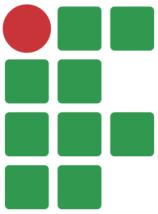

INSTITUTO FEDERAL DA PARAÍBA
CAMPUS CAMPINA GRANDE
TECNOLOGIA EM TELEMÁTICA
DISCIPLINA DE PROGRAMAÇÃO III
PROF. VICTOR ANDRÉ PINHO DE OLIVEIRA

Programação III

Aula 5 - Arquitetando o Servidor - Parte 1

Introdução

Olá,

Na semana anterior, nós concluímos o tema “Preparando dados para transmissão”. Dentro desse tema, nós vimos como podemos enviar tipos de dados comuns a todas as linguagens, tipos de dados específicos do Python e também como enviar dados por meio de padrões universais. Adicionalmente, nós vimos como podemos reduzir a quantidade de bytes transmitidos usando compactação. E, claro, não poderíamos deixar de lembrar, vimos várias abordagens de como resolver a questão do *framing* em transmissões TCP.

Na aula de hoje, nós vamos entrar em um novo tema, a saber: “Arquitetando o Servidor”. Já havia sido mencionado (e percebido) que nossas implementações de Servidores estavam limitadas a atender um e somente um Cliente por vez. Fizemos assim por razões didáticas, enquanto nos concentramos em outros temas. Agora, porém, chegou o momento de vermos como podemos empregar técnicas que nos permitem atender a vários Clientes TCP simultaneamente.

E quanto ao UDP? Embora seja possível aplicar as técnicas que serão apresentadas, não precisamos nos preocupar tanto em relação ao atendimento de vários Clientes UDP em razão de como os Sockets UDP funcionam, isto é, sem conexão. Além do mais, o UDP deve ser usado em trocas de mensagens curtas e objetivas, do tipo pergunta/resposta. Por essa razão, considerando o que foi elencado neste parágrafo, nossa atenção continuará voltada para Sockets TCP.

Bom proveito!

Arquitetando o Servidor

Se quisermos implementar Servidores na categoria de Servidores profissionais, tal como os utilizados na Internet, precisamos implementá-los de modo que eles possam atender a vários Clientes simultaneamente.

Para tanto, existem algumas abordagens que podem ser empregadas, como, por exemplo, Servidores de várias *threads*, Servidores de vários processos, Servidores assíncronos.

Python oferece suporte para todas as abordagens citadas no parágrafo anterior. Tanto em um nível mais baixo, quanto em um nível mais alto. Com isso, quero dizer que Python oferece meios para que o programador faça as coisas do seu jeito, mais personalizado e usando um pouco mais de código, e meios "prontos", através de *frameworks*, onde já temos uma infraestrutura de código que facilita e encapsula a parte "baixa", nos deixando livres para trabalhar apenas na parte que diz respeito à troca de mensagens.

Na aula de hoje, veremos como implementar Servidores de várias *threads* "no braço", isto é, sem fazer uso de *frameworks*. Depois que entendemos como tudo funciona, o uso de um *framework* se torna praticamente trivial. Não veremos Servidores de vários processos porque os conceitos aprendidos de *threads* podem se aplicar aos processos com algumas modificações no código. Deixaremos Servidores assíncronos para próxima aula.

Começaremos com o código de uma aplicação Cliente/Servidor que servirá de base e, em seguida, avançaremos incrementando o código e explicando seus efeitos.

Cliente/Servidor Base

Abaixo, temos os códigos do Servidor e do Cliente que servirá de base para as próximas Listagens. Estamos utilizando a abordagem de *framing* que usa um caractere delimitador para indicar o fim de cada mensagem.

Servidor

```
import socket

def recvall(sock, delim):
    data = b""
    while True:
       more = sock.recv(1)

       if more == delim:
             break
       data += more
    return data

def atende_cliente(sc):
    print("Atendendo Cliente",sc.getpeername())

    msg = recvall(sc,b"\n").decode().upper() + "\n"
    sc.sendall(msg.encode())
```

```
    sc.close()

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(1)

    print("Ouvindo em", sock.getsockname())

    while True:
       sc, sockname = sock.accept()

       atende_cliente(sc)

    return 0
```

Em essência, esse é o mesmo código que utilizamos outras vezes. O Servidor recebe uma mensagem do Cliente e devolve ela em maiúsculas. Temos a função recvall que recebe o caractere (em bytes) delimitador e retorna a mensagem sem ele. Na função main, criamos o socket TCP à maneira como vínhamos fazendo e em seguida entramos no while True. Note que, após o accept(), invocamos a função atende_cliente, passando o socket conectado. Fiz isso para deixar o código pronto para aplicarmos *threads*, concentrando a porção do código que atende o Cliente numa função em sseparado. A função, atende_cliente, por sua vez, imprime o endereço do socket Cliente, recebe a mensagem e devolve em caixa alta.

