طراحی سرور FTP توسط برنامه نویسی سوکت

این برنامه شامل دو فایل اصلی client.py و server.py و یک فایل مشترک config.py می‌باشد. همان‌طور که از نام‌ها بر می‌آید، بخش سرور پروژه در فایل server.py و بخش کلاینت نیز در فایل client.py تعریف شده‌اند. فایل config.py نیز شامل توابع و داده‌های مشترکی بین این دو، مثل اطلاعات پیکر بندی سرور مثل ip و پورت استفاده استفاده در اتصال سوکت، لیست دستورات برنامه‌ی کلاینت می‌باشد.

محتوای فایل کانفیگ:

```
mport tqdm

TCP_IP = "localhost" # ocal server
TCP_PORT = ٤١٩٢٣
BUFFER_SIZE = ١٠٢٤ # Standard buffer size
```

لیست دستورات برنامه‌ی کلاینت، که به راحتی با تغییر مقادیر دیکشنری CMDs، می‌توان اقدام به تغییر این دستورات کرد:

```
CMDs = {'connect': '.$', 'download': '.dl', 'upload': '.+', 'remove': '.-', 'fetch':
'...', 'exit': '.x', 'disconnect': '.!'}
```

پوشه های پیش‌فرض:

```
(SERVER_DIR, CLIENT_DIR) = ("server", "client")
```

تابع short_size با ظیفه‌ی تبدیل اندازه‌ی فایل‌ها از بایت به واحدهای مناسب‌تر:

```
def short_size(size_byte):
    units = ('eb', 'tb', 'gb', 'mb', 'kb', 'b')
    index = len(units) - ١

    while size_byte >= ١٠٢٤ and index < len(units):
        index -= ١
        size_byte /= ١٠٢٤

    return "%.٢f %s" % (size_byte, units[index])
```

و تابع زیر نیز وظیفه‌ی ساخت یک نوار وضعیت، برای زمان آپلود یا دانلود فایل‌ها را دارد، که برای این کار از کتابخانه‌ی خارجی tqdm استفاده شده است.

```
def make_progress(filename, filesize):
    return tqdm.tqdm(range(filesize), f"file: {filename}", unit="B", unit_scale=True,
unit_divisor=١٠٢٤)
```

برنامه‌نویسی این پروژه تماماً شی‌گرا می‌باشد. در بخش سرور یک کلاس اصلی به نام FtpServer تعریف شده، که یک سرور مستقل می‌باشد و در کد برنامه، با ساخت یک شی از روی آن و فراخوانی تابع standby سرور شروع به فعالیت می‌کند. آی‌پی و پورت استفاده شده برای اتصال سوکت و همچنین حجم بافر در حین تبادل داده را می‌توان به عنوان پارامتر به آبجکت کلاس FtpServer ارائه داد که البته درصورت عدم ارائه آن‌ها مقادیر پیش‌فرض فایل config.py درنظر گرفته می‌شوند. البته درصورت استفاده از مقادیر غیر پیشفرض، همین مقادیر باید به کلاس ClientInterface در فایل client.py نیز ارائه شود. همچنین حین تعریف آبجکت FtpServer می‌توان نام پوشه‌ی مخصوص آپلود سرور را نیز تعیین کرد، که البته درصورت عدم تعیین آن، مقدار پیشفرض فایل کانفیگ (server) به عنوان نام پوشه در نظر گرفته می‌شود. نمونه ای از تعریف و راه اندازی آبجکت سرور:

```
FtpServer(dir = input("enter the relative path of the folder you want to be shared: ") or SERVER_DIR).standby()
```

همانطور که مشاهده می‌شود، درصورتی که کاربر در این بخش نام پوشه‌ی خاصی را اگر مطرح نکند و ورودی خالی اعمال کند، همان پوشه پیشفرض سرور انتخاب خواهد شد. همین موارد برای یک آبجکت ClientInterface نیز صدق می‌کند .پوشه دانلود کلاینت به صورت پیشفرض client نام گذاری می‌شود، که البته در فایل کانفیگ قابل تغییر است و همچنین هنگام شروع برنامه کلاینت، نیز همانند بخش سرور، می‌توان یک پوشه خاص نیز مشخص نمود. در صورت وارد کردن رشته‌ی خالی در این بخش، همان فولدر پیشفرض کلاینت انتخاب می‌شود. نمونه ای از تعریف و راه اندازی آبجکت کلاینت اینترفیس:

```
ClientInterface(dir = input("enter the relative path of the folder you want to download files into: ") or CLIENT_DIR).standby()
```

