Sistemas Distribuídos - Projeto 1

Daniel José Gomes de Lima
RA 11201921053

Sistemas Distribuídos DA2 2023.2

O link do vídeo demonstração do relatório abaixo pode ser acessado através do Youtube através do **link**.

As funcionalidades do **servidor** são, JOIN, SEARCH e UPDATE:

JOIN

Quando um cliente envia uma solicitação de join (método 'JOIN'), a função **doJoin** é chamada. Essa função recebe a conexão do cliente (*c*), o endereço do cliente (*addr*) e os dados da solicitação (data).

A lista de arquivos fornecida pelo cliente é obtida a partir de *data['data']['filesPath']*. Os arquivos são adicionados à lista de pares do servidor usando o endereço do cliente como chave e os arquivos como valores. Uma mensagem "Peer [IP]:[Port] adicionado com os arquivos [arquivos]" é exibida no console. Uma resposta "JOIN_OK" é enviada de volta ao cliente usando *c.send(b'JOIN_OK')*. A função **doJoin** é executada em uma thread separada para

```
def doJoin(c, addr, data):
    ## Executa a lógica de JOIN
    files = data['data']['filesPath']
    self.listOfPeers[addr[1]] = os.listdir(files)
    print("Peer {}:{} adicionado com os arquivos {}".format(addr[0], addr[1], os.listdir(files)))
    c.send(b'JOIN_OK')
```

que o servidor possa continuar aceitando outras conexões.

SEARCH

Quando um cliente envia uma solicitação de busca (método 'SEARCH'), a função **doSearch** é chamada. Essa função recebe a conexão do cliente (*c*), o endereço do cliente (*addr*) e os dados da solicitação (*data*). A consulta de busca é obtida a partir de *data['data']['query']*.

O servidor percorre a lista de pares e verifica se os arquivos correspondem à consulta de busca. Os pares que possuem os arquivos correspondentes são adicionados a uma lista de resultados. A lista de resultados é convertida em uma string é enviada de volta ao cliente usando *c.sendall(str(result).encode())*.

```
def doSearch(c, addr, data):
    ## Busca o arquivo na lista de peers pelo nome e reporta os peers que correspondem a pesquisa
    search = data['data']['query']
    result = []
    print(search)
    print("Peer {}:{} solicitou arquivo {}".format(addr[0], addr[1], search))
    for key, values in self.listOfPeers.items():
        if search in values:
            result.append(key)
    c.sendall(str(result).encode())
```

UPDATE

Quando um cliente envia uma solicitação de atualização (método 'UPDATE'), a função **doUpdate** é chamada. Essa função recebe a conexão do cliente (*c*), o endereço do cliente (*addr*) e os dados da solicitação (*data*). Os arquivos enviados pelo cliente são obtidos a partir de *data['data']['files']*. Os arquivos são atualizados na lista de pares do servidor usando o endereço do cliente como chave. Uma mensagem com a lista atualizada de pares é exibida no console. Uma resposta "*UPDATE_OK*" é enviada de volta ao cliente usando *c.sendall(b'UPDATE_OK')*.

```
def doUpdate(c, addr, data):
    ## Recebe, executa o UPDATE e returna a resposta ao peer
    if 'data' in data and 'files' in data['data']:
        files = data['data']['files']
        self.listOfPeers[addr[1]] = files
        c.sendall(b'UPDATE_OK')
    else:
        print("Json Inválido")
```

Threads

No servidor, o uso de threads é fundamental para permitir que múltiplos clientes se conectem e sejam atendidos simultaneamente. Isso garante que o servidor seja capaz de lidar com várias solicitações concorrentes sem bloquear a execução de outras tarefas.

Quando o servidor aceita uma nova conexão de cliente, uma nova thread é criada para lidar com essa conexão específica. Essa abordagem permite que várias conexões sejam tratadas em paralelo, sem interferir umas com as outras.

Ao utilizar threads no servidor, cada conexão é tratada de forma independente, permitindo que o servidor atenda a solicitações de diferentes clientes simultaneamente. Isso resulta em um servidor mais responsivo e escalável.

