typeflag, struct FTW* ftwbuf)
{
    return S_ISREG(sb->st_mode) ? -1 : remove(path);
}
```

در این تابع چک میکنیم که مسیر داده شده مربوط به فولدر باشد. اگر مربوط به فولدر بود تابع remove را روی آن صدا میزنیم (این تابع مسیر داده شده را پاک میکند) و نتیجه آن را بازمیگردانیم و اگر این مسیر فولدر نبود منفی یک بازمیگردانیم.

تابع pwd():

```
string ServiceThread::pwd() const noexcept
{
    return PWD_SUCCESS("/" + get_path(""));
}
```

در اینجا مسیری که در آن هستیم را بازمیگرداند. توجه کنید امکان خطا وجود ندارد.

تابع ls():

```
string ServiceThread::ls() const noexcept
{
    dirent* dir_path;
    DIR* path = opendir(get_path(".").c_str());
    string result;
    while ((dir_path = readdir(path)) != nullptr)
    {
        string name = string(dir_path->d_name);
        if (name != "." && name != "..")
            result += name + '\n';
    }
    Server::send_buffer(data_socket, result.c_str(), result.size() == 0 ? Server::DEFAULT_BUFFER_SIZE : result.size());
    return LS_SUCCESS;
}
```

در این تابع با استفاده از تابع opendir دایرکتوری فعلی باز میشود و روی فایل‌ها و فولدرهای آن پیمایش می‌شود. در صورتی که آنها . یا .. نبودند به رشته نتیجه اضافه میشود. در نهایت نتیجه به دست آمده از طریق کانال داده برای کلاینت ارسال میشود و پیام موفقیت به عنوان response بازگردانده میشود.

تابع retr():

```
string ServiceThread::retr(const vector<string>& command) const noexcept
{
    uint8_t block[Server::DEFAULT_BUFFER_SIZE];
    int file_size = get_file_size(command[1]);
    if (file_size < 0)
    {
        Server::send_buffer(data_socket, to_string(file_size));
        return ERROR;
    }
    if (!check_user_permission(get_path(command[1])))
    {
        Server::send_buffer(data_socket, to_string(-1));
        return FILE_UNAVAILABLE;
    }
    if (logged_in_user->data_cap < file_size)
    {
        Server::send_buffer(data_socket, to_string(-1));
        return DATA_CAP_ERROR;
    }
    logged_in_user->data_cap -= file_size;
    Server::send_buffer(data_socket, to_string(file_size));
    FILE* file = fopen(get_path(command[1]).c_str(), "r");
    while (!feof(file))
    {
        int size = fread(block, sizeof(uint8_t), Server::DEFAULT_BUFFER_SIZE, file);
        send(data_socket, block, size, 0);
    }
    fclose(file);
    return RETR_SUCCESS;
}
```

قرارداد بین کلاینت و سرور به این صورت است که اول اندازه فایل ارسال میشود و پس از آن فایل باز شده و قطعه قطعه خوانده میشود و هر قطعه برای کلاینت ارسال میشود. پس در ابتدا اندازه فایل را به دست می‌آوریم. اگر این مقدار کمتر از صفر بود اول اندازه آن برای کلاینت ارسال میشود و بعد از آن ارور بازگردانده میشود. بعد از آن اجازه دسترسی بررسی میشود. اگر کاربر اجازه دسترسی به فایل را نداشت منفی یک برای آن ارسال میشود و ارور بازگردانده میشود. توجه کنید نیازمند این ارسال منفی یک هستیم. زیرا در سمت کلاینت تابع recv صدا شده و کلاینت منتظر دریافت است. اگر چیزی برای آن ارسال نشود بلاک میشود. بعد از اجازه دسترسی چک میکنیم آیا کاربر حجم مورد نیاز برای دانلود فایل را دارد یا خیر. در صورتی که حجمش کم بود DATA_CAP_ERROR به عنوان response بازگردانده میشود. بعد از آن حجم فایل از حجم کاربر کم میشود. فایل مورد نظر باز میشود و بلاک بلاک از آن خوانده میشود و در هر مرحله از خواندن بلاک خوانده شده ارسال میشود. در نهایت موفقیت به عنوان response بازگردانده میشود.

تابع manage_command():

این تابع اصلی این کلاس است.

