

Professor: Marcos Fábio Jardini

Nome: Bruno Cesar de Almeida Ribeiro - RA 210570

Nome: Caio Lucas Dias - RA 190120

Nome: Gustavo Moreira de Mello - RA 180525

Nome: Kaetano Cesar Apolinario Rodrigues - RA 190157

Nome: Leonardo José Ferreira Corrêa - RA 210726

Github do projeto: <https://github.com/GustavoMMello01/HTTP-Server>

Servidor HTTP

A transferência de dados na internet ocorre por protocolos de comunicação, um conjunto de regras estabelecidas para garantir a ordem, segurança e otimização das transmissões na rede. Dentre eles, o HTTP é o mais prevalente e, por consequência, o mais reconhecido. A sigla "HTTP" significa: "Protocolo de Transferência de Hipertexto". Esse protocolo facilita a transferência de textos, imagens e outros recursos.

A função principal do HTTP é transferir páginas e outros dados de um servidor para um navegador, permitindo que os usuários acessem e visualizem o conteúdo. É por isso que os URLs dos sites geralmente começam com "http" ou "https".

O processo de comunicação via HTTP geralmente ocorre da seguinte forma: um navegador (cliente) faz uma requisição a um servidor (muitas vezes o web server). Após processar essa requisição, o servidor HTTP responde, enviando os dados solicitados de volta ao navegador. Como mostra a imagem abaixo.

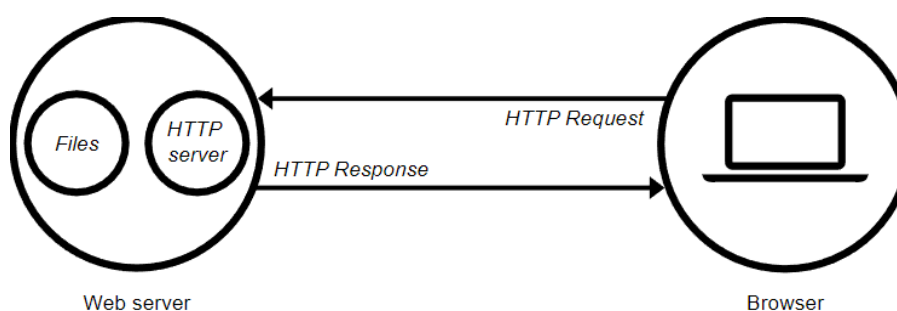


Imagem 1: **Servidor HTTP em nível mais básico.** Fonte: https://developer.mozilla.org/pt-BR/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server

Desafio de implementação de Servidor HTTP

Para a solução do desafio, o grupo desenvolveu um servidor HTTP simples utilizando a linguagem Python sem bibliotecas que pudessem facilitar as mensagens de status e implementação. O projeto foi estruturado em três arquivos principais na root: main.py, handlers.py e logger.py.

É no arquivo main.py que o servidor é configurado e executado. O grupo utilizou a biblioteca socket para criar um servidor que escuta no endereço "localhost" e na porta "8080". Com o auxílio da biblioteca threading, o grupo implementou a capacidade de manipular múltiplas conexões simultâneas. Quando uma nova conexão é estabelecida, uma nova thread é criada e a função handle_client é chamada para tratar da requisição.

```
def main():  
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    server_socket.bind((HOST, PORT))  
    server_socket.listen(MAX_CONNECTIONS)  
  
    print(f"Servidor HTTP rodando em http://{HOST}:{PORT}")  
  
    while True:  
        client_socket, client_addr = server_socket.accept()  
        client_socket.settimeout(TIMEOUT)  
        client_thread = threading.Thread(target=handle_client, args=(client_socket,))  
        client_thread.start()  
  
if __name__ == "__main__":  
    main()
```

Imagem 2: Configuração na main.py. Fonte: Autoria própria.

O socket é configurado nas três primeiras linhas como mostra a figura acima. De acordo com as variáveis escolhidas pelo usuário no começo do código:

```
# Configurações do servidor  
HOST = "localhost"  
PORT = 8080  
MAX_CONNECTIONS = 5  
TIMEOUT = 20
```

Imagem 3: Configuração do servidor. Fonte: Autoria própria.

