

Compartilhamento de arquivos por FTP

EzShare

Gabriel Henrique do Nascimento Neres - 221029140

Guilherme Nonato da Silva - 221002020

Resumo— O presente relatório visa explicar o processo de criação de um programa que efetua uma conexão Servidor-Cliente, e possibilita o envio e download de arquivos de texto, contando com uma interface interativa para um possível usuário. Ao decorrer do desenvolvimento, foram usados os conceitos de protocolos, servidores, camadas de rede, análise de pacotes, caminho de dados, entre outros.

Palavras-chave — FTP, Servidor, Python, Wireshark.

- Exemplos de uso do cliente com o fluxo de mensagem entre cliente e servidor;
- Análise da comunicação cliente-servidor pelo *Wireshark* com a identificação dos endereços IP e a análise das mensagens para comprovar o bom funcionamento do cliente.
- Uma última para extrair conclusões dos processos feitos, além de considerar o que foi aprendido ao decorrer do projeto.

I. INTRODUÇÃO

O intuito do trabalho a ser desenvolvido é compreender e implementar um cliente capaz de se comunicar com um servidor FTP, contando com uma interface que permita a um possível usuário utilizar o programa criado intuitivamente. Utilizaremos a versão 3.11 do Python para o desenvolvimento do código, tanto para configurar o cliente quanto para desenvolver a interface gráfica. Para o servidor será utilizado o servidor FTP já implementado por padrão no sistema operacional Windows.

O presente relatório se dividirá em mais três seções:

- Uma para situar o conteúdo teórico do projeto e contextualizar o material aprendido em aula;
- Uma focada em analisar e descrever as implementações feitas e os resultados obtidos pelo programa desenvolvido, sendo subdividido em três tópicos:
 - Funcionalidades do cliente;

II. FUNDAMENTAÇÃO TEÓRICA

O projeto da construção de um cliente para um servidor do tipo FTP necessitou do estudo e a compreensão de alguns dados teóricos. Dentre as informações utilizadas podem ser elencadas os conceitos ligados ao protocolo utilizado por um servidor FTP, o funcionamento de uma comunicação TCP e os princípios da transferência de dados.

O **protocolo FTP** [3] define diversas instruções e funcionalidades baseado no protocolo do TELNET. Os principais conceitos que precisaram ser conhecidos para o desenvolvimento do cliente que iria se comunicar com o servidor foram:

- **Caminho de dados:** Para a transferência de dados entre o servidor e o cliente é preciso estabelecer um caminho de dados entre os dois. O caminho de dados é um canal que é criado para a transferência de um único "arquivo" entre o *host* e o servidor, o que cria a necessidade de iniciar

e finalizar a conexão de dados entre eles para cada solicitação feita.

- **Conexão Passiva e Ativa:** O conceito foi importante para determinar qual tipo de conexão deveria ser estabelecida pelo cliente para alcançar o objetivo desejado. Esse conceito está diretamente ligado com a criação do caminho de dados.

III. ANÁLISE DE RESULTADOS

No desenvolvimento do sistema que seria estudado foi necessária a implementação do servidor FTP. Em função da simplicidade para a configuração, o servidor FTP disponibilizado pelo sistema operacional Windows foi utilizado para ser o servidor a ser acessado pelo cliente desenvolvido.

O servidor não precisou de nenhuma configuração especial. Foram definidos o IP do servidor, que para facilitar a obtenção dos dados que serão utilizados posteriormente foi utilizado o 127.0.0.1 (Indicativo de rede local), a porta de acesso 21 (padrão do servidor FTP) e um usuário e senha para acessar o servidor. Dessa forma, são concluídas as configurações do servidor.

Na próxima parte da análise será definida como foi feita a implementação do cliente. Para isso, a explicação foi dividida em subseções para a obtenção de uma melhor organização dos pontos observados.

A. Funcionalidades do cliente

Na busca de um cliente que desempenhasse todas as ações desejadas, foram necessários diversos comandos FTPs que precisaram ser reunidos em "macro"funções que seriam capazes de realizar todas as comunicações básicas entre cliente e servidor.