```
if (logged_in_user == nullptr)
{
    if (command[0] == "ls" || command[0] == "retr")
        Server::send_buffer(data_socket, "");
    return NEED_LOGIN;
}
```

در ابتدای آن چک میشود آیا کاربری لاگین کرده است یا خیر. اگر کاربر لاگین نکرده بود خطای مربوطه بازگردانده میشود. توجه کنید یک رشته خالی در صورتی که دستور ls یا retr باشد از طریق کانال داده ارسال میشود تا کلاینت بلاک نکند.

```
if (command[0] == "pwd")
{
    if (command.size() != 1)
        return SYNTAX_ERROR;
    return pwd();
}
```

اگر دستور pwd باشد ابتدا تعداد آرگومان‌ها چک میشود و در صورت نبود مشکل تابع pwd صدا زده میشود.

```
if (command[0] == "mkd")
{
    if (command.size() != 2)
        return SYNTAX_ERROR;
    if (mkdir(get_path(command[1]).c_str(), S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH) < 0)
        return ERROR;
    log = logged_in_user->username + " created " + command[1];
    return MKDIR_SUCCESS(command[1]);
}
```

اگر دستور mkd باشد ابتدا تعداد آرگومان‌ها چک میشود و در صورت نبود مشکل تابع mkdir با دسترسی‌هایی که مشاهده میکنید صدا زده میشود. در صورت موفق بودن ساخت پوشه لاگ مناسب قرار داده میشود و پیامی مبنی بر موفقیت به عنوان response بازگردانده میشود.

```
if (command[0] == "dele")
{
    if (command.size() != 3)
        return SYNTAX_ERROR;
    string result = dele(command);
    if (result != ERROR && result != SYNTAX_ERROR)
        log = logged_in_user->username + " removed " + command[2];
    return result;
}
```

اگر دستور وارد شده dele باشد ابتدا تعداد آرگومان‌ها چک میشود در صورت عدم وجود مشکل تابع dele صدا زده میشود در صورتی که خروجی این تابع هم مشکلی نداشت لاگ مناسب مقداردهی میشود.

```
if (command[0] == "cwd")
{
    if (command.size() > 2)
        return SYNTAX_ERROR;
    if (command.size() == 1)
        return cwd("");
    return cwd(command[1]);
}
```

اگر دستور وارد شده cwd باشد ابتدا تعداد آگومان‌ها چک میشود. اگر این دستور بدون آرگومان وارد شد تابع cwd با رشته خالی صدا زده میشود. در غیر این صورت تابع cwd با آرگومان داده شده صدا زده میشود.

```
if (command[0] == "rename")
{
    if (command.size() != 3)
        return SYNTAX_ERROR;
    if (!check_user_permission(command[1]))
        return FILE_UNAVAILABLE;
    if (rename(get_path(command[1]).c_str(), get_path(command[2]).c_str()) < 0)
        return ERROR;
    log = logged_in_user->username + " renamed " + command[1] + " to " + command[2];
    return RENAME_SUCCESS;
}
```

اگر دستور rename وارد شده باشد ابتدا تعداد آرگومان‌ها چک میشود. بعد از آن دسترسی کاربر وارد شده چک میشود. اگر مشکلی وجود نداشت تابع rename صدا زده میشود و در صورت موفقیت این تابع لاگ مناسب تولید میشود. در انتها پیامی مبنی بر موفقیت بازگردانده می‌شود. اورورهای مربوط به هر بخش نیز با توجه به کد مشخص است.

```
if (command[0] == "quit")
{
    if (command.size() != 1)
        return SYNTAX_ERROR;
    finish = true;
    log = logged_in_user->username + " left";
    return QUIT_SUCCESS;
}
```

اگر دستور quit وارد شده باشد ابتدا تعداد آرگومان‌ها چک میشود. پس از آن پرچم مربوط به تمام شدن thread برابر true می‌شود و پس از آن لاگ مناسب تولید میشود.

```
if (command[0] == "ls")
{
    if (command.size() != 1)
    {
        Server::send_buffer(data_socket, "");
        return SYNTAX_ERROR;
    }
    return ls();
}
```

اگر دستور ls وارد شده باشد ابتدا تعداد آرگومان‌ها چک میشود. در صورت عدم موفقیت یک پیام خالی از طریق کانال داده ارسال میشود تا کلاینت بلاک نشود. در صورت موفقیت تابع ls صدا زده میشود و خروجی تولید میشود.