No while, a conexão é aceita e configurada um tempo limite, para então ser criado uma nova thread para tratada a requisição do cliente. O código completo pode ser visto na imagem 4:

```
server.py > main
1  from handlers import handle_client
2  import socket
3  import threading
4
5  # Configurações do servidor
6  HOST = "localhost"
7  PORT = 8080
8  MAX_CONNECTIONS = 5
9  TIMEOUT = 20
10
11 def main():
12     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     server_socket.bind((HOST, PORT))
14     server_socket.listen(MAX_CONNECTIONS)
15
16     print(f"Servidor HTTP rodando em http://{HOST}:{PORT}")
17
18     while True:
19         client_socket, client_addr = server_socket.accept()
20         client_socket.settimeout(TIMEOUT)
21         client_thread = threading.Thread(target=handle_client, args=(client_socket,))
22         client_thread.start()
23
24 if __name__ == "__main__":
25     main()
26
```

Imagem 4: Main.py. Fonte: Autoria própria

O arquivo handlers.py, contém a lógica para lidar com diferentes métodos HTTP como GET, PUT, POST, DELETE e HEAD. A função handle_client analisa a requisição recebida, determina o tipo de método e processa de acordo como o método esperado:

- GET: O servidor tenta localizar e retornar o arquivo solicitado.
- PUT: Ele tenta criar ou atualizar um recurso. A implementação é mais uma simulação, pois atualiza o recurso com uma mensagem padrão.
- POST: Semelhante ao PUT, ele tenta criar um recurso.
- DELETE: O método foi incluído como exemplo e responde que a implementação não foi realizada.
- HEAD: Retorna apenas os cabeçalhos da resposta, sem o corpo.

Se o arquivo solicitado existe, ele é lido e seu tipo de mídia (MIME type) é determinado com base em sua extensão (como .html, .jpg, .mp4, etc.). A resposta é então formatada de acordo com o protocolo HTTP e enviada de volta ao cliente. Todas as requisições são registradas usando a função log_request.

Basicamente, primeiro é definido as informações básicas do servidor e inicializado as variáveis para a resposta:

```
def handle_client(client_socket):  
    server = "server/1.0"  
    content_type = ""  
    response_length = 0  
  
    client_addr = client_socket.getpeername()  
    request = client_socket.recv(1024).decode()  
  
    # Inicializa variáveis para método e caminho  
    method = ""  
    path = ""  
  
    # Analisa a requisição
```

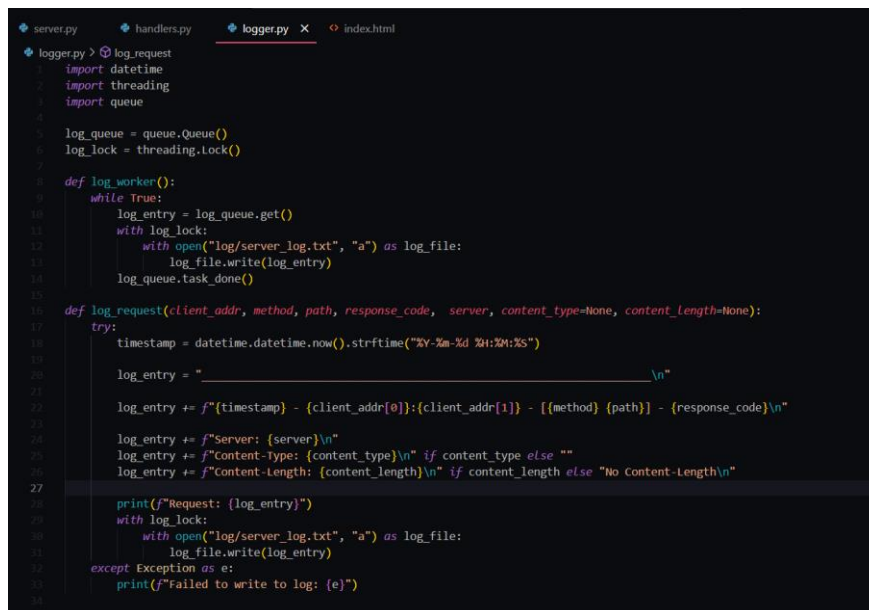
Imagem 5: Informações básicas do servidor. Fonte: Autoria própria

Caso a requisição não esteja vazia, ele faz a request, sendo que primeiro ele divide a requisição em linhas, para então pegar a primeira linha (onde está localizado o método e o caminho), finalmente verifica qual é o tipo de requisição (GET, POST, PUT, DELETE, HEAD), para então tratar e mandar as respostas corretas.

```
# Analisa a requisição  
if request:  
    request_lines = request.split("\n")  
    first_line = request_lines[0].split()  
    method = first_line[0]  
    path = first_line[1]  
  
    if method == "GET":  
        file_path = os.path.join(BASE_DIR, path[1:])  
  
        if os.path.exists(file_path) and os.path.isfile(file_path):  
            response_code = "200 OK"  
            with open(file_path, "rb") as file:  
                response_data = file.read()  
  
            if file_path.endswith(".html"): #verifica se o arquivo é um html  
                content_type = "text/html"  
            elif file_path.endswith(".jpg"): #verifica se o arquivo é uma imagem  
                content_type = "image/jpeg"  
            elif file_path.endswith(".mp4"): #verifica se o arquivo é um video  
                content_type = "video/mp4"  
            elif file_path.endswith(".mp3"): #verifica se o arquivo é um audio  
                content_type = "audio/mp3"  
  
            content_length = len(response_data) # armazena o tamanho do arquivo  
        else: #caso o arquivo não exista  
            response_code = "404 Not Found"  
            response_data = b"File not found"
```

