UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS ARARANGUÁ
Sistemas Operacionais
Prof. Roberto Rodrigues Filho

Servidor Web Concorrente – 2023/2

Neste tutorial iremos aprender, na prática, os conceitos necessários para se construir um servidor web concorrente. Para isso, utilizaremos a linguagem de programação Python para realizar o nosso projeto. Visitaremos, brevemente, o conceito de *socket* TCP e o conceito de *threads*. No final deste tutorial, você deverá juntar esses conceitos e implementar um servidor web concorrente. Quase todas as peças do quebra-cabeça para montar o servidor web será dado neste tutorial.

1. Instalação do Ambiente de Desenvolvimento

Para iniciarmos devemos nos certificar que temos o ambiente de desenvolvimento apropriadamente instalado em nossos computadores. Precisamos que você tenha o editor de texto de sua preferência e o Python3 instalados em sua máquina. Escolhemos a linguagem Python por ter uma sintaxe simples e ser de fácil instalação. Caso não tenha o Python3 instalado em sua máquina, siga o tutorial de instalação de acordo com o seu sistema operacional. Para instalar o Python3:

- no Windows siga este tutorial.
- no Linux siga este tutorial.
- no MacOS siga este tutorial.

Após instalar a linguagem Python, siga o restante do tutorial para aprender, na prática, sobre os conceitos de sockets TCP e *threads*.

Atenção! Para executar os programas que estão escritos neste tutorial, escreva o código que estão nas figuras em um arquivo texto, e o salve com a extensão "*.py*". Por exemplo, "*servidor_tcp.py*". Para executar o código, abra um terminal e digite "*python3 servidor_tcp.py*". Nos exemplos sobre *socket* que iremos ver a seguir, sempre inicie o servidor antes de executar o cliente. Para facilitar o estudo dos programas com *socket*, **sempre** abra dois terminais: um para executar o servidor e um para executar o cliente.

2. *Sockets* TCP em Python

Nesta seção iremos analisar o código de um servidor e cliente TCP. Lembre-se que *socket* é basicamente uma interface, que permite processos enviarem e receberem mensagens de outros processos pela rede. Como discutido em sala de aula, o *socket* TCP precisará de um servidor, ou seja, um processo que ficará esperando as mensagens, e um processo cliente, ou seja, um processo que iniciará a comunicação e enviará a primeira mensagem. No contexto do nosso servidor web, nosso servidor será o servidor web que iremos implementar, e o cliente, pode ser o navegador ou outro programa em Python que iremos escrever para testar nosso servidor.

```python
import socket

# criação do socket TCP
socketTCP = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# realização do bind do processo na porta 2023
socketTCP.bind(("127.0.0.1", 2023))
socketTCP.listen()
print("Servidor TCP está esperando conexões!")

# bloqueio da thread principal - esperando novas conexões
socketCliente, endereco = socketTCP.accept()
print(f"Endereco do cliente {endereco}")

# conexão feita, esperando requisição do cliente
msg = socketCliente.recv(1024)

# Mensagem chegou! Vamos imprimi-la na tela!
print(msg)

# enviando resposta para o cliente
socketCliente.sendall(str.encode("Resposta do servidor!"))
socketCliente.close()
```

Figura 1 – Código do servidor TCP

A Fig. 1 mostra o código de um servidor TCP extremamente simples para podermos começar a entender como *sockets* funcionam. Notem que a primeira coisa que fazemos para trabalhar com qualquer tipo de *socket* é importar a biblioteca *socket* (como mostra a linha 1 da Fig. 1). Em seguida, criaremos um objeto *socket* do tipo TCP na linha 4. Para isso, usamos o parâmetro "*socket.SOCK_STREAM*".

A partir de então, fazemos o *bind* do servidor na porta 2023, como ilustra a linha 7 da Fig.1. A função *bind* é uma função extremamente importante. Ela é responsável por pedir para o Sistema Operacional (SO) associar o processo servidor a porta 2023, caso não tenha nenhum outo processo associado a porta 2023. Lembre-se que a porta é o número que identifica o processo em uma máquina, e assim que chega mensagens para a porta 2023, o SO sabe que precisa enviar a mensagem que chegou da rede para o nosso processo. Caso já exista um processo associado a porta 2023, o SO avisará nosso processo desse conflito que terminará sua execução com uma mensagem de erro.