A primeira funcionalidade implementada foi a de criação do socket para a realização da comunicação. Para isso será utilizada uma biblioteca padrão do Python chamada Socket [2]. O código para isso pode estar definido como:

```
def criarSocket(host,port):
    socket = s.socket(s.AF_INET,
        s.SOCK_STREAM)
    endereco = (host, port)
    socket.connect(endereco)
    receberMensagem(socket)
    return socket
```

Com o socket definido, pode ser feita a conexão e definição das configurações do servidor. Para isso será usado o usuário e senha criados e será definido o tipo de código ASCII utilizado.

```
# Faz login no servidor
def Login(socket, user, senha):
    usuario = "user {u}\r\n"
    .format(u = user)
    comunicar(socket, usuario) # Login
    password = "pass {s}\r\n"
    .format(s = senha)
    comunicar(socket, password) # Senha

# Estabelece uma conexao com o
# servidor (com login) e configura
def Conectar(Usuario, Senha):
    socket = criarSocket(host, port)
    Login(socket, Usuario, Senha)
    comunicar(socket, "opts UTF8 ON\r\n")
    return socket
```

Sendo assim estão definidas as funções de conectar e logar ao servidor a partir de um socket feito pelo cliente. As próximas funções a serem definidas são a de envio e recebimento de mensagens do servidor.

Envio: Manda a mensagem informada pelo socket.

```
# Tenta enviar uma mensagem ao servidor
def mandarMensagem(socket, mensagem):
    try:
        socket.settimeout(0.2)
        socket.send(mensagem.
            encode('utf-8'))
    except:
        print("Time out")
```

Recebimento: Recebe todas as mensagens vindas pelo socket até que ocorra o *timeout* durante alguma execução do loop.

```
# Tenta recebe as mensagens do servidor
```

```
def receberMensagem(socket):
    Timeout = False
    mensagemTotal = ""
    Vazio = 0
    # Recebe so servidor ate superar
    #o limite de tempo
    while not Timeout:
        try:
            socket.settimeout(0.2)
            resposta = socket.recv(2048)
            dados = resposta.decode('utf-8')
            if dados in ["", "\n", "\r", "\r\n"]:
                Vazio += 1
                if Vazio >= 2:
                    return mensagemTotal
            else:
                print(dados)
                mensagemTotal = mensagemTotal
                                + dados
        except:
            Timeout = True
    return mensagemTotal
```

Com essas funções, é possível realizar todo o tipo de interação com o servidor, mas para simplificar a implementação da interface do cliente foram construídas mais funções. Dentre as principais funções construídas está a responsável por iniciar um caminho de dados.

```
def caminhoDeDados(socket):
    mandarMensagem(socket, "pasv\r\n")
    dados = receberMensagem(socket)
    eliminar = ",.()\r\n"
    endereco = [item for item in dados
                .split(" ")[-1].split(",")]
    P1 = endereco[-2]
    P2 = "".join([caracter for caracter in
                  endereco[-1] if caracter not in
                  eliminar])
    porta = int(P1)*256 + int(P2)
    caminho = s.socket(s.AF_INET, s.
                       SOCK_STREAM)
    end = (host, porta)
    caminho.connect(end)
    receberMensagem(caminho)
    return caminho
```

Neste trecho de código o comando PASV é enviado ao servidor para iniciar uma conexão passiva com o mesmo. Utilizando o IP obtido dessa conexão, um novo socket é criado para

a transmissão de dados e é retornado para a parte do código que solicitou esse socket. É importante ressaltar que o socket gerado só pode ser usado para uma única comunicação troca de dados com o servidor.

Utilizando tudo que foi construído até agora, é possível construir funções que utilizem retornos do caminho de dados. Em todas elas, o comando será enviada ao servidor pelo socket inicial após a criação do caminho de dados e a resposta esperada será obtida pelo socket do caminho de dados. As respostas obtidas pelo socket normal são usadas para garantir que a transição ocorreu como o esperado.

