UFABC

UNIVERSIDADE FEDERAL DO ABC

Sistemas Distribuídos

Projeto de Programação – Napster com Sockets em Python

Pedro Victor Marcelino Jordão Motta RA: 11201921599

Santo André
2023

Screencast

Segue o link do screencast disponível no youtube e Google Drive:

- Youtube: https://youtu.be/q1fI2q_P2aE
- Google Drive: https://drive.google.com/file/d/16mdKQT9_plRmEj9aGPzLluWDwWZ5iJB3/view?usp=sharing

Projeto

Seguindo as instruções propostas no documento, o sistema foi dividido em um server.py e cliente.py. Em ambos foi utilizado estruturas de JSON para lidar com informações pela facilidade de manipulação e por ser uma estrutura amplamente utilizada. A explicação em alto nível das funções implementadas em cada programa será feita a seguir, além da explicação e exemplos das estruturas utilizadas:

cliente.py

- Criação da classe Peer e a função init inicializa todos os parâmetros do Peer, capturando porta, pasta do peer e estabelecendo conexão com servidor, lembrando que essa conexão ainda não foi aprovada e processada pelo servidor.

```
7   class Peer:
8       # Inicializaçao dos parametros
9       def __init__(self):
10          self.arquivos = []
11          self.serverPort = 1099
12          self.address_server = '127.0.0.1'
13          self.inicializar_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14          self.inicializar_socket.connect((self.address_server, self.serverPort))
15          self.clientPort = self.inicializar_socket.getsockname()[1]
16          self.pasta_peer = os.path.join(os.getcwd(), "peers")
17
18          if not os.path.isdir(self.pasta_peer):
19              os.mkdir(self.pasta_peer)
20
21          self.pasta_peer = os.path.join(self.pasta_peer, str(self.clientPort))
22          os.mkdir(self.pasta_peer)
23          print("Pasta do cliente: " + self.pasta_peer)
24
25
```

Imagem 1.

- A função setup_cliente (linha 28) é responsável por estabelecer a conexão de fato com o servidor e disponibilizar os arquivos.

```
28  def setup_cliente(self):
29      print("O seu nome na rede é: " + str(self.clientPort))
30      print("Está conectando no servidor: " + str(self.address_server) + ":" + str(self.serverPort))
31
32      while True:
33          print("----------------")
34          print("Os arquivos que disponibilizados são:")
35          print(os.listdir(self.pasta_peer))
36          print("1. Iniciar Conexão")
37          print("2. Atualizar os arquivos")
38          print("----------------")
39
40          entrada = input()
41          if entrada == "1":
42              break
43
44      self.arquivos = os.listdir(self.pasta_peer)
45      arquivos_formatados = ['"{}"'.format(item) for item in self.arquivos]
46      arquivos_concatenados = ','.join(arquivos_formatados)
47      message = '''{{"method": "JOIN", "data": {{"files": [{files}]}}}}'''.format(files=arquivos_concatenados)
48      self.inicializar_socket.sendall(message.encode())
49
50      if self.inicializar_socket.recv(1024).decode() == "JOIN_OK":
51          print(f"Sou peer {self.address_server}:{self.clientPort} com arquivos:[{arquivos_concatenados}]")
52      else:
53          print("Falha na conexão!")
54
```

Imagem 2.

- A função comandos_cliente (linha 107) é responsável por cuidar da requisição do cliente ao servidor, dentro dessa função recebemos a ação do cliente e tratamos de forma separada, não tratamos possíveis erros de escrita:

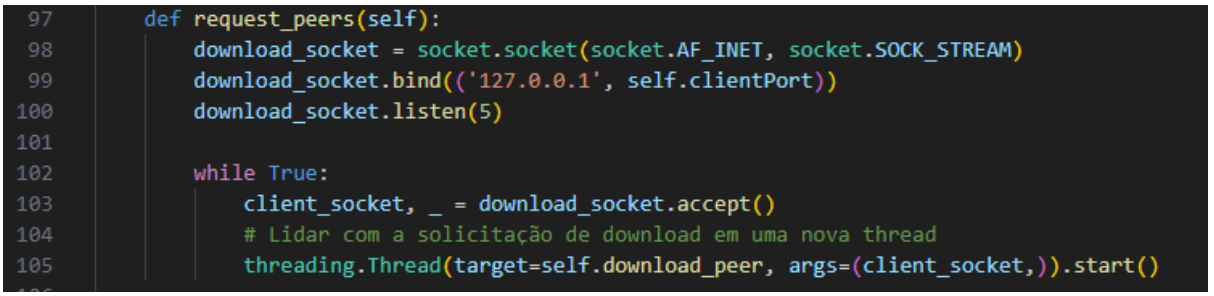
```
107 def comandos_cliente(self, un):
108     print("Insira algum dos comandos aceitos: \n - SEARCH \n - DOWNLOAD \n")
109     entrada = input().upper()
110
111     if entrada == "SEARCH":
112         entrada = input("Insira o arquivo que deseja buscar:\n")
113         message = '''{"method":"SEARCH","data":{"query":"%s"}}''' % entrada
114
115         # Envia json com os arquivos que é necessário buscar
116         self.inicializar_socket.sendall(message.encode())
117         print(f"Peers com arquivo solicitado: {self.inicializar_socket.recv(1024).decode()} ")
118
119     elif entrada == "DOWNLOAD":
120         arquivo = input("Insira o nome do arquivo que deseja baixar:\n")
121         endereco_peer_arquivo = int(input("Insira a porta do peer a partir do qual deseja baixar o arquivo:\n"))
122         message = '''{{"method": "DOWNLOAD_REQUEST", "data": {{"filename": "{}", "port": {}}}}}'''.format(arquivo, endereco_peer_arquivo)
```

