UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Fábio Alves Bocampagni

Socket programming

Implementação das soluções da primeira lista de programação de sockets.

RIO DE JANEIRO
2022

Todos os códigos mencionados podem ser encontrados aqui.

Diferentemente do livro, utilizou-se a rotina gethostname() para definir os hosts. Assim foi possível que apontasse para a mesma máquina ambos os processos, criando uma conexão de hosts que apontam para o mesmo ambiente com processos rodando em portas diferentes, simulando a comunicação entre máquinas distintas.

1. Na seção 2.7.1 do livro de referência, na 7ª edição, são implementados os códigos do cliente (UDPClient.py) e do servidor (UDPServer.py) para uma aplicação em que o servidor recebe um único trecho de texto do cliente e o devolve com todas as letras em maiúsculo. No entanto, o cliente só consegue enviar um único trecho de texto antes de finalizar a conexão. Altere o código do cliente para permitir que o cliente envie vários trechos de textos, ou seja, de tal forma que o cliente envie texto múltiplas vezes. (Obs: ambos os códigos precisarão ser ligeiramente adaptados? ou apenas o cliente?).

No caso UDP, haja vista que o cliente já recebe indeferidas requisições, apenas é necessário que o cliente não se encerre após a primeira comunicação. Com isso, algumas mudanças precisam ser feitas apenas no lado do cliente.

No link para o github, o arquivo UdpClient.py contém a solução mencionada.

```python
from socket import *


serverName = gethostname()
servePort = 12000
clientSocket = socket(AF_INET,SOCK_DGRAM)

print("Para fechar o cliente, digite 0.")
while True:
    message = input("Entre com a mensagem em minusculo: ")
    if message == '0':
        break
    clientSocket.sendto(message.encode(),(serverName,servePort))

    modifiedMessage, serverAdress = clientSocket.recvfrom(2048)
    print("Resposta do servidor: ", modifiedMessage.decode())


clientSocket.close()
```

Vale ressaltar que para encerrar o processo, o valor “0” deverá ser inserido no terminal. Do contrário, a comunicação com o UdpServer.py irá acontecer indefinidamente, sempre tendo 1 requisição e 1 resposta.

Como mencionado, não há necessidade de mudar nada no UdpServer.py.

2. Repita a questão anterior para o caso em que o cliente e servidor usam TCP. Discuta as possibilidades, vantagens e desvantagens, ao implementar no modo persistente e não persistente. Experimente executar vários clientes em paralelo, e veja o que acontece.

Já para o caso TCP, devido ao fato de uma conexão ser estabelecida entre cliente e servidor antes da comunicação de dados, é necessário mais mudanças.

O cliente, para que não se encerre depois da primeira requisição precisará de um mecanismo de loop, mantendo assim a conexão estabelecida viva no lado do cliente.

O arquivo TcpClient.py possui as mudanças mencionadas:

```python
from socket import *

serverName = gethostname()
serverPort = 12000

clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))


while 1:
    sentence = input('Entre com uma palavra em minusculo: ')
    clientSocket.send(sentence.encode())

    if sentence == 'close':
        break

    modifiedMessage = clientSocket.recv(1024)
    print('From server: ', modifiedMessage.decode())



clientSocket.close()
```

Do lado do servidor, como queremos manter a conexão com o cliente, precisamos criar um outro laço de repetição depois de criarmos o socket que atenderá o cliente que chegou.

Vale ressaltar a existência de dois sockets, o de “boas-vindas”, ou seja, o socket que recebe todas as requisições, nesse caso, o serverSocket. E o socket que é criado para atender as requisições de fato, nesse caso, o connectionSocket.

Assim, depois da confirmação da criação do connectionSocket, iremos travar o processo em um loop que fará com que o connectionSocket continue se comunicando com o cliente que o abriu. Assim, ambos conseguem trocar mensagens. A conexão se encerra quando o cliente envia “close” como mensagem.

Vale ressaltar que o processo do TcpServer.py ficará travado e não atenderá outras requisições que chegam enquanto houver a existência de um connectionSocket. Dessa forma, o mesmo não funciona bem para clientes em paralelo. Uma solução para isso é o uso de threads para lidar com as connectionsSockets separadamente.