A primeira destas implementações é uma função que retorne tudo que está em um determinado diretório do servidor.

```
#Lista os objetos no caminho indicado
def Listar(socket, path):
    caminho = caminhoDeDados(socket)
    mensagem = "list "+path+final
    mandarMensagem(socket, mensagem)
    confirmacao = receberMensagem(socket)
    conteudo = receberMensagem(caminho)
    receberMensagem(socket)
    caminho.close()
    return confirmacao, conteudo
```

A próxima função solicita um arquivo do servidor e cria uma cópia no cliente em um diretório especificado.

```
def Download(socket, pathS, pathL):
    caminho = caminhoDeDados(socket)
    mensagem = "RETR "+pathS+final
    mandarMensagem(socket, mensagem)
    confirmacao = receberMensagem(socket)
    conteudo = receberMensagem(caminho)
    receberMensagem(socket)
    caminho.close()
    criarArquivo(pathL, conteudo)
    return confirmacao
```

```
def criarArquivo(pathL, conteudo):
    try:
        if type(pathL) == type(["lista", "de",
                                "teste"]):
            path = "".join(x+" " for x in pathL)
            with open(file=path, mode = "x")
```

```

                                as f:
        f.write(conteudo)
    else:
        with open(pathL, "x") as f:
            f.write(conteudo)
    except FileNotFoundError:
        print("O path tem problemas")
    except FileExistsError:
        print("Arquivo com o mesmo nome")

```

De maneira similar é possível enviar um arquivo de um diretório específico do cliente ao servidor.

```

def Upload(socket, pathS, pathL):
    caminho = caminhoDeDados(socket)
    path = "".join(x+" " for x in pathS)
    mensagem = "STOR "+path+final
    texto = mandarArquivo(caminho, pathL)
    mandarMensagem(socket, mensagem)
    confirmacao = receberMensagem(socket)
    conteudo = receberMensagem(caminho)
    receberMensagem(socket)
    caminho.close()
    return confirmacao

def mandarArquivo(socket, L):
    try:
        if type(pathL) == type(["lista"]):
            p = "".join(x+" " for x in L)
            mandarMensagem(socket, lerArq(p))
        else:
            mandarMensagem(socket, lerArq(L))
    except FileNotFoundError:
        print("Path tem problemas")
    except FileExistsError:
        print("Arquivo com o mesmo nome")

def lerArq(pathLocal):
    with open(pathLocal, "r") as f:
        tc = f.readlines()
        string = "".join([x for x in tc])
    return string

```

Dessa forma, são apresentadas todas as funções que foram implementadas no cliente.

B. Exemplos de uso do cliente

Para exemplificar o funcionamento do cliente, podem ser observados dois processos de comunicação com o servidor. No primeiro processo, o cliente apenas estabelece a conexão com

o servidor para deixá-lo disponível para poder enviar e receber arquivos posteriormente.

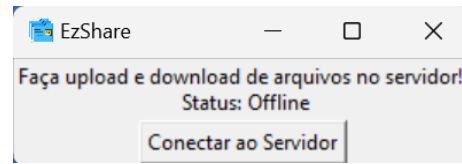


Fig. 1: Tela com o servidor desconectado.

Ao clicar no botão de "Conectar ao Servidor" o cliente inicia uma série de comandos ao servidor. Primeiramente, é enviada uma mensagem TCP do cliente ao servidor para iniciar o HandShaking. Com o recebimento da resposta do servidor e o envio da confirmação por parte do cliente. Com isso é dado início os processos diretamente ligados ao protocolo FTP.

O usuário do login é enviado ao servidor, e a resposta obtida é o pedido de uma senha. Enviando a senha é obtida a confirmação da conexão com o servidor e permitindo a troca de informações internas do servidor. A última configuração realizada na conexão com o servidor é a definição do tipo de ASCII utilizado como UTF8.

```

220 Microsoft FTP Service
331 Password required
230 User logged in.
200 OPTS UTF8 command successful - UTF8 encoding now ON.

