Compartilhamento de arquivos por FTP

EzShare

Gabriel Henrique do Nascimento Neres - 221029140
Guilherme Nonato da Silva - 221002020

***Resumo*— O presente relatório visa**

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

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;
  - 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.

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 todos 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 inciar 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.

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

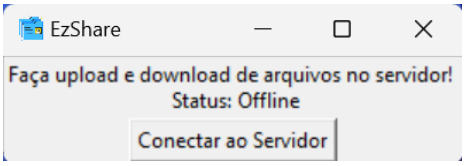

Fig. 1: Tela com o servidor desconectado.

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.

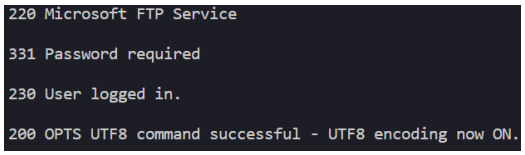

Fig. 2: Respostas do servidor.

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.

C. *WireShark*

IV. Conclusões

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