```
if (command[0] == "retr")
{
    if (command.size() != 2)
    {
        Server::send_buffer(data_socket, to_string(-1));
        return SYNTAX_ERROR;
    }
    string result = retr(command);
    if (result == RETR_SUCCESS)
        log = logged_in_user->username + " successfully downloaded " + command[1];
    return result;
}
```

اگر دستور retr وارد شده باشد ابتدا تعداد آرگومان‌ها چک میشود. در صورت عدم موفقیت یک پیام خالی از طریق کانال داده ارسال میشود تا کلاینت بلاک نشود. در صورت موفقیت تابع retr صدا زده میشود و اگر این دستور موفقیت آمیز بود لاگ مناسب تولید میشود.

```
    if (command[0] == "help")
    {
        if (command.size() != 1)
            return SYNTAX_ERROR;
        return string("214\n")
                + string("USER [name], Its argument is used to specify the user's string. It is used for user authentication.\n")
                + string("PASS [password], Its argument is used to match the user specified in the USER command.\n")
                + string("PWD, It is used to get the current directory. The response is the working directory path.\n")
                + string("MKD [name], Its argument is used to specify in which directory path, the new directory is to be created.\n")
                + string("DELE, Its argument is used to determine the name of the target filename or directory path that is to be rermoved.\n")
                + string("\tUse option -f [filename] to delete a specific file.\n")
                + string("\tUse option -d [directory path] to delete a specific directory.\n")
                + string("LS, displays all the file names in the current working directory.\n")
                + string("CWD [path], Changes the working directory and its argument is used to specify the target path to which the user wishes to
switch.\n")
                + string("RENAME [from] [to], Its first argument is used to state the name of the file that is to be changed and its second argument
 is used as the new file name.\n")
                + string("RETR [name], Its argument is used to retrieve the named file from server and the file in question is downloaded upon compl
eting the transfer.\n")
                + string("HELP, Enlists all the available commands along with a concise desciption.\n")
                + string("QUIT, logs out the current logged in user from the system.");
    }
```

در صورتی که دستور وارد شده help باشد ابتدا تعداد آرگومان‌ها چک میشود. بعد از آن پیامی تولید میشود که شرح کار سرور است و این پیام بازگردانده میشود.

فایل server_main.cpp

```cpp
int main(int argc, char** argv)
{
    if (argc < 3)
    {
        cerr << "Too few args" << endl;
        return 1;
    }
    Server* server = new Server(argv[1], argv[2]);
    server->run();
    return 0;
}
```

در این فایل main سرور قرار دارد. در ابتدا چک میشود تعداد آرگومان‌های خط فرمان صحیح باشد و بعد از آن یک نمونه از سرور تولید میشود. بعد از آن هم متد run صدا زده میشود.

کلاینت

فایل include/client.h:

این فایل شامل header کلاس کلاینت است. متدهای عمومی آن به شرح زیر است:

```
public:
    Client(const std::string& config_path);
    ~Client();

    std::string send_request(const std::string& command);
```

کانستراکتور این کلاس که در ورودی آدرس فایل کانفیگ را میگیرد	Client()
دیستراکتور این کلاس	~Client()
این تابع یک رشته میگیرد که همان دستوری است که باید برای سرور ارسال شود و نتیجه این دستور را به عنوان مقدار بازگشتی برمیگرداند	send_request()

متدهای خصوصی این کلاس به شرح زیر است:

```
private:
    void create_socket(int& sock);
    void connect_socket(int sock, int port) const;
    std::string receive_response(const std::string& command) const;
    void read_config_file(const std::string& config_path);
```

این تابع یک رفرنس به عنوان ورودی میگیرد و یک سوکت باز میکند و سوکت دیسکریپتور تولید شده را به ورودی assign میکند	create_socket()
سوکت به آدرس متصل میشود	connect_socket()
این تابع پاسخ متناظر با دستور ارسال شده را از سرور دریافت میکند	receive_response()
این تابع فایل کانفیگ را میخواند و فیلدهای مربوط را پر میکند	read_config_file()

فیلدهای این کلاس به شرح زیر هستند:

