

Universidade Federal do Espírito Santo  
Centro Tecnológico

Redes de Computadores  
Prof. Magnos Martinello

## **Projeto de Programação de Sockets**

### **Servidor Web**

#### **Objetivo**

Desenvolver um servidor web simples em Python (ou outra linguagem a seu critério) que pode processar uma requisição HTTP por vez e retornar o conteúdo de um arquivo HTML solicitado ou uma mensagem de erro “404 Not Found” caso o arquivo não exista.

#### **Tarefas**

**1. Criação de Sockets**

- Crie um socket TCP.
- Vincule o socket a um endereço IP e porta específicos.

**2. Processamento de Requisições HTTP**

- Aceite uma conexão de cliente.
- Receba a requisição HTTP do cliente.
- Analise a requisição para obter o nome do arquivo solicitado.

**3. Resposta HTTP**

- Verifique se o arquivo solicitado está presente no sistema de arquivos do servidor.
  - Se presente, leia o arquivo e crie uma resposta HTTP contendo o conteúdo do arquivo precedido por cabeçalhos apropriados.
  - Se ausente, crie uma mensagem de resposta HTTP “404 Not Found”.
- Envie a resposta HTTP de volta ao cliente.

**4. Testes**

- Coloque um arquivo HTML no diretório do servidor.
- Execute o servidor e acesse o arquivo através de um navegador web de um host diferente.
- Teste o acesso a um arquivo inexistente para verificar a mensagem “404 Not Found”.

## ===== Descrição do laboratório =====

### Lab 1: Laboratório de Servidor Web

Neste laboratório, você aprenderá os conceitos básicos de programação de sockets para conexões TCP em Python: como criar um socket, vinculá-lo a um endereço e porta específicos, além de enviar e receber um pacote HTTP. Você também aprenderá alguns conceitos básicos sobre o formato de cabeçalhos HTTP.

Você desenvolverá um servidor web que tratará uma requisição HTTP por vez. Seu servidor web deve aceitar e analisar a requisição HTTP, obter o arquivo solicitado do sistema de arquivos do servidor, criar uma mensagem de resposta HTTP consistindo do arquivo solicitado precedido por linhas de cabeçalho, e então enviar a resposta diretamente para o cliente. Se o arquivo solicitado não estiver presente no servidor, o servidor deve enviar uma mensagem HTTP "404 Not Found" de volta ao cliente.

### Código

Abaixo você encontrará o código esqueleto para o servidor web. Você deve completar o código esqueleto. Os locais onde você precisa preencher o código estão marcados com #Fill in start e #Fill in end.

Cada local pode exigir uma ou mais linhas de código.

### Executando o Servidor

Coloque um arquivo HTML (por exemplo, HelloWorld.html) no mesmo diretório onde o servidor está localizado. Execute o programa do servidor. Determine o endereço IP do host que está executando o servidor (por exemplo, 128.238.251.26). De outro host, abra um navegador e forneça a URL correspondente. Por exemplo:

<http://128.238.251.26:6789/HelloWorld.html>

'HelloWorld.html' é o nome do arquivo que você colocou no diretório do servidor. Note também o uso do número da porta após os dois pontos. Você precisa substituir este número de porta pelo número que você usou no código do servidor. No exemplo acima, usamos o número de porta 6789. O navegador deve então exibir o conteúdo de HelloWorld.html. Se você omitir ":6789", o navegador assumirá a porta 80 e você obterá a página web do servidor apenas se seu servidor estiver escutando na porta 80.

Em seguida, tente obter um arquivo que não está presente no servidor. Você deve receber uma mensagem "404 Not Found".