Imagem 3.

Nessa etapa recebemos a entrada, que definirá cada ação do peer, como supracitado foi utilizado a estrutura de dados JSON, visto na linha 113,122 para enviar a outros peers e ao servidor.

A tratativa DOWNLOAD é feita em duas etapas, a primeira é fazer uma requisição ao servidor para realmente confirmar se o peer solicitado contém o arquivo, se o servidor confirmar que contém, é estabelecido a conexão através da função abaixo e é feito recebimento do como demonstrado na imagem 6.

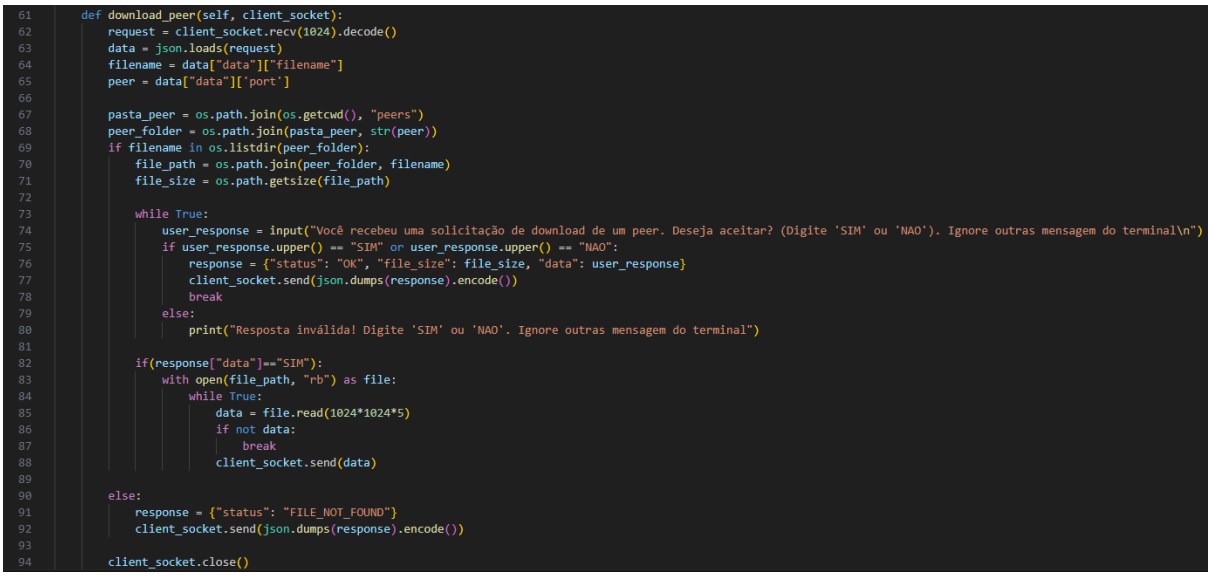
- A função request_peers (linha 97) é responsável por abrir um socket para esperar a conexão com outros peers que solicitarem um arquivo específico e estabelecer a conexão, a cada solicitação é criado uma nova thread



```
97      def request_peers(self):
98          download_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
99          download_socket.bind(('127.0.0.1', self.clientPort))
100         download_socket.listen(5)
101
102         while True:
103             client_socket, _ = download_socket.accept()
104             # Lidar com a solicitação de download em uma nova thread
105             threading.Thread(target=self.download_peer, args=(client_socket,)).start()
```

Imagem 4.