در توابع سازنده هر دوی این کلاس‌ها صرفا مقداردهی فیلدهای self.ip، self.port، self.buffer_size، self.dir و برخی متغیرهای دیگر (مثلا socket) و ساخت پوشه‌های مربوطه (درصورت عدم وجود) صورت می‌پذیرد. سپس با فراخوانی تابع standby در سرور، سرور شروع به گوش دادن برای اتصال کاربران جدید و مدیریت درخواست‌هایشان می‌کند. اجرای این تابع در بخش کلاینت نیز، وظیفه‌ی نمایش منو و دریافت دستورات کاربر بصورت خط به خط دارد؛ ضمن اینکه برنامه‌ی کلاینت طوری طراحی شده‌است که کاربر قادر باشد چند دستور را بصورت پشت سر هم و در یک خط هم اجرا کند. یعنی کاربر می‌تواند چند دستور به صورت تک به تک در خط های مستقل وارد کند، و یا تمام آن ها را در یک خط و متوالی وارد کند و برنامه آن-ها را پشت سر هم اجرا می‌کند. ابتدا کد بخش کلاینت را بررسی کنیم:

تابع استندبای ابتدا منو را چاپ می‌کند و سپس منتظر ورود دستور وردی توسط کاربر می‌ماند، پس از وارد شدن دستور، اقدام به پردازش دستور(ها) می‌کند.

```
    def standby(self):
        print(self.get_menu())

        # standby:
        while True:
                statement = input("\n----------------------------------------command
line----------------------------------------------\n ")
            self.process(statement)
```

تابع get_menu متن منو را بصورت منظم تولید می‌کند تا بعدا در تابع standby استفاده گردد.

```
    def get_menu(self):
        return '\n\tcommands manual\t\n-----------------------------------------------
-----------------------------------------------\n%s\t\t: connect to server\n' %
CMDs["connect"] + \
               '%s file_path \t: upload a file\t\n%s\t\t: fetch files list\n' %
(CMDs["upload"], CMDs["fetch"]) + \
               '%s file_path \t: download a file\t\n%s file_path \t: remove a
file\n%s                \t: disconnect\n' % (CMDs["download"], CMDs["remove"],
CMDs['disconnect']) + \
            '%s           \t: exit' % (CMDs['exit'])
```

و نهایتا تابع process ورودی کاربر را تفسیر می‌کند. این تابع در واقع ابتدا متن ورودی را بر حسب کاراکتر اسپیس از هم جدا کرده و تبدیل به آرایه می‌کند. سپس به دنبال به کلماتی که با کاراکتر . شروع شده‌اند میگردد. این کلمات مشخص کننده‌ی یکی از دستورات شش گانه هستند و در جلوی هر کدام از آن‌ها پارامترهای مربوط به هر دستور قرار دارند، با توجه به چنین معماری‌ای، دستورات می‌توانند به صورت متوالی هم وارد شوند؛ برای مثال دستور آپلود فایل به این صورت خواهد بود:

```
.+ file_name file_hierarchy_in_server
```

تابع process هر کدام از این دستورات را شناسایی می‌کند، سپس تابع مربوطه را همراه با پارامترهای مربوطه فراخوانی میکند.

```
    def process(self, statement):
        try:
            print(statement)
            terms = statement.split()
            for i, term in enumerate(terms):
```

```
                if term[۰] == '.': # dot is commands start sign
                    lwrterm = term.lower()
                    if lwrterm == CMDs['connect']:
                        print("\n----------------------------------------connection--
----------------------------------------\n ")
                        self.connect()
                    elif lwrterm == CMDs['upload']:
```

برای جلوگیری از طولانی شدن، کد بصورت فشرده آورده شده است.