```
static constexpr size_t DEFAULT_BUFFER_SIZE = 4096;
static constexpr char DEFAULT_SERVER_IP[] = "127.0.0.1";

int socket_descriptor;
int data_socket_descriptor;
int command_channel_port;
int data_channel_port;
```

DEFAULT_BUFFER_SIZE	اندازه بافر دیفالت که کاربردهای متعددی دارد
DEFAULT_SERVER_IP	آدرس پیش فرض سرور که لوکال هاست است
socket_descriptor	سوکت مربوط به کانال دستور
data_socket_descriptor	سوکت مربوط به کانال داده
command_channel_port	پورت مربوط به کانال دستور
data_channel_port	پورت مربوط به کانال داده

حال به بررسی پیاده سازی این توابع میپردازیم.

فایل client.cpp:

تابع create_socket:

```
void Client::create_socket(int& sock)
{
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        throw system_error(errno, system_category(), "Creating socket failed");
}
```

این تابع یک سوکت جدید میسازد و در صورت عدم موفقیت یک استثنا پرتاب میکند.

تابع conncet_socket:

```
void Client::connect_socket(int sock, int port) const
{
    struct sockaddr_in server_address;

    inet_pton(AF_INET, DEFAULT_SERVER_IP, &server_address.sin_addr);
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(port);

    if (connect(sock, reinterpret_cast<struct sockaddr*>(&server_address), sizeof(server_address)) < 0)
        throw system_error(errno, system_category(), "Connection failed");
}
```

در ابتدا یک ساختار آدرس ساخته میشود و مقادیر آن ست می‌شود. سپس به آدرس داده شده اتصال صورت میگیرد. در صورت عدم موفقیت یک استثنا پرتاب میشود.

تابع read_config_file:

```
void Client::read_config_file(const std::string& config_path)
{
    Json::CharReaderBuilder reader;
    Json::Value config_root;
    ifstream config_file(config_path);
    config_file >> config_root;

    command_channel_port = config_root["commandChannelPort"].asInt();
    data_channel_port = config_root["dataChannelPort"].asInt();
}
```

در این تابع فایل کانفیگ خوانده میشود (مانند سمت سرور) و پورت دو کانال داده و دستور از آن استخراج میشود.

کانستراکتور Client:

```
Client::Client(const std::string& config_path)
{
    read_config_file(config_path);
    create_socket(socket_descriptor);
    create_socket(data_socket_descriptor);
    connect_socket(socket_descriptor, command_channel_port);
    connect_socket(data_socket_descriptor, data_channel_port);
}
```

ابتدا فایل کانفیگ خوانده میشود. بعد از آن سوکت‌های داده و دستور ایجاد میشوند و هردوی آنها متصل میشوند.

دیستراکتور Client~:

```
Client::~Client()
{
    close(socket_descriptor);
    close(data_socket_descriptor);
}
```

در دیستراکتور سوکت‌های تولید شده بسته میشوند.

```
string Client::send_request(const string& command)
{
    if (send(socket_descriptor, command.c_str(), command.size(), 0) < 0)
        throw system_error(errno, system_category(), "Sending data failed");

    if (send(data_socket_descriptor, command.c_str(), command.size(), 0) < 0)
        throw system_error(errno, system_category(), "Sending data failed");

    return receive_response(command);
}
```

در این تابع ابتدا دستور داده شده به هر دو سوکت داده و دستور ارسال میشود. علت اینکه به هر دو ارسال میشود این است که در سمت سرور هر دو اکسپت شوند و بتوان پاسخ را در صورت نیاز به هر دو داد. (دستوراتی مانند ls از هر دو سوکت پاسخ ارسال میکنند). در انتها نیز تابع receive_response صدا زده میشود تا پاسخ دستور ارسال شده از سرور دریافت شود و به عنوان مقدار بازگشتی بازگردانده شود.

تابع receive_response:

```
int size;
uint8_t buffer[DEFAULT_BUFFER_SIZE];
string command_type, command_arg;
istringstream ss(command);
ss >> command_type;
```

در ابتدای تابع تعدادی متغیر تعریف میشود. همچنین نوع دستور استخراج میشود و در متغیر command_type ذخیره میشود.