O TcpServer.py contém as mudanças mencionadas:

```
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))

serverSocket.listen(1)
print('Servidor pronto para ser usado')

while True:
    connectionSocket, addr = serverSocket.accept()
    print("Requisição de:", addr)

    while 1:
        sentence = connectionSocket.recv(1024).decode()
        print("Mensagem:", sentence)
        if sentence == 'close':
            break
        capitalizedSentence = sentence.upper()
        connectionSocket.send(capitalizedSentence.encode())

    connectionSocket.close()
```

Nesse caso independentemente do tipo de conexão, persistente ou não, ainda teríamos o problema do não funcionamento para clientes em paralelo, haja vista que uma conexão precisa ser estabelecida entre cliente e servidor antes dos dados serem enviados, dessa forma, quando o primeiro cliente conecta-se com o servidor, iríamos para dentro do “while 1:”, ainda impossibilitando de haver clientes sendo atendidos em paralelo.

3. Repita a questão anterior, mas agora de tal forma que o servidor possa atender vários clientes em paralelo. Faça a questão tanto para UDP quanto TCP. No caso do UDP, cada thread ou processo será responsável por um trecho de texto, de forma completamente independente, como se os clientes emitindo vários textos, na realidade, fossem vários clientes. No caso de TCP, você pode escolher se irá considerar o modo persistente ou não persistente.

Para o caso UDP, não há a necessidade de mudar nada, nem no cliente nem no servidor, haja vista que a não necessidade de uma conexão entre cliente e servidor faz com que o paralelismo de clientes funcione como esperado.

Já no caso TCP, como mencionado no item 2, é necessário a implementação de threads para lidarem com o bloqueio do processo, dessa forma, o threadedTcpServer.py possui as mudanças necessárias para que as especificações mencionadas funcionem:

```python
from socket import *
import _thread


serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))

serverSocket.listen(5)
print('Servidor pronto para ser usado')

def handle_client(client_socket, addr):
    while True:
        data = client_socket.recv(1024).decode().upper()
        if data == 'close': break
        print('Response to:', addr)
        print('Response data:', data)
        client_socket.send(data.encode())
    client_socket.close()

while True:
    connectionSocket, addr = serverSocket.accept()
    print("Requisição de:", addr)
    _thread.start_new_thread(handle_client ,(connectionSocket, addr))
```

4. Faça um programa cliente cuja única função é enviar os N primeiros números naturais para um servidor pré-especificado. Faça também um programa servidor que imprima esses números no terminal à medida que eles vão chegando. Para esta questão, considere experimentar os dois tipos de protocolos de transporte, TCP e UDP. Você observou alguma diferença? Caso sim, qual a causa dessa diferença? Caso não, alguma diferença era esperada?

Devido a natureza de stream de bytes do TCP, é esperado um comportamento diferente do UDP, que reconhece bem as fronteiras da mensagem.

Devido ao fato do TCP colocar as mensagens em buffers, existe a possibilidade, e quando digo isso quero dizer que é bem provável disso acontecer, das mensagens não chegarem no servidor uma a uma, e sim em pacotes de mensagens, todas ordenadas pois o TCP garante a ordem das mensagens e garante a chegada delas.

Dessa forma, o naturalNumberUdpClient.py, naturalNumberUdpServer.py implementam o problema para o protocolo UDP e o naturalNumberTcpClient.py, naturalNumberTcpServer.py, para o caso TCP.

```python
from socket import *


serverName = gethostname()
servePort = 12000
clientSocket = socket(AF_INET,SOCK_DGRAM)

print("Para fechar o cliente, digite 0.")
naturalNumber = int(input("Entre com um número: "))
if naturalNumber == 0:
    clientSocket.close()

for x in range(naturalNumber):
    clientSocket.sendto(str(x).encode(),(serverName,servePort))


clientSocket.close()
```

naturalNumberUdpClient.py

```python
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET,SOCK_DGRAM)

serverSocket.bind(('',serverPort))

print('O servidor está pronto para uso')

while True:
    number, clientAdress = serverSocket.recvfrom(2048)
    print(number.decode())
```

naturalNumberUdpServer.py

```python
from socket import *

serverName = gethostname()
serverPort = 12000

clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))


naturalNumber = int(input('Entre com um número '))

if naturalNumber == 0:
    clientSocket.close()

for x in range(naturalNumber):
    clientSocket.send(str(x).encode())


clientSocket.close()
```

naturalNumberTcpClient.py

```python
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))

serverSocket.listen(5)
print('Servidor pronto para ser usado')

while True:
    connectionSocket, addr = serverSocket.accept()
    print("Requisição de:", addr)

    while 1:
        naturalNumber = connectionSocket.recv(2048)
        if naturalNumber.decode() == "":
            break
        print(naturalNumber)

    connectionSocket.close()
```