و اما تابع `standby` در سرور به شرح زیر است:

```
    def standby(self):
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((self.ip, self.port))
        self.server.listen(۱)
         print(f"server started listening on {self.ip}:{self.port}; server is now on
stand by for new clients...")

        while True:
            client_socket, ip_address = self.server.accept()

            client = Client(socket=client_socket, ip=ip_address)

            # receive the client's username
            thread = threading.Thread(target=self.listen۲, args=(client,))
            thread.start()
```

این تابع ابتدا سرور را آماده سازی می‌کند و آن را روی آدرس مشخص شده `bind` می کند. نهایتا در حلقه بی‌نهایت `while True` سرور همواره منتظر اتصال کلاینت‌ها می‌ماند، با اتصال هر کاربر، سرور توسط دستور `self.server.accept()` آدرس و آبجکت سوکت آن کلاینت را بدست می‌آورد. تمامی ارتباط‌های آینده با کاربر توسط همین آبجکت `socket` انجام میپذیرد. برای راحتی کار با کلاینت‌ها کلاس دیگری با نام `Client` تعریف شده است. که در ادامه به بررسی آن می‌پردازیم. برای مدیریت پیام‌های هر کاربر، نیاز است که در ی حلقه‌ی لامتناهی دیگر منتظر ارسال داده از سوی وی بمانیم، که این امر مشکلی پیش میآورد؛ اینکه س از اتصال کاربر اول دیگر هیچ کاربر دیگری نمی‌تواند به سرور متصل شود؛ چون سرور همواره در حلقه بی‌نهایت مربوط به کلاینت اول گیر می‌کند. برای حل این مشکل از تکنیک برنامه نویسی چندنخی (`multithread programming`) استفاده می‌شود.

بدین صورت، هر کلاینت در نخ مجزای خودش توسط سرور مدیریت می‌شود و سرور در حلقه‌ی بی-نهایت خاصی گیر نمی‌افتد.

```
class Client:
    objs = {}
```

```
    def __init__(self, socket, ip) -> None:
        self.socket = socket
        self.ip = ip
        self.id = f'{ip[٠]}{ip[١]}'.replace('.', '')
        self.connection_date = time.ctime()
        while self.id in Client.objs:
            self.id += str(randrange(٠, len(Client.objs)))
        Client.objs[self.id] = self
        print(f"{self.ip} [id: {self.id}] has been connected!")

    def disconnect(self):
        if self.socket:
            self.socket.close()
        if self.id in Client.objs:
            del Client.objs[self.id]
        print(f"{self.ip} [id: {self.id}] has been disconnected!")

    def disconnect_all(self):
        for each in Client.objs.values():
            each.disconnect()
        Client.objs.clear()

    def synchronize(self):
        self.socket.send(b"١")
```

کل کلاس کلاینت به صورت بالا می‌باشد. این کلاس، وظیفه‌ی ذخیره آبجکت سوکت و آدرس کلاینت و تاریخ اتصال وی را دارد. همچنین در سازنده‌ی این کلاس، طبق یک الگوریتم ساده، از روی آیپی و شماره پورت کلاینت، یک آیدی منحصر به فرد هم برای وی ساخته می‌شود. از سوی دیگر، این کلاس وظیفه‌ی ذخیره‌ی داده‌های همه کلاینت‌ها را توسط متغیر استاتیک objs دارد؛ بدین صورت که در انتهای تابع سازنده‌ی این کلاس، هر ابجکت به این لیست objs اضافه می‌شود. در نتیجه بصورت اوماتیک با ساختن هر کلاینت، لیست کل کلاینت‌ها اپدیت می‌شود که توسط کلید آیدی، می‌توان به هر کلاینت مورد نظر دسترسی یافت. یک تابع هم با وظیفه‌ی دیسکانکت کردن کاربر از سرور وجود دارد که وظیفه‌ی بستن سوکت ارتباطی بین سرور و کلاینت و پاک کردن مشخصات آن کاربر از لیست objs را دارد. تابع disconnect_all هم وظیفه خالی کردن سرور از تمامی کلاینت‌هایش را دارد. یک تابع synchronize هم وجوددارد که معادل آن در بخش ClientInterface نیز وجود دارد که این تابع وظیفه‌ی همگام سازی کلاینت و سرور حین تبادل اطلاعات را دارد.