=====

```
# Importar módulo de socket

from socket import *
import sys # Para encerrar o programa

serverSocket = socket(AF_INET, SOCK_STREAM)

# Preparar um socket de servidor
# Fill in start
serverSocket.bind(('', 6789))
serverSocket.listen(1)
# Fill in end

while True:
    # Estabelecer a conexão
    print('Ready to serve...')
    connectionSocket, addr = serverSocket.accept() # Fill in start, Fill in end

    try:
        message = connectionSocket.recv(1024) # Fill in start, Fill in end
        filename = message.split()[1]
        f = open(filename[1:])
        outputdata = f.read() # Fill in start, Fill in end

        # Enviar uma linha de cabeçalho HTTP para o socket
        # Fill in start
        connectionSocket.send("HTTP/1.1 200 OK\r\n\r\n".encode())
        # Fill in end

        # Enviar o conteúdo do arquivo solicitado para o cliente
        for i in range(0, len(outputdata)):
            connectionSocket.send(outputdata[i].encode())
        connectionSocket.send("\r\n".encode())

        connectionSocket.close()
```

```

except IOError:
    # Enviar mensagem de resposta para arquivo não encontrado
    # Fill in start
    connectionSocket.send("HTTP/1.1 404 Not Found\r\n\r\n".encode())
    connectionSocket.send("<html><head></head><body><h1>404 Not
Found</h1></body></html>\r\n".encode())
    # Fill in end

    # Fechar o socket do cliente
    # Fill in start
    connectionSocket.close()
    # Fill in end

serverSocket.close()
sys.exit() # Termina o programa após enviar os dados correspondentes

```

## Explicação do Código

### 1. Importações e Configuração Inicial:

- Importamos os módulos necessários `socket` e `sys`.
- Criamos um socket TCP com `socket(AF_INET, SOCK_STREAM)`.

### 2. Preparação do Socket do Servidor:

- Vinculamos o socket a todas as interfaces de rede disponíveis no host e à porta 6789 usando `serverSocket.bind('', 6789)`.
- Colocamos o socket em modo de escuta para aceitar conexões com `serverSocket.listen(1)`.

### 3. Loop Principal:

- Entramos em um loop infinito onde o servidor estará sempre pronto para servir.
- Aceitamos uma conexão de cliente com `connectionSocket, addr = serverSocket.accept()`.

### 4. Tratamento de Requisições:

- Recebemos a mensagem do cliente com `message = connectionSocket.recv(1024)`.
- Extraímos o nome do arquivo solicitado da mensagem HTTP.
- Abrimos o arquivo solicitado e lemos seu conteúdo.

#### 5. Envio de Resposta HTTP:

- Se o arquivo for encontrado, enviamos uma linha de cabeçalho HTTP "200 OK" e o conteúdo do arquivo.
- Se o arquivo não for encontrado, capturamos a exceção `IOError` e enviamos uma mensagem de erro "404 Not Found".

#### 6. Fechamento da Conexão:

- Fechamos o socket do cliente após enviar a resposta.

#### 7. Encerramento do Servidor:

- Encerramos o servidor ao final do loop (isso é apenas simbólico aqui, já que o loop é infinito).

### Testando o Servidor

1. Coloque um arquivo HTML, por exemplo, `HelloWorld.html`, no mesmo diretório do servidor.
2. Execute o servidor.
3. No navegador, acesse `http://<seu_ip>:6789/HelloWorld.html`.
4. Verifique se o conteúdo do arquivo HTML é exibido.
5. Tente acessar um arquivo inexistente para verificar a mensagem "404 Not Found".



## Fase 2 – Servidor Multithread

1. **Servidor Multithread:** Atualmente, o servidor web trata apenas uma requisição HTTP por vez. Implemente um servidor multithread que seja capaz de **servir múltiplas requisições simultaneamente**. Utilizando threading, crie primeiro uma thread principal no qual seu servidor modificado escutará os clientes em uma porta fixa. Quando receber uma requisição de conexão TCP

de um cliente, ele configurará a conexão TCP através de outra porta e atenderá a requisição do cliente em um thread separado. Haverá uma conexão TCP separada em um thread separado para cada par de requisição/resposta.

### **Objetivo:**

Permitir que o servidor atenda múltiplas conexões simultaneamente com threads.