naturalNumberTcpServer.py

5. Nesta tarefa, você terá que restringir o acesso a um dado que está no servidor para somente usuários autenticados. Para isso, faça uma aplicação cliente que solicite o usuário e a senha, envie esses dados para o servidor (não se preocupe com criptografia) e então imprima no terminal a mensagem enviada pelo servidor. No lado do servidor, você deve implementar uma verificação do usuário e da senha enviados pelo cliente (pode ser com uma simples comparação com valores pré-estabelecidos) e então enviar de volta ou uma mensagem de erro (caso as informações de login estejam incorretas) ou enviar uma mensagem de sua preferência (caso as informações de login estejam corretas).

Para enviar duas informações independentes na mesma requisição, juntou-se em uma mesma string o usuário e a senha, do lado do cliente, e foi enviado para o servidor. Seguindo o mesmo parâmetro que criou o objeto de usuário e senha do lado do cliente, foi separado do lado do servidor e comparado com os valores pré determinado, constantes no código, para autorizar ou não o host que criou aquela requisição.

Os arquivos passportClient.py e passportServer.py possuem a solução mencionada.

```python
from socket import *
import json

serverName = gethostname()
serverPort = 12000

clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))


usuario = input('login: ')
senha = input('password: ')

requestData = {
    "user": usuario,
    "password": senha
}

jsonData = json.dumps(requestData)

clientSocket.send(jsonData.encode())

responseMessage = clientSocket.recv(1024)
print('Message from server: ', responseMessage.decode())



clientSocket.close()
```

passportClient.py

```python
from socket import *
import json

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))

serverSocket.listen(1)
print('Servidor pronto para ser usado')

USUARIO = 'SADOC'
SENHA = 'UFRJ'

while True:
    connectionSocket, addr = serverSocket.accept()
    print("Requisição de:", addr)

    while 1:
        requestData = connectionSocket.recv(1024).decode()

        if requestData == '':
            break

        requestObject = json.loads(requestData)

        if requestObject['user'] == USUARIO and requestObject['password'] == SENHA:
            print("Host autorizado: ", addr)
            connectionSocket.send("Autorizado".encode())
        else:
            print("Host não autorizado: ", addr)
            connectionSocket.send("Nao autorizado".encode())
            connectionSocket.close()



    connectionSocket.close()
```

passportServer.py

6. Bora fazer um chatbot pra ajudar a tirar dúvidas sobre a nossa disciplina? A aplicação é a seguinte: o comportamento do lado do servidor é descrito na Figura 1, onde as elipses correspondem a estados, as caixas vermelhas correspondem às mensagens que o servidor envia para um cliente assim que atinge cada estado e as caixas verdes são as condições que precisam ser satisfeitas para haver a troca de estado. A função do lado do cliente, por sua vez, é simplesmente se manter em um loop com três atividades simples: solicita ao usuário uma entrada de texto, envia esse texto pro servidor e imprime a resposta que receberá do servidor. Você deve implementar a interrupção desse loop para que o cliente finalize sua conexão com o servidor quando este finalizar a conexão com o cliente, ou seja, quando o servidor atingir o estado "Finalizar". Utilize sockets TCP para a comunicação nesta aplicação. Note que essa é uma comunicação *stateful* diferente do HTTP que é *stateless*. Comente as vantagens e desvantagens das duas abordagens de comunicação.

Para a implementação da solução da especificação mencionada, foi necessário a criação de estados controlados tanto no lado do servidor quanto no lado do cliente, apesar dessas informações estarem alheias ao conhecimento do cliente.

O cliente envia, junto com a sua mensagem, um estado o qual é conferido também no lado do servidor para que sua resposta seja coerente com o que está sendo solicitado.

Devido ao fato de estarmos utilizando o TCP, é necessário estabelecer uma conexão antes de começarmos a enviar os dados propriamente. Assim, após a conexão, o cliente é convidado a dar olá ao nosso chatbot da turma, após essa interação, o fluxo segue como especificado no diagrama da última folha.