با توجه به توضیح کلاس Client، می‌توان دید که در تابع standby سرور، همواره یک ابجکت از کلاینت ساخته می‌شود که این ابجکت به عنوان پارامتر به نخ مجزای اجرای تابع listen۲ برای ان کلاینت

خاص پاس داده می‌شود. از این رو تابع listen۲ که برای هر کلاینت بصورت مجزا کار می‌کند، تمام داده‌های مورد نیاز از آن کاربر را در دسترس خواهد داشت. همچنین با ساخت هر آبجکت، لیست کل کلاینت‌ها هم به صورت اتوماتیک اپدیت میگردد.

```
    def listen۲(self, client = None):
        # listen to a specific client
        if not client:
            return
        try:
            while True:
                # Enter into a while loop to recieve commands from client
                print("\n\twaiting for instruction")
                data = client.socket.recv(self.buff_size).decode()
                print(f"recieved instruction: {data}")
                # Check the command and respond correctly
                operation = ''
                if data == CMDs['upload']:
                    operation = 'uploading'
                    self.upload(client)
                elif data == CMDs['fetch']:
                    operation = 'fetching'
                    self.fetch(client)
                elif data == CMDs['download']:
                    operation = 'downloading'
                    self.download(client)
                elif data == CMDs['remove']:
                    operation = 'removing'
                    self.remove(client)
                elif data == CMDs['disconnect'] or data == CMDs['exit'] or not data:
                    client.disconnect()

        except Exception as e:
            print(f"something went wrong while {operation} because: ", str(e), "\n\t
... disconnecting...")
            if client:
                client.disconnect()
```

تابع listen۲ برای هر کاربر وظیفه گوش دادن به وی، از طریق درگاه سوکت مربوطه را دارد. با دریافت هر پیام جدید از کلاینت، این تابع، تعیین می‌کند کلاینت قصد انجام کدام یک از عملیات پنجگانه را دارد. در هر کدام یک از موارد، این تابع با فراخوانی تابع مناسب اقدام به مدیریت درخواست کاربر می‌کند. همچنین یک متغیر operation تغریف شده است. درصورتی که با ایجاد مشکل در حین اجرای دستورات این تابع، برنامه به

بخشexcept از ساختار try-except هدایت گردد، از طریق این فیلد می‌توان به کاربر اطلاع داد در کدام یک از عکلیات مشکل پیش آمده است.

در ادامه هر یک از عملیات ۵ گانه اصلی را شرح می‌دهیم؛ با توجه به اینکه ساختار برنامه شرح داده شده‌است درک این توابع نیازی به توضیح خط به خط کدها نیست و ما فقط به بررسی الگوریتم این عملیات ها میپردازیم.

عملیات اتصال:

در این بخش، کلاینت با اجرای دستور connect روی آبجت سوکت خودش، اقدام به اتصال به سرور می‌کند؛ بصورت زیر:

```
self.socket.connect((self.ip, self.port))
```

طبیعی است که برای این اتصال، باید آدرس آیپی و پورت با آدرس سرور مطابقت داشته باشد. با اجرای این تابع، درخواست این کلاینت به سرور ارسال می‌شود؛ اینجاست که سرور با اجرای دستور server_socket.accept() اقدام به پذیرش این کاربر می‌کند. که کد مربوط به آن در بالا در تابع standby قرار داده شده است.

دستور اتصال در بخش کلاینت به صورت زیر می‌باشد و نیاز به پرامتری هم ندارد:

.$

عملیات آپلود:

کاربر با اجرای دستور زیر اقدام به آپلود یک فایل میکند:

مسیر_درختی_فایل_در_سرور آدرس_فایل +.