```
if (command_type == "ls")
{
    memset(buffer, 0, DEFAULT_BUFFER_SIZE);
    if ((size = recv(data_socket_descriptor, buffer, DEFAULT_BUFFER_SIZE, 0)) < 0)
        throw system_error(errno, system_category(), "Receiving response failed");
    cout << buffer;
}
```

اگر دستور ls بود لیست فایل‌ها از کانال داده ارسال شده است. پس این قسمت پاسخ را از کانال داده میخواند و در صورت عدم موفقیت یک استثنا پرتاب میکند. بعد از دریافت پاسخ آن را چاپ میکند.

```
if (command_type == "retr")
{
    ss >> command_arg;
    string file_name;
    if (command_arg.find('/') != string::npos)
        file_name = command_arg.substr(command_arg.rfind("/") + 1, command_arg.size());
    else
        file_name = command_arg;

    char file_size[DEFAULT_BUFFER_SIZE];
    if ((size = recv(data_socket_descriptor, file_size, DEFAULT_BUFFER_SIZE, 0)) < 0)
        throw system_error(errno, system_category(), "Receiving response failed");

    int length = atoi(file_size);
    if (length > -1)
    {
        fstream file(file_name, fstream::out | fstream::trunc);
        memset(buffer, 0, DEFAULT_BUFFER_SIZE);
        while (length > 0 && ((size = recv(data_socket_descriptor, buffer, DEFAULT_BUFFER_SIZE, 0)) > -1))
        {
            file.write(reinterpret_cast<char*>(buffer), size);
            memset(buffer, 0, DEFAULT_BUFFER_SIZE);
            length -= size;
        }
        file.close();
    }
}
```

اگر دستور retr بود ابتدا اسم فایل استخراج میشود (توجه کنید پیاده سازی به صورتی است که میتوان مسیر یک فایل را برای دانلود در این دستور استفاده کرد) بعد از آن اندازه فایل که از سمت سرور ارسال شده دریافت میشود. حال اگر به خاطر داشته باشید وقتی این مقدار از منفی یک بیشتر است یعنی مشکلی پیش نیامده است. پس یک فایل با نام فایل خواسته شده ایجاد میشود و هر بلاک از داده که از سمت سرور ارسال میشود در آن ذخیره میشود. در نهایت فایل ذخیره میشود. توجه کنید هنگامی دریافت داده به اتمام میرسد که تعداد بایت‌های خوانده شده با اندازه فایل برابر باشد. (هر دفعه تعداد بایت‌های خوانده شده از اندازه کم میشود تا به صفر برسد)

```
memset(buffer, 0, DEFAULT_BUFFER_SIZE);
if ((size = recv(socket_descriptor, buffer, DEFAULT_BUFFER_SIZE, 0)) < 0)
    throw system_error(errno, system_category(), "Receiving response failed");

string result(reinterpret_cast<char*>(buffer));
result.resize(size);
return result;
```

در نهایت نیز پاسخ به دستور از کانال دستور خوانده میشود و در صورت عدم موفقیت استثنا پرتاب میشود. این پاسخ به صورت یک رشته در آورده میشود و به عنوان مقدار بازگشتی بازگردانده میشود.

فایل client_main.cpp

```cpp
int main(int argc, char** argv)
{
    if (argc < 2)
    {
        cerr << "Too few args" << endl;
        return 1;
    }
    Client* client = new Client(argv[1]);
    while (true)
    {
        string command, response;
        getline(cin, command);
        response = client->send_request(command);
        cout << response << endl;
        if (response == QUIT_SUCCESS)
            break;
    }

    return 0;
}
```

در این فایل main کلاینت وجود دارد. ابتدا چک میشود که آرگومان‌های خط فرمان به تعداد باشد. سپس یک کلاینت با آدرس فایل کانفیگ داده شده در خط فرمان ساخته میشود. در یک حلقه بی نهایت از ورودی دستور خوانده میشود و تابع send_request صدا زده میشود. پاسخ این تابع به عنوان نتیجه دستور به کاربر نمایش داده میشود. همچنین اگر پاسخ خروج موفقیت آمیز بود از حلقه خارج میشویم و اجرای برنامه خاتمه می‌یابد.