Após a realização do *bind*, temos que usar o método "*listen()*" (linha 8 da Fig. 1) para preparar o *socket* para aceitar novas conexões. Na sequência, o *thread* principal do nosso servidor bloqueia na linha 12 esperando novas conexões. Note que a partir do momento que o cliente estabelece a conexão com o servidor, o método "*accept()*" retorna um outro objeto *socket* exclusivo para enviar e receber mensagens com o cliente que acaba de se conectar ao servidor. Na sequência, nosso código irá usar esse novo *socket* para ler a mensagem enviada pelo cliente, e na linha 22, enviar a resposta para o cliente.

Depois que a mensagem enviada pelo cliente chega ao servidor, o *thread* principal do servidor segue sua execução da linha 16, imprime a mensagem na tela e usa o *socket* exclusivo para enviar mensagem para o cliente. No final, o servidor encerra a conexão e finaliza sua execução.

Na Fig. 4, mostrada abaixo, temos o código cliente que interage com o servidor TCP. Assim como fizemos no servidor, iniciamos o código importando a biblioteca *socket.* Em seguida, fazemos a criação do *socket* TCP, da mesma forma como fazemos no servidor, passando o parâmetro "*socket.SOCK_STREAM*". Na linha 4, o cliente estabelece uma conexão com o servidor usando o método "*connect()*", passando como parâmetro o endereço IP do servidor e a porta que identifica o processo do servidor na máquina onde o servidor TCP está executando.

```
1   import socket
2
3   socketTCP = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4   socketTCP.connect(("127.0.0.1", 2023))
5   socketTCP.sendall(b"Mensagem para o servidor!")
6   msg = socketTCP.recv(1024)
7   print(f"Received {msg!r}")
8   socketTCP.close()
```

Figura 2 – Código do cliente TCP.

Após o estabelecimento da conexão com o servidor TCP, o cliente então envia sua mensagem usando o método "*sendall()*". Na linha 6, o *thread* principal do processo cliente bloqueia a execução do processo esperando a resposta do servidor, e só segue a diante quando o processo cliente receber a resposta. Em seguida, após o cliente receber a resposta, ele imprime na tela a mensagem recebida, fecha a conexão com o servidor e finaliza sua execução.

3. *Threads* em Python

Nesta seção, iremos descrever o conceito de *threads* e como podemos usá-lo em aplicações em Python. Dominar o conceito de *threads* tanto teoricamente, quanto na prática é importante para as próximas etapas deste tutorial. Se você tem dúvidas do conceito de *threads* e processos, sugiro que leia o capítulo 2, intitulado "Processos e *Threads*" do livro "Sistemas Operacionais Modernos" do autor Andrew S. Tanenbaum, 4ª edição.

Thread é o fluxo de execução do programa. Ao executar um programa em um computador, o sistema operacional cria um processo com todas as informações necessárias para executá-lo, incluindo os dados de entrada e saída, arquivos abertos, etc. Ao executar o processo, um *thread* principal é iniciado, dando início a execução do programa.

Neste tutorial, faremos o uso da criação de *threads* para quando o fluxo de execução principal do servidor ficar bloqueado, esperando novas mensagens de clientes, o servidor poder, em paralelo, continuar atendendo as requisições que já foram aceitas. A partir do conhecimento prático do uso de *threads*, podemos então criar servidores concorrentes.

Ilustramos, na Fig. 3 abaixo, a utilização de *threads*. Nosso programa fará duas atividades de forma paralela (*i.e.*, ao mesmo tempo). Iniciamos nosso programa importando a biblioteca *threading* e *time*. Usaremos *threads* para criarmos e coordenarmos as execuções paralelas, e usaremos *time* para colocarmos um *thread* para "dormir" durante um tempo.

```
1   from threading import Thread
2   from time import sleep
3
4   def minha_funcao(arg):
5       for i in range(arg):
6           print("executando dentro de um outro thread...")
7           sleep(1)
8
9
10  if __name__ == "__main__":
11      thread = Thread(target = minha_funcao, args = (20, ))
12      thread.start()
13      for i in range(20):
14          print("executando no thread principal...")
15          sleep(0.5)
16      print("for principal finalizou")
17      thread.join()
18      print("Thread finalizou...")
```