پس از اجرای این دستور کلاینت، با اجرای تابع upload خود، اقدام به ارسال درخواست اپلود به سرور میکند و با ارسال داده مناسب اعلام میکند که آماده ارسال فایل می‌باشد:

```
def communicate(self, data):
    self.socket.send(data.encode('utf-۸'))
```

سپس سرور با اجرای self.synchronize() اقدام به همگام شدن با کلاینت کرده و اعلام امادگی برای دریافت مشخصات فایل میکند.

برای ارسال درخواست‌های این چنینی از سمت کاربر به سرور (یا بالعکس)، باید رشته‌ی مربوطه به ساختار بایت تبدیل شود که برای این منظور تابع communicate در بخش کلاینت تعریف شده است.

سرور نیز با دریافت درخواست کلاینت، با فراخانی تابع upload خود، آماده دریافت میشود. در این بین، کلاینت با ترکیب اسم اصلی فایل، با ساختار درختی موردنظر کاربر، آدرس جدید فایل را در سرور طراحی می‌کند. سپس از طریق sys.getsizeof تعداد بایت مورد نیاز برای ارسال چنین ادرسی را بدست آورده، سپس این سایز را به صورت داده‌ی ۲ بایتی برای سرور ارسال می‌کند و سپس سرور به صورت زیر این عدد را دریافت می‌کند:

```
struct.unpack("h", client.socket.recv(۲))[۰]
```

پس از آن سرور با دریافت این عدد، دقیقا میداند که برای دریافت نام و آدرس فایل دقیقا باید انتظار چند بایت را داشته باشد:

```
client.socket.recv(file_name_size).decode()
```

فراخوانی تابع decode به این خاطر است که داده‌ها به صورت بایت ارسال شده‌اند و نیاز به تبدیل مجدد به رشته وجود دارد.

سرور پس از دریافت آدرس کامل فایل، اقدام به تجزیه رشته بر حسب / کرده و نام تک تک پوشه‌های مسیر فایل را بدست می‌آورد. سپس اقدام به ساخت تمام آن پوشه‌هایی که وجود ندارند، می‌کند. چرا که در غیر این صورت ساخت آن فایل در مسیر مورد نظر با مشکل مواجه میشود.

پس از این مرحله، سرور با اجرای دستور client.synchronize()، به کلاینت اعلام می‌کند که اماده‌ی دریافت داده‌ی بعدی است. درنتیجه کلاینت این بار حجم فایل را به بایت، تحت داده‌ی ۴ بایتی ارسال می‌کند. حال کلاینت اقدام به ارسال تکه تکه‌ی فایل می‌کند، که حجم این تکه ها توسط سایز بافر مشخص میگردد. فایلی که در کلاینت با مود read binary (rb) باز شده است، تکه به تکه خوانده شده و برای سرور ارسال می‌شود. در این حین سرور تا زمانی که حجم دریافت شده‌اش کمتر از حجم فایل باشد به خواندن ادامه می‌دهد و آن را در فایلی که با مود wb (write binary) باز کرده است ذخیره می‌کند. در این میان نیز یک

progress bar توسط تابع make_progress ساخته می‌شود که در حلقه‌ی مربوط به دریافت فایل در سرور، اقدام به اپدیت شدن می‌کند. همین روند برای حلقه‌ی ارسال فایل در کلاینت نیز می‌افتد و کلاینت نیز پروگرس بار مخصوص خود را دارد. در نهایت هم حجم فایل، و کل زمان سپری شده‌ی عملیات، این بار از سمت سرور به کلاینت ارسال می‌شود تا کلاینت از وضعیت دریافت سرور مطلع شود.

در این بین، کد مربوط به بخش های کلاینت و سرور، توسط ساختارهای به جا و مناسب try-except مدیریت شده‌اند و هرگونه خطایی که در این بین رخ بدهد توسط این ساختار ها هندل می شود.

عملیات دانلود

دستور این عملیات به فرم زیر میباشد:

آدرس_فایل_درخواستی_در_سرور dl.