Cliente

```
import socket

def recvall(sock, delim):
    data = b""
    while True:
       more = sock.recv(1)

       if more == delim:
            break
       data += more
    return data

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    sock.connect(('127.0.0.1',50000))

    print(" Socket: ",sock.getsockname())

    msg = input("Entre com uma mensagem: ") + "\n"

    sock.sendall(msg.encode())
    print(recvall(sock,b"\n").decode())

    sock.close()

    return 0
```

O Cliente, por sua vez, espera uma entrada do usuário, acrescenta o caractere "\n" como terminador ao texto digitado e envia para o Servidor. Em seguida, recebe a mensagem devidamente processada.

Esse mesmo Cliente servirá para todos os Servidores desta aula.

Servidor com *Threads*

Todo programa em Python tem pelo menos uma *thread*, chamada de *thread* principal. Fundamentalmente, uma ***thread*** é um fluxo de execução dentro de um processo. Se temos várias *threads*, isso quer dizer que temos vários fluxos de execução simultâneos em um mesmo processo.

Cada *thread* é independente uma da outra, mas o código do programa, as variáveis, os dados etc., ou seja, tudo que pertence ao processo, é compartilhado entre as *threads*. Por um lado isso é muito bom, mas, quando dois ou mais *threads* querem acessar a mesma variável, por exemplo, isso pode acabar gerando problemas variados. Imagine que uma *thread* está lendo uma variável enquanto outras duas estão alterando o valor! Loucura, loucura, loucura. Felizmente, temos alguns recursos para resolver isso, ponto que será tratado mais à frente nesta aula.

O código a seguir mostra como podemos implementar um Servidor com várias *threads*. Temos a *thread* principal, que ficará responsável por atender as conexões, e as *threads* de "atendimento", responsáveis por atender cada conexão que chega.

Irei reproduzir apenas a parte do código que sofrerá alterações:

```
import socket
import threading

#--trecho cortado--

def main():
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(5)

    print("Ouvindo em", sock.getsockname())

    while True:
       sc, sockname = sock.accept()

       t = threading.Thread(target=atende_cliente,args=(sc,))
       t.start()

    return 0
```

Analise o código acima. Percebe como não tem nada de nebuloso? Pois bem, o código se inicia realizando as importações. A novidade está com a importação do módulo threading. O módulo nos fornece os mecanismos para trabalharmos com *threads* em nossos programas.

Seguindo para a função main, como agora estamos trabalhando com vários atendimentos, alteramos o argumento passado ao método listen para 5, o que quer dizer que estamos aumentando a fila de atendimento. A ideia é permitir que mais Clientes cheguem, pois serão logo atendidos.

Mais adiante, no while True, criamos uma *thread* para cada nova conexão que chega. Thread é uma Classe e recebe seus argumentos (do construtor) todos nomeados. O parâmetro target recebe o nome da função que a nova *thread* vai executar, e o parâmetro args recebe uma tupla com os argumentos da função passada a target. Como a função atende_cliente só recebe um argumento, a tupla passada a args só tem um elemento. Atribuímos o objeto Thread à t e em seguida invocamos o método start().

É na invocação do método start que a nova *thread* ganha vida. Nesse ponto, a *thread* principal segue seu fluxo normal e a nova *thread* começa a executar na função atende_cliente. A nova *thread* permanecerá viva até que a função atende_cliente retorne.

Experimente rodar vários Clientes com esse Servidor com *threads*. Depois compare com o Servidor sem *threads*.

Servidor com *Threads* usando Classes

No tópico anterior, vimos como podemos incluir *threads* em nosso Servidor (e em nossos programas de maneira geral). Utilizamos uma abordagem que passa o nome de uma função a um objeto Thread que será invocada e executada pela nova *thread*.

Existe outra forma de empregarmos *threads* em nossos programas Python: herdando a Classe Thread e reimplementando o método chamado run. Eu, particularmente, prefiro esta forma à anterior. Além de mais elegante, deixa o programa mais flexível e organizado.

```
import socket
import threading

#--trecho cortado--

class Atendimento(threading.Thread):
    def __init__(self,sc):
       super().__init__()
       self.sc = sc

    def run(self):
       print("Atendendo Cliente",self.sc.getpeername())

       msg = recvall(self.sc,b"\n").decode().upper() + "\n"
       self.sc.sendall(msg.encode())

       self.sc.close()


def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(5)

    print("Ouvindo em", sock.getsockname())

    while True:
       sc, sockname = sock.accept()

       t = Atendimento(sc)
```

```
        t.start()

    return 0
```

O que mudou? Agora temos uma Classe chamada de Atendimento que herda a classe Thread do módulo threading. Implementamos o método __init__ apenas para receber o socket conectado e armazenar em um atributo da Classe. O método run contém o mesmo código que a antiga função atende_cliente, do tópico anterior. Note que esse método não deve ser invocado diretamente. Devemos chamar o método start. Este sim, cria a *thread*, que por sua vez executará run.