### **Tarefas:**

Importar o módulo **threading**:

```
from threading import Thread
```

**Encapsular o código de atendimento ao cliente em uma função:**

```
def handle_client(connectionSocket):  
    # Coloque aqui o código atual que está dentro do bloco  
    try/except
```

**Criar uma thread para cada conexão:**

Substitua o trecho que atualmente chama `connectionSocket, addr = serverSocket.accept()` por:

```
connectionSocket, addr = serverSocket.accept()  
client_thread = Thread(target=handle_client,  
args=(connectionSocket,))
```

```
1. client_thread.start()
```

**2. Cliente HTTP Personalizado:** Em vez de usar um navegador, escreva seu próprio cliente HTTP para testar seu servidor. Seu

cliente conectará ao servidor usando uma conexão TCP, enviará uma requisição HTTP ao servidor e exibirá a resposta do servidor como saída. Você pode assumir que a requisição HTTP enviada é do método GET. O cliente deve aceitar argumentos de linha de comando especificando o endereço IP ou nome do host do servidor, a porta em que o servidor está escutando e o caminho onde o objeto solicitado está armazenado no servidor. O seguinte é um formato de comando de entrada para executar o cliente:

```
client.py server_host server_port filename
```

---

## Esboço de Comparação de Desempenho: Servidor Web Multithread vs. I/O Multiplexado (select)

 **Objetivo**

Avaliar e comparar o desempenho de duas abordagens para um servidor web simples:

- **Multithread:** cada conexão cliente é tratada em uma thread separada.
  - **I/O multiplexado (select):** todas as conexões são tratadas no mesmo thread com monitoramento de múltiplos sockets via `select()`.
- 

## Configuração Experimental

### Ambiente

- Linguagem: Python 3.x
- Sistema Operacional: Linux/Unix-based
- Número de núcleos: X
- RAM disponível: Y GB
- Conexões simultâneas simuladas: de 1 até N (ex: 1, 10, 50, 100, 200)
- Arquivo HTML de teste: `HelloWorld.html` com tamanho fixo (ex: 5 KB)

### Ferramentas de carga

`ab` (ApacheBench):

```
ab -n 1000 -c 50 http://<host>:<port>/HelloWorld.html
```

-



- Alternativas: `wrk`, `hey`, `curl + script`, etc.

---

## Implementações a Comparar

### 1. Servidor Multithread


- Criação de uma thread por requisição.
- Simples de implementar.
- Pode ter sobrecarga com número elevado de conexões.






### 2. Servidor com `select`

- Um único loop com `select.select()` monitorando múltiplos sockets.
- Menor overhead de criação de threads.
- Mais complexo, mas mais escalável em cenários com muitas conexões ociosas.

---

## Métricas de Avaliação

Métrica	Descrição
 Tempo de resposta médio	Tempo médio de resposta por requisição

 Latência	Tempo para primeira resposta
 Throughput	Número de requisições atendidas por segundo
 Tempo total	Duração total para N requisições
 Uso de CPU	Carga de CPU sob diferentes cargas
 Uso de memória	Consumo de memória com N conexões simultâneas

---

## Plano de Testes

Nº de Conexões Simultâneas	Nº Total de Requisições
1	100
10	500

50

1000

100

2000

200

3000

Execute cada teste para os dois servidores e registre os resultados.

---



## Resultados Esperados (Hipóteses)

- **Multithread** tende a ter bom desempenho até ~100 conexões, mas pode degradar rapidamente em cargas muito altas por overhead de threads.
  - **select** deve ter melhor escalabilidade sob cargas maiores, com uso de CPU/memória mais estável.
  - Latência pode ser ligeiramente menor com multithread em cargas leves (sem overhead de polling), mas throughput será limitado.
- 



## Análise e Discussão

- Comparar os gráficos de tempo médio de resposta e throughput.
- Discutir quando cada abordagem é mais indicada:

- **Multithread**: cenários com poucas conexões pesadas.
- **select**: muitas conexões ociosas ou curtas.



## Extensões Futuras (opcional)

- Comparar também com **asyncio** ou **epoll** (Linux).
  - Usar arquivos maiores (100 KB, 1 MB) e conexões lentas (simular clientes com **tc**).
  - Implementar um log de benchmark em JSON para análise automatizada.
-