این دستور تابع download در کلاینت را فراخوانی می‌کند. این عملیات کاملا شبیه عمل آپلود می-باشد؛ با این تفاوت که در اینجا جای نقش‌های سرور و کلاینت تعویض می‌شود. ابتدا کلاینت، همانند بخش آپلود، با ارسال طول نام فایل، سرور را اماده‌ی دریافت تعداد مشخصی بایت می‌کند. طبق همان الگوریتم سرور نهایتا به ادرس فایل دست می یابد. در این حین، کلاینت، نام پوشه‌های موجود در مسیر را با تقسیم رشته‌ی آدرس فایل بر حسب /، را بدست آورده و اقدام به ساخت آن دسته پوشه‌هایی در این مسیر می‌کند که در حافظه‌ی کلاینت وجود ندارند. چرا که درغیر اینصورت عملیات ذخیره فایل روی کلاینت به مشکل می‌خورد. اگر آدرس صحیح باشد و آن فایل روی سرور وجود داشته باشد، سرور حجم فایل را برای کلاینت ارسال میکند، و این بار دقیقا بصورت برعکس بخش آپلود، سرور اقدام به ارسال تکه‌تکه‌ی فایل و کلاینت اقدام به دریافت آن می‌کند. این دریافت برای کلاینت تا زمانی که حجم داده‌ی دریافتی کمتر از حجم فایل باشد ادامه می‌یابد. همچنین در این حین، هم در کلاین و هم در سرور، progress bar مخصوص ساخته و دائم بروزرسانی می-شود تا زمانی که عملیات دانلود تکمیل گردد. پس از این کلاینت با اجرای self.synchronize() اقدام به همگام سازی با سرور و اعلام امادگی برای دریافت داده‌ی بعدی را می‌کند و سرور نیز بلافاصله مدت زمان سپری شده حین دانلود را برای کلاینت ارسال می‌کند.

یک دلیل دیگر برای همگام سازی این است که، گاهی ممکن است حین عملیات، وقتی یکی از طرفین اقدام به دو ارسال پیاپی کند، طرف متقابل ممکن است تعداد بایت بیشتری، نسبت به حجم در نظر گرفته شده در یافت کند؛ یعنی به نوعی دو داده‌ی مجزای ارسال شده دچار ادغام ناخوئاسته میشوند. در این صورت با انجام عملیات ارسال و دریافت بصورت نوبتی بین کلاینت و سرور، این مشکل حل میشود. یعنی به ازای هر ارسال از سمت طرفی، یک دریافت پس از آن انجام می‌دهد و اینگونه از ادغام بایت ها و داده‌ها جلوگیری میگردد.

عملیات دریافت لیست فایل ها

فرم این دستور به صورت زیر می‌باشد:

[مسیرخاص] ...

پس از اجرای این دستور، تابع fetch در کلاینت اجرا می‌گردد که وظیفه‌ی ارسال درخواست fetch به سرور و مدیریت آن را داردو با ارسال درخواست مربوطه به سرور، سرور شروع به ارسال لیست فایل‌های موجود در مسیر ریشه و یا مسیری خاص در پوشه‌ی مربوط به خود می‌کند. سرور پس از دریافت این درخواست، با فراوانی تابع fetch شروع به این فرایند می‌کند.

ابتدا سرور با استفاده از دستور os.listdir لیست تمامی فایل ها و پوشه های موجود در مسیر خواسته شده را بدست اورده و به صورت تک به تک اقدام به ارسال می‌کند. ابتدا تعداد کل برای کلاینت ارسال می‌شود تا کلاینت بداند چه تعداد دور حلقه برای دریافت نیاز دارد. در سمت سرور، در دور حلقه مربوط به ارسال، یک بار نام فایل و سپس حجم فایل ارسال می‌شود. اگر ایتم مورد نظر پوشه بود، به جای حجم آن، عبارت "directory" ارسال می‌شود. از سوی دیگر در کلاینت هم درهر دور حلقه این داده‌ها به ترتیب دریافت می-شوند. اگر داده‌ی ارسال حجم فایل باشد، توسط تابع short_size به حجم مناسب تبدیل می‌شود، و اگر محتوای file_detail برابر با directory بود در کنار نام پو.شه نمایش داده می شود تا کاربر بداند که این آیتم پوشه است نه فایل. در هر مسیر هم جمع کل حجم فایل‌های فقط همان مسیر در اخر نمایش داده می-شود.