- A função download_peer é responsável por receber a requisição de outro peer, verificar se realmente contém aquele arquivo, se contém é perguntando ao peer se ele quer compartilhar, e manda uma mensagem falando sobre o tamanho do arquivo e sua resposta, que servirá como parâmetro para verificar se foi recebido o arquivo de forma integral, nessa função pode enviar o arquivo e encerra a conexão ou não dependendo da resposta do peer que contém o arquivo.



```
61  def download_peer(self, client_socket):
62      request = client_socket.recv(1024).decode()
63      data = json.loads(request)
64      filename = data["data"]["filename"]
65      peer = data["data"]['port']
66
67      pasta_peer = os.path.join(os.getcwd(), "peers")
68      peer_folder = os.path.join(pasta_peer, str(peer))
69      if filename in os.listdir(peer_folder):
70          file_path = os.path.join(peer_folder, filename)
71          file_size = os.path.getsize(file_path)
72
73          while True:
74              user_response = input("Você recebeu uma solicitação de download de um peer. Deseja aceitar? (Digite 'SIM' ou 'NAO'). Ignore outras mensagem do terminal\n")
75              if user_response.upper() == "SIM" or user_response.upper() == "NAO":
76                  response = {"status": "OK", "file_size": file_size, "data": user_response}
77                  client_socket.send(json.dumps(response).encode())
78                  break
79              else:
80                  print("Resposta inválida! Digite 'SIM' ou 'NAO'. Ignore outras mensagem do terminal")
81
82          if(response["data"]=="SIM"):
83              with open(file_path, "rb") as file:
84                  while True:
85                      data = file.read(1024*1024*5)
86                      if not data:
87                          break
88                      client_socket.send(data)
89
90      else:
91          response = {"status": "FILE_NOT_FOUND"}
92          client_socket.send(json.dumps(response).encode())
93
94      client_socket.close()
```

Imagem 5.

Segue a tratativa de recebimento do arquivo, recebendo uma mensagem de outro peer, que dirá se tem o arquivo e o tamanho dele, para ser um parâmetro de verificação se recebeu o arquivo de forma integral e update.

```
138             # Recebe resposta do peer para verificar se ele contém realmente o arquivo e se ele quer compartilhar
139             response = peer_socket.recv(1024).decode()
140             response_data = json.loads(response)
141             if response_data["status"] == "OK" and response_data['data']=="SIM":
142                 file_size = response_data["file_size"]
143                 received_data = b""
144                 bytes_received = 0
145
146                 # Recebendo os dados e verificando o tamanho do arquivo
147                 while bytes_received < file_size:
148                     data = peer_socket.recv(1024*1024*5)
149                     received_data += data
150                     bytes_received += len(data)
151
152                  # Gravando os dados recebidos
153                 caminho_arquivo = os.path.join(self.pasta_peer, arquivo)
154                 with open(caminho_arquivo, "wb") as file:
155                     file.write(received_data)
156
157                 print(f"Arquivo {arquivo} baixado com sucesso na pasta {caminho_arquivo}")
158
159                 # Lista os arquivos no peer
160                 self.arquivos = os.listdir(self.pasta_peer)
161                 arquivos_formatados = ['"{}"'.format(item) for item in self.arquivos]
162                 arquivos_concatenados = ','.join(arquivos_formatados)
163                 message = '''{{"method": "UPDATE", "data": {{"files": [{files}]}}}}'''.format(files=arquivos_concatenados)
164
165                 # Envia json com os arquivos do peer ao servidor
166                 self.inicializar_socket.sendall(message.encode())
167                 resposta_uptade = self.inicializar_socket.recv(1024).decode()
168                 if(resposta_uptade=="UPDATE_OK"):
169                     pass
```

Imagem 6.

server.py

- Foi feita a definição da classe Server, junto com a criação de um função com algum parâmetros e a função config (linha 93), é responsável por inicializar o socket

```
6  class Server:
7      def __init__(self):
8          self.listOfPeers = dict()
9          self.port = 1099
10
```

```
93  def config(self):
94      self.inicializar_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
95      self.inicializar_socket.bind(('127.0.0.1', self.port))
96
```

Imagem 7.

- A função update (linha 13), é responsável por o update após ser feito o download de um arquivo de outro peer

```
11
12      # Função uptade, feita após ser baixado um arquivo
13      def uptade(self, peer, addr, data):
14          if 'data' in data and 'files' in data['data']:
15              files = data['data']['files']
16              self.listOfPeers[addr[1]] = files
17              peer.sendall(b'UPDATE_OK')
18          else:
19              print("Estrutura do JSON errada.")
20
```

Imagem 8.

- Função search (linha 22), é responsável por procurar quais peers contém o arquivo desejado

```
21      # Função responsável por procurar quais peers contém o arquivo desejado
22      def search(self, peer, data):
23          search = data['data']['query']
24          resultado = []
25          for peers, values in self.listOfPeers.items():
26              if search in values:
27                  resultado.append(str(f"127.0.0.1:{peers}"))
28          print(f"Peer 127.0.0.1:{peer.getpeername()[1]}  solicitou arquivo {search}.")
29          peer.sendall(str(resultado).encode())
30
```

Imagem 9.