Os arquivos chatbotClient.py e chatbotServer.py implementam a solução mencionada.

```python
from socket import *
import json

serverName = gethostname()
serverPort = 12000

clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))

ESTADO_INICIAL = 0

print("Diga olá ao chatbot da turma: ")
while 1:

    #Se estado final, sai do while.
    if ESTADO_INICIAL == 3:
        break

    entrada = input()
    data = str(ESTADO_INICIAL)+"-"+entrada
    clientSocket.send(data.encode())

    responseMessage = clientSocket.recv(1024)
    print(responseMessage.decode())

    #Se estado de escolha dos serviços, e o valor digitado for diferente das opções, mandar para o servidor processar e sair do while.
    if ESTADO_INICIAL == 2 and entrada not in [1,2,3]:
        break

    ESTADO_INICIAL+=1



clientSocket.close()
```

chatbotClient.py

```python
from socket import *
import json

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))

serverSocket.listen(1)
print('Servidor pronto para ser usado')


while True:
    connectionSocket, addr = serverSocket.accept()
    print("Conexão com:", addr)
    connectionSocket.send("Olá ! Bem vindo ! Qual o seu nome ?".encode())
    while 1:
        message = connectionSocket.recv(1024).decode()
        params = message.split("-")
        if params[0] == 0:
            connectionSocket.send("Olá ! Bem vindo ! Qual o seu nome ?\n".encode())

        elif params[0] == '1':
            connectionSocket.send((f"Certo, {params[1]}!\nComo posso te ajudar ?\nDigite o número que corresponde à opção desejada: \n1 - Agendar um horário de

        elif params[0] == '2' and params[1] not in ['1','2','3']:
            connectionSocket.send("Obrigado por utilizar nossos serviços!\nAté logo!".encode())
            break

        elif params[0] == '2':
            if params[1] == '1':
                connectionSocket.send("Para agendar uma monitoria, basta enviar um e-mail para cainafigueiredo@poli.ufrj.br\n".encode())
                connectionSocket.send("Obrigado por utilizar nossos serviços!\nAté logo!\n".encode())

            elif params[1] == '2':
                connectionSocket.send("Fique atento para as datas das próximas atividades.\nConfira o que vem por ai !\n\nP1: 28 de maio de 2022\nLista 3: 29 de
                connectionSocket.send("Obrigado por utilizar nossos serviços!\nAté logo!\n".encode())

            elif params[1] == '3':
                connectionSocket.send("Quer falar com o professor ?\nO e-email dele é sadoc@dcc.ufrj.br\n".encode())
                connectionSocket.send("Obrigado por utilizar nossos serviços!\nAté logo!\n".encode())

            break



    connectionSocket.close()
```

chatbotServer.py

P29

Originalmente, o cliente não especificava nenhuma porta quando criava o socket, deixando o sistema operacional determinar a porta que o processo iria ser executado. Agora, especificando a porta do processo para 5432, que é a mesma porta padrão do banco de dados postgres, estamos fixando algo que em teoria deveria ser responsabilidade do sistema operacional, pois o mesmo gerencia melhor as portas, dado o seu caráter dinâmico, como por exemplo, se já houver um processo de bando de dados rodando nessa porta, ou um container docker rodando nessa mesma porta e tentarmos executar o cliente, o mesmo não será executado pois há um processo utilizando a porta desejada.

Utilizando o bind, estamos falando que aquela porta será reservada única e exclusivamente para aquele processo até que o mesmo termine e que os outros processos podem o encontrar naquela porta.

Devido ao fato do servidor receber o host e a porta no retorno da rotina serverSocket.recvfrom(2048), ele funciona dinamicamente para todas as portas, não sendo necessário atualização do código.

P30

Sim, é possível configurar navegadores para abrirem múltiplas conexões simultâneas com um site. A vantagem é que o download dos recursos será mais rápido, a desvantagem é que você irá monopolizar o bandwidth, logo, diminuindo o download dos outros usuários que compartilham o mesmo link físico.

P31

Para aplicações onde a ordem dos bytes seja de extrema importância, como sistemas de login, a forma como o stream de bytes do TCP funciona e a forma como os dados são garantidamente entregados para a outra ponta do sistema, constrói a premissa de sistemas como esse, pois necessita-se de garantias quanto a ordem e conformidade das chegadas.

Por outro lado, apesar desse fluxo de dados ser algo que é oferecido para os processos que implementam o protocolo TCP, o mesmo não envia os bytes 1 por 1 e sim os armazena em buffers.

Delimitar esses buffers tal como implementar uma solução na ponta do servidor para que a leitura seja bem feita é necessário, dessa forma, aplicações onde necessita-se de muitas trocas de mensagens e com dados divergentes necessitam de implementar na mão as fronteiras de suas mensagens, o que pode ser bem custoso e refletir na escolha do UDP como protocolo de comunicação, haja vista que suas fronteiras são bem definidas.