As funcionalidades do **peer** são, JOIN, SEARCH e DOWNLOAD:

JOIN

A função **callJoin** é responsável por enviar a solicitação de join para o servidor. Ela cria uma mensagem JSON com o método "JOIN" e os dados contendo o caminho dos arquivos do peer. A mensagem é enviada para o servidor através do socket usando *socket.sendall(message.encode())*. O peer aguarda a resposta do servidor usando *self.socketServer.recv(1024).decode()*. Se a resposta for "JOIN_OK", ele exibe uma mensagem no console indicando que foi adicionado ao servidor junto com os arquivos que ele compartilha.

```python
def callJoin(self):
    ## Pega o Path dos arquivos na pasta e envia ao server
    pastaPeer = os.getcwd() + "/p2p/" + str(self.socketServer.getsockname()[1])
    pathArquivos = pastaPeer
    message = json.dumps({"method": "JOIN", "data": {"filesPath": pathArquivos}})
    self.socketServer.sendall(message.encode())

    ## Ao receber o JOIN_OK do server monta a mensagem de JOIN
    if self.socketServer.recv(1024).decode() == "JOIN_OK":
        arquivos = os.listdir(pastaPeer)
        arquivosFormatados = ['"{}"'.format(item) for item in arquivos]
        arquivosConcatenados = ','.join(arquivosFormatados)
        print(f"Sou o Peer {self.addressServer}:{self.clientPort} com arquivos: {arquivosConcatenados}")
    else:
        raise Exception("Falha na Conexao! Programa finalizado.")
```

SEARCH

A função **callSearch** é responsável por enviar uma solicitação de busca ao servidor. Ela cria uma mensagem JSON com o método "SEARCH" e a consulta de busca. A mensagem é enviada para o servidor através do socket usando *socket.sendall(message.encode())*. O Peer aguarda a resposta do servidor usando *serverSocket.recv(1024).decode()* e exibe no console a lista de peers que possuem o arquivo solicitado.

```python
def callSearch(self, serverSocket):
    ## Monta e realiza a chamada de SEARCH ao servidor
    try:
        entrada = input("Insira o arquivo que deseja buscar:\n")
        message = '''{"method":"SEARCH","data":{"query":"%s"}}''' % entrada
        serverSocket.sendall(message.encode())
        print(f"Peers com arquivo solicitado: {serverSocket.recv(1024).decode()} ")
        return entrada
    except Exception as e:
        print(f"Erro ao buscar arquivo: {str(e)}")
```

DOWNLOAD

A função **callDownload** é responsável por iniciar o processo de download de um arquivo de um peer. O peer solicita o nome do arquivo e o endereço do peer a partir do qual deseja baixar o arquivo. Ele estabelece uma conexão com o peer usando um novo socket (*peerSocket*) e envia uma solicitação de download com o nome do arquivo e o endereço do peer. O peer recebe a resposta do peer remoto usando *peerSocket.recv(1024).decode()*. Se o status for "OK", ele inicia o processo de download.

O peer recebe os dados do arquivo em partes usando *peerSocket.recv(1024 * 1024 * 1024)* e os escreve em um novo arquivo local. Após o download concluído com sucesso, o peer exibe uma mensagem no console indicando o sucesso do download e a localização do arquivo baixado.

```python
def callDownload(self, serverSocket):
    ## Monta a Requisição de download do arquivo e envia ao outro peer
    try:
        arquivo = input("Insira o nome do arquivo que deseja baixar:\n")
        endereco_peer_arquivo = int(input("Insira a porta do peer a partir do qual deseja baixar o arquivo:\n"))

        peerSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        peerSocket.connect((self.addressServer, endereco_peer_arquivo))

        message = '''{{"method": "DOWNLOAD_REQUEST", "data": {{"filename": "{}", "port": {}}}}}'''.format(arquivo,
                                                                                                          endereco_peer_arquivo)
        peerSocket.sendall(message.encode())

        response = peerSocket.recv(1024).decode()
        data = json.loads(response)

        ## Avalia a resposta do outro peer à solicitação de download
        if data["status"] == "OK":
            fileSize = data["fileSize"]
            data = b""
            response = peerSocket.recv(1024).decode()
            data = json.loads(response)
            ## Executa o Download
            if data["status"] == "downloadAccepted":
                data = peerSocket.recv(fileSize)
                filePath = os.path.join(self.folderPeer, arquivo)
                with open(filePath, "wb") as file:
                    file.write(data)
                print(""Arquivo " + arquivo + "baixado com sucesso na pasta " + filePath)

                ## Envia a pensagem de UPDATE ao server
                arquivos = os.listdir(self.folderPeer)
                arquivos_formatados = ['"{}"'.format(item) for item in arquivos]
                arquivos_concatenados = ','.join(arquivos_formatados)
                message = '''{{"method": "UPDATE", "data": {{"files": [{files}]}}}}'''.format(
                    files=arquivos_concatenados)
                serverSocket.sendall(message.encode())
                serverSocket.recv(1024).decode()
            else:
                print("Download Rejeitado")
        else:
            print("Arquivo não encontrado.")
    except Exception as e:
        print(f"Erro ao processar download: verifique se as informações enviadas estão corretas!")
```