```

Fig. 2: Respostas do servidor.

O segundo processo que pode ser observado é feito após a conexão com o servidor. Como inicialmente o servidor se encontrará vazio, o cliente desejará transferir algum arquivo ao servidor. Partindo dessa ideia, o processo pode ser descrito abstraindo a parte referente a conexão com o servidor.

Para iniciar o processo de upload, basta o usuário clicar no botão "Abrir" e irá ser solicitado

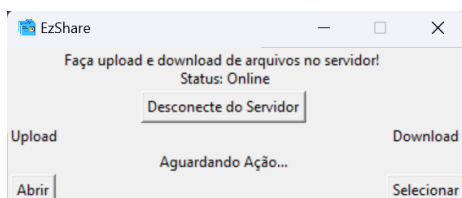


Fig. 3: Tela com o servidor conectado.

o arquivo desejado no sistema. Neste ponto são iniciadas as preparações para as comunicações Cliente-Servidor.

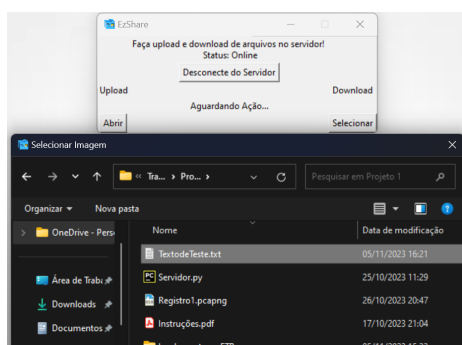


Fig. 4: Após clicar no botão, pode ser escolhido o arquivo desejado.

Tendo armazenado o endereço do arquivo no computador do cliente e o socket de comunicação já criado, o botão de enviar arquivo pode ser selecionado para determinar em que pasta do servidor deve ser colocado o arquivo desejado e o nome do arquivo.

Neste ponto, é iniciada a comunicação efetiva entre cliente e servidor. Utilizando a função "Upload" criada no cliente é iniciada a comunicação. Um novo caminho de dados será aberto, para isso será enviado um comando "PASV" ao servidor, o qual deverá retornar o IP e uma porta de acesso ao cliente.

Com a nova porta informada um novo socket é iniciado com o servidor para servir de caminho de dados. Com o caminho de dados aberto, os dados do arquivo são enviados por ele e em seguida o comando "STOR" é enviado ao

servidor pelo socket inicial. Com isso, o servidor é capaz de compreender que deve salvar os dados enviados no formato de texto indicado por padrão.

Tendo recebido corretamente todos os dados, uma resposta é retornada ao cliente e o caminho de dados é fechado. Sendo assim, o processo de envio de dados ao servidor está finalizado.

C. WireShark

Para analisar o funcionamento do cliente, o segundo cenário apresentado nos exemplos de funcionamento do cliente foi utilizado como base. Para a análise, foi observado o seguinte trecho da captura.

```

FTP 50 Request: pasv
TCP 44 21 → 52842 [ACK] Seq=256 Ack=60 Win=2161152 Len=0
FTP 92 Response: 227 Entering Passive Mode (127,0,0,1,28
TCP 44 52842 → 21 [ACK] Seq=60 Ack=304 Win=327168 Len=0
TCP 56 52849 → 52848 [SYN] Seq=0 Win=65535 Len=0 MSS=654
TCP 56 52848 → 52849 [SYN, ACK] Seq=0 Ack=1 Win=65535 Le
TCP 44 52849 → 52848 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
FTP-DA... 109 FTP Data: 65 bytes (PASV) (STOR \Teste8.txt)
TCP 44 52848 → 52849 [ACK] Seq=1 Ack=66 Win=2161152 Len=
FTP 62 Request: STOR \Teste8.txt
TCP 44 21 → 52842 [ACK] Seq=304 Ack=78 Win=2161152 Len=0
FTP 98 Response: 125 Data connection already open; Trans
TCP 44 52842 → 21 [ACK] Seq=78 Ack=358 Win=326912 Len=0
TCP 44 52849 → 52848 [FIN, ACK] Seq=66 Ack=1 Win=2161152
TCP 44 52848 → 52849 [ACK] Seq=1 Ack=67 Win=2161152 Len=
TCP 44 52848 → 52849 [FIN, ACK] Seq=1 Ack=67 Win=2161152
TCP 44 52849 → 52848 [ACK] Seq=67 Ack=2 Win=2161152 Len=