- Função search_download (linha 28), é responsável por verificar se o peer contém realmente o arquivo, antes de estabelecer conexão. É acionada quando um peer faz uma requisição DOWNLOAD.

```
33      def search_download(self, peer, data):
34          file = data['data']['filename']
35          port = str(data['data']['port'])
36
37          for peers, values in self.listOfPeers.items():
38              if file in values and str(port) in str(peers):
39                  peer.sendall(str("OK").encode())
40                  break
41          else:
42              peer.sendall(str("Não encontrado no banco de dados do servidor").encode())
43
```

Imagem 10.

- Função join (linha 46), é responsável por realizar o JOIN na estrutura do servidor

```
46      def join(self, peer, addr, data):
47          if 'data' in data and 'files' in data['data']:
48              files = data['data']['files']
49
50              self.listOfPeers[addr[1]] = files
51              peer.send(b'JOIN_OK')
52              print(f"Peer 127.0.0.1:{addr[1]} adicionado com arquivos {files}")
53          else:
54              print("JSON inválido")
55
```

Imagem 11.

- Função receiveData (linha 57), é responsável por receber todos os dados do servidor

```
57      def receiveData(self, peer):
58          data = ""
59          while True:
60              try:
61                  received = peer.recv(1024)
62                  data += received.decode()
63              except:
64                  return data
```

Imagem 12.

- Função serverUP, que é responsável por receber todas as requisições e tratar cada requisição em um Treadh diferente.

```
68  def serverUp(self, peer, addr):
69      peer.settimeout(3)
70      while True:
71          response = self.receiveData(peer)
72
73          # Trata cada ação em uma Thread diferente
74          try:
75              if response != "":
76                  data = json.loads(response)
77
78                  if data['method'] == "JOIN":
79                      threading.Thread(target=self.join, args=(peer, addr, data)).start()
80
81                  if data['method'] == "SEARCH":
82                      threading.Thread(target=self.search, args=(peer, data)).start()
83
84                  if data['method'] == "UPDATE":
85                      threading.Thread(target=self.uptade, args=(peer, addr, data)).start()
86
87                  if data['method'] == "DOWNLOAD_REQUEST":
88                      threading.Thread(target=self.search_download, args=(peer, data)).start()
89
90          except json.JSONDecodeError as e:
91              print("Erro no datagrama do JSON:", str(e))
```

Imagem 13.

- Por fim a função run, que chamará o config() e aceita as conexões dos peers e o processo de inicialização da classe e do server:

```
97      def run(self):
98          self.config()
99          print(f"Servidor rodando no endereço 127.0.0.1:{self.port}")
100         self.inicializar_socket.listen(1)
101
102         while True:
103             peer, addr = self.inicializar_socket.accept()
104             threading.Thread(target=self.serverUp, args=(peer, addr)).start()
105
106
107
108 server = Server()
109 server.run()
```

Imagem 14.

Threads

Foi utilizado Threads no cliente.py:

- Na linha 105, dentro da função request_peers, pois quando uma conexão é aceita com sucesso, é necessário permitir que múltiplos peers se conectem e realizem downloads simultaneamente sem bloquear o processo principal, por isso que é criado uma Thread.
- Na linha 184, dentro da função comandos_clientes utiliza threads para permitir que o programa execute várias tarefas simultaneamente
- Nas linhas 193 e 194, permite que o programa atenda a solicitações de outros peers e faça solicitação ao servidor ao mesmo tempo.

Foi utilizado Threads no server.py:

- Na linha 79,82,85,88 para que cada requisição seja tratada em uma thread separada, permitindo que as funções join, search, update e search_download sejam chamadas em threads distintas.
- Na linha 104 foi usado para a função serverUp para permitir que o servidor possa receber várias conexões de peers simultaneamente.

Transferência de arquivos

A transferência é somente no arquivo cliente.py, para o peer que contém o arquivo foi feito da linha 82 até 88, na qual ele transfere apenas se concordar em compartilhar, a velocidade de transferência foi escolhida de forma arbitrária.

Já pelo peer que quer receber o arquivo ele recebe na mesma velocidade que o outro peer envia, mas temos uma etapa de validação, na qual temos o tamanho do arquivo e a quantidade de bytes transferidos, a ação só irá finalizar, caso receba todos os bytes do arquivo original, esse processo é feito nas linhas 147 até 150.

Para arquivos maiores, podemos aumentar essa velocidade de transferência.

Código

Google Drive:
https://drive.google.com/file/d/1XS2GlPSHv-4VgTXamchzZ8DnWiKEAZwQ/view?usp=sharing

GitsGitHub:
https://gist.github.com/pedrovmjm/3c758dc6a794028a7363ab0c061d23e5