در روند ارسال همواره توسط تابع synchronize در هر دو طرف، اقدام به هماهنگ سازی کلاینتو سرور میشود. همچنین کاربر می‌تواند با ارسال پارامتر اضافی(مسیر خاص)، اقدام به مشاهده‌ی محتوای یک

پوشه‌ی خاص در سرور نماید. مثلا اگر دوست نداشت صرفا محتوای مسیر root را مشاهده بکند، می‌تواند آدرس پوشه مورد نظر در سرور را وارد نماید، تا محتوای سرور در آن مسیر به نمایش درآید. مسیر مورد نظر هم توسط همان الگوریتم ارسال نام در بخش آپلود و دانلود، در ابتدای تابع fetch سرور دریافت میگردد. در تمامی این مراحل توسط استفاده درست از ساختار try-except تمامی خطاهای احتمالی مدیریت می‌شوند.

عملیات حذف

فرم این دستور در کلاینت به صورت زیر می‌باشد:

آدرس_فایل_در_سرور -.

این دستور، باعث اجرای تابع remove در کلاینت و درنتیجه ارسال درخواست remove برای سرور میگردد. پس از ارسال این درخواست به سرور توسط کلاینت، سرور با فراخوانی تابع remove اقدام به حذف می‌کند.در ابتدای این تابع، با دستور

```
file_name = client.socket.recv(self.buff_size).decode()
```

ادرس فایل مربوطه از کلاینت دریافت می‌شود. پس از بدست آوردن ادرس فایل بصورت کامل، روی سرور، ابتدا سرور بررسی می‌کند آیا چنین فایلی وجود دارد یا نه. اگر نه، با ارسال مقدار -۱ به کلاینت، به وی اطلاع می‌دهد که اشتباهی در نام فایل وجود دارد. و کلاینت هم اگر چنین مقداری دریافت کند بلافاصله پیام مناسب را نمایش می‌دهد. اما اگر فایل در سرور موجود بود، با ارسال مقدار ۱، کلاینت را آماده می‌کند تا از کاربر بپرسد آیا مطمئن هست که می‌خواهد فایل را پاک کند؟

اگر کاربر مقدار yes یا y را وارد کند، بلافاصله با ارسال این رشته به سرور، سرور اقدام به پاک سازی فایل از طریق os.remove(file_name) کرده و پس از آن نتیجه‌ی ۱ را به عنوان کد موفقیت آمیز بودن عملیانت دوباره به کلاینت می‌فرستد. در هر مرحله کلاینت پیغام مناسب را نمایش می‌دهد. ساختار try-except مثل بقیه موارد در اینجا هم استفاده شده است.

عملیات قطع اتصال و خروج

این دستورات به صورت زیر هستند و پارامتر خاصی ندارند.

خروج: .x

قطع اتصال: .!

با اجرای این دستورات، بترتیب توابع زیر اجرا می‌شوند:

```
def disconnect(self):
    try:
        if self.socket:
            self.communicate(CMDs['disconnect'])
            # Wait for server go-ahead
            self.socket.recv(self.buffer_size)
            self.socket.close()
    except:
        pass
    finally:
        print("you are now disconnected!")

def exit(self):
    self.disconnect()
    print("bye bye!")
    exit(۰)
```

این دستورات ابتدا، یک درخواست disconnect برای سرور میفرستند. دریافت این درخواست در سرور منجر به انجام client.disconnect شده که کد آن قبلا بررسی شده است. اما به طور خلاصه عملیات قطع اتصال، باعث بسته شدن اتصال سوکت بین سرور و کلاینت مربوطه می‌شود و سپس داده‌ه های مربوط به این کلاینت از سرور پاک می‌شوند تا سرور آزاد تر گردد.

تابع exit هم قبل از خارج شدن از برنامه‌ی کلاینت، ابتدا اقدام به فراخوانی disconnect می‌کند.

بدین ترتیب کلیت کار سرور و کلاینت به طور کلی و با شرح توضیحات لازم بررسی شد. توضیحات خط به خط مورد نیاز برای هر خط کد نیز درون فایل‌های کد به صورت کامنت ذکر شده‌اند.