Figura 3 – Programa que executam funções concorrentes

No código da Fig. 5, fazemos a declaração de uma função *"minha_funcao()"* que recebe como parâmetro o intervalo que será usado no *for* da linha 5. Essa função simplesmente irá executar por "args" vezes (ou seja, o número de vezes passado por parâmetro para a função), e a cada laço de execução do *for*, será impresso a mensagem *"executando dentro de uma outra thread"*. Depois de imprimir a mensagem, o *thread* irá "dormir" por 1 segundo (ou seja, irá fazer uma pausa na linha 7 por 1 segundo), depois "acordar" e continuar a execução de um novo laço do *for*.

O programa irá iniciar sua execução na função principal, declarada na linha 10. O *thread* principal iniciará sua execução a partir da linha 11, que cria um *thread* da função *"minha_funcao"*, passando o parâmetro 20. Até este ponto, apesar de um segundo *thread* ter sido criado, ele ainda **não** está em execução. Somente a partir da linha 12, quando chamamos o método "*start()*" o thread criado para executar a função "*minha_funcao*" entra em execução, enquanto o *thread* principal continua executando, ao mesmo tempo, o laço da linha 13 em diante.

Note que na linha 17 tem a função "*join()*". Quando o *thread* principal executar esse método, ele pára sua execução e espera a função "*minha_funcao*" terminar de executar, e só então, segue sua execução, imprimindo a mensagem da linha 18 na tela e finalizando o programa.

Aproveite para explorar o código exemplo de *threads* e faça algumas modificações para aprofunda seu entendimento do código. Por exemplo, escreva uma segunda função que executará em paralelo com o *thread* principal e o *thread* que executa a função "*minha_funcao*". Tente alterar o tempo que cada *thread* "dorme" para ver como a nova versão do programa se comporta.

Exercícios:

Para cada questão abaixo, escreva códigos em Python que irá responder à questão e escreva, em um arquivo a parte, como resposta para as questões, a descrição de como o código funciona em detalhes. Quanto mais detalhes de como o código funciona, melhor.

1. Modifique o programa do servidor TCP para receber, de forma iterativa, várias mensagens dos clientes. A dica principal para criação desse programa é usar o "*while True:*" para fazer o servidor voltar a receber novas conexões, depois que ele finalizar o tratamento da primeira.
2. Escreva um programa em Python usando somente a biblioteca *sockets* para receber uma requisição HTTP e retorne a mensagem *"<h1> Hello World </h1>*". Para fazer a requisição, você pode usar o seu navegador preferido entrando na barra de endereço o seguinte endereço http://localhost:2023/ considerando que o servidor esteja executando na porta 2023. Para ler a requisição do navegador, você pode usar o código abaixo (ou uma versão similar criada por você mesmo):

```
requisicao = socketCliente.recv(1024).decode()
print(requisicao)
resposta = 'HTTP/1.0 200 OK\n\n<h1>Hello World</h1>'
socketCliente.sendall(resposta.encode())
socketCliente.close()
```

Figura 4 – Parte do servidor que lê requisições HTTP e responde.

Note que o "*print(requisicao)*" vai mostrar na tela a requisição em HTTP realizada pelo seu navegador. Você pode usar as informações dessa requisição para saber mais sobre o que o cliente está requisitando. Isso vai ser essencial para responder a próxima questão.

3. Escreva um servidor web, com base no código que vocês desenvolveram até então, para que cada requisição que chegar ao servidor seja tratada por um *thread* diferente. Faça também o servidor responder com diferentes arquivos dependendo a URL requisitada pelo cliente. Por exemplo, se a URL for '*localhost:2023/index.html*', o seu servidor deve ler o arquivo index.html de alguma pasta e retornar para o navegador, ou retornar uma mensagem de erro, por exemplo com um "404 – NOT FOUND". É necessário também desenvolver um cliente em Python que também usa *threads* para realizar requisições em paralelo para o servidor. Além do código do servidor, vocês devem descrever o funcionamento (linha por linha) do servidor web implementado, discutindo o que acontece quando múltiplas requisições chegam ao mesmo tempo no servidor.