Imagem 6: Exemplo do tratamento de um GET. Fonte: Autoria própria

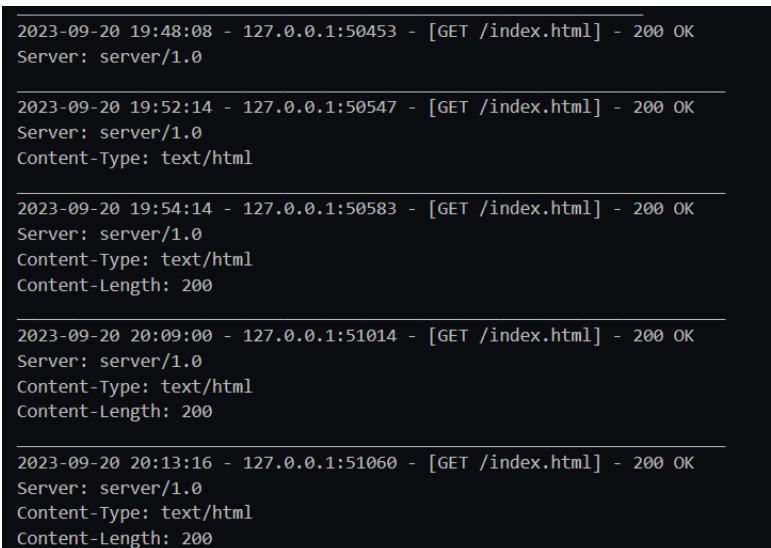
Finalmente, no arquivo logger.py, é implementado a funcionalidade de registro (logging). O grupo utilizou uma combinação de filas (usando a biblioteca queue) e bloqueio (usando a biblioteca threading) para garantir que as entradas de log sejam escritas no arquivo de log de maneira segura e sem conflitos. Cada entrada de log contém informações como timestamp, endereço IP do cliente, método HTTP usado, caminho solicitado, código de resposta, tipo de conteúdo e tamanho do conteúdo.



```
server.py handlers.py logger.py x index.html
logger.py > log_request
1 import datetime
2 import threading
3 import queue
4
5 log_queue = queue.Queue()
6 log_lock = threading.Lock()
7
8 def log_worker():
9     while True:
10         log_entry = log_queue.get()
11         with log_lock:
12             with open("log/server_log.txt", "a") as log_file:
13                 log_file.write(log_entry)
14         log_queue.task_done()
15
16 def log_request(client_addr, method, path, response_code, server, content_type=None, content_length=None):
17     try:
18         timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
19
20         log_entry = "
21
22         log_entry += f"{timestamp} - {client_addr[0]}:{client_addr[1]} - [{method}] {path} - {response_code}\n"
23
24         log_entry += f"Server: {server}\n"
25         log_entry += f"Content-Type: {content_type}\n" if content_type else ""
26         log_entry += f"Content-Length: {content_length}\n" if content_length else "No Content-Length\n"
27
28         print(f"Request: {log_entry}")
29         with log_lock:
30             with open("log/server_log.txt", "a") as log_file:
31                 log_file.write(log_entry)
32     except Exception as e:
33         print(f"Failed to write to log: {e}")
34
```

Imagem 7: Logger.py. Fonte: Autoria própria

Assim, o log é salvo na pasta log, de acordo com o seguinte formato:



```
2023-09-20 19:48:08 - 127.0.0.1:50453 - [GET /index.html] - 200 OK
Server: server/1.0

2023-09-20 19:52:14 - 127.0.0.1:50547 - [GET /index.html] - 200 OK
Server: server/1.0
Content-Type: text/html

2023-09-20 19:54:14 - 127.0.0.1:50583 - [GET /index.html] - 200 OK
Server: server/1.0
Content-Type: text/html
Content-Length: 200

2023-09-20 20:09:00 - 127.0.0.1:51014 - [GET /index.html] - 200 OK
Server: server/1.0
Content-Type: text/html
Content-Length: 200

2023-09-20 20:13:16 - 127.0.0.1:51060 - [GET /index.html] - 200 OK
Server: server/1.0
Content-Type: text/html
Content-Length: 200
```

Imagem 8: Log gerado. Fonte: Autoria própria

Em resumo, o grupo desenvolveu um servidor HTTP multi-threaded básico que pode receber e responder a várias requisições simultaneamente. Através da estruturação modular do código em diferentes arquivos, o grupo garantiu uma organização clara e uma manutenção facilitada do projeto.

Bibliografia

CHRISTOFOLLI, D. HTTP. Disponível em: <https://medium.com/@danielchristofolli/http-604ba899b755>. Acesso em: 26/09/2023.

MOZILLA DEVELOPER NETWORK. O que é um servidor web? Disponível em: https://developer.mozilla.org/pt-BR/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server. Acesso em: 26/09/2023.