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 = 41923

BUFFER\_SIZE = 1024 # 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) - 1

    while size\_byte >= 1024 and index < len(units):

        index -= 1

        size\_byte /= 1024

    return "%.2f %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=1024)

برنامه­نویسی این پروژه تماماً شی­گرا می­باشد. در بخش سرور یک کلاس اصلی به نام FtpServer تعریف شده، که یک سرور مستثل 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 آن­ها 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[0] == '.': # 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(1)

        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.listen2, 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 بی­نهایت خاصی گیر نمی­افتد.

class Client:

    objs = {}

    def \_\_init\_\_(self, socket, ip) -> None:

        self.socket = socket

        self.ip = ip

        self.id = f'{ip[0]}{ip[1]}'.replace('.', '')

        self.connection\_date = time.ctime()

        while self.id in Client.objs:

            self.id += str(randrange(0, 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"1")

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 listen2 برای ان کلاینت خاص پاس داده می­شود. از این رو تابع listen2 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 listen2(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 listen2 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 بخش except 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 5 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:

.+ آدرس\_فایل مسیر\_درختی\_فایل\_در\_سرور

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-8'))

       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 2 بایتی برای سرور ارسال می­کند و سپس سرور به صورت زیر این عدد را دریافت می­کند:

struct.unpack("h", client.socket.recv(2))[0]

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 4 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:

.dl آدرس\_فایل\_درخواستی\_در\_سرور

این دستور تابع download در کلاینت را فراخوانی می­کند. این عملیات کاملا شبیه عمل آپلود می­باشد؛ 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 می­شود 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 می­شوند. 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 می­شود.

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 -1 به کلاینت، به وی اطلاع می­دهد که اشتباهی در نام فایل وجود دارد. و کلاینت هم اگر چنین مقداری دریافت کند بلافاصه پیام مناسب را نمایش می­دهد. اما اگر فایل در سرور موجود بود، با ارسال مقدار 1، 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 1 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(0)

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.