UPLOAD

A função callUpload é responsável por tratar as solicitações de upload recebidas de outros peers. Ela é chamada em uma nova thread quando uma solicitação de upload é recebida pelo peer. Dentro da função callUpload, o peer lida com a solicitação recebida do peer remoto. Ele extrai o nome do arquivo e a porta do peer remoto a partir dos dados da solicitação. Em seguida, o peer verifica se o arquivo solicitado existe em sua própria pasta. Se o arquivo existir, ele envia uma resposta ao peer remoto informando que o arquivo foi encontrado e envia os dados do arquivo em partes para o peer remoto.

Essa função permite que o peer compartilhe arquivos com outros pares e lide com as solicitações de download recebidas de forma assíncrona, sem interromper a execução principal do peer.

```python
def callUpload(self, socket):
    ## Monta o UPLOAD do arquivo
    try:
        request = socket.recv(1024).decode()
        data = json.loads(request)
        filename = data["data"]["filename"]
        peer = data["data"]["port"]

        folder = os.path.join(os.path.join(os.getcwd(), "p2p"), str(peer))
        if filename in os.listdir(folder):
            path = os.path.join(folder, filename)
            size = os.path.getsize(path)
            response = {"status": "OK", "fileSize": size}
            socket.send(json.dumps(response).encode())
            self.isDownloading = True
            ## Executa a lógica de aprovar ou não o download de um peer
            while True:
                userInput = input("Gostaria de Aprovar o download? (SIM/NAO): ")
                if userInput == "SIM":
                    response = {"status": "downloadAccepted"}
                    socket.send(json.dumps(response).encode())
                    size = os.path.getsize(path)
                    with open(path, "rb") as file:
                        while True:
                            data = file.read()
                            if not data:
                                break
                            socket.sendall(data)
                    break
                elif userInput == "NAO":
                    response = {"status": "downloadRejected"}
                    socket.send(json.dumps(response).encode())
                    break
            self.isDownloading = False
        else:
            response = {"status": "fileNotFound"}
            socket.send(json.dumps(response).encode())
    except Exception as e:
        print(f"Erro ao processar upload: {str(e)}")
    finally:
        socket.close()
        threading.Thread(target=self.comandosCliente, args=(self.socketServer,)).start()
```

THREADS

Nos peers, o uso de threads também é importante para permitir a execução simultânea de diferentes tarefas e a comunicação eficiente com outros pares.Por exemplo, os peers podem usar threads para lidar com a entrada do usuário e comandos, para que possam receber comandos enquanto aguardam outras operações, como solicitações de download ou busca por arquivos.

Além disso, os peers podem criar threads para tratar solicitações de upload recebidas de outros pares. Essa abordagem permite que o peer atue como servidor para compartilhar arquivos com outros pares, sem interromper a execução principal do peer. O uso de threads nos peers permite que eles executem várias tarefas simultaneamente, garantindo que possam lidar com as operações de compartilhamento de arquivos, busca e download de forma eficiente e sem bloqueios.

Comunicação utilizando JSON

A comunicação no servidor P2P utiliza o formato JSON para trocar informações entre o servidor e os clientes. O servidor recebe as solicitações dos clientes em formato JSON e as decodifica. Ele identifica o método da solicitação (JOIN, SEARCH ou UPDATE) e encaminha a solicitação para a função correspondente. Os dados relevantes da solicitação são extraídos do JSON e processados pela função apropriada. Em seguida, o servidor cria uma resposta em JSON com os dados apropriados e a envia de volta ao cliente.

Transferência de grandes arquivos

O tamanho dos arquivos não é especificamente tratado, mas há uma abordagem em que a vazão de dados é definida para um valor alto de 1024 * 1024 * 1024 bytes por envio de mensagem. Essa abordagem permite o envio de grandes quantidades de dados de uma vez, o que pode ser útil para transferir arquivos de tamanho considerável.

Referências

https://realpython.com/intro-to-python-threading/
https://imasters.com.br/back-end/threads-em-python
https://stackoverflow.com/questions/25495593/when-using-socket-recv-the-program-hangs-up-when-there-are-no-more-bytes-to-r