Dentro da função main, mais especificamente, após o atendimento (accept), instanciamos um objeto Atendimento passando o socket conectado. Em seguida, invocamos o método start. Nesse ponto, a *thread* principal segue o fluxo normal e a nova *thread* vai executar o método run.

Em termos práticos, esse Servidor faz exatamente a mesma coisa que o Servidor do tópico anterior.

Servidor com *Threads* Limitadas

Agora nosso Servidor é capaz de atender a uma gama de Clientes ao mesmo tempo. Tantos quantos chegarem serão atendidos. Até que… bem, até que a memória ou os recursos do computador se esgotem! Resolvemos a limitação do Servidor em atender apenas um Cliente por vez, mas acabamos introduzindo um novo problema :(.

Contornar esse problema é muito fácil: basta definirmos a quantidade máxima de *threads* para o nosso Servidor e, então, impedir que *threads* além desse limite sejam criados.

Esse é um ponto importante a ser considerado porque um Servidor na Internet está sujeito a todo tipo de ataques, inclusive ataques DoS (*Denial of Service* - Negação de Serviço). Se não limitarmos a quantidade de *threads* em nosso Servidor, estaremos vulneráveis à esse tipo de ataque, pois alguém mal intencionado pode, muito facilmente, gerar várias conexões falsas ao nosso Servidor visando, tão somente, esgotar seus recursos e derrubar o Serviço.

O código a seguir modifica o código do tópico anterior e apresenta uma forma de fazermos isso:

```
import socket
import threading

#--trecho cortado--

def main():
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(5)

    print("Ouvindo em", sock.getsockname())

    MAX_THREADS = 4
    l_threads = []

    while True:
       sc, sockname = sock.accept()

       l_threads.append(Atendimento(sc))
       l_threads[-1].start()

       while len(l_threads) >= MAX_THREADS:
            l_threads = [t for t in l_threads if t.is_alive()]

    return 0
```

Antes de entrar no while True, declaramos uma variável MAX_THREADS que vai armazenar o limite de *threads* de nosso Servidor. Em seguida, declaramos uma lista vazia (l_threads), cujo objetivo é armazenar cada objeto Thread criado.

Dentro do loop, depois de aceitar a conexão, instanciamos um Atendimento (o nosso objeto Thread) já anexando à lista l_threads. Na linha seguinte, acessamos o objeto recém criado por meio da lista e iniciamos a *thread* (start).

Lembre-se que a *thread* principal segue seu fluxo normal. Assim, temos um while que ficará executando enquanto o número de *threads* em execução (excetuando-se a *thread* principal) seja igual (ou superior) à quantidade máxima (MAX_THREADS) permitida. Isso vai impedir que o fluxo principal volte para o início de loop e aceite uma nova conexão caso a quantidade máxima de *threads* tenha sido alcançada.

Ainda no while, empregamos um recurso do Python chamado de *List Comprehension*, com a finalidade de criar uma nova lista apenas com as *threads* ativas. A Classe Thread possui um método is_alive que retorna True caso a *thread* ainda esteja em execução e False caso contrário. O while se encerra quando o tamanho da lista (a quantidade de threads) for menor que MAX_THREADS.

Servidor com *Threads* e Travas

Existem situações em que o Servidor precisa manter algum tipo de registro dos dados recebidos dos seus Clientes. Em Python, podemos simplesmente fazer uso de uma Lista

ou um Dicionário, por exemplo, e armazenar tudo que for recebido. Mas, conforme comentado previamente, em um Servidor com várias *threads*, podem ocorrer acessos simultâneos de leitura/escrita sobre a mesma variável. Isso gera um problema: tente imaginar o resultado de uma leitura de uma variável por uma *thread*, enquanto duas outras escrevem conteúdos diferentes. O resultado é imprevisível e, portanto, não podemos permitir que isso aconteça.