```

Fig. 5: Trecho da captura

Analisando o que foi obtido, os IPs do cliente e servidor são os mesmos, visto que foi utilizada a rede local para hospedar o servidor durante os testes. A diferença que é observada ao longo das mensagens são as portas de saída e chegada de cada uma delas.

Para demonstrar o bom funcionamento do cliente, pode ser observada a criação da conexão passiva e a transmissão de dados ao servidor para ser realizado um "store". Inicialmente, são trocadas mensagens, de maneira alternada, do tipo FTP e TCP entre os dois para estabelecer a conexão passiva, a qual é confirmada com a criação do socket que é percebida por três trocas TCP entre Servidor e Cliente (*HandShaking*).

Partindo disso, os dados que querem ser salvos são transmitidos ao servidor pelo caminho de dados e podem ser observados a baixo:

```
02 00 00 00 45 00 00 69 cf 9f 40 00 00 06 00 00 ....E..i..@:....
7f 00 00 01 7f 00 00 01 ce 71 ce 70 30 5c 5a 1f .....q.p0\Z
31 17 c8 b8 50 18 20 fa 9e a9 00 00 55 6d 20 61 1...p.....Um a
72 71 75 69 76 6f 20 64 65 20 74 65 78 74 6f 20 rquivo d e texto
70 61 72 61 20 74 65 73 74 65 20 64 65 20 75 70 para tes te de up
6c 6f 61 64 0a 0a 4d 61 69 73 20 6c 69 6e 68 61 load..Ma is linha
73 20 70 61 72 61 20 74 65 73 74 61 72 s para t estar
```

Fig. 6: Dados para serem salvos.

Nesta mesma parte, pode ser observado como o arquivo do tipo texto que foi transmitido foi convertido para uma representação Hexadecimal, que na prática foi transmitida de forma binária, o que também representa o encapsulamento dos dados transmitidos.

No restante das linhas obtidas, a conexão é retratada como um sucesso e o socket do caminho de dados é encerrado, finalizando todas as operações propostas pelo cliente. Sendo assim, o funcionamento do cliente é como o esperado.

Para demonstrar o funcionamento, o grupo realizou a gravação de um vídeo de apresentação do projeto, o qual está disponível no link: <https://clipchamp.com/watch/nFDDmon84ZO>.

Da mesma forma o código pode ser encontrado no **GitHub** em: <https://github.com/TheMasterGame0/ImplementacaoFTP>.

IV. CONCLUSÕES

Baseado no que foi observado ao longo do estudo, é possível concluir que é possível estabelecer uma conexão entre um cliente simples e um servidor capaz de cumprir com seus objetivos básicos. O caminho e comportamento dos dados trocados também podem ser rastreados e analisados para fins didáticos com as ferramentas certas.

REFERÊNCIAS

- [1] tkinter — Python interface to Tcl/Tk, <https://docs.python.org/3/library/tkinter.html>
- [2] socket — Low-level networking interface, <https://docs.python.org/3/library/socket.html>
- [3] J. Postel, J. Reynolds. *FILE TRANSFER PROTOCOL*, <https://www.rfc-editor.org/rfc/rfc959>
- [4] J. Klensin, A. Hoenes. *FTP Command and Extension Registry*, <https://www.rfc-editor.org/rfc/rfc5797>
- [5] Wireshark User's Guide, https://www.wireshark.org/docs/wsug_html_chunked/