O próximo Servidor aproveita o código do tópico anterior e mantém o registro de temperaturas máximas e mínimas de cidades enviadas por Clientes. Para que ele funcione adequadamente, empregamos uma técnica chamada de Trava (Lock) para impedir que duas *threads* acessem a mesma variável ao mesmo tempo. Para testar o Servidor abaixo, os Clientes deverão enviar uma mensagem no formato CSV contendo o nome da cidade, a temperatura máxima e a temperatura mínima.

Segue o código:

```
import socket
import threading

#--trecho cortado--

class Atendimento(threading.Thread):
    def __init__(self,sc,temp,trava):
       super().__init__()
       self.sc = sc
       self.temp = temp
       self.trava = trava

    def run(self):
       print("Atendendo Cliente",self.sc.getpeername())

       msg = recvall(self.sc,b"\n").decode()

       dados = msg.split(",")
       print("Cliente", self.sc.getpeername(), "enviou",dados)

       self.trava.acquire()
       self.temp[dados[0]] = dados[1:]
       self.trava.release()

       self.sc.sendall(b"BYE\n")

       self.sc.close()
```

```
def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(5)

    print("Ouvindo em", sock.getsockname())

    MAX_THREADS = 4
    l_threads = []

    trava = threading.Lock()
    temp = {}

    while True:
       sc, sockname = sock.accept()

       l_threads.append(Atendimento(sc, temp, trava))
       l_threads[-1].start()

       trava.acquire()
       print(temp)
       trava.release()

       while len(l_threads) >= MAX_THREADS:
            l_threads = [t for t in l_threads if t.is_alive()]

    return 0
```

Começando pela main, a novidade se encontra imediatamente acima do loop while True. Instanciamos um objeto Lock do módulo threading e atribuímos à variável trava. Logo em seguida, criamos um dicionário vazio.

Dentro do loop, encontramos a mesma estrutura do código anterior. Duas diferenças: a primeira é que a nossa Classe Atendimento agora recebe mais dois argumentos; e a segunda é que, a cada loop, o servidor imprime o registro das temperaturas. Atente que, antes de imprimir o registro, invocamos o método acquire da nossa trava. Isso é suficiente para impedir que outras *threads*, ao tentarem requerer o mesmo acesso depois, fiquem bloqueadas até a liberação do recurso, momento em que invocamos o método release.

Voltando à Classe Atendimento, alteramos o construtor de modo que ele recebe dois novos argumentos: o registro de temperaturas (dicionário) e a trava. Muita atenção

nesse ponto: para que a trava funcione, é preciso compartilhar o mesmo objeto trava por todas as *threads*.

Dentro do método run, recebemos a mensagem e transformamos os dados recebido em uma lista. Depois imprimimos o que foi recebido na tela e, então, salvamos o novo registro no dicionário temp. Note, mais uma vez, que protegemos o acesso ao dicionário com a trava adquirindo permissão para acessar (acquire) e liberando após o acesso (release).

Por fim, como o Cliente espera uma resposta, enviamos um b"BYE\n" e fechamos o socket.

Servidor com Threads e Fila Síncronas

No tópico anterior, nós usamos Travas (Lock) para evitar que várias *threads* acessem, simultaneamente, a mesma variável. O Servidor em questão recebia os dados de vários Clientes e apenas imprimia os registro sem realizar nenhum processamento sobre eles.

Neste tópico, queremos simular uma situação em que o Servidor continua recebendo dados de vários Clientes (sendo atendidos por *threads*), mas que realiza algum processamento na medida em que esses dados vão chegando ao Servidor. Sendo mais específico, vamos aproveitar a ideia do Servidor anterior, que recebe uma cidade e as temperaturas máxima e mínima em CSV, mas que apresenta, em tempo real, as cidades com temperatura mais quente e mais fria com suas respectivas temperaturas.

É possível resolver essa situação com o uso de Travas, mas Python nos oferece um recurso que torna tudo mais simples e interessante: o módulo queue. Esse módulo contém uma Classe chamada Queue (além de mais outras duas), uma espécie de estrutura FIFO[1], já preparada para ser usada em aplicações *multi-threaded* (de várias *threads*). Isso quer dizer que não precisamos nos preocupar com Travas porque a própria Classe já lida com isso. Se duas *threads* tentarem inserir algum dado na Queue, a segunda thread fica bloqueada até que a primeira termine.

Usar a Classe Queue é extremamente útil em situações do tipo produtor-consumidor, isto é, em situações onde alguma(s) *thread(s)* produz(em) o conteúdo e outra(s) consome(m). E é exatamente essa nossa situação. Vejamos o código:

```
import socket
import threading
import queue

#--trecho cortado--

class Atendimento(threading.Thread):
```

[1] Estrutura FIFO é uma estrutura do tipo fila. FIFO é o acrônimo de *First In First Out* - Primeiro que Entra é o Primeiro que Sai.

```
    def __init__(self,sc,fila_proc):
        super().__init__()
        self.sc = sc
        self.fila_proc = fila_proc

    def run(self):
        print("Atendendo Cliente",self.sc.getpeername())

        msg = recvall(self.sc,b"\n").decode()

        dados = msg.split(",")
        print("Cliente", self.sc.getpeername(), "enviou",dados)

        self.fila_proc.put(dados)

        self.sc.sendall(b"BYE\n")
        self.sc.close()


class ProcessamentoDados(threading.Thread):
    def __init__(self,fila_proc):
        super().__init__()
        self.fila_proc = fila_proc

    def run(self):
        print("Aguardando dados para processar...")

        temp = {}
        while True:
            dados = self.fila_proc.get()

            temp[dados[0]] = list(map(int,dados[1:]))
            mais_quente = max(temp,key=lambda k :
temp.get(k)[0])
            mais_fria = min(temp,key=lambda k :
temp.get(k)[1])

            print("Cidade mais quente:",
mais_quente,"com",temp[mais_quente][0])
            print("Cidade mais fria:",
mais_fria,"com",temp[mais_fria][1])
```

```
            self.fila_proc.task_done()


def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(5)

    print("Ouvindo em", sock.getsockname())

    MAX_THREADS = 4
    l_threads = []

    fila_proc = queue.Queue()
    ProcessamentoDados(fila_proc).start()

    while True:
      sc, sockname = sock.accept()

      l_threads.append(Atendimento(sc, fila_proc))
      l_threads[-1].start()

      while len(l_threads) >= MAX_THREADS:
           l_threads = [t for t in l_threads if t.is_alive()]

    return 0
```

Logo na seção de imports, Incluímos o import para o módulo queue. Mais abaixo, é possível observar que continuamos com nossa Classe Atendimento. Desta vez, porém, o construtor espera receber apenas a fila além do socket conectado. Dentro do método run, continuamos dando um split para transformar a mensagem recebida em uma lista, mas mudamos a forma como armazenamos. Invocamos o método put da queue fila_proc. Note que, se duas *threads* tentarem invocar o método put ao mesmo tempo, uma delas ficará bloqueada até que a outra termine.

Criamos outra Classe chamada de ProcessamentoDados, que também herda de Thread. O construtor recebe apenas a queue fila_proc. No método run, temos um while True que ficará eternamente consumindo os dados de fila_proc. O método get nos permite obter o primeiro dado da queue fila_proc. Mais uma vez, se duas *threads* tentarem acessar o método, somente uma delas conseguirá o acesso enquanto a outra ficará bloqueada.

Continuando, como os dados em fila_proc já estão em forma de lista (cidade, temperatura máxima e temperatura mínima), inserimos os dados no dicionário temp usando a cidade como chave para o dicionário. Além disso, como as temperaturas estão em string usamos a função nativa map para convertê-los em int. Em seguida, usamos a função nativa max para obter a cidade com maior temperatura e a função nativa min para obter a cidade com menor temperatura. Passamos para o parâmetro key uma função lambda que conduz a ordenação considerando o elemento correto da lista dentro do dicionário.

No fim da função run, imprimimos as cidades mais quente e mais fria com suas respectivas temperaturas e invocamos o método task_done da queue fila_proc. O método task_done não tem muito efeito em nosso código, pois não precisaremos sincronizar as *threads* posteriormente. Mas ele serve para indicar que o processamento foi concluído em cima do último dado obtido (get) de fila_proc. Só para que fique mais claro, analisar se a queue está vazia com o método empty não garante que todas as *threads* terminaram seu trabalho. Pode ocorrer de a fila de trabalhos estar vazia, mas ainda haver algum processamento sobre ela, por isso a importância do método task_done. Como nosso Servidor fica rodando para sempre, e também a *thread* ProcessamentoDados não temos que nos preocupar com isso.

Na função, antes de entrar no while True, instanciamos uma queue e iniciamos a thread ProcessamentoDados, passando, obviamente a queue fila_proc. Essa thread terá unicamente a responsabilidade de consumir os dados da queue fila_proc e processá-los. Dentro do loop, atendemos as conexões que chegam, gerando um máximo de quatro *threads* Atendimento. Também passamos a queue fila_proc.

Chegamos ao final de mais uma aula. 🤗

Nesta aula estudamos como implementar um Servidor com *threads* visando atender a múltiplos Clientes simultaneamente. Na aula que vem, continuaremos com o tema “Arquitetando e Servidor” e veremos como podemos criar um Servidor assíncrono.

Bons